UNIVERSITÄT TRIER

ZIMK - Zentrum für
Informations-, Medien- und
Kommunikationstechnologie

Bernhard Baltes-Götz

Einführung in das Programmieren mit C# 9.0

Die Weiterentwicklung dieses Manuskripts behandelt (am 20.10.2023) die

C# - Version 11

und ist hier verfügbar:

https://bebagoe.de/csharp/

2021 (Rev. 230831)

Herausgeber: Zentrum für Informations-, Medien- und Kommunikationstechnologie (ZIMK) an der Universität Trier
Universitätsring 15
D-54286 Trier
WWW: zimk.uni-trier.de
E-Mail: zimk@uni-trier.de
Autor: Bernhard Baltes-Götz
WWW: https://www.uni-trier.de/?id=54994
E-Mail: baltes@uni-trier.de
Lektorat: Paul Frischknecht

Vorwort

Dieses Manuskript entstand als Begleitlektüre zu einem C# - Einführungskurs und einem Vertiefungskurs, den das Zentrum für Informations-, Medien- und Kommunikationstechnologie (ZIMK) an der Universität Trier im Wintersemester 2020/2021 bzw. im Sommersemester 2021 angeboten hat, ist aber auch für das Selbststudium geeignet.

Lerninhalte und -ziele

C# ist eine von der Firma Microsoft für die .NET - Plattform entwickelte und von der internationalen IT-Normungsorganisation ECMA[1] standardisierte Programmiersprache, die auf den Vorbildern C++ (siehe z. B. Baltes-Götz 2003) und Java (siehe z. B. Baltes-Götz & Götz 2020) aufbaut, aber auch etliche Weiterentwicklungen bietet.

Die .NET - Plattform hat sich als Standard für die Softwareentwicklung unter Windows etabliert, und ist durch das Open Source - Projekt *Mono* erfolgreich auf andere Betriebssysteme (Linux, macOS) portiert worden.[2] Neben Mono entwickelt auch das von Microsoft geförderte Open Source - Projekt *.NET Core* eine Betriebssystem-unabhängige, unter Linux, macOS und Windows verfügbare .NET - Implementation. Bis zum Herbst 2020 koexistierten die folgenden .NET-Implementationen:

- das nur für Windows verfügbare .NET Framework (aktuelle Version: 4.8)
- das für Linux, macOS und Windows verfügbare .NET Core (aktuelle Version: 3.1)
- das für Linux, macOS und Windows verfügbare Mono (aktuelle Version: 6.12)
- Xamarin
- UWP

Im November 2020 ist das im Wesentlichen plattformunabhängige .NET 5.0 zusammen mit C# 9.0 erschienen, um alle bisherigen .NET - Implementationen allmählich abzulösen. Man kann .NET 5.0 als Weiterentwicklung von .NET Core 3.1 auffassen, wobei die Versionsnummer 4 wegen der Verwechslungsgefahr mit dem .NET Framework 4.x übersprungen wurde.[3] Unter Windows unterstützt .NET 5.0 auch Anwendungen mit den etablierten GUI-Techniken (*Graphical User Interface*) UWP, WPF und WinForms.[4]

Wir werden im Kurs bzw. Manuskript mit dem Betriebssystem Windows 10 (64 Bit) und mit den folgenden .NET - Implementationen arbeiten:

- .NET 5.0 mit C# 9.0
- .NET Framework 4.8 mit C# 7.3
  Vereinzelt ist ein Rückgriff auf diese Implementation erforderlich, weil veraltete .NET Framework - Bestandteile nicht in .NET Core (also auch nicht in .NET 5.0) übernommen wurden.

Von der Firma Xamarin, die einst die Mono-Entwicklung initiiert hat und mittlerweile von der Firma Microsoft übernommen worden ist, stammt ein attraktiver Ansatz zur Entwicklung mobiler Apps für Android und iOS in C#.[5] Auch Xamarin soll im Wesentlichen im neuen plattformunab-

---

[1] Ursprünglich stand ECMA für ***E**uropean **C**omputer **M**anufacturers **A**ssociation*, doch wurde diese Bedeutung abgelegt, um den nunmehr globalen Anspruch der Organisation zu dokumentieren (siehe z. B. http://www.ecma-international.org/).

[2] Webseite des Projekts: http://www.mono-project.com/
Zu Details der Entstehungsgeschichte von Mono siehe https://en.wikipedia.org/wiki/Mono_(software)

[3] https://docs.microsoft.com/en-us/dotnet/core/dotnet-five

[4] https://entwickler.de/online/windowsdeveloper/dotnet-framework-dotnet-core-mono-579891813.html

[5] https://www.xamarin.com/

hängigen .NET aufgehen (aber erst in der für November 2021 geplanten Version 6) und dabei offenbar zu einem *.NET Multi-platform App UI* (.NET MAUI) beitragen, das Android, iOS, macOS und Windows unterstützt.[1] Damit wäre in .NET eine relativ plattformunabhängige grafische Bedienoberfläche (leider ohne Linux-Support) realisierbar.

Weil C# auch zur Entwicklung von Server-Lösungen gut geeignet ist, handelt es sich insgesamt um eine außerordentlich vielseitige Programmiersprache.

Ein Vorteil der .NET - Plattform(en) ist die freie Wahl zwischen verschiedenen Programmiersprachen, doch kann C# trotz der großen Konkurrenz als die bevorzugte .NET - Programmiersprache gelten. Schließlich wurde die Plattform selbst überwiegend in C# entwickelt.

Wenngleich die Netzorientierung im Namen der Plattform betont wird, ist C# (wie jede andere .NET - Programmiersprache) universell einsetzbar. Der Kurs behandelt wesentliche Konzepte und Methoden der objektorientierten Softwareentwicklung (z. B. elementare Sprachelemente, Klassen, Vererbung, Polymorphie, Schnittstellen) und berücksichtigt viele Standardthemen der Programmierpraxis (z. B. Ausnahmebehandlung, Dateizugriff, grafische Bedienoberflächen, Multithreading).

Voraussetzungen bei den Kursteilnehmern

- **Programmierkenntnisse werden *nicht* vorausgesetzt.**
  Teilnehmer *mit* Programmiererfahrung werden sich in den ersten Kapiteln eventuell etwas langweilen.
- **Motivation**
  Es ist mit einem erheblichen Zeitaufwand bei der Lektüre und bei der aktiven Auseinandersetzung mit dem Stoff (z. B. durch das Lösen von Übungsaufgaben) zu rechnen. Als Gegenleistung kann man hochrelevante Techniken erlernen und viel Erfolg (also Spaß) erleben.

Software zum Üben

Für die unverzichtbaren Übungen sollte ein Rechner mit einer aktuellen C# - Entwicklungsumgebung zur Verfügung stehen. Im Kurs wird die kostenlose Community-Variante von Microsofts Visual Studio 2019 bevorzugt. Details zum Bezug und zur Installation der Software finden sich im Kapitel 3.

Dateien zum Manuskript

Die aktuelle Version dieses Manuskripts ist zusammen mit den behandelten Beispielen und Lösungsvorschlägen zu vielen Übungsaufgaben auf dem Webserver der Universität Trier zu finden:

https://www.uni-trier.de/index.php?id=22777

[1] https://www.heise.de/developer/meldung/Build-2020-Aus-Xamarin-Forms-wird-MAUI-4724947.html
https://devblogs.microsoft.com/dotnet/introducing-net-multi-platform-app-ui/

Hinweise auf Fehler und Mängel im Text werden unter der Mail-Adresse

baltes@uni-trier.de

dankbar entgegengenommen.[1]

Trier, im September 2021 Bernhard Baltes-Götz

[1] Für aufmerksame Hinweise auf mittlerweile behobene Fehler möchte ich mich bei Soeren Borchers, Colin Christ, Paul Frischknecht, Marian Peters und Jens Weber herzlich bedanken. Paul Frischknecht hat Hunderte von sachlichen und sprachlichen Fehlern gemeldet und so in herausragender Weise die Qualität des Textes gesteigert. Seine Beiträge haben mindestens das Niveau eines professionellen Lektorats.

Inhaltsverzeichnis

1 Einstieg in die objektorientierte Software-Entwicklung mit C#

In diesem Kapitel soll eine Vorstellung davon vermittelt werden, was ein Computerprogramm (in C#) ist. Dabei kommen einige Grundbegriffe der Informatik zur Sprache, wobei wir uns aber nicht unnötig lange von der Praxis fernhalten wollen.

1.1 Was ist ein Computerprogramm?

Ein Computerprogramm besteht im Wesentlichen (von Medien und anderen Ressourcen einmal abgesehen) aus einer Menge von wohlgeformten und wohlgeordneten *Definitionen* und *Anweisungen* zur Bewältigung bestimmter Aufgaben. Ein Programm muss ...

- den betroffenen Anwendungsbereich **modellieren**
  Beispiel: In einem Programm zur Verwaltung einer Spedition sind z. B. Kunden, Aufträge, Mitarbeiter, Fahrzeuge, Einsatzfahrten, (Ent-)ladestationen und kommunikative Prozesse (Nachrichten zwischen beteiligten Akteuren) zu repräsentieren.
- **Algorithmen** realisieren, die in endlich vielen Schritten und unter Verwendung von endlich vielen Betriebsmitteln (z. B. Speicher, CPU-Leistung) bestimmte Ausgangszustände in akzeptable Zielzustände überführen.
  Beispiel: Im Speditionsprogramm muss u. a. für jede Tour zu den meist mehreren (Ent-)ladestationen eine optimale Route ermittelt werden (hinsichtlich Kraftstoffverbrauch, Fahrtzeit, Mautkosten etc.).

Wir wollen präzisere und komplettere Definitionen zum komplexen Begriff eines Computerprogramms den Informatik-Lehrbüchern überlassen (siehe z. B. Goll & Heinisch 2016) und stattdessen ein Beispiel im Detail betrachten, um einen Einstieg in die Materie zu finden.

Bei der Suche nach einem geeigneten C# - Einstiegsbeispiel tritt ein Dilemma auf:

- Einfache Beispiele sind angenehm, aber für das Programmieren mit C# nicht besonders repräsentativ. Z. B. ist von Objektorientierung außer einem gewissen Formalismus nichts vorhanden.
- Repräsentative C# - Programme eignen sich in der Regel wegen ihrer Länge und Komplexität (aus der Sicht eines Anfängers) nicht für den Einstieg. Z. B. können wir das eben zur Illustration einer realen Aufgabenstellung verwendete, aber potentiell sehr aufwändige, Speditionsverwaltungsprogramm jetzt nicht vorstellen.

Wir betrachten stattdessen ein Beispielprogramm, das trotz angestrebter Einfachheit nicht auf objektorientiertes Programmieren (OOP) verzichtet. Seine Aufgabe besteht darin, elementare Operationen mit Brüchen auszuführen (Kürzen, Addieren), womit es etwa einem Schüler beim Anfertigen der Hausaufgaben (zur Kontrolle der eigenen Lösungen) nützlich sein kann. Das Beispiel wird in sukzessive ausgebauter Form im Kurs noch oft verwendet.

1.2 Objektorientierte Analyse und Modellierung

Einer objektorientierten Programmierung geht die **objektorientierte Analyse** der Aufgabenstellung voraus mit dem Ziel einer Modellierung durch kooperierende **Klassen**. Man identifiziert per **Abstraktion** die beteiligten Kategorien von Individuen bzw. Objekten und definiert für sie jeweils eine **Klasse**. Eine solche Klasse ist gekennzeichnet durch:

- **Merkmale** (**Instanz- bzw. Klassenvariablen, Felder**)
  Viele Merkmale gehören zu den *Objekten* bzw. *Instanzen* der Klasse (z. B. Zähler und Nenner eines Bruchs), manche gehören zur Klasse selbst (z. B. Anzahl der bei einem Programmeinsatz bisher erzeugten Brüche). Im letztlich entstehenden Programm landet jede Merkmalsausprägung in einer sogenannten **Variablen**. Dies ist ein benannter Speicherplatz, der Werte eines bestimmten Typs (z. B. Zahlen, Zeichenfolgen) aufnehmen kann. Variablen zur Repräsentation der Merkmale von Objekten oder Klassen werden oft als **Felder** bezeichnet.
- **Handlungskompetenzen** (**Methoden**)
  Analog zu den Merkmalen sind auch die Handlungskompetenzen entweder individuellen Objekten bzw. Instanzen zugeordnet (z. B. Kürzen bei Brüchen) oder der Klasse selbst (z. B. Informieren über die Anzahl der erzeugten Brüche). Im letztlich entstehenden Programm sind die Handlungskompetenzen durch sogenannte **Methoden** repräsentiert. Diese ausführbaren Programmbestandteile realisieren die oben angesprochenen Algorithmen. Die Kommunikation zwischen Klassen und Objekten besteht darin, ein anderes Objekt oder eine andere Klasse aufzufordern, eine bestimmte Methode auszuführen.

Eine Klasse …

- beinhaltet meist einen **Bauplan** für konkrete Objekte, die im Programmablauf nach Bedarf erzeugt und mit der Ausführung bestimmter Methoden beauftragt werden,
- kann andererseits aber auch **Akteur** sein (Methoden ausführen und aufrufen).

Weil der Begriff *Klasse* gegenüber dem Begriff *Objekt* dominiert, hätte man eigentlich die Bezeichnung *klassenorientierte Programmierung* wählen sollen. Allerdings gibt es keinen ernsthaften Grund, die eingeführte Bezeichnung *objektorientierte Programmierung* zu ändern.

Dass jedes Objekt gleich in eine Klasse („Schublade“) gesteckt wird, mögen die Anhänger einer ausgeprägt individualistischen Weltanschauung bedauern. Auf einem geeigneten Abstraktionsniveau betrachtet lassen sich jedoch die meisten Objekte der realen Welt ohne großen Informationsverlust in Klassen einteilen. Bei einer definitiv nur *einfach* zu besetzenden Rolle kann eine Klasse zum Einsatz kommen, die ausnahmsweise *nicht* zum Instanziieren (Erzeugen von Objekten) gedacht ist, sondern als Akteur.

In unserem Bruchrechnungsbeispiel ergibt sich bei der objektorientierten Analyse, dass vorläufig nur eine Klasse zum Modellieren von Brüchen benötigt wird. Beim möglichen Ausbau des Programms zu einem Bruchrechnungstrainer kommen jedoch weitere Klassen hinzu (z. B. `Aufgabe`, `Schüler`).

Dass Zähler und Nenner die zentralen **Merkmale** eines Bruchs sind, bedarf keiner Begründung. Sie werden in der Klassendefinition durch Felder zum Speichern von ganzen Zahlen (C# - Datentyp **int**) repräsentiert, die folgende Namen erhalten sollen:

- `zaehler`
- `nenner`

Eine wichtige, auf den ersten Blick leicht zu übersehende Entscheidung der Modellierungsphase besteht darin, beim Zähler und beim Nenner eines Bruchs auch negative ganze Zahlen zu erlauben.[1] Alternativ könnte man ...

[1] Auf die Möglichkeit, alternative `Bruch`-Definitionen in Erwägung zu ziehen, hat Paul Frischknecht hingewiesen.

- beim Nenner negative Werte verbieten, um folgende Beispiele auszuschließen:

$$\frac{2}{-3}, \frac{-2}{-3}$$

- beim Zähler und beim Nenner negative Werte verbieten, weil ein Bruch als Anteil aufgefasst und daher stets größer oder gleich null sein sollte.

Indem beim Zähler und beim Nenner auch negative ganze Zahlen zugelassen werden, findet sich unter den Objekten der resultierenden Klasse z. B. eine einfache Lösung für die folgende Gleichung, was irgendwann vorteilhaft sein könnte:

$$-3x = 2 \Leftrightarrow x = \frac{2}{-3}$$

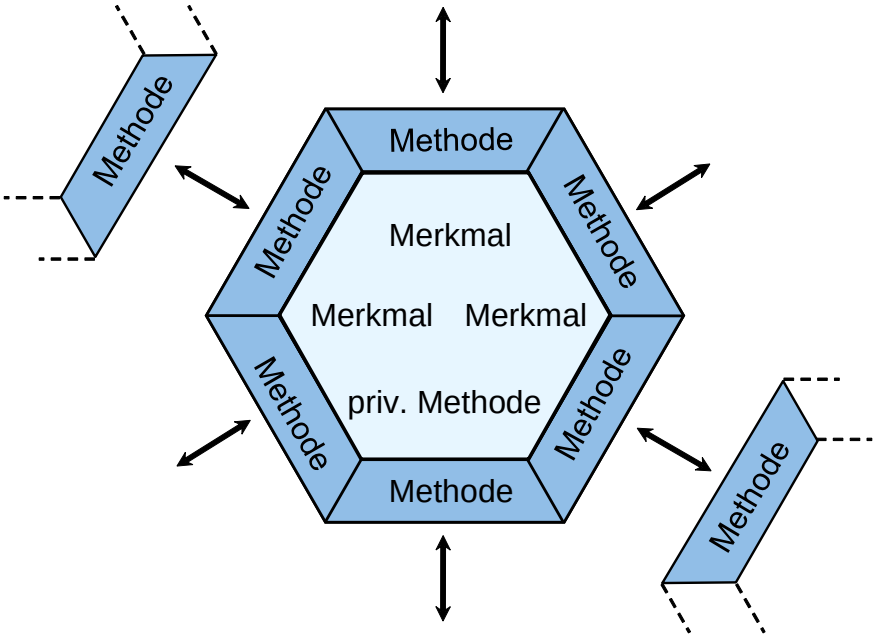
Auf das oben angedeutete *klassenbezogene* Merkmal mit der Anzahl bereits erzeugter Brüche wird vorläufig verzichtet.

Im objektorientierten Paradigma ist jede Klasse für die Manipulation ihrer Merkmalsausprägungen selbst verantwortlich. Diese sollen **eingekapselt** und vor dem direkten Zugriff durch fremde Klassen geschützt sein. So kann sichergestellt werden, dass nur sinnvolle Änderungen der Merkmalsausprägungen möglich sind. Außerdem wird aus später zu erläuternden Gründen die Produktivität der Software-Entwicklung durch die Datenkapselung gefördert.

Demgegenüber sind die **Handlungskompetenzen** (Methoden) einer Klasse in der Regel von anderen Klassen ansprechbar, wobei es aber auch *private* Methoden für den ausschließlich internen Gebrauch gibt. Die *öffentlichen* Methoden einer Klasse bilden ihre **Schnittstelle** zur Kommunikation mit anderen Klassen. Man spricht auch vom **API** (*Application Programming Interface*) einer Klasse.

Die folgende, an Goll & Heinisch (2016) angelehnte Abbildung zeigt für eine Klasse ...

- im gekapselten Bereich ihre Merkmale sowie eine private Methode
- die Kommunikationsschnittstelle mit den öffentlichen Methoden



Die **Objekte** (Exemplare, Instanzen) einer Klasse, d.h. die nach diesem Bauplan erzeugten Individuen, sollen in der Lage sein, auf eine Reihe von **Nachrichten** mit einem bestimmten Verhalten zu reagieren. In unserem Beispiel sollte die Klasse `Bruch` z. B. eine Methode zum Kürzen besitzen. Dann kann einem konkreten `Bruch`-Objekt durch Aufrufen dieser Methode die Nachricht zugestellt werden, dass es Zähler und Nenner kürzen soll.

Sich unter einem Bruch ein Objekt vorzustellen, das Nachrichten empfängt und mit einem passenden Verhalten beantwortet, ist etwas gewöhnungsbedürftig. In der realen Welt sind Brüche, die sich selbst auf ein Signal hin kürzen, nicht unbedingt alltäglich, wenngleich möglich (z. B. als didaktisches Spielzeug). Die objektorientierte Modellierung eines Anwendungsbereichs ist nicht unbedingt eine direkte Abbildung, sondern eher eine *Rekonstruktion*. Einerseits soll der Anwendungsbereich im Modell gut repräsentiert sein, andererseits soll eine möglichst stabile, wartungsfreundliche, gut erweiterbare und wiederverwendbare Software entstehen.

Um (Objekten aus) fremden Klassen trotz Datenkapselung die Veränderung einer Merkmalsausprägung zu erlauben, müssen entsprechende Methoden (mit geeigneten Kontrollmechanismen) angeboten werden. Unsere Klasse `Bruch` sollte also über Methoden zum Verändern von Zähler und Nenner verfügen. Bei einem geschützten Merkmal ist auch der direkte *Lese*zugriff ausgeschlossen, sodass im `Bruch`-Beispiel auch noch Methoden zum Ermitteln von Zähler und Nenner erforderlich sind. Eine konsequente Umsetzung der Datenkapselung erzwingt also eventuell eine ganze Serie von Methoden zum Lesen und Ändern von Merkmalsausprägungen.

Mit diesem Aufwand werden aber gravierende Vorteile realisiert:

- **Stabilität**
  Die Merkmalsausprägungen sind vor unsinnigen und gefährlichen Zugriffen geschützt, wenn Veränderungen nur über die vom Klassendesigner entworfenen Methoden möglich sind. Treten trotzdem Fehler auf, sind diese leichter zu identifizieren, weil nur wenige Methoden verantwortlich sein können.
- **Produktivität**
  Durch Datenkapselung wird die **Modularisierung** unterstützt, sodass große Softwaresysteme beherrschbar werden und zahlreiche Programmierer möglichst reibungslos zusammenarbeiten können. Der Klassendesigner trägt die Verantwortung dafür, dass die von ihm entworfenen Methoden korrekt arbeiten. Andere Programmierer müssen beim Verwenden einer Klasse lediglich die Methoden der Schnittstelle kennen. Das Innenleben einer Klasse kann vom Designer nach Bedarf geändert werden, ohne dass andere Programmbestandteile angepasst werden müssen. Bei einer sorgfältig entworfenen Klasse stehen die Chancen gut, dass sie in mehreren Software-Projekten genutzt werden kann (**Wiederverwendbarkeit**). Besonders günstig ist die Recycling-Quote bei den Klassen der .NET - Standardbibliothek (***B**ase **C**lass **L**ibrary*, BCL) (siehe Abschnitt 2.6), von denen alle C# - Programmierer regen Gebrauch machen. Auch die Klasse `Bruch` aus dem Beispielprojekt besitzt einiges Potential zur Wiederverwendung.

Im Vergleich zu anderen objektorientierten Programmiersprachen wie z. B. Java und C++ bietet C# mit den sogenannten **Eigenschaften** (engl.: **properties)** eine Möglichkeit, den Aufwand mit den Methoden zum Lesen oder Verändern von Merkmalsausprägungen für den Designer und vor allem für den Nutzer einer Klasse zu reduzieren. In der Klasse `Bruch` werden wir z. B. zum Feld `nenner` die *Eigenschaft* `Nenner` (großer Anfangsbuchstabe!) definieren, welche dem Nutzer der Klasse `Bruch` *Methoden* zum Lesen und Setzen des Nenners bietet, wobei dieselbe Syntax wie beim direkten Zugriff auf das Feld verwendet werden kann. Um dieses Argument zu illustrieren, greifen wir der Beschäftigung mit elementaren C# - Sprachelementen vor. In der folgenden Anweisung wird der `nenner` eines `Bruch`-Objekts mit dem Namen `b1` auf den Wert 4 gesetzt:

```
b1.Nenner = 4;
```

Während der Entwickler der Klasse `Bruch` Zugriffs*methoden* bereitzustellen hat (siehe unten), sehen die Nutzer (das sind die Entwickler *anderer* Klassen) eine öffentliche *Eigenschaft*. Langfristig werden Sie diese Ergänzung des objektorientierten Sprachumfangs zu schätzen lernen. Momentan ist sie eher eine Belastung, da Sie vielleicht erstmals mit der Grundarchitektur einer Klasse konfrontiert werden, und die fundamentale Unterscheidung zwischen Merkmalen und Methoden einer Klas-

se durch die C# - Eigenschaften unscharf zu werden scheint. Letztlich erspart eine C# - Eigenschaft wie `Nenner` dem Nutzer lediglich die Verwendung von Zugriffs*methoden*, z. B.

```
b1.SetzeNenner(4);
```

Damit kann man die C# - Eigenschaften in die Kategorie *syntactic sugar* (Mössenböck 2019, S. 3) einordnen, was aber keine Abwertung sein soll. Wegen ihrer intensiven Nutzung in C# - Programmen ist ein Auftritt der Eigenschaften im ersten Kursbeispiel trotz der angesprochenen didaktischen Probleme gerechtfertigt.

In realen (komplexeren) Programmen wird keinesfalls *jedes* gekapselte Feld über eine Eigenschaft zum Lesen und geschützten Schreiben für die Außenwelt zugänglich gemacht.

Insgesamt sollen die Objekte unserer Klasse `Bruch` die folgenden Eigenschaften besitzen bzw. Methoden beherrschen:

- `Nenner` (Eigenschaft zum Feld `nenner`)
  Das Objekt wird beauftragt, seinen `nenner`-Zustand mitzuteilen bzw. zu verändern. Ein direkter Zugriff auf das Merkmal soll fremden Klassen nicht erlaubt sein (Datenkapselung). Bei dieser Vorgehensweise kann ein `Bruch`-Objekt z. B. verhindern, dass sein `nenner` auf null gesetzt wird. Wie und wo die Kontrolle stattfindet, ist bald zu sehen.
- `Zaehler` (Eigenschaft zum Feld `zaehler`)
  Das Objekt wird beauftragt, seinen `zaehler`-Zustand mitzuteilen bzw. zu verändern. Die Eigenschaft `Zaehler` bringt im Gegensatz zur Eigenschaft `Nenner` keinen großen Gewinn an Sicherheit im Vergleich zu einem für andere Klassen direkt zugänglichen Feld. Sie ist aber der Einheitlichkeit und damit der Einfachheit halber angebracht und hält die Möglichkeit offen, das Merkmal `zaehler` einmal anders zu realisieren.
- `Kuerze()`
  Durch diese Methode wird das angesprochene Objekt beauftragt, `zaehler` und `nenner` zu kürzen. Welcher Algorithmus dazu benutzt wird, bleibt dem Klassendesigner überlassen.
- `Addiere(Bruch b)`
  Durch diese Methode wird das angesprochene Objekt beauftragt, den als Parameter (siehe unten) übergebenen Bruch zum eigenen Wert zu addieren.
- `Frage()`
  Durch diese Methode wird das angesprochene Objekt beauftragt, `zaehler` und `nenner` beim Anwender via Konsole (Eingabeaufforderung) zu erfragen.
- `Zeige()`
  Durch diese Methode wird das angesprochene Objekt beauftragt, `zaehler` und `nenner` auf der Konsole anzuzeigen.

Man verwendet für die in einer Klasse definierten Bestandteile oft die Bezeichnung **Member**, gelegentlich auch die deutsche Übersetzung **Mitglieder**. Unsere Klasse `Bruch` enthält folgende Mitglieder:

- Felder
  `zaehler`, `nenner`
- Eigenschaften
  `Zaehler`, `Nenner`
  Bei einer Eigenschaft handelt es sich letztlich um ein Paar von Methoden.
- Methoden
  `Kuerze()`, `Addiere()`, `Frage()` und `Zeige()`

Durch die in C# signifikante (!) Groß-/Kleinschreibung der Namen kann man leichter unterscheiden zwischen:

- privaten Merkmalen (per Konvention mit kleinem Anfangsbuchstaben)
- Eigenschaften und Methoden (per Konvention mit großem Anfangsbuchstaben)

Später folgen detaillierte Empfehlungen zur Verwendung von Bezeichnern in C#.

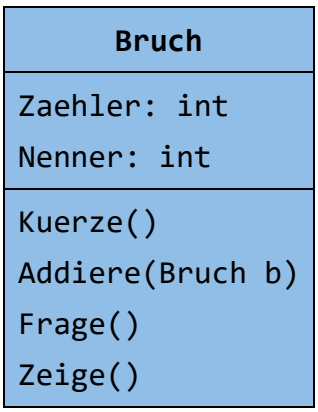
Von kommunizierenden Objekten und Klassen mit Handlungskompetenzen zu sprechen, mag als übertriebener Anthropomorphismus (als Vermenschlichung) erscheinen. Bei der Ausführung von Methoden sind Objekte und Klassen selbstverständlich streng determiniert, während Menschen bei Kommunikation und Handlungsplanung ihren freien Willen einbringen, Spontanität, Kreativität und auch Emotionen besitzen. Fußball spielende Roboter (als besonders anschauliche Objekte aufgefasst) zeigen allerdings mittlerweile schon recht weitsichtige und auch überraschende Spielzüge. Was sie noch zu lernen haben, sind vielleicht Strafraumschwalben, absichtliches Handspiel etc. Nach diesen Randbemerkungen kehren wir zum Programmierkurs zurück, um möglichst bald freundliche und kluge Objekte erstellen zu können.

Um die durch objektorientierte Analyse gewonnene Modellierung eines Anwendungsbereichs standardisiert und übersichtlich zu beschreiben, wurde die **Unified Modeling Language** (UML) entwickelt, die bevorzugt mit Diagrammen arbeitet.[1] Hier wird eine Klasse durch ein Rechteck mit drei Bereichen dargestellt:

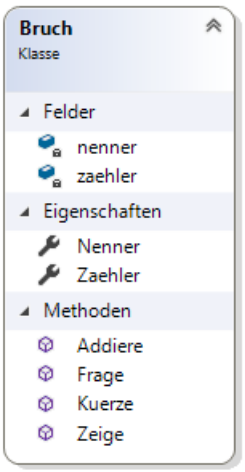
- Oben steht der **Name** der Klasse.
- In der Mitte stehen die **Merkmale**.
  Hinter dem Namen eines Merkmals gibt man seinen Datentyp (siehe unten) an. Bei den Eigenschaften von C# handelt es sich nach obigen Erläuterungen eigentlich um Zugriffs*methoden*, die aber syntaktisch wie (öffentlich verfügbare) Felder angesprochen werden. Es hängt vom Adressaten eines Klassendiagramms (z. B. Entwickler-Teamkollege oder Anwender der Klasse) ab, ob man als Merkmale die Felder, die Eigenschaften oder beides angibt. Stellt man Felder und Eigenschaften getrennt dar, resultiert ein UML-Klassendiagramm mit *vier* Bereichen (siehe unten).
- Unten stehen die **Handlungskompetenzen** (Methoden).
  In Anlehnung an eine in vielen Programmiersprachen (z. B. in C#) übliche und noch ausführlich zu behandelnde Syntax zur Methodendefinition gibt man für die Argumente eines Methodenaufrufs (mit Spezifikationen zur gewünschten Ausführungsart bzw. mit Details der gesendeten Nachricht) den Datentyp an.

Bei der `Bruch`-Klasse erhält man die folgende Darstellung, wenn die Eigenschaften aus der Anwenderperspektive betrachtet werden (als Merkmale und nicht als Methodenpaare):

[1] Während die UML im akademischen Bereich nachdrücklich empfohlen wird, ist ihre Verwendung in der Software-Branche noch entwicklungsfähig, wie empirische Studien gezeigt haben (siehe z. B. Baltes & Diehl 2014, Petre 2013).



Die im Kurs bevorzugte Entwicklungsumgebung Visual Studio Community 2019 erstellt auf Wunsch zur Klasse `Bruch` das folgende Diagramm, wobei Felder, Eigenschaften und Methoden als Member zu sehen sind:



Sind bei einer Anwendung *mehrere* Klassen beteiligt, dann sind auch die *Beziehungen* zwischen den Klassen wesentliche Bestandteile des Modells. In einem UML-Klassendiagramm können u. a. die folgenden Beziehungen zwischen Klassen (bzw. zwischen den Objekten von Klassen) dargestellt werden:

- Spezialisierung bzw. Vererbung („Ist-ein - Beziehung“)
  Beispiel: Ein Lieferwagen ist ein spezielles Auto.
- Komposition („Hat - Beziehung“)
  Beispiel: Ein Auto hat einen Motor.
- Assoziation („Kennt - Beziehung“)
  Beispiel: Ein (intelligentes, autonomes) Auto kennt eine Liste von Parkplätzen.

Nach der sorgfältigen Modellierung per UML muss übrigens die Programmierung nicht am Punkt null beginnen, weil professionelle UML-Werkzeuge Teile des Quellcodes automatisch aus dem Modell erzeugen können. Die im Kurs bevorzugte Entwicklungsumgebung *Visual Studio* hat ein solches Werkzeug allerdings nur bis zur Version 2015 enthalten.

Weiterführende Informationen zur objektorientierten Analyse und Modellierung bieten z. B. Balzert (2011) und Booch et al. (2007).

1.3 Objektorientierte Programmierung

In unserem Beispielprojekt soll nun die Klasse `Bruch` in der Programmiersprache C# kodiert werden, wobei die Felder (Instanzvariablen) zu deklarieren, sowie Eigenschaften und Methoden zu implementieren sind. Es resultiert der sogenannte **Quellcode**, der am besten in einer Textdatei namens **Bruch.cs** untergebracht wird.

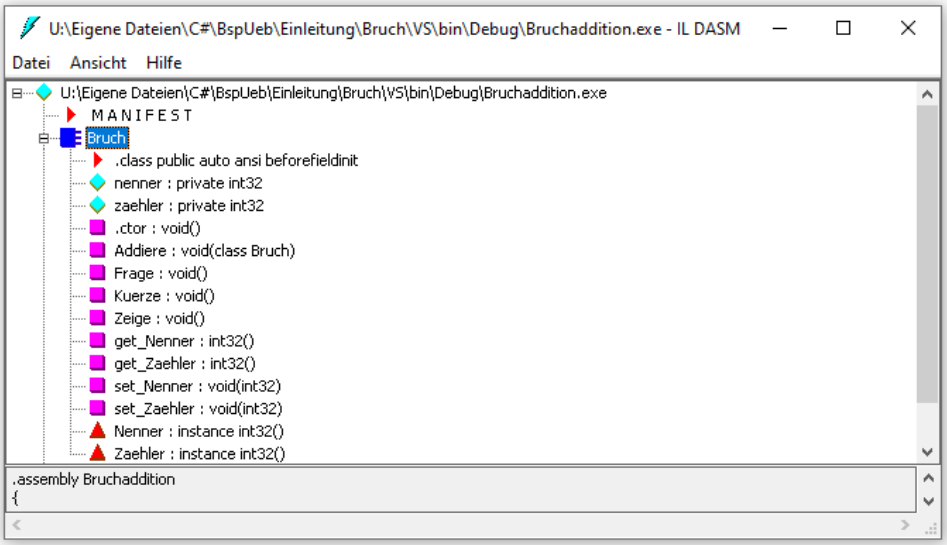
Zwar sind Ihnen die meisten Details der folgenden Klassendefinition selbstverständlich jetzt noch fremd, doch sind die Variablendeklarationen sowie die Eigenschafts- und Methodenimplementationen als zentrale Bestandteile leicht zu erkennen:

```
using System;

public class Bruch {
   int zaehler,  // wird automatisch mit 0 initialisiert
       nenner = 1;

   public int Zaehler {
      get {
         return zaehler;
      }
      set {
         zaehler = value;
      }
   }

   public int Nenner {
      get {
         return nenner;
      }
      set {
         if (value != 0)
            nenner = value;
      }
   }

   public void Zeige() {
      Console.WriteLine("   {0}\n  -----\n   {1}\n", zaehler, nenner);
   }

   public void Kuerze() {
      // größten gemeinsamen Teiler mit dem Euklidischen Algorithmus bestimmen
      if (zaehler != 0) {
         int az = Math.Abs(zaehler);
         int an = Math.Abs(nenner);
         while (az != an)
            if (az > an)
               az = az - an;
            else
               an = an - az;
         zaehler = zaehler / az;
         nenner  = nenner / az;
      } else
         nenner = 1;
   }

   public void Addiere(Bruch b) {
      zaehler = zaehler*b.nenner + b.zaehler*nenner;
      nenner = nenner*b.nenner;
      Kuerze();
   }
```

```
    public void Frage() {
       Console.Write("Zähler: ");
       zaehler = Convert.ToInt32(Console.ReadLine());
       Console.Write("Nenner: ");
       Nenner = Convert.ToInt32(Console.ReadLine());
    }
}
```

Weil für die beiden Felder (`zaehler`, `nenner`) die voreingestellte **private**-Deklaration unverändert gilt, ist im Beispielprogramm das Prinzip der Datenkapselung realisiert. Eigenschaften und Methoden werden durch die Verwendung des Modifikators **public** für die Verwendung in klassenfremden Methoden freigegeben. Außerdem wird für die Klasse selbst mit dem Modifikator **public** die Verwendung in beliebigen .NET - Programmen erlaubt.

Das zusammen mit der im Kurs bevorzugten Entwicklungsumgebung *Visual Studio Community 2019* (siehe Abschnitt 3.1) installierte Hilfsprogramm **ILDasm** (ausführbare Datei: **ildasm.exe**) liefert die folgende Beschreibung der Klasse `Bruch`:[1]



Felder werden durch eine türkis gefärbte Raute (🔷), Methoden durch ein Magenta-farbiges Quadrat (🟪) und Eigenschaften durch ein rotes, mit der Spitze nach oben zeigendes Dreieck (🔺) dargestellt.

Hier bestätigt sich übrigens die Andeutung von Abschnitt 1.2, dass hinter den C# - Eigenschaften letztlich Methoden für Lese- und Schreibzugriffe stehen (siehe z. B. `get_Nenner()`, `set_Nenner()`).

Wie Sie bei späteren Beispielen erfahren werden, dienen in einem objektorientierten Programm beileibe nicht alle Klassen zur Modellierung des Aufgabenbereichs. Es sind auch Objekte aus der Welt des Computers zu repräsentieren (z. B. Fenster der Benutzeroberfläche, Netzwerkverbindungen, Störungen des normalen Programmablaufs).

---

[1] Mit dem Visual Studio 2019 wird das Programm **ildasm.exe** unter Windows 10 im folgenden Ordner installiert:
C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.8 Tools
Wie der Speicherort vermuten lässt, kann das Programm nur Assemblies für das .NET Framework verarbeiten.

1.4 Algorithmen

Zu Beginn von Kapitel 1 wurden mit der *Modellierung des Anwendungsbereichs* und der *Realisierung von Algorithmen* zwei wichtige Aufgaben der Software-Entwicklung genannt, von denen die letztgenannte bisher kaum zur Sprache kam. Auch im weiteren Verlauf des Kurses wird die explizite Diskussion von Algorithmen (z. B. hinsichtlich Korrektheit, Terminierung und Aufwand) keinen großen Raum einnehmen. Wir werden uns intensiv mit der Programmiersprache C# sowie der .NET - Klassenbibliothek beschäftigen und dabei mit möglichst einfachen Beispielprogrammen (Algorithmen) arbeiten. Damit die Beschäftigung mit Algorithmen im Kurs nicht ganz fehlt, werden wir im Rahmen des Bruchrechnungsbeispiels alternative Verfahren zum Kürzen von Brüchen betrachten.

Unser Einführungsbeispiel verwendet in der Methode `Kuerze()` den bekannten und nicht gänzlich trivialen **euklidischen Algorithmus**, um den größten gemeinsamen Teiler (GGT) von Zähler und Nenner eines Bruchs zu bestimmen, durch den zum optimalen Kürzen beide Zahlen zu dividieren sind. Beim euklidischen Algorithmus wird die leicht zu beweisende Aussage genutzt, dass für zwei natürliche Zahlen (1, 2, 3, …) u und v ($u > v$) der GGT gleich dem GGT von v und (u - v) ist:

Ist t ein Teiler von u und v, dann gibt es natürliche Zahlen t_u und t_v mit $t_u > t_v$ und

$$u = t_u \cdot t \quad \text{sowie} \quad v = t_v \cdot t$$

Folglich ist t auch ein Teiler von (u - v), denn:

$$u - v = (t_u - t_v) \cdot t$$

Ist andererseits t ein Teiler von u und (u – v), dann gibt es natürliche Zahlen t_u und t_d mit $t_u > t_d$ und

$$u = t_u \cdot t \quad \text{sowie} \quad (u - v) = t_d \cdot t$$

Folglich ist t auch ein Teiler von v:

$$u - (u - v) = v = (t_u - t_d) \cdot t$$

Weil die Paare (u, v) und (v, u - v) dieselben Mengen gemeinsamer Teiler besitzen, sind auch die größten gemeinsamen Teiler identisch.

Beim Übergang von

$$(u, v) \quad \text{mit } u > v > 0$$

zu

$$(v, u - v) \quad \text{mit } v > 0 \text{ und } u - v > 0$$

wird die größere von den beiden Zahlen durch eine echt kleinere Zahl ersetzt, während der GGT identisch bleibt.

Wenn v und (u - v) in einem Prozessschritt identisch werden, ist der GGT gefunden. Das muss nach endlich vielen Schritten passieren, denn:

- Solange die beiden Zahlen im aktuellen Schritt k noch *verschieden* sind, resultieren im nächsten Schritt k+1 zwei neue Zahlen mit einem echt kleineren Maximum.
- Alle Zahlen bleiben > 0.
- Das Verfahren endet in endlich vielen Schritten, eventuell mit $v = u - v = 1$.

Weil die Zahl 1 als trivialer Teiler zugelassen ist, existiert zu zwei natürlichen Zahlen immer ein größter gemeinsamer Teiler, der eventuell gleich 1 ist.

Diese Ergebnisse werden in der Methode `Kuerze()` folgendermaßen ausgenutzt:

> Es wird geprüft, ob Zähler und Nenner identisch sind. Trifft dies zu, ist der GGT gefunden (identisch mit Zähler und Nenner). Anderenfalls wird die größere der beiden Zahlen durch deren Differenz ersetzt, und mit diesem vereinfachten Problem startet das Verfahren neu.

Man erhält auf jeden Fall in endlich vielen Schritten zwei identische Zahlen und damit den GGT.

Der beschriebene Algorithmus eignet sich dank seiner Einfachheit gut für das Einführungsbeispiel, ist aber in Bezug auf den erforderlichen Berechnungsaufwand nicht überzeugend. In einer Übungsaufgabe zum Abschnitt 4.7 werden Sie eine erheblich effizientere Variante implementieren.

1.5 Startklasse und Main() - Methode

Bislang wurde im Anwendungsbeispiel aufgrund einer objektorientierten Analyse des Aufgabenbereichs die Klasse `Bruch` entworfen und in C# realisiert. Wir verwenden nun die Klasse `Bruch` in einer Konsolenanwendung zur Addition von zwei Brüchen. Dabei bringen wir einen Akteur ins Spiel, der `Bruch`-Objekte erzeugt und ihnen Nachrichten zustellt, die (zusammen mit dem Verhalten des Anwenders) den Programmablauf voranbringen.

In diesem Zusammenhang ist von Bedeutung, dass es in *jedem* C# - Programm eine **Startklasse** geben muss, die eine Methode mit dem Namen **Main()** in ihrem klassenbezogenen Handlungsrepertoire besitzt.[1] Beim Start eines Programms wird die Startklasse aufgefordert, ihre **Main()** - Methode auszuführen. Diese Methode kann als *Einsprungpunkt* (engl. *Entrypoint*) für das Programm bezeichnet werden.

Es bietet sich an, die oben angedachte Handlungssequenz des Bruchadditionsprogramms in der obligatorischen **Main()** - Methode der Startklasse unterzubringen.

Obwohl prinzipiell möglich, ist es nicht sinnvoll, die auf Wiederverwendbarkeit hin konzipierte Klasse `Bruch` mit der Startmethode für eine sehr spezielle Anwendung zu belasten. Daher definieren wir eine zusätzliche Klasse namens `Bruchaddition`, die nicht als Bauplan für Objekte dienen soll und auch kaum Recycling-Chancen besitzt. Ihr Handlungsrepertoire kann sich auf die *Klassenmethode* **Main()** zur Ablaufsteuerung im Bruchadditionsprogramm beschränken. Indem wir eine *neue* Klasse definieren und dort `Bruch`-Objekte verwenden, wird u. a. gleich demonstriert, wie leicht das Hauptergebnis unserer bisherigen Arbeit (die Klasse `Bruch`) für verschiedene Projekte genutzt werden kann.

In der `Bruchaddition`-Methode **Main()** werden zwei Objekte (Instanzen) aus der Klasse `Bruch` per **new**-Operator erzeugt und mit der Ausführung verschiedener Methoden beauftragt:[2]

---

[1] Genau genommen kann statt einer Klasse auch eine sogenannte *Struktur* die Rolle des Starters übernehmen und die **Main()** - Methode realisieren, siehe:
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/main-and-command-args/
Mit den Strukturen, die als leichtgewichtige Klassen aufgefasst werden können, beschäftigen wir uns im Abschnitt 6.1.

[2] Mit dem Erzeugen von Objekten einer Klasse per **new**-Operator werden wir uns später noch ausführlich beschäftigen. Dabei ist eine spezielle Instanzmethode beteiligt, ein sogenannter **Konstruktor**.

Quellcode in **Bruchaddition.cs**	Ausgabe (Eingaben **fett**)

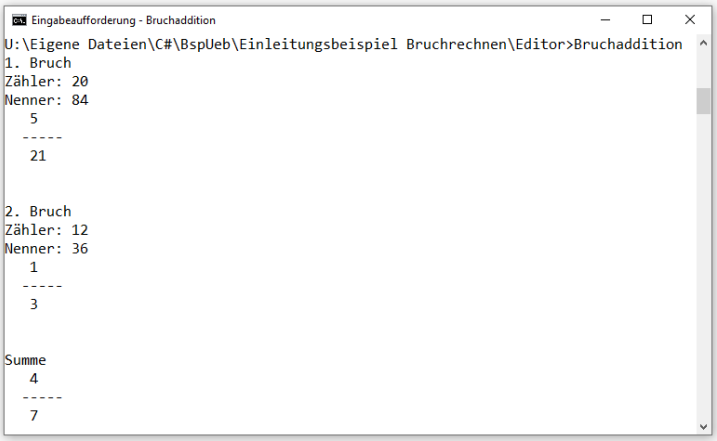
```
using System;

class Bruchaddition {
   static void Main() {
      Bruch b1 = new Bruch(), b2 = new Bruch();

      Console.WriteLine("1. Bruch");
      b1.Frage();
      b1.Kuerze();
      b1.Zeige();

      Console.WriteLine("\n2. Bruch");
      b2.Frage();
      b2.Kuerze();
      b2.Zeige();

      Console.WriteLine("\nSumme");
      b1.Addiere(b2);
      b1.Zeige();
      Console.ReadLine();
   }
}
```

```
1. Bruch
Zähler: 20
Nenner: 84
    5
  -----
   21


2. Bruch
Zähler: 12
Nenner: 36
    1
  -----
    3


Summe
    4
  -----
    7
```

Zum Ausprobieren startet man aus dem Ordner

...\BspUeb\Einleitungsbeispiel Bruchrechnen\Editor

(zu finden an der im Vorwort vereinbarten Stelle) das Programm **Bruchaddition.exe**, z. B.:



In der **Main()** - Methode der Klasse `Bruchaddition` kommen nicht nur `Bruch`-Objekte zum Einsatz. Wir nutzen dort auch die Kompetenzen der Klasse **Console** aus der Standardbibliothek und rufen ihre Klassenmethoden **WriteLine()** und **ReadLine()** auf.

Wir haben zur Lösung der Aufgabe, ein Programm für die Addition von zwei Brüchen zu erstellen, zwei Klassen mit folgender Rollenverteilung definiert:

- Die Klasse `Bruch` enthält den Bauplan für die wesentlichen Akteure im Aufgabenbereich. Dort alle Merkmale und Handlungskompetenzen von Brüchen zu konzentrieren, hat folgende Vorteile:
    - Die Klasse kann in verschiedenen Programmen eingesetzt werden (Wiederverwendbarkeit). Dies fällt vor allem deshalb so leicht, weil die Objekte Handlungskompetenzen (Methoden) besitzen **und** alle erforderlichen Instanzvariablen mitbringen.
    - Beim Umgang mit den `Bruch`-Objekten sind wenige Probleme zu erwarten, weil nur klasseneigene Methoden direkten Zugang zu kritischen Merkmalen haben (Datenkapselung). Sollten doch Fehler auftreten, sind die Ursachen in der Regel schnell identifiziert.

  Wir müssen bei der Definition der Klasse `Bruch` ihre allgemeine Verfügbarkeit explizit mit dem Zugriffsmodifikator **public** genehmigen. Per Voreinstellung ist eine Klasse nur intern (in der eigenen Übersetzungseinheit) verfügbar, was im Kurs noch ausführlich zu behandeln ist.[1]
- Die Klasse `Bruchaddition` dient *nicht* als Bauplan für Objekte, sondern enthält die Klassenmethode **Main()**, die beim Programmstart automatisch aufgerufen wird und dann für einen speziellen Einsatz von `Bruch`-Objekten sorgt. Mit einer Wiederverwendung des `Bruchaddition`-Quellcodes in anderen Projekten ist kaum zu rechnen.

In der Regel bringt man den Quellcode jeder Klasse in einer eigenen Textdatei unter, die den Namen der Klasse trägt, ergänzt um die Namenserweiterung **.cs**. In C# sind allerdings Quellcodedateien mit mehreren Klassen und einem beliebigen Dateinamen erlaubt.

Während bei den Namen von C# - Klassen die Groß-/Kleinschreibung signifikant ist, spielt sie bei Dateinamen unter Windows bekanntlich keine Rolle. Wenn eine C# - Quellcodedatei der üblichen Praxis folgend genau *eine* Klassendefinition enthält, sollte der Dateiname *inkl. Groß-/Kleinschreibung* von der Klasse übernommen werden. Microsoft hat seine Konvention zur Benennung der Dateien mit den BCL-Klassen offenbar geändert:

- Die Namen der Dateien mit dem .NET Framework - Quellcode verwenden nur Kleinbuchstaben, z. B. **string.cs**.[2]
- Die Namen der Dateien mit dem .NET 5.0 - Quellcode übernehmen die Groß-/Kleinschreibung von den Klassennamen, z. B. **String.cs**.[3]

1.6 Ausblick auf Anwendungen mit grafischer Bedienoberfläche

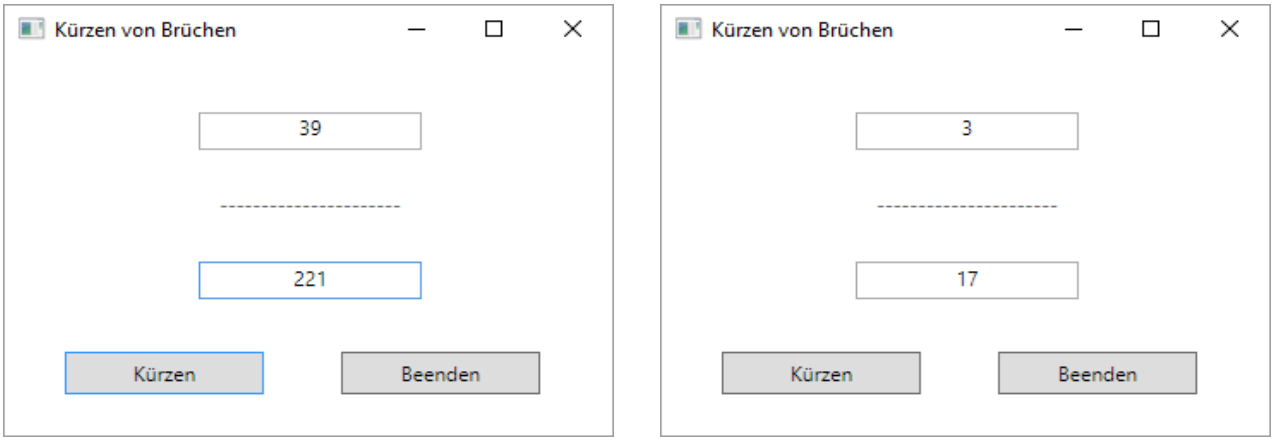
Das obige Beispielprogramm arbeitet der Einfachheit halber mit einer Konsolen-orientierten Ein- und Ausgabe. Nachdem wir im Kurs in dieser übersichtlichen Umgebung grundlegende C# - Sprachelemente kennengelernt haben, werden wir uns selbstverständlich auch mit der Programmierung von grafischen *Bedienoberflächen* beschäftigen. Im folgenden Programm zum Kürzen von Brüchen wird die oben definierte Klasse `Bruch` verwendet, wobei an Stelle ihrer Methoden `Frage()` und `Zeige()` jedoch grafikorientierte Techniken zum Einsatz kommen:

---

[1] Die Übersetzungseinheit (das Assembly **Bruchaddition.exe**) enthält die Klassen `Bruchaddition` und `Bruch`, sodass hier der Zugriffsmodifikator **public** für die Klasse `Bruch` keine Rolle spielt. Im Abschnitt 1.6 mit einem Ausblick auf Anwendungen mit grafischer Bedienoberfläche wird jedoch ein Programm vorgestellt, das ein eigenständiges, „externes“ Assembly mit der Klasse `Bruch` verwendet. Das ist der typische Fall, weil die Klasse `Bruch` nicht neu übersetzt werden muss.

[2] Der Quellcode der BCL zum .NET Framework ist hier zu finden: https://referencesource.microsoft.com/download.html

[3] Der Quellcode der BCL zum .NET 5.0 ist hier zu finden: https://github.com/dotnet/runtime, freundlicher präsentiert in: https://source.dot.net/



Mit dem Quellcode zur Gestaltung der grafischen Oberfläche könnten Sie im Moment noch nicht allzu viel anfangen. Wir werden das Programm im Abschnitt 5.12 erstellen.

1.7 Zusammenfassung zum Kapitel 1

Im Kapitel 1 sollten Sie einen ersten Eindruck von der Software-Entwicklung mit C# gewinnen. Alle dabei erwähnten Konzepte der objektorientierten Programmierung und die technischen Details der Realisierung in C# werden bald systematisch behandelt und sollten Ihnen daher im Moment noch keine Kopfschmerzen bereiten. Trotzdem kann es nicht schaden, an dieser Stelle einige Kernaussagen vom ersten Kapitel zu wiederholen:

- Vor der Programmentwicklung findet die **objektorientierte Analyse** der Aufgabenstellung statt. Dabei werden per Abstraktion die beteiligten Klassen und ihre Beziehungen identifiziert.
- Ein Programm besteht aus **Klassen**. Unsere Beispielprogramme zum Erlernen elementarer Sprachelemente werden oft mit einer einzigen Klasse auskommen. Praxisgerechte Programme bestehen in der Regel aus zahlreichen Klassen.
- Eine Klasse ist charakterisiert durch **Merkmale und Methoden**.
- Ein Merkmal bzw. eine Methode wird entweder den Objekten einer Klasse oder der Klasse selbst zugeordnet.
- Bei einer technischen Betrachtungsweise besitzt eine Klasse Felder, Eigenschaften und Methoden als Member.
- Die Felder sind in der Regel vor dem direkten Zugriff durch andere Klassen geschützt. Die Methoden sind hingegen oft auch für andere Klassen nutzbar und ermöglichen somit eine Kommunikation.
- In C# kann fremden Klassen für ein Feld durch eine sogenannte *Eigenschaft* ein kontrollierter Zugang verschafft werden, wobei es sich letztlich um ein Paar von Methoden für den lesenden und den schreibenden Zugriff handelt.
- Eine Klasse dient in der Regel als **Bauplan für Objekte**, kann aber auch selbst aktiv werden (Methoden ausführen und aufrufen).
- In den Methodendefinitionen werden Algorithmen realisiert. Dabei kommen selbst erstellte Klassen zum Einsatz, aber auch vordefinierte Klassen aus diversen Bibliotheken. Von den einbezogenen Bibliotheken spielt die .NET - Standardbibliothek (*Base Class Library*, BCL) eine herausragende Rolle.

- Im Programmablauf kommunizieren die Akteure (Objekte und Klassen) durch den Aufruf von Methoden miteinander, wobei aber in der Regel noch „externe Kommunikationspartner“ (z. B. Benutzer, andere Programme) beteiligt sind.
- Beim Programmstart wird die Startklasse vom Laufzeitsystem aufgefordert, die Methode **Main()** auszuführen. Ein Hauptzweck dieser Methode besteht oft darin, Objekte zu erzeugen und somit „Leben auf die objektorientierte Bühne zu bringen“.

1.8 Übungsaufgaben zum Kapitel 1

1) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. Die Methoden einer Klasse sind grundsätzlich öffentlich.
2. Bei den Eigenschaften von C# handelt es sich um Paare von Zugriffs*methoden*, die aber syntaktisch wie (öffentlich verfügbare) Felder verwendet werden.
3. In C# besitzt jede Klasse eine Methode namens **Main()** in ihrem klassenbezogenen Handlungsrepertoire.
4. Der Zustand eines Objekts darf nur durch Methoden der eigenen Klasse verändert werden.

2) Warum steigt die Produktivität der Software-Entwicklung durch objektorientiertes Programmieren?

2 Die .NET - Plattform

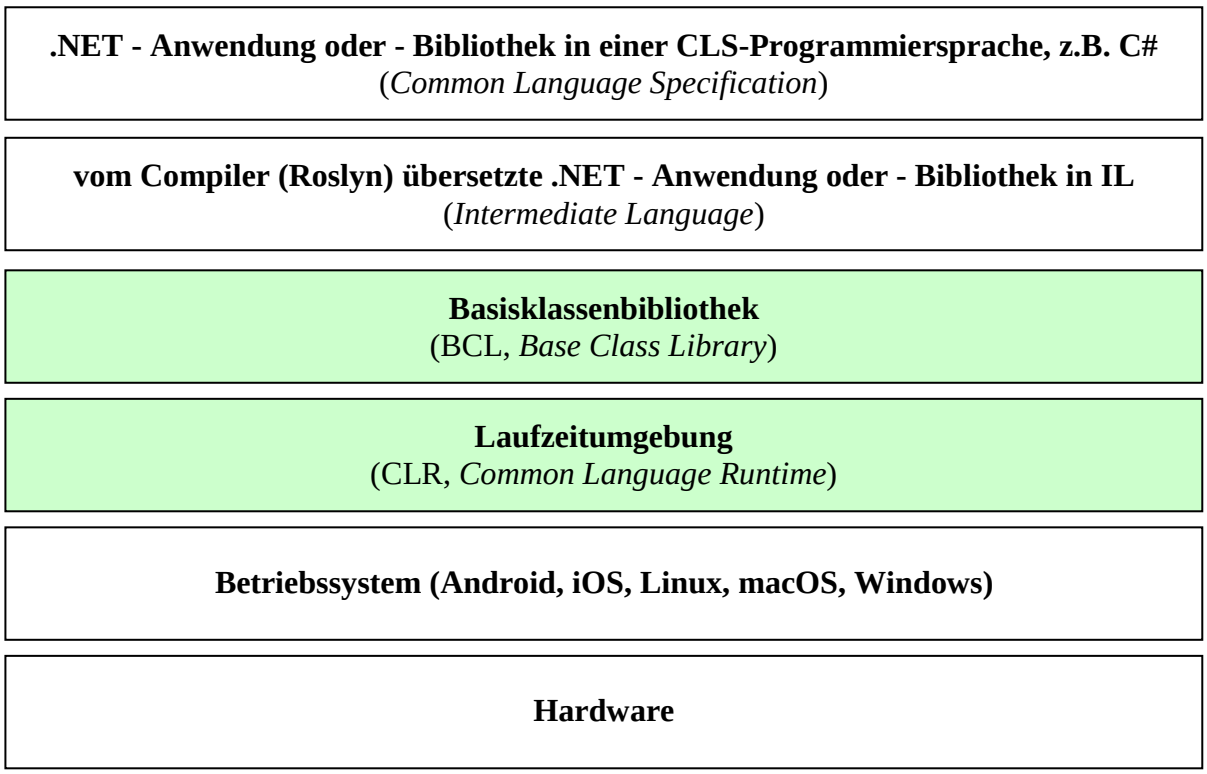
Eben haben Sie C# als eine Programmiersprache kennengelernt, die Ausdrucksmittel zur Modellierung des Anwendungsbereichs und zur Formulierung von Algorithmen bereitstellt. Unter einem *Programm* wurde dabei der vom Entwickler zu formulierende *Quellcode* verstanden. Während *Sie* derartige Texte bald ohne große Mühe lesen und begreifen werden, kann die CPU (*Central Processing Unit*) eines Rechners nur einen maschinenspezifischen Satz von Befehlen verstehen, die als Folge von Nullen und Einsen kodiert werden müssen (*Maschinencode*). Die ebenfalls CPU-spezifische Assembler-Sprache stellt eine für Menschen lesbare Form des Maschinencodes dar. Mit dem Assembler- bzw. Maschinenbefehl

```
mov eax, 4
```

einer CPU aus der x86-Familie, die auf Desktop-Computern in der Regel anzutreffen ist, wird z. B. der Wert 4 in das EAX-Register (ein Speicherort im Prozessor) geschrieben. Die CPU holt sich einen Maschinenbefehl nach dem anderen aus dem Hauptspeicher und führt ihn aus, wobei heutzutage (2020) die CPU eines handelsüblichen Arbeitsplatzrechners (mit GHz-Taktfrequenz und zahlreichen Kernen/Threads) mehrere hundert Milliarden Befehle pro Sekunde (*Instructions Per Second*, IPS) schafft.[1]

Ein Quellcode-Programm muss also erst in Maschinencode übersetzt werden, damit es von einem Rechner ausgeführt werden kann. Wie dies bei C# und anderen .NET - Programmiersprachen geschieht, sehen wir uns nun näher an. Wir behandeln nicht die komplette .NET - Softwarearchitektur, sondern beschränken uns auf die für Programmierer wichtigen Hintergrundinformationen.

Die folgende Abbildung zeigt, welcher Stapel von Technologien bei der Ausführung eines in C# geschriebenen Programms für eine .NET - Implementation beteiligt ist:



[1] https://de.wikipedia.org/wiki/Instruktionen_pro_Sekunde
https://en.wikipedia.org/wiki/Instructions_per_second

2.1 .NET - Implementationen

Bei der .NET - Plattform handelt es sich um eine von der Firma Microsoft entwickelte Sammlung von Technologien zur Modernisierung und Vereinfachung der Anwendungsentwicklung. Wie schon im Vorwort angesprochen, hat sich das .NET - Ökosystem diversifiziert, soll aber mit .NET 5 wieder vereinheitlicht werden.

In der folgenden Tabelle werden von den auf der Microsoft-Webseite

https://docs.microsoft.com/en-us/dotnet/standard/net-standard

unterschiedenen .NET - Implementationen sechs beschrieben:

Implementation	**Unterstützte Betriebssysteme**	**Version im März 2021**	**Mögliche Anwendungstypen**
.NET	Linux, macOS, Windows	5.0	Konsole, Web, Desktop (Windows)
.NET Core	Linux, macOS, Windows	3.1	Konsole, Web, Desktop (Windows)[1]
.NET Framework	Windows	4.8	Desktop, Web
Mono[2]	Linux, macOS, Windows	6.12	Desktop, Web
Xamarin (Mobil)	Android, iOS	11.0 (Android) 14.0 (iOS)	Mobil
UWP (Universal Windows Platform)	Windows 10, Xbox, HoloLens	10.0.19041	Desktop

C# kann in allen .NET - Implementationen genutzt werden, wobei aber unterschiedliche Sprachversionen verfügbar sind.

Als gemeinsame Basis für alle .NET - Implementationen hat Microsoft den **.NET Standard** definiert. Eine Version dieses Standards besteht aus einer Sammlung von Spezifikationen für .NET - APIs. Auf der oben genannten Webseite wird für die verschiedenen Versionen der einzelnen Implementationen dokumentiert, welche Version des .NET Standards sie jeweils unterstützen. Z. B. wird die Version 2.0 unterstützt von:

- .NET 5.0
- .NET Core 3.0
- .NET Framework ab 4.6.1
- Mono ab 5.4
- Xamarin.Android ab 8.0
- Xamarin.iOS ab 10.14
- UWP ab 10.0.16299

Zusammen mit dem Visual Studio 2019 ab Version 16.8.x ist .NET 5.0 zusammen mit seinem Vorgänger .NET Core bequem zu installieren (siehe Abschnitt 3.1). Weil das .NET Framework ein Bestandteil von Windows 10 ist, haben wir im Kurs zur Verfügung:

- .NET Framework 4.8 mit C# 7.3
  Weil das .NET Framework Bestandteil aller aktuellen Windows-Betriebssysteme ist, können Programme für das .NET Framework auf allen Windows-Rechnern ohne die vorherige Installation einer Laufzeitumgebung ausgeführt werden.

---

[1] Ab .NET Core 3.0 werden *unter Windows* auch GUI-Anwendungen unterstützt.

[2] http://www.mono-project.com/

- .NET Core 3.1 mit C# 8.0
  Weil .NET 5.0 als der Nachfolger von .NET Core betrachtet werden kann, werden wir uns im Kurs nur selten speziell um .NET Core kümmern.
- .NET 5.0 mit C# 9.0
  .NET 5.0 bietet die aktuellste C# - Version und kann fast alle im Kurs vorgestellten Programme ausführen.

2.1.1 .NET Framework vs. .NET 5.0

Als Laufzeitumgebung für .NET - Programme kommen aktuell (im März 2021) unter Windows vor allem in Frage:

- .NET Framework 4.x mit C# 7.3
- .NET 5.0 mit C# 9.0

2.1.1.1 .NET Framework

Das .NET - Framework ist ein fester Bestandteil des Betriebssystems Windows und war lange Zeit die einzige .NET -Implementation. Es hat seit 2002 zahlreiche Aktualisierungen erfahren und ist 2019 bei der finalen Version 4.8 angekommen, die nicht mehr weiterentwickelt wird. Sein wesentlicher Vorteil gegenüber den jüngeren .NET - Implementationen .NET Core und .NET 5.0 besteht darin, dass ein für das .NET Framework entwickeltes Programm auf jedem Windows-Rechner ausgeführt werden kann, ohne dass für die Installation einer Laufzeitumgebung gesorgt werden muss.[1]

Das .NET Framework und die Programmiersprache C# sind gemeinsam entwickelt worden, wie die folgende Tabelle zeigt:[2]

.NET - Framework	CLR-Version	Erscheinungsjahr	C#
1.0	1.0	2002	1.0
1.1	1.1	2003	1.2
2.0	2.0	2005	2.0
3.0	2.0	2006	2.0
3.5	2.0	2007	3.0
4.0	4	2010	4.0
4.5	4	2012	5.0
4.6	4	2015	6.0
4.6.2	4	2017	7.0
4.7	4	2017	7.1
4.7.1	4	2017	7.2
4.7.2	4	2018	7.3
4.8	4	2019	7.3

Wer Anwendungen entwickeln möchte, die auf jedem aktuellen Windows-Rechner die erforderliche Laufzeitumgebung (.NET Framework) in gut gepflegtem Zustand vorfinden, muss sich auf C# 7.3 beschränken, weil die jüngeren C# - Versionen nur für .NET Core (C# 8.0) bzw. .NET 5.0 (C# 9.0) verfügbar sind.

---

[1] Zwar wurden die aktuellen Windows-Varianten mit unterschiedlichen Framework-Versionen ausgeliefert (siehe unten), doch ist auch die minimal vorhandene .NET Framework - Version 4.5.1 für die meisten Zwecke ausreichend.

[2] http://stackoverflow.com/questions/247621/what-are-the-correct-version-numbers-for-c
https://en.wikipedia.org/wiki/C_Sharp_%28programming_language%29
https://en.wikipedia.org/wiki/.NET_Framework_version_history

Der folgenden Tabelle ist für die aktuellen (noch durch Sicherheits-Updates unterstützten) Windows-Versionen die bei Auslieferung enthaltene .NET Framework - Version zu entnehmen:[1]

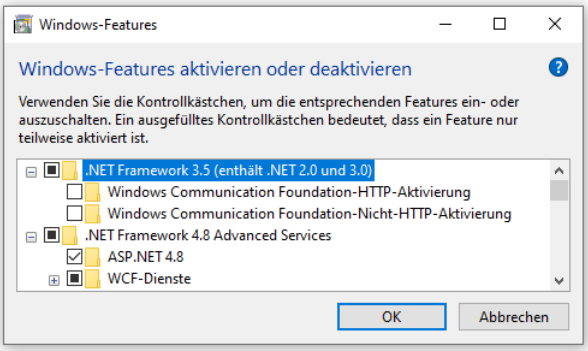
Windows-Version	Ausgelieferte .NET - Framework -Version
8.1	4.5.1
10 (1803) 10 (1809)	4.7.2
ab 10, 1903	4.8

Durch kostenlos bei Microsoft verfügbare Installationspakete lässt sich die mit einem Betriebssystem ausgelieferte .NET - Version aktualisieren.[2] So ist es z. B. problemlos (und kostenlos) möglich, unter Windows 8.1 die aktuelle Version 4.8 des .NET - Frameworks zu installieren.

Es kann erforderlich werden, eine ältere Framework-Version via Windows-Systemsteuerung (zu starten durch Eingabe von *Systemsteuerung* oder *Control* im Taskleisten-Suchfeld von Windows) über

Programme und Features > Windows-Features aktivieren oder deaktivieren

freizuschalten, weil .NET - Programme zur Vermeidung von Versionskonflikten von einer CLR mit einem bestimmten Versionsstand ausgeführt werden wollen:



Die .NET - Version 3.5 bringt freundlicherweise die älteren Versionen 3.0 und 2.0 gleich mit.

Als .NET Framework - Installationsordner werden (ohne Änderungsmöglichkeit) verwendet:

.NET Framework - Version	Installationsordner
2.0	x86: %SystemRoot%\Microsoft.NET\Framework\v2.0.50727 x64: %SystemRoot%\Microsoft.NET\Framework64\v2.0.50727
3.0	x86: %SystemRoot%\Microsoft.NET\Framework\v3.0 x64: %SystemRoot%\Microsoft.NET\Framework64\v3.0
3.5	x86: %SystemRoot%\Microsoft.NET\Framework\v3.5 x64: %SystemRoot%\Microsoft.NET\Framework64\v3.5
ab 4.0	x86: %SystemRoot%\Microsoft.NET\Framework\v4.0.30319 x64: %SystemRoot%\Microsoft.NET\Framework64\v4.0.30319

Damit auf einem PC mit 64-bittiger Windows-Installation auch x86-abhängige Assemblies (siehe Abschnitt 2.4.7) laufen, werden dort *zwei* .NET Framework - Laufzeitumgebungen (mit 32 bzw. 64 Bit) installiert.

---

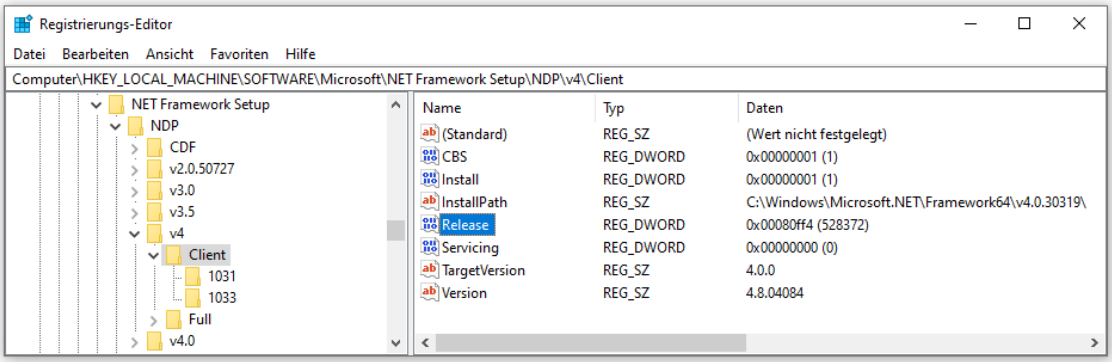
1 https://msdn.microsoft.com/de-de/library/5a4x27ek(v=vs.110).aspx

2 https://dotnet.microsoft.com/download/dotnet-framework

Zur Beschreibung des Verfahrens, wie die auf einem Rechner installierte .NET - Version festzustellen ist, benötigt Microsoft einen erstaunlich langen „Gewusst wie“ - Artikel.[1] Ab .NET 4.5 ist dem Registry-Key

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Full

die **Release**-Angabe



zu entnehmen und mit einer Tabelle zu vergleichen, z. B.:

DWORD-Eintrag	**Version**
Auf Systemen mit Windows 8.1: 378675	.NET Framework 4.5.1
Auf allen anderen Windows-Versionen: 378758	
Auf Systemen mit Windows 10, 1803: 461808	.NET Framework 4.7.2
Auf allen anderen Betriebssystemversionen: 461814	
Auf Systemen mit Windows 10, 1903, 1909: 528040	.NET Framework 4.8
Auf Systemen mit Windows 10, 2004: 528372	
Auf allen anderen Windows-Versionen: 528049	

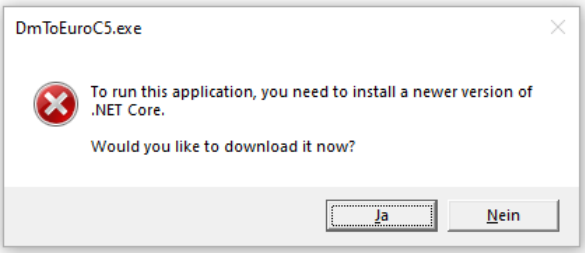
Das obige Bildschirmfoto gehört also zu einem Rechner mit dem Betriebssystem Windows 10, 2004.

2.1.1.2 .NET 5.0

Wählt man im Visual Studio für ein neues Projekt mit grafischer Bedienoberfläche z. B. die Vorlage **WPF App (.NET)**, dann ...

- kann man zur Entwicklung die C# - Version 9.0 verwenden,
- lässt sich das entstehende Programm von der .NET Framework - Laufzeitumgebung *nicht* ausführen.

Auf einem Rechner mit Windows 10 läuft das resultierende Programm also zunächst nicht:



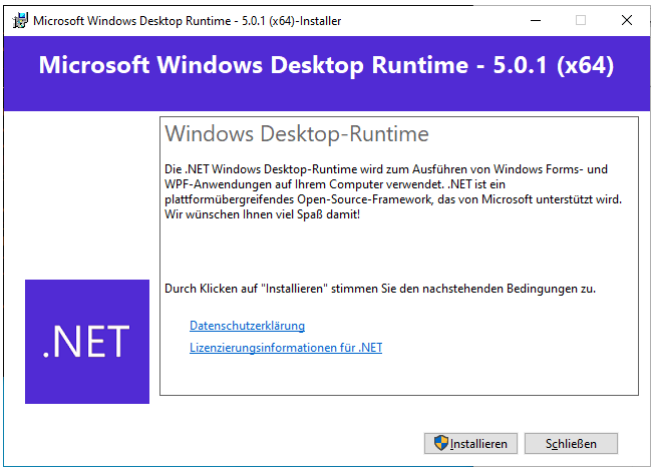
Um das Programm nutzen zu können, muss ein Kunde (z. B. über den Schalter **Ja** im Fehlermeldungsfenster) die **.NET Desktop Runtime 5** (z. B. in der Datei **windowsdesktop-runtime-5.0.1-win-x64.exe**) von der folgenden Webseite

---

[1] https://msdn.microsoft.com/de-de/library/hh925568(v=vs.110).aspx

https://dotnet.microsoft.com/download/dotnet/5.0

herunterladen und installieren:



Immerhin wird eine installierte .NET 5.0 - Desktop - Runtime später beim Windows-Update automatisch aktuell gehalten.[1]

Im weiteren Verlauf von Kapitel 2 verwenden wir das .NET Framework, weil sich in dieser auf Windows beschränkten .NET - Implementation der Prozess der Software-Erstellung etwas einfacher darstellt als in dem auf Betriebssystem-Universalität angelegten .NET 5.0. Z. B. lässt sich im .NET Framework aus C# - Quellcodedateien durch einen einfachen Aufruf des Compilers **csc.exe** ein ausführbares Programm erstellen, während .NET 5.0 einen komplexeren Erstellungsprozess benötigt, um den sich später die Entwicklungsumgebung kümmern wird. Für das im Kapitel 2 zu vermittelnde grundsätzliche Verständnis der .NET Plattform sind die Implementationsunterschiede irrelevant. Allen Implementationen gemeinsam ist z. B. die zweistufige Übersetzung von .NET - Quellcode ...

- über den IL-Code
- in den Maschinencode.

2.2 C# - Compiler und IL

Den (z. B. mit einem beliebigen Texteditor verfassten) C# - Quellcode übersetzt ein C# - **Compiler** in die **Intermediate Language** (**IL**).[2] Wenngleich dieser Zwischencode von keiner CPU direkt ausgeführt werden kann, hat er doch bereits viele Verarbeitungsschritte auf dem Weg vom Quell- zum Maschinencode durchlaufen. Die Übersetzung des Zwischencodes in die Maschinensprache einer konkreten CPU geschieht *Just-In-Time* bei der Ausführung des Programms durch die CLR (siehe Abschnitt 2.5).[3]

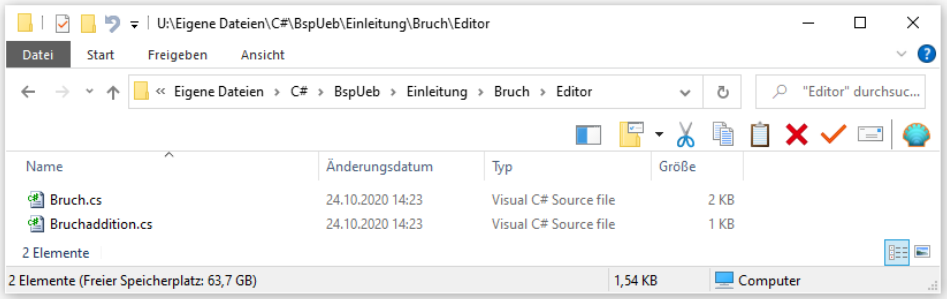
Befinden sich die beiden Quellcodedateien **Bruch.cs** und **Bruchaddition.cs** im aktuellen Verzeichnis

---

1 https://devblogs.microsoft.com/dotnet/net-core-updates-coming-to-microsoft-update/

2 Frühere Bezeichnungen:
- Die Bezeichnung **Common Intermediate Language (CIL)** ist noch weit verbreitet.
- Die ursprüngliche Bezeichnung **Microsoft Intermediate Language (MSIL)** wird nicht mehr verwendet.

3 Die Übersetzung vom IL-Code in Maschinencode sollte auf jeder Plattform gelingen, für die eine CLR verfügbar ist. Z. B. sollten sich die auf einem Rechner mit 64-bittiger Windows-Version erstellten .NET-Programme problemlos in einer 32-Bit - Umgebung nutzen lassen. Grundsätzlich ist diese Portabilität tatsächlich gegeben, jedoch wird im Abschnitt 2.4.7 von Ausnahmen zu berichten sein.



eines Konsolenfensters, dann kann ihre Übersetzung durch den zusammen mit dem Visual Studio 2019 installierten C# - Compiler **csc.exe** über das folgende Kommando

```
"C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\MSBuild\Current\Bin\Roslyn\csc" *.cs
```

veranlasst werden:

Weil sich der Ordner

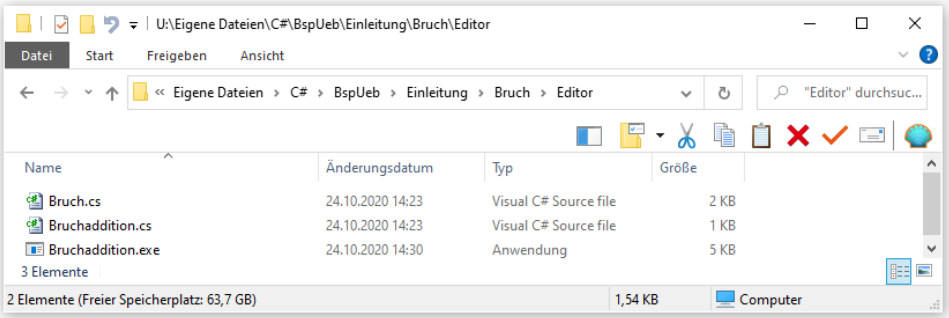
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\MSBuild\Current\Bin\Roslyn

mit dem C# - Compiler **csc.exe** per Voreinstellung nicht im Suchpfad für ausführbare Programme befindet, muss er im Kommando angegeben werden. Der eigentliche Auftrag an den Compiler, sämtliche Dateien im aktuellen Verzeichnis mit der Namenserweiterung **.cs** zu übersetzen, ist sehr übersichtlich:

```
>csc *.cs
```

Wir werden uns im Abschnitt 3.2.2 noch mit einigen Details des Compiler-Aufrufs beschäftigen.

Im Übersetzungsergebnis **Bruchaddition.exe**



ist u. a. der IL-Code der beiden Klassen `Bruch` und `Bruchaddition` enthalten. Besonders kurz sind die implizit zu einer C# - Eigenschaft (vgl. Abschnitt 1.2) vom Compiler erstellten **get**- und **set**-Methoden, z. B.:[1]

[1] Diese Ausgabe liefert das im Abschnitt 1.3 angesprochene Werkzeug **ILDasm** nach einem Doppelklick auf die interessierende Methode (siehe Seite 7).

```
public int Nenner {
  get {
    return nenner;
  }
  set {
    if (value != 0)
      nenner = value;
  }
}
```

C# - Compiler

⇒

Bruch::get_Nenner : int32()

Suchen Weitersuchen

```
.method public hidebysig specialname instance int32
        get_Nenner() cil managed
{
  // Code size       12 (0xc)
  .maxstack  1
  .locals init (int32 V_0)
  IL_0000:  nop
  IL_0001:  ldarg.0
  IL_0002:  ldfld      int32 Bruch::nenner
  IL_0007:  stloc.0
  IL_0008:  br.s       IL_000a
  IL_000a:  ldloc.0
  IL_000b:  ret
} // end of method Bruch::get_Nenner
```

Bruch::set_Nenner : void(int32)

Suchen Weitersuchen

```
.method public hidebysig specialname instance void
        set_Nenner(int32 'value') cil managed
{
  // Code size       17 (0x11)
  .maxstack  2
  .locals init (bool V_0)
  IL_0000:  nop
  IL_0001:  ldarg.1
  IL_0002:  ldc.i4.0
  IL_0003:  ceq
  IL_0005:  stloc.0
  IL_0006:  ldloc.0
  IL_0007:  brtrue.s   IL_0010
  IL_0009:  ldarg.0
  IL_000a:  ldarg.1
  IL_000b:  stfld      int32 Bruch::nenner
  IL_0010:  ret
} // end of method Bruch::set_Nenner
```

Offenbar resultiert aus der `Bruch`-Eigenschaft `Nenner` u. a. die Methode `get_Nenner()`, deren IL-Code aus 7 Anweisungen besteht.

Die konzeptionelle Verwandtschaft der IL mit dem Bytecode der Java-Plattform ist unverkennbar.

Alle .NET - Quellcodedateien werden unabhängig von der verwendeten Programmiersprache in denselben Zwischencode (in die Intermediate Language) übersetzt. Bei Beachtung gewisser Regeln (siehe nächsten Abschnitt) können verschiedene Programmierer bei der Erstellung von Klassen für ein gemeinsames Projekt jeweils die individuell bevorzugte Programmiersprache verwenden.

2.3 Common Language Specification

Mittlerweile sind für viele Programmiersprachen IL-Compiler verfügbar (z. B. C#, C++/CLI, Eiffel, F#, JScript.NET, IronPython, IronRuby, Visual Basic .NET, COBOL).[1] Allerdings unterstützen die .NET-Compiler in der Regel nicht den gesamten IL-Sprachumfang, sodass sich mit den Compilern zu verschiedenen .NET-Sprachen durchaus Klassen produzieren lassen, die *nicht* zusammenarbeiten können. Beispielweise erstellt der C# -Compiler bedenkenlos eine öffentliche Klasse mit zwei öffentlichen Methoden, deren Namen sich nur durch die Groß-/Kleinschreibung unterscheiden (z. B. `TuWas()` und `tuWas()`). Mit einer solchen Klasse können aber die Produktionen von Visual Basic .NET *nicht* kooperieren.

Microsoft hat unter dem Namen **Common Language Specification** (CLS) einen Sprachumfang definiert, den *jede* .NET-Programmiersprache erfüllen muss.[2] Beschränkt man sich bei der Klassendefinition auf diesen kleinsten gemeinsamen Nenner, ist die Interoperabilität mit anderen CLS-kompatiblen Klassen sichergestellt. Die .NET - Compiler überwachen die CLS-Kompatibilität, sodass z. B. der C# - Compiler im eben beschriebenen Beispiel warnt:

---

1 https://de.wikipedia.org/wiki/Liste_von_.NET-Sprachen

2 https://docs.microsoft.com/de-de/dotnet/standard/language-independence-and-language-independent-components

```
ClsIncomplient.cs(12,13): warning CS3005: Der Bezeichner
        "ClsIncomplient.TuWas()", der sich nur hinsichtlich der Groß- und
        Kleinschreibung unterscheidet, ist nicht CLS-kompatibel.
```

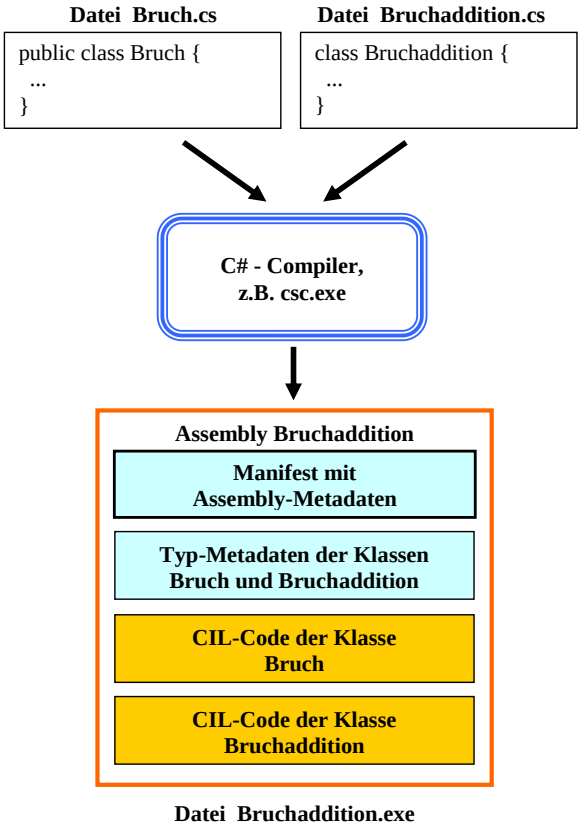
2.4 Assemblies und Metadaten

Die von einem .NET - Compiler erzeugten Binärdateien (mit IL-Code) werden als **Assemblies** bezeichnet und haben die Namenserweiterung:

- **.exe** (bei .NET - Anwendungen) oder
- **.dll** (bei .NET - Bibliotheken)

Man übergibt dem Compiler im Allgemeinen *mehrere* Quellcodedateien mit jeweils einer Klassendefinition (siehe Beispiel im Abschnitt 2.2) und erhält als Ergebnis *ein* Assembly. In der Konsolen-Variante unseres Beispiels lassen wir vom Compiler ein Exe-Assembly mit dem Zwischencode der Klassen `Bruch` und `Bruchaddition` erzeugen.

Die folgende Abbildung (nach Mössenböck 2019, S. 7) fasst wesentliche Informationen über Quellcode, C# - Compiler, IL-Code und Assemblies anhand des Bruchadditionsbeispiels zusammen und zeigt mit den anschließend zu beschreibenden Metadaten weitere wichtige Bestandteile eines .NET - Assemblies:



Von der Option, Ressourcen (z. B. Medien) in ein Assembly aufzunehmen, wird im Beispiel kein Gebrauch gemacht.

2.4.1 Typ-Metadaten

Ein Assembly enthält **Typ-Metadaten**, die alle enthaltenen Typen (Klassen und sonstige Datentypen) beschreiben und vom Laufzeitsystem (*Common Language Runtime*, siehe unten) für die Verwaltung der Typen genutzt werden. Zu jeder Klasse sind z. B. Informationen über ihre Felder, Methoden und Eigenschaften vorhanden.

Im `Bruchadditions`-Assembly sind z. B. über die Felder `zaehler` und `nenner` sowie über die `get_Zaehler` - Methode zur `Zaehler`-Eigenschaft der Klasse `Bruch` folgende Typ-Metadaten verfügbar:[1]

Neben **Definitionstabellen** mit Angaben zu den *eigenen* Klassen enthalten die Metadaten auch **Referenztabellen** mit Informationen zu den *fremden* Klassen, die im Assembly benutzt (referenziert) werden.

2.4.2 Manifest mit Assembly-Metadaten

Außerdem enthält ein Assembly das sogenannte **Manifest** mit den **Assembly-Metadaten**. Dazu gehören:

- Name und Version des Assemblies
  Durch eine systematische Versionsverwaltung, die bei DLL-Assemblies besonders relevant ist, sollen im .NET - Framework Probleme mit Versionsunverträglichkeiten vermieden werden, die Windows-Anwender und -Entwickler unter der Bezeichnung *DLL-Hölle* kennen.
- Sicherheitsmerkmale bei signierten Assemblies
  Dazu gehört insbesondere der öffentliche Schlüssel des Herausgebers. Das Signieren ist erforderlich bei Assemblies, die im GAC (***G**lobal **A**ssembly **C**ache*) eines lokalen Rechners abgelegt werden sollen (siehe Abschnitt 2.4.5).
- Informationen über die Abhängigkeit von anderen Assemblies (z. B. aus der BCL)
  Auch hier sorgen exakte Versionsangaben für die Vermeidung von Versionsunverträglichkeiten.

Das Assembly **Bruchaddition.exe** hat noch keine Versionshistorie, ist abhängig vom Assembly **mscorlib.dll** (in der Version 4:0:0:0), und hat keine Sicherheitsmerkmale:[2]

---

[1] Die Typ-Metadaten eines Assemblies erhält man in **ILDasm** über die Tastenkombination **Strg+M**.

[2] Die Assembly-Metadaten erhält man in **ILDasm** per Doppelklick auf **MANIFEST**.

Im .NET *Framework* kann ein Assembly aus *mehreren* Dateien bestehen (siehe Abschnitt 2.4.4), wobei das Manifest nur in *einer* Datei vorhanden ist und die Namen der weiteren zum Assembly gehörenden Dateien enthält. In diesem Fall kann das Manifest auch in einer eigenen Datei untergebracht werden.

2.4.3 Versionsangaben zu Assemblies

Wird bei der Programmausführung von der Laufzeitumgebung (Common Language Runtime, CLR, siehe Abschnitt 2.5) festgestellt, dass ein auszuführendes Assembly (z. B. **Bruchaddition.exe**) laut Manifest ein anderes Assembly (im Beispiel: **mscorlib.dll**) in einer bestimmten Version benötigt, dann wird genau diese Version des externen Assemblies gesucht und ggf. geladen. Dabei sind *alle* Stellen in der Assembly-Version relevant. Würde z. B. das in praktisch jedem .NET-Programm benötigte Assembly **mscorlib.dll** im Rahmen einer .NET Framework - Aktualisierung eine neue Assembly-Version erhalten (z. B. 4.0.0.1), dann müsste **Bruchaddition.exe** neu übersetzt werden, um das neue Assembly nutzen zu können.[1] Um diesen Aufwand zu vermeiden, hat sich folgende Praxis eingebürgert:

- Solange ein verbessertes Assembly abwärts-kompatibel ist, soll die Assembly-Version unverändert bleiben.
- Wenn ein renoviertes Assembly nicht abwärtskompatibel ist, dann muss es eine neue Version erhalten.

Daher hat **mscorelib.dll** in allen .NET Framework - Versionen 4.x die Assembly-Version 4.0.0.0. Analog hatte das zu einem .NET Framework 2.x gehörige Assembly **mscorelib.dll** stets die Version 2.0.0.0.

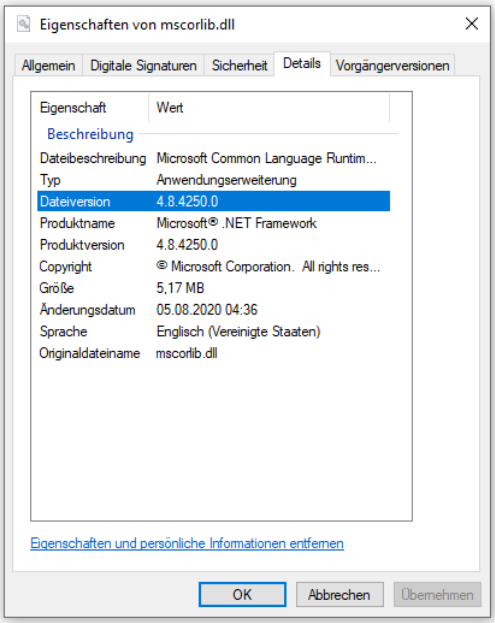
Nun wird aber zum Problem, dass ein .NET - Programm in seinem Assembly-Manifest nicht zum Ausdruck bringen kann, dass es die Datei **mscorlib.dll** z. B. aus dem .NET Framework ab Version 4.7.2 benötigt. Das Problem ist durch eine Anwendungskonfigurationsdatei zu lösen (siehe z. B. Abschnitt 3.3.4.4).

Microsoft hat für Assemblies gleich drei Versionsangaben vorgesehen:

---

[1] Eine (im Kurs nicht behandelte) Alternative wäre das **Assembly Binding Redirection**, siehe https://docs.microsoft.com/en-us/dotnet/framework/configure-apps/how-to-enable-and-disable-automatic-binding-redirection

- Die bisher diskutierte **Assembly-Version** (technisch realisiert über das **AssemblyVersionAttribute**)[1]
  Beim Laden eines referenzierten Assemblies während der Programmausführung interessiert sich die Laufzeitumgebung (CLR) ausschließlich für die Assembly-Version.
- Die **Assembly-Dateiversion** (technisch realisiert über das **AssemblyFileVersionAttribute**)
  Die Assemblies in der .NET Framework - Standardbibliothek oder in anderen Bibliotheken entwickeln sich ständig weiter. U. a. zur Verwendung bei der Anwendungsverteilung ist die Assembly-Dateiversion vorgesehen. Den aktuellen Stand erfährt man z. B. per Windows-Explorer:



  Bei einem Assembly aus der .NET Framework - Standardbibliothek stimmen die beiden ersten Stellen meist mit der Framework-Version überein.
- Die **Assembly-Produktversion** (technisch realisiert über das **AssemblyInformationalVersionAttribute**)
  Der Windows-Explorer zeigt zu einer Assembly-Datei neben der Dateiversion noch die Produktversion an. Sie dient keinen technischen Zwecken, sondern zur Kommunikation bzw. Vermarktung. Oft stimmen Datei- und Produktversion überein.

2.4.4 Multidatei-Assemblies im .NET Framework

Neben dem bisher beschriebenen *Einzeldatei*-Assembly, das in *einer* Binärdatei den Zwischencode und die Metadaten inklusive Manifest enthält, kennt das .NET *Framework* auch das **Multidatei-Assembly**, das Einsteiger gefahrlos ignorieren dürfen. Es besteht aus mehreren **Moduldateien** mit Zwischencode und zugehörigen Typ-Metadaten. Das Manifest des gesamten Assemblies steckt entweder in einer Moduldatei oder in einer separaten Datei. Diese Architektur hat z. B. dann Vorteile, wenn ein Klient über eine langsame Netzverbindung auf ein Assembly zugreift, weil sich der Transport auf die tatsächlich benötigten Module beschränken kann. Ohne die Aufteilung in Module müsste das gesamte Assembly transportiert werden. Für Moduldateien ohne eigenes Manifest wird meist die Namenserweiterung **.netmodule** verwendet.

---

[1] Attribute sind spezielle Klassen, mit denen wir uns im Kapitel 14 beschäftigen werden.

2.4.5 Private und globale Assemblies im .NET Framework

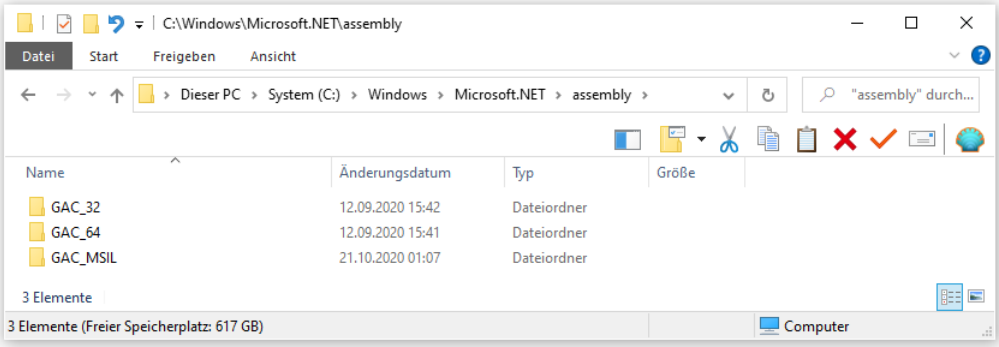
Ein Assembly benötigt in der Regel weitere Assemblies, deren Typen (z. B. Klassen) verwendet werden. Dabei kommen private und systemweit verfügbare Assemblies in Frage. Private Assemblies werden meist im Ordner der Anwendung untergebracht, doch können mit Hilfe der Anwendungskonfigurationsdatei auch andere Ordner verwendet werden.

Im .NET Framework ist für systemweit verfügbare Assemblies der **Global Assembly Cache** (GAC) vorgesehen. Hier ist vor allem die Base Class Library untergebracht, doch können auch andere Assemblies aufgenommen werden, wobei Microsoft eine zurückhaltende GAC-Verwendung empfiehlt.[1]

Seit der Framework-Version 4 enthält der Ordner

C:\Windows\Microsoft.NET\assembly

die zur Ausführung eines .NET - Programms verwendeten GAC-Assemblies:[2]

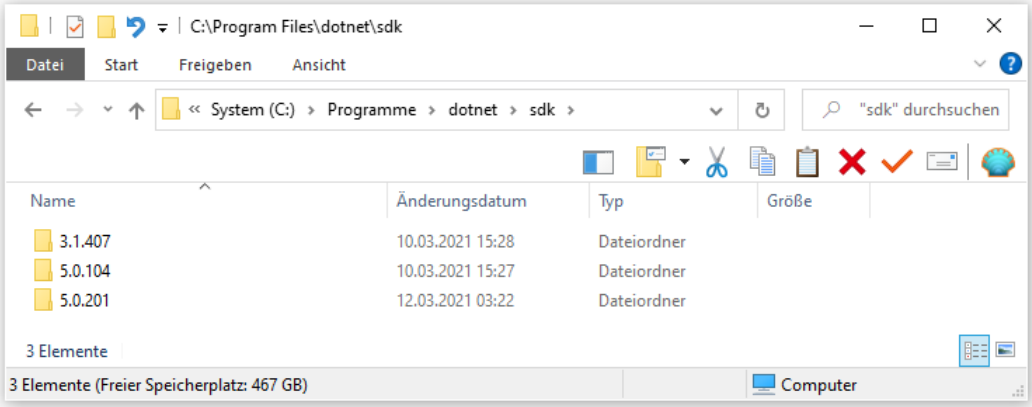


Für ältere Framework-Versionen ist der GAC im Ordner **C:\Windows\assembly** untergebracht.

Eine Besonderheit bei GAC-Assemblies ist die erforderliche Signatur, die für eine global eindeutige Identifikation sorgt, sodass man von der Verpflichtung zu *starken Namen* spricht.

2.4.6 Installationsordner von .NET Core und .NET 5.0 unter Windows

Bei der im Kurs bzw. Manuskript empfohlenen Visual Studio - Installation (siehe Abschnitt 3.1.3) landen die Laufzeitumgebungen von .NET Core 3.1 und .NET 5.0 in den folgenden Ordnern:



Anhand der Ordnernamen lassen sich die installierten Versionen schnell überblicken.

---

[1] https://docs.microsoft.com/en-us/dotnet/framework/app-domains/gac

[2] Der C# - Compiler verwendet einen separaten GAC-Ordner.

2.4.7 Plattformspezifische Assemblies

Manche Assemblies müssen mit herkömmlicher Windows-Software (*unmanaged code*) kooperieren, z. B. durch die Nutzung von Funktionen in den traditionellen Maschinencode-DLLs mit dem Windows-API (***A**pplication **P**rogramming **I**nterface*). Daraus resultiert eine Abhängigkeit von der Prozessorarchitektur, für die solche herkömmliche Windows-Software erstellt wurde. Ein besonders wichtiges Beispiel ist das Bibliotheks-Assembly **mscorlib.dll** aus dem .NET Framework, das elementare und oft benötigte BCL-Klassen implementiert und dabei Dienste des Betriebssystems nutzt. Auf einem Rechner mit 64-bittiger Windowsinstallation werden daher zwei Versionen dieses Assemblies benötigt.

Eine 64-bittige Windows-Version enthält ein 32-Bit - Subsystem namens **WOW64** (*Windows on Windows 64*), sodass 32-Bit-Software in der Regel problemlos in der gewohnten Umgebung ablaufen kann. Dabei herrscht eine strenge Trennung, sodass z. B. ein 64-bittig ausgeführtes Assembly keine DLL oder Komponente mit 32-Bit-Architektur benutzen kann. Stützt sich ein Assembly auf eine solche Software, muss es 32-bittig ausgeführt werden. Die Missachtung dieser Regel wird mit einem Laufzeitfehler bestraft.

Weil sich eine Abhängigkeit von der Systemumgebung nicht immer vermeiden lässt, besitzen Assemblies das Attribut **ProcessorArchitecture** mit den folgenden möglichen Werten (in Klammern die Bezeichnungen für den C# - Compiler):

- Prozessor mit AMD64- oder EM64T-Befehlssatz (x64)
- ARM-Prozessor (Advanced RISC Machine) (arm)
- ARM- Prozessor mit 64-Bit (arm64)
- IA64 (Itanium)
- CIL (anycpu)
  Das Assembly wird bevorzugt in einem 64-Bit - Prozess ausgeführt. Falls dies nicht möglich ist (z. B. auf einer 32-Bit-Installation von Windows), wird das Assembly in einem 32-Bit - Prozess ausgeführt.
- CIL mit 32-Bit-Präferenz (anycpu32bitpreferred)
  Das Assembly wird in einem 32-Bit -Prozess ausgeführt.
- x86-Prozessor (x86)
  Das Assembly wird von einem x86-kompatiblen Prozessor im 32-Bit - Modus ausgeführt.

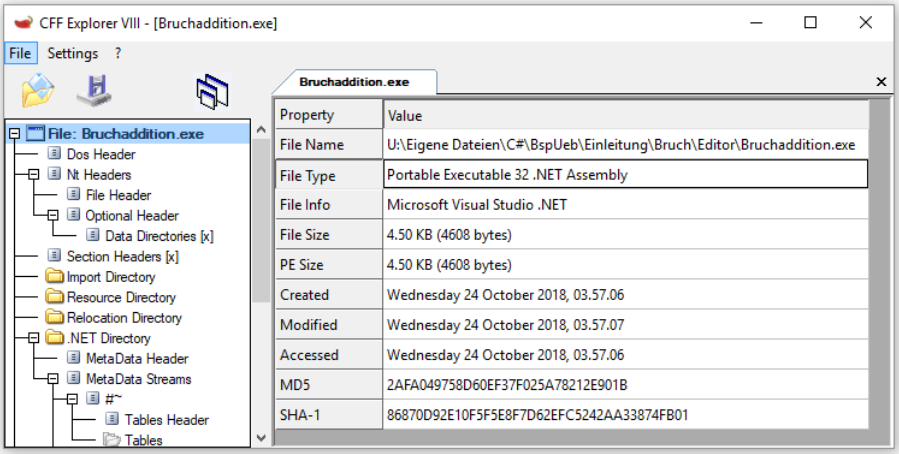
Beim Übersetzen wird das Attribut über eine Option gesetzt, die beim C# - Compiler **platform** heißt und die Voreinstellung **CIL** (**anycpu**) hat, wie der folgende Auszug aus einer (mit `csc -?` angeforderten) Hilfe-Ausgabe des C# - Compilers (vgl. Abschnitt 2.2) zeigt:

```
                        Visual C# Compiler Optionen

   . . .
-platform:<Zeichenfolge>      Schränkt ein, auf welchen Plattformen dieser Code ausgeführt
                              werden kann: x86, Itanium, x64, arm, arm64, anycpu32bitpreferred
                              oder anycpu. Die Standardeinstellung ist "anycpu".
```

Plattform**un**abhängig sind nur die Assemblies mit der **ProcessorArchitecture CIL**.

Unter einer Windows-Version mit 64 Bit produziert der C# - Compiler bei der voreingestellten Prozessorarchitektur **CIL** ein **Portable Executable 32 .NET Assembly**, wie der kostenlos verfügbare **CFF-Explorer** von Daniel Pistelli zeigt:[1]

[1] Verfügbar über die Webseite: https://ntcore.com/?page_id=388

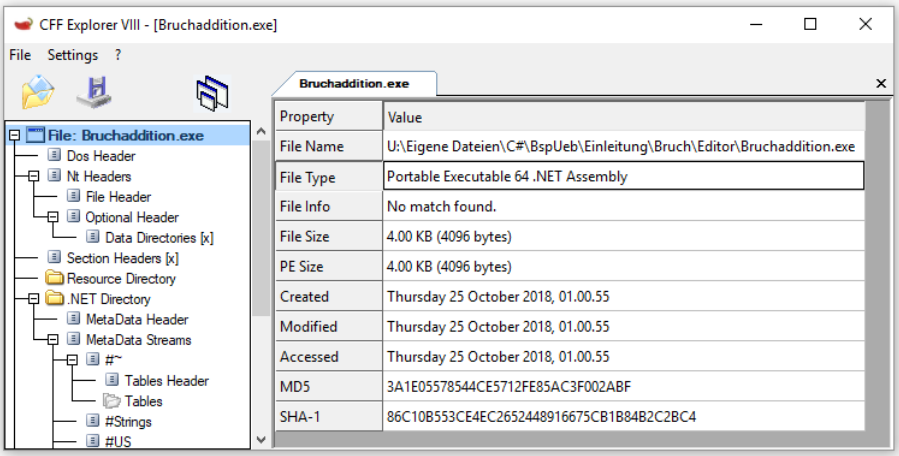


Ein solches Assembly wird unter Windows 64 in einem 64-Bit - Prozess ausgeführt und kann selbstverständlich auch unter Windows 32 genutzt werden.

Wird per Compiler-Option die **ProcessorArchitecture x64** gewählt,

```
>csc *.cs /platform:x64
```

dann resultiert ein **Portable Executable 64 .NET Assembly:**



Es wird auf jeden Fall in einem 64-Bit - Prozess ausgeführt und kann daher unter Windows 32 *nicht* genutzt werden. Eine solche Wahl der Zielplattform ist erforderlich, wenn ein Assembly mit einer Maschinencode-DLL oder mit einer Komponente kooperieren muss, die nur mit 64-Bit-Architektur verfügbar ist.

Wird per Compiler-Option die **ProcessorArchitecture x86** gewählt,

```
>csc *.cs /platform:x86
```

dann resultiert ein Assembly, das unter Windows 64 in einem 32-Bit - Prozess ausgeführt wird. Eine solche Wahl der Zielplattform ist erforderlich, wenn ein Assembly mit einer Maschinencode-DLL oder mit einer Komponente kooperieren muss, die nur mit 32-Bit-Architektur verfügbar ist. Der CFF-Explorer zeigt in diesem Fall **Portable Executable 32 .NET Assembly** als **File Type** an (wie bei der Prozessorarchitektur **CIL**).

2.4.8 Vergleich mit der COM-Technologie

Die Metadaten der .NET -Technologie sorgen dafür, dass die Klassen eines Assemblies unproblematisch von beliebigen .NET - Programmen genutzt werden können. Durch das .NET - Framework soll die COM-Architektur (*Component Object Model*) abgelöst werden, die in den 90er Jahren des

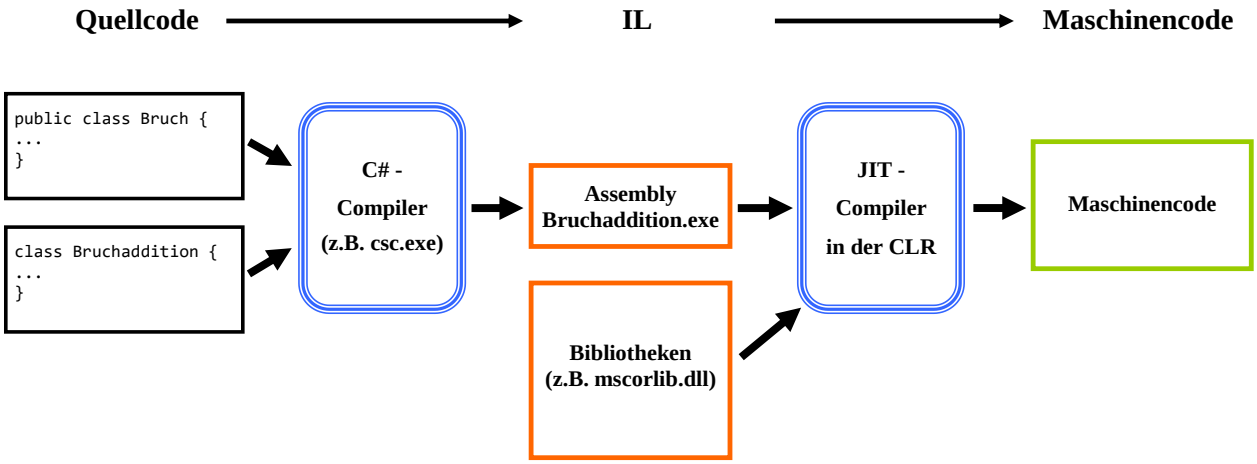
letzten Jahrhunderts geschaffen wurde, um unter Windows die sprachübergreifende Interoperabilität von Software-Komponenten zu ermöglichen. Hier werden ebenfalls Metadaten bereitgestellt, welche die Typen eines COM-Servers beschreiben. Diese Metadaten werden in der IDL (*Interface Definition Language*) erstellt, befinden sich in separaten Binärdateien (Typbibliotheken) und müssen in die Windows-Registry eingetragen werden. Es kann zu Problemen kommen, weil die Metadaten zu einem COM-Server nicht auffindbar oder fehlerhaft sind. Die Metadaten eines .NET - Assemblies sind demgegenüber stets vorhanden und aktuell. Sie bieten außerdem wichtige Informationen (z. B. Version, Sprache, Abhängigkeiten), die in einer COM-Typbibliothek nur spärlich oder gar nicht vorhanden sind. Außerdem ist bei .NET - Metadaten kein Registry-Eintrag erforderlich.

2.5 CLR und JIT-Compiler

Bislang haben Sie erfahren, dass aus dem C# - Quellcode durch einen Compiler (z. B. **csc.exe**) ein Assembly mit Zwischencode (IL) erzeugt wird. Beim Programmstart ist eine weitere Übersetzung in den Maschinencode der aktuellen CPU erforderlich. Diese Aufgabe wird von der Laufzeitumgebung für .NET - Anwendungen, der **CLR** (*Common Language Runtime*) erledigt. Dazu besitzt sie einen JIT-Compiler (***J**ust **I**n **T**ime*), der Leistungseinbußen aufgrund der zweistufigen Übersetzungsprozedur minimiert (z. B. durch das Speichern von mehrfach benötigtem Maschinencode).

.NET - Programme können auf jedem Windows-Rechner mit passender Laufzeitumgebung auf übliche Weise gestartet werden, z. B. per Doppelklick auf eine Datei mit der Namenserweiterung **.exe**.[1]

In der folgenden Abbildung ist der Weg vom Quellcode bis zum ausführbaren Maschinencode für das Bruchadditionsbeispiel dargestellt:



Sorgen um mangelnde Performanz aufgrund der indirekten Übersetzung sind übrigens unbegründet. Eine Untersuchung von Schäpers & Huttary (2003) ergab, dass die Zwischencode-Sprachen C# und Java bei diversen Benchmarks sehr gut mit den „echten Compiler-Sprachen“ C++ und Delphi mithalten können. Ein Grund ist wohl darin zu sehen, dass die meist sehr aktuelle CLR die aktuell vorhandene CPU besser kennt und z. B. Befehlserweiterungen besser ausnutzt als ein Maschinencode-Compiler, der ...

---

[1] Bei einem Programm für das .NET Framework ist eine Laufzeitumgebung mit einer ausreichenden Version mit hoher Wahrscheinlichkeit vorhanden. Bei einem .NET - Programm muss die Laufzeitumgebung nachinstalliert werden.

- in der Regel älter ist als die modernen Prozessoren,
- auf maximale Kompatibilität Wert gelegt und manche CPU-Spezifika nicht unterstützt hat, damit sein Produkt auf möglichst vielen CPUs ablauffähig ist.

Das im .NET - Framework enthaltene Hilfsprogramm **ngen.exe** (*Native Image Generator)* kann aus einem Assembly Maschinencode erzeugen und abspeichern, sodass bei der späteren Programmausführung kein JIT-Compiler mehr benötigt wird.[1] Das führt aber nicht unbedingt zu einer beschleunigten Programmausführung (siehe Richter 2012). Ein wesentlicher, eben schon im Zusammenhang mit den Maschinencode-Compilern behandelter Grund besteht darin, dass **ngen.exe** zurückhaltende Annahmen über die aktuelle Ausführungsumgebung (CLR, CPU) machen muss.

Die CLR hat bei der Verwaltung von .NET - Anwendungen neben der Übersetzung noch weitere Aufgaben zu erfüllen, z. B.:

- **Verifikation des IL-Codes**
  Irreguläre Aktionen einer .NET - Anwendung werden vom Verifier der CLR verhindert. Das macht .NET - Anwendungen sehr stabil.
- **Unterstützung der .NET - Anwendungen bei der Speicherverwaltung**
  Überflüssig gewordene Objekte werden vom Garbage Collector (Müllsammler) der CLR automatisch entsorgt. Mit diesem Thema werden wir uns noch ausführlich beschäftigen.
- **Überwachung von Code mit beschränkten Rechten**
  Bis zur Version 3.5 war im .NET - Framework die CAS-Technik (***C**ode **A**ccess **S**ecurity*) mit einer fein granulierten Rechteverwaltung enthalten. So sollten die Sicherheitsmechanismen des Betriebssystems ergänzt werden, das jedem ausgeführten Programm ebenso viele Rechte einräumt wie dem angemeldeten *Benutzer*. Leider scheiterte die CAS-Technik an der zu hohen Komplexität. Seit dem .NET Framework 4.0 haben Assemblies dieselben Rechte wie native Windows-Programme. Beschränkungen nach dem Sandkasten-Prinzip gelten jedoch für Code, der nicht vollständig vertrauenswürdig ist.[2]

2.6 BCL und Namensräume

Damit Programmierer nicht das Rad (und ähnliche Dinge) neu erfinden müssen, bietet die .NET - Plattform eine umfangreiche Bibliothek mit fertigen Klassen für nahezu alle Routineaufgaben. Dass die als **Base Class Library** (BCL) bezeichnete Standardbibliothek in *allen* .NET - Programmiersprachen zur Verfügung steht, ist für C# - Einsteiger noch wenig relevant.[3] Bei der späteren Teamarbeit kann die sprachunabhängige .NET - Architektur jedoch sehr bedeutsam werden, wenn Anhänger verschiedener Programmiersprachen zusammentreffen.

Bevor man selbst eine Klasse oder Methode entwickelt, sollte man unbedingt die BCL auf die Existenz einer Lösung untersuchen, denn die Lösungen in der BCL ...

- sind leistungsoptimiert und sorgfältig getestet,
- werden ständig weiterentwickelt.

Durch die Verwendung der BCL steigert man in der Regel die Qualität der entstehenden Software, spart viel Zeit und verbessert auch noch die Lesbarkeit des Quellcodes, weil die BCL-Lösungen vielen Entwicklern vertraut sind.

---

[1] In der 64-Bit Variante des .NET Frameworks ab Version 4.0 ist **ngen.exe** in diesem Ordner zu finden:
%SystemRoot%\Microsoft.NET\Framework64\v4.0.30319

[2] Siehe https://docs.microsoft.com/de-de/dotnet/framework/misc/how-to-run-partially-trusted-code-in-a-sandbox

[3] Statt von der ***B**ase **C**lass **L**ibrary* wird im .NET Framework auch von der ***F**ramework **C**lass **L**ibrary* (FCL) gesprochen.

Damit nicht alle Klassen und sonstige, später noch vorzustellende Datentypen im globalen Namensraum koexistieren müssen, was unweigerlich zu Namenskollisionen führen würde, hat man das Konzept der **Namensräume** eingeführt. So dürfen z. B. zwei Klassen denselben Namen tragen, sofern sie sich in verschiedenen Namensräumen befinden. Ihre **voll qualifizierten Namen** (inkl. Namensraumbezeichnung) sind dann verschieden. Die Option der Namensräume wird speziell in der BCL intensiv dazu genutzt, die enorme Zahl von Klassen und sonstigen Datentypen nach funktionaler Verwandtschaft in übersichtliche Gruppen aufzuteilen. Namensräume sind aber keinesfalls auf die BCL beschränkt, und die von C# - Entwicklungsumgebungen angebotenen Vorlagen für neue Projekte (siehe unten) definieren meist per **namespace**-Anweisung einen eigenen Namensraum für jedes Projekt, z. B.:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace MyFirstApp {
    class Program {
        static void Main(string[] args) {
        }
    }
}
```

Bei kleinen Beispielprogrammen sind Namensräume jedoch überflüssig, und wir werden meist der Übersichtlichkeit halber darauf verzichten. Als Konsequenz dieser Vorgehensweise landen Klassen und andere Typen im globalen Namensraum, und die Gefahr von Namenskollisionen steigt. Auch im Bruchrechnungs-Einstiegsbeispiel (siehe Abschnitt 1.2) wurde der Einfachheit halber auf die Definition eines Namensraums verzichtet.

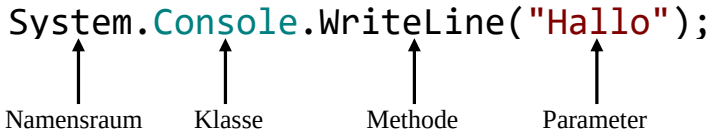
Bei professionellen Projekten sollte man unbedingt Namensräume verwenden und dabei auch auf sinnvolle Namensraumbezeichnungen achten. Microsoft empfiehlt das folgende Schema für die Namensraumbezeichnungen:[1]

<Company>.{<Product>|<Technology>}[.<Feature>][.<Subnamespace>]

Beispiele:

- **Microsoft.Win32**
- **IBM.Data.DB2**

Eine .NET - Klasse muss im Quellcode grundsätzlich mit ihrem voll qualifizierten Namen angesprochen werden, z. B.:



Um in einem Programm die Klassen eines Namensraums vereinfacht (ohne Namensraumpräfix) ansprechen zu können, muss der Namensraum am Anfang des Quelltexts per **using**-Direktive importiert werden, z. B.:[2]

---

[1] https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/names-of-namespaces

[2] Manche Code-Prüfungswerkzeuge empfehlen, bei einer Quellcodedatei mit Namensraumdeklaration die **using**-Direktiven *innerhalb* des Namensraums vorzunehmen, z. B. die Visual Studio - Erweiterung **StyleCop**, die aus der Entwicklungsumgebung über den Menübefehl

Erweiterungen > Erweiterungen verwalten

zu erreichen ist:

```
using System;
  . . .
Console.WriteLine("Hallo");
```

Durch diese **using**-Anweisung wird dafür gesorgt, dass die CLR jedem einfachen und im globalen Namensraum nicht auffindbaren Klassennamen das Präfix **System** voranstellt und dann die referenzierten Assemblies (siehe unten) nach dem vervollständigten Namen durchsucht. Selbstverständlich können auch *mehrere* Namensräume importiert werden, was den Suchaufwand für die CLR erhöht, ohne spürbare Verzögerungen zu bewirken.

Bei Namenskollisionen gewinnt der lokalste Bezeichner. Im folgenden Beispiel werden der Namensraumbezeichner **System** und der Klassenname **Console** (im Namensraum **System**) durch lokale Variablennamen verdeckt:

```
using System;
class Prog {
   static int Console = 13;
   static int System = 1;
   static void Main() {
      Console.WriteLine("Hallo");
   }
}
```

Infolgedessen bewirkt die folgende Zeile

```
Console.WriteLine("Hallo");
```

keinen Methodenaufruf, weil der Compiler `Console` als **int**-Variablennamen betrachtet und meldet, dass der Datentyp **int** keine Definition für den Bezeichner `WriteLine` enthalte:

```
Fehler CS1061 "int" enthält keine Definition für "WriteLine"
```

In der folgenden Zeile

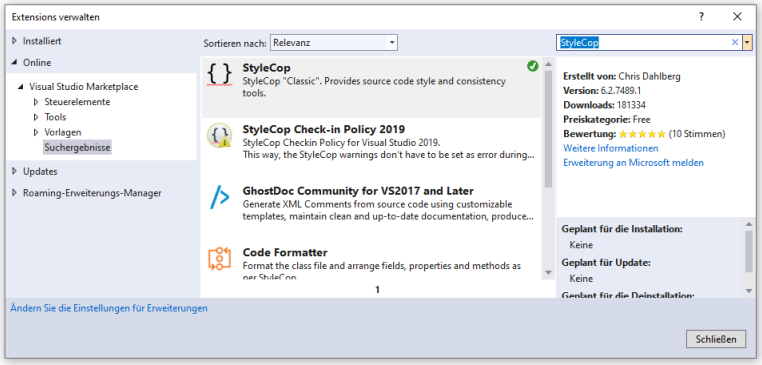
```
System.Console.WriteLine("Hallo");
```

betrachtet der Compiler `System` als **int**-Variablennamen und bemängelt, dass der Datentyp **int** keine Definition für den Bezeichner `Console` enthalte. Mit dem Schlüsselwort **global** und dem **::** - **Operator** kann man eine im globalen Namensraum beginnende Namensauflösung anordnen, und die folgende Zeile führt trotz der Verdeckungen zum erwünschten Methodenaufruf:

```
global::System.Console.WriteLine("Hallo");
```

Sie werden in eigenen Programmen das Verdecken von wichtigen Bezeichnern aus der BCL sicher unterlassen. Wenn komplexe Softwaresysteme unter Beteiligung vieler Programmierer entstehen, sind Namenskollisionen aber nicht auszuschließen. Der von Entwicklungsumgebungen automatisch

---



Während diese Empfehlung für den (zu vermeidenden!) Fall einer Quellcodedatei mit *mehreren* Namensraumdeklarationen sinnvoll ist, sind die Argumente im Fall einer einzelnen Namensraumdeklaration sehr subtil. Wir machen uns daher keine Sorgen bei der Verwendung der Projektvorlagen unserer Entwicklungsumgebung.

erstellte Quellcode enthält daher häufig den **::** - Operator in Verbindung mit dem Schlüsselwort **global**.

Namensräume und Assemblies sind zwei voneinander *unabhängige* Organisationsstrukturen:

- Klassen, die zum selben Namensraum gehören, können in verschiedenen Assemblies implementiert sein.
- In einem Assembly können Klassen aus verschiedenen Namensräumen implementiert werden, was aber nicht zu empfehlen ist.

Namensräume können hierarchisch untergliedert werden, was speziell bei großen Bibliotheken für Ordnung und entsprechend lange vollqualifizierte Namen mit Punkten zwischen den Unterraum-Bezeichnungen sorgt. Die .NET - Standardbibliothek (BCL) verwendet **System** als Wurzelnamensraum und enthält z. B. im Namensraum

System.Windows.Media.Imaging

Klassen und andere Datentypen zur Unterstützung von Grafiken im Bitmap-Format.

Es ist zu beachten, dass durch eine **using**-Direktive für einen Namensraum eventuell vorhandene untergeordnete (eingeschachtelte) Namensräume *nicht* einbezogen werden. Durch

```
using System;
```

wird also z. B. der Namensraum **System.Windows.Media.Imaging** *nicht* importiert.

Einen ersten Eindruck vom Leistungsvermögen der BCL vermittelt die folgende *Auswahl* ihrer Namensräume. Diese Auflistung ist für Programmier*einsteiger* allerdings von begrenztem Wert und sollte bei diesem Leserkreis keine Verunsicherung durch die große Anzahl fremder Begriffe auslösen:

Namensraum	**Inhalt**
System	... enthält grundlegende Basisklassen sowie Klassen für Dienstleistungen wie Konsolenkommunikation oder mathematische Berechnungen. U. a. befindet sich hier die Klasse **Console**, die wir im Einführungsbeispiel für den Zugriff auf Bildschirm und Tastatur verwendet haben.
System.Collections	... enthält Container zum Verwalten von Listen, Warteschlangen, Bitarrays, Hashtabellen etc.
System.Data	... enthält zusammen mit diversen untergeordneten Namensräumen (z. B. **System.Data.SqlClient**) die Klassen zur Datenbankbearbeitung.
System.IO	... enthält Klassen für die Ein-/Ausgabebehandlung im Datenstrom-Paradigma.
System.Net	... enthält Klassen für die Netzwerk-Programmierung.
System.Reflection	... ermöglichst es u. a., zur Laufzeit Informationen über Klassen abzufragen oder neue Methoden zu erzeugen. Dabei werden die Metadaten in den .NET - Assemblies genutzt.
System.Security	... enthält Klassen, die sich z. B. mit Verschlüsselungs-Techniken beschäftigen.
System.Threading	... unterstützt parallele Ausführungsfäden.
System.Web	... unterstützt die Entwicklung von Internet-Anwendungen (inkl. ASP.NET).
System.Windows.Controls	... enthält Klassen für die Steuerelemente einer Windows-Anwendung mit einer grafischen Bedienoberfläche in WPF-Technik (z. B. Befehlsschalter, Textfelder, Menüs).
System.XML	... enthält Klassen für den Umgang mit XML-Dokumenten.

An dieser Stelle sollte vor allem geklärt werden, dass beim Einstieg in die .NET - Programmierung mit C# ...

- einerseits eine **Programmiersprache** mit bestimmter Syntax und Semantik zu erlernen
- und andererseits eine umfangreiche **Klassenbibliothek** zu studieren ist, die im Sinne der im Abschnitt 1.3 geschilderten Vorteile der objektorientierten Programmierung wesentlich an der Funktionalität eines Programms beteiligt ist.

2.7 Zusammenfassung zum Kapitel 2

Als Vorteile der .NET - Technologie für die Software-Entwicklung sind u. a. zu nennen:

- **Sprachintegration**
  Mit C# erstellte Klassen können z. B. auch von VB.NET - Programmierern genutzt werden, sofern bei der Entwicklung auf CLS-Kompatibilität (*Common Language Specification*) geachtet wurde (vgl. Abschnitt 2.3).
- **Portabilität**
  .NET - Anwendungen sind nicht auf Windows beschränkt, sondern (mit gewissen Einschränkungen) auch unter anderen Betriebssystemen einsetzbar. Die Open Source - Projekte **Mono** und .**NET Core** sind auf diesem Weg weit vorangekommen. Das seit Herbst 2020 verfügbare **.NET 5** reduziert die Diversifizierung der .NET - Implementationen. Microsoft verspricht, zusammen mit .NET 6 eine plattformunabhängige grafische Bedienoberfläche namens MAUI (*Multi-platform App UI*) fertigzustellen.
- **Breites Anwendungsspektrum**
  Es kann Software für praktisch jeden Einsatzzweck zur Verwendung auf einem Arbeitsplatzrechner, auf einem Server oder auf einem Smartphone entstehen.

Wir haben im Kapitel 2 u. a. folgende Begriffe kennengelernt:

- **Intermediate Language (IL)**
  .NET - Compiler (z. B. **csc.exe** bei C#) übersetzen den Quellcode in die *Intermediate Language*.
- **Common Language Specification (CLS)**
  Den von allen .NET - Programmiersprachen zu erfüllenden Sprachumfang bezeichnet man als *Common Language Specification* (CLS). Beschränkt man sich bei der Klassendefinition auf diesen kleinsten gemeinsamen Nenner, ist die Interoperabilität mit anderen CLS-kompatiblen Klassen sichergestellt.
- **Assembly**
  Beim Übersetzen von (im Allgemeinen mehreren) Quellcodedateien entsteht ein Assembly, das die kleinste Einheit von .NET - Software ist hinsichtlich:
  - Weitergabe
  - Versionierung
  - Sicherheitsverwaltung

  Ein Assembly kann beliebig viele Klassen implementieren. In einem ausführbaren Programm (mit der Dateinamenserweiterung **.exe**) ist eine Startklasse (mit Methode **Main()**) vorhanden. Bei einem Bibliotheks-Assembly endet der Dateiname mit der Erweiterung **.dll**.
- **Metadaten**
  Ein Assembly enthält neben dem IL-Code auch Metadaten. Die *Typ-Metadaten* enthalten eine Beschreibung der implementierten und der referenzierten Typen. Im Manifest sind die *Assembly-Metadaten* mit Angaben zur Version, zur Sicherheit und zur Abhängigkeit von anderen Assemblies enthalten.

- **Common Language Runtime (CLR) mit Just-In-Time (JIT) - Compiler**
  Die Ausführungsumgebung für IL-Code, der auch als *managed code* bezeichnet wird, besitzt einen Just-In-Time - Compiler zur Übersetzung von IL-Code in nativen Maschinencode. Außerdem kümmert sich die CLR um Stabilität (Code-Verifikation), Speicherverwaltung (Garbage Collection) und Überwachung von Code mit beschränkten Rechten.
- **Namensräume**
  Indem man eine Klasse in einen Namensraum einfügt, ergänzt man ihren Namen um ein Präfix und vermeidet Namenskollisionen. Man fasst in der Regel funktional verwandte Klassen in einen gemeinsamen Namensraum zusammen, der nach Bedarf hierarchisch in Unterräume aufgeteilt werden kann.
- **Base Class Library (BCL)**
  In der voluminösen und universellen Standardbibliothek der .NET - Plattform wird von Namensräumen reichlich Gebrauch gemacht.

2.8 Übungsaufgaben zum Kapitel 2

1) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. In C# kann man nur Software für Windows entwickeln.
2. Die .NET - Standardbibliothek wurde überwiegend in C# programmiert.
3. Durch .NET 5.0 sollen das .NET Framework und .NET Core zusammengeführt werden.
4. Die Klassen in einem mit C# erstellten DLL-Assembly können auch in anderen .NET - Programmiersprachen (z. B. VB.NET) genutzt werden.

2) Welche Aufgaben erfüllt die Common Language Runtime (CLR)?

3) In welcher Beziehung stehen Assemblies und Namensräume?

4) Was bedeuten die Abkürzungen IL, BCL, CLS, COM?

3 Werkzeuge zum Entwickeln von C# - Programmen

In diesem Kapitel werden kostenlos verfügbare Werkzeuge zum Entwickeln von .NET - Anwendungen in der Programmiersprache C# beschrieben. Zunächst beschränken wir uns puristisch auf einen simplen Texteditor zum Erstellen des Quellcodes und ein Konsolenfenster für den direkten Aufruf eines C# - Compilers, der durch Parameter des Startkommandos über Auftragsdetails informiert wird (z. B. über die Namen der zu übersetzenden Quellcode-Dateien). In dieser sehr übersichtlichen „Entwicklungsumgebung“ werden die grundsätzlichen Arbeitsschritte und einige Randbedingungen besonders deutlich.

Wir werden nicht irgendeinen C# - Compiler verwenden, sondern den zusammen mit dem Visual Studio 2019 in den folgenden Ordner

C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\MSBuild\Current\Bin\Roslyn

installierten Compiler aus dem **Roslyn-Projekt**, der auch bei der C# - Entwicklung mit dem Visual Studio im Hintergrund für die Übersetzung von Quellcode in die IL zuständig ist.[1] Daher gehen wir folgendermaßen vor:

- Wir installieren das Visual Studio Community 2019.
- Jedoch verwenden wir zunächst nur den darin enthaltenen Compiler, um die prinzipiellen Arbeitsschritte bei der Software-Entwicklung kennenzulernen. Weil das im .NET Framework besonders leicht möglich ist, verwenden wir diese .NET - Implementation.
- Im weiteren Verlauf des Kurses kommt dann das Visual Studio als bequeme und leistungsfähige Entwicklungsumgebung zum Einsatz.

3.1 Microsoft Visual Studio Community 2019

Im November 2020 ist die Visual Studio 2019 Community 16.8.x verfügbar (inkl. Support für .NET 5.0). Wenn Sie diesen Text lesen, ist mit Sicherheit eine höhere Version aktuell. Erfahrungsgemäß bleiben aber die im Manuskript enthaltenen Erläuterungen zur Entwicklungsumgebung größtenteils gültig.

3.1.1 Voraussetzungen

Die Systemvoraussetzungen sind sehr bescheiden:[2]

- Windows 8.1 oder 10 sowie die korrespondierenden Versionen von Windows Server
- Prozessor mit 1,8 GHz (mindestens zwei Prozessorkerne empfohlen)
- 2 GB RAM (8 GB empfohlen)
- Für unsere Zwecke genügen ca. 5 GB Massenspeicher (auf der SSD oder Festplatte)

---

[1] https://github.com/dotnet/roslyn
Das .NET Framework enthält (unter Win-x64 im Ordner **C:\Windows\Microsoft.NET\Framework64\v4.0.30319**) ebenfalls einen C# - Compiler namens **csc.exe**, der aber nur den C# - Sprachumfang bis zur Version 5 unterstützt und daher im Kurs nicht verwendet wird. Für die Beispiele im Abschnitt 3.2 würde dieser Compiler reichen, z. B.:

```
U:\Eigene Dateien\C#\Kurs\Hallo\Editor>C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc *.cs
U:\Eigene Dateien\C#\BspUeb\Einleitung\Bruch\Editor>C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc
*.cs
Microsoft (R) Visual C# Compiler version 4.8.4084.0 for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.

This compiler is provided as part of the Microsoft (R) .NET Framework, but only supports language ver-
sions up to C# 5, which is no longer the latest version. For compilers that support newer versions of
the C# programming language, see http://go.microsoft.com/fwlink/?LinkID=533240
```

[2] https://docs.microsoft.com/de-de/visualstudio/releases/2019/system-requirements#visual-studio-2019-system-requirements

3.1.2 Bezugsquelle

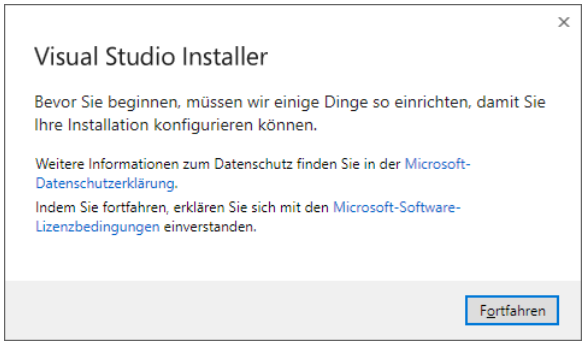
Auf der Webseite

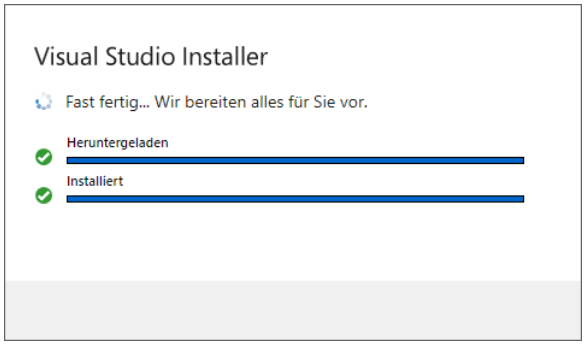
https://visualstudio.microsoft.com/downloads/

kann man den kostenlosen Download eines Web-Installers zu Visual Studio Community 2019 anfordern, der die benötigten Dateien während der Installation aus dem Internet bezieht.

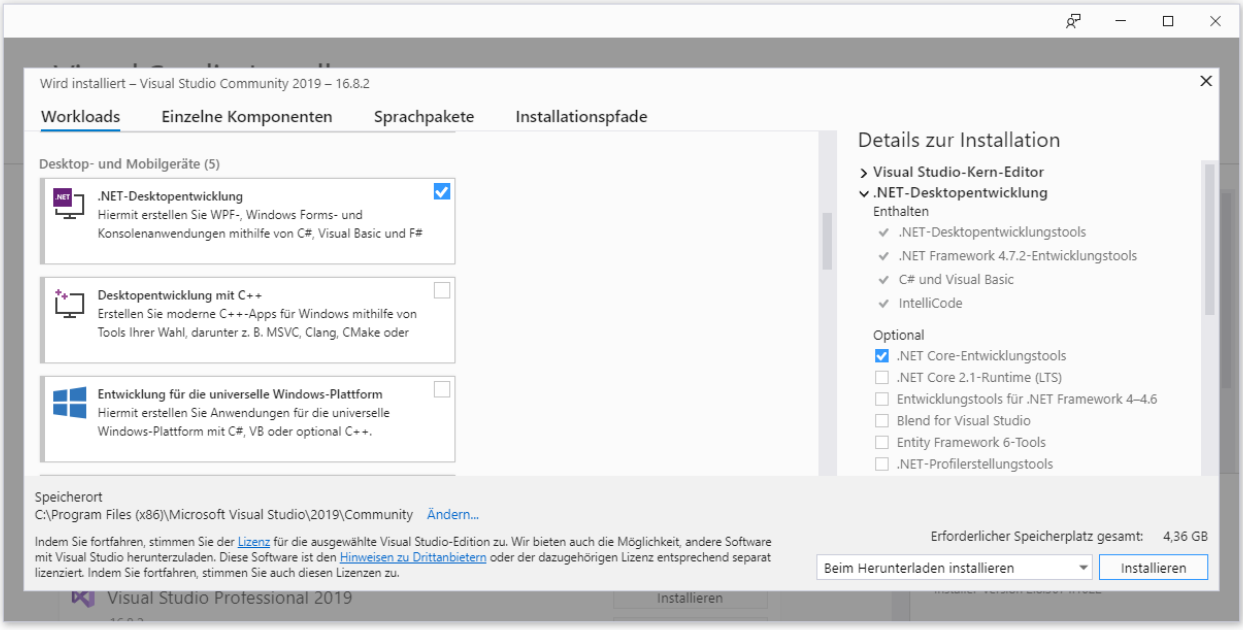
3.1.3 Installation

Der Web-Installer (z. B. **vs_community__1896031153.1477601941.exe**) führt nach dem Start einige Vorbereitungen aus, die wenige Minuten in Anspruch nehmen:





Anschließend erscheint der **Visual Studio - Installer** mit dem folgenden Dialog zur Wahl der **Workloads** (geplanten Einsatzfelder):



Wir beschränken uns auf die **.NET - Desktopentwicklung** und verzichten auch auf die optionalen Installationsdetails mit den folgenden Ausnahmen:

- **.NET Core-Entwicklungstools**
  Damit ist im Visual Studio ab Version 16.8 für die Unterstützung von .NET 5 und .NET Core gesorgt.
- **.NET Framework 4.8-Entwicklungstools**
  Damit kann die höchste Version des in Windows integrierten .NET Frameworks unterstützt werden.

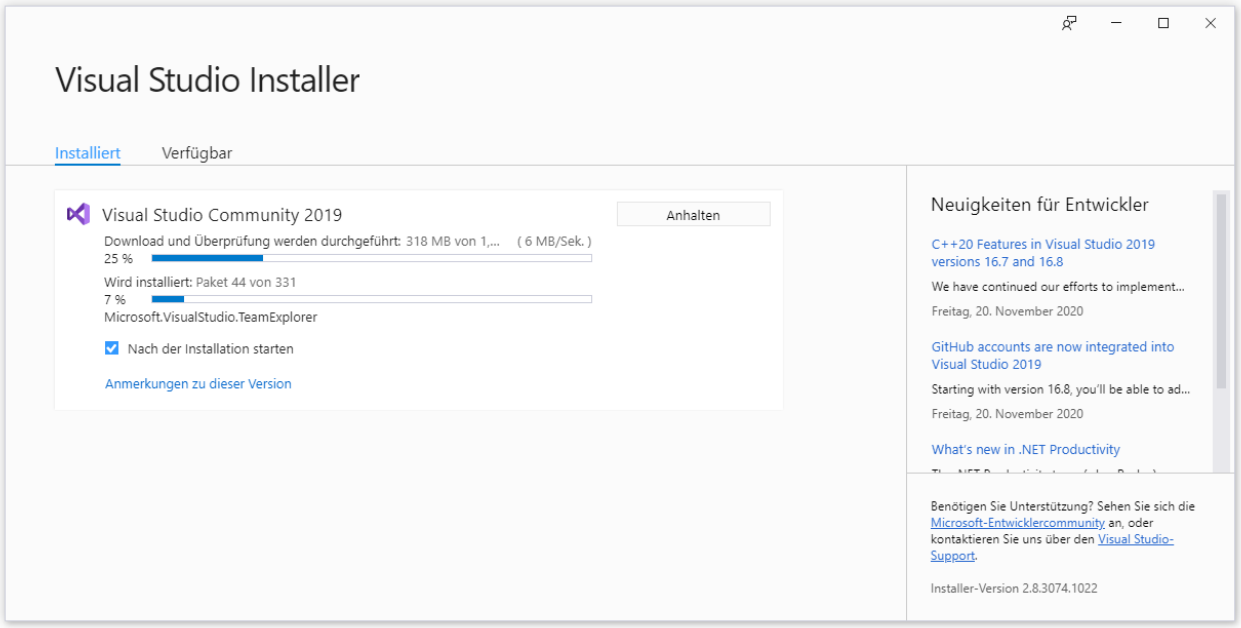
Für den geplanten Installationsumfang kündigt der Installer einen Speicherplatzbedarf von 4,36 GB an.

Bei Bedarf kann der **Visual Studio - Installer** später (über einen Link im Windows-Startmenü) erneut gestartet werden, um den Installationsumfang zu erweitern (siehe Abschnitt 3.3.2).

Auf einem Rechner mit ...

- Windows 10 (64 Bit), Version 2004
- einer Intel-CPU Core i3 aus dem Jahr 2010
- 8 GB RAM
- und einer SSD als Massenspeicher

hat die Installation ca. 5 Minuten gedauert:



Die Installation erfolgt in die Ordner:

C:\Program Files (x86)\Microsoft Visual Studio\2019\Community
C:\Program Files (x86)\Microsoft SDKs

3.2 Entwicklung mit Texteditor und Kommandozeilen-Compiler

Wie angekündigt verwenden wir zunächst nicht die Hauptfunktion der eben installierten Entwicklungsumgebung, sondern nur den enthaltenen Compiler, um die prinzipiellen Arbeitsschritte bei der Software-Entwicklung kennenzulernen. Weil das im .NET Framework besonders leicht möglich ist, verwenden wir diese .NET - Implementation. Damit nehmen wir keine nennenswerte Einschränkung in Kauf, sondern nutzen (wie schon im Kapitel 2) den didaktischen Vorteil, C# - Quellcode durch einen Compiler-Aufruf direkt (ohne den Extraaufwand einer Projektdefinitionsdatei) in IL-Code übersetzen zu können.

3.2.1 Editieren

Grundsätzlich kann man zum Erstellen einer Quellcode-Datei einen beliebigen Texteditor verwenden, z. B. das im Windows-Zubehör enthaltene Programm **Notepad** (alias **Editor**). Um das Erstellen, Kompilieren und Ausführen von C# - Programmen ohne großen Aufwand üben zu können, erstellen wir das unvermeidliche **Hallo**-Programm:

```
using System;

class Hallo {
   static void Main() {
      Console.WriteLine("Hallo, echt .NET hier!");
   }
}
```

Im Einleitungsbeispiel (siehe Kapitel 1) wurde einiger Aufwand in Kauf genommen, um einen halbwegs realistischen Eindruck von objektorientierter Programmierung (OOP) zu vermitteln. Das `Hallo`-Beispiel ist zwar angenehm einfach aufgebaut, kann aber als „pseudo-objektorientiert“ (POO) kritisiert werden. Es ist eine einzige Klasse namens `Hallo` mit einer einzigen Methode namens **Main()** vorhanden. Beim Programmstart wird die Klasse `Hallo` von der CLR aufgefordert, ihre **Main()** - Methode auszuführen. Trotz Klassendefinition haben wir es praktisch mit einer Prozedur historischer Bauart zu tun, was für den einfachen Zweck des Programms durchaus angemessen ist. In den Kapiteln 3 und 4 werden wir solche pseudo-objektorientierten (POO-) Programme benutzen, um elementare Sprachelemente in möglichst einfacher Umgebung kennenzulernen. Aus den letzten Ausführungen ergibt sich, dass C# zwar eine objektorientierte Programmierweise nahelegen und unterstützen, aber nicht erzwingen kann.

Das Hallo-Programm eignet sich aufgrund seiner Kürze zum Erläutern wichtiger Regeln, an die Sie sich so langsam gewöhnen sollten. Alle in der folgenden Auflistung enthaltenen Themen werden aber später noch einmal systematisch und ausführlich behandelt:

- In der ersten Zeile wird per **using**-Direktive der Namensraum **System** importiert, damit die dort enthaltene Klasse **Console** im Programm ohne Namensraumpräfix angesprochen werden kann (vgl. Abschnitt 2.6). Diese für C# - Programme typische Vorgehensweise soll auch im `Hallo`-Beispiel vorgeführt werden, obwohl sie hier den Schreibaufwand sogar vergrößert.
- Nach dem Schlüsselwort **class** folgt der weitgehend frei wählbare Klassenname.[1] Hier ist wie bei allen Bezeichnern zu beachten, dass C# zwischen Groß- und Kleinbuchstaben unterscheidet.
- Dem Kopf der Klassendefinition (bestehend aus dem Schlüsselwort **class** und dem Namen der Klasse) folgt der mit geschweiften Klammern eingerahmte Rumpf.
- Weil die `Hallo`-Klasse startfähig sein soll, muss sie eine Methode namens **Main()** besitzen. Diese wird beim Programmstart automatisch aufgerufen und dient bei „echten“ OOP-Programmen oft dazu, Objekte (direkt oder indirekt) zu erzeugen.

[1] Ein paar Restriktionen gibt es beim Klassennamen schon. Z. B. sind die reservierten Wörter der Programmiersprache C# verboten. Nähere Informationen folgen später.

- Die Definition der Methode **Main()** wird von zwei Schlüsselwörtern eingeleitet, deren Bedeutung für Neugierige hier schon beschrieben wird:[1]
    - **static**
      Mit diesem Modifikator wird **Main()** als **Klassenmethode** gekennzeichnet. Im Unterschied zu den *Instanz*methoden der *Objekte* gehören die Klassenmethoden, oft auch als *statische Methoden* bezeichnet, zur *Klasse* und können ohne vorherige Objektkreation ausgeführt werden (vgl. Abschnitt 1.2). Die beim Programmstart automatisch auszuführende **Main()** - Methode der Startklasse muss auf jeden Fall durch den Modifikator **static** als Klassenmethode gekennzeichnet werden. In einem objektorientierten Programm hat sie oft die Aufgabe, die ersten Objekte zu erzeugen (siehe unsere Klasse `Bruchaddition` auf Seite 11).
    - **void**
      Im Beispiel wird für die Methode **Main()** der Rückgabetyp **void** verwendet, weil die Methode keinen Rückgabewert liefert. Mit den Rückgabewerten von Methoden werden wir uns noch gründlich beschäftigen.
- In der **Parameterliste** einer Methode kann die gewünschte Arbeitsweise näher spezifiziert werden. Hinter dem Methodennamen muss auf jeden Fall eine durch runde Klammern eingerahmte Parameterliste angegeben werden, gegebenenfalls - wie bei der Methode **Main()** - eine leere.
- Dem Kopf einer Methodendefinition folgt der mit geschweiften Klammern eingerahmte Rumpf mit Variablendeklarationen und sonstigen Anweisungen.

---

[1] Diese Mega-Fußnote sollte nur lesen, wer im Hallo-Beispielprogramm (z. B. aufgrund von Erfahrungen mit anderen C# - Beschreibungen) den Modifikator **public** am Anfang der Klassen- und der Methodendefinition vermisst:

Die Methode **Main()** wird beim Programmstart von der CLR aufgerufen. Weil es sich bei der CLR aus Sicht des Programms um einen *externen* Akteur handelt, liegt es nahe, die Methode **Main()** explizit über den Modifikator **public** für die Öffentlichkeit freizugeben. Generell ist nämlich in C# eine Methode (oder ein Feld) **private** und folglich nur innerhalb der Klasse verfügbar. In der Tat findet man in der Literatur viele Hallo-Beispielprogramme (z. B. bei Gunnerson 2002, Louis et al. 2002, Mössenböck 2019) mit dem Modifikator **public** im Kopf der **Main()** - Definition, z. B.:

```
using System;
class Hallo {
    public static void Main() {
        Console.WriteLine("Hallo, echt .NET hier!");
    }
}
```

Allerdings wird **Main()** grundsätzlich *nur* von der CLR aufgerufen, die eben *nicht* wie eine fremde Klasse eingeordnet wird. Laut C# - Sprachdefinition (ECMA 2017) ist für die **Main()** - Methode nur der Modifikator **static** vorgeschrieben, und demgemäß erweist sich der Modifikator **public** in der Praxis als überflüssig.

In den Hallo-Beispielprogrammen einiger Autoren (z. B. Richter 2012) wird nicht nur die Methode **Main()**, sondern auch die *Klasse* als **public** definiert, z. B.:

```
using System;
public class Hallo {
    public static void Main() {
        Console.WriteLine("Hallo, echt .NET hier!");
    }
}
```

Dies ist *nicht* erforderlich, weil in C# eine (nicht eingeschachtelte) Klasse per Voreinstellung die Schutzstufe **internal** (siehe unten) besitzt und folglich im gesamten Assembly bekannt ist, in das sie vom Compiler einbezogen wird (siehe unten). Außerdem wird die einzige Klasse der Hallo-Beispielprogramme von keiner einzigen anderen Klasse gesucht und benutzt.

Die im aktuellen Manuskript (aber z. B. auch von Albahari & Johannsen 2020) bevorzugte Variante, im Hallo-Beispiel und in vergleichbaren Programmen auf den **public**-Modifikator komplett zu verzichten, kann folgendermaßen begründet werden:

- Konformität mit der ECMA-Sprachbeschreibung (siehe ECMA 2017)
- Verzicht auf überflüssige, unzureichend begründete Forderungen

- In der **Main()** - Methode unserer `Hallo`-Klasse wird die (statische) **WriteLine()** - Methode der Klasse **Console** dazu benutzt, einen Text an die Standardausgabe zu senden. Zwischen dem Klassen- und dem Methodennamen steht ein Punkt.
- Während unsere **Main()** - Methodendefinition mit einer leeren Parameterliste auskommt, benötigt der im Methodenrumpf enthaltene **Aufruf** der (statischen) Methode **Console.WriteLine()** einen Aktualparameter, damit der gewünschte Effekt erzielt wird. Wir geben eine durch doppelte Anführungszeichen begrenzte Zeichenfolge an, die auf der Konsole erscheinen soll.
- Bei einem Methodenaufruf handelt es sich um eine **Anweisung**, die in C# mit einem Semikolon abzuschließen ist.

Es dient der Übersichtlichkeit, zusammengehörige Programmteile durch eine gemeinsame Einrücktiefe zu kennzeichnen. Man realisiert die Einrückungen am einfachsten mit der Tabulatortaste, aber auch Leerzeichen sind erlaubt. Für den Compiler sind die Einrückungen irrelevant.

Speichen Sie Ihr Quellprogramm unter dem Namen **Hallo.cs** in einem geeigneten Verzeichnis, z. B. in der Datei

U:\Eigene Dateien\C#\Kurs\Hallo\Editor\Hallo.cs

Im Unterschied zur Programmiersprache Java müssen in C# der Klassen- und der Dateiname *nicht* übereinstimmen, zwecks Übersichtlichkeit sollten sie es in der Regel aber doch tun.

3.2.2 Übersetzen in die IL

Unsere Entwicklungsumgebung Visual Studio 2019 verwendet einen C# - Compiler aus dem **Roslyn-Projekt**.[1] Eine Kommandozeilenversion dieses Compilers mit dem Namen **csc.exe** findet sich nach der (im Abschnitt 3.1 beschriebenen) Installation der Entwicklungsumgebung auf einem Windows-Rechner mit 64-Bit - Architektur im folgenden Ordner:

C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\MSBuild\Current\Bin\Roslyn

Öffnen Sie ein Konsolenfenster, wechseln Sie in das Verzeichnis mit der eben erstellten Quellcodedatei **Hallo.cs**, und lassen Sie das Programm vom C# - Compiler **csc.exe** übersetzen:

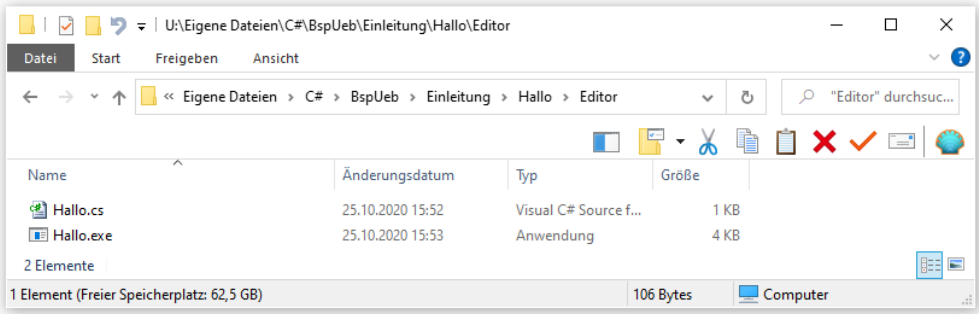
Beim Compiler-Aufruf den Namen der Datei mit der Klassendefinition groß zu schreiben, ist nicht erforderlich, weil Windows bekanntlich bei Datennamen *nicht* zwischen Groß- und Kleinschreibung unterscheidet. Allerdings ist die Schreibweise des Quellcode-Dateinamens im Compiler-Aufruf doch nicht ganz irrelevant, weil sie für den Assembly-Dateinamen übernommen wird.

Wer längerfristig per Texteditor und Kommandozeilen-Compiler programmieren möchte, sollte den Ordner mit **csc.exe** in die Definition der Umgebungsvariablen PATH aufnehmen.

Falls beim Übersetzen keine Probleme auftreten (siehe Abschnitt 3.2.4), meldet sich der Rechner nach kurzer Tätigkeit mit einer neuen Kommando-Aufforderung zurück, und die Quellcodedatei **Hallo.cs** erhält Gesellschaft durch die Assembly-Datei **Hallo.exe**, z. B.:

---

[1] https://github.com/dotnet/roslyn



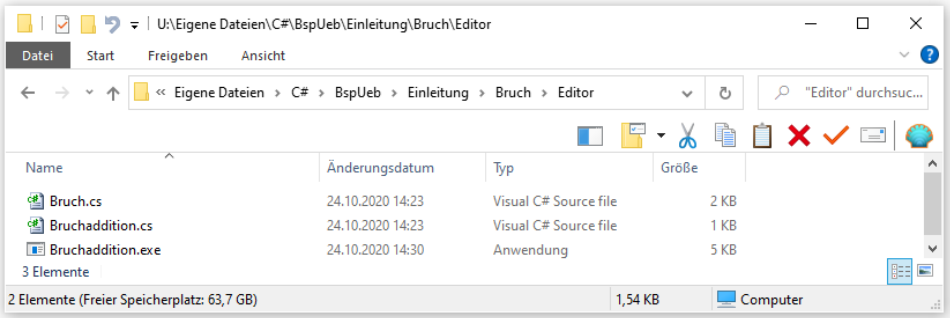
Sind *mehrere* Quellcodedateien in ein Assembly zu übersetzen, gibt man sie beim **csc**-Aufruf hintereinander an, z. B.

```
>csc  Bruch.cs BruchAddition.cs
```

Mit Hilfe von Jokerzeichen lässt sich der Schreibaufwand reduzieren, z. B.:

```
>csc  *.cs
```

Das entstehende Assembly erbt seinen Namen von der Quellcodedatei mit der Startklasse:



Über die Befehlszeilenoption **out** kann der Assembly-Name aber auch frei gewählt werden, z. B.

```
>csc  -out:ba.exe  *.cs
```

Die im weiteren Verlauf des aktuellen Abschnitts noch folgenden Erläuterungen zu den Compiler-Optionen sind für die Software-Entwicklung durchaus relevant. Allerdings müssen wir uns nicht mit syntaktischen Details des **csc**-Kommandos befassen, sondern können auf die Unterstützung durch die Entwicklungsumgebung vertrauen (siehe Abschnitt 3.3.6):

- Die Entwicklungsumgebung erstellt im Hintergrund passende Übersetzungskommandos.
- Oft sind die voreingestellten Compiler-Optionen akzeptabel.
- Sind Anpassungen erforderlich, kann dies meist in einem bequemen Dialogfenster geschehen (siehe Abschnitt 3.3.6).

Über die Befehlszeilenoption **reference** (Kurzform: **r**) wird dem Compiler eine Liste von Assemblies bekannt gegeben, welche die im zu übersetzenden Quellcode verwendeten (referenzierten) Klassen implementieren, z. B.:[1]

```
>csc  -reference:mylib.dll  Prog.cs
```

Zwei Referenz-Assemblies sind durch ein Semikolon zu trennen. Bei Verwendung einer *relativen* Pfadangabe sucht der Compiler in der folgenden Reihenfolge an mehreren Orten nach den referenzierten Dateien:

---

[1] Das im Beispiel auftauchende Assembly **PresentationFramework.dll** liegt im CLR-Unterordner **WPF**, der explizit angegeben werden muss.

- im aktuellen Verzeichnis
- in einem per **lib** - Compiler-Option benannten Ordner

Zusätzlich zu den explizit referenzierten Assemblies wird generell das Bibliotheks-Assembly **mscorlib.dll** durchsucht, das elementare und oft benötigte BCL-Klassen implementiert.

Der C# - Compiler **csc.exe** liest per Voreinstellung Referenzen und sonstige Optionen aus einer **Response-Datei**, wenn dies nicht per `noconfig`-Option verhindert wird. Diese Datei muss sich im selben Verzeichnis befinden wie **csc.exe** und den Namen **csc.rsp** tragen. Im Verzeichnis

C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\MSBuild\Current\Bin\Roslyn

befindet sich eine Response-Datei mit dem folgenden Inhalt:

```
# Licensed to the .NET Foundation under one or more agreements.
# The .NET Foundation licenses this file to you under the MIT license.
# See the LICENSE file in the project root for more information.

# This file contains command-line options that the C#
# command line compiler (CSC) will process as part
# of every compilation, unless the "/noconfig" option
# is specified.

# Reference the common Framework libraries
/r:Accessibility.dll
/r:Microsoft.CSharp.dll
/r:System.Configuration.dll
/r:System.Configuration.Install.dll
/r:System.Core.dll
/r:System.Data.dll
/r:System.Data.DataSetExtensions.dll
/r:System.Data.Linq.dll
/r:System.Data.OracleClient.dll
/r:System.Deployment.dll
/r:System.Design.dll
/r:System.DirectoryServices.dll
/r:System.dll
/r:System.Drawing.Design.dll
/r:System.Drawing.dll
/r:System.EnterpriseServices.dll
/r:System.Management.dll
/r:System.Messaging.dll
/r:System.Runtime.Remoting.dll
/r:System.Runtime.Serialization.dll
/r:System.Runtime.Serialization.Formatters.Soap.dll
/r:System.Security.dll
/r:System.ServiceModel.dll
/r:System.ServiceModel.Web.dll
/r:System.ServiceProcess.dll
/r:System.Transactions.dll
/r:System.Web.dll
/r:System.Web.Extensions.Design.dll
/r:System.Web.Extensions.dll
/r:System.Web.Mobile.dll
/r:System.Web.RegularExpressions.dll
/r:System.Web.Services.dll
/r:System.Windows.Forms.dll
/r:System.Workflow.Activities.dll
/r:System.Workflow.ComponentModel.dll
/r:System.Workflow.Runtime.dll
/r:System.Xml.dll
/r:System.Xml.Linq.dll
```

Wie an den **csc.rsp** - Einträgen zu sehen ist, kann die Befehlszeilenoption **reference** durch ihren Anfangsbuchstaben abgekürzt werden.

Mit der Befehlszeilenoption

```
-noconfig
```

verhindert man die Auswertung der voreingestellten Antwortdatei. Folglich wird (zeitsparend!) nur noch das zentrale Bibliotheks-Assembly **mscorlib.dll** automatisch durchsucht. Entwicklungsumge-

bungen bieten zur Verwaltung der in einem Projekt benötigten Referenzen bequeme Bedienelemente (siehe Abschnitt 3.3.6.1).

Mit der Befehlszeilenoption **target** kann ein **Ausgabetyp** gewählt werden:

- **exe**: ausführbares Konsolenprogramm
  Dies ist die Voreinstellung und musste daher in obigen Beispielen nicht angegeben werden.
  Beispiel:
  ```
  >csc  -target:exe  Hallo.cs
  ```
- **winexe**: ausführbares Windowsprogramm
  Im Unterschied zum Typ **exe** wird kein Konsolenfenster angezeigt, was im `Hallo`-Beispiel zu einem sinnlosen Programm ohne jeglichen Bildschirmauftritt führen würde. Im folgenden Beispiel wird ein WPF-Programm übersetzt, wobei einige zusätzliche DLL-Assemblies referenziert werden müssen:
  ```
  >csc -target:winexe
   /r:C:\Windows\Microsoft.NET\Framework64\v4.0.30319\WPF\PresentationCore.dll
   /r:C:\Windows\Microsoft.NET\Framework64\v4.0.30319\WPF\PresentationFramework.dll
   /r:C:\Windows\Microsoft.NET\Framework64\v4.0.30319\WPF\WindowsBase.dll WinExeDemo.cs
  ```
- **library**: DLL-Assembly
  Die resultierende Bibliothek kann analog zum Assembly **mscorlib.dll** der .NET - Standardbibliothek von anderen Assemblies genutzt werden.
  Beispiel:
  ```
  >csc  -target:library  Simput.cs
  ```
- **module**: Teil eines Multidatei-Assemblies (vgl. Abschnitt 2.2)

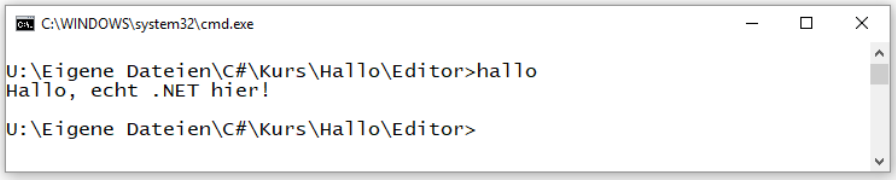
Die Compiler-Option **platform**, mit der die Abhängigkeit eines Assemblies von einer Prozessor-Architektur deklariert werden kann, wurde schon im Abschnitt 2.4.7 erwähnt. Durch Verzicht auf diese Option verwendet man die voreingestellte Plattform **CIL** (bzw. **anycpu**). Unter Windows 64 produziert der C# - Compiler daraufhin ein **Portable Executable 32 .NET Assembly**. Es wird unter Windows 64 in einem 64-Bit - Prozess ausgeführt und läuft selbstverständlich auch unter Windows 32.

Über weitere Befehlszeilenoptionen informiert der Compiler beim folgenden Aufruf

```
>csc -?
```

3.2.3 Ausführen

.NET - Framework - Programme können auf jedem Windows-Rechner mit passendem Framework auf übliche Weise gestartet werden, z. B. per Doppelklick auf den im Windows-Explorer angezeigten Dateinamen. Das Hallo-Programm startet man am besten im Konsolenfenster durch Abschicken seines Namens, z. B.:



Trotz der strengen Unterscheidung zwischen Groß- und Kleinbuchstaben im C# - Quellcode und trotz unserer Entscheidung für einen *großen* Anfangsbuchstaben im Klassennamen `Hallo`, ist auf der Ebene des Windows-Dateisystems, also z. B. beim Starten eines C# - Programms, die Groß-/Kleinschreibung irrelevant.

3.2.4 Programmfehler beheben

Die vielfältigen Fehler, die wir mit naturgesetzlicher Unvermeidlichkeit beim Programmieren machen, kann man einteilen in:

- **Syntaxfehler**
  Diese verstoßen gegen eine Syntaxregel der verwendeten Programmiersprache, werden vom Compiler gemeldet und sind daher schon vor dem Starten eines Programms relativ leicht zu beseitigen.
- **Semantikfehler**
  Hier liegt *kein* Syntaxfehler vor, aber das Programm verhält sich anders als erwartet, wiederholt z. B. ständig eine nutzlose Aktion („Endlosschleife“) oder stürzt mit einem Laufzeitfehler ab. In jedem Fall sind die Benutzer verärgert, wenn sie auf einen solchen Fehler stoßen.

Die C# - Designer haben (z. B. durch strenge Typisierung) dafür gesorgt, dass möglichst viele Fehler vom Compiler aufgedeckt werden können, also zur Kategorie der Syntaxfehler gehören.[1]

Wir wollen am Beispiel eines provozierten Syntaxfehlers überprüfen, ob der C# - Compiler im .NET - Framework hilfreiche Fehlermeldungen produziert. Wenn im `Hallo`-Programm der Bezeichner **Console** fälschlicherweise mit kleinem Anfangsbuchstaben geschrieben wird,

```
using System;
class Hallo {
    static void Main() {
        console.WriteLine("Hallo, echt .NET hier!");
    }
}
```

Dann meldet der Compiler:

```
Hallo.cs(4,3): error CS0103: Der Name 'console' ist im aktuellen Kontext nicht
         vorhanden.
```

Der Compiler hat die fehlerhafte Stelle sehr gut lokalisiert: Datei **Hallo.cs**, Zeile 4, Spalte 3 (vor dem kleinen *c* stehen zwei Tabulatorzeichen). Auch die Fehlerbeschreibung fällt ziemlich eindeutig aus. Wer Erfahrungen mit Programmiersprachen wie Visual Basic oder Delphi hat, muss sich eventuell noch daran gewöhnen, dass in C# die Groß-/Kleinschreibung signifikant ist.

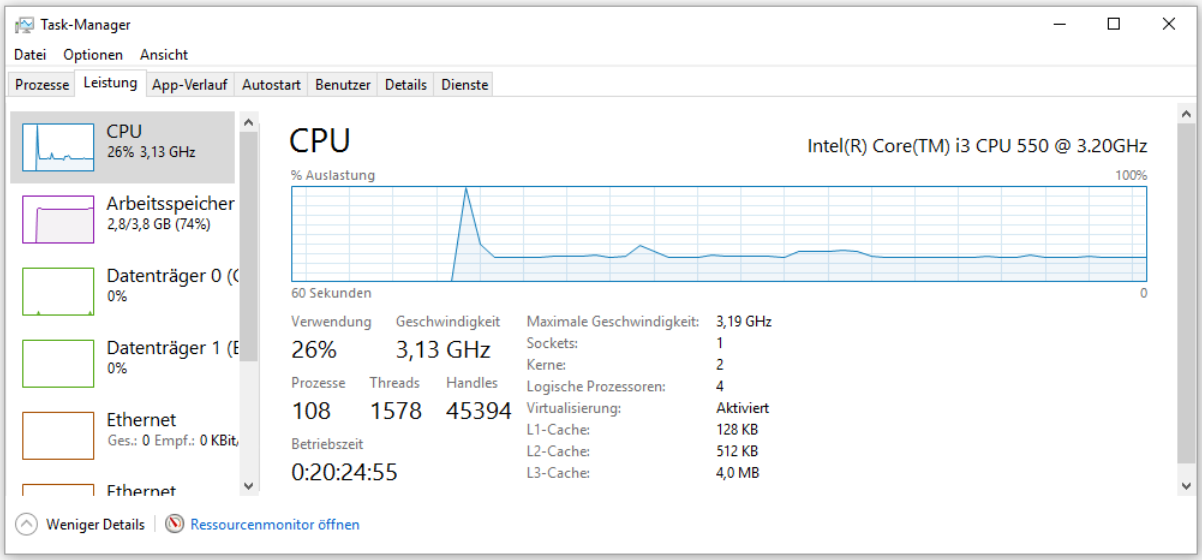
Während sich in die äußerst simple Klasse `Hallo` kaum ein Semantikfehler einbauen lässt, ist das im Bruchadditionsbeispiel aus Kapitel 1 leicht zu bewerkstelligen. Wird z. B. in der `Nenner` - Eigenschafts-Implementierung bei der Absicherung gegen Nullwerte der Ungleich-Operator (!=) durch sein Gegenteil (==) ersetzt, ist keine C# - Syntaxregel verletzt:

---

[1] Seit C# 4.0 ist mit dem Datentyp **dynamic** eine praktischen Zwängen (z. B. bei der Kooperation mit typfreien Skriptsprachen) geschuldete Ausnahme von der strengen Typisierung vorhanden. An Stelle des Compilers ist hier die CLR für die Typprüfung verantwortlich, was die Wahrscheinlichkeit von Laufzeitfehlern erhöht. Dieser Datentyp sollte nur in begründeten Ausnahmefällen verwendet werden; im Kurs kommt er nicht zum Einsatz.

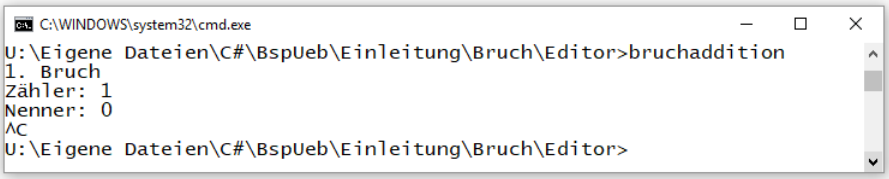
```
public int Nenner {
    get {
        return nenner;
    }
    set {
        if (value == 0)    // semantischer Fehler!
            nenner = value;
    }
}
```

Bei Eingabe kritischer „Brüche“ (wie z. B. $\frac{1}{0}$) verursacht die fehlerhafte Definition der Klasse `Bruch` aber ein unerwünschtes Verhalten des Programms `Bruchaddition`: Die Methode `Kuerze()` (vgl. Abschnitt 1.3) gerät in eine Endlosschleife, und das Programm verbraucht dabei reichlich Rechenzeit, wie der Windows-Taskmanager auf einem Rechner mit der Intel-CPU Core i3 (mit 4 logischen Kernen) zeigt:



Das sinnlos rotierende Programm lastet einen logischen Kern komplett aus, vergeudet also ca. 25% der CPU-Zeit.

Ein derart außer Kontrolle geratenes Konsolenprogramm beendet man z. B. mit der Tastenkombination **Strg+C**:



3.3 Entwicklung mit dem Visual Studio Community 2019

Auf die Dauer ist das Hantieren mit Notepad und Kommandozeile beim Entwickeln von C# - Software keine ernsthafte Option. Im weiteren Verlauf des Kurses werden wir das im Abschnitt 3.1 installierte Visual Studio 2019 der Firma Microsoft als bequeme und leistungsfähige Entwicklungsumgebung verwenden. Das Visual Studio hat mit seinen verschiedenen Varianten seit vielen Jahren einen großen Marktanteil bei der Software-Entwicklung unter Windows und wird von Fremdfirmen durch Erweiterungen für diverse Zwecke versorgt.

Das Visual Studio bietet u. a. folgende Leistungen:

- Intelligenter Editor (z. B. mit kontextsensitiver Auflistung von Optionen zur Syntaxvervollständigung)
- Grafischer Designer für die Bedienoberfläche eines Programms
- Verschiedene Assistenten, z. B. zur Datenbankanbindung
- Seit der Version 2019 enthält die Community Edition auch die CodeLens-Funktion, die bisher der kostenpflichtigen Enterprise-Edition vorbehalten war. Sie ist per Voreinstellung aktiv und zeigt z. B. für Typen und Methoden die Häufigkeit der Verwendung im Projekt an.

Die Firma Microsoft stellt mit der Visual Studio Community 2019 eine kostenlose Einstiegsversion ihrer Entwicklungsumgebung zur Verfügung, die für unsere Kurszwecke sehr gut geeignet ist und viele darüber hinausgehende Optionen bietet, z. B.:

- Neben Windows-Programmen kann man auch .NET - Anwendungen erstellen, die unter alternativen Betriebssystemen laufen (z. B. Android, iOS).
- Neben C# und VB.NET werden noch weitere .NET - Programmiersprachen unterstützt (z. B. C++, F#, Python).
- Weitere Projekt-Typen (z. B. Low Level C++)
  Bei der Liste der unterstützten Projektvorlagen steht die Community-Edition dem kommerziellen Visual Studio Professional *nicht* nach.
- Wie bei den kommerziellen Varianten steht die Plugin-Technik zur Erweiterung der Entwicklungsumgebung zur Verfügung.
- Visueller Klassen-Designer
  Ein vom Designer erstelltes Diagramm zur Klasse `Bruch` aus dem Einstiegsbeispiel war schon im Abschnitt 1.2 zu sehen.

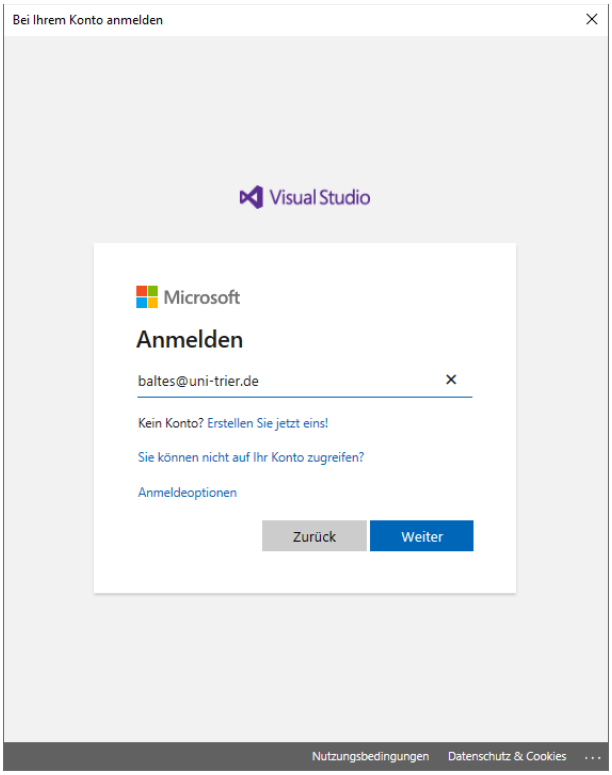
Viele dieser Leistungen gehören allerdings nicht zur Standardinstallation, sondern erhöhen nach Auswahl im Installationsprogramm den Bedarf an Massenspeicher erheblich.

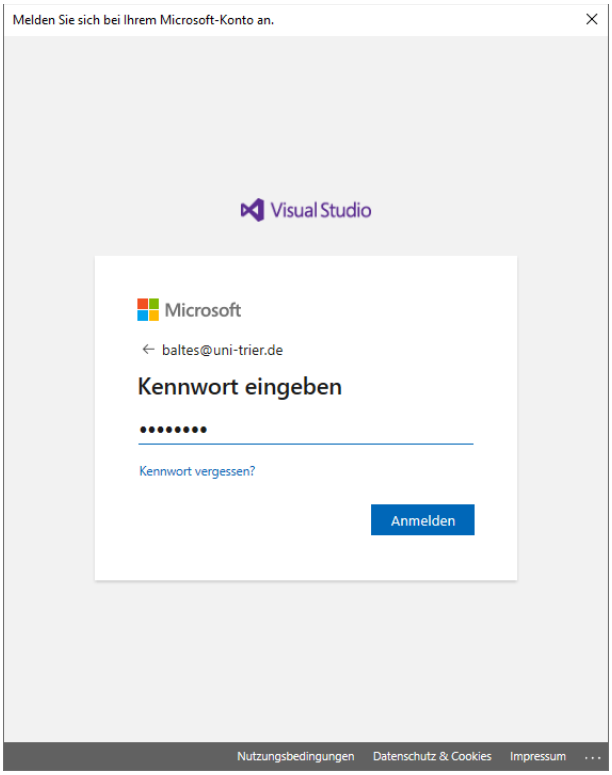
3.3.1 Initialisierung und Registrierung

Wir starten nun endlich das Visual Studio Community 2019. Microsoft verlangt für die kostenlose Variante seiner Entwicklungsumgebung eine Registrierung durch die Anmeldung mit einem Microsoft-Konto, die sich in den ersten 30 Tagen aufschieben lässt (Klick auf **Jetzt nicht, vielleicht später**):[1]

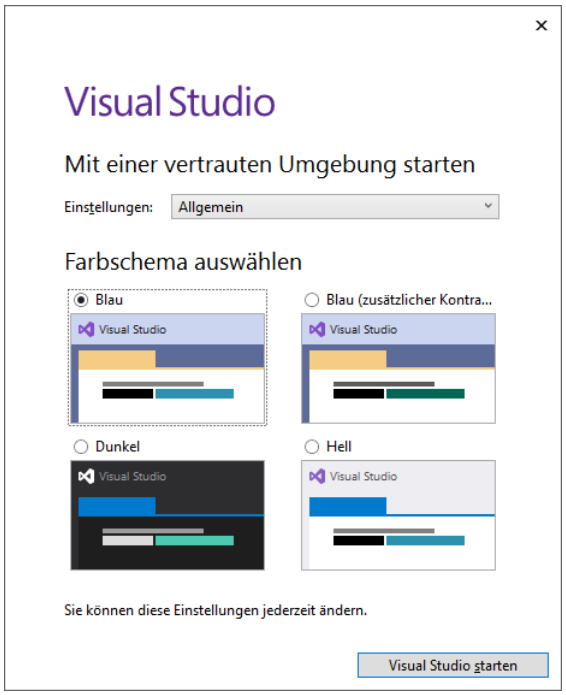
[1] Die 30 Tage mit unregistrierter Nutzungsdauer werden allerdings nicht pro Benutzer gewährt, sondern pro PC.

Mit einem vorhandenen Microsoft-Konto ist die Registrierung nach dem **Anmelden** schnell erledigt:



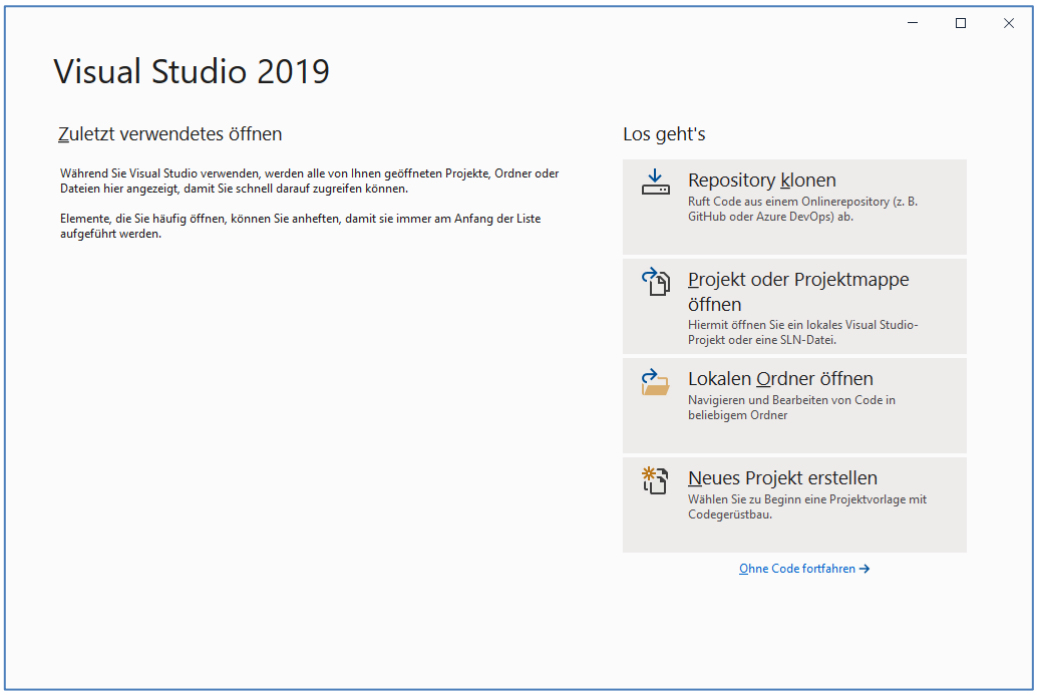


Sind keine Einstellungen von einer Vorversion vorhanden, kann man ein Farbschema für die Bedienoberfläche der Entwicklungsumgebung wählen:



Beim ersten Start der Entwicklungsumgebung sind einmalige Vorbereitungen erforderlich, die in Abhängigkeit vom verwendeten Entwicklungsrechner Sekunden oder Minuten dauern:

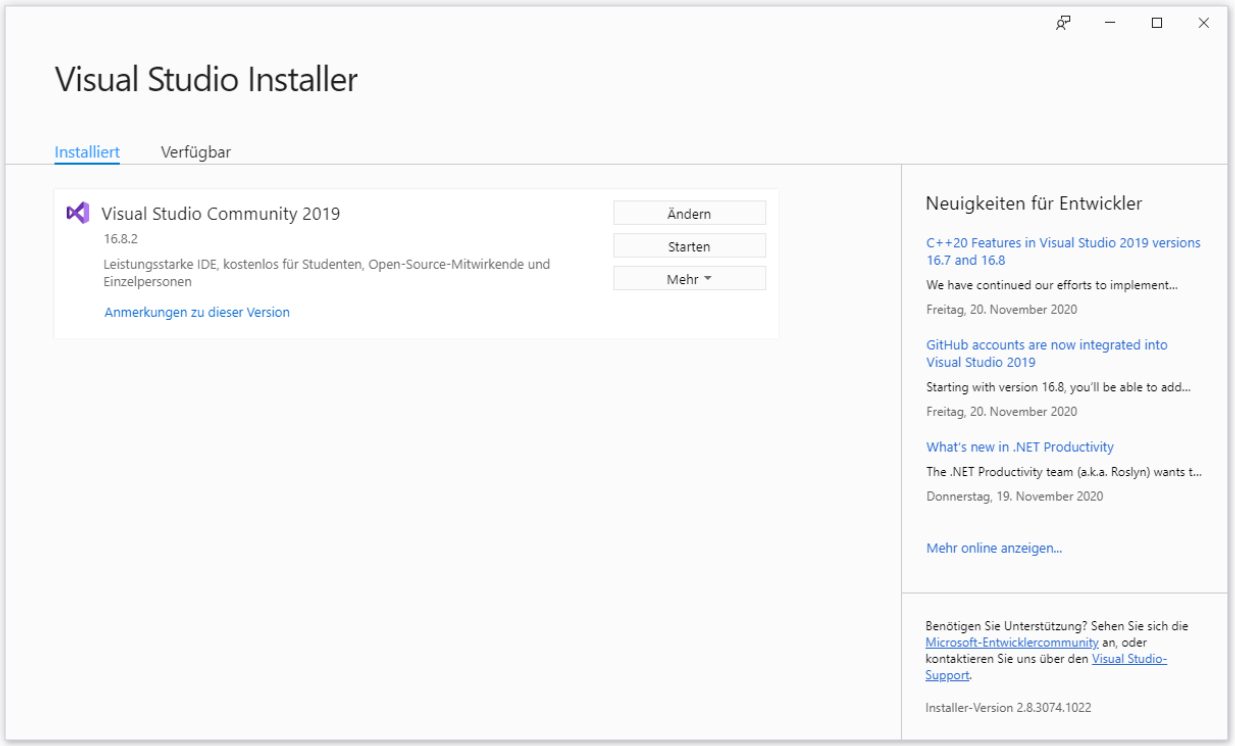
Schließlich kann es losgehen:



3.3.2 Spätere Anpassungen und Aktualisierungen der Installation

Eine Visual Studio - Installation lässt sich flexibel ändern, indem z. B. Sprachpakete oder Einzelkomponenten (wie der UML - Klassen-Designer) aufgenommen oder entfernt werden.

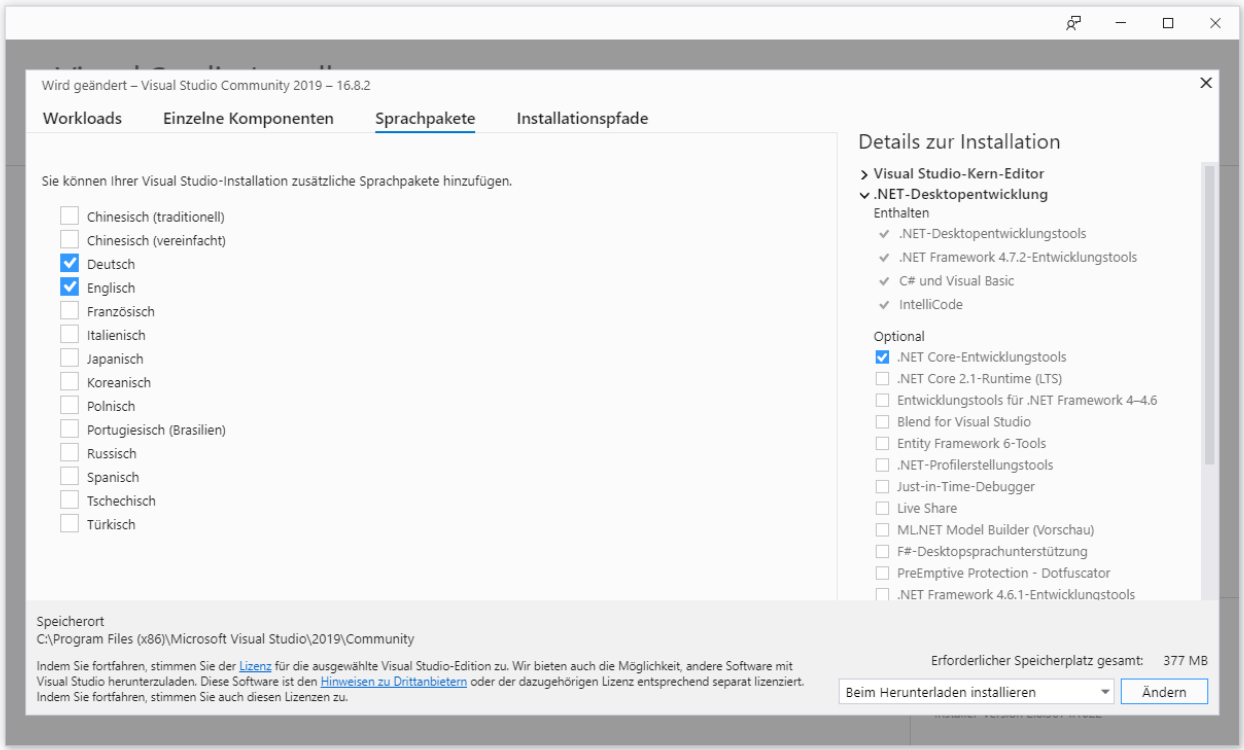
Starten Sie den **Visual Studio - Installer** über seinen Eintrag im Startmenü,



und klicken Sie auf **Ändern**.

3.3.2.1 Bedienoberfläche in englischer Sprache

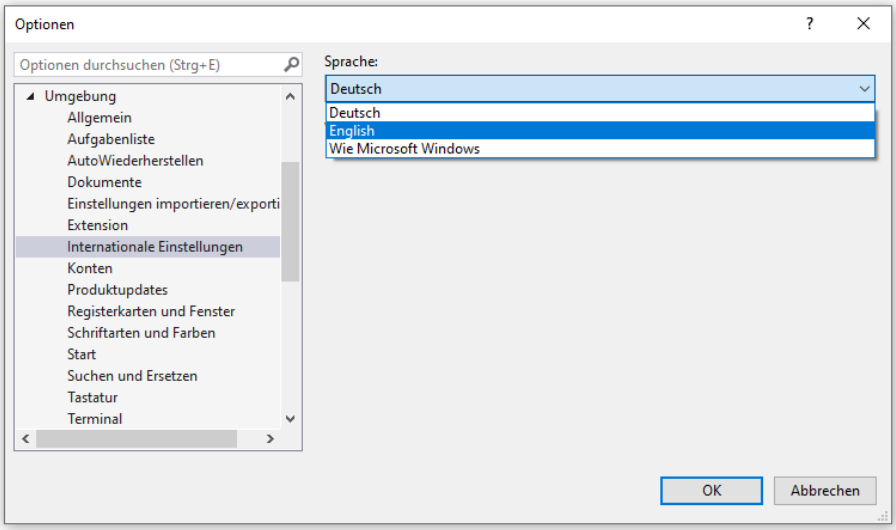
Ein (temporäres) Umschalten auf eine englische Bedienoberfläche kann nützlich sein, weil sich die meisten Anleitungen und Tipps auf die englische Bedienoberfläche beziehen. Daher sollten Sie auf der Registerkarte **Sprachpakete** das englische Sprachpaket hinzufügen:



Nach einem Klick auf **Ändern** ist die Installationserweiterung in wenigen Minuten erledigt, und nach dem nächsten Start der Entwicklungsumgebung kann über

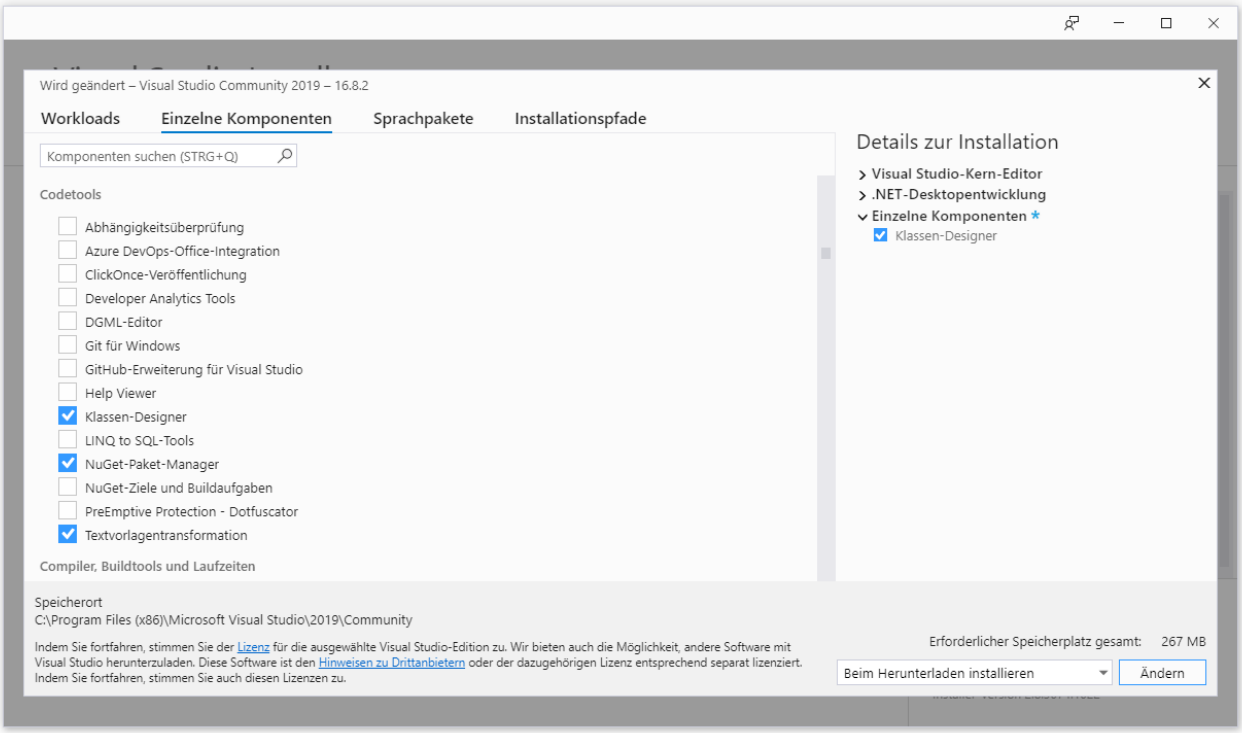
Extras > Optionen > Umgebung > Internationale Einstellungen

die Sprache der Bedienoberfläche geändert werden:



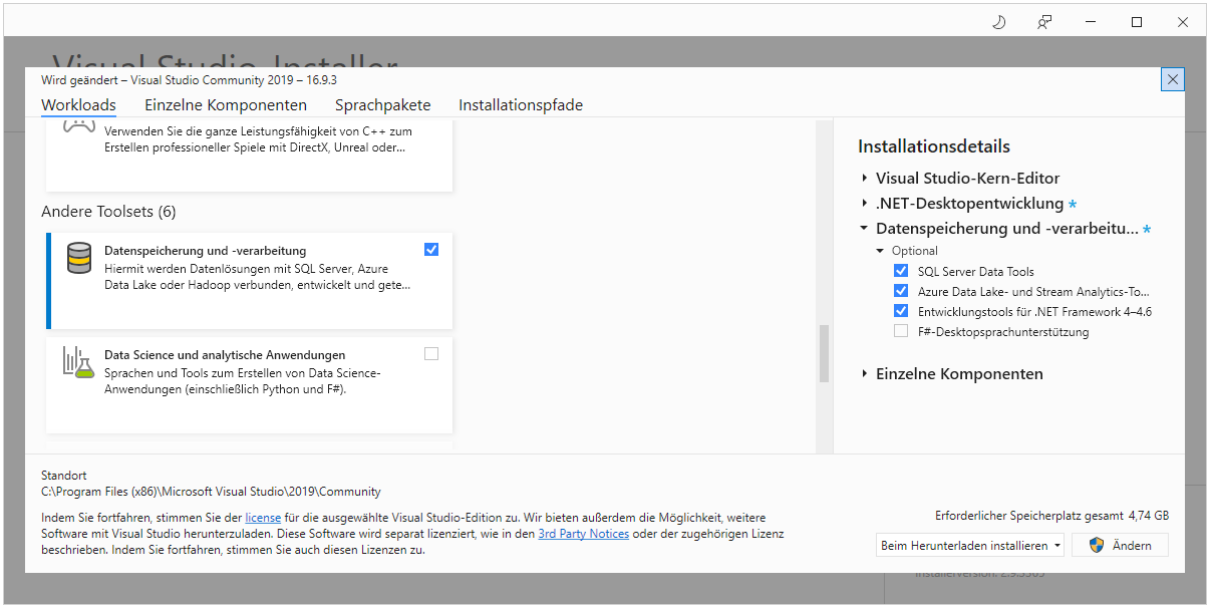
3.3.2.2 Klassen-Designer

Auf der Registerkarte mit **einzelnen Komponenten** hält der **Visual Studio - Installer** einige attraktive Optionen bereit. Im Bereich mit den **Codetools** sollten Sie den **Klassen-Designer** ergänzen, um in Ihren Projekten UML-Diagramme verwenden zu können (vgl. Abschnitt 1.2):

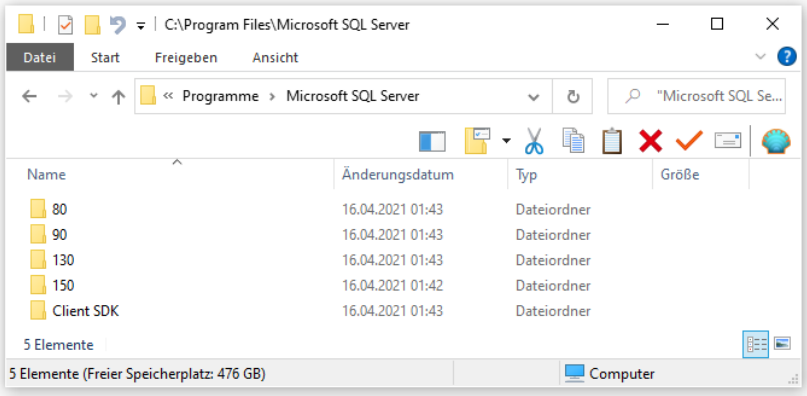


3.3.2.3 SQL Server Data Tools

Im Kapitel 18 werden wir in die Datenbankprogrammierung einsteigen und dazu auch einen SQL Server installieren. Damit das Visual Studio mit dem SQL - Server kooperieren kann, muss die Entwicklungsumgebung um die SQL Server Data Tools (SSDT) erweitert werden. Dazu wählen wir im Visual Studio Installer auf der Registerkarte **Workloads** die **Datenspeicherung und -verarbeitung**:

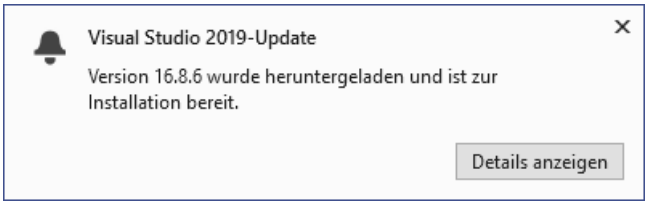


Es wird u. a. die LocalDB-Variante von Microsofts SQL Server 2016 installiert:

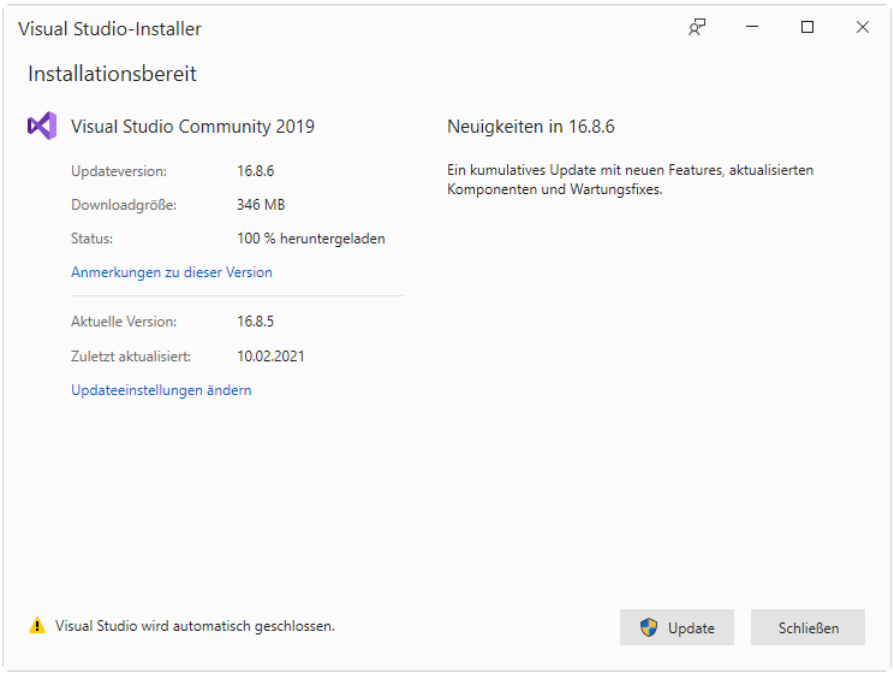


3.3.2.4 Kontinuierliche Aktualisierungen

Im Abstand von wenigen Wochen erscheinen Aktualisierungen der Benutzerumgebung, die sich mit dem Visual Studio - Installer bequem durchführen lassen. Das laufende Visual Studio informiert wie im folgenden Beispiel über ein verfügbares und schon heruntergeladenes Update:



Nach einem Klick auf Details anzeigen startet der **Visual Studio - Installer**:



Nach einem Klick auf **Update** wird ...

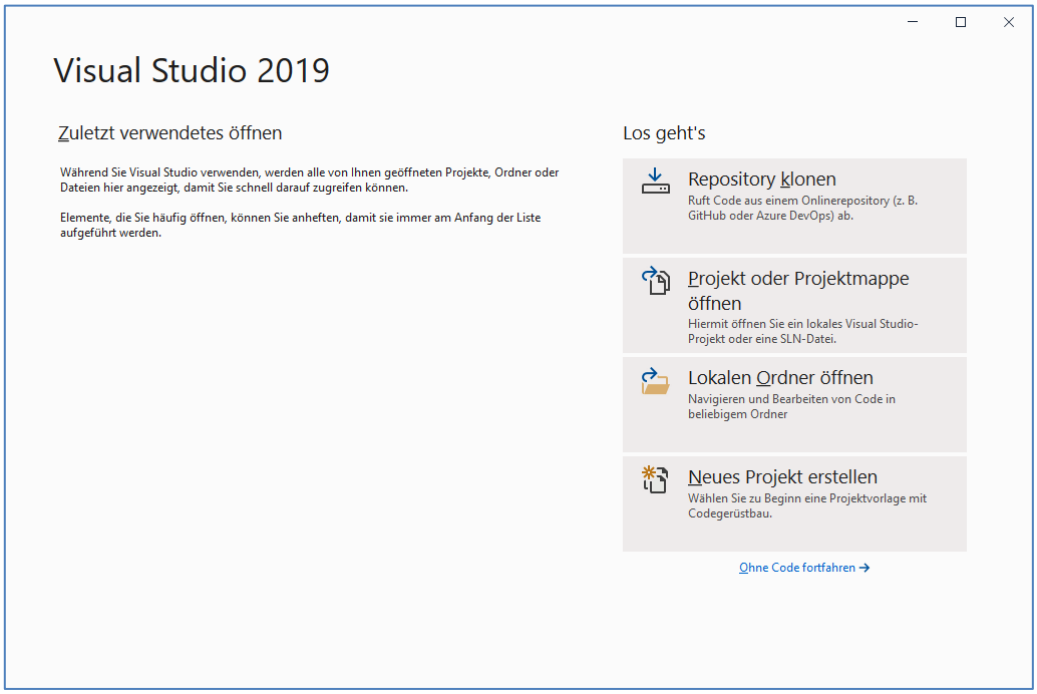
- das Visual Studio automatisch geschlossen,
- das Update installiert,
- das Visual Studio wieder gestartet.

3.3.3 Ein erstes Konsolen-Projekt

Nachdem wir im Abschnitt 3.2 aus didaktischen Gründen ausschließlich mit dem .NET Framework gearbeitet haben, verwenden wir nun auch .NET 5.0, um wichtige Unterschiede zwischen den beiden .NET - Implementationen erläutern zu können.

3.3.3.1 .NET Framework

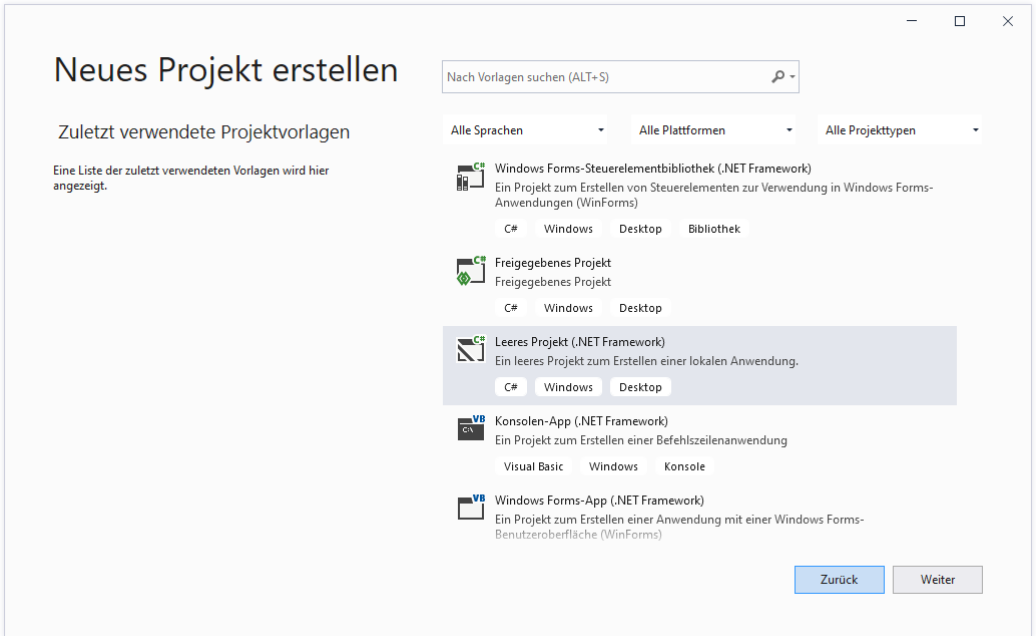
Wir öffnen über die Option **Neues Projekt erstellen** im Begrüßungsdialog,



oder bei bereits vorhandenem Visual Studio - Anwendungsfenster mit dem Schalter  oder mit dem Menübefehl

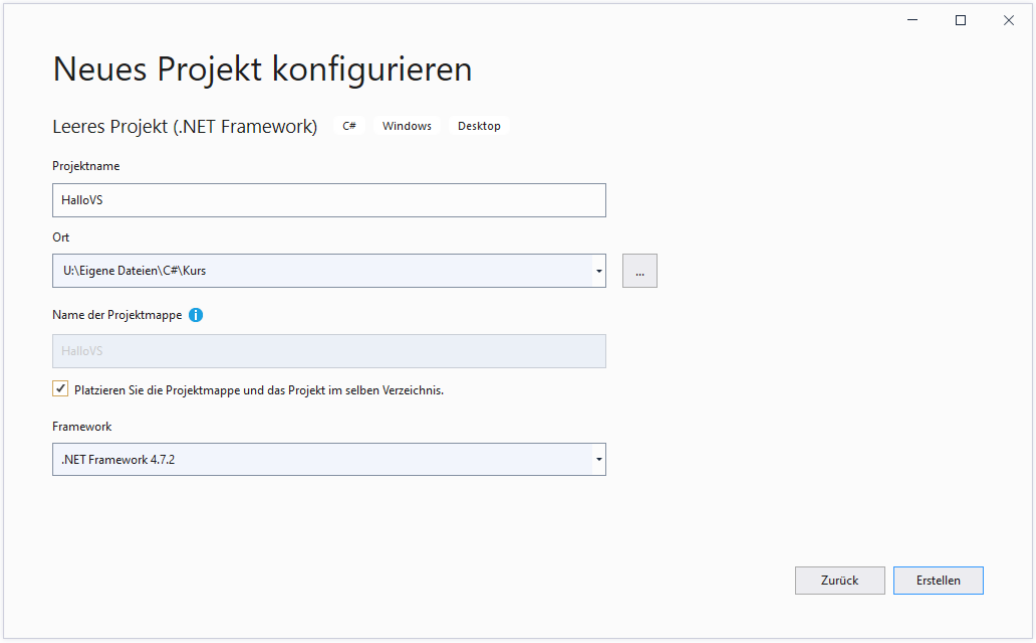
Datei > Neu > Projekt

den Dialog für **neue Projekte**:



Wir wählen ein **Leeres Projekt (.NET Framework)** und machen **weiter**.

Im nächsten Dialog tragen wir den **Projektnamen** `HalloVS` ein, der als **Name der Projektmappe** übernommen wird:



Als **Ort**, an dem ein Unterordner für das Projekt angelegt werden soll, schlägt das Visual Studio beim Windows-Benutzer `otto` vor:

C:\Users\otto\source\repos

Auf einem ZIMK-Pool-PC an der Universität Trier eignet sich als Speicherort z. B.:

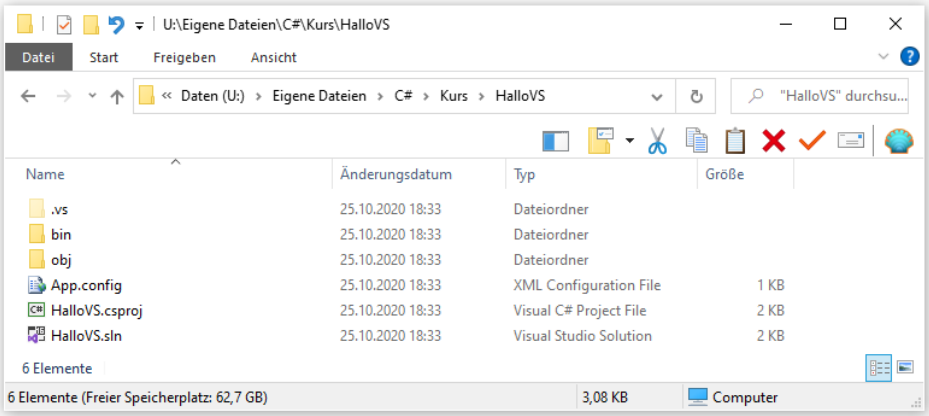
U:\Eigene Dateien\C#\Kurs

Jedes Projekt gehört zu einer **Projektmappe**, die eine *Familie* von zusammengehörigen Projekten (z. B. Client- und Server-Anwendung für einen Dienst) verwaltet und von der Entwicklungsumgebung automatisch angelegt wird. Von der englischen Version der Entwicklungsumgebung wird eine Projektmappe als *Solution* bezeichnet, und gelegentlich taucht die deutsche Übersetzung *Lösung* auf. Bei unseren Kursbeispielen werden die Projektmappen jeweils nur ein einziges Projekt enthalten. In dieser Situation ist es überflüssig, im Ordner einer Projektmappe einen Unterordner für das einzige enthaltene Projekt anzulegen. Daher markieren wir das Kontrollkästchen **Platzieren Sie die Projektmappe und das Projekt im selben Verzeichnis** und wählen damit eine flachere Projektdateiverwaltung.

Als vorausgesetztes **Framework** akzeptieren wir die Voreinstellung 4.7.2.

Nach einem Mausklick auf **Erstellen** entsteht im Projektordner

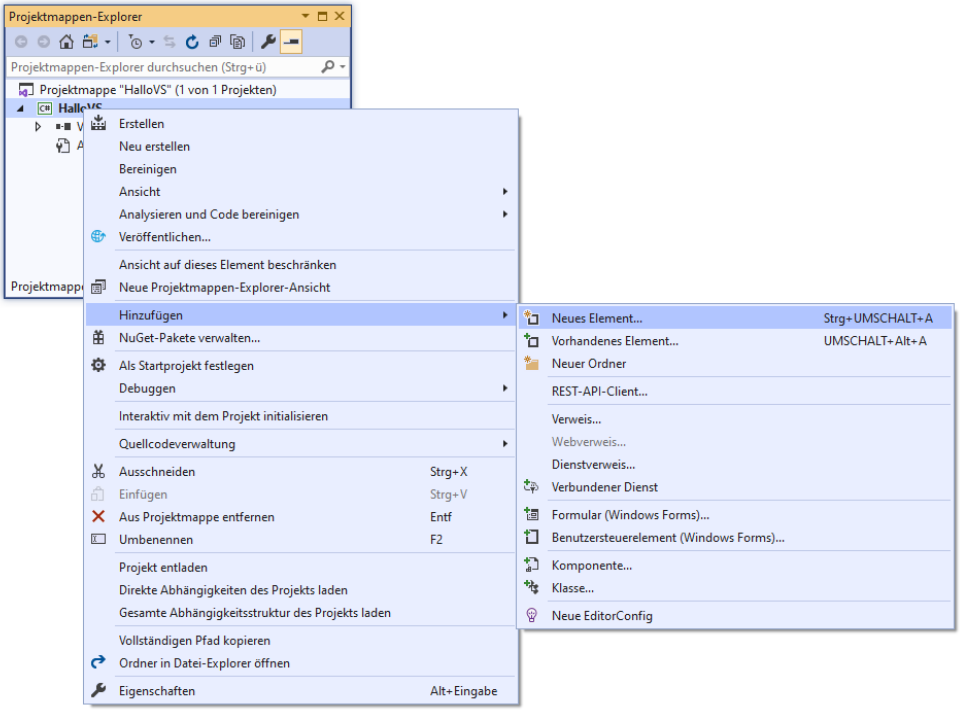
U:\Eigene Dateien\C#\Kurs\HalloVS



das neue Projekt mit den folgenden Dateien, über die sich das Projekt später (z. B. per Doppelklick) öffnen lässt:

- **HalloVS.csproj** (Projekt)
- **HalloVS.sln** (Projektmappe)

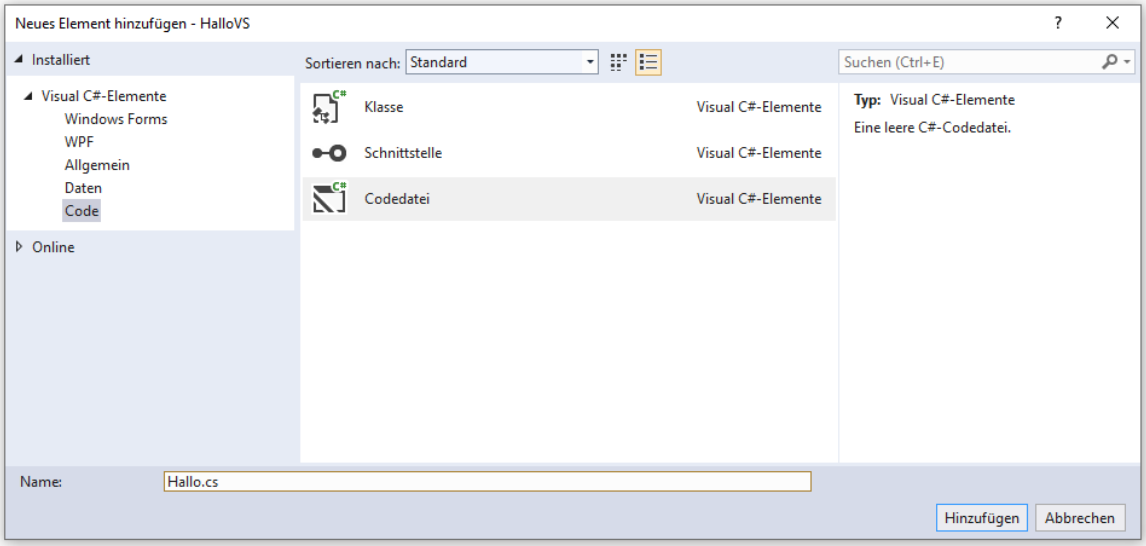
Öffnen Sie im **Projektmappen-Explorer** der Entwicklungsumgebung (am rechten Fensterrand) per Maus-Rechtsklick das Kontextmenü zum *Projekt* `HalloVS` (nicht zur *Projektmappe*), und fügen Sie ein **neues Element** hinzu:



Zum selben Zweck taugt bei markiertem Projekt auch der Menübefehl

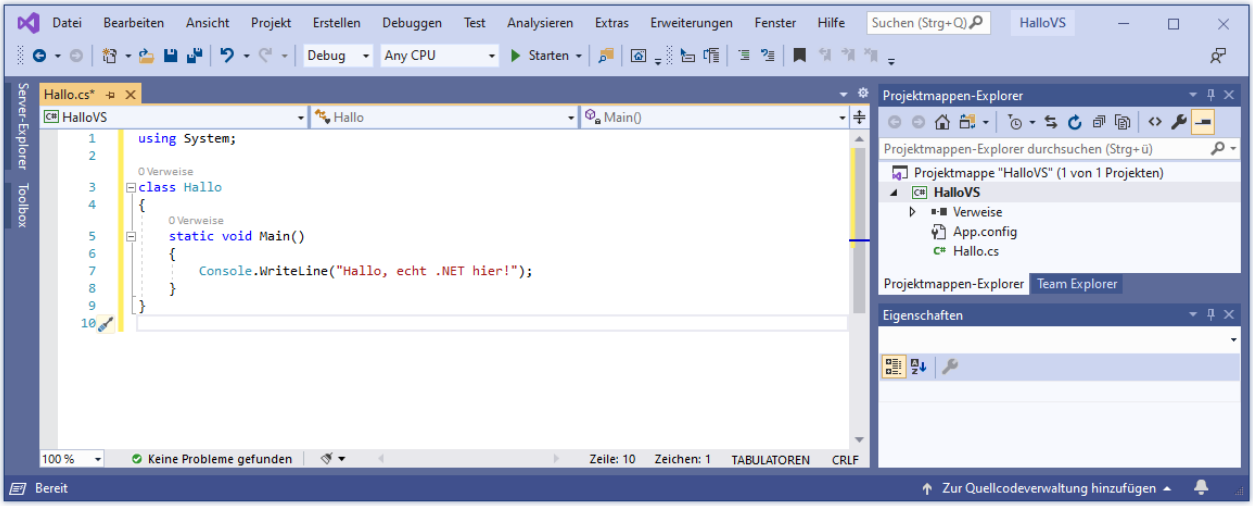
Projekt > Neues Element hinzufügen

Entscheiden Sie sich im Dialog für **neue Elemente** für eine **Codedatei** mit dem **Namen Hallo.cs**:



Nach dem **Hinzufügen** ist die Datei im Quellcode-Editor der Entwicklungsumgebung geöffnet.

Wir übernehmen den Quellcode vom `Hallo`-Beispielprogramm aus dem Abschnitt 3.2.1:

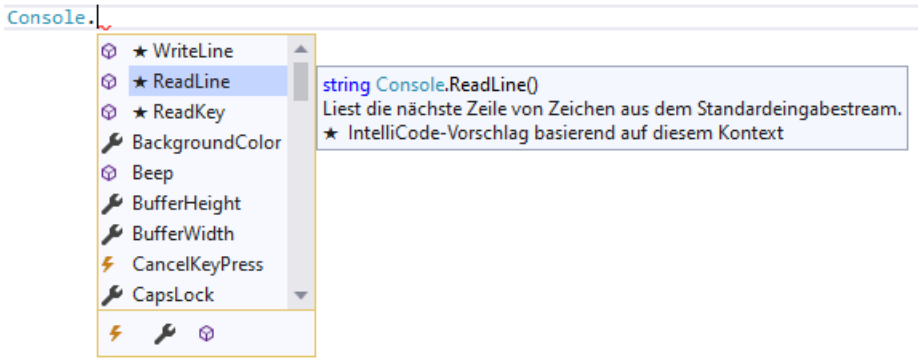


Damit bei einem Programmstart ...

- per Doppelklick auf das fertige Exe-Assembly in einem Windows-Explorer - Fenster
- oder im Rahmen der Entwicklungsumgebung mit dem Schalter ▶ Starten oder der Funktionstaste **F5**

das automatisch erscheinende Konsolenfenster mit unserem Programm nach der Hallo-Ausgabe *nicht* spontan verschwindet, sondern bis zur Betätigung der **Enter**-Taste stehen bleibt, bitten wir am Ende der **Main()** - Methodendefinition die Klasse **Console**, ihre Methode **ReadLine()** auszuführen. Diese Methode wartet auf die **Enter**-Taste und verhindert so, dass die Konsolenanwendung nach der Bildschirmausgabe sofort verschwindet.

Wir starten im Quellcodeeditor in einer neuen Zeile mit dem Namen der Klasse und setzen einen Punkt dahinter. Nun erscheinen die erlaubten Fortsetzungen, also im konkreten Fall die öffentlichen Member der Klasse **Console**. Die **IntelliCode**-Vorschläge mit den Kontext-abhängigen Favoriten erweisen sich als brauchbar:



Nach einem Doppelklick auf **ReadLine** setzen wir hinter den Methodennamen ein Paar runder Klammern um den Methodenaufruf zu komplettieren.

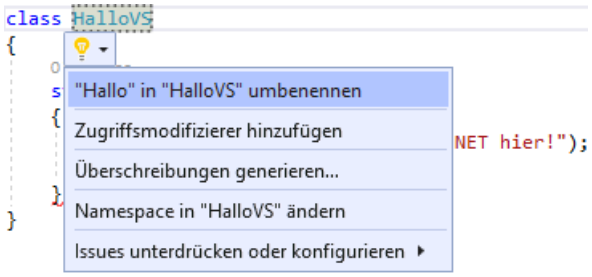
Der Quellcode-Editor unserer Entwicklungsumgebung bietet außerdem …

- farbliche Unterscheidung verschiedener Sprachbestandteile
- automatische Quellcode-Formatierung (z. B. bei Einrückungen)
- automatische Syntaxprüfung, z. B.:

```
static void Main()
{
    Console.WriteLine("Hallo, echt .NET hier!");
    Console.ReadLine()
}
```

  Wir haben mittlerweile einen Methodenaufruf erstellt, der aber keine vollständige Anweisung ist, was unsere Entwicklungsumgebung durch rotes Unterschlängeln der mutmaßlichen Fehlerstelle reklamiert. Wir ergänzen das an dieser Stelle erforderliche Semikolon.

Außerdem passen wir den Namen der Startklasse (aktuell: `Hallo`) an den Projektnamen `HalloVS` an. Das Umbenennen einer Klasse, Methode oder Variablen kann in einem komplexen Projekt diverse Quellcodeänderungen erfordern, ist also aufwändig und fehleranfällig. Zum Glück kann unsere Entwicklungsumgebung solche Umgestaltungen, die man zu den *Refaktorierungen* rechnet, sehr gut unterstützen. Wenn Sie mit der Maus auf den geänderten Klassennamen zeigen, erscheint eine Glühbirne,



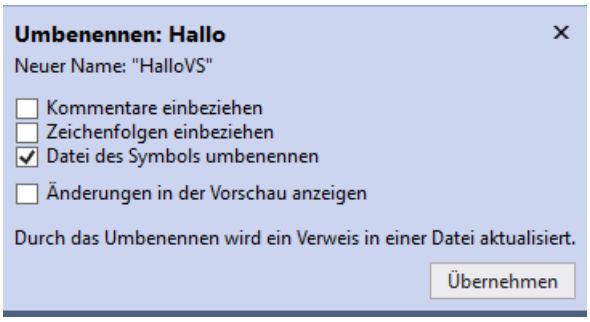
deren Drop-Down - Menü das Projekt-globale Umbenennen der Klasse anbietet. In dem sehr einfachen Programm ist kein weiteres Auftreten des Klassennamens zu aktualisieren. Sie werden aber bald die Möglichkeit schätzen lernen, in einem komplexen Programm einen Bezeichner zu ändern, ohne über die damit an vielen Stellen erzwungenen Aktualisierungen nachdenken zu müssen.

Statt durch Umbenennen eine schwierige Lage zu provozieren, auf die das Visual Studio mit einem Geistesblitz reagiert, sollte man das Umbenennen mit Ansage über die Bühne bringen:

- Setzen Sie die Schreibmarke auf den zu ändernden Bezeichner.
- Fordern Sie das Umbenennen an mit der Tastenkombination **Strg+R**, **Strg+R** (Tastenkombination zweimal wiederholen) oder mit dem folgenden Menübefehl:

  Bearbeiten > Umgestalten > Umbenennen

- Ändern Sie den Bezeichner und klicken Sie auf **Übernehmen**:



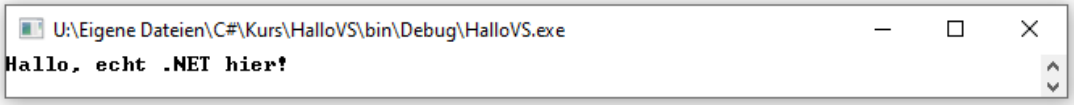
Aufgrund des markierten Kontrollkästchens **Datei des Symbols umbenennen** wird auch die zur Klasse gehörige Quellcodedatei umbenannt (in **HalloVS.cs**).

Das Projekt kann jederzeit über den Schalter oder den Menübefehl

Datei > Alles Speichern

gespeichert werden. Allerdings speichert die Entwicklungsumgebung auch unaufgefordert zu passenden Gelegenheiten.

Über den Schalter Starten oder die Funktionstaste **F5** veranlasst man das Übersetzen und das anschließende Starten der Anwendung:

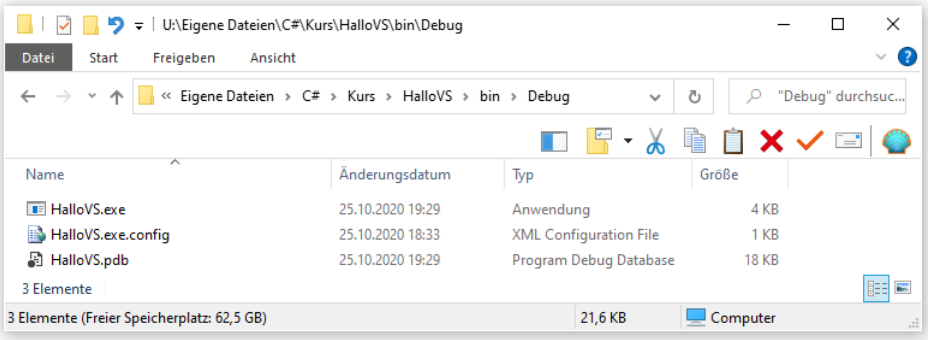


Sobald Sie die **Enter**-Taste drücken, kehrt die von der Klasse **Console** ausgeführte **ReadLine()** - Methode zurück. Danach endet mit der **Main()** - Methode das Programm, und das Konsolenfenster verschwindet.

Beim Programmstart mit **F5**, Starten oder mit dem Menübefehl

Debuggen > Debugging starten

wird die Debug-Konfiguration verwendet, sodass der Compiler ein gut testbares, aber nicht leistungsoptimiertes Assembly erzeugt und im Projekt-Unterordner **…\bin\Debug** ablegt, z. B.:



Außerdem zeigt das Visual Studio während der Programmaktivität diverse diagnostische Informationen an, mit denen wir momentan noch nicht viel anfangen können.

Wird das Programm im Rahmen der Entwicklungsumgebung mit **Strg+F5** oder mit dem Menübefehl[1]

[1] Sollte der Menübefehl fehlen und die Tastenkombination ohne Reaktion bleiben, dann hilft es, eine Quellcodedatei im Editor zu öffnen.

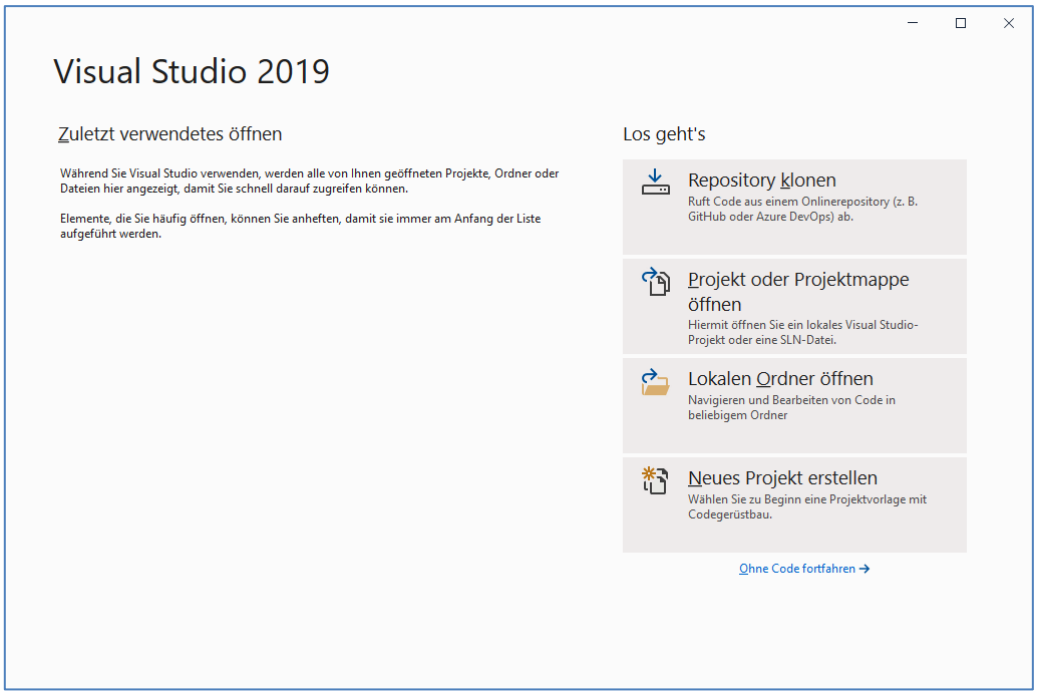
Debuggen > Starten ohne Debugging

gestartet, dann verwendet die Entwicklungsumgebung ebenfalls die Debug-Konfiguration (mit dem Ausgabeordner **…\bin\Debug**), verzichtet aber während des Programmlaufs auf die Anzeige diagnostischer Informationen. Außerdem wird hinter den Programmaufruf ein **Pause**-Kommando gesetzt, sodass kein **ReadLine()** - Methodenaufruf benötigt wird, um die letzte Ausgabe des Programms auf dem Bildschirm zu halten, z. B.:

Mit der bei auslieferungsbereiten Programmen sinnvollen Release-Konfiguration werden wir uns im Abschnitt 3.3.4.4 beschäftigen. In jedem Fall ist das Assembly **HalloVS.exe** einsatzfähig und kann auf jedem Windows-Rechner mit passendem .NET - Framework ausgeführt werden.

3.3.3.2 .NET 5.0

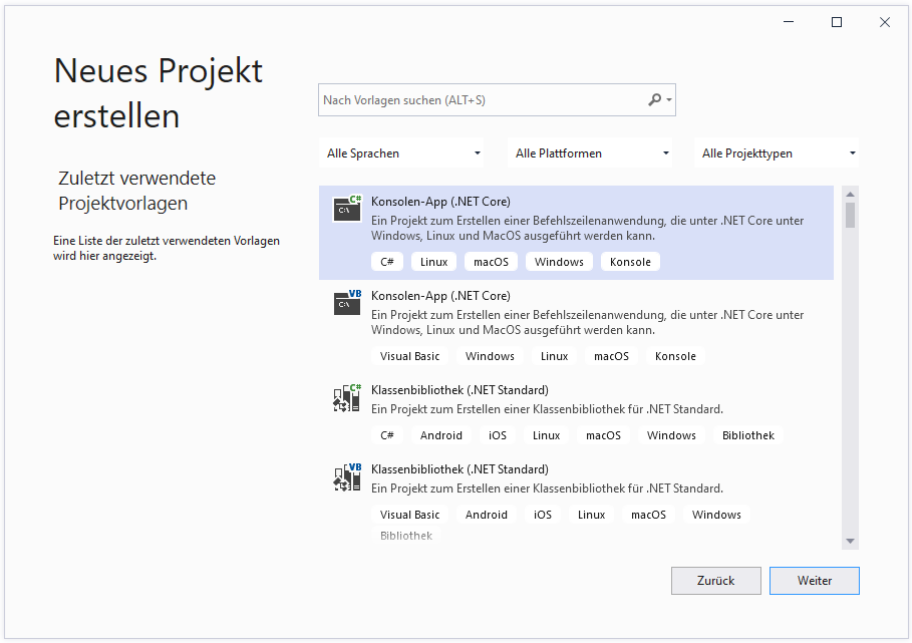
Wir öffnen über die Option **Neues Projekt erstellen** im Begrüßungsdialog,



oder bei bereits vorhandenem Visual Studio - Anwendungsfenster mit dem Schalter  oder mit dem Menübefehl

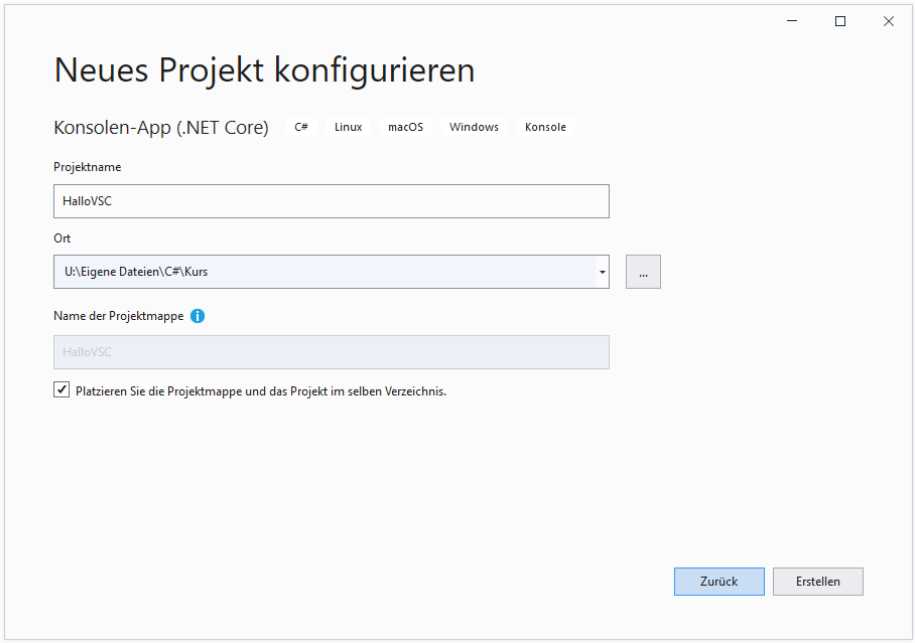
Datei > Neu > Projekt

den Dialog für **neue Projekte**:



Wir wählen eine **Konsolen-App (.NET Core)** und machen **weiter**.

Im nächsten Dialog tragen wir den **Projektnamen** `HalloVSC` ein, der als **Name der Projektmappe** übernommen wird:



Als **Ort**, an dem ein Unterordner für das Projekt angelegt werden soll, schlägt das Visual Studio beim Windows-Benutzer `otto` vor:

C:\Users\otto\source\repos

Auf einem ZIMK-Pool-PC an der Universität Trier eignet sich als Speicherort z. B.:

U:\Eigene Dateien\C#\Kurs

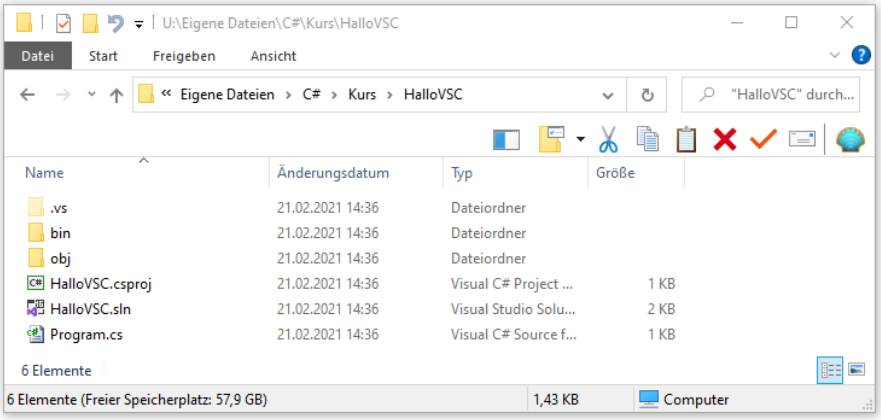
Wie schon beim Konsolenprojekt für das .NET Framework praktiziert (siehe Abschnitt 3.3.3.1), markieren wir das Kontrollkästchen

Platzieren Sie die Projektmappe und das Projekt im selben Verzeichnis

und wählen damit eine flache Projektdateiverwaltung.

Nach einem Mausklick auf **Erstellen** entsteht im Projektordner

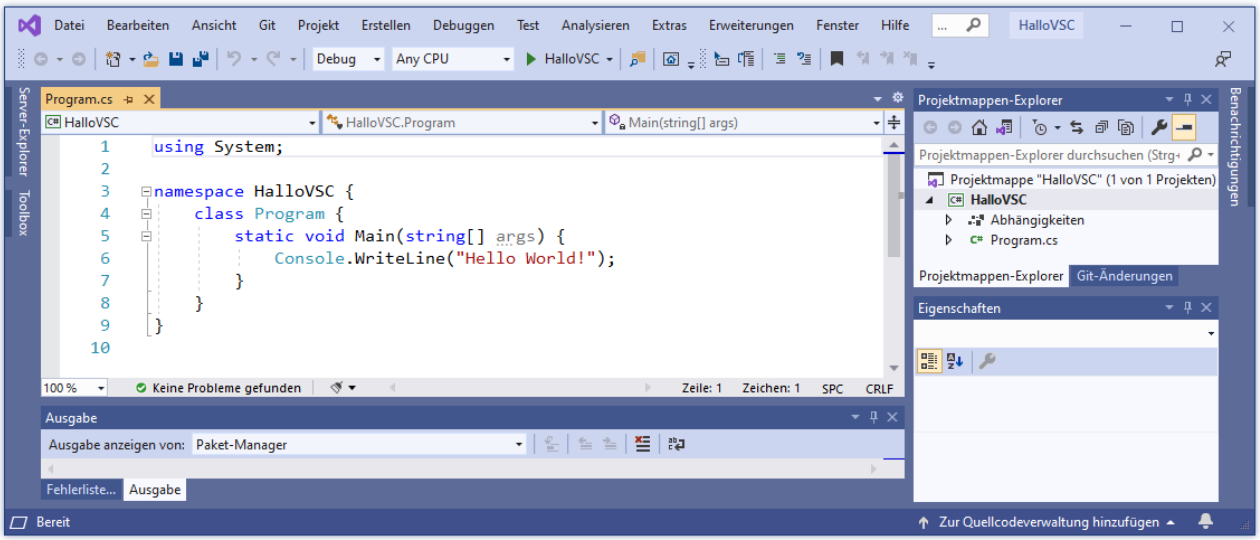
U:\Eigene Dateien\C#\Kurs\HalloVSC



das neue Projekt mit den folgenden Dateien, über die sich das Projekt später (z. B. per Doppelklick) öffnen lässt:

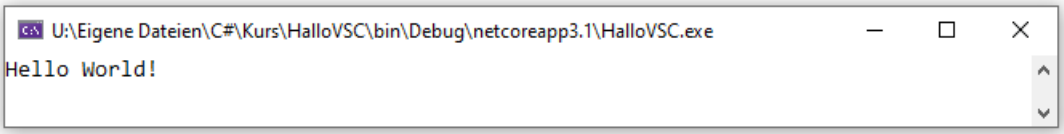
- **HalloVSC.csproj** (Projekt)
- **HalloVSC.sln** (Projektmappe)

Das Visual Studio hat ein „vollständiges“ Hallo-Programm erstellt, das sogar einen eigenen Namensraum definiert:

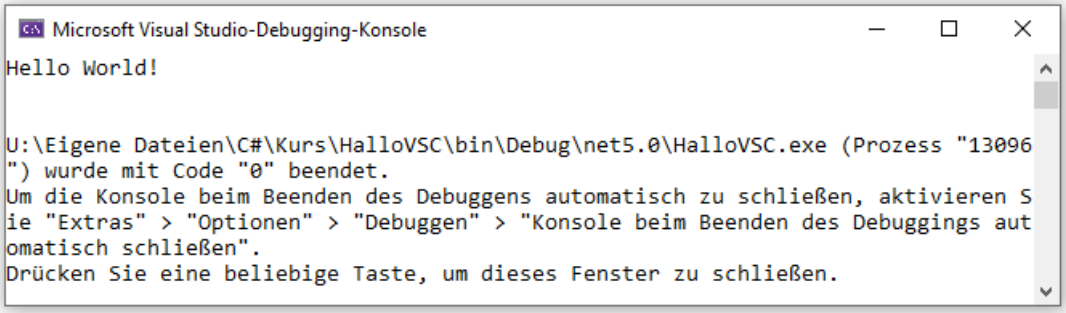


Damit bei einem Programmstart per Doppelklick auf die fertig erstellte Datei **HalloVSC.exe** in einem Windows-Explorer - Fenster das automatisch erscheinende Konsolenfenster mit unserem Programm nach der Hallo-Ausgabe *nicht* spontan verschwindet, sondern bis zur Betätigung der **Enter**-Taste stehen bleibt, bitten wir am Ende der **Main()** - Methodendefinition die Klasse **Console**, ihre Methode **ReadLine()** auszuführen. Diese Methode wartet auf die **Enter**-Taste und verhindert so, dass die Konsolenanwendung nach der Bildschirmausgabe sofort verschwindet. Wie sich dieser Methodenaufruf bequem und tippfehlerfrei mit Hilfe der **IntelliCode**-Vorschläge der Entwicklungsumgebung erstellen lässt, wurde im Abschnitt 3.3.3.1 vorgeführt.

Über den Schalter ▶ HalloVSC oder die Funktionstaste **F5** veranlasst man im Visual Studio das Übersetzen und das anschließende Starten der Anwendung:



Sobald Sie die **Enter**-Taste drücken, kehrt die von der Klasse **Console** ausgeführte **ReadLine()** - Methode zurück. Danach endet mit der **Main()** - Methode das Programm. Um das aus dem Visual Studio gestartete Konsolenfenster zu schließen,



ist noch eine weitere Betätigung der **Enter**-Taste erforderlich.

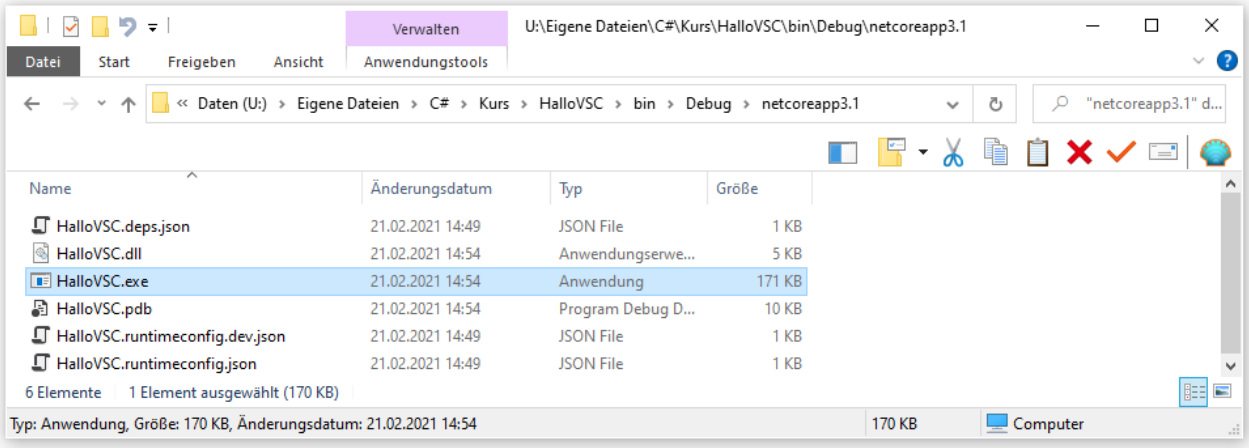
Beim Programmstart mit **F5**, ▶ HalloVSC oder mit dem Menübefehl

Debuggen > Debugging starten

wird die Debug-Konfiguration verwendet, sodass der Compiler ein gut testbares, aber nicht leistungsoptimiertes Assembly erzeugt und im folgenden Projekt-Unterordner

...\bin\Debug\netcoreapp3.1

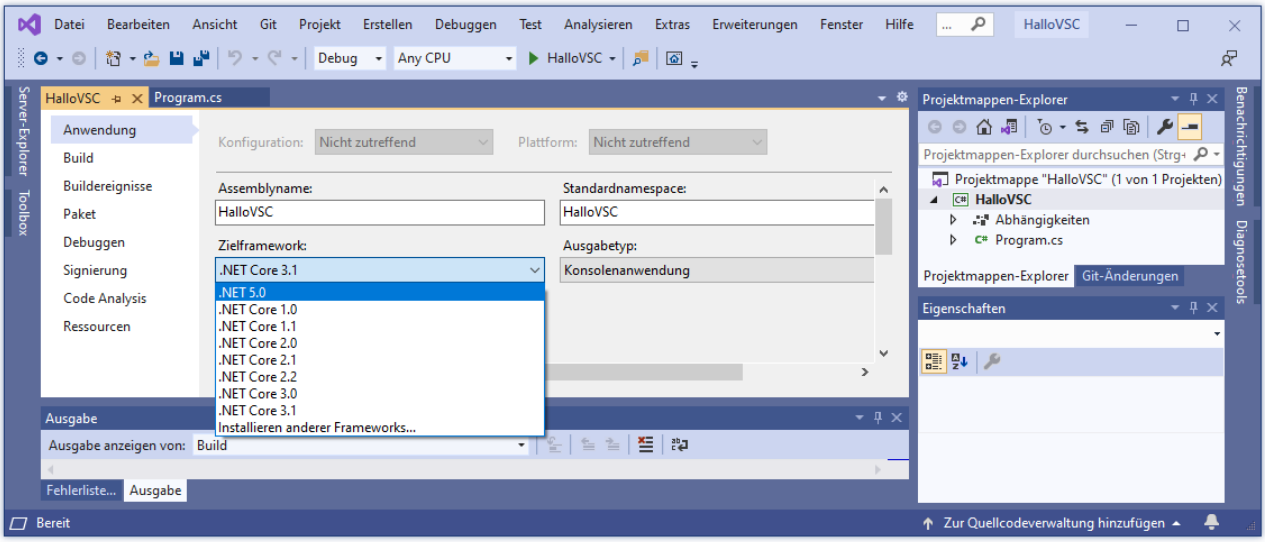
ablegt, z. B.:



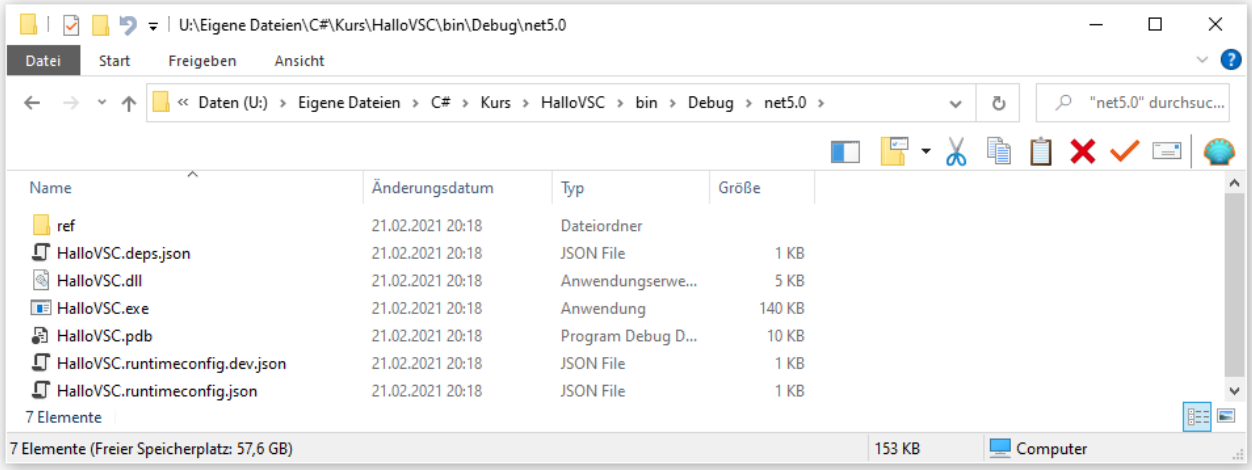
Ersetzt man über

Projekt > Eigenschaften

das voreingestellte **.NET Core 3.1** als **Zielframework** durch **.NET 5.0**,

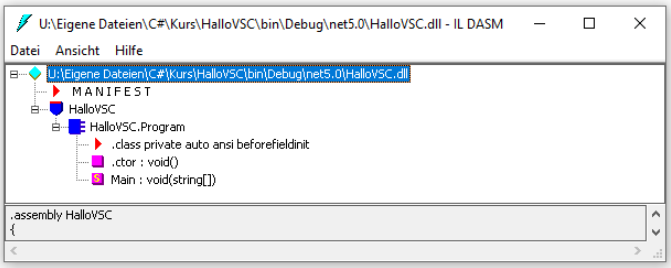


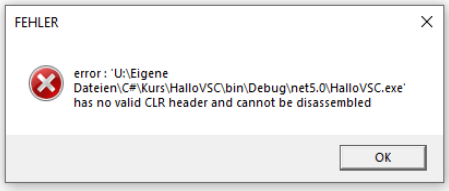
dann ändert sich der Ausgabeordner für die Projekterstellung:



Die Liste der erstellten Dateien unterscheidet sich von der analogen Liste im Hallo-Projekt für das .NET Framework (siehe Abschnitt 3.3.3.1):

- Zur Konfiguration dienen mehrere Dateien im **JSON**-Format (*JavaScript Object Notation*), während das .NET Framework - Projekt *eine* Anwendungskonfigurationsdatei mit der Namenserweiterung **.config** enthält.
- Das erstellte Assembly mit dem ausführbaren Programm befindet sich in der Datei **HalloVSC.dll**, während die Datei **HalloVSC.exe** ein unter Windows benötigter Starthelfer ist. Dass nicht **HalloVSC.exe** sondern **HalloVSC.dll** das ausführbare Assembly enthält, lässt sich mit dem **ILDasm** verifizieren:





- Zum Starten der Anwendung setzt man unter Windows wie gewohnt einen Doppelklick auf die **EXE**-Datei. Der erwünschte Effekt stellt sich aber nur dann ein, wenn zuvor die benötigte .NET Core - bzw. .NET 5.0 - Laufzeitumgebung installiert worden ist. Anderenfalls erscheint beim Startversuch die folgende Fehlermeldung:

```
>HalloVSC.exe
It was not possible to find any compatible framework version
The specified framework 'Microsoft.NETCore.App', version '5.0.0' was not found.
```

- Die Datei **HalloVSC.pdb**, zu der ein Analogon auch im .NET Framework - Projekt vorhanden ist, enthält Informationen für die Lokalisation von eventuell auftretenden Laufzeitfehlern (siehe Abschnitt 3.3.4.4).[1]

Wird das Programm im Rahmen der Entwicklungsumgebung mit **Strg+F5** oder mit dem Menübefehl

Debuggen > Starten ohne Debugging

gestartet, dann verwendet die Entwicklungsumgebung ebenfalls die Debug-Konfiguration (unter .NET 5.0 mit dem Ausgabeordner **...\bin\Debug\net5.0**), verzichtet aber während des Programmlaufs auf die Anzeige diagnostischer Informationen.

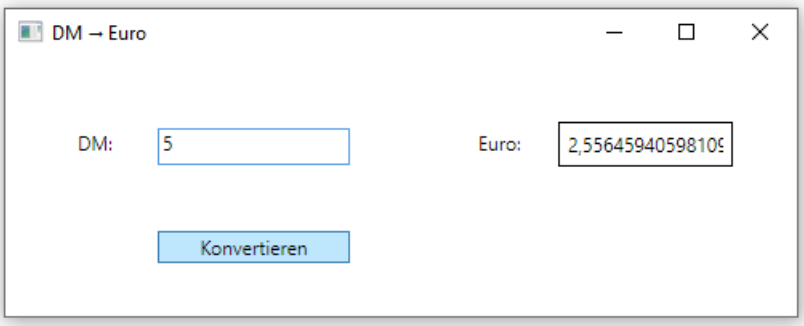
Mit der bei auslieferungsbereiten Programmen sinnvollen Release-Konfiguration werden wir uns im Abschnitt 3.3.4.4 beschäftigen.

3.3.4 Eine erste GUI-Anwendung

Dieser Abschnitt bietet zwecks Steigerung Ihrer Motivation einen Vorausblick auf die Erstellung von Programmen mit GUI-Bedienung (*Graphical User Interface*) unter Verwendung von RAD-Werkzeugen (***R**apid **A**pplication **D**evelopment*).

3.3.4.1 Projekt für das .NET Framework anlegen

Es entsteht ein Währungskonverter mit grafischer Bedienoberfläche, der DM-Beträge in Euro-Beträge wandeln kann:[2]



Wir entwickeln zunächst für das .NET Framework und behandeln später im Abschnitt 3.3.4.5 die (relativ wenigen) Abweichungen bei der Entwicklung desselben Programms für .NET 5.0.

Nach dem Menübefehl

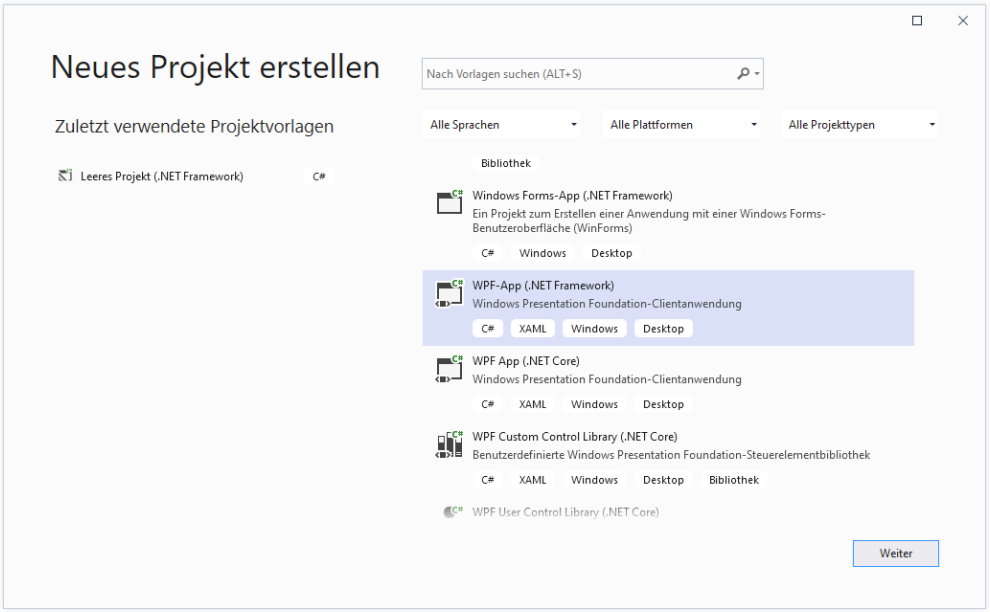
Datei > Neu > Projekt

wählen wir im folgenden Dialog

---

[1] PDB steht für *Program Data Base*.

[2] Benötigt wird ein solches Programm z. B. noch bei der Auszahlung von Erbschaftsbeträgen aus Testamenten, welche in der DM-Ära verfasst worden sind.



die Projektvorlage **WPF-App (.NET Framework)**, sodass unsere Anwendung die GUI-Bibliothek *Windows Presentation Foundation* (WPF) verwendet, die im Kurs gegenüber ...

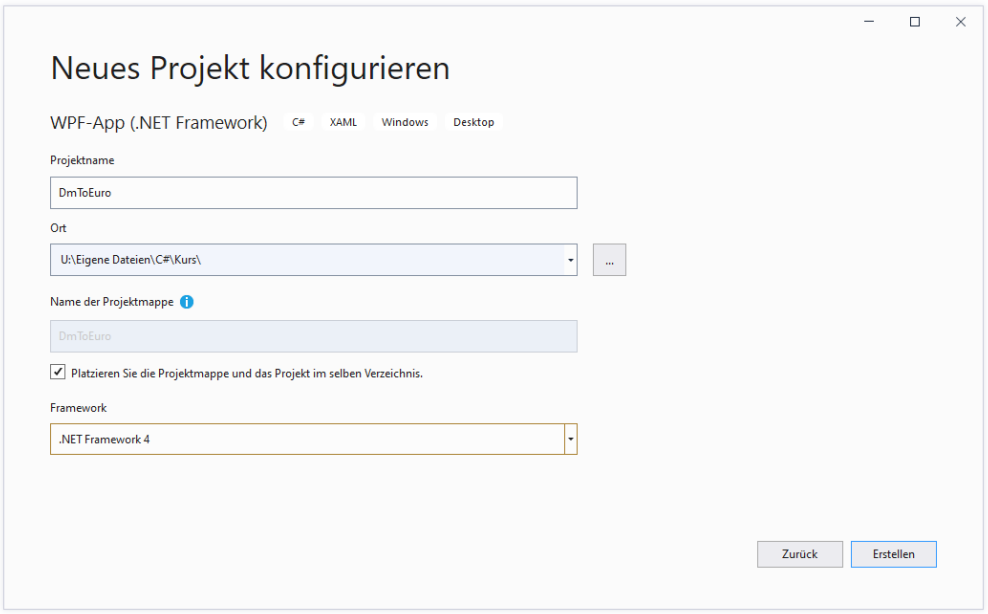
- der älteren GUI-Bibliothek *WinForms*
- und gegenüber der von Microsoft wenig glücklich und wenig erfolgreich als Zukunft der Windows-Programmierung forcierten UWP-Technologie[1]

bevorzugt wird.

Als **Name** für das Projekt eignet sich z. B. `DmToEuro`, und als **Ort** (übergeordnetes Verzeichnis) zum neu anzulegenden Projektordner können Sie auf einem ZIMK-Pool-PC der Universität Trier analog zu Abschnitt 3.3.3 z. B.

U:\Eigene Dateien\C#\Kurs

verwenden.



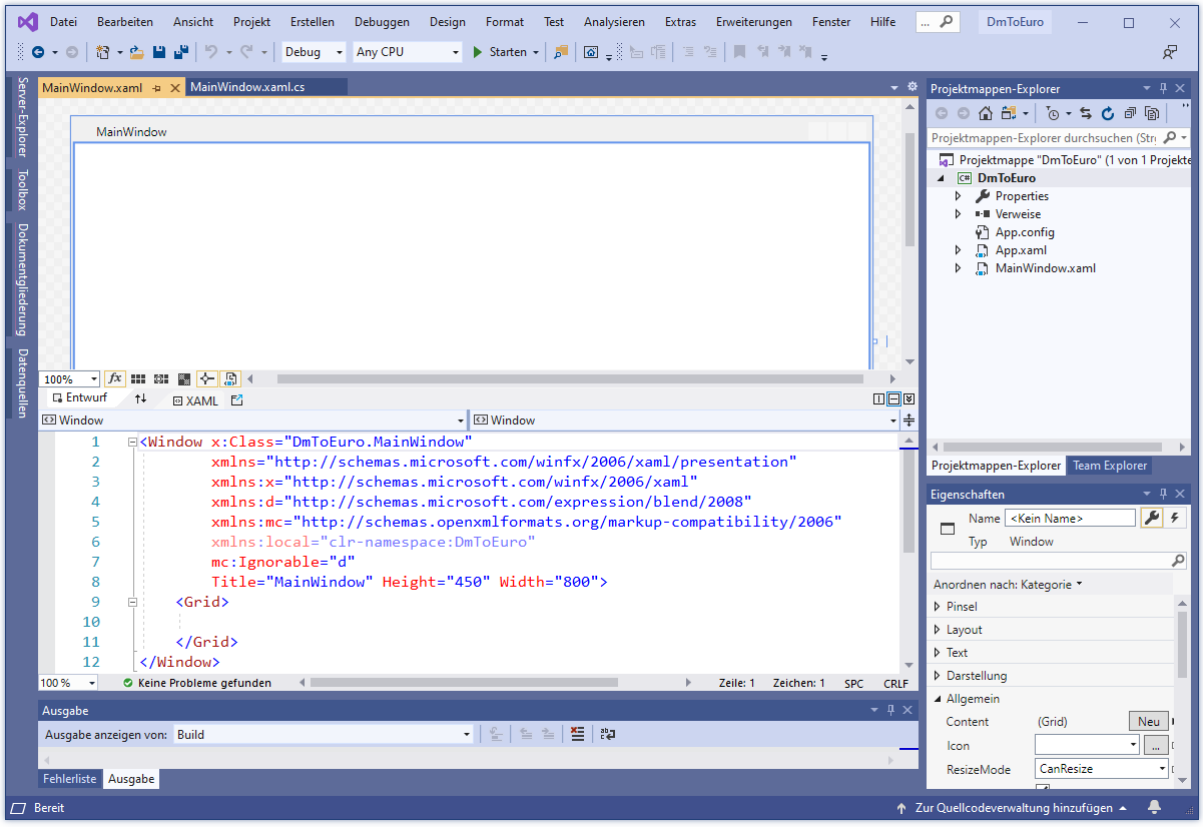
[1] https://www.thurrott.com/dev/206351/microsoft-confirms-uwp-is-not-the-future-of-windows-apps

Indem wir lediglich ein .NET - **Framework** in der Version 4 voraussetzen, wird das entstehende Programm auch unter Windows 8.1 und folglich auf jedem Windows-Rechner mit einer noch von Microsoft mit Sicherheits-Updates unterstützten Windows-Version laufen.

Wie schon im Abschnitt 3.3.3 begründet wurde, ist es bei unseren Übungsprojekten meist sinnvoll, **Projektmappe und Projekt im selben Verzeichnis** abzulegen.

Wenige Sekunden nach einem Mausklick auf **Erstellen** präsentiert das Visual Studio im **Projektmappen-Explorer** am rechten Fensterrand eine Baumansicht zur Projektmappenverwaltung. Hier erscheint ein Eintrag für jedes Projekt in der potentiell aus mehreren Projekten bestehenden Projektmappe. Im aktuellen, für den Kurs typischen Beispiel befindet sich nur *ein* Projekt in der Mappe, und beide tragen denselben Namen. Zu jedem Projekt werden u. a. die Quellcode-Dateien zu den Klassendefinitionen sowie die Verweise auf benötigte Bibliotheks-Assemblies (siehe Abschnitt 3.3.6.1) aufgelistet. Wir werden die Bestandteile bei passender Gelegenheit behandeln.

Auf der linken Seite präsentiert der Fenster- bzw. WPF-Designer einen Rohling für das Fenster der Anwendung:

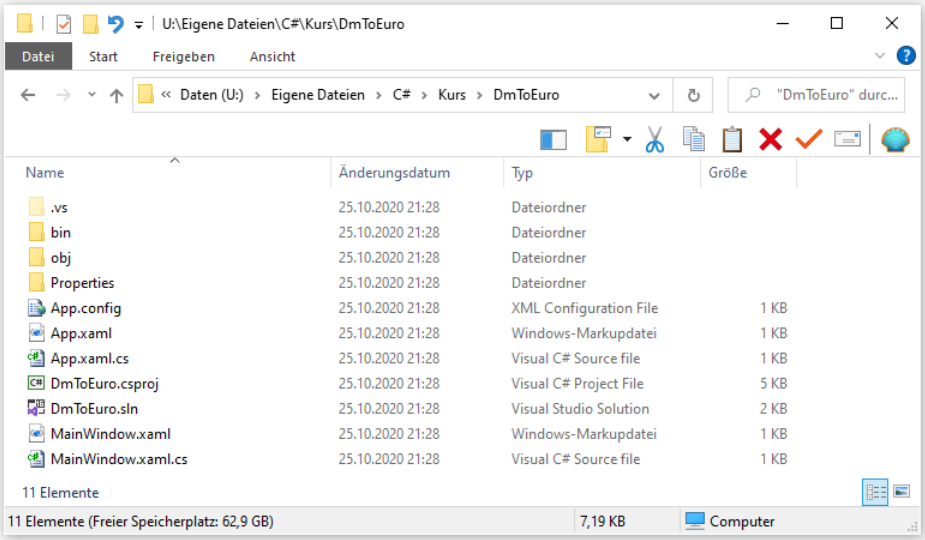


Falls Sie unter dem **Projektmappen-Explorer** kein **Eigenschaften**-Fenster sehen sollten, schalten Sie es bitte mit dem Menübefehl

Ansicht > Eigenschaftenfenster

oder mit der Funktionstaste **F4** ein.

Weil wir uns *gegen* einen Projektunterordner im Projektmappenordner entschieden haben, resultieren im Beispiel folgende Ordner und Dateien:



Bei Verwendung der im Dialog für neue Projekte voreingestellten Programmiersprache C# besitzt eine Projektdatei die Namenserweiterung **.csproj**, sodass im Beispiel resultiert:

U:\Eigene Dateien\C#\Kurs\DmToEuro\DmToEuro.csproj

Bei einer Projektmappendatei wird die Namenserweiterung **.sln** (Abkürzung für *Solution*) verwendet, sodass im Beispiel resultiert:

U:\Eigene Dateien\C#\Kurs\DmToEuro\DmToEuro.sln

Um ein Projekt über den Windows-Explorer zu öffnen, setzt man einen Doppelklick auf die Projekt- oder auf die Projektmappendatei.

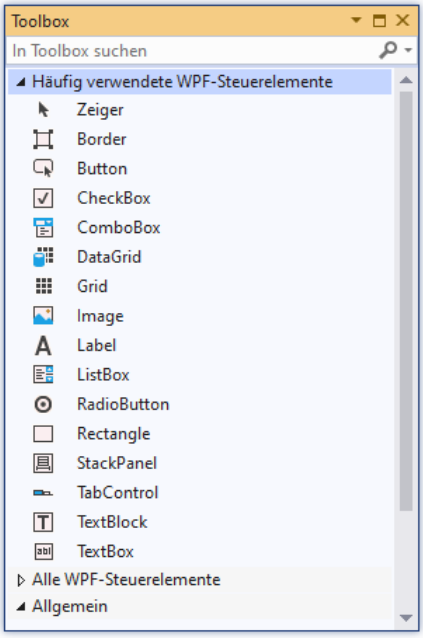
3.3.4.2 Bedienoberfläche entwerfen

Wir machen uns nun daran, das Anwendungsfenster unseres Währungskonverters mit den benötigten Bedienelementen (Steuerelementen, Controls) auszustatten. Während wir mit dem WPF-Designer fast wie mit einem Grafikprogramm arbeiten, erstellt und pflegt dieser Assistent eine Deklaration der Benutzeroberfläche in der *EXtensible Application Markup Language* (XAML). Mit zunehmendem Wissen über die XAML-Beschreibungssprache werden wir später unsere Abhängigkeit vom Assistenten reduzieren. In der aktuellen Lernphase bearbeiten wir den XAML-Code nur indirekt mit Hilfe des WPF-Designers.

Die Bedienelemente können aus dem **Toolbox**-Fenster per Drag & Drop (Ziehen & Ablegen) übernommen werden. Öffnen Sie dieses Fenster mit dem Menübefehl

Ansicht > Toolbox

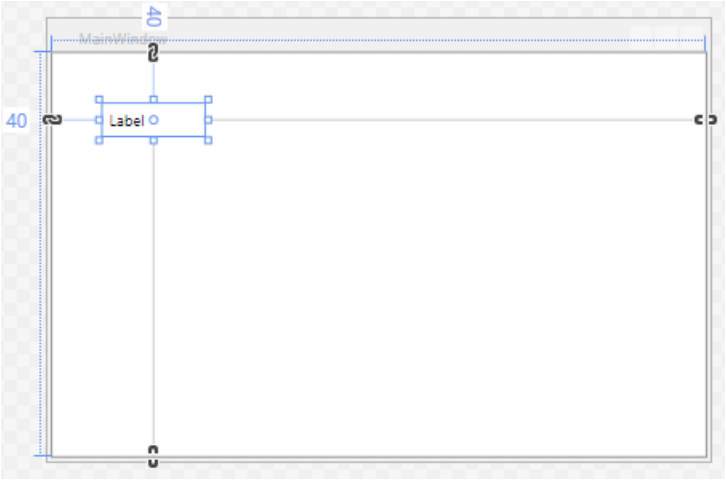
oder durch einen Mausklick auf die **Toolbox**-Schaltfläche am linken Fensterrand, und erweitern Sie nötigenfalls die Liste mit den **Häufig verwendeten WPF-Steuerelementen**:



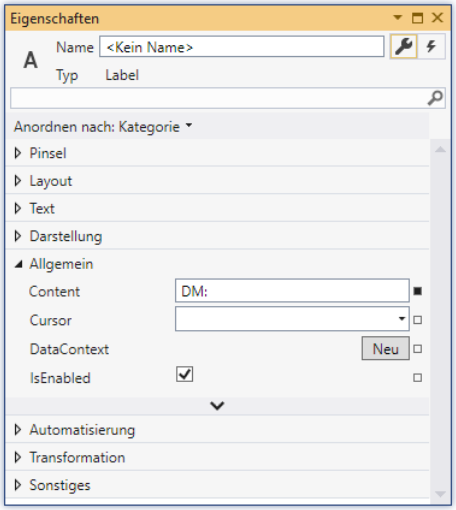
Erstellen Sie ein **Label**-Objekt (die Bezeichnung *Objekt* ist durchaus im Sinn von Kapitel 1 gemeint) auf dem Formular, ...

- entweder per Doppelklick auf den **Toolbox**-Eintrag **Label**
- oder per Drag & Drop (Ziehen und Ablegen), indem Sie einen linken Mausklick auf den **Toolbox**-Eintrag **Label** setzen, die Maus dann mit gedrückter linker Taste zum Ziel bewegen und dort die Taste wieder loslassen.

Durch die Wahl einer passenden Mauszeigerposition erhält man das Werkzeug zum Bewegen von Objekten oder das Werkzeug ⟷ zur horizontalen Größenveränderung. Damit lässt sich der folgende Zustand herstellen:

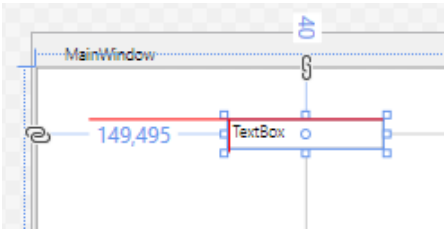


Ändern Sie die Beschriftung des **Label**-Objekts, indem Sie bei markiertem **Label**-Objekt im **Eigenschaften**-Fenster (unten rechts) einen passenden Wert für die **Allgemein**-Eigenschaft **Content** wählen:

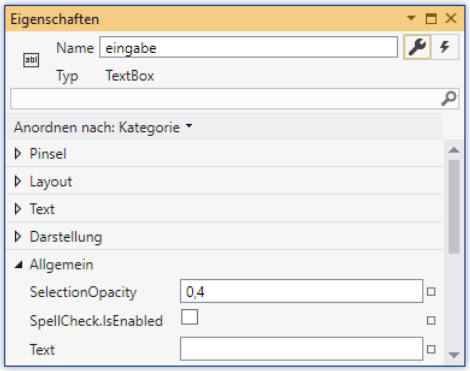


Dabei ist *Eigenschaft* durchaus im Sinn von Abschnitt 1.2 gemeint.

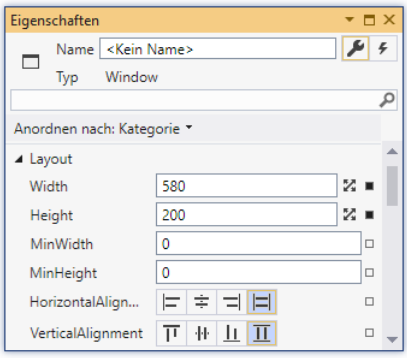
Setzen Sie ein Objekt der Klasse **TextBox** rechts neben das **Label**-Objekt, wobei die Entwicklungsumgebung die Anpassung von Position und Größe durch Hilfslinien erleichtert. In das **TextBox**-Steuerelement sollen die Benutzer unseres Programms den zu konvertierenden DM-Betrag eintragen:



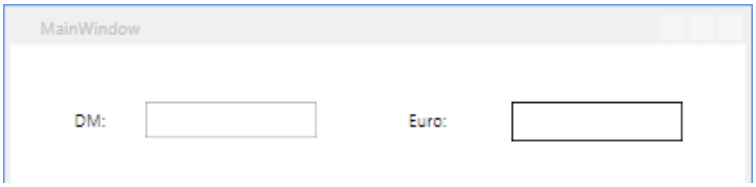
Der initial vorhandene Text stört und sollte gelöscht werden (Eigenschaft **Text**) in der Kategorie **Allgemein** des **Eigenschaften**-Fensters). Weil das **TextBox**-Objekt in der gleich zu erstellenden Konvertierungsmethode angesprochen werden muss, erhält es einen Namen (z. B.: `eingabe`):



Mittlerweile hat sich gezeigt, dass die vorgegebene Fensterbreite überdimensioniert ist. Markieren Sie das gesamte Fenster (oder das **Window**-Element im XAML-Code), nicht etwa den **Grid**-Layoutcontainer (das **Grid**-Element im XAML-Code). Tragen Sie im **Eigenschaften**-Fenster passende Werte für die **Layout**-Eigenschaften **Width** und **Height** ein, z. B.:



Setzen Sie zur Ausgabe des Euro-Betrags zwei weitere **Label**-Objekte auf das Fenster:

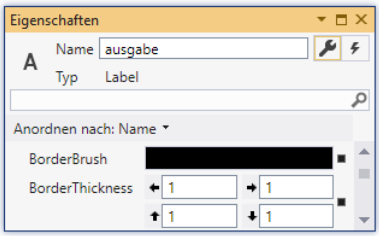


Während das erste Label wie das DM-Pendant zur Beschriftung dient, soll das zweite den Ergebnisbetrag anzeigen, also beim Programmstart leer sein. Passen Sie also die **Content**-Eigenschaftsausprägungen der Objekte passend an.

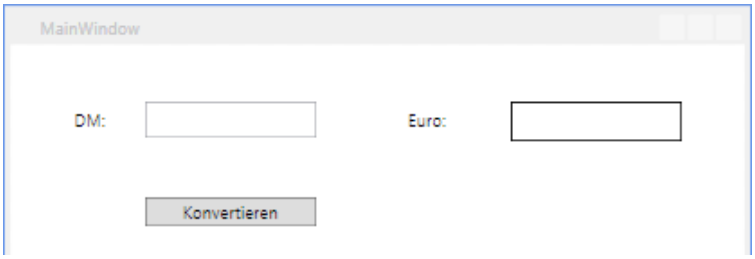
Um dem zweiten Label auch initial einen optischen Auftritt zu verschaffen, sollten Sie einen Rand anzeigen lassen. Dazu ist über die Eigenschaft **BorderBrush** eine Randfarbe und über die Eigenschaft **BorderThickness** eine Randstärke geeignet festzulegen. Um eine Eigenschaft zu lokalisieren, können Sie ....

- die zugehörige Kategorie raten und aufklappen,
- ein **Anordnen** der Eigenschaften nach dem **Namen** veranlassen,
- nach der Eigenschaft suchen.

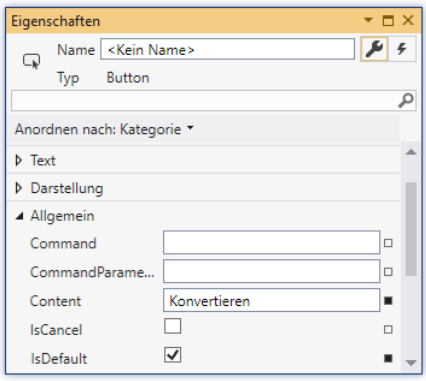
Vor allem benötigt das zur Anzeige des resultierenden Euro-Betrags vorgesehene **Label**-Objekt einen Namen (z. B. `ausgabe`), damit es in der gleich zu erstellenden Konvertierungsmethode angesprochen werden kann:



Setzen Sie nun noch ein **Button**-Objekt auf das Fenster, damit die Benutzer per Mausklick die Konvertierung des zuvor eingegebenen DM-Betrags anfordern können:



Auch beim **Button**-Objekt sorgt man über die **Content**-Eigenschaft für eine passende Beschriftung. Wenn die Eigenschaft **IsDefault** in der Kategorie **Allgemein** per Kontrollkästchen den Wert **true** erhält, dann kann der Schalter im Programm auch per **Enter**-Taste ausgelöst werden:

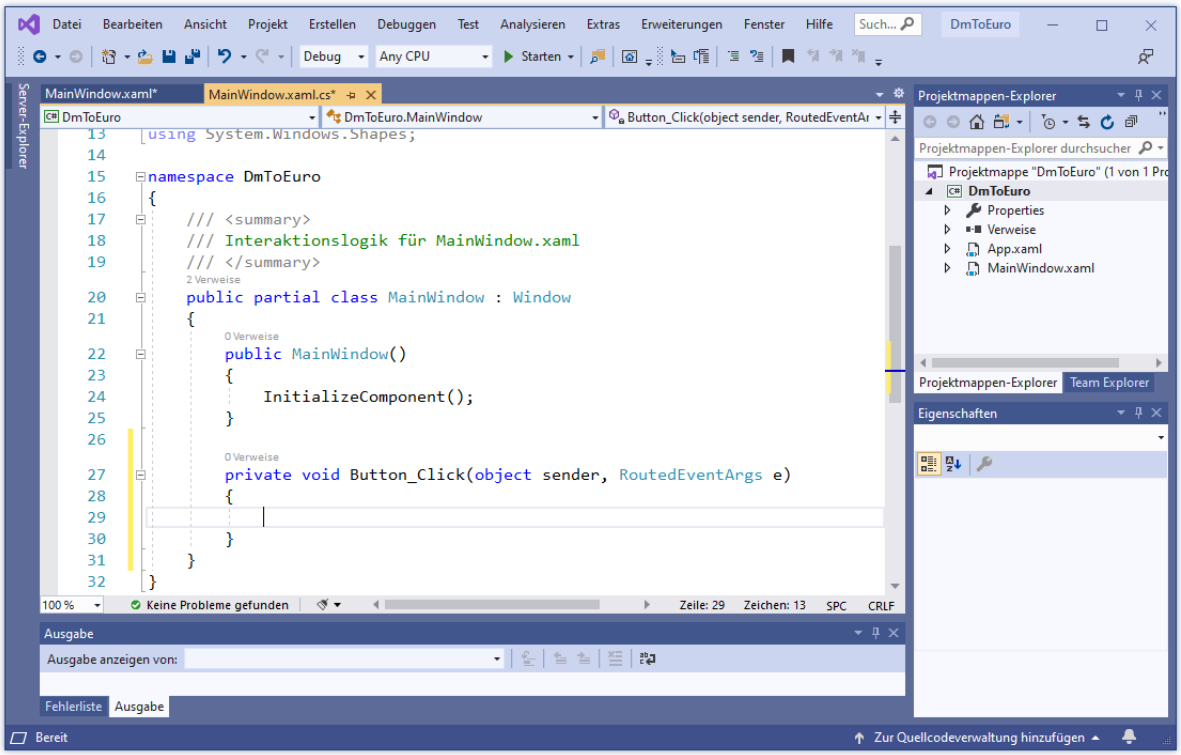


3.3.4.3 Behandlungsmethode zum Click-Ereignis des Befehlsschalters erstellen

Nun ist die Bedienoberfläche des entstehenden Programms in einem akzeptablen Zustand, und wir können uns um die Funktionalität kümmern. Dazu ist eine Methode zu erstellen, die bei einem Mausklick auf den Befehlsschalter ausgeführt werden soll. Sie hat folgende Aufgaben:

- Beim **TextBox**-Objekt die aktuell eingetragene Zeichenfolge erfragen,
- diese Zeichenfolge nach Möglichkeit in eine Zahl wandeln,
- das Ergebnis durch den DM-Euro - Umrechnungsfaktor 1,95583 dividieren,
- den Euro-Betrag in eine Zeichenfolge wandeln und das umrahmte **Label**-Objekt auffordern, diese Zeichenfolge anzuzeigen.

Sobald Sie einen Doppelklick auf das **Button**-Objekt setzen, öffnet das Visual Studio den Quellcode-Editor und fügt dort eine Methode namens `Button_Click()` ein, die im fertigen Programm nach jedem Mausklick auf den Schalter ausgeführt wird:

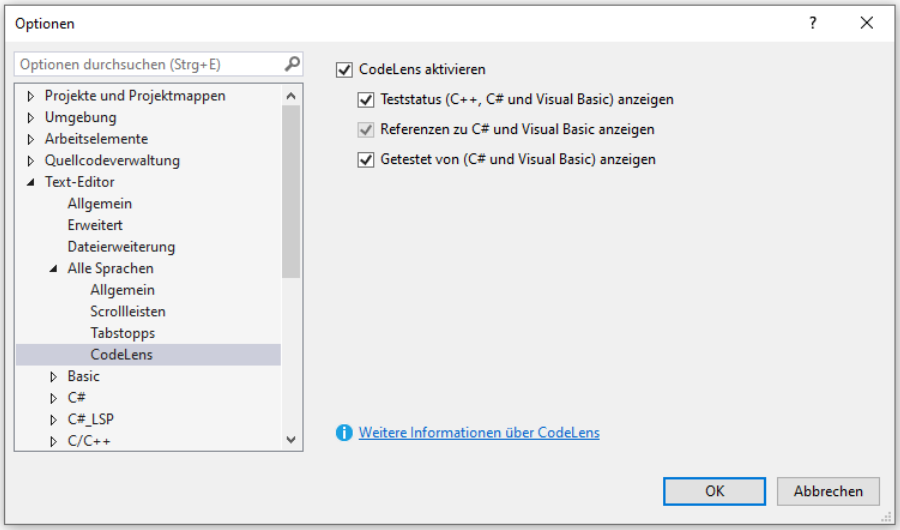


Nun wird auch erkennbar, dass wir gerade dabei sind, mit Assistentenhilfe eine Klasse namens `MainWindow` zu definieren. Für das Projekt hat die Entwicklungsumgebung den Namensraum `DmToEuro` definiert, wobei die Bezeichnung mit dem Projektnamen übereinstimmt.

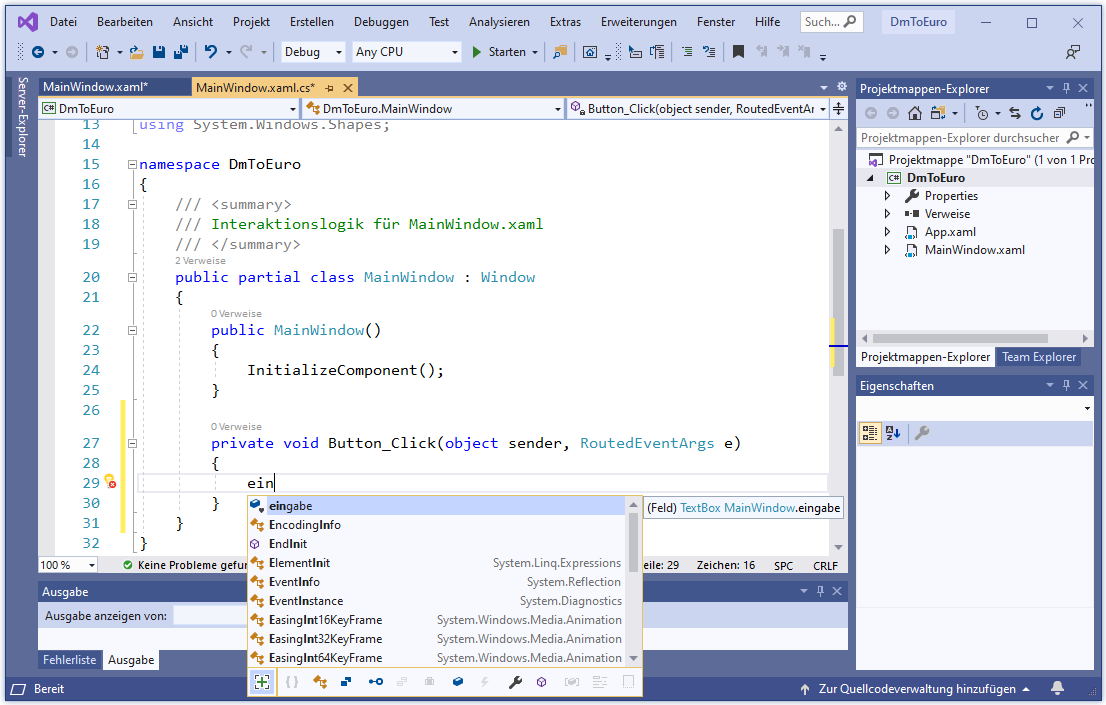
Vor den Definitionsköpfen für die Klasse bzw. für die Methoden hat die per Voreinstellung aktive CodeLens-Funktion die Anzahl der im Quellcode des Projekts vorhandenen Verweise eingetragen. Wenn das stört, kann die Funktion nach

Extras > Optionen > Text-Editor > Alle Sprachen > CodeLens

abgeschaltet werden:



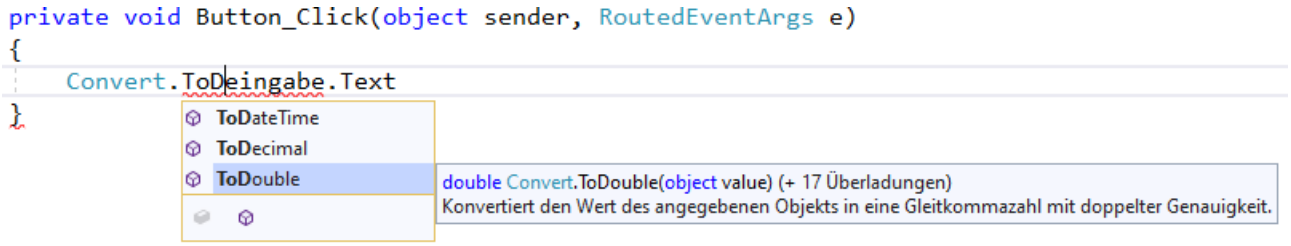
Sobald Sie damit beginnen, im Rumpf der Methode `Button_Click()` den Namen des **TextBox**-Objekts einzutippen, um sich über seine **Text**-Eigenschaft nach der vom Benutzer eingetippten Zeichenfolge zu erkundigen, erahnt das Visual Studio Ihre Absicht und bietet mögliche Fortsetzungen der Anweisung an:



Akzeptieren Sie den Vorschlag `eingabe` der sogenannten *IntelliSense*-Technik per Tabulatortaste, und setzen Sie einen Punkt hinter den Objektnamen. Nun erscheint eine Liste mit den Methoden

und Eigenschaften des Objekts. Das **TextBox**-Objekt soll seine **Text**-Eigenschaft abliefern. Wählen Sie diese Eigenschaft aus der Liste (z. B. per Doppelklick).

Wir verwenden die erfragte Zeichenfolge als Parameter (Argument) in einem Aufruf der Methode **ToDouble()**, welche die Klasse **Convert** beherrscht. Bei der erforderlichen Erweiterung der gerade entstehenden C# - Anweisung bewährt sich wiederum die IntelliSense-Technik unserer Entwicklungsumgebung:



Nach einem Doppelklick auf **ToDouble** müssen wir lediglich die runden Klammern um den Parameter ergänzen, um den Methodenaufruf zu komplettieren.

Der Quellcode-Editor unserer Entwicklungsumgebung bietet außerdem …

- farbliche Unterscheidung verschiedener Sprachestandteile
- automatische Quellcode-Formatierung (z. B. bei Einrückungen)
- automatische Syntaxprüfung, z. B.:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Convert.ToDouble(eingabe.Text)
}
```

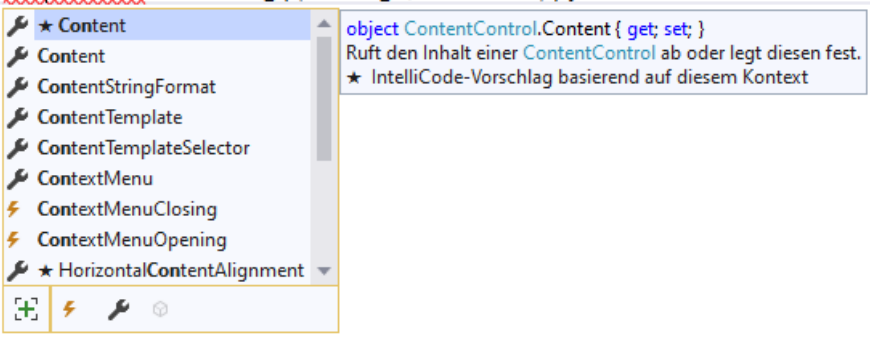
  Wir haben mittlerweile einen Methodenaufruf erstellt, der aber keine vollständige Anweisung ist, was unsere Entwicklungsumgebung durch rotes Unterschlängeln der mutmaßlichen Fehlerstelle reklamiert. Wir ergänzen das an dieser Stelle erforderliche Semikolon.

Um bei den weiteren Verarbeitungsschritten einen überlangen Ausdruck zu vermeiden, benutzen wir zur lokalen Zwischenspeicherung des DM-Betrags in der Methode `Button_Click()` eine lokale Variable namens `betrag` vom Typ **double** (siehe Abschnitt 4.3.4):

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    double betrag = Convert.ToDouble(eingabe.Text);
}
```

Die von **ToDouble()** als Rückgabe gelieferte und mittlerweile in der Variablen `betrag` gespeicherte Zahl muss durch 1,95583 dividiert werden. Den resultierenden Euro-Betrag lassen wir durch die Methode **ToString()** der Klasse **Convert** in eine Zeichenfolge (ein Objekt der Klasse **String**) wandeln. Das Endergebnis muss dem **Label**-Objekt `ausgabe` als neuer Wert der Eigenschaft **Content** übergegeben werden:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    double betrag = Convert.ToDouble(eingabe.Text);
    ausgabe.ConConvert.ToString((betrag / 1.95583));
}
```



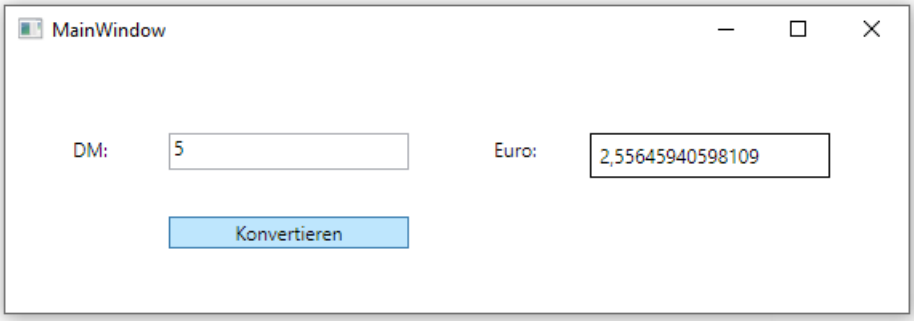
So sieht die fertige Methode `Button_Click()` aus:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
     double betrag = Convert.ToDouble(eingabe.Text);
     ausgabe.Content = Convert.ToString(betrag / 1.95583);
}
```

Bislang wurden Methoden(aufrufe) als Nachrichten in der Kommunikation zwischen Klassen und Objekten dargestellt (siehe z. B. Abschnitt 1.2). Die Methode `Button_Click()` wird jedoch kaum im Quellcode direkt aufgerufen. Es handelt sich um eine sogenannte *Rückrufmethode* (engl.: *callback method*) im Rahmen der Ereignisbehandlung im .NET - Framework. Sie wird von der CLR aufgerufen, wenn ein Anwender das Klickereignis des Befehlsschalters ausgelöst hat.

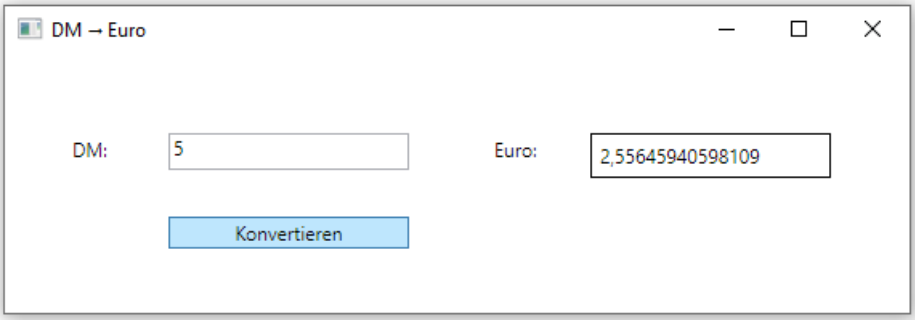
3.3.4.4 Testen und verbessern

Wir lassen das Programm über die Tastenkombination **Strg+F5** übersetzen und ausführen:



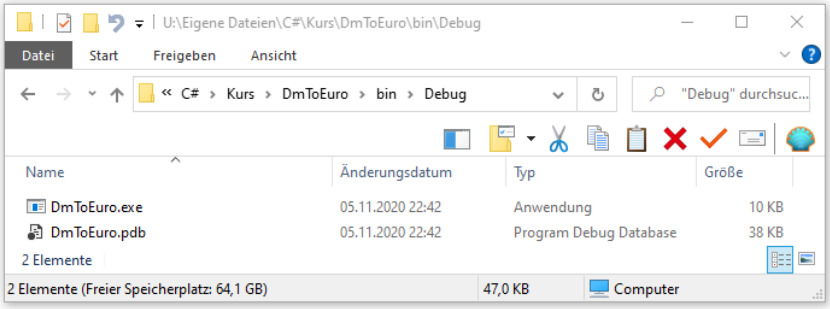
Spontan fällt als Mangel der wenig informative Fenstertitel auf. Markieren Sie bei aktivem Fenster-Designer das Hauptfenster und ändern Sie per Eigenschaftenfenster seine **Title**-Eigenschaft (Kategorie **Allgemein**):[1]

[1] Falls jemand wissen möchte, wie man den rechtsgerichteten Pfeil eintippt: Tastenkombination **Alt+26** auf dem Ziffernblock der Tastatur.

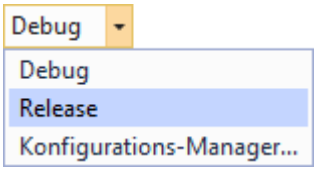


In der Endversion des Programms werden wir einen benutzerfreundlich gerundeten Euro-Betrag präsentieren.

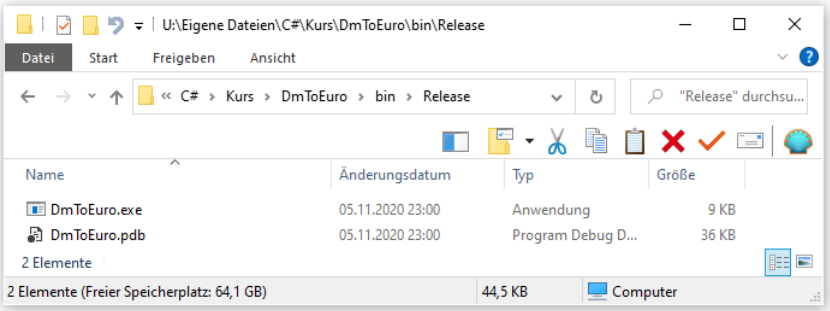
Ist die **Debug**-Konfiguration aktiv, erstellt der Compiler ein gut testbares, aber nicht leistungsoptimiertes Assembly und legt es im Projekt-Unterordner **…\bin\Debug** ab:



Zur Änderung der Konfiguration steht in der Symbolleiste **Standard** ein Bedienelement bereit:



Ist die **Release**-Konfiguration aktiv, erstellt der Compiler (z. B. nach dem Programmstart über die Tastenkombination **Strg+F5**) ein leistungsoptimiertes, aber weniger gut testbares Assembly und legt es im Projekt-Unterordner **…\bin\Release** ab:



Neben dem ausführbaren Assembly **DmToEuro.exe** erstellt der Compiler noch weitere Dateien:

- **DmToEuro.exe.config**
  Bei Bedarf, d.h. in Abhängigkeit von den gewählten Projekt-Eigenschaften legt das Visual Studio eine Anwendungskonfigurationsdatei an, z. B. bei der Wahl des .NET - Kompatibilitätsniveaus 4.5.1:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5.1" />
    </startup>
</configuration>
```

  Im Element **supportedRuntime** informiert ...
  - das Attribut **version** darüber, welche CLR-Version unterstützt wird. Bei modernen Assemblies ist hier stets die Versionsangabe 4.0 anzutreffen.
  - das Attribut **sku** (*stock-keeping unit*) darüber, für welche .NET Framework - Version die Anwendung erstellt wurde.

  Wenn eine **config**-Datei mit dem Element **supportedRuntime** vorhanden ist, und auf dem aktuellen Rechner nicht mindestens die im **sku**-Attribut genannte .NET Framework - Version installiert ist, dann verhindert Windows den Programmstart mit einer Fehlermeldung, die dazu auffordert, die erforderliche Runtime-Version zu installieren. Fehlt die **config**-Datei oder das Element **supportedRuntime**, dann wird das Programm gestartet und stürzt ab, sobald erstmals eine Framework-Funktionalität benötigt wird, die auf dem lokalen Rechner nicht vorhanden ist. Je nach der tatsächlich verwendeten Framework-Funktionalität läuft das Programm eventuell aber auch korrekt.
  Im aktuellen Beispiel wird lediglich das .NET-Framework 4 vorausgesetzt, und das Visual Studio erstellt keine Anwendungskonfigurationsdatei.

- **DmToEuro.pdb**
  In der **pdb**-Datei (*Program Debug Database*) stecken Informationen, die das Debugging (die Fehlersuche) und das Profiling (die Leistungsoptimierung) unterstützen. Zwar findet in einem Projekt die Fehlersuche hauptsächlich mit der Debug-Konfiguration statt, doch kommt die Fehlersuche auch bei der durch Leistungsoptimierungen potentiell fehleranfälligeren Release-Konfiguration in Frage. Beim Profiling ist die Release-Konfiguration sogar der attraktivere Anwendungsfall. Insgesamt gibt es also gute Gründe dafür, für beide Konfigurationen eine **pdb**-Datei zu erstellen.[1] Per Voreinstellung erstellt das Visual Studio für die Debug-Konfiguration eine umfangreichere **pdb**-Datei.

In jedem Fall ist das Assembly **DmToEuro.exe** einsatzfähig und kann auf jedem Windows-Rechner ausgeführt werden. Weil sich die benötigten Bibliotheks-Assemblies im GAC (Global Assembly Cache) befinden und keine Hilfsdateien erforderlich sind, muss zur „Installation“ des Programms lediglich die EXE-Datei an ihren Einsatzort kopiert werden. Verzichtet man auf die Verteilung der **pdb**-Datei, dann kann auf dem Rechner eines Kunden mit havariertem Programm schlechter nach der Fehlerursache gesucht werden. Wenn allerdings eine Kooperation des Kunden bei der Fehlersuche (z. B. per Remote-Debugging) kaum zu erwarten ist, dann kann auf die Verteilung der **pdb**-Datei verzichtet werden.

Soll eine Anwendung nur erstellt (aber nicht ausgeführt) werden, wählt man die Tastenkombination **Strg-Umschalt+B** oder den Menübefehl

Erstellen > Projektmappe erstellen

Trotz der guten Erfahrungen mit der GUI-Programmierung werden wir uns die Grundbegriffe der Programmierung in den nächsten Kapiteln im Rahmen von möglichst einfachen Konsolenprojekten erarbeiten.

---

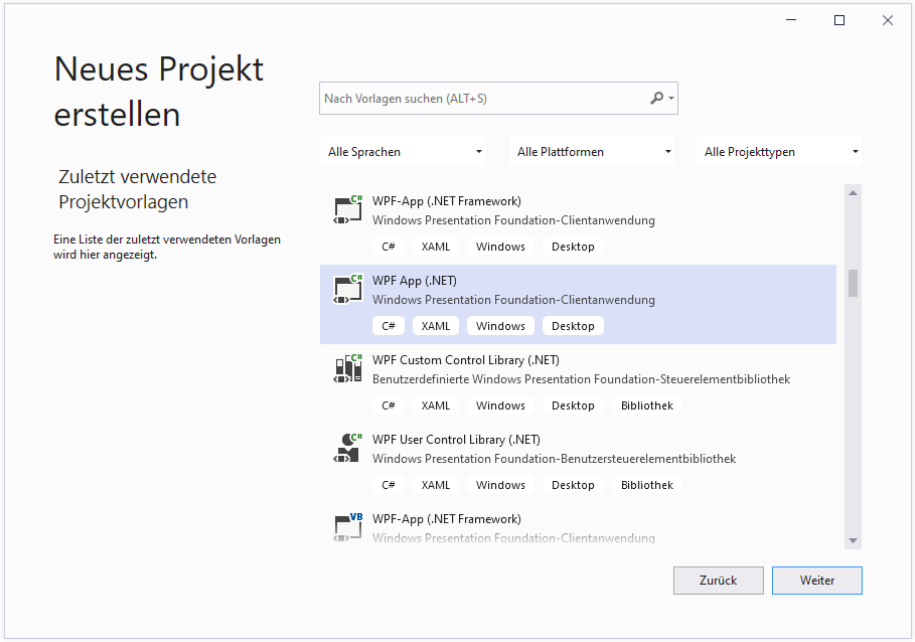
[1] https://stackoverflow.com/questions/5457095/release-generating-pdb-files-why

3.3.4.5 Entwicklung für .NET 5.0

Bei der Entwicklung des Währungskonverters für .NET 5.0 wählen wird nach dem Menübefehl

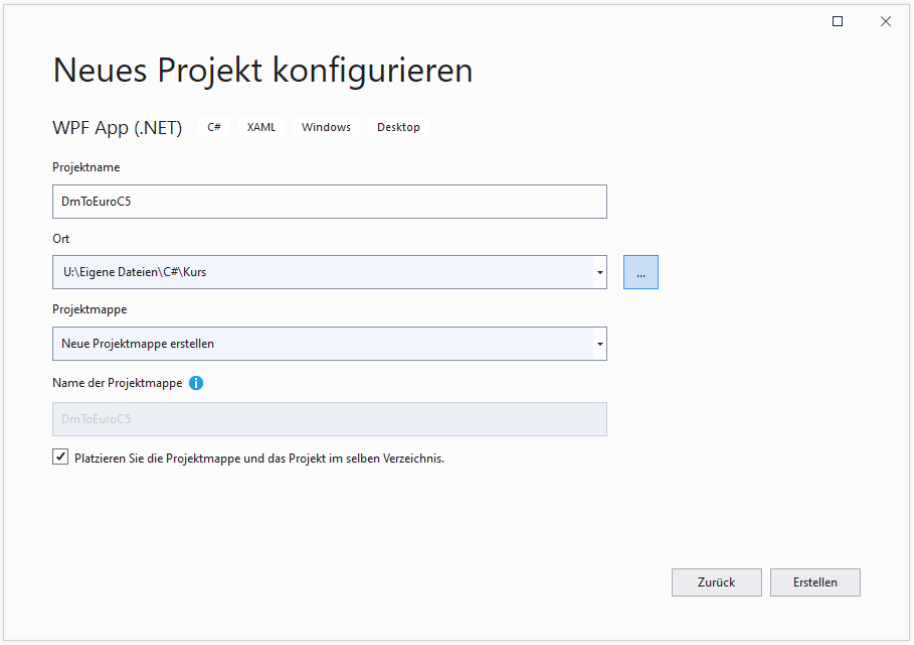
Datei > Neu > Projekt

im folgenden Dialog

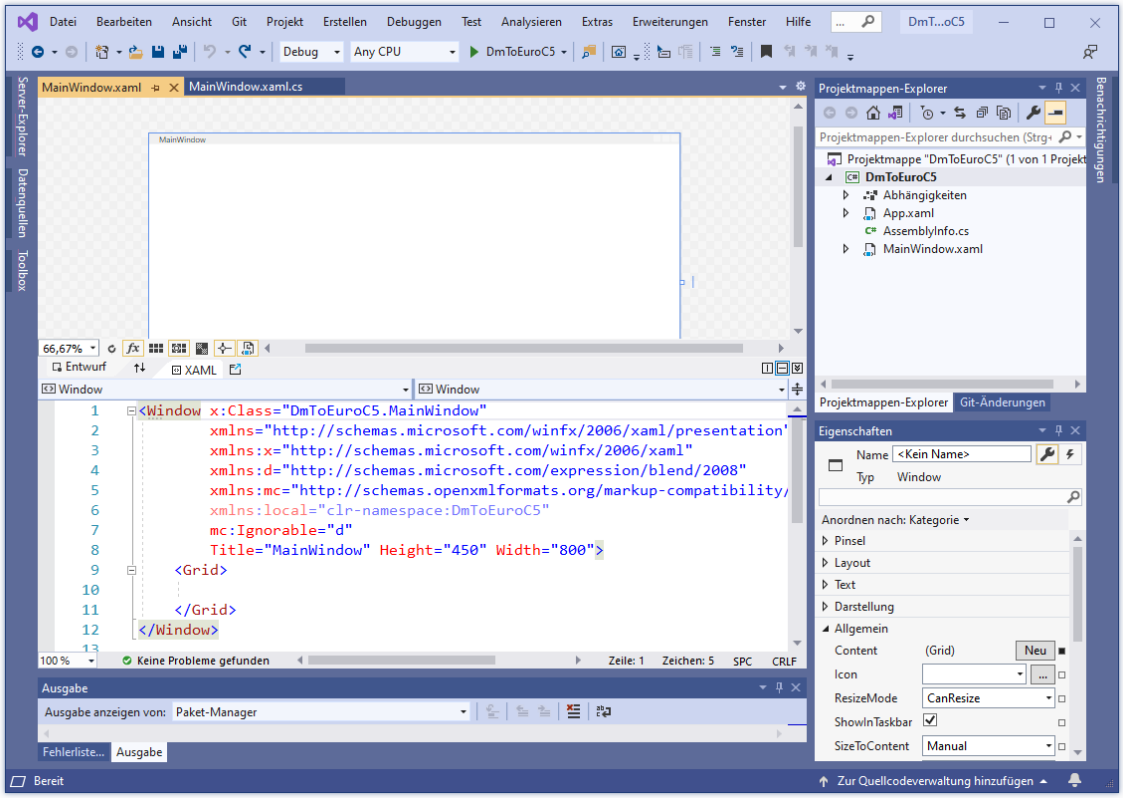


die Projektvorlage **WPF-App (.NET)**. Als **Projektname** eignet sich z. B. `DmToEuroC5`, und als **Ort** (übergeordnetes Verzeichnis) zum neu anzulegenden Projektordner bewährt sich auf einem ZIMK-Pool-PC der Universität Trier z. B.

U:\Eigene Dateien\C#\Kurs



Am Arbeitsplatz der Entwicklungsumgebung sind kaum Unterschiede im Vergleich zum .NET Framework - Projekt festzustellen:



Im **Projektmappen-Explorer** ist (ohne Relevanz für das aktuelle Projekt) der Projektknoten **Verweise** ersetzt durch den Knoten **Abhängigkeiten** (vgl. Abschnitt 3.3.6.1).

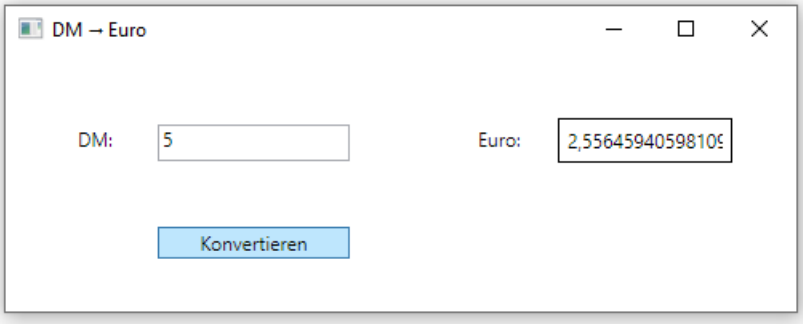
Nach

Projekt > Eigenschaften

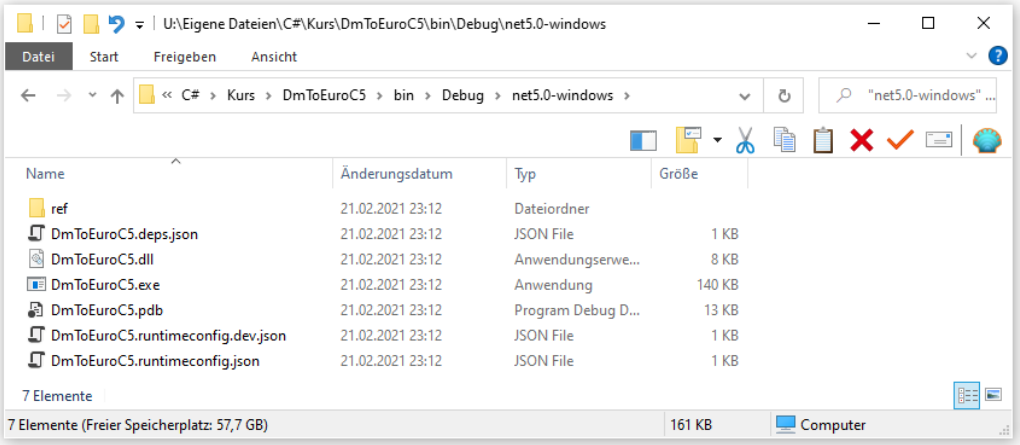
ersetzen wir das voreingestellte **.NET Core 3.1** als **Zielframework** durch **.NET 5.0**.

Beim Design der Bedienoberfläche und beim Verfassen der Methode, die bei einem Mausklick auf den Befehlsschalter ausgeführt werden soll, kann man exakt genauso vorgehen wie im .NET Framework - Projekt.

Wir lassen das Programm über die Tastenkombination **Strg+F5** erstellen und ausführen:

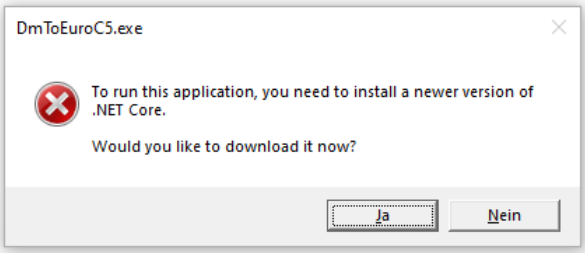


Der Ausgabeordner für die Debug-Konfiguration



enthält andere Dateien als das Gegenstück aus dem .NET - Framework - Projekt:

- Die Anwendungskonfigurationsdatei **DmToEuro.exe.config** ist ersetzt durch etliche Konfigurationsdateien im **JSON**-Format (*JavaScript Object Notation*).
- Das erstellte Assembly mit dem ausführbaren Programm befindet sich in der Datei **DmToEuroC5.dll**, während die Datei **DmToEuroC5.exe** ein unter Windows benötigter Starthelfer ist (vgl. Abschnitt 3.3.3.2).
- Zum Starten der Anwendung setzt man unter Windows einen Doppelklick auf die **EXE**-Datei. Der erwünschte Effekt stellt sich aber nur dann ein, wenn zuvor die benötigte .NET 5.0 - Laufzeitumgebung installiert worden ist. Anderenfalls erscheint beim Startversuch die folgende Fehlermeldung:



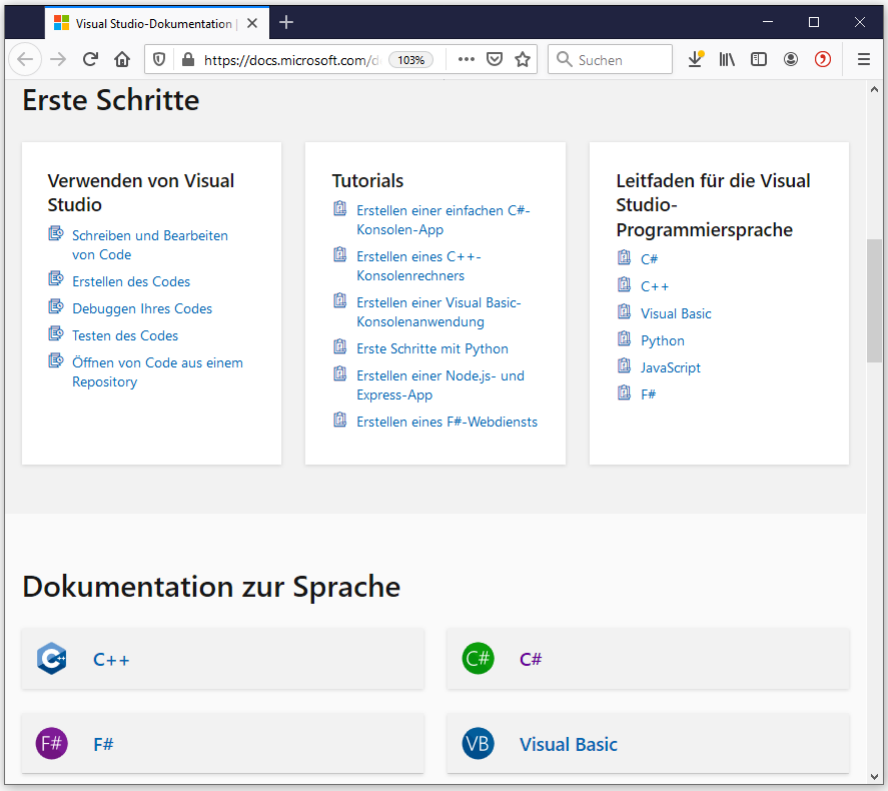
- Die beiden Projekte (bzw. .NET - Implementationen) unterschieden sich *nicht* bei der Verwendung einer **pdb**-Datei (*Program Debug Database*) mit Informationen, die das Debugging (die Fehlersuche) und das Profiling (die Leistungsoptimierung) unterstützen.

3.3.5 BCL-Dokumentation und andere Hilfeinhalte

Die über den Menübefehl

Hilfe > Hilfe anzeigen

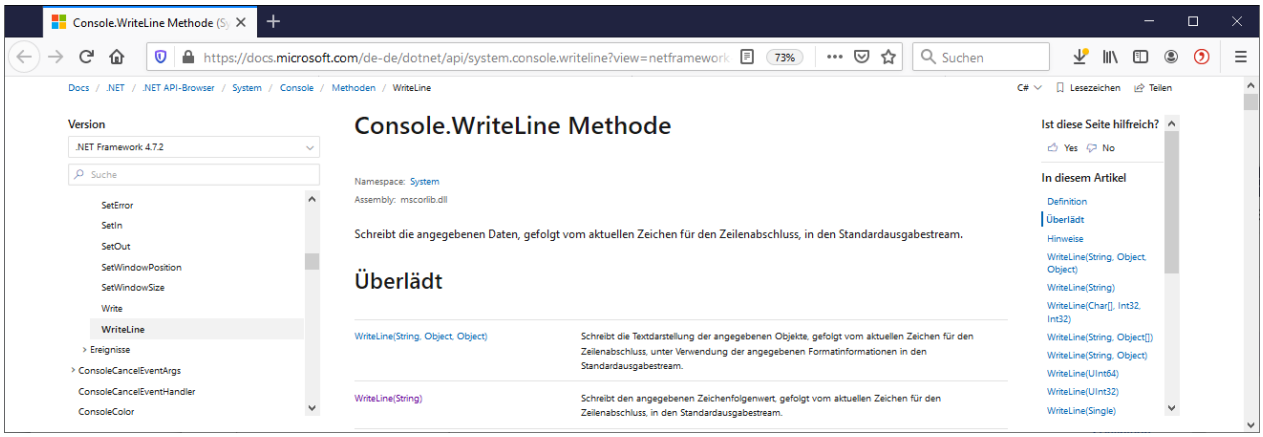
angezeigten Hilfeinhalte werden per Voreinstellung aus dem Internet bezogen und von einem Browser angezeigt, z. B.:



Wir benötigen die Hilfefunktion vor allem dazu, um Informationen über Bestandteile der BCL nachzuschlagen. Zu einem markierten (die Einfügemarke enthaltenden) C# - Schlüsselwort oder BCL-Bezeichner im Quellcode-Editor der Entwicklungsumgebung erreicht man die zugehörige Dokumentation besonders bequem über die Funktionstaste **F1**. In der folgenden Situation

```
static void Main()
{
    Console.WriteLine("Hallo, echt .NET hier!");
    Console.ReadLine();
}
```

erhält man über **F1** sehr schnell ausführliche Informationen zur Methode **WriteLine()** der Klasse **Console**:



3.3.6 Erstellungs-Optionen in der Entwicklungsumgebung setzen

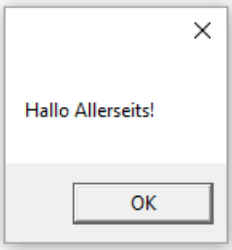
Im Abschnitt 3.2.2 zur Übersetzung von Quellcode in die Intermediate Language (IL) wurden wichtige Optionen des Compiler-Aufrufs behandelt, u. a.:

- Ausgabetyp und Name des resultierenden Assemblies
- Referenzen auf Assemblies, die der Compiler nach Klassen durchsuchen soll

Im Abschnitt 2.4.7 wurde die Zielplattform behandelt (z. B. CIL, x86, x64), die dem Compiler ebenfalls mitzuteilen ist.

Beim Einsatz unserer Entwicklungsumgebung werden die Compileraufrufe automatisch erstellt und im Hintergrund abgesetzt. Die gewünschten Compiler-Optionen wählt man in bequemen Dialogfenstern.

Um dies üben zu können, erstellen wir nun mit dem Visual Studio ein Hallo-Projekt gemäß Abschnitt 3.3.3, ersetzen aber die Textausgabe durch das folgende Nachrichtenfenster:



Während die generelle GUI-Programmierung relativ anspruchsvoll ist, gelingt die Präsentation eines Nachrichtenfensters mit Leichtigkeit. Es ist lediglich ein Aufruf der statischen Methode **Show()** der Klasse **MessageBox** an Stelle des **Console.WriteLine()** - Aufrufs erforderlich, z. B.:

```
MessageBox.Show("Hallo Allerseits!");
```

Außerdem kann der Methodenaufruf

```
Console.ReadLine();
```

entfallen, weil kein Konsolenfenster offen gehalten werden muss (vgl. Abschnitt 3.3.6.2).

3.3.6.1 Assembly-Referenzen

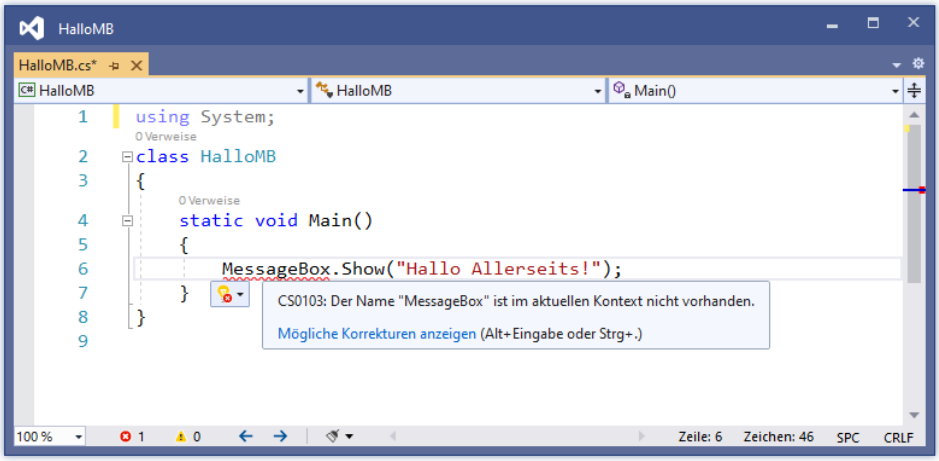
Die Assemblies mit BCL-Typen werden in der Regel automatisch referenziert. Bei der Verwaltung der Abhängigkeiten von weiteren Assemblies bestehen Unterschiede zwischen den .NET - Implementationen.

3.3.6.1.1 .NET Framework

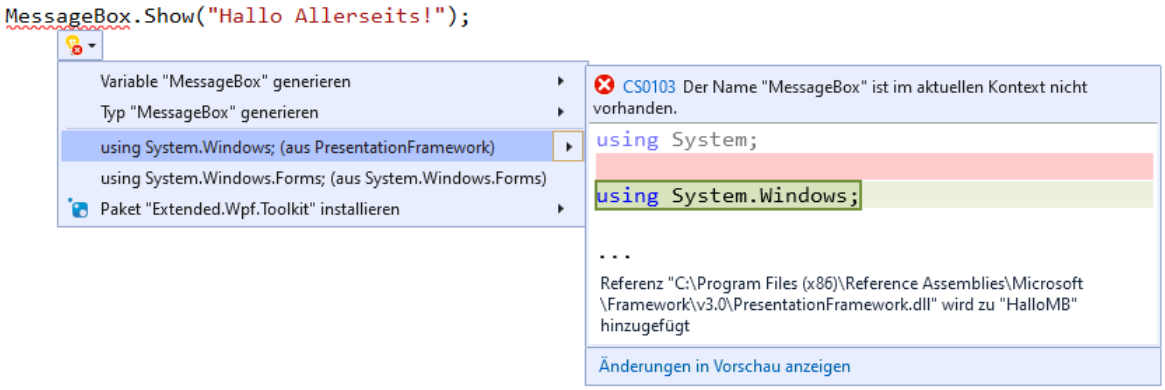
Im Visual Studio dient bei .NET Framework - Projekten der Knoten **Verweise** im **Projektmappen-Explorer** zur Verwaltung von Assembly-Referenzen.[1]

Im Hallo-Programm führt die Ersetzung der **WriteLine()** - Ausgabe durch einen **MessageBox**-Auftritt zu einer Fehlermeldung:

[1] Im .NET Framework können Assembly-Referenzen bei Verzicht auf eine Entwicklungsumgebung auch über die Befehlszeilenoption **-reference** für den C# - Compiler gesetzt werden (siehe Abschnitt 3.2.2).

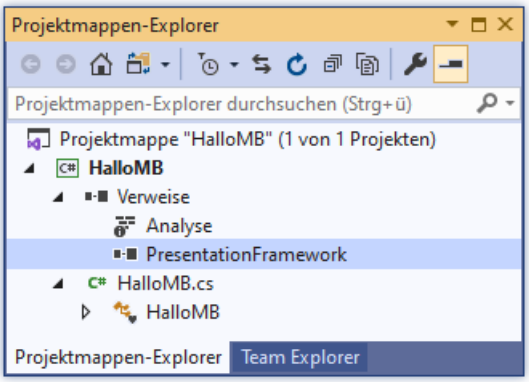


Das Drop-Down - Menü zur Glühbirne kennt den Lösungsweg:

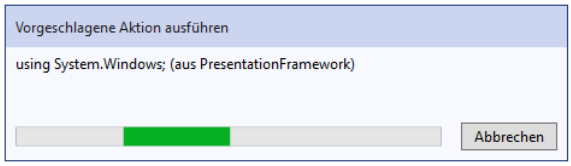


Anschließend löschen wir noch die überflüssige, an der gedämpften Farbgebung zu erkennende **using**-Direktive für den Namensraum **System**.

Weil das Visual Studio mit beachtlicher Weitsicht eine Referenz auf das die Klasse **MessageBox** im Namensraum **System.Windows** implementierende BCL-Assembly **PresentationFramework.dll** in das Projekt aufgenommen hat, ist das Programm startbereit:[1]

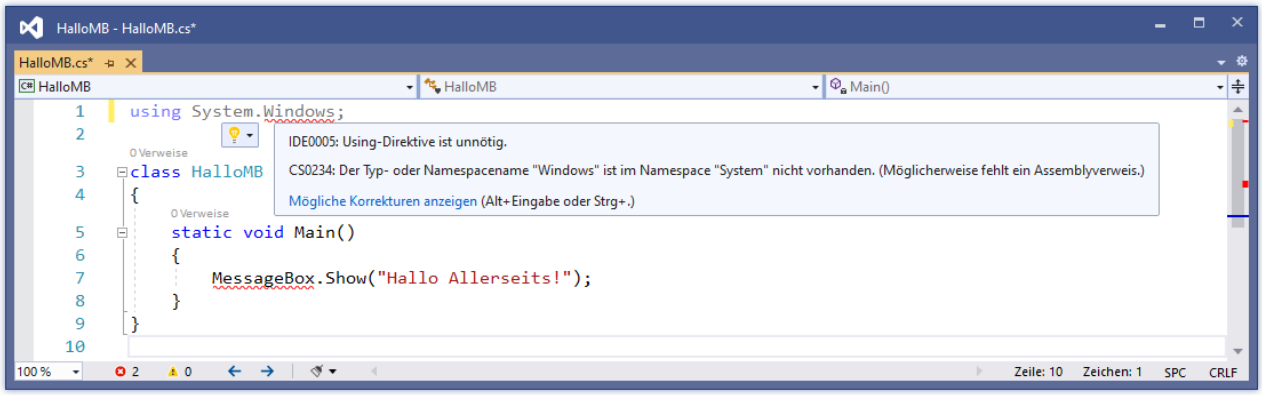


[1] Eventuell versucht das Visual Studio allerdings endlos, die vorgeschlagene Aktion auszuführen:

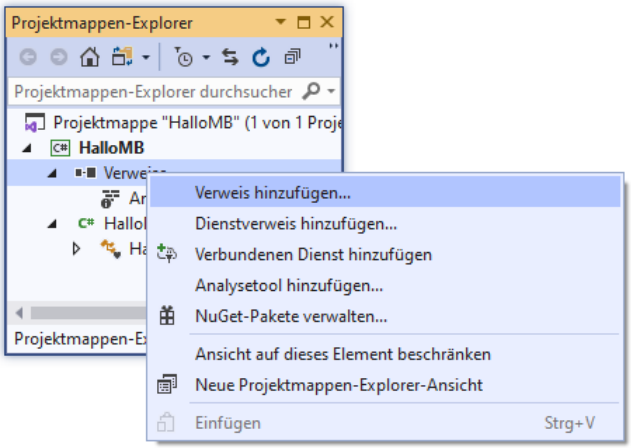


Der Fehler soll in der Version 16.9 behoben sein (siehe https://developercommunity2.visualstudio.com/t/Infinite-suggested-action-never-ends/1223055).

Der Übung halber, löschen wir den Verweis und gehen davon aus, die korrekte **using**-Direktive selbst erstellt zu haben. Der folgende korrekte (!) Quellcode wird als fehlerhaft reklamiert:

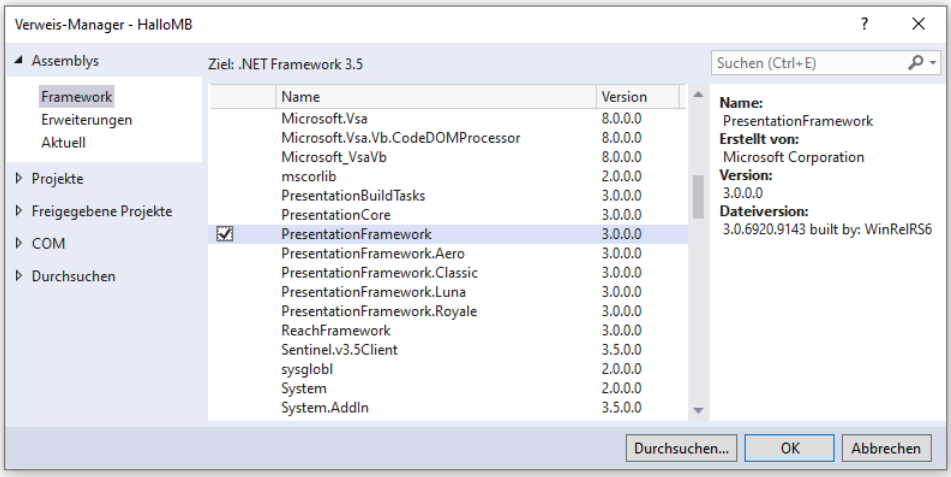


Das Visual Studio liefert den wertvollen Hinweis auf einen eventuell fehlenden Assembly-Verweis: Weil sich das die Klasse **MessageBox** implementierende Assembly **PresentationFramework.dll** nicht in der beim Übersetzen zu berücksichtigenden Referenzenliste des Projekts befindet, wird die Klasse nicht gefunden. Folglich benötigt unser Projekt eine Referenz auf das Assembly **PresentationFramework.dll**.[1] Wählen Sie dazu im **Projektmappen-Explorer** aus dem Kontextmenü zum Projekt-Knoten **Verweise** die Option **Verweis hinzufügen**:

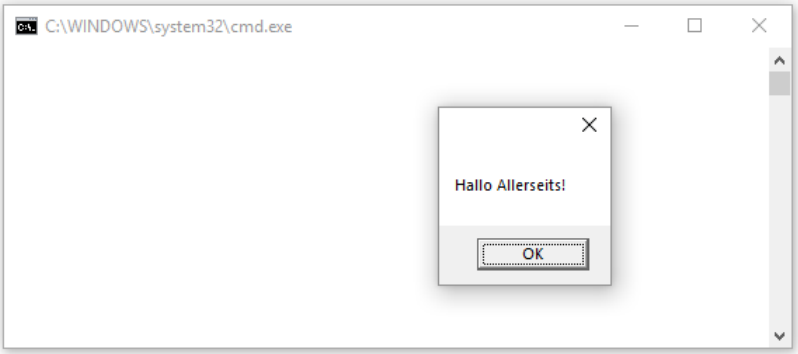


Nun kann in folgender Dialogbox das zu durchsuchende Assembly lokalisiert und über sein Kontrollkästchen in die Verweisliste aufgenommen werden:

[1] Bei Verwendung der Projektvorlage **WPF-App** legt die Entwicklungsumgebung automatisch eine Referenz auf das Assembly **PresentationFramework.dll** an.



Anschließend taucht der Verweis im Projektmappen-Explorer auf (siehe oben), und die Reklamation im Quellcode-Editor verschwindet. Wenn Sie das Programm starten, erscheint die gewünschte Messagebox:



3.3.6.1.2 .NET 5.0

Ein in .NET Core - Projekten (inklusive .NET 5.0) besonders populäres, aber auch in .NET Framework - Projekten verfügbares Verfahren zur Integration von Bibliotheks-Assemblies besteht in der Nutzung von NuGet-Paketen. Microsoft hat unter dem Namen *NuGet* Spezifikationen und eine Infrastruktur geschaffen, um die Erstellung, Verteilung und Nutzung von Paketen mit .NET - Bibliotheks-Assemblies zu unterstützen.[1] In der Regel werden die Pakete auf dem öffentlichen Server

https://www.nuget.org/

zur Verfügung gestellt, dessen Verwendung gut in das Visual Studio integriert ist.

Wir verwenden in einem Beispiel das NuGet-Paket **Newtonsoft.Json** mit Typen zur Unterstützung des JSON-Formats (JavaScript Object Notation) bei der Serialisierung von Objekten.[2] Das Serialisieren kann z. B. zum Speichern von Objekten in Dateien verwendet werden (siehe Abschnitt 16.5.2).

Das NuGet-Paket **Newtonsoft.Json** (auch bekannt unter dem Namen *Json.NET*) ist aktuell sehr populär als Lösung zum (De-)Serialisieren von .NET-Objekten unter Verwendung des JSON-Formats. Es nimmt in der Hitliste der beliebtesten NuGet-Pakete den ersten Platz ein mit großem

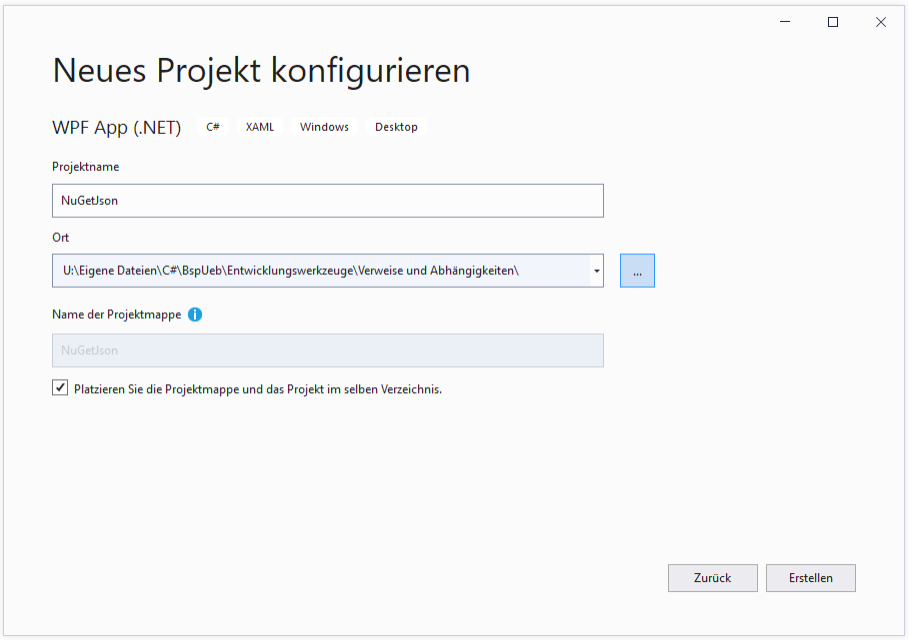
---

1 https://docs.microsoft.com/de-de/nuget/what-is-nuget

2 https://docs.microsoft.com/de-de/nuget/quickstart/install-and-use-a-package-in-visual-studio

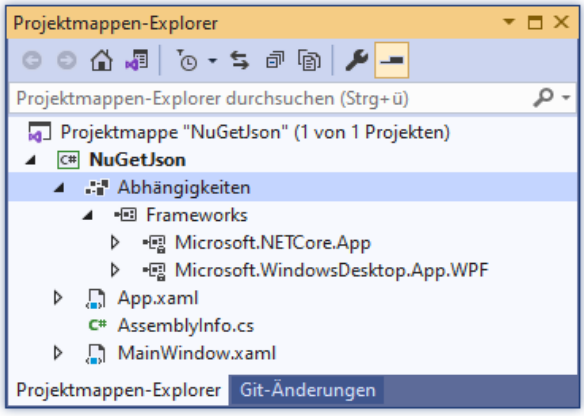
Abstand vor dem Zweitplatzierten.[1] Eventuell wird **Newtonsoft.Json** aber zukünftig an Bedeutung verlieren, weil Microsoft mit dem Erscheinen von .NET Core 3.0 eine konkurrierende Lösung in die BCL aufgenommen hat (Namensraum **System.Text.Json**).

Wie im Abschnitt 3.3.4.5 wird die Projektvorlage **WPF-App (.NET)** genutzt, um ein Programm mit grafischer Bedienoberfläche in WPF-Technik (*Windows Presentation Foundation*) zu erstellen:

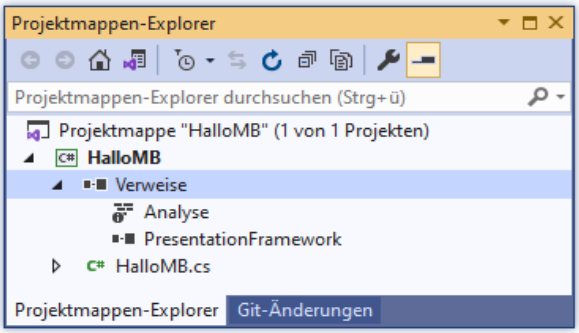


Im **Projektmappen-Explorer** des neuen Projekts befindet sich ein Knoten zur Verwaltung von **Abhängigkeiten** an derselben Stelle, die in einem .NET - Framework - Projekt vom Knoten zur Verwaltung der **Verweise** eingenommen wird:

.NET Core oder .NET 5.0

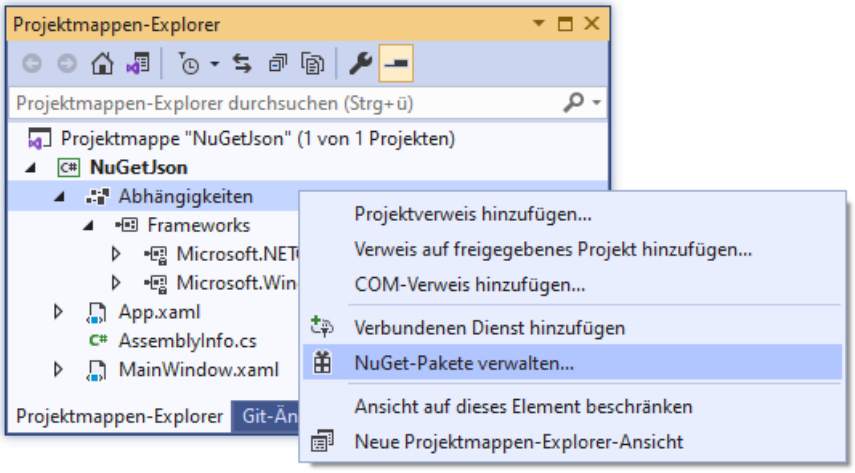


.NET Framework



Wir wählen aus dem Kontextmenü zu den **Abhängigkeiten** das Item **NuGet-Pakete verwalten**:

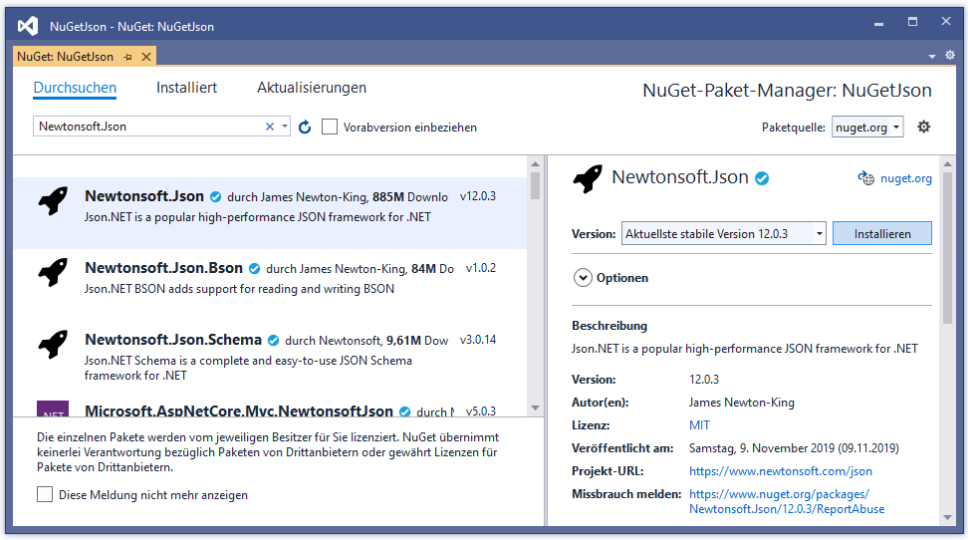
[1] https://www.nuget.org/stats/packages



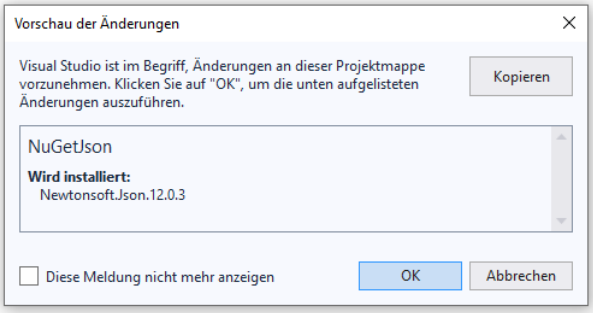
Daraufhin erscheint der **NuGet-Paket-Manager**, der folgende Leistungen anbietet:

- Die Webseite https://www.nuget.org/ nach Paketen **durchsuchen**
- Die im Projekt **installierten** Pakete verwalten
- Die Pakete des Projekts **aktualisieren**

Mit Hilfe des Suchfelds ist das benötigte Paket schnell gefunden:

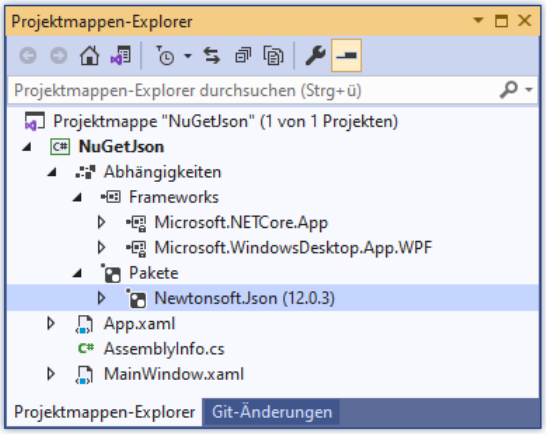


Wir veranlassen das **Installieren** der stabilen Version 12.0.3 und bestätigen per Klick auf **OK**:



In der Liste der vom Paket **Newtonsoft.Json** unterstützten .NET - Implementationen taucht .NET 5.0 nicht explizit auf. Allerdings wird die Version 2.0 von .NET Standard genannt, und dieser Standard wird von .NET 5.0 erfüllt (siehe Abschnitt 2.1).

Im Projektmappen-Explorer wird anschließend die neue **Abhängigkeit** angezeigt:

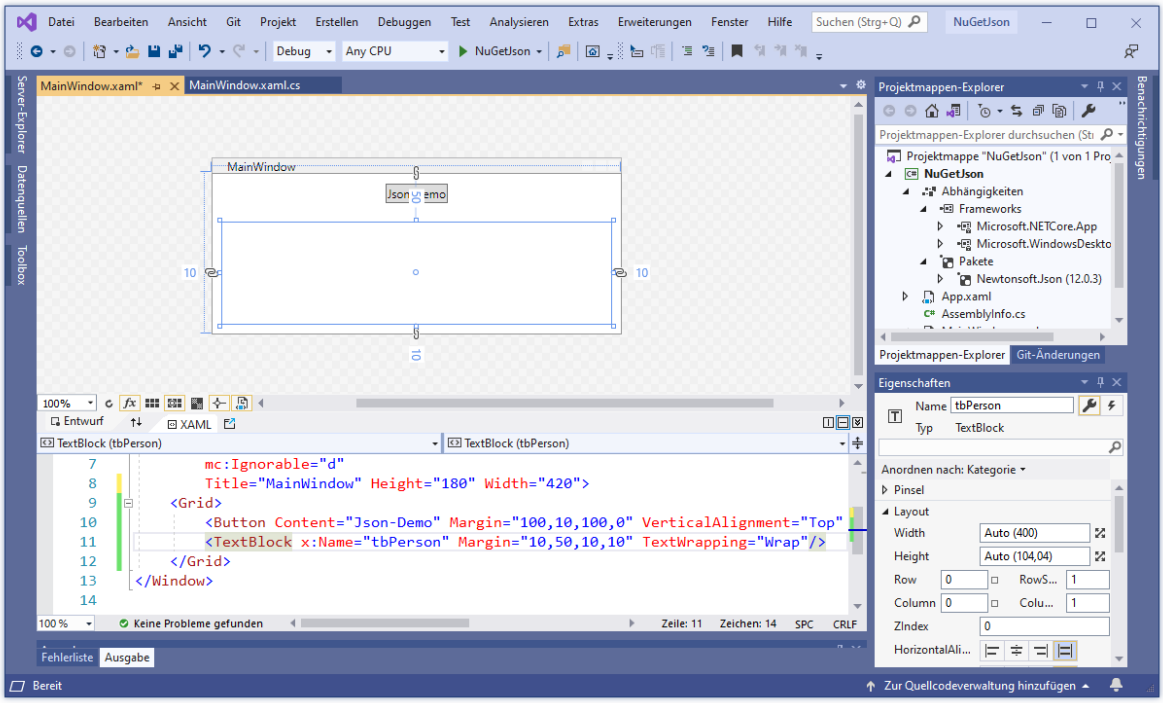


In der Projektdatei **NuGetJson.csproj** befindet sich nach dem Speichern ein **PackageReference** - Element, das die Abhängigkeit des Projekts vom Paket **Newtonsoft.Json** beschreibt:

```
<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net5.0-windows</TargetFramework>
    <UseWPF>true</UseWPF>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Newtonsoft.Json" Version="12.0.3" />
  </ItemGroup>

</Project>
```

Nebenbei erfahren wir, wie **Microsoft.WindowsDesktop.App.WPF** in die Liste der Abhängigkeiten eines .NET Core - Projekts gelangt. Aufgrund der gewählten Projektvorlage **WPF-App (.NET)** ist das Element **UseWPF** in die Projektdatei eingetragen worden.

Wir setzen mit Hilfe der Toolbox ein **Button**-Objekt und ein **TextBlock**-Objekt mit dem Namen `tbPerson` auf das Anwendungsfenster (siehe Abschnitt 3.3.4.2):



Analog zu Abschnitt 3.3.4.3 lassen wir per Doppelklick auf den Schalter eine Behandlungsmethode zu seinem **Click**-Ereignis erstellen, die vom Visual Studio den Namen `Button_Click()` erhält. In der Quellcodedatei **MainWindow.xaml.cs** definieren wir eine rudimentäre Klasse namens `Person`, wobei der Einfachheit halber auf die Datenkapselung verzichtet wird.[1]

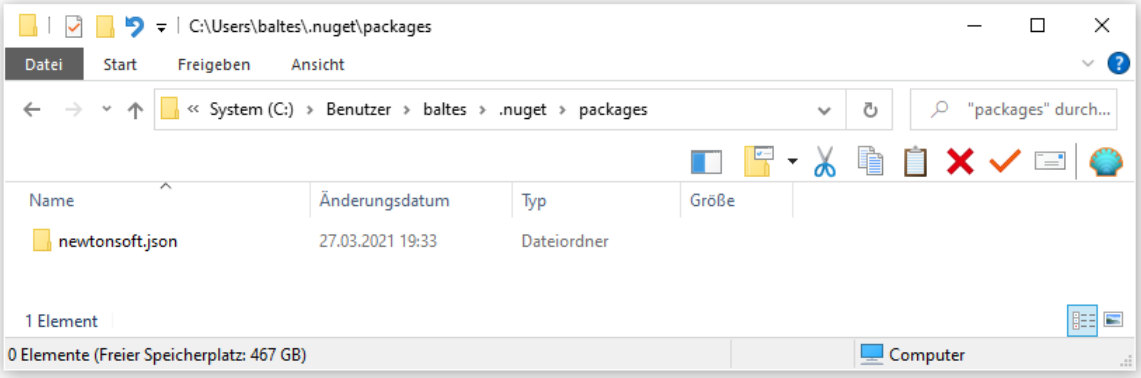
```
using System;
using System.Windows;
using Newtonsoft.Json;

namespace NuGetJson {
    public class Person {
        public string Name;
        public string Email;
        public DateTime Datum;
    }
    public partial class MainWindow : Window {
        public MainWindow() {
            InitializeComponent();
        }

        private void Button_Click(object sender, RoutedEventArgs e) {
            Person person = new Person();
            person.Name = "Kurt Rempremerding";
            person.Email = "kurt@rempremerding.de";
            person.Datum = new DateTime(1950, 12, 8);
            tbPerson.Text = JsonConvert.SerializeObject(person, Formatting.Indented);
        }
    }
}
```

In der Methode `Button_Click()` wird per **new**-Operator ein Objekt der Klasse `Person` erstellt. Aus seinen Merkmalsausprägungen entsteht mit Hilfe der statischen Methode **SerializeObject()** der Klasse **JsonConvert** aus dem Paket **Newtonsoft.Json** eine Zeichenfolge, die anschließend an die **Text**-Eigenschaft des **TextBlock**-Objekts übergeben wird.

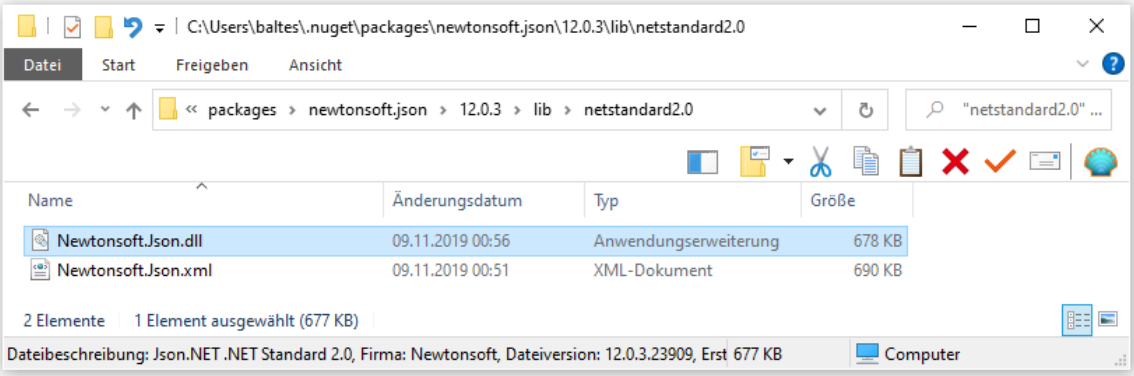
Wir lassen das Programm über die Tastenkombination **Strg+F5** übersetzen und ausführen. Nach einem Kick auf den Schalter erscheint das Ergebnis der Objektserialisierung im JSON-Format:

Das NuGet-Paket **Newtonsoft.Json** wird unter Windows im benutzereigenen Ordner **.nuget** abgelegt, z. B.:

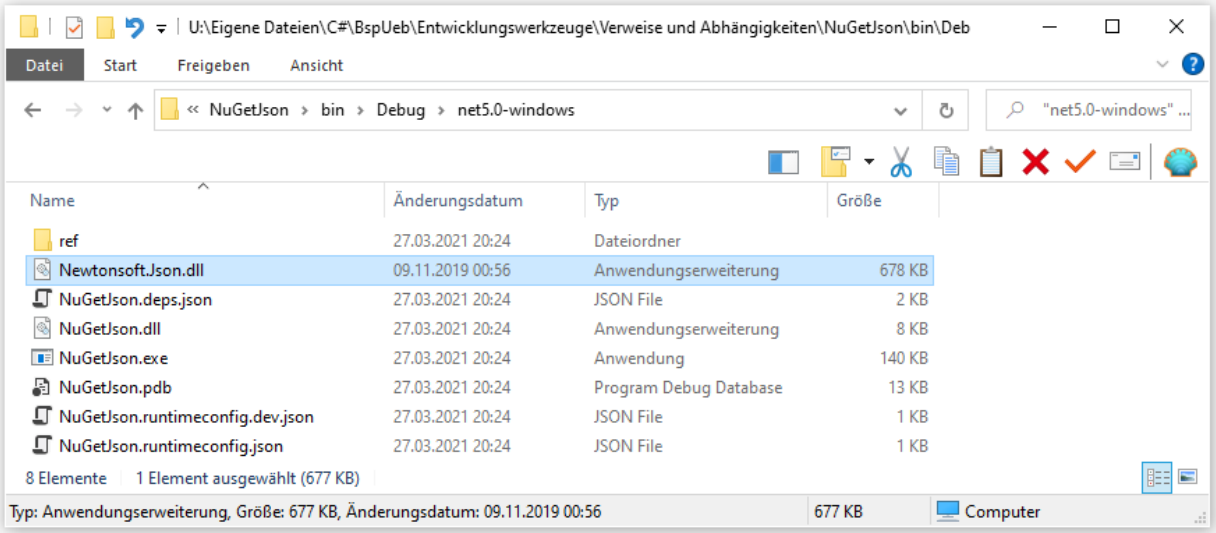
[1] Wir erlauben uns diese Nachlässigkeit, damit das Beispiel einfach bleibt und möglichst wenige Vorgriffe auf spätere Kapitel benötigt.



Im Beispiel befindet sich die zur Programmausführung taugliche Datei **Newtonsoft.Json.dll** im Unterordner **...\lib\netstandard2.0** des NuGet-Pakets



sowie im Erstellungs-Ausgabeordner des Projekts:



Eine Programmausführung außerhalb der Entwicklungsumgebung (z. B. per Doppelklick auf die Datei **NuGetJson.exe**) scheitert, wenn die Datei **Newtonsoft.Json.dll** an den beiden beschriebenen Aufenthaltsorten fehlt.

Fehlt das NuGet-Paket beim Öffnen der Projektdatei, dann wird es vom Visual Studio heruntergeladen. Wenn das NuGet-Paket vorhanden ist, die benötigte Datei **Newtonsoft.Json.dll** im Ordner **...\lib\netstandard2.0** aber fehlt, dann scheitert die Erstellung.

3.3.6.2 Ausgabetyp, Assembly-Name, Namensraum und Zielframework

Im Beispiel aus dem Abschnitt 3.3.6.1.1 erscheint neben dem Nachrichtenfenster auch ein ziemlich überflüssiges, leeres Konsolenfenster. Um seinen Auftritt zu verhindern, kann man im .NET - Framework per Compiler-Option den Ausgabetyp **exe** durch die Alternative **winexe** ersetzen (vgl. Abschnitt 3.2.2). Für alle .NET - Implementationen lässt sich der **Ausgabetyp** im Visual Studio nach

Projekt > Eigenschaften > Anwendung

im folgenden Dialog ändern:

Bei einem Programm mit dem **Ausgabetyp winexe** (**Windows-Anwendung**) gehen alle Konsolenausgaben verloren, sodass man diesen Ausgabetyp mit Bedacht wählen sollte.

Neben dem **Ausgabetyp** lassen sich auf der **Anwendung**s-Seite des Projekteigenschaftsdialogs noch weitere wichtige Einstellungen vornehmen:

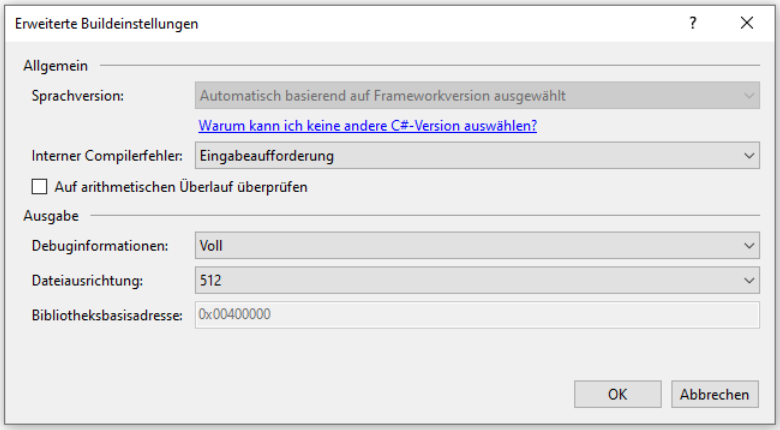
- **Assemblyname**
  Per Voreinstellung übernimmt das erstellte Assembly seinen Namen vom Projekt.
- **Standardnamespace**
  Für größere Projekte sollte ein eigener Namensraum definiert werden. Von der Entwicklungsumgebung erstellter Code verwendet den **Standardnamespace**, der initial mit dem Projektnamen identisch ist. Eine Änderung wirkt sich auf *neue* Quellcodedateien aus, aber *nicht* auf vorhandene.
- **Zielframework**
  Welche minimale Version der gewählten .NET - Implementation (.NET Framework, .NET Core oder .NET) wird vorausgesetzt? Wenn eine gewünschte Version in der angebotenen Liste fehlt, hilft eine Ergänzungsinstallation über den **Visual Studio - Installer** (siehe Abschnitt 3.3.2).

3.3.6.3 C# - Sprachversion

Bis zu Version 2017 unserer Entwicklungsumgebung konnte die im Projekt verwendete C# - Sprachversion nach folgenden Schritten

- Menübefehl **Projekt > Eigenschaften**
- Registerkarte **Build**
- Schalter **Erweitert**

überprüft und geändert werden. In der Version 2019 wird hingegen statt eines Drop-Down - Menüs nur noch eine Erläuterung angeboten:



Auf der verlinkten Webseite

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/configure-language-version

ist zu erfahren, ...

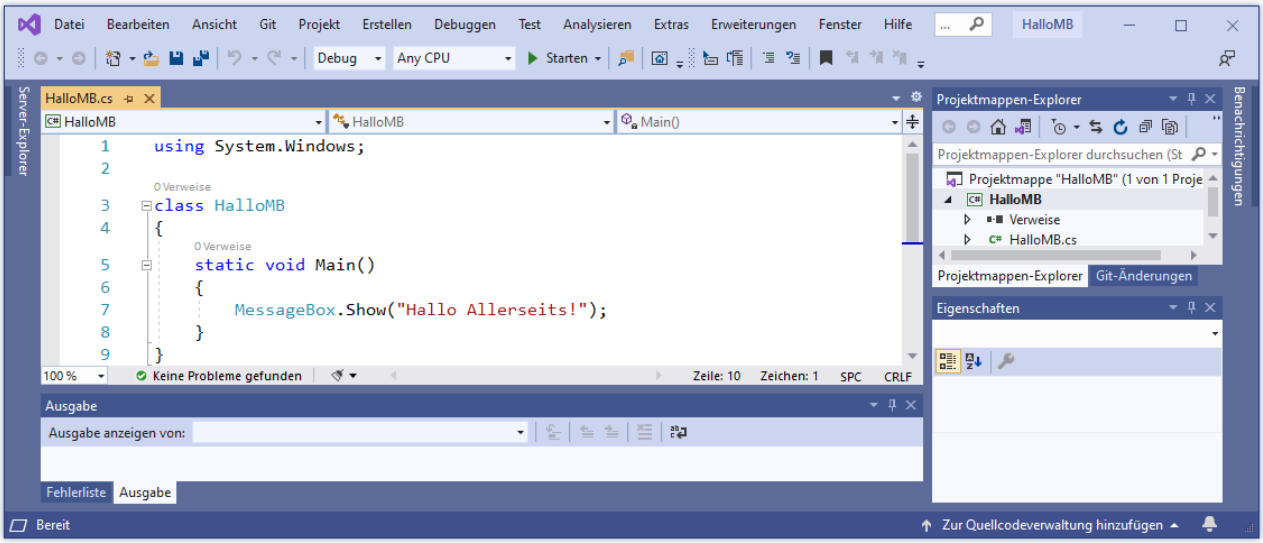
- dass nun der C# - Compiler aufgrund des im Projekt eingestellten Zielframeworks eine Sprachversion wählt,
- dass über die Projektkonfigurationsdatei (mit der Namenserweiterung **.csproj**) noch eine Wahl der Sprachversion möglich ist.

Bei den im Kurs bzw. Manuskript verwendeten .NET - Implementationen sind folgende C# - Versionen voreingestellt:

- .NET 5.0: C# 9.0
- .NET Framework: C# 7.3

3.3.6.4 Zielplattform

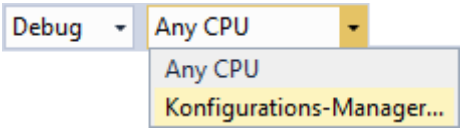
In Visual Studio 2019 ist die Zielplattform **Any CPU** voreingestellt, z. B.:



Dies ist konsistent mit der Beobachtung aus dem Abschnitt 2.4.7, dass der C# - Kommandozeilen - Compiler die voreingestellte Zielplattform **CIL** verwendet. Der Compiler produziert daraufhin ein Assembly, das unter Windows 64 in einem 64-Bit - Prozess ausgeführt wird und auch unter Windows 32 läuft.

Obwohl es nur selten erforderlich ist, eine alternative Zielplattform einzustellen (siehe Diskussion im Abschnitt 2.4.7), werden die erforderlichen Schritte anschließend beschrieben. Soll für eine Pro-

jektmappe eine alternative Zielplattform wählbar sein, muss man den Konfigurations-Manager bemühen, was über die Plattformliste in der Symbolleiste **Standard**

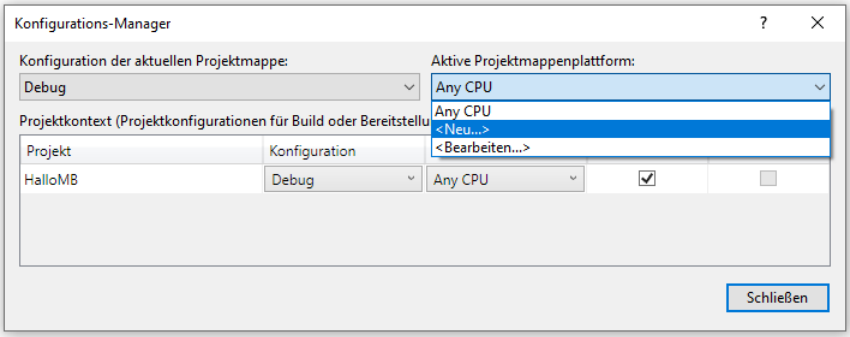


oder den Menübefehl

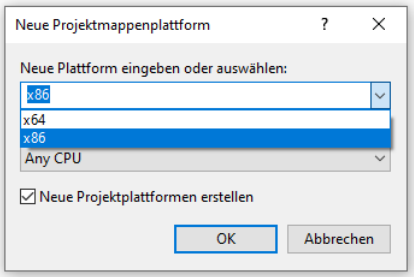
Erstellen > Konfigurations-Manager

möglich ist. Nun kann man z. B. so vorgehen:

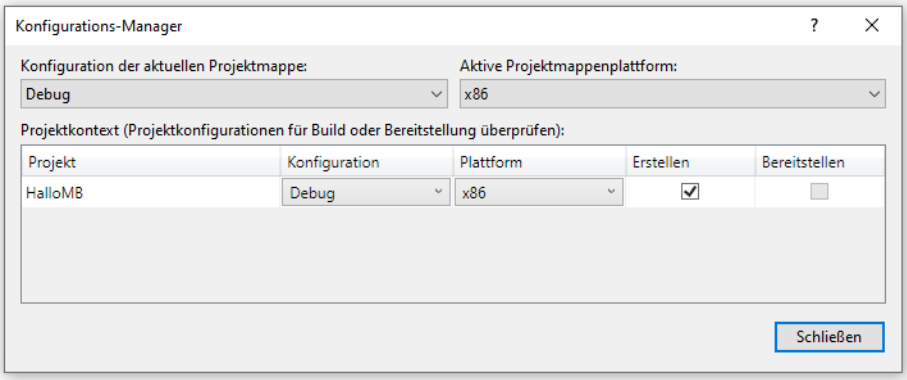
- Man legt eine neue **Projektmappenplattform** an:



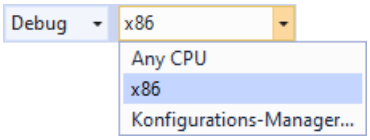
- Im folgenden Dialog wählt man eine Plattform, z. B.:



- Die Markierung beim Kontrollkästchen **Neue Projektplattformen erstellen** sollte belassen werden, sodass nach dem Quittieren mit **OK** bei allen Projekten in der Mappe die neue Zielplattform voreingestellt ist:



Anschließend kann ein Assembly mit der gewünschten Zielplattform erstellt werden:

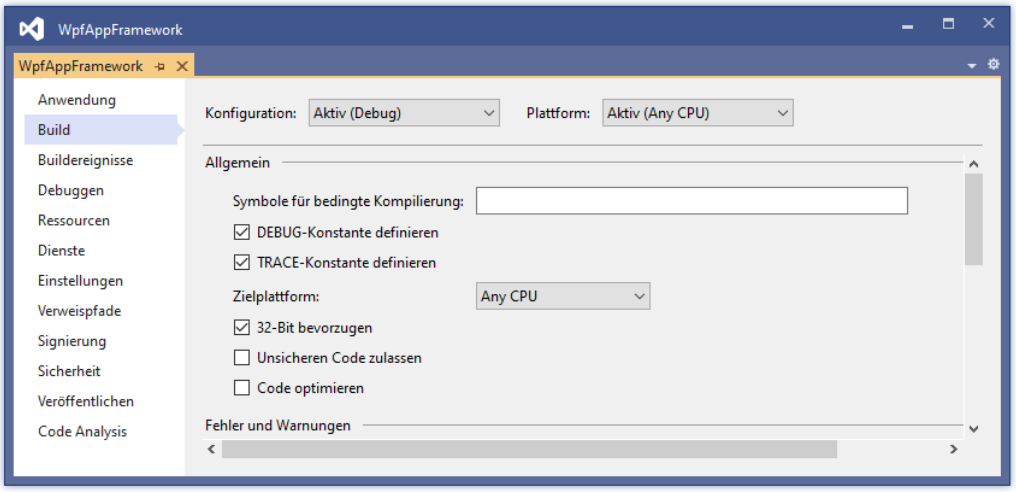


Bei den Beispielprojekten im Kurs werden wir wohl kaum einen Grund finden, die vom Visual Studio vorgeschlagene Zielplattform **Any CPU** zu ändern.

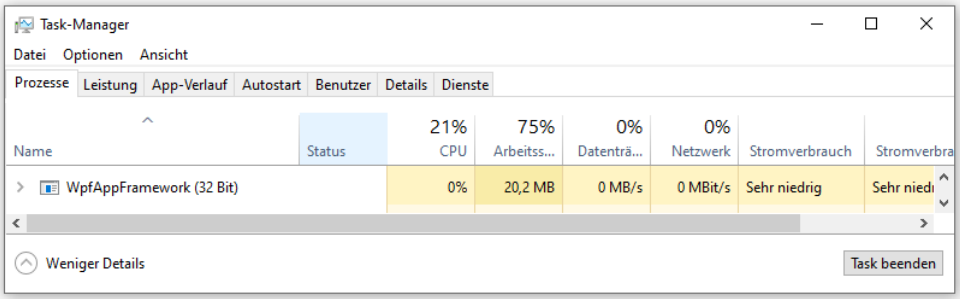
Bei einem auf der Vorlage **WPF App (.NET Framework)** basierenden Projekt zeigt sich nach

Projekt > Eigenschaften > Build

eine spezielle Konfiguration der **Zielplattform**:



Dass die Bevorzugung von 32-Bit ernst gemeint ist, zeigt bei laufendem Programm ein Blick auf den Taskmanager von Windows:



Unter der Zielplattform **Any CPU mit 32 Bit-Bevorzugung** ist folgendes zu verstehen:[1]

- Auf einem Windows-System mit 32 Bit läuft das Programm als 32-Bit - Prozess. Die IL wird in x86 - Maschinencode übersetzt.
- Auf einem Windows-System mit 64 Bit läuft das Programm als 32-Bit - Prozess. Die IL wird in x86 - Maschinencode übersetzt.
- Auf einem ARM-Windows-System läuft das Programm als 32-Bit - Prozess. Die IL wird in ARM - Maschinencode übersetzt.

[1] https://stackoverflow.com/questions/12066638/what-is-the-purpose-of-the-prefer-32-bit-setting-in-visual-studio-and-how-does

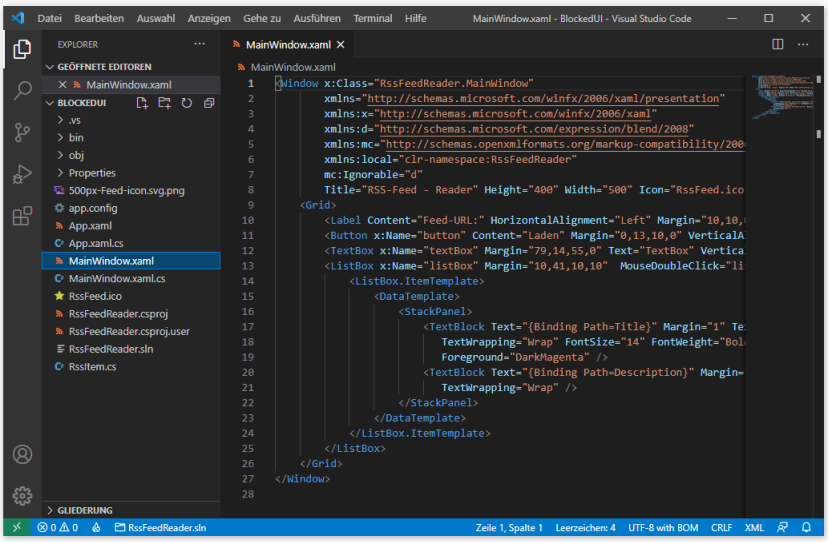
3.4 Microsoft Visual Studio Code

Microsoft bietet unter dem Namen *VS Code* eine leichtgewichtige Alternative zum Visual Studio an, die einige Pluspunkte aufzuweisen hat:

- Verfügbarkeit für Linux, macOS und Windows
- Geringe Hardwareanforderungen
- Eine für unsere Zwecke ausreichende Unterstützung bei der Software-Entwicklung mit C# (inklusive Editor mit IntelliSense, Umbenennungs-Refaktorierung, CodeLens etc.)

Als für uns relevante Nachteile im Vergleich zum Visual Studio sind u. a. zu nennen:

- Es fehlt ein grafischer Fenster-Designer, sodass man bei der Entwicklung von Anwendungen mit grafischer Bedienoberfläche (zurzeit nur möglich für Windows) um das Editieren des XAML-Codes nicht herumkommt:



- Bei der Entwicklung von Datenbankanwendungen (siehe Kapitel 18 und 20) fehlt ein Werkzeug, das mit dem ins Visual Studio integrierten **SQL Server Objekt-Explorer** vergleichbar ist.

Die genannten Nachteile sind nicht allzu gravierend, doch ist für die Entwicklung unter Windows trotzdem das Visual Studio komfortabler. Bei sehr schwacher Hardware-Ausstattung können allerdings die geringeren Anforderungen von VS Code den Ausschlag geben (siehe Abschnitt 3.4.1). Unter Linux ist das VS Code für die Entwicklung mit C# die erste Wahl.[1]

3.4.1 Voraussetzungen

Die im September 2021 genannten Systemvoraussetzungen für VS Code sind sehr bescheiden:[2]

- Betriebssystem:
  - Linux:
    - Debian: Ubuntu Desktop 16.04, Debian 9
    - Red Hat: Red Hat Enterprise Linux 7, CentOS 8, Fedora 24
  - macOS: 10.11
  - Windows: 8.1 oder 10
- Prozessor mit 1,6 GHz
- 1 GB RAM

---

[1] Das VS Code konkurriert auf dem Mac mit einer speziellen Visual Studio - Version, die auf dem Xamarin Studio basiert. Mangels Erfahrung sind dazu keine Empfehlungen möglich.

[2] https://code.visualstudio.com/docs/supporting/requirements

- Für die C# - Entwicklung werden unter Windows ca. 600 MB Massenspeicherplatz (auf einer SSD oder Festplatte) verwendet, also im Vergleich zum Visual Studio ca. 4 GB eingespart (vgl. Abschnitt 3.1.3).

Unter Linux sind folgende Bibliotheken erforderlich:

- GLIBCXX ab Version 3.4.21
- GLIBC ab Version 2.15

Unter Windows wird das .NET Framework 4.5.2 benötigt.

3.4.2 Bezugsquelle

Von der Webseite

https://code.visualstudio.com/download

kann man das Visual Studio Code für Linux, macOS und Windows beziehen. Unter Linux und Windows wird zwischen einer benutzer- und einer system-seitigen Installation unterschieden, wobei die benutzer-seitige Installation wegen der größeren Bequemlichkeit bei der Aktualisierung empfohlen wird.

3.4.3 Installation

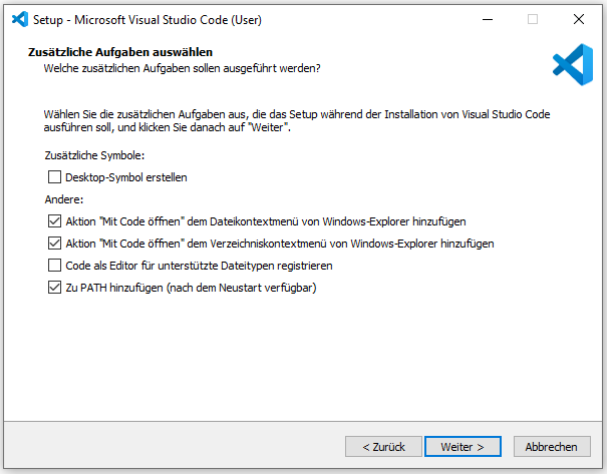
Microsoft empfiehlt die benutzer-bezogene Installation im Vergleich zur system-bezogenen Installation, weil ...

- bei der benutzer-bezogenen Installation *keine* Administratorrechte benötigt werden,
- sodass Aktualisierungen einfacher durchgeführt werden können.

Das benutzer-seitige Windows-Installationsprogramm (z. B. **VSCodeUserSetup-x64-1.55.0.exe**) verwendet für einen Benutzer namens `otto` ohne Wahlmöglichkeit den folgenden Installationsordner:[1]

C:\Users\otto\AppData\Local\Programs\Microsoft VS Code

Werden im Fenster mit den **zusätzlichen Aufgaben** die Optionen **Mit Code öffnen** markiert, dann lassen sich Projekte bequem per Explorer-Fenster und Kontextmenü öffnen:

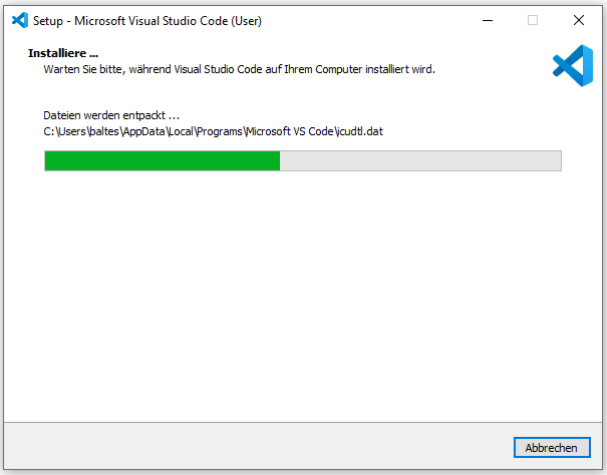


[1] Die Arbeit mit der VS Code - Version 1.60.0 wurde im September 2021 erheblich behindert durch die Fehlermeldung:

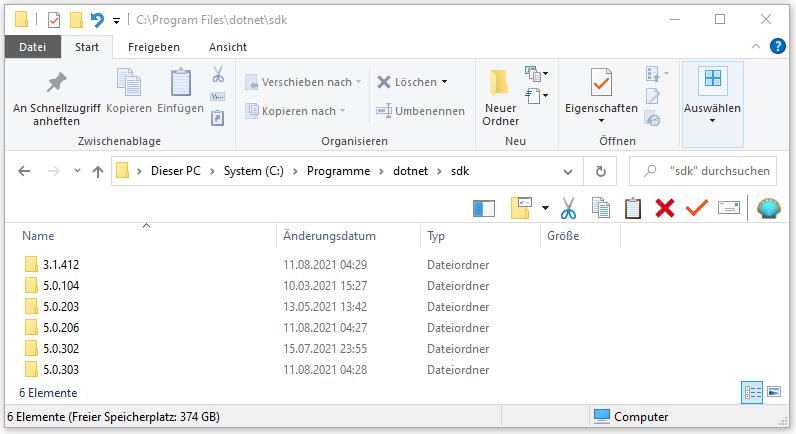
Unable to generate assets to build and debug. OmniSharp server is not running.

Daher wird im Manuskript die Version 1.55.0 verwendet.

Die Installation dauert nur wenige Sekunden:



Zusätzlich wird das .NET SDK in einer Version ab 5.0 benötigt. Die im Abschnitt 3.1 beschriebene Visual Studio - Installation beinhaltet auch das .NET 5.0 SDK, erkennbar an der Existenz des Ordners **C:\Program Files\dotnet\sdk** mit Unterordnern für vorhandene SDK-Installationen von .NET Core und .NET 5.0:



Für die korrekte Funktion von VS Code ist unter Windows das Programm

C:\Program Files\dotnet\dotnet.exe

erforderlich, das über den Suchpfad für ausführbare Programme erreichbar sein muss, was in einem Konsolenfenster mit dem Befehl **where** verifiziert werden kann:

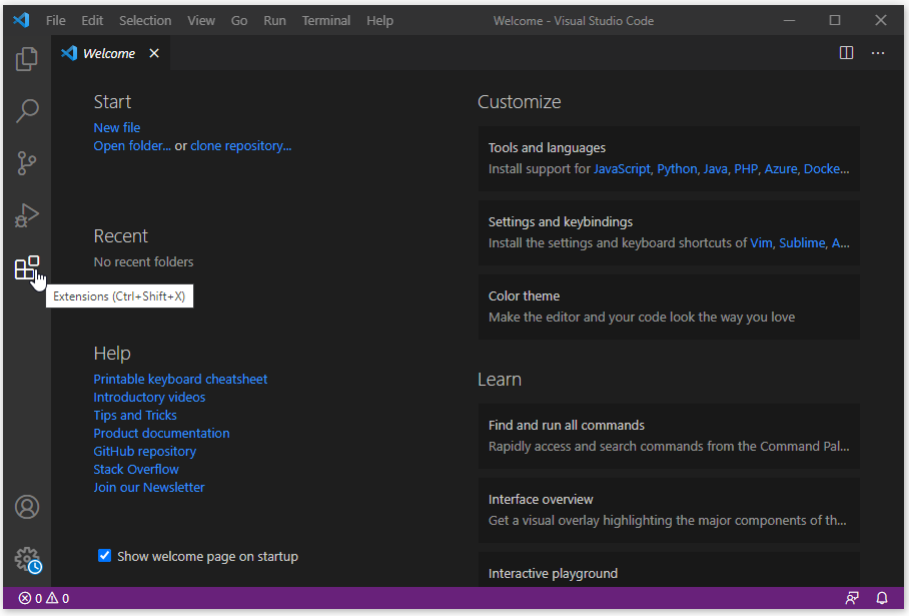
Ein separates .NET SDK - Installationsprogramm (für Linux, macOS oder Windows) ist hier zu finden

https://dotnet.microsoft.com/download/dotnet/5.0

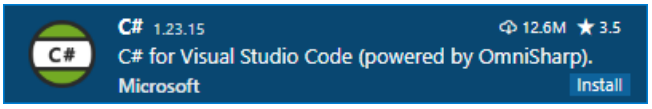
Die VS Code - Konfiguration des Windows-Benutzers `otto` landet im folgenden Ordner:

C:\Users\otto\AppData\Roaming\Code

Für die Entwicklung von Software in C# wird in VS Code eine Erweiterung benötigt, die nach dem Start der Entwicklungsumgebung installiert werden kann. Man klickt zunächst auf den für **Extensions** zuständigen Schalter:



Dann wählt man (bei Bedarf unterstützt durch die Eintragung **C# for Visual Studio Code** im Suchfeld) diese Erweiterung:



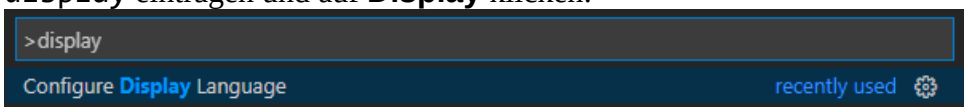
Für den Windows-Benutzer `otto` wird die Erweiterung in den folgenden Ordner installiert:

C:\Users\otto\.vscode\extensions\ms-dotnettools.csharp-1.23.15

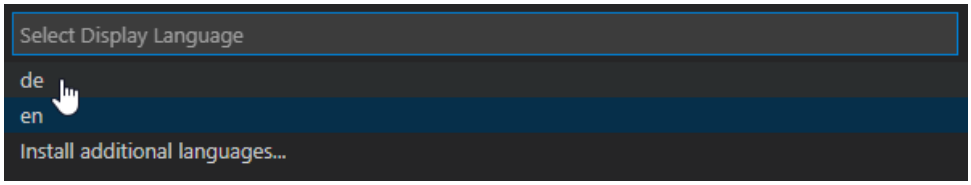
VS Code legt Erweiterungen grundsätzlich benutzer-bezogen ab, sodass Aktualisierungen mit normalen Benutzerrechten möglich sind. Solche Aktualisierungen finden nötigenfalls automatisch beim Zugriff statt.

Wer die Erweiterung **German Language Pack for Visual Studio Code** installiert, kann anschließen folgendermaßen die deutsche Bedienoberfläche aktivieren:

- Menübefehl **View > Command Palette**
- `display` eintragen und auf **Display** klicken:



- Klick auf **de**



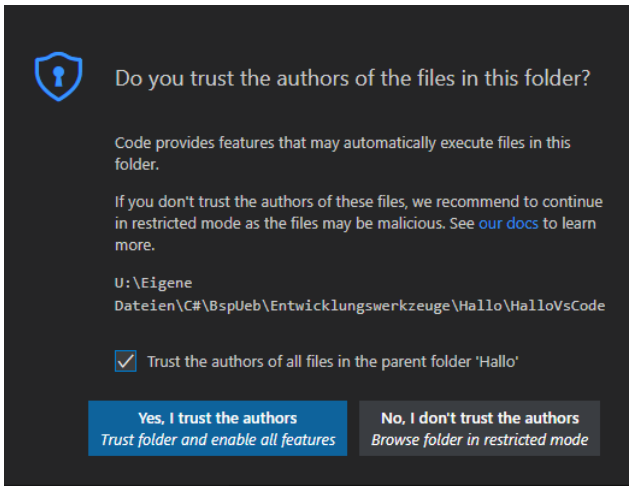
- Visual Studio Code neu starten

3.4.4 Ein erstes Konsolen-Projekt

3.4.4.1 Projekt erstellen

Wir erstellen ein Konsolenprogramm für .NET 5.0 mit dem Namen `HalloVSCode`:

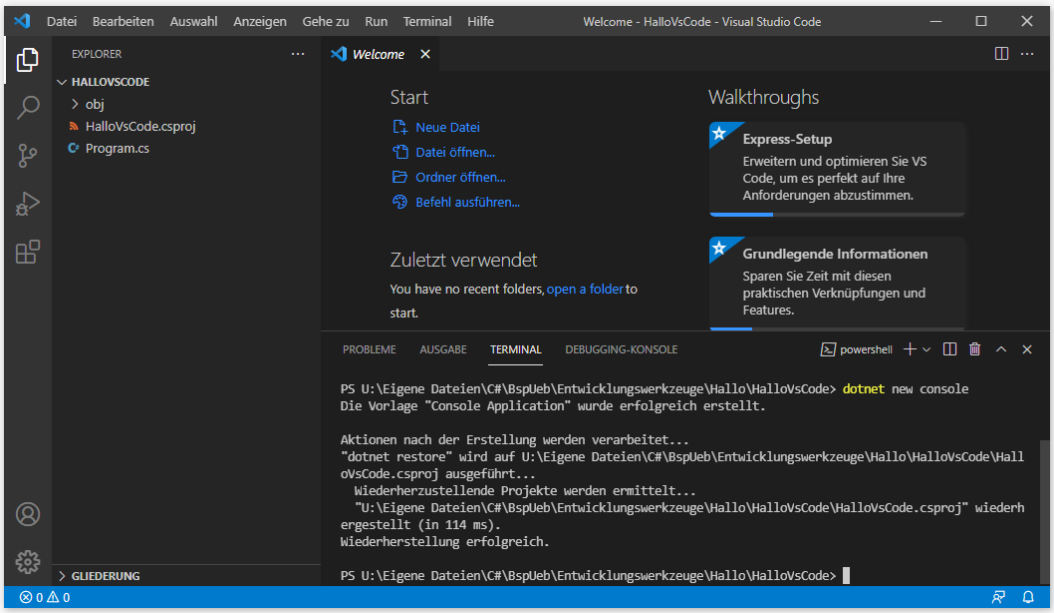
- Wählen Sie im VS Code den Menübefehl **Datei > Ordner öffnen**
- Legen Sie im Dialog **Ordner öffnen** einen neuen Ordner an, und **wählen** Sie diesen **aus**.
- Bestätigen Sie ggf. Ihr Vertrauen gegenüber den Autoren der im neuen Ordner sowie im übergeordneten Ordner befindlichen Dateien:



- Wählen Sie in VS Code den Menübefehl **Anzeigen > Terminal**.
- Schicken Sie im **TERMINAL** den folgenden Befehl ab, der eine Anwendung unter Verwendung der Vorlage **console** erstellt:

```
dotnet new console
```

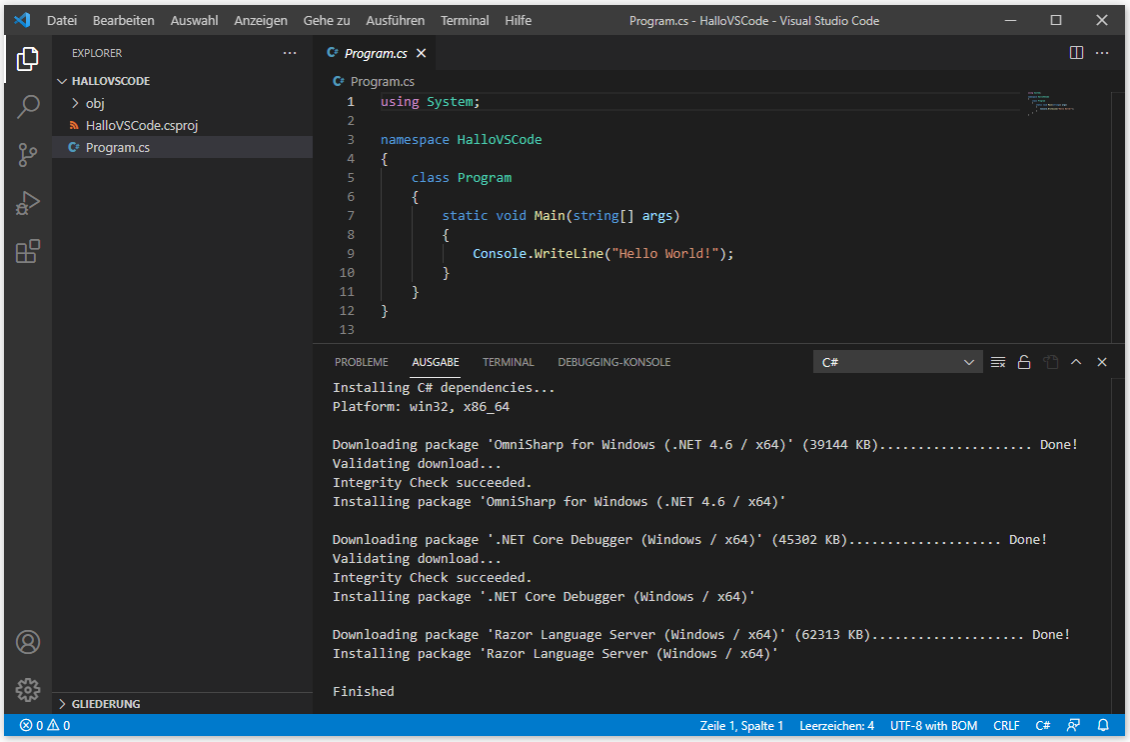
Im **TERMINAL** erscheint eine Erfolgsmeldung,



und der VSCode-**EXPLORER**, den man bei Bedarf über sein Symbol  in den Vordergrund holt, zeigt zum neuen Projekt u. a.:

- den vom Ordner übernommenen Namen `HALLOVSCODE`
- die Projektkonfigurationsdatei **HalloVSCode.csproj**
- die C# - Quellcodedatei **Program.cs**

Nach einem Mausklick auf die Quellcodedatei **Program.cs** erscheint ein automatisch erstelltes Hallo-Programm im Editor:

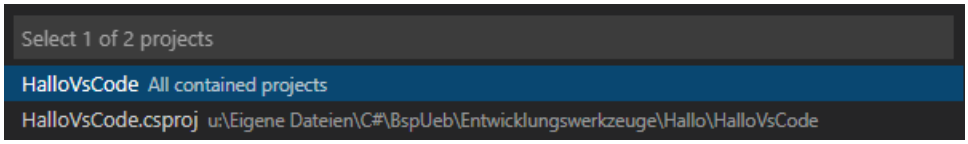


Es unterscheidet sich praktisch nicht vom Hallo-Programm, das wir im Abschnitt 3.3.3 vom Visual Studio erhalten haben.

Beim Laden der ersten C# - Quellcodedatei werden die **OmniSharp**-Unterstützungswerkzeuge heruntergeladen und installiert (siehe Bildschirmfoto).

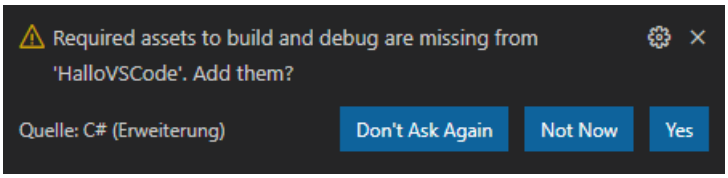
3.4.4.2 Startkonfiguration

Wenn nach der Installation der **OmniSharp**-Unterstützungswerkzeuge die folgende Aufforderung

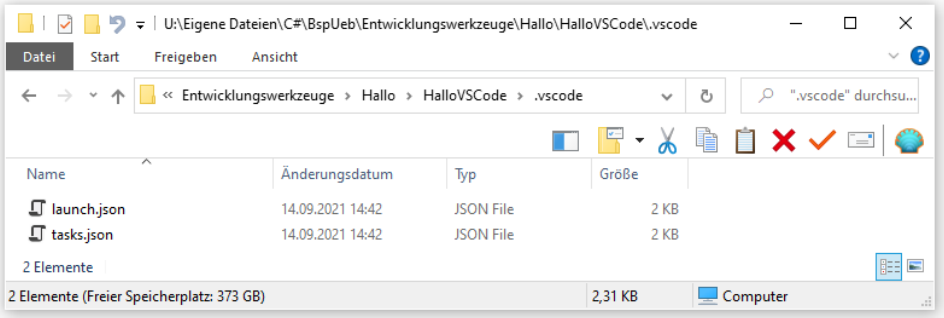


erscheint, dann sollten Sie per Mausklick die erste Option (mit dem Namen des Projektordners) wählen.

Nach einigen Sekunden erscheint die folgende Anfrage, die mit **Yes** beantwortet werden sollte:



Daraufhin entstehen im Projektunterordner **.vscode** die Konfigurationsdateien **launch.json** und **tasks.json**, die zum Starten der Anwendung benötigt werden:



Es ist kein Problem, wenn die Abfrage verschwunden ist, ohne Ihre Entscheidung abzuwarten. Man kann die Erstellung der Startkonfiguration jederzeit anfordern:

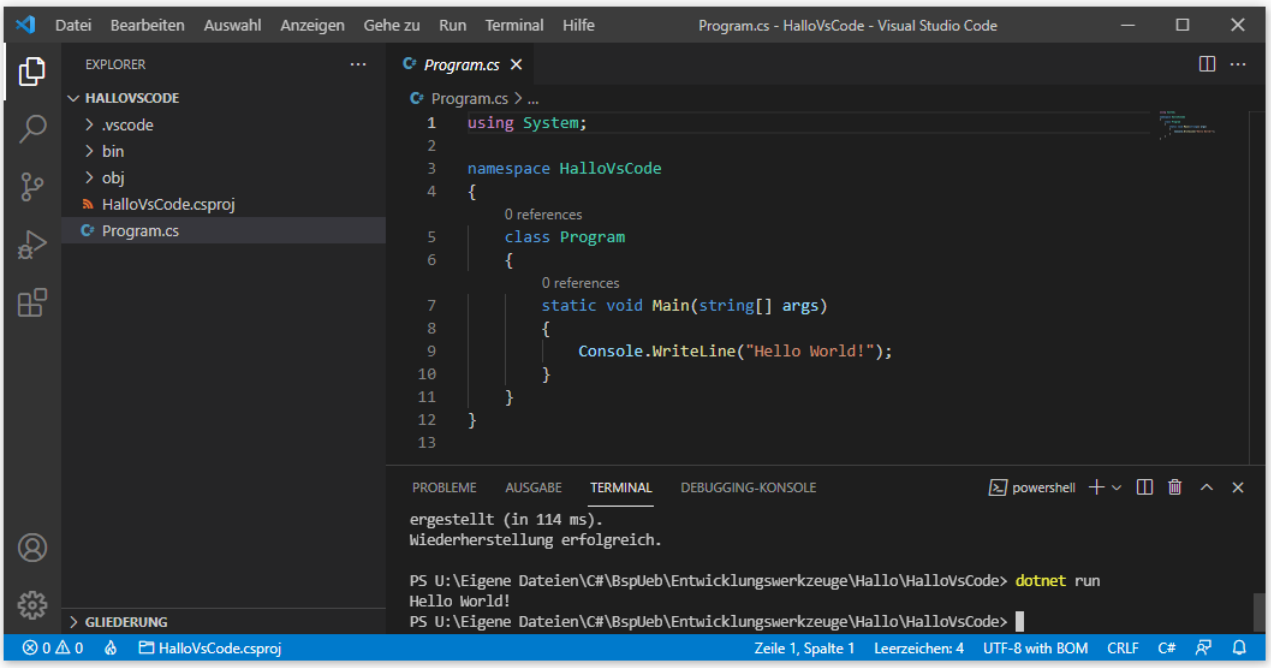
- Menübefehl **Anzeigen > Befehlspalette**
- Befehl:
- `.NET: Generate Assets for Build and Debug`
- **Enter**-Taste

3.4.4.3 Programm ausführen

Schicken Sie zum Starten des Programms im **TERMINAL** den folgenden Befehl ab:

```
dotnet run
```

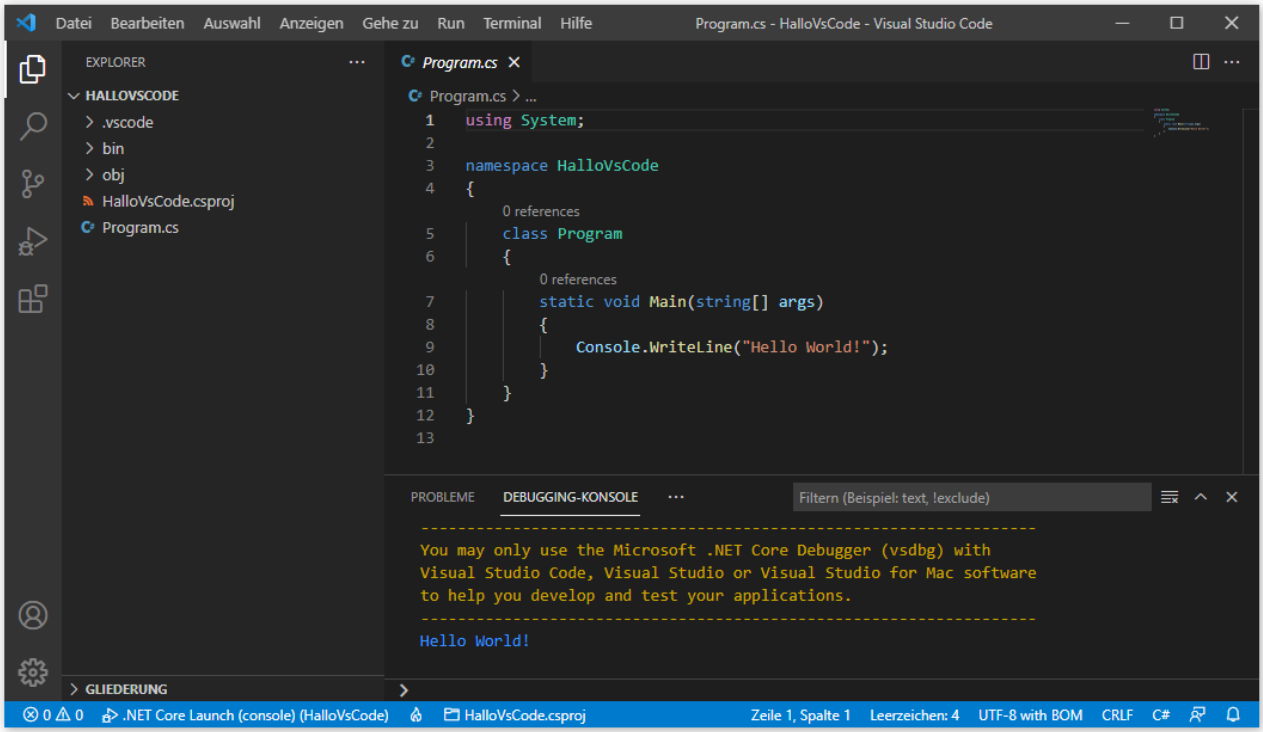
Die Ausgabe des Programms erscheint im **TERMINAL**:



Das Programm lässt sich auch ohne **TERMINAL** starten:

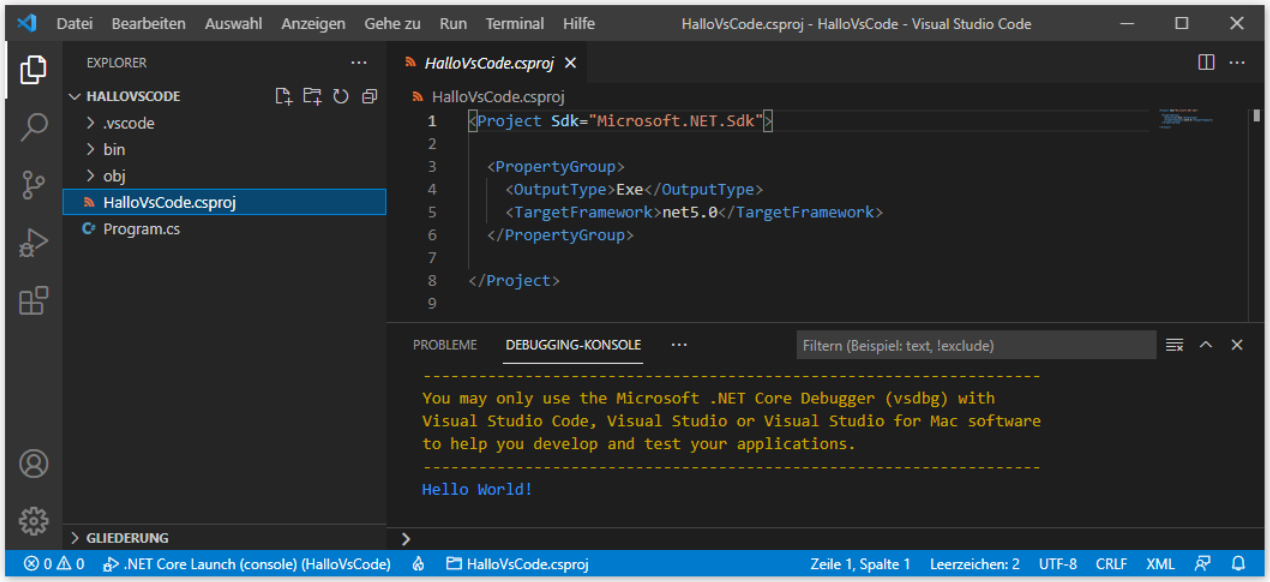
- Menübefehl **Ausführen > Ohne Debuggen ausführen**
- Tastenkombination **Strg-F5**

Diesmal erscheint die Ausgabe in der **DEBUGGING-KONSOLE**:



3.4.4.4 Projektkonfiguration

Um die Einstellungen für ein Projekt zu überprüfen oder zu verändern, editiert man die Projektkonfigurationsdatei (Namenserweiterung **.csproj**), z. B.:



Im Beispiel ist zu erfahren:

- Im Projekt wird ein Exe-Assembly erstellt.
- Das **TargetFramework** ist .NET 5.0.

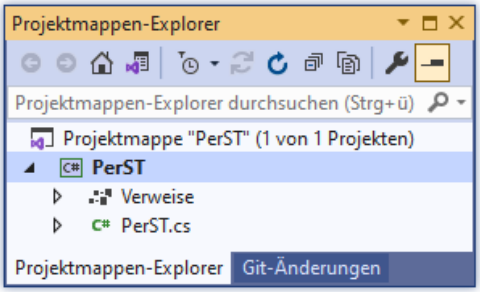
3.5 Hinweise zu den Projekten mit Beispielen und Übungen

In der zum Manuskript gelieferten Datei **BspUeb.zip** (siehe Vorwort zum Bezug) befinden sich zahlreiche, mit dem Visual Studio Community 2019 erstellte Projekte mit Beispielen und Lösungsvorschlägen zu Übungsaufgaben.

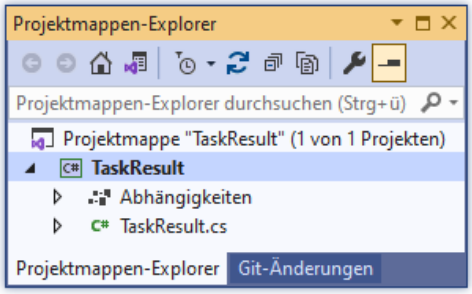
Während des Kurses, aus dem das Manuskript entstanden ist, sind .NET 5.0 und die davon abhängige C# - Version 9.0 erschienen, sodass während des Kurses ein Umstieg von der zuvor bevorzugten Zielplattform .NET Framework 4.8 (mit C# 7.3) auf .NET 5.0 (mit C# 9.0) naheliegend war. Für die frühen, z. B. mit elementaren Sprachelementen beschäftigten Kapitel des Manuskripts resultiert aus der Beschränkung auf C# 7.3 kein Nachteil, weil die Spracherweiterungen hier keine Rolle spielen.

Mit der Zielplattform hat sich auch die Projektordnerstruktur im Visual Studio geändert, sodass in der Sammlung von Beispielen und Musterlösungen (in der Datei **BspUeb.zip**) Projekte im .NET Framework 4.8 - Format und Projekte im .NET 5.0 - Format koexistieren. Der Projektmappen-Explorer lässt auf den ersten Blick die Zielplattform eines Projekts erkennen, z. B.:

.NET Framework 4.8 - Projekt



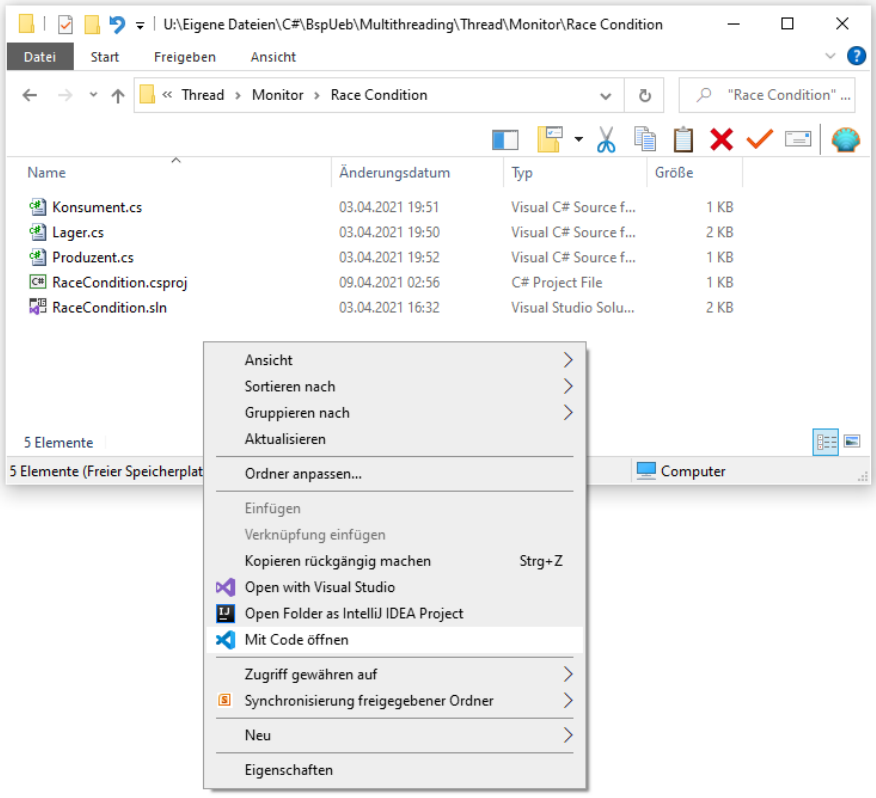
.NET 5.0 - Projekt



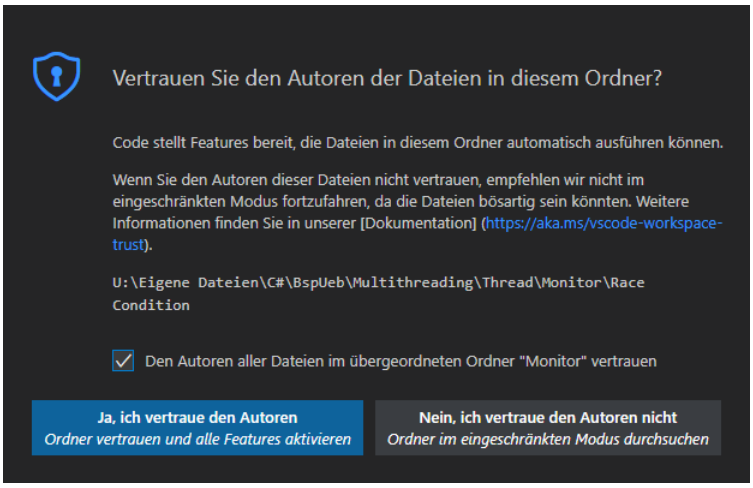
Beide Projektformate lassen sich mit dem Visual Studio 2019 per Doppelklick auf die **csproj**- oder auf die **sln**-Datei im Projektordner problemlos öffnen.

Mit dem VS Code lassen sich leider nur die Kurs-Projekte im .NET 5.0 - Format auf einfache Weise öffnen und ausführen:

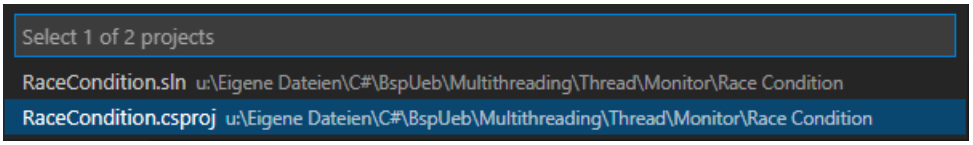
- Wählen Sie aus dem Kontextmenü zum Projektordner die Option **Mit Code öffnen**, z. B.:



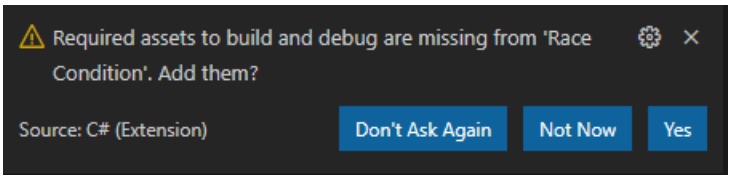
- Bestätigen Sie Ihr Vertrauen gegenüber den Autoren der im Projektordner sowie im übergeordneten Ordner befindlichen Dateien:



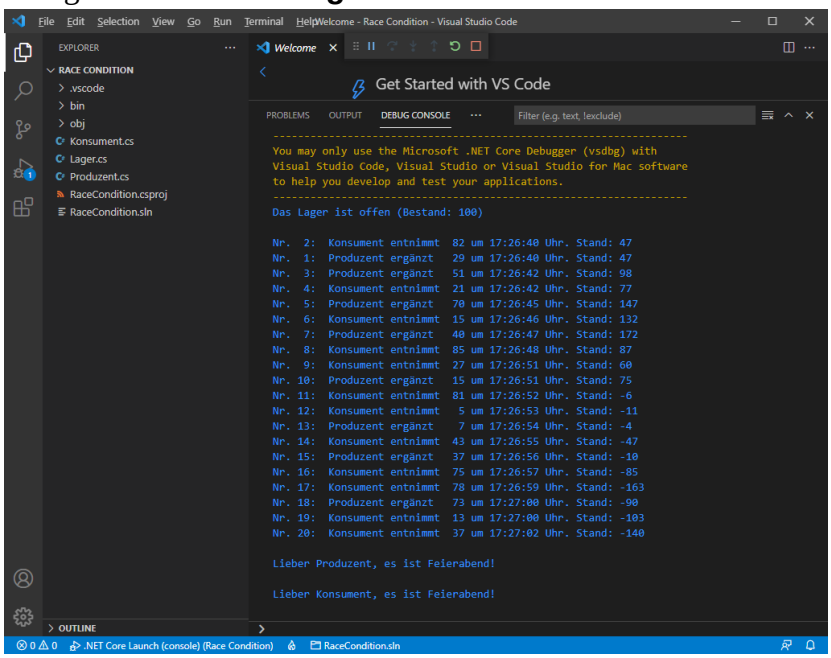
- Wählen Sie die Projektdatei (Namenserweiterung **.csproj**):



- Die folgende Anfrage sollte mit **Yes** beantwortet werden, damit im Projektunterordner **.vscode** die Konfigurationsdateien **launch.json** und **tasks.json** entstehen, die zum Starten der Anwendung benötigt werden:



- Starten Sie das Programm z. B. mit **Strg+F5**:



Von den Kurs-Projekten im .NET Framework 4.8 - Format können nur die Quellcodedateien in VS Code geöffnet werden. Leider lassen sich Visual Studio - Projekte für das .NET Framework 4.8 nur mit Handarbeit in Projekte für .NET 5.0 konvertieren.

Bei der Übernahme von Quellcode aus der PDF-Datei via Windows-Zwischenablage geht leider die Formatierung verloren. Optische Verluste kann das Visual Studio beim Einfügen durch die automatische Formatierung wieder egalisieren. Wenn allerdings beim Transfer Leerzeichen aus einer Zeichenfolge entfernt worden sind, dann verhält sich der eingefügte Quellcode nicht mehr korrekt. Daher sollte der Quellcode in der Regel aus dem ausgepackten Zip-Archiv **BspUeb.zip** übernommen werden. Seine Ordner bilden ungefähr die Kapitel und Abschnitte des Skripts ab.

3.6 Übungsaufgaben zum Kapitel 3

1) Installieren Sie nach Möglichkeit auf Ihrem privaten PC das Visual Studio Community 2019 (gemäß Abschnitt 3.1) und/oder Visual Studio Code (gemäß Abschnitt 3.4).

2) Experimentieren Sie mit dem `Hallo`-Beispielprogramm im Abschnitt 3.2.1:

- Ergänzen Sie weitere Ausgabeanweisungen.
- Erstellen Sie eine Variante ohne **using**-Direktive (vgl. Abschnitt 2.6).

3) Beseitigen Sie die Fehler in der folgenden Variante des `Hallo`-Programms:

```
Using System;
class Hallo {
  static void Moin() {
    Console.WriteLn("Hallo, echt .NET hier!);
  }
```

4 Elementare Sprachelemente

Im Kapitel 1 wurde anhand eines halbwegs realistischen Beispiels versucht, einen ersten Eindruck von der objektorientierten Software-Entwicklung mit C# zu vermitteln. Nun erarbeiten wir uns die Details der Programmiersprache C# und beginnen dabei mit elementaren Sprachelementen. Diese dienen zur Realisation von Algorithmen innerhalb von Methoden und sehen in C# nicht wesentlich anders aus als in älteren, *nicht* objektorientierten Sprachen (z. B. in C).

4.1 Einstieg

4.1.1 Aufbau von einfachen C# - Programmen

Sie haben schon einiges über den Aufbau von C# - Programmen erfahren:

- Ein C# - Programm besteht aus **Klassen**. Für das Bruchrechnungsbeispiel im Kapitel 1 wurden die Klassen `Bruch` und `Bruchaddition` definiert. In den Methoden der beiden Klassen kommen Klassen aus der Base Class Library (BCL) zum Einsatz (**Console**, **Math** und **Convert**).
- Eine **Klassendefinition** besteht aus …
  - dem **Kopf**
    Er enthält nach dem Schlüsselwort **class** den Namen der Klasse. Soll eine Klasse für beliebige andere Klassen (in fremden Assemblies) nutzbar sein, dann muss dem Schlüsselwort **class** der Zugriffsmodifikator **public** vorangestellt werden, z. B.:
    ```
    public class Bruch {
      . . .
    }
    ```
  - und dem **Rumpf**
    Begrenzt durch ein Paar geschweifter Klammern befinden sich hier …
    - die Deklarationen der **Felder**
    - die Definitionen der **Methoden** und **Eigenschaften**.
- Auch eine **Methodendefinition** besteht aus …
  - dem **Kopf**
    Hier werden vereinbart: Modifikatoren (z. B. mit der Sichtbarkeit **public**), Rückgabetyp, Name der Methode und Parameterliste. All diese Bestandteile werden noch ausführlich erläutert.
  - und dem **Rumpf**
    Begrenzt durch ein Paar geschweifter Klammern befinden sich hier **Anweisungen** zur Realisation von Algorithmen, durch die z. B. Instanzvariablen des agierenden Objekts verändert werden, wobei lokale Variablen zum Speichern von Zwischenergebnissen zum Einsatz kommen. Der Unterschied zwischen Instanzvariablen (Merkmalen von Objekten), statischen Variablen (Merkmalen von Klassen) und lokalen Variablen von Methoden wird im Abschnitt 4.3 erläutert.

  Details zur Definition einer Methode und zur Definition einer Eigenschaft, die als Paar von Methoden für den lesenden und den schreibenden Zugriff auf ein Feld aufgefasst werden kann, folgen im Abschnitt 4.1.2.
- Eine **Anweisung** ist die kleinste ausführbare Einheit eines Programms. In C# sind bis auf wenige Ausnahmen alle Anweisungen mit einem **Semikolon** abzuschließen.

- Von den Klassen eines Programms muss eine **startfähig** sein. Dazu benötigt sie eine Methode mit dem Namen **Main()** und folgenden Besonderheiten:
  - Modifikator **static**
  - Rückgabetyp **int** oder **void**

  Diese Methode wird beim Programmstart vom Laufzeitsystem (von der CLR) aufgerufen. Wenn die **Main()** - Methode ihrem Aufrufer (also der CLR bzw. dem Betriebssystem) per **return**-Anweisung (siehe Abschnitt 5.3.1.2) eine ganze Zahl als Information über den (Miss)erfolg ihrer Tätigkeit liefern möchte (z. B. 0: alles gut gegangen, 1: Fehler), dann ist in der Methodendefinition der Rückgabetyp **int** anzugeben.[1] Fehlt eine solche Rückgabe, ist der Pseudorückgabetyp **void** anzugeben. Beim Bruchadditions-Beispiel im Kapitel 1 ist die Klasse `Bruchaddition` startfähig. Ihre **Main()** - Methode verwendet den Pseudorückgabetyp **void**.
- Meist verwendet man für den Quellcode einer Klasse jeweils eine eigene Datei mit demselben Namen wie die Klasse und **.cs** als Namenserweiterung.
- Die ersten Zeilen einer C# - Quellcodedatei enthalten meist **using**-Direktiven zum Import von **Namensräumen**, damit die dortigen Klassen später ohne Namensraumpräfix vor dem Klassennamen angesprochen werden können.
- Die zu einem Programm gehörigen Quellcodedateien werden gemeinsam vom **Compiler** in die **Intermediate Language** (IL) übersetzt, z. B.:

  ```
  >csc  Bruch.cs  Bruchaddition.cs
  ```

  Das resultierende Assembly übernimmt seinen Namen per Voreinstellung von der Datei mit der Startklasse und enthält neben dem IL-Code auch Typ- und Assembly-Metadaten.

Während der Beschäftigung mit elementaren C# - Sprachelementen werden wir mit einer sehr einfachen und nicht sonderlich objektorientierten Programmstruktur arbeiten, die Sie schon aus dem `Hallo`-Beispiel kennen (siehe z. B. Abschnitt 3.2.1). Es wird nur *eine* Klasse definiert, und diese Klasse enthält nur eine einzige Methodendefinition. Weil die Klasse startfähig sein muss, liegt **Main** als Name der Methode fest, und wir erhalten die folgende Programmstruktur:

```
using System;
class Prog {
   static void Main() {
      // Platz zum Üben elementarer Sprachelemente
   }
}
```

Damit die wenig objektorientierten Beispiele Ihren Programmierstil nicht prägen, wurde zu Beginn des Kurses (im Kapitel 1) eine Anwendung vorgestellt, die bereits etliche OOP-Prinzipien realisiert.

Wie sich im Visual Studio Konsolenprojekte für das .NET Framework 4.x bzw. für .NET 5.0 erstellen lassen, die zum Üben der elementaren Sprachelemente geeignet sind, wurde im Abschnitt 3.3.3 beschrieben. Fertige Übungsprojekte sind hier zu finden:

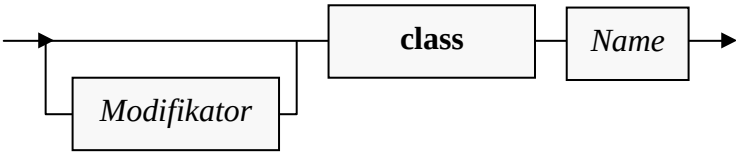
…\BspUeb\Elementare Sprachelemente\Prog
…\BspUeb\Elementare Sprachelemente\Prog5

[1] Nach einem beendeten Programmeinsatz in einem per **cmd.exe** gestarteten Konsolenfenster befindet sich der **return**-Wert in der Pseudo-Umgebungsvariablen **errorlevel**.

4.1.2 Syntaxdiagramm

Um für C# - Sprachbestandteile (z. B. Definitionen oder Anweisungen) die Bildungsvorschriften kompakt und genau zu beschreiben, werden wir im Kurs u. a. sogenannte **Syntaxdiagramme** einsetzen, für die folgende Vereinbarungen gelten:

- Man bewegt sich in Pfeilrichtung durch das Syntaxdiagramm und gelangt dabei zu Rechtecken, welche die an der jeweiligen Stelle zulässigen Sprachbestandteile angeben, wie z. B. im folgenden Syntaxdiagramm zum Kopf einer Klassendefinition:



- Für **konstante** (**terminale**) Sprachbestandteile, die aus einem Rechteck exakt in der angegebenen Form in konkreten Quellcode zu übernehmen sind, wird **fette** Schrift verwendet.
- *Platzhalter* sind an *kursiver* Schrift zu erkennen. Im konkreten Quellcode muss anstelle des Platzhalters eine zulässige Realisation stehen, und die zugehörigen Bildungsregeln sind an anderer Stelle (z. B. in einem anderen Syntaxdiagramm) erklärt.
- Bei einer Verzweigung kann man sich für eine Richtung entscheiden, wenn nicht per Pfeil eine Bewegungsrichtung vorgeschrieben ist. Zulässige Realisationen zum obigen Segment sind also z. B.:
  - `class Bruchaddition`
  - `public class Bruch`

  Verboten sind hingegen z. B. folgende Sequenzen:
  - `class public Bruchaddition`
  - `Bruchaddition public class`
- Als Klassenmodifikator ist uns bisher nur der Zugriffsmodifikator **public** begegnet, der für die allgemeine Verfügbarkeit einer Klasse (in beliebigen Assemblies) sorgt. Später werden Sie noch weitere Klassenmodifikatoren kennenlernen. Sicher kommt niemand auf die Idee, z. B. den Modifikator **public** mehrfach zu vergeben und damit gegen eine Syntaxregel zu verstoßen. Das obige (möglichst einfach gehaltene) Syntaxdiagrammsegment lässt diese offenbar sinnlose Praxis zu. Es bieten sich zwei Lösungen an:
  - Das Syntaxdiagramm mit einem gesteigerten Aufwand an Formalismus präzisieren.
  - Durch eine generelle Zusatzregel die Mehrfachverwendung eines Modifikators verbieten.

  Im Manuskript wird die zweite Lösung verwendet.
- Bei den Syntaxdiagrammen im Manuskript wird angestrebt, dass sie möglichst übersichtlich sind und nur zulässige Syntax erlauben. Es ist hingegen *nicht* garantiert, dass jede erlaubte Syntax berücksichtigt ist.

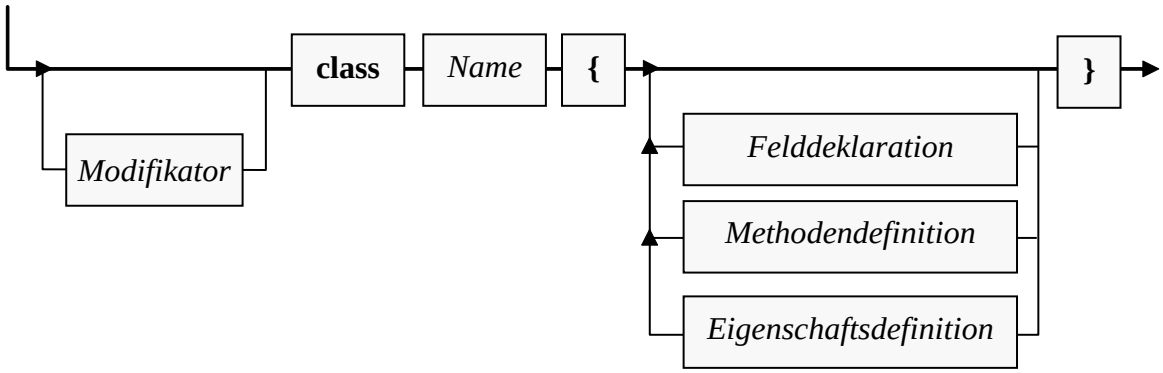
Als Beispiele betrachten wir anschließend die Syntaxdiagramme zur Definition von Klassen, Methoden und Eigenschaften. Aus didaktischen Gründen zeigen die Diagramme nur solche Sprachbestandteile, die bisher in einem Beispiel verwendet oder im Text beschrieben worden sind, sodass sie langfristig nicht als Referenz taugen. Trotz der Vereinfachung sind die Syntaxdiagramme für die meisten Leser vermutlich nicht voll verständlich, weil etliche Bestandteile noch nicht systematisch beschrieben wurden (z. B. Modifikator, Feld- und Parameterdeklaration).

Im aktuellen Abschnitt 4.1.2 geht es primär darum, Syntaxdiagramme als metasprachliche Hilfsmittel einzuführen. Vielleicht tragen aber die vorgestellten Beispiele trotz der gerade angesprochenen Kompromisse auch zur allmählichen Festigung der wichtigen Begriffe *Klasse*, *Methode* und *Eigenschaft* bei. Auf keinen Fall handelt es sich bei den nächsten drei Abschnitten um die „offizielle“ Behandlung dieser Begriffe.

4.1.2.1 Klassendefinition

Wir arbeiten vorerst mit dem folgenden, leicht vereinfachten Klassenbegriff:

Klassendefinition



Solange man sich auf zulässigen Pfaden bewegt (immer in Pfeilrichtung, eventuell auch in Schleifen), an den Stationen (Rechtecken) entweder den konstanten Sprachbestandteil exakt übernimmt oder den Platzhalter auf zulässige (an anderer Stelle erläuterte) Weise ersetzt, entsteht eine syntaktisch korrekte Klassendefinition.

Als Beispiel betrachten wir die im Kapitel 1 vorgestellte Klasse `Bruch`:

Modifikator Name

Feld-deklarationen

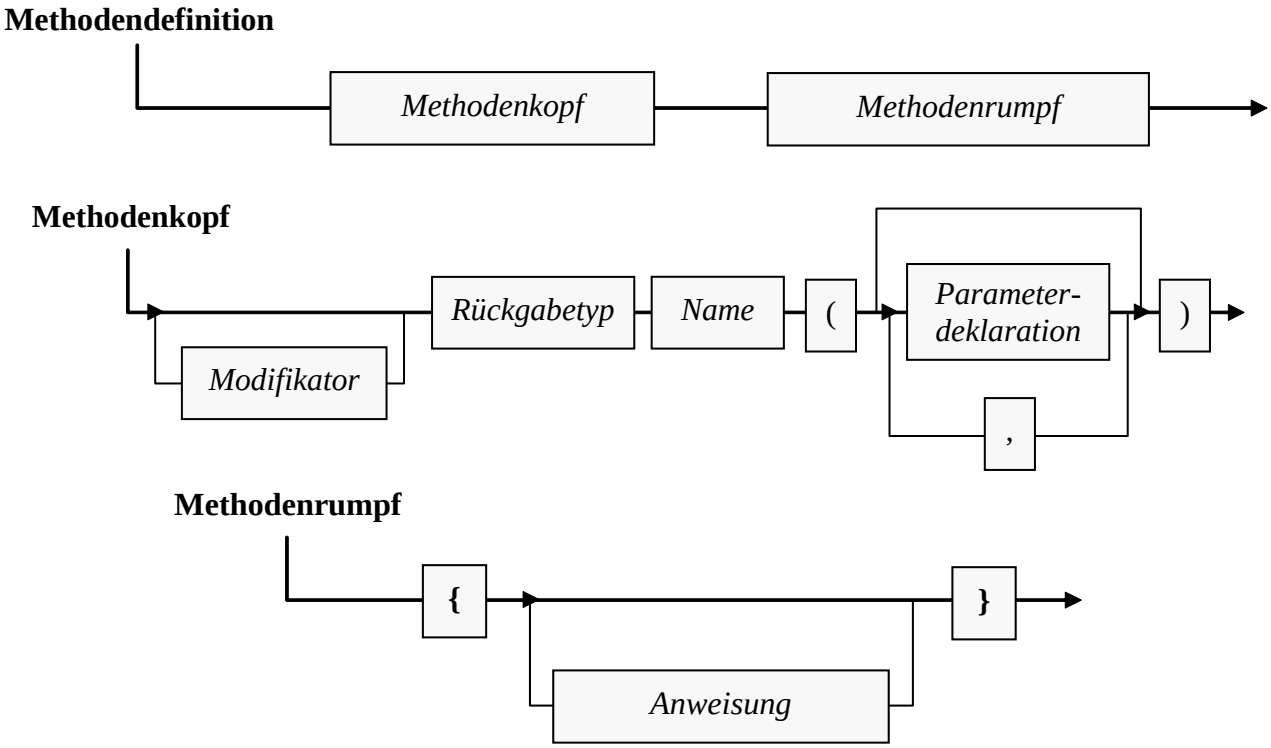
Eigenschafts-definitionen

Methoden-definitionen

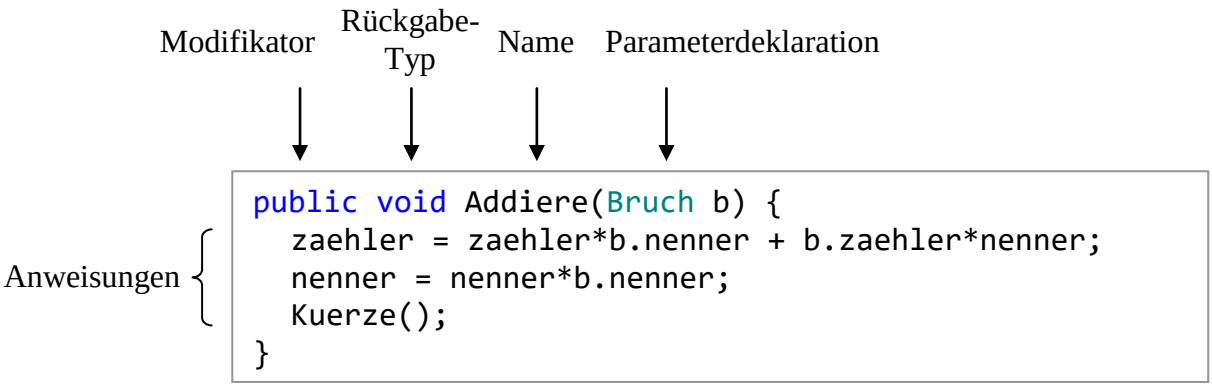
```
public class Bruch {

   int zaehler,
       nenner = 1;

   public int Zaehler {
      get {
         return zaehler;
      }
      set {
         zaehler = value;
      }
   }

   public int Nenner {
      . . .
   }

   public void Zeige() {
      Console.WriteLine("   {0}\n -----\n   {1}\n",
                        zaehler, nenner);
   }

   public void Kuerze() {
      . . .
   }

   public void Addiere(Bruch b) {
      . . .
   }

   public void Frage() {
      . . .
   }

}
```

4.1.2.2 Methodendefinition

Weil *ein* Syntaxdiagramm für die komplette Methodendefinition etwas unübersichtlich wäre, betrachten wir separate Diagramme für die Begriffe *Methodenkopf* und *Methodenrumpf*:



Als Beispiel betrachten wir die Definition der `Bruch`-Methode `Addiere()`:




```
public void Addiere(Bruch b) {
  zaehler = zaehler*b.nenner + b.zaehler*nenner;
  nenner = nenner*b.nenner;
  Kuerze();
}
```

Zur Erläuterung des Begriffs *Parameterdeklaration* beschränken wir uns vorläufig auf das Beispiel in der `Addiere()` - Definition. Es enthält:

- einen Datentyp (Klasse `Bruch`)
- und einen Parameternamen (`b`).

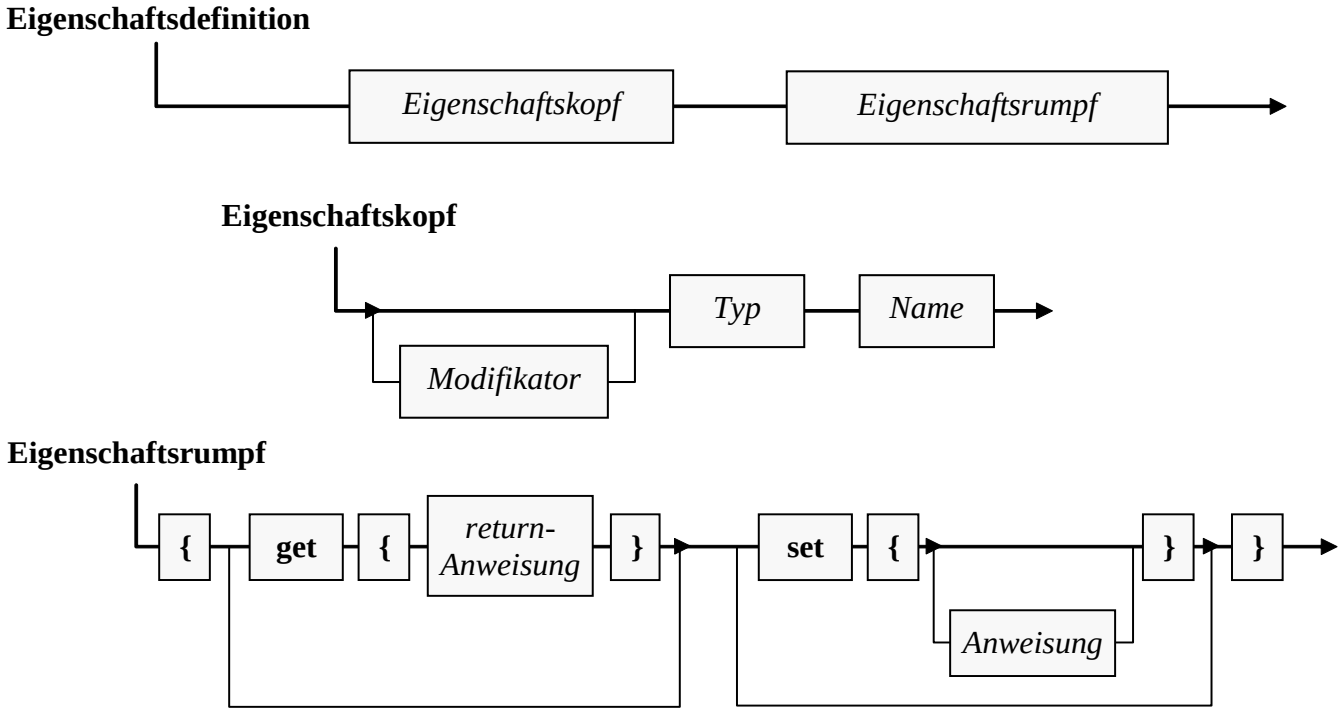
In vielen Methoden werden sogenannte *lokale Variablen* (vgl. Abschnitt 4.3.6) deklariert, z. B. in der `Bruch`-Methode `Kuerze()`:

```
public void Kuerze() {
  if (zaehler != 0) {
    int az = Math.Abs(zaehler);
    int an = Math.Abs(nenner);
      .  .  .
  }
}
```

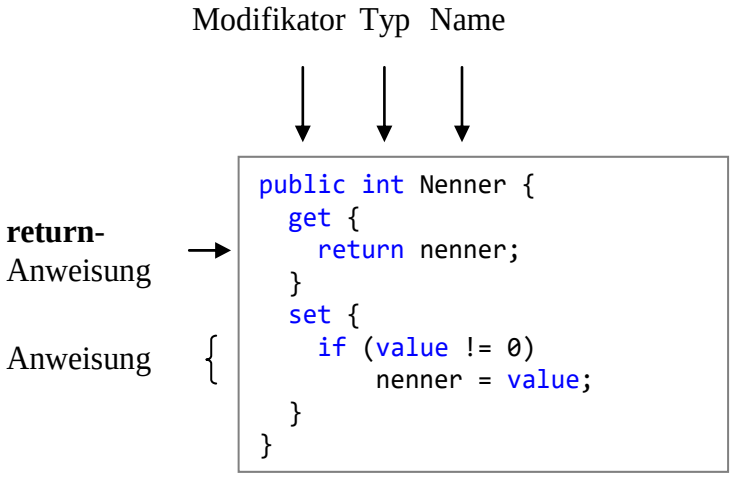
Weil wir bald u. a. die *Variablendeklarationsanweisung* kennenlernen werden, benötigt das Syntaxdiagramm zum Methodenrumpf jedoch (im Unterschied zum Klassendefinitionsdiagramm) *kein* separates Rechteck für die Variablendeklaration.

4.1.2.3 Eigenschaftsdefinition

Auch beim Syntaxdiagramm für den Eigenschaftsbegriff gehen wir schrittweise vor:



Als Beispiel betrachten wir die `Bruch`-Eigenschaft `Nenner`:



In der **set**-Definition spricht das Schlüsselwort **value** den Wert an, den der Aufrufer der Eigenschaft zuweisen möchte.

4.1.3 Hinweise zur Gestaltung des Quellcodes

Zur Formatierung von C# - Programmen haben sich Konventionen entwickelt, die wir bei passender Gelegenheit besprechen werden. Der Compiler ist hinsichtlich der Formatierung sehr tolerant und beschränkt sich auf folgende Regeln:

- Die einzelnen Bestandteile einer Definition oder Anweisung müssen in der richtigen **Reihenfolge** stehen.
- Zwischen zwei Sprachbestandteilen muss ein **Trennzeichen** stehen, wobei das Leerzeichen, das Tabulatorzeichen und der Zeilenumbruch erlaubt sind. Diese Trennzeichen dürfen sogar in beliebigen Anzahlen und Kombinationen auftreten. *Innerhalb* eines Sprachbestandteils (z. B. Namens) sind Trennzeichen (z. B. Zeilenumbruch) natürlich verboten.
- Zeichen mit festgelegter Bedeutung wie z. B. ";", "(", "+", ">" sind **selbstisolierend**, d.h. vor und nach ihnen sind keine Trennzeichen nötig (aber erlaubt).

Wer dieses Manuskript am Bildschirm liest oder an einen Farbdrucker geschickt hat, profitiert hoffentlich von der farblichen Gestaltung der Code-Beispiele. Es handelt sich um die Syntaxhervorhebungen der Entwicklungsumgebung Visual Studio, die via Windows-Zwischenablage in den Text übernommen wurden.[1]

Manche Programmierer setzen die öffnende geschweifte Klammer zum Rumpf einer Klassen-, Methoden- oder Eigenschaftsdefinition ans Ende der Kopfzeile (siehe linkes Beispiel), andere bevorzugen den Anfang der Folgezeile (siehe rechtes Beispiel):

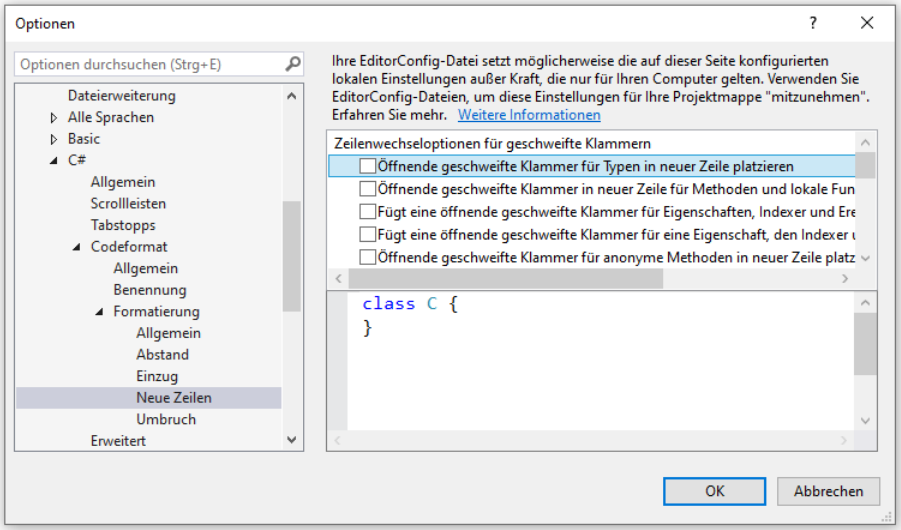
```
class Hallo {
  static void Main() {
    System.Console.WriteLine("Hallo");
  }
}
```

```
class Hallo
{
  static void Main()
  {
     System.Console.WriteLine("Hallo");
  }
}
```

Das Visual Studio verwendet per Voreinstellung die rechte Variante, kann aber nach

Extras > Optionen > Text-Editor > C# > Codeformat > Formatierung > Neue Zeilen

umgestimmt werden:



[1] Über **Extras > Optionen > Text-Editor > C# > Erweitert** kann man im Visual Studio beim **Editor-Farbschema** zwischen den Optionen **2017** und **2019** wählen. Leider gehen beim Transfer von Quellcode über die Windows-Zwischenablage einige Farben verloren. Beim Farbschema 2017 sind die Verluste weniger gravierend, sodass diese Variante für das Manuskript gewählt wurde. Nichtsdestotrotz musste die verlorene Cyan-Färbung von Typnamen manuell restauriert werden, was sicher nicht vollständig gelungen ist.

Weitere Hinweise zur übersichtlichen Gestaltung des C# - Quellcodes finden sich z. B. in Microsofts Online-Informationsangebot zu C#.[1]

4.1.4 Kommentar

C# bietet folgende Möglichkeiten, den Quelltext zu kommentieren:

- **Zeilenrestkommentar**
  Alle Zeichen vom ersten doppelten Schrägstrich (**//**) bis zum Ende der Zeile gelten als Kommentar, z. B.:

```
private int zaehler; // wird automatisch mit 0 initialisiert
```

  Im Beispiel wird eine Variablendeklarationsanweisung in derselben Zeile kommentiert.

- **Mehrzeilenkommentar**
  Ein durch **/*** eingeleiteter Kommentar muss explizit durch ***/** terminiert werden. In der Regel wird diese Syntax für einen ausführlichen Kommentar verwendet, der sich über mehrere Zeilen erstreckt, z. B.:

```
/*
Ein Bruch-Objekt verhindert, dass sein Nenner auf 0
gesetzt wird, und hat daher stets einen definierten Wert.
*/
public int Nenner {
  . . .
}
```

  Ein mehrzeiliger Kommentar eignet sich auch dazu, ein Programmsegment (vorübergehend) zu deaktivieren, ohne es löschen zu müssen.

  Speziell in Beispielen für Lehrtexte wird manchmal von der Möglichkeit Gebrauch gemacht, hinter einem explizit terminierten Kommentar in derselben Zeile eine Definition oder Anweisung fortzusetzen, z. B.:

```
void meth() {/* Anweisungen */}
```

  Weil der explizit terminierte Kommentar (jedenfalls ohne farbliche Hervorhebung der auskommentierten Passage) etwas unübersichtlich ist, wird er selten verwendet.

- **Dokumentationskommentar**
  Neben den Kommentaren, welche ausschließlich das Lesen des Quelltexts unterstützen sollen, kennt C# noch den Dokumentationskommentar. Er darf vor einem benutzerdefinierten Typ (z. B. einer Klasse) oder vor einem Klassen-Member (z. B. Feld, Eigenschaft, Methode) stehen und wird in *jeder* Zeile durch drei Schrägstriche eingeleitet, z. B.:

```
/// <summary>
/// Im set-Teil der Nenner-Eigenschaft wird verhindert, dass der Nenner auf null
/// gesetzt wird.
/// </summary>
public int Nenner {
   . . .
}
```

  Durch **/**** eingeleitete und durch ***/** terminierte *mehrzeilige* Dokumentationskommentare erfordern die Beachtung spezieller Regeln und sind wegen des damit verbundenen Fehlerrisikos nicht zu empfehlen.

  Die Dokumentationskommentare in zu übersetzenden Quellcodedateien werden vom Compiler bei einem Aufruf mit der Option **doc** in eine separate XML-Dokumentationsdatei umgesetzt, z. B.:

---

[1] https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/inside-a-program/coding-conventions

```
>csc  *.cs  -doc:Bruch.xml
```

Im Visual Studio 2019 fordert man die Erstellung einer XML-Dokumentationsdatei zu einem Projekt folgendermaßen an:

Projekt > Eigenschaften > Build > XML-Dokumentationsdatei

Aus einer XML-Dokumentationsdatei kann über meist kostenlos verfügbare Hilfsprogramme eine leichter lesbare HTML-Dokumentation erstellt werden.[1] In Microsofts Entwicklungsumgebungen fehlt leider eine entsprechende Funktionalität.

Wenn man im Editor unserer Entwicklungsumgebung das Auskommentieren eines markierten Blocks mit dem Menübefehl

Bearbeiten > Erweitert > Auswahl auskommentieren

bzw. mit der Tastenkombination **Strg+K**, **Strg+C** veranlasst, dann werden doppelte Schrägstriche vor jede Zeile gesetzt.[2] Wendet man den Menübefehl

Bearbeiten > Erweitert > Auskommentierung der Auswahl aufheben

bzw. die Tastenkombination **Strg+K**, **Strg +U** auf einen zuvor mit Doppelschrägstrichen auskommentierten Block an, dann werden die Kommentarschrägstriche entfernt.

Weil der Compiler die Kommentare ignoriert und keinesfalls in den ausführbaren Code einfügt, darf ein Kommentar schutzbedürftige Informationen enthalten.

4.1.5 Namen

Für Klassen, Felder, Eigenschaften, Methoden, Parameter und sonstige Elemente eines C# - Programms benötigen wir Namen, wobei folgende Regeln gelten:

- Die Länge eines Namens ist nicht begrenzt.
  Zwar fördern kurze Namen die Übersicht im Quellcode, doch ist die Verständlichkeit eines Namens noch wichtiger als die Kürze.
- Das erste Zeichen muss ein Buchstabe oder ein Unterstrich sein, danach dürfen außerdem auch Ziffern auftreten.
- C# - Programme werden intern im **Unicode**-Zeichensatz dargestellt. Daher erlaubt C# im Unterschied zu anderen Programmiersprachen in Namen auch Umlaute oder sonstige nationale Sonderzeichen, die als Buchstaben gelten.
- Die Groß-/Kleinschreibung ist *signifikant*. Für den C# - Compiler sind also z. B.

  ```
  Anzahl   anzahl   ANZAHL
  ```

  grundverschiedene Namen.
- Die folgenden **reservierten Schlüsselwörter** dürfen nicht als Namen verwendet werden:[3]

---

[1] Auf der folgenden Stack Overflow - Seite werden einige Hilfsprogramme erwähnt (z. B. *Sandcastle*): https://stackoverflow.com/questions/21589870/how-to-convert-the-c-sharp-xml-documentation-generated-by-the-compiler-to-some-m

[2] Ist allerdings nur ein Teil *einer* Zeile markiert, dann bewirkt die Tastenkombination **Strg+K**, **Strg+C** eine Wandlung in einen explizit terminierten Kommentar.

[3] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/index

abstract	as	base	bool	break	byte	case	catch	char
checked	class	const	continue	decimal	default	delegate	do	double
else	enum	event	explicit	extern	false	finally	fixed	float
for	foreach	goto	if	implicit	in	int	interface	internal
is	lock	long	namespace	new	null	object	operator	out
override	params	private	protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string	struct	switch	this
throw	true	try	typeof	uint	ulong	unchecked	unsafe	ushort
using	virtual	void	volatile	while				

Um ein reserviertes Wort doch als Name verwenden zu können, muss das @-Zeichen vorangestellt werden. Im folgenden Beispiel wird auf diese Weise der Variablenname `@object` realisiert:

```
object @object = new object();
Console.WriteLine(@object);
```

- Daneben gibt es **kontextbezogen-reservierte Schlüsselwörter**, die nur in bestimmten Kontexten als Namen verboten sind:[1]

add	and	alias	args	ascending	async	await	by	descending
dynamic	equals	from	get	global	group	init	into	join
let	managed	nameof	nint	not	notnull	nuint	on	or
orderby	partial	record	remove	select	set	unmanaged	value	var
when	where	with	yield					

  Vorsichtshalber sollte man sie generell als Namen vermeiden.
- Namen müssen im aktuellen Kontext (siehe unten) eindeutig sein.

Während Sie obige Regeln einhalten *müssen*, ist die Beachtung der folgenden Konventionen freiwillig, aber empfehlenswert (vgl. Mössenböck 2019, S. 14):

- Die Namen von *lokalen* (methodeninternen) Variablen (siehe Abschnitt 4.3.3), Methodenparametern und *privaten* (gekapselten, nur klassenintern ansprechbaren) Feldern werden *klein* geschrieben, z. B.:

`az`	lokale Variable in der `Bruch`-Methode `Kuerze()`
`b`	Parameter in der `Bruch`-Methode `Addiere()`
`nenner`	privates Feld in der Klasse `Bruch`

  Sonstige Namen (z. B. von Klassen, Methoden oder Eigenschaften) beginnen mit einem großen Buchstaben, z. B.:

`Bruch`	Name einer Klasse
`Kuerze()`	Name einer Methode
`Nenner`	Name einer Eigenschaft

- Bei zusammengesetzten Namen beginnt jedes Wort mit einem Großbuchstaben (*Pascal Casing*), z. B.:[2]

  `WriteLine()`

---

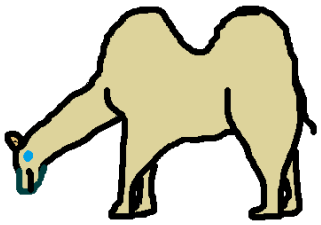
[1] https://msdn.microsoft.com/en-us/library/the35c6y.aspx, https://msdn.microsoft.com/en-us/library/bb310804.aspx

[2] Ob diese Benennungskonvention auf die altehrwürdige Programmiersprache Pascal zurückgeht, lässt sich nicht eindeutig belegen. Wir beschäftigen uns nicht weiter mit der etymologischen Frage (nach der Wortherkunft).

Eine Ausnahme stellen die nach obiger Empfehlung mit einem Kleinbuchstaben zu beginnenden Namen dar (*Camel Casing*), z. B.:

```
numberOfObjects
```

Das zur Vermeidung von Urheberrechtsproblemen handgemalte Tier kann hoffentlich trotz ästhetischer Mängel zur Klärung des Begriffs *Camel Casing* beitragen:



Tritt eine aus zwei Buchstaben bestehende Abkürzung als Namensbestandteil auf, dann werden bei Verwendung von Pascal Casing *beide* Buchstaben groß geschrieben, z. B.:

```
DBConnector
```

Bei Verwendung von Camel Casing werden beide Buchstaben klein geschrieben, z. B.:

```
dbConnector
```

Gelegentlich wird bei zusammengesetzten Namen der Unterstrich zur Verbesserung der Lesbarkeit verwendet, was der folgende Methodenname demonstriert, den das Visual Studio 2019 im `DmToEuro`-Beispielprogramm erstellt hat (siehe Abschnitt 3.3.4.3):

```
private void Button_Click(object sender, RoutedEventArgs e)
```

Ausgehend von der folgenden Startseite gibt Microsoft detaillierte Empfehlungen zur Verwendung von Namen:

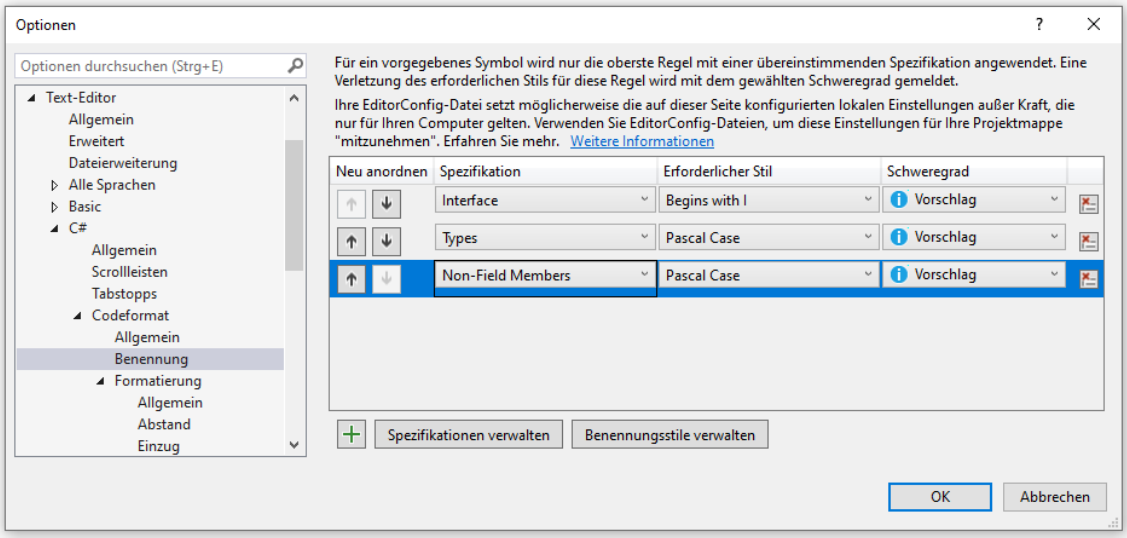
https://docs.microsoft.com/de-de/dotnet/standard/design-guidelines/naming-guidelines

Dort wird mehrfach davon abgeraten, Unterstriche zum Separieren von Namensbestandteilen zu verwenden. Der im Visual Studio 2019 enthaltene Assistent für WPF-Projekte hat sich bei der Kreation des Methodennamens `Button_Click()` allerdings nicht an diese Konvention gehalten (siehe oben).

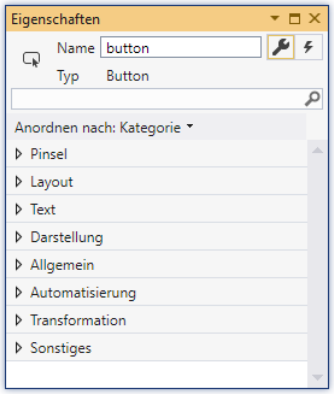
Das Camel Casing wird von Microsoft auf den oben zitierten Webseiten nur für Parameternamen empfohlen. Dazu passend enthält das Visual Studio 2019 in dem über

Extras > Optionen >Text-Editor > C# > Codeformat > Benennung

erreichbaren Einstellungsdialog die folgenden Voreinstellungen zur Code-Beurteilung



Lässt man den Assistenten für WPF-Projekte zu einem **Button**-Objekt, das durch einen klein geschriebenen Feldnamen ansprechbar ist,



eine **Click**-Behandlungsmethode erstellen, dann resultiert ein Methodenname, der mit einem Kleinbuchstaben startet. Aufgrund der dritten Regel im obigen Dialog kritisiert anschließend das Visual Studio seine Produktion (erkennbar an einer punktierten Unterstreichung):

```
private void button_Click_1(object sender, RoutedEventArgs e) {

}
```

Man könnte den klein geschriebenen Feldnamen als Auslöser für die Inkonsistenz betrachten. Allerdings wird dieser Bezeichner nicht kritisiert:[1]

```
internal System.Windows.Controls.Button button;
```

4.1.6 Übungsaufgaben zum Abschnitt 4.1

1) Welche Varianten der Methode **Main()** sind zum Starten eines Programms geeignet?

```
static void main() { . . . }
public static void Main() { . . . }
static int Main() { . . . }
static double Main() { . . . }
static void Main() { . . . }
```

2) Welche von den folgenden Namen sind unzulässig?

```
4you   maiLink   else   Lösung   b_3
```

4.2 Ausgabe bei Konsolenanwendungen

Eine formatierte Konsolenausgabe lässt sich in C# recht bequem über die Methode **Console.WriteLine()** erzeugen. Das folgende Beispiel stammt aus der Methode `Zeige()` unserer Klasse `Bruch` (siehe Abschnitt 1.3):

```
Console.WriteLine("   {0}\n  -----\n   {1}\n", zaehler, nenner);
```

Es handelt es sich um eine statische Methode der Klasse **Console** aus dem Namensraum **System**, d.h.:

---

[1] In welcher Quellcode-Datei die `button`-Deklaration zu finden ist, wird später im Kapitel 12 über WPF verraten.

- Der Namensraum **System** muss am Beginn der Quelle per **using**-Direktive importiert werden, um die Klasse **Console** ohne Namensraumpräfix ansprechen zu können.
- Weil es sich um eine **statische** Methode handelt, richten wir den Methodenaufruf nicht an ein **Console**-*Objekt*, sondern an die Klasse selbst.
- Im Methodenaufruf sind Klassen- und Methodenname durch einen **Punkt** zu trennen.

WriteLine() schließt jede Ausgabe automatisch mit einer Zeilenschaltung ab. Wo dies unerwünscht ist, setzt man die ansonsten äquivalente **Console**-Methode **Write()** ein.

Sie kennen bereits zwei nützliche Spezialisierungen der **WriteLine()** - Methode (später werden wir von *Überladungen* sprechen):

- Im obigen Beispiel ist die *formatierte* Ausgabe von zwei Werten zu sehen, wobei ein einleitender Zeichenfolgen-Parameter angibt, wie die Ausgabe der restlichen Parameter erfolgen soll. Auf diese Technik gehen wir im Abschnitt 4.2.2 näher ein.
- Oft reicht die im Abschnitt 4.2.1 behandelte Ausgabe einer (zusammengesetzten) Zeichenfolge.

4.2.1 Ausgabe einer (zusammengesetzten) Zeichenfolge

Im `Hallo`-Beispiel (vgl. Abschnitt 3.2.1) haben wir der **WriteLine()** - Methode als einzigen Parameter eine Zeichenkette zur Ausgabe auf dem Bildschirm übergeben:

```
Console.WriteLine("Hallo, echt .NET hier!");
```

Übergebene Argumente anderen Typs werden vor der Ausgabe automatisch in eine Zeichenfolge konvertiert, z. B. der Wert einer ganzzahligen Variablen (siehe unten):

```
int i = 4711;
Console.WriteLine(i);
```

Besonders angenehm ist die Möglichkeit, mehrere Teilausgaben mit dem „+“ - Operator zu verketten, z. B.:

```
int i = 4711;
Console.WriteLine("i hat den Wert: " + i);
```

Der Wert der ganzzahligen Variablen `i` wird in eine Zeichenfolge gewandelt, die anschließend an die Zeichenfolge `"i hat den Wert: "` angehängt und mit ihr zusammen ausgegeben wird:

Durch die bequeme Zeichenfolgenverkettung mit dem „+“ - Operator und die automatische Konvertierung von beliebigen Datentypen in eine Zeichenfolge ist die **WriteLine()** - Ausgabe recht flexibel. Außerdem erlauben die folgenden **Escape-Sequenzen** (vgl. Abschnitt 4.3.10.4), die wie gewöhnliche Zeichen in die Ausgabezeichenfolge geschrieben werden, eine Gestaltung der Ausgabe:

\n	Zeilenwechsel (*new line*)
\t	Horizontaler Tabulator

Beispiel:

Quellcode-Fragment	Ausgabe

`int i = 47, j = 1771;`
`Console.WriteLine("Ausgabe:\n\t" + i + "\n\t" + j);`
	`Ausgabe:`
	`47`
	`1771`

Noch mehr Gestaltungsmöglichkeiten bietet die im nächsten Abschnitt behandelte formatierte Ausgabe.

4.2.2 Formatierte Ausgabe

Die gleich zu beschreibenden Formatierungstechniken sind nicht nur bei Konsolenausgaben zu gebrauchen. Wir werden später auf analoge Weise mit der statischen Methode **Format()** der Klasse **String** formatierte Zeichenfolgen erzeugen, die z. B. im Rahmen einer grafischen Bedienoberfläche zum Einsatz kommen (siehe Übungsaufgabe im Abschnitt 4.3.11).

4.2.2.1 Traditionelle Variante mit Platzhaltern

Bei der traditionellen Variante der formatierten Ausgabe per **WriteLine()** oder **Write()** wird als erster Parameter eine Zeichenfolge übergeben, die Platzhalter mit optionalen Formatierungsangaben für die restlichen, auf der Konsole auszugebenden Parameter enthält, z. B.:

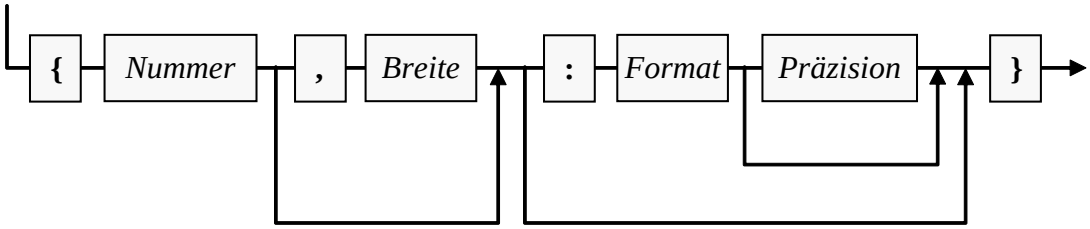
```
Console.WriteLine("   {0}\n  -----\n   {1}\n", zaehler, nenner);
```

Bei einem Bruch mit dem Zähler 5 und dem Nenner 8 resultiert die Ausgabe:

```
  5
-----
  8
```

Für einen Platzhalter ist folgende Syntax vorgeschrieben:

Platzhalter für die formatierte Ausgabe



Darin bedeuten:

Nummer	Fortlaufende Nummer des auszugebenden Arguments, bei 0 beginnend
Breite	Ausgabebreite für das zugehörige Argument Positive Werte bewirken eine *rechtsbündige*, negative Werte eine *linksbündige* Ausgabe.
Format	Formatspezifikation gemäß anschließender Tabelle
Präzision	Anzahl der Nachkommastellen oder sonstige Präzisionsangabe (abhängig vom Format), muss der Formatangabe unmittelbar folgen (ohne trennende Leerstellen)

Bei der Ausgabe von Zahlen werden u. a. folgende Formate unterstützt:[1]

[1] Hier dokumentiert Microsoft die Formatzeichenfolgen für Zahlen:
https://docs.microsoft.com/de-de/dotnet/standard/base-types/standard-numeric-format-strings

Format	**Beschreibung**	**Beispiele mit den Variablen** `int i = 32483; float f = 21.415926f;`	
		WriteLine-Parameterliste	Ausgabe
d, D	**Ganze Dezimalzahl**	`("{0,7:d}", i)`	`  32483`
f, F	**Festformatierte Kommazahl** Präzision: Anzahl der Nachkommastellen	`("{0,7:f2}", f)`	`  21,42`
e, E	**Exponentialnotation** Präzision: Anzahl Stellen in der Mantisse	`("{0:e}", f)` `("{0:e2}", f)`	`2,141593e+001` `2,14e+001`
ohne	Bei fehlender Formatangabe entscheidet der Compiler.	`("{0,10}", i)` `("{0,10}", f)`	`     32483` `  21,41593`

In der Formatierungszeichenfolge sind auch auszugebende gewöhnliche Zeichen und Escape-Sequenzen (vgl. Abschnitt 4.3.10.4) erlaubt:

\n	Zeilenwechsel (*new line*)
\t	Horizontaler Tabulator

Beispiel:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  int i = 47, j = 1771;`
`  double d = 3.1415926;`
`  Console.WriteLine("Feste Breite:\n{0,8}\n{1,8}\n{2,8:f3}" +`
`  "\nTabulatoren:\n\t{0}\n\t{1}\n\t{2}", i, j, d);`
`  }`
`}`
	`Feste Breite:`
	`  47`
	`  1771`
	`  3,142`
	`Tabulatoren:`
	`  47`
	`  1771`
	`  3,1415926`

Auf eine Formatierungszeichenfolge mit *k* verschiedenen Platzhalternummern müssen entsprechend viele Ausdrücke (z. B. Variablen) mit einem zum jeweiligen Platzhalterformat passenden Datentyp folgen. Eine Platzhalternummer darf in der Formatierungszeichenfolge mehrfach auftreten.

Um die geschweiften Klammern als gewöhnliche Zeichen in eine Formatierungszeichenfolge aufzunehmen, müssen sie verdoppelt werden, z. B.:

Quellcodesegment	Ausgabe
`int i = 47, j = 1771;` `Console.WriteLine("{{{0}, {1}}}", i, j);`	`{47, 1771}`

4.2.2.2 Zeichenfolgeninterpolation

Seit der C# - Version 6.0 erlaubt die sogenannte *Zeichenfolgeninterpolation* eine Vereinfachung bei der formatierten Ausgabe. Statt die formatiert auszugebenden Ausdrücke *hinter* die Formatierungszeichenfolge zu schreiben, setzt man sie *in* die Zeichenfolge an Stelle der bisher verwendeten Platzhalternummern. Damit der Compiler die neue Syntax von der traditionellen unterscheiden kann, muss ein **$**-Zeichen als Präfix vor die Formatierungszeichenfolge gesetzt werden. Im Beispielprogramm aus dem letzten Abschnitt kann der **WriteLine()** - Aufruf mit der neuen Technik übersichtlicher formuliert werden:

Hier dokumentiert Microsoft die Formatzeichenfolgen für Datum und Uhrzeit:
https://docs.microsoft.com/de-de/dotnet/standard/base-types/standard-date-and-time-format-strings

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  int i = 47, j = 1771;`
`  double d = 3.1415926;`
`  Console.WriteLine($"Feste Breite:\n{i,8}\n{j,8}\n{d,8:f3}" +`
`  $"\nTabulatoren:\n\t{i}\n\t{j}\n\t{d}");`
`  }`
`}`
	`Feste Breite:`
	`  47`
	`  1771`
	`  3,142`
	`Tabulatoren:`
	`  47`
	`  1771`
	`  3,1415926`

4.2.3 Übungsaufgaben zum Abschnitt 4.2

1) Wie ist das fehlerhafte „Rechenergebnis“ im folgenden Programm zu erklären?

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  Console.WriteLine("3,3 + 2 = " + 3.3 + 2);`
`  }`
`}`
	`3,3 + 2 = 3,32`

Sorgen Sie mit einem Paar runder Klammern dafür, dass die folgende Ausgabe erscheint.

```
3,3 + 2 = 5,3
```

Verbringen Sie nicht zu viel Zeit mit der Aufgabe, weil wir die genauen technischen Hintergründe erst im Abschnitt 4.5.10 behandeln werden.

2) Schreiben Sie ein Programm, das aufgrund der folgenden Variablendeklaration und -initialisierung

```
int i = 4711, j = 471, k = 47, m = 4;
```

mit zwei **WriteLine()** - Aufrufen diese Ausgabe produziert:

```
Rechtsbündig:
        i = 4711
        j =  471
        k =   47
        m =    4

Linksbündig:
        4711 (i)
        471  (j)
        47   (k)
        4    (m)
```

4.3 Variablen und Datentypen

Während ein Programm läuft, müssen zahlreiche Daten im Arbeitsspeicher des Rechners abgelegt werden und anschließend mehr oder weniger lange für lesende und schreibende Zugriffe verfügbar sein, z. B.:

- Die Merkmalsausprägungen eines Objekts werden aufbewahrt, solange das Objekt existiert.
- Die zur Ausführung einer Methode benötigten Daten werden bis zum Ende der Methodenausführung gespeichert.

Zum Speichern eines Werts (z. B. einer ganzen Zahl) wird eine sogenannte **Variable** verwendet, worunter Sie sich einen **benannten Speicherplatz für einen Wert mit einem bestimmten Datentyp** (z. B. Ganzzahl) vorstellen können.

Eine Variable erlaubt (bei bestehender Zugriffsberechtigung) über ihren Namen den lesenden oder schreibenden Zugriff auf den zugeordneten Platz im Arbeitsspeicher, z. B.:

```
using System;
class Prog {
   static void Main() {
      int ivar;                        // Deklaration von ivar
      ivar = 4711;                     // schreibender Zugriff auf ivar
      Console.WriteLine(ivar);         // lesender Zugriff auf ivar
   }
}
```

Als wichtige Eigenschaften einer C# - Variablen halten wir fest:

- **Name**
  Es sind beliebige Bezeichner gemäß Abschnitt 4.1.5 erlaubt.
- **Datentyp**
  Damit sind festgelegt: Zulässige Werte (hinsichtlich Art und Größe), Speicherplatzbedarf, zulässige Operationen.
- **Aktueller Wert**
- **Ort im Hauptspeicher**
  Im Unterschied zu anderen Programmiersprachen (z. B. C++) spielt in C# die Verwaltung von Speicher*adressen* praktisch keine Rolle. Wir werden jedoch später zwei wichtige Speicher*regionen* unterscheiden (*Stack* und *Heap*). Dieses Hintergrundwissen hilft z. B., wenn eine **StackOverflowException** gemeldet wird.

4.3.1 Strenge Compiler-Überwachung bei C# - Variablen

Um die Details bei der Verwaltung der Variablen im Arbeitsspeicher müssen wir uns nicht kümmern, da wir schließlich mit einer problemorientierten, „höheren“ Programmiersprache arbeiten. Allerdings verlangt C# beim Umgang mit Variablen im Vergleich zu anderen Programmier- oder Skriptsprachen einige Sorgfalt, letztlich mit dem Ziel, Fehler zu vermeiden bzw. frühzeitig zu erkennen.

4.3.1.1 Explizite Deklaration

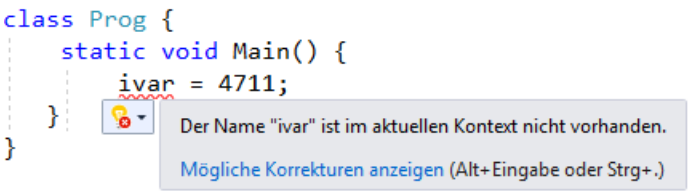
Variablen müssen **explizit deklariert** werden. In der folgenden Anweisung

```
int ivar;
```

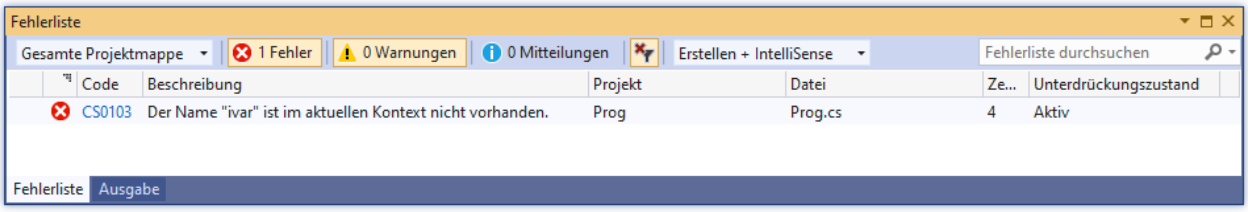
wird die Variable `ivar` vom Typ **int** (zum Speichern einer ganzen Zahl) deklariert. Wenn Sie versuchen, eine nicht deklarierte Variable zu verwenden, beschwert sich der Compiler, z. B.:

```
Prog.cs(5,3): error CS0103: Der Name "ivar" ist im aktuellen Kontext nicht
vorhanden.
```

Im Visual Studio informiert schon der Quellcode-Editor über das Problem, z. B.:



Versucht man trotzdem eine Übersetzung (z. B. angefordert über **Strg+F5**), dann erscheint die Compiler-Reklamation in der **Fehlerliste**:



Durch den Deklarationszwang werden z. B. Programmierfehler wegen falsch geschriebener Variablennamen verhindert. Auch in der Programmiersprache VB.NET (Visual Basic .NET) besteht per Voreinstellung Deklarationszwang. Hier lässt er sich jedoch mit der Compiler-Option **Option Explicit Off** aufheben. Diese *nicht* empfehlenswerte Option macht es möglich, das Verhalten vieler Skriptsprachen zu simulieren:

VB.NET - Quellcode	Ausgabe

`Option Explicit Off`
`Module Module1`
`  Sub Main()`
`    ii = 12      ' Die Variable ii wird ohne Deklaration verwendet.`
`    ij = ii + 1  ' Der Tippfehler fällt nicht auf.`
`    Console.WriteLine(ii)`
`  End Sub`
`End Module`
	`12`

4.3.1.2 Statische Typisierung

In C# ist für jede Variable bei der Deklaration ein fester (später nicht mehr änderbarer) **Datentyp** anzugeben.[1] Er legt fest, …

- welche Art von Daten (z. B. ganze Zahlen, Zeichen, Adressen von `Bruch`-Objekten) in der Variablen gespeichert werden können,
- welche Operationen auf die Variable angewendet werden dürfen.

Der Compiler kennt zu jeder Variablen den Datentyp und kann daher **Typsicherheit** garantieren, d.h. die Zuweisung von Werten mit einem unpassenden Datentyp verhindern. Außerdem kann auf (zeitaufwändige) Typprüfungen *zur Laufzeit* verzichtet werden. In der folgenden Anweisung

```
int ivar = 4711;
```

wird die Variable `ivar` vom Typ **int** deklariert, der für ganze Zahlen im Bereich von -2147483648 bis 2147483647 geeignet ist (siehe Abschnitt 4.3.4).

Seit C# 4.0 ermöglicht der Datentyp **dynamic** eine Ausnahme von der statischen Typisierung, um einige sehr spezielle Aufgaben zu erleichtern:

- Kooperation mit typfreien Skriptsprachen wie z. B. *IronPython*
- Nutzung von COM (*Component Object Model*) - APIs (z. B. zur Office-Automatisierung)
- Zugriffe auf das HTML-Dokumentobjektmodell (DOM)

An Stelle des Compilers ist hier die CLR für die Typprüfung verantwortlich, und Fehler werden zur Laufzeit entdeckt. Das ist ärgerlich für die Benutzer und peinlich für die Entwickler. Der Datentyp

---

[1] Halten Sie bitte die *statische Typisierung* (im Sinn von *unveränderlicher* Typfestlegung) in begrifflicher Distanz zu den bereits erwähnten *statischen Variablen* (im Sinn von *klassenbezogenen* Variablen). Das Wort *statisch* ist eingeführter Bestandteil bei beiden Begriffen, sodass es mir nicht sinnvoll erschien, eine andere Bezeichnung vorzunehmen, um die Doppelbedeutung zu vermeiden.

dynamic sollte nur in begründeten Ausnahmefällen verwendet werden; im Kurs kommt er nicht zum Einsatz.[1]

4.3.1.3 Initialisierung

In der folgenden Anweisung

```
int ivar = 4711;
```

erhält die Variable `ivar` bei der Deklaration gleich den Initialisierungswert 4711. Auf diese oder andere Weise müssen Sie jeder *lokalen*, d.h. innerhalb einer Methode deklarierten Variablen einen Wert zuweisen, bevor Sie zum ersten Mal lesend darauf zugreifen (vgl. Abschnitt 4.3.6). Weil ein Verstoß gegen diese Regel vom C# - Compiler verhindert wird, und weil die zu einem Objekt oder zu einer Klasse gehörigen Variablen (siehe unten) automatisch initialisiert werden, hat in C# jede Variable beim Lesezugriff stets einen definierten Wert.

4.3.2 Wert- und Referenztypen

Bei der objektorientierten Programmierung werden neben den traditionellen Variablen zur Aufbewahrung von Zahlen, Zeichen oder Wahrheitswerten auch Variablen benötigt, welche die **Speicheradresse eines Objekts** aufnehmen und die Kommunikation mit dem Objekt ermöglichen.

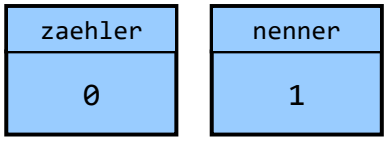
4.3.2.1 Werttypen

Die traditionellen Datentypen werden in C# als *Werttypen* (engl.: *value types*) bezeichnet. Variablen mit einem Werttyp sind auch in C# unverzichtbar (z. B. als Felder von Klassen oder als lokale Variablen in Methoden), obwohl sie „nur“ zur Verwaltung ihres Inhalts dienen und keine Rolle bei der Kommunikation mit Objekten spielen.

In der `Bruch`-Klassendefinition (siehe Kapitel 1) haben die Felder für Zähler und Nenner eines Objekts den Werttyp **int**, können also eine Ganzzahl im Bereich von –2147483648 bis 2147483647 aufnehmen. Sie werden in der folgenden Anweisung deklariert, wobei das Feld `nenner` auch noch einen Initialisierungswert erhält:

```
int zaehler, nenner = 1;
```

Beim Feld `zaehler` wird auf die explizite Initialisierung verzichtet, sodass die automatische Null-Initialisierung von Feldern greift. Für ein frisch erzeugtes `Bruch`-Objekt befinden sich im Arbeitsspeicher folgende Instanzvariablen (Felder):



In der `Bruch`-Methode `Kuerze()` tritt u. a. die lokale Variable `az` auf, die ebenfalls den Werttyp **int** besitzt:

```
int az = Math.Abs(zaehler);
```

Wie Sie bereits wissen, ist bei *lokalen* (innerhalb einer Methode deklarierten) Variablen vor dem ersten Lesezugriff eine Initialisierung erforderlich. Im Beispiel findet diese gleich bei der Deklaration statt (siehe Abschnitt 4.3.6).

---

[1] Hier finden Sie die Online-Dokumentation zum Typ **dynamic**:
https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/builtin-types/reference-types#the-dynamic-type

4.3.2.2 Referenztypen

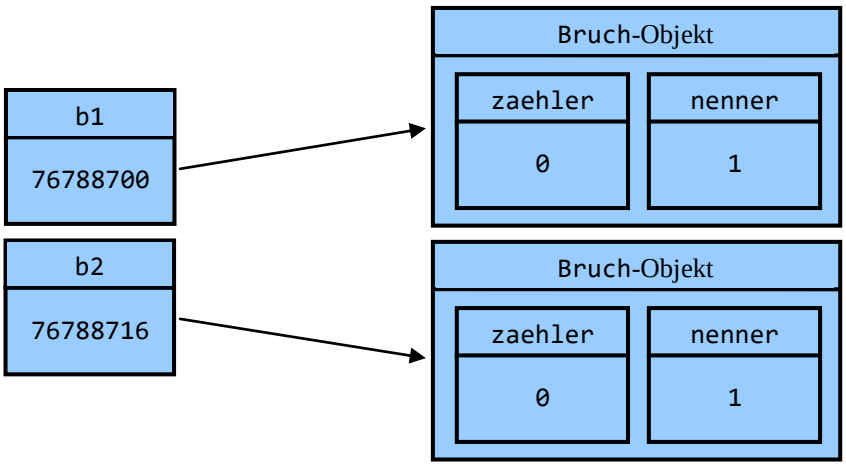
Eine Variable mit Referenztyp nimmt die **Speicheradresse eines Objekts** aus einer bestimmten Klasse auf. Sobald ein solches Objekt erzeugt und seine Speicheradresse der Referenzvariablen zugewiesen worden ist, kann das Objekt über die Referenzvariable angesprochen werden. Von den Variablen mit Werttyp unterscheidet sich eine Referenzvariable also …

- durch ihren speziellen Inhalt (Objektadresse)
- und durch ihre Rolle bei der Kommunikation mit Objekten.

Man kann jede Klasse (aus der BCL übernommen oder selbst definiert) als Datentyp verwenden, also Referenzvariablen von diesem Typ deklarieren.[1] In der **Main()** - Methode der im Abschnitt 1.5 vorgestellten Klasse `Bruchaddition` werden z. B. die Referenzvariablen `b1` und `b2` aus der Klasse `Bruch` deklariert:

```
Bruch b1 = new Bruch(), b2 = new Bruch();
```

Sie erhalten als Initialisierungswert jeweils eine Referenz auf ein (per **new**-Operator, siehe Abschnitt 5.4.2) neu erzeugtes `Bruch`-Objekt. Daraus resultiert im Arbeitsspeicher die folgende Situation:



Das von `b1` referenzierte `Bruch`-Objekt wurde bei einem konkreten Programmlauf von der CLR an der Speicheradresse 76788700 untergebracht. Wir müssen diese Adresse nicht kennen, sondern sprechen das dort abgelegte Objekt über die Referenzvariable an, z. B. in der folgenden Anweisung aus der **Main()** - Methode der Klasse `Bruchaddition`:

```
b1.Frage();
```

Jedes `Bruch`-Objekt enthält die Felder (Instanzvariablen) `zaehler` und `nenner` vom Werttyp **int**.

Zur Beziehung der Begriffe *Objekt* und *Variable* halten wir fest:

- Ein Objekt enthält im Allgemeinen mehrere Felder (Instanzvariablen) von beliebigem Typ. So enthält z. B. ein `Bruch`-Objekt die Felder `zaehler` und `nenner` vom Werttyp **int** (zur Aufnahme einer Ganzzahl). Bei einer späteren Erweiterung der `Bruch`-Klassendefinition werden ihre Objekte auch ein Feld mit Referenztyp erhalten.

---

[1] Hier wird aus didaktischen Gründen etwas gemogelt: Die später kurz erwähnten statischen Klassen lassen sich *nicht* als Datentyp verwenden (siehe Abschnitt 5.6.5).

- Eine Referenzvariable dient zur Aufnahme einer Objektadresse.
  So kann z. B. eine Variable vom Datentyp `Bruch` die Adresse eines `Bruch`-Objekts aufnehmen und zur Kommunikation mit diesem Objekt dienen. Es ist ohne weiteres möglich und oft sinnvoll, dass mehrere Referenzvariablen die Adresse *desselben* Objekts enthalten. Das Objekt existiert unabhängig vom Schicksal einer konkreten Referenzvariablen, wird jedoch überflüssig (und damit zum potentiellen Opfer des im Abschnitt 2.5 erwähnten Garbage Collectors), wenn im gesamten Programm keine einzige Referenz (Kommunikationsmöglichkeit) mehr vorhanden ist. Referenzvariablen werden als Felder von Klassen und als lokale Variablen von Methoden oder Eigenschaften benötigt.

4.3.3 Klassifikation von Variablen nach der Zuordnung

Nach der Zuordnung zu einer *Methode* oder *Eigenschaft*, zu einem *Objekt* oder zu einer *Klasse* unterscheidet man:

- **Lokale Variablen**
  Sie werden innerhalb einer Methode oder Eigenschaft deklariert. Ihre Gültigkeit beschränkt sich auf die Methode bzw. Eigenschaft, genauer: auf einen Anweisungsblock innerhalb der Methode bzw. Eigenschaft (siehe Abschnitt 4.3.8).

  Solange eine Methode bzw. Eigenschaft ausgeführt wird, befinden sich ihre Variablen in einem Speicherbereich, den man als **Stack** (dt.: *Stapel*) bezeichnet. Die Abbildung im Abschnitt 4.3.2.2 zeigt die lokalen Variablen `b1` und `b2` aus der **Main()** - Methode der Klasse `Bruchaddition`, die als Referenzvariablen auf Objekte der Klasse `Bruch` zeigen.

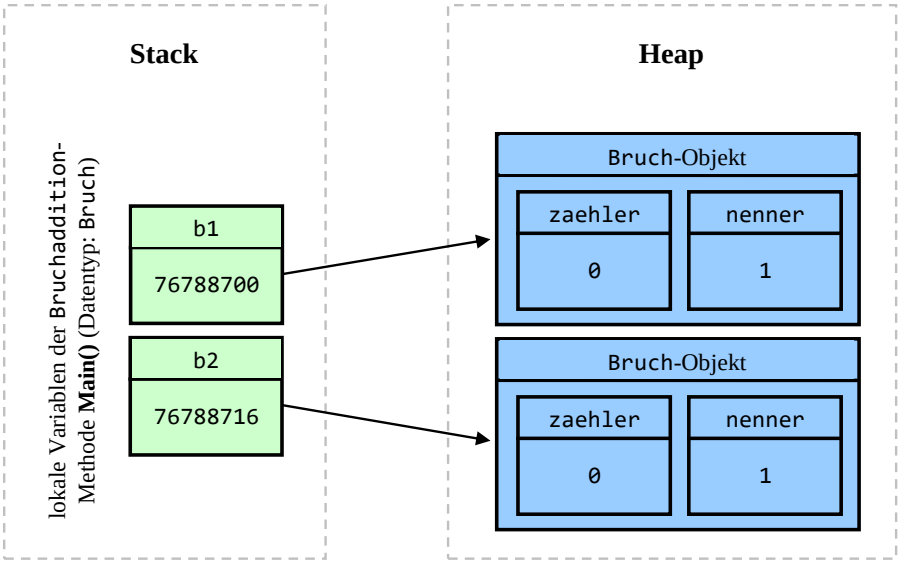
- **Instanzvariablen (nicht-statische Felder)**
  Instanzvariablen werden außerhalb jeder Methode deklariert. Jedes Objekt (man kann auch sagen: *jede Instanz*) einer Klasse verfügt über einen vollständigen Satz der Instanzvariablen dieser Klasse. So besitzt z. B. jedes Objekt der Klasse `Bruch` einen `zaehler` und einen `nenner`.

  Solange ein Objekt existiert, befinden es sich mit all seinen Instanzvariablen in einem Speicherbereich, den man als **Heap** (dt.: *Haufen*) bezeichnet.

- **Klassenvariablen (statische Felder)**
  Klassenvariablen werden außerhalb jeder Methode deklariert und erhalten dabei den Modifikator **static**. Diese Variablen beziehen sich auf eine Klasse insgesamt, nicht auf einzelne Instanzen der Klasse. Z. B. kann man in einer Klassenvariablen festhalten, wie viele Objekte der Klasse bei einem Programmeinsatz bereits erzeugt worden sind. In unserem Bruchrechnungs-Beispielprojekt haben wir der Einfachheit halber bisher auf statische Felder verzichtet.

  Während jedes Objekt einer Klasse über einen eigenen Satz mit allen Instanzvariablen verfügt, die beim Erzeugen des Objekts auf dem Heap angelegt werden, existieren Klassenvariablen nur *einmal*. Sie werden beim Laden der Klasse zusammen mit anderen typbezogenen Informationen (z. B. Methodentabelle) auf dem Heap abgelegt.

Die im Wesentlichen schon aus dem Abschnitt 4.3.2.2 bekannte Abbildung zur Lage im Arbeitsspeicher bei Ausführung der **Main()** - Methode der Klasse `Bruchaddition` aus unserem OOP-Standardbeispiel (vgl. Abschnitt 1) wird anschließend ein wenig präzisiert. Durch Farben und Ortsangaben wird für die beteiligten lokalen Variablen bzw. Instanzvariablen die Zuordnung zu einer Methode bzw. zu einem Objekt und die damit verbundene Speicherablage verdeutlicht:



Die lokalen Referenzvariablen `b1` und `b2` der Methode **Main()** befinden sich im Stack-Bereich des Arbeitsspeichers und enthalten jeweils die Adresse eines `Bruch`-Objekts. Jedes `Bruch`-Objekt enthält die Felder (Instanzvariablen) `zaehler` und `nenner` vom Werttyp **int** und befindet sich im Heap-Bereich des Arbeitsspeichers.

Auf Instanz- und Klassenvariablen kann in allen Methoden der eigenen Klasse zugegriffen werden. Wenn (abweichend vom Prinzip der Datenkapselung) entsprechende Rechte eingeräumt werden, ist dies auch in Methoden fremder Klassen möglich.

Im Kapitel 4 werden wir ausschließlich mit *lokalen* Variablen arbeiten. Im Zusammenhang mit der systematischen Behandlung der objektorientierten Programmierung werden die Instanz- und Klassenvariablen ausführlich erläutert.

Im Unterschied zu anderen Programmiersprachen (z. B. C++) ist es in C# *nicht* möglich, sogenannte *globale* Variablen außerhalb von Klassen zu deklarieren.

4.3.4 Elementare Datentypen

Als *elementare Datentypen* sollen die in C# vordefinierten Werttypen zur Aufnahme von einzelnen Zahlen, Zeichen oder Wahrheitswerten bezeichnet werden. Speziell für Zahlen existieren diverse Datentypen, die sich hinsichtlich Speichertechnik, Wertebereich und Platzbedarf unterscheiden. Von der folgenden Tabelle sollte man sich vor allem merken, wo sie im Bedarfsfall zu finden ist. Eventuell sind Sie aber auch jetzt schon neugierig auf einige Details:

Typ	Beschreibung	Werte	Bits
sbyte	Diese Variablentypen speichern ganze Zahlen *mit* Vorzeichen, sodass auch negative Werte aufgenommen werden können. Beispiel: `int zaehler = -7`	–128 … 127	8
short		–32768 … 32767	16
int		–2147483648 ... 2147483647	32
long		–9223372036854775808 … 9223372036854775807	64
byte	Diese Variablentypen speichern ganze Zahlen ≥ 0. Beispiel: `byte alter = 31;` Weil der Typ **uint** nicht CLS-kompatibel ist (*Common Language Specification*), sollte er nach Möglichkeit durch den Typ **int** ersetzt werden.[1]	0 … 255	8
ushort		0 … 65535	16
uint		0 ... 4294967295	32
ulong		0 … 18446744073709551615	64
float	Variablen vom Typ **float** speichern Gleitkommazahlen nach der Norm IEEE-754 (32 Bit) mit einer Genauigkeit von mind. 7 signifikanten Dezimalstellen in der Mantisse. Beispiel: `float pi = 3.141593f;` **float**-Literale (siehe unten) benötigen das Suffix **f** (oder **F**).	Minimum: $-3{,}402823 \cdot 10^{38}$ Maximum: $3{,}402823 \cdot 10^{38}$ Kleinster positiver Betrag: $1{,}401298 \cdot 10^{-45}$	32 1 für das Vorz., 8 für den Expon., 23 für die Mantisse
double	Variablen vom Typ **double** speichern Gleitkommazahlen nach der Norm IEEE-754 (64 Bit) mit einer Genauigkeit von mind. 15 signifikanten Dezimalstellen in der Mantisse. Beispiel: `double ph=1.57079632679490;`	Minimum: $-1{,}79769313486232 \cdot 10^{308}$ Maximum: $1{,}79769313486232 \cdot 10^{308}$ Kleinster positiver Betrag: $4{,}94065645841247 \cdot 10^{-324}$	64 1 für das Vorz., 11 für den Expon., 52 für die Mantisse
decimal	Variablen vom Typ **decimal** speichern Gleitkommazahlen mit einer Genauigkeit von mind. 28 signifikanten Dezimalstellen in der Mantisse und eignen sich besonders für die **Finanzmathematik**, wo Rundungsfehler zu vermeiden sind. Beispiel: `decimal p = 2344.2554634m;` **decimal**-Literale (siehe unten) benötigen das Suffix **m** (oder **M**).	Minimum: $-(2^{96}-1) \approx -7{,}9 \cdot 10^{28}$ Maximum: $2^{96}-1 \approx 7{,}9 \cdot 10^{28}$ Kleinster positiver Betrag: 10^{-28}	128 1 für das Vorz., 5 für den Expon., 96 für die Mantisse, restl. Bits ungenutzt Im Exponenten sind nur die Werte 0 bis 28 erlaubt, die negativ interpret. werden.

[1] In C# ist der Typ **uint** erlaubt, aber er ist nicht in der CLS vorgeschrieben. Es kann also eine CLS-Sprache ohne einen vergleichbaren Typ geben. Man möchte aber Assemblies erstellen, die in allen Sprachen genutzt werden können, siehe z. B.:
https://docs.microsoft.com/en-us/dotnet/standard/language-independence-and-language-independent-components?redirectedfrom=MSDN

Typ	Beschreibung	Werte	Bits
char	Variablen vom Typ **char** speichern ein Unicode-Zeichen. Im Speicher landet aber nicht die Gestalt des Zeichens, sondern seine Nummer im Zeichensatz. Daher zählt **char** zu den ganzzahligen (integralen) Datentypen. Beispiel: `char zeichen = 'j';` **char** - Literale (siehe unten) sind durch *einfache* Anführungszeichen zu begrenzen.	Unicode-Zeichen Tabellen mit allen Unicode-Zeichen sind z. B. auf der folgenden Webseite des Unicode-Konsortiums zu finden: http://www.unicode.org/charts/	16
bool	Variablen vom Typ **bool** speichern Wahrheitswerte. Beispiel: `bool cond = false;`	**true**, **false**	1

Eine Variable mit einem integralen Datentyp (z. B. **int** oder **byte**) speichert eine ganze Zahl *exakt*, sofern es nicht durch eine Wertebereichsüberschreitung zu einem Überlauf und damit zu einem sinnlosen Speicherinhalt kommt (siehe Abschnitt 4.6.1).

Eine Variable zur Aufnahme einer *Gleitkommazahl* (synonym: *Gleitpunkt-* oder *Fließkommazahl*, engl.: *floating point number*) dient zur *approximativen* Darstellung einer reellen Zahl. Dabei werden drei Bestandteile separat gespeichert: Vorzeichen, Mantisse und Exponent. Diese ergeben nach folgender Formel den dargestellten Wert, wobei ***b*** für die Basis eines Zahlensystems steht (meist verwendet: 2 oder 10):

$$\textbf{Wert} = \textbf{Vorzeichen} \cdot \textbf{Mantisse} \cdot b^{\textbf{Exponent}}$$

Bei dieser von Konrad Zuse entwickelten Darstellungstechnik resultiert im Vergleich zur Festkommadarstellung bei gleichem Speicherplatzbedarf ein erheblich größerer Wertebereich.[1] Während die Mantisse für die Genauigkeit sorgt, speichert der Exponentialfaktor die Größenordnung, z. B.:

$$-0{,}0000001252612 = (-1) \cdot 1{,}252612 \cdot 10^{-7}$$
$$1252612000000000 = (1) \cdot 1{,}252612 \cdot 10^{15}$$

Durch eine Änderung des Exponenten könnte man das Dezimalkomma durch die Mantisse „gleiten“ lassen. Allerdings wird in der Regel durch eine Restriktion der Mantisse (z. B. auf das Intervall [1; 2)) für eine eindeutige Darstellung gesorgt.

Weil der verfügbare Speicher für Mantisse und Exponent begrenzt ist (siehe obige Tabelle), bilden die Gleitkommazahlen nur eine endliche (aber für die meisten praktischen Zwecke ausreichende) Teilmenge der reellen Zahlen.

Die binären Gleitkommatypen **float** und **double** sind optimiert hinsichtlich Speicherplatzbedarf, Wertebereich und Verarbeitungsgeschwindigkeit, doch ist ihre beschränkte Genauigkeit speziell bei finanzmathematischen Anwendungen oft inakzeptabel.

In einer Variablen vom dezimalen Gleitkommatyp **decimal** kann jede Dezimalzahl mit maximal 28 Stellen (Vorkommastellen + Nachkommastellen) exakt gespeichert werden. Allerdings sind der Speicherplatzbedarf und der Zeitaufwand zur Verarbeitung im Vergleich zu den binären Gleitkommatypen deutlich erhöht.

---

[1] Quelle: http://de.wikipedia.org/wiki/Konrad_Zuse

Nähere Informationen über die Darstellung von Gleitkommazahlen im Arbeitsspeicher eines Computers folgen für speziell interessierte Leser gleich im Abschnitt 4.3.5.

4.3.5 Darstellung von Gleitkommazahlen im Arbeitsspeicher

Dieser Abschnitt kann beim ersten Lesen des Manuskripts übersprungen werden. Er enthält wichtige Details zu binären Gleitkommatypen, ist also relevant für Software, die solche Typen in wesentlichem Umfang verwendet (z. B. für mathematische oder naturwissenschaftliche Aufgaben).

4.3.5.1 Binäre Gleitkommadarstellung

Bei den binären Gleitkommatypen **float** und **double** werden auch „relativ glatte“ Zahlen in der Regel nicht genau gespeichert, wie das folgende Programm zeigt:

Quellcode	Ausgabe .NET Framework 4.8	Ausgabe .NET 5.0

```
using System;
class Prog {
   static void Main() {
      double df13 = 1.3f;
      double df125 = 1.25f;
      Console.WriteLine("{0,20:f16}", df13);
      Console.WriteLine("{0,20:f16}", df125);
      float ff13 = 1.3f;
      Console.WriteLine("\n{0,20:f16}", ff13);
   }
}
```

```
  1,2999999523162800
  1,2500000000000000

  1,3000000000000000
```

```
  1,2999999523162842
  1,2500000000000000

  1,2999999523162842
```

Während die Zahl 1,25 im **float**-Format (7 signifikante Dezimalstellen) fehlerfrei gespeichert werden kann, gelingt das bei der Zahl 1,3 *nicht*. In .NET 5.0 ist das Problem offensichtlich.[1] Im .NET Framework 4.8 ist hingegen zur Demonstration ein Trick erforderlich, weil die Ungenauigkeit bei der Ausgabe weggerundet wird: Es wird ein **float**-Wert (erzwungen per Literal-Suffix **f**) in einer **double**-Variablen abgelegt und dann mit 16 Dezimalstellen ausgegeben.

Diese Ergebnisse sind durch das Speichern der Zahlen im **binären Gleitkommaformat** nach der vom ***I**nstitute of **E**lectrical and **E**lectronics **E**ngineers* (IEEE) veröffentlichten Norm **IEEE-754** zu erklären, wobei jede Zahl als Produkt aus drei getrennt zu speichernden Faktoren dargestellt wird:[2]

$$\textbf{Vorzeichen} \cdot \textbf{Mantisse} \cdot 2^{\textbf{Exponent}}$$

Im ersten Bit einer **float**- oder **double** - Variable wird das Vorzeichen gespeichert (0: positiv, 1: negativ).

Für die Ablage des Exponenten (zur Basis 2) als Ganzzahl stehen 8 (**float**) bzw. 11 (**double**) Bits zur Verfügung. Allerdings sind im Exponenten die Werte 0 und 255 (**float**) bzw. 0 und 2047 (**double**) für Spezialfälle (z. B. denormalisierte Darstellung, +/- Unendlich) reserviert. Um auch die für Zahlen mit einem Betrag kleiner Eins benötigten *negativen* Exponenten darstellen zu können, werden Exponenten mit einer Verschiebung (*Bias*) um den Wert 127 (**float**) bzw. 1023 (**double**) abgespeichert und interpretiert. Besitzt z. B. eine **float**-Zahl den Exponenten 0, landet der Wert

$$01111111_{bin} = 127$$

---

[1] Man könnte einwenden, dass bei der Darstellung des wahren Wertes `1,3000000000000000` durch die Approximation `1,2999999523162842` nur eine Stelle korrekt ist und damit die Zusicherung von sieben signifikanten Dezimalstellen in Zweifel ziehen. Allerdings ist die Zusicherung so gemeint, dass beim *Runden* auf bis zu sieben Stellen alle Ziffern stimmen, was im Beispiel der Fall ist. Bei der Ermittlung der korrekten Dezimalstellen wird übrigens auch die Vorkommastelle mitgezählt.

[2] https://de.wikipedia.org/wiki/IEEE_754

im Speicher, und bei negativen Exponenten resultieren dort Werte kleiner als 127.

Abgesehen von betragsmäßig sehr kleinen Zahlen (siehe unten) werden die **float**- und **double**-Werte **normalisiert**, d.h. auf eine Mantisse im Intervall [1; 2) gebracht, z. B.:

$$24{,}48 = 1{,}53 \cdot 2^4$$

$$0{,}2448 = 1{,}9584 \cdot 2^{-3}$$

Zur Speicherung der Mantisse werden 23 (**float**) bzw. 52 (**double**) Bits verwendet. Das *i*-te Mantissen-Bit (von links nach rechts mit 1 beginnend nummeriert) hat die Wertigkeit 2^{-i}, sodass sich der *dezimale* Mantissenwert folgendermaßen ergibt:

$$1+m \quad \text{mit} \quad m = \sum_{i=1}^{23 \text{ bzw. } 52} b_i 2^{-i}, \; b_i \in \{0,1\}$$

Der Summenindex *i* startet mit 1, weil die führende 1 (= 2^0) der normalisierten Mantisse *nicht* abgespeichert wird (*hidden bit*). Daher stehen alle Bits für die Restmantisse (die Nachkommastellen) zur Verfügung mit dem Effekt einer verbesserten Genauigkeit. Oft wird daher die Anzahl der Mantissen-Bits mit 24 (**float**) bzw. 53 (**double**) angegeben.

Eine **float**- bzw. **double**-Variable mit dem Vorzeichen *v* (0 oder 1), dem Exponenten *e* und dem dezimalen Mantissenwert (1 + *m*) speichert also bei normalisierter Darstellung den Wert:

$$\mathbf{(-1)^v \cdot (1 + m) \cdot 2^{e-127}} \quad \text{bzw.} \quad \mathbf{(-1)^v \cdot (1 + m) \cdot 2^{e-1023}}$$

In der folgenden Tabelle finden Sie einige normalisierte **float**-Werte:

Wert	**float-Darstellung (normalisiert)**		
	Vorz.	**Exponent**	**Mantisse**
0,75 = $(-1)^0 \cdot 2^{(126-127)} \cdot (1+0{,}5)$	0	01111110	10000000000000000000000
1,0 = $(-1)^0 \cdot 2^{(127-127)} \cdot (1+0{,}0)$	0	01111111	00000000000000000000000
1,25 = $(-1)^0 \cdot 2^{(127-127)} \cdot (1+0{,}25)$	0	01111111	01000000000000000000000
-2,0 = $(-1)^1 \cdot 2^{(128-127)} \cdot (1+0{,}0)$	1	10000000	00000000000000000000000
2,75 = $(-1)^0 \cdot 2^{(128-127)} \cdot (1+0{,}25+0{,}125)$	0	10000000	01100000000000000000000
-3,5 = $(-1)^1 \cdot 2^{(128-127)} \cdot (1+0{,}5+0{,}25)$	1	10000000	11000000000000000000000

Nun kommen wir endlich zur Erklärung der eingangs dargestellten Genauigkeitsunterschiede beim Speichern der Zahlen 1,25 und 1,3. Während die Restmantisse 0,25 perfekt dargestellt werden kann,

$$\begin{aligned} 0{,}25 &= 0 \cdot 2^{-1} + 1 \cdot 2^{-2} \\ &= 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} \end{aligned}$$

gelingt dies bei der Restmantisse 0,3 nur approximativ:

$$\begin{aligned} 0{,}3 &= 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} + \ldots \\ &= 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} + 0 \cdot \frac{1}{8} + 0 \cdot \frac{1}{16} + 1 \cdot \frac{1}{32} + \ldots \end{aligned}$$

Sehr aufmerksame Leser werden sich darüber wundern, wieso die Tabelle mit den elementaren Datentypen im Abschnitt 4.3.4 z. B.

$$1{,}40129846 \cdot 10^{-45}$$

als betragsmäßig kleinsten positiven **float**-Wert nennt, obwohl der minimale Exponent nach obigen Überlegungen -126 (= 1 - 127) beträgt, was zum (gerundeten) dezimalen Exponentialfaktor

$$1{,}175 \cdot 10^{-38}$$

führt. Dahinter steckt die *denormalisierte* (synonym: *subnormale*) Gleitkommadarstellung, die als Ergänzung zur bisher beschriebenen normalisierten Darstellung eingeführt wurde, um eine bessere Annäherung an die Zahl Null zu erreichen. Bei der denormalisierten Gleitkommadarstellung sind die Exponenten-Bits alle auf 0 gesetzt, und dem Exponentialfaktor wird der feste Wert 2^{-126} (**float**) bzw. 2^{-1022} (**double**) zugeordnet. Die Mantissen-Bits haben dieselben Wertigkeiten (2^{-i}) wie bei der normalisierten Darstellung (siehe oben). Weil es kein *hidden bit* gibt, stellen sie aber nun einen dezimalen Wert im Intervall [0, 1) dar. Eine **float**- bzw. **double**-Variable mit dem Vorzeichen *v* (0 oder 1), mit komplett auf 0 gesetzten Exponenten-Bits und dem dezimalen Mantissenwert *m* speichert also bei denormalisierter Darstellung die Zahl:

$$\mathbf{(-1)^v \cdot m \cdot 2^{-126}} \quad \text{bzw.} \quad \mathbf{(-1)^v \cdot m \cdot 2^{-1022}}$$

In der folgenden Tabelle finden Sie einige denormalisierte **float**-Werte:

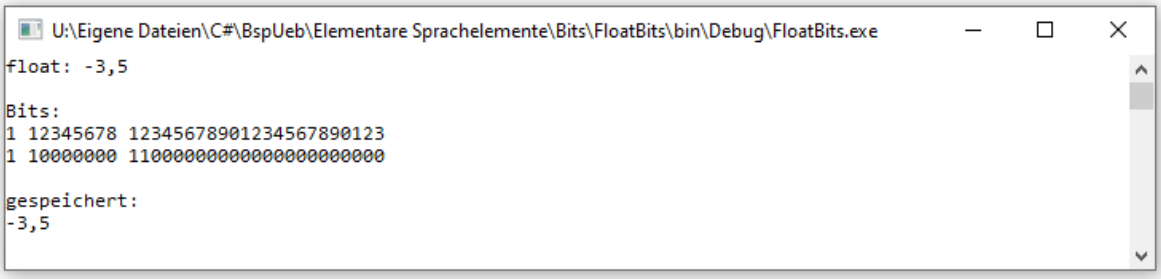
Wert	**float-Darstellung (denormalisiert)**		
	Vorz.	**Exponent**	**Mantisse**
$0{,}0 = (-1)^0 \cdot 2^{-126} \cdot 0$	0	00000000	00000000000000000000000
$-5{,}877472 \cdot 10^{-39} \approx (-1)^1 \cdot 2^{-126} \cdot 2^{-1}$	1	00000000	10000000000000000000000
$1{,}401298 \cdot 10^{-45} \approx (-1)^0 \cdot 2^{-126} \cdot 2^{-23}$	0	00000000	00000000000000000000001

Weil die Mantissen-Bits auch zur Darstellung der Größenordnung verwendet werden, schwindet die Genauigkeit mit der Annäherung an die Null.[1]

Visual Studio - Projekte zur Anzeige der Bits einer (de)normalisierten **float**- bzw. **double**-Zahl finden Sie in den Ordnern

…\BspUeb\Elementare Sprachelemente\Bits\FloatBits
…\BspUeb\Elementare Sprachelemente\Bits\DoubleBits

Diese Programme werden nicht beschrieben, weil die erforderlichen Techniken (speziell die Nachbildung von C++ - Unions in C# mit Hilfe von Attributen, siehe Kapitel 14) im Kurs ansonsten keine Rolle spielen. Eine Beispielausgabe des Programms `FloatBits`:



[1] Bei einer formatierten Ausgabe in Exponentialnotation (vgl. Abschnitt 4.2.2) liegt die Anzahl der signifikanten Dezimalstellen in der Mantisse deutlich unter 7.

4.3.5.2 Dezimale Gleitkommadarstellung

Neben den eben beschriebenen *binären* Gleitkommatypen (mit der Basis 2 für die Mantisse und den Exponenten) bietet C# auch den *dezimalen* Gleitkommatyp **decimal**, dessen Speicherorganisation die Basis 10 verwendet. Dabei werden 102 Bits folgendermaßen eingesetzt:[1]

- 1 Bit für das Vorzeichen
- 96 Bits für die Mantisse
  Hier wird eine Ganzzahl im Bereich von 0 bis 2^{96}-1 gespeichert.
- 5 Bits für den Exponenten
  Hier wird die *Anzahl* der dezimalen Nachkommastellen als Ganzzahl gespeichert, wobei nur Werte von 0 bis 28 erlaubt sind.

Eine **decimal**-Variable mit dem Vorzeichen *v* (0 oder 1), der Mantisse *m* und dem Exponenten *e* speichert den Wert:

$$\mathbf{(-1)^{v} \cdot m \cdot 10^{-e}}$$

Durch die 96-Manitissen-Bits einer **decimal**-Variablen lassen sich alle natürlichen Zahlen mit max. 28 Stellen darstellen:

$$\overbrace{9999999999999999999999999999}^{\text{28 Stellen}} < 2^{96}\text{-}1 < \overbrace{99999999999999999999999999999}^{\text{29 Stellen}}$$

Folglich kann jede Dezimalzahl mit maximal 28 Stellen (Vorkommastellen + Nachkommastellen) exakt in einer **decimal**-Variablen gespeichert werden.

Wie im Abschnitt 4.3.5.1 zu sehen war, wird die Zahl 1,3 im *binären* Gleitkommaformat durch eine prinzipiell unendliche Serie von Brüchen mit einer Zweierpotenz im Nenner dargestellt, sodass die Begrenzung des Speichers zu einer Ungenauigkeit führt:

$$1{,}3 = 1 + \frac{1}{4} + \frac{1}{32} + \frac{1}{64} + \frac{1}{512} + \frac{1}{1024} + \ldots$$

Im *dezimalen* Gleitkommaformat wird die Zahl hingegen exakt gespeichert:

$$1{,}3 = 13 \cdot 10^{-1}$$

Das folgende Programm demonstriert den Genauigkeitsvorteil des Datentyps **decimal** im finanzmathematisch relevanten Wertebereich gegenüber den binären Gleitkommatypen **float** und **double**:

Quellcode	Ausgabe .NET Framework 4.8	Ausgabe .NET 5.0

`using System;`
`class Prog {`
`  static void Main() {`
`  Console.WriteLine(10.0f - 9.9f);`
`  Console.WriteLine(10.0 - 9.9);`
`  Console.WriteLine(10.0m - 9.9m);`
`  }`
`}`
	`0,1000004`
	`0,0999999999999996`
	`0,1`
		`0,10000038`
		`0,09999999999999964`
		`0,1`

[1] Von den insgesamt belegten 128 Bits bleiben einige ungenutzt.

Allerdings hat der Typ **decimal** auch Nachteile im Vergleich zu **float** bzw. **double**:

- Kleiner Wertebereich
  Beispielweise kann die Zahl 10^{30} in einer **double**-Variablen (wenn auch nicht exakt) gespeichert werden, während sie außerhalb des **decimal**-Wertebereichs liegt.
- Hoher Speicherbedarf
  Eine **double**-Variable belegt nur halb so viel Speicherplatz wie eine **decimal**-Variable.
- Hoher Zeitaufwand bei arithmetischen Operationen
  Bei der Aufgabe,

$$1300000000 - \sum_{i=1}^{1000000000} 1{,}3$$

  zu berechnen, ergaben sich für die Datentypen **double** und **decimal** folgende Genauigkeits- und Laufzeitunterschiede:[1]

.NET Framework 4.8	.NET 5.0

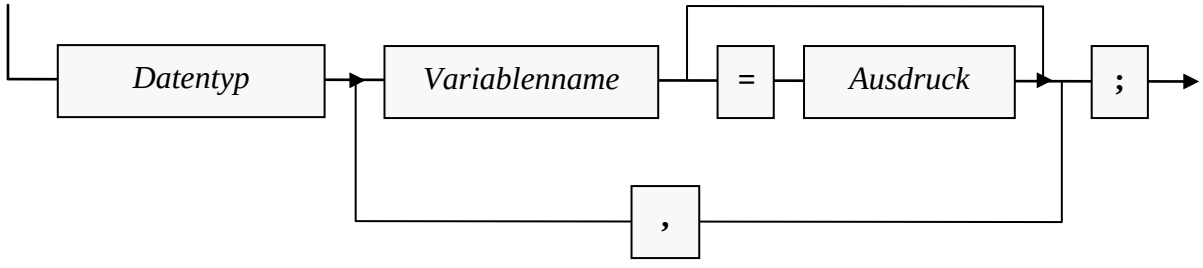
`double:`
`  Abweichung:  -24,5162951946259`
`  Benöt. Zeit:  990,8876 Millisek.`
 	
`decimal:`
`  Abweichung:  0,0`
`  Benöt. Zeit:  19463,5288 Millisek.`
	`double:`
	`  Abweichung:  -24,516295194625854`
	`  Benöt. Zeit:  3427,0437 Millisek.`
	
	`decimal:`
	`  Abweichung:  0,0`
	`  Benöt. Zeit:  25053,7655 Millisek.`

  Die gut bezahlten Verantwortlichen vieler Banken, die sich gerne als „Global Player" betätigen und dabei den vollen Sinn der beiden Worte ausschöpfen (mit Niederlassungen in Schanghai, New York, Mumbai etc. und einem Verhalten wie im Spielcasino) wären heilfroh, wenn nach einem Spiel mit 1,3 Milliarden Euro Einsatz nur 24,52 Euro in der Kasse fehlen würden. Generell sind im Finanzsektor solche Fehlbeträge aber unerwünscht, sodass man bei finanzmathematischen Aufgaben trotz des erhöhten Zeitaufwands (im Beispiel: ca. Faktor 20) den Datentyp **decimal** verwenden sollte. Für manche monetäre Berechnungen taugen auch die Ganzzahltypen **int** und **long**, die bei Addition und Subtraktion (z. B. mit Cent-Werten) wenig Zeitaufwand verursachen und eine perfekte Genauigkeit bieten, sofern ihr Wertebereich nicht verlassen wird.
- Keine Unterstützung für Sonderfälle wie +/- Unendlich und NaN (***Not a Number***) (vgl. Abschnitt 4.6)

4.3.6 Variablendeklaration, Initialisierung und Wertzuweisung

In C#-Programmen muss jede Variable vor ihrer Verwendung deklariert werden. Dabei sind auf jeden Fall ein Datentyp und ein Name anzugeben, wie das folgende Syntaxdiagramm zur **Variablendeklarationsanweisung** für *lokale* Variablen zeigt:

Deklaration einer lokalen Variablen



[1] Gemessen auf einem PC mit Intel-CPU Core i3 550 unter Windows 10 (64 Bit)

Als Datentypen kommen in Frage (vgl. Abschnitt 4.3.2):

- Werttypen, z. B.
  ```
  int i;
  ```
- Referenztypen, also Klassen (aus der BCL oder selbst definiert), z. B.
  ```
  Bruch b;
  ```

Wir betrachten vorläufig nur die innerhalb einer Methode oder Eigenschaft existierenden *lokalen* Variablen. Ihre Deklaration darf im Rumpf einer Methoden- bzw. Eigenschaftsdefinition an beliebiger Stelle *vor* der ersten Verwendung erscheinen. Um den (im Abschnitt 4.3.8 behandelten) Gültigkeitsbereich einer lokalen Variablen zur Vermeidung von Fehlern zu minimieren, sollte sie unmittelbar vor der ersten Verwendung deklariert werden (Bloch 2018, S. 261).

Es ist üblich, die Namen von lokalen Variablen mit einem Kleinbuchstaben beginnen zu lassen, also das sogenannte *Camel Casing* zu verwenden (vgl. Abschnitt 4.1.5).

Neu deklarierte Variablen kann man optional auch gleich **initialisieren**, also auf einen gewünschten Wert bringen, z. B.:

```
int i = 4711;
Bruch b = new Bruch();
```

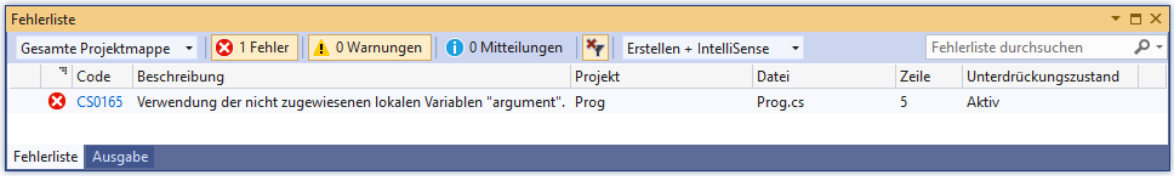
Im zweiten Beispiel wird ein `Bruch`-Objekt per **new**-Operator (siehe Abschnitt 5.4.2) erzeugt, und anschließend dessen Adresse in die Referenzvariable `b` geschrieben.

Mit der Konstruktion von gültigen *Ausdrücken*, die einen Wert von passendem Datentyp liefern müssen, werden wir uns noch ausführlich beschäftigen.

Weil *lokale* Variablen *nicht* automatisch initialisiert werden, muss man ihnen vor dem ersten lesenden Zugriff einen Wert zuweisen. Ein Verstoß gegen diese Regel wird vom C# - Compiler verhindert, wie das folgende Programm demonstriert:

```
using System;
class Prog {
  static void Main() {
    int argument;
    Console.WriteLine("Argument =  {0}", argument);
  }
}
```

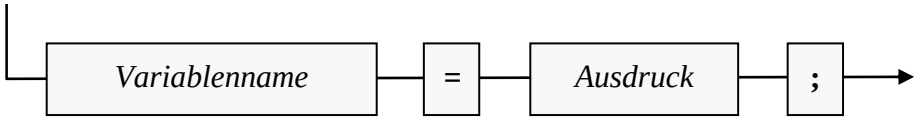
Der Compiler meint dazu:



Weil *Instanz- und Klassenvariablen* automatisch mit der typspezifischen Null initialisiert werden (siehe unten), ist in C# dafür gesorgt, dass alle Variablen beim Lesezugriff stets einen definierten Wert haben.

Um den Wert einer Variablen im weiteren Programmablauf zu verändern, verwendet man eine **Wertzuweisung**, die zu den einfachsten und am häufigsten benötigten Anweisungen gehört:

Wertzuweisungsanweisung



Beispiel:

```
int az = Math.Abs(zaehler);
```

Durch diese Wertzuweisungsanweisung aus der `Kuerze()` - Methode unserer Klasse `Bruch` (siehe Kapitel 1) erhält die **int**-Variable `az` den Wert des Ausdrucks `Math.Abs(zaehler)`.

Es wird sich bald herausstellen, dass auch ein Ausdruck stets einen Datentyp hat. Bei der Wertzuweisung muss dieser Typ kompatibel zum Datentyp der Variablen sein.

Mittlerweile haben Sie zwei Sorten von C# - Anweisungen kennengelernt:

- Deklaration von lokalen Variablen
- Wertzuweisung

4.3.7 Implizite und zielorientierte Typisierung

4.3.7.1 Implizite Typisierung

Wenn der Compiler den Datentyp erkennen kann, darf bei der Deklaration von lokalen Variablen seit C# 3.0 über das Schlüsselwort **var** die implizite Typisierung verwendet werden:

```
var i = 4711;
var b = new Bruch();
```

Dabei wird das Prinzip der strengen und statischen Typisierung keinesfalls aufgehoben. Sobald der Mauszeiger über einem Variablennamen verharrt, zeigt sich die Entwicklungsumgebung perfekt über den Datentyp informiert:

Eventuell war im Beispiel für die Variable `d1` aber der Datentyp **double** vorgesehen, und versehentlich ein **int**-Literal zum Initialisieren verwenden worden. Dann kann der falsche Datentyp zum gravierenden Problem werden, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  var d1 = 2147483647;`
`  double d2 = 2147483647;`
`  d1 = d1 + 5;`
`  d2 =  d2 + 5;`
`  Console.WriteLine($"{d1}\n{d2}");`
`  }`
`}`
	`-2147483644`
	`2147483652`

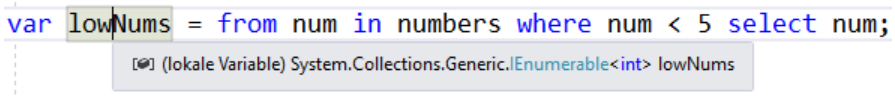
Das Addieren der Zahl 5 führt bei `d1` aufgrund der fehlerhaften impliziten Typisierung zu einem Ganzzahlüberlauf mit sinnlosem Ergebnis (siehe Abschnitt 4.6), während bei der explizit typisierten Variablen `d2` dieselbe Operation korrekt ausgeführt wird.

Bei umständlich zu schreibenden Datentypen ist **var** aber eine erhebliche Schreibvereinfachung, und der gute Vorsatz zur stets expliziten Notation von Datentypen gerät in Gefahr. Im folgenden Beispiel dient das konkretisierte generische Interface **IEnumerable<int>** aus dem Namensraum **System.Collections.Generic** als Datentyp:[1]

[1] Die Begriffe *generisch* und *Interface* sowie die im Ausdruck verwendete LINQ-Technik (*Language Integrated Query*) werden später behandelt.

```
System.Collections.Generic.IEnumerable<int> lowNums =
    from num in numbers where num < 5 select num;
```

Im Beispiel lassen sich mit dem Schlüsselwort **var** ca. 40 Zeichen einsparen, und der Compiler behält trotzdem den Überblick:



4.3.7.2 Zieltypisierte new-Ausdrücke

Obwohl wir uns mit dem **new**-Operator noch nicht systematisch beschäftigt haben (siehe Abschnitt 5.4.2), ist Ihnen die folgende Anweisung aus dem Bruchrechnungsprojekt vermutlich relativ vertraut:

```
Bruch b = new Bruch();
```

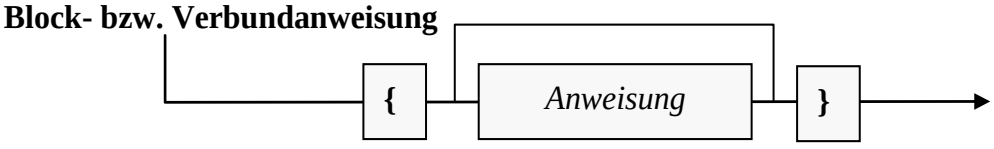
Auf den **new**-Operator folgt ein sogenannter *Konstruktor*, der zum Erstellen und Initialisieren eines neuen `Bruch`-Objekts dient und den Namen der Klasse trägt. Zusammengenommen steht in obiger Anweisung rechts vom Zuweisungszeichen ein **new**-Ausdruck. Dass dort der Konstruktor notiert und damit der Name der Klasse auch dann wiederholt werden muss, wenn in der Deklarationsanweisung der Datentyp der zu initialisierenden Variablen explizit angegeben wurde, haben wohl einige Entwickler als störende Redundanz erlebt (als Verstoß gegen das DRY-Prinzip: Don't repeat yourself). Seit C# 9.0 ist eine Schreibvereinfachung zur Vermeidung der doppelten Nennung des Klassennamens erlaubt, sodass im Beispiel resultiert:

```
Bruch b = new();
```

Die von Microsoft für die neue Option verwendete englischsprachige Bezeichnung *target-typed new expression* wurde in der Überschrift zum aktuellen Abschnitt übersetzt mit *zieltypisierte new-Ausdrücke*.[1] Gegen die im Abschnitt 4.3.7.1 beschriebene Schreibvereinfachung durch Verwendung des Schlüsselworts **var** an Stelle des Datentyps in einer Variablendeklaration wird oft eingewendet, dass die Lesbarkeit des Quellcodes leidet. Das ist bei einem zieltypisierten **new**-Ausdruck nicht zu befürchten.

4.3.8 Blöcke und Sichtbarkeitsbereiche für lokale Variablen

Wie Sie bereits wissen, besteht der Rumpf einer Methodendefinition aus einem Block mit beliebig vielen Anweisungen, der durch geschweifte Klammern begrenzt ist. Innerhalb des Methodenrumpfs können weitere Anweisungsblöcke gebildet werden, wiederum durch geschweifte Klammen begrenzt. Man spricht hier auch von einer **Block- bzw. Verbundanweisung**, und diese kann überall stehen, wo eine einzelne Anweisung erlaubt ist:[2]



Unter den Anweisungen eines Blocks dürfen sich selbstverständlich wiederum Blockanweisungen befinden. Einfacher ausgedrückt: Blöcke dürfen geschachtelt werden.

---

[1] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/target-typed-new

[2] Ein Block *ohne* enthaltene Anweisung

```
{}
```

wird vom Compiler als Anweisung akzeptiert, z. B. als Rumpf einer Methode, die keinerlei Tätigkeit entfalten soll.

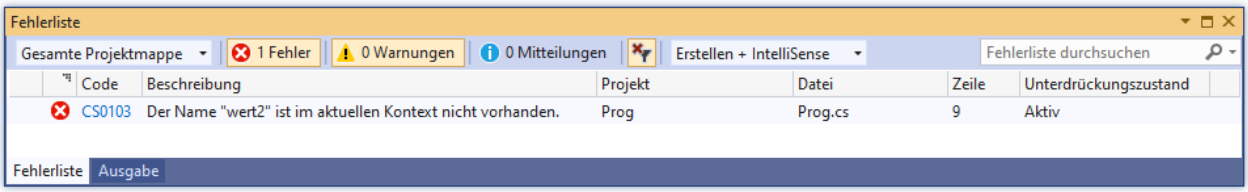
Oft treten Blöcke als Bestandteile von bedingten Anweisungen, Fallunterscheidungen und Wiederholungsanweisungen auf (siehe Abschnitt 4.7), z. B. in der Methode `Kuerze()` der Klasse `Bruch` (siehe Kapitel 1):

```
public void Kuerze() {
 if (zaehler != 0) {
  int az = Math.Abs(zaehler);
  int an = Math.Abs(nenner);
    . . .
    . . .
  zaehler /= az;
  nenner  /= az;
 } else
  nenner = 1;
}
```

Anweisungsblöcke haben einen wichtigen Effekt auf die Sichtbarkeit (alias: Gültigkeit) der darin deklarierten Variablen: Eine lokale Variable ist gültig von der deklarierenden Anweisung bis zur schließenden Klammer des Blocks, in dem sich die Deklaration befindet. Nur in diesem **Sichtbarkeitsbereich** (alias: *Gültigkeitsbereich* oder *Kontext*, engl. *scope*) kann sie über ihren Namen angesprochen werden. Beim Versuch, das folgende (weitgehend sinnfreie) Beispielprogramm

```
using System;
class Prog {
  static void Main() {
    int wert1 = 1;
    if (wert1 == 1) {
      int wert2 = 2;
      Console.WriteLine("Gesamtwert = " + (wert1 + wert2));
    }
    Console.WriteLine("Wert2 = " + wert2);
  }
}
```

zu übersetzen, erhält man vom Compiler die Fehlermeldung:



Wird die fehlerhafte Zeile auskommentiert, lässt sich das Programm übersetzen. In dem zur **if**-Anweisung gehörenden Block ist die im übergeordneten Block der **Main()** - Methode deklarierte Variable `wert1` also sichtbar, und die folgende Anweisung bietet keinen Anlass zur Kritik:

```
Console.WriteLine("Gesamtwert = " + (wert1 + wert2));
```

Bei hierarchisch geschachtelten Blöcken ist es in C# *nicht* erlaubt, auf mehreren Stufen Variablen mit identischem Namen zu deklarieren. Diese kaum sinnvolle Option ist z. B. in der Programmiersprache C++ vorhanden und erlaubt dort Fehler, die schwer aufzuspüren sind. In C# gehört ein eingeschachtelter Block zum Gültigkeitsbereich des umgebenden Blocks.

Der Sichtbarkeitsbereich einer lokalen Variablen sollte möglichst klein gehalten werden, um die Lesbarkeit und die Wartungsfreundlichkeit des Quellcodes zu verbessern. Vor allem wird auf diese Weise das Risiko von Programmierfehlern reduziert. Wird eine Variable zu früh deklariert, bestehen viele Gelegenheiten für schädliche Wertzuweisungen. Aus einer längst überwundenen Verpflichtung alter Programmiersprachen ist bei manchen Programmierern die Gewohnheit entstanden,

alle lokalen Variablen am Blockbeginn zu deklarieren. Stattdessen sollten lokale Variablen zur Minimierung ihres Sichtbarkeitsbereichs unmittelbar vor der ersten Verwendung deklariert werden (Bloch 2018, S. 261).

Zur übersichtlichen Gestaltung von C# - Programmen ist das Einrücken von Anweisungsblöcken sehr zu empfehlen, wobei Sie die Position der einleitenden Blockklammer und die Einrücktiefe nach persönlichem Geschmack wählen können, z. B.:

```
if (wert1 == 1)
{
    int wert2 = 2;
    Console.WriteLine("Wert2 = "+wert2);
}
```

```
if (wert1 == 1) {
  int wert2 = 2;
  Console.WriteLine("Wert2 = "+wert2);
}
```

Das Visual Studio unterstützt uns bei der Einhaltung einer Konvention und verwendet dabei per Voreinstellung die linke Variante. Im Abschnitt 4.1.3 wurde schon gezeigt, dass man diese Voreinstellung über

Extras > Optionen > Text-Editor > C# > Codeformat > Formatierung > Neue Zeilen

ändern kann.

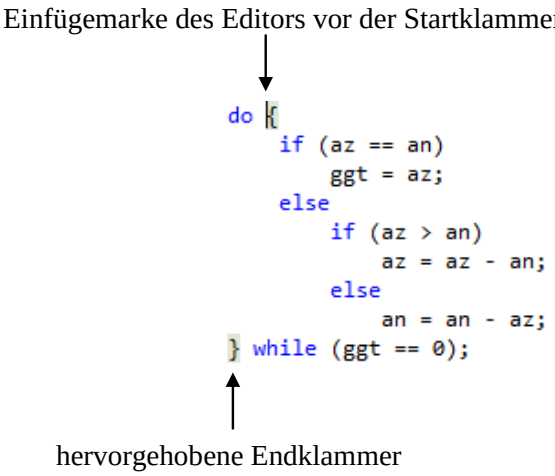
Im Quellcodeeditor der Entwicklungsumgebung kann ein markierter Block aus mehreren Zeilen mit

Tab komplett nach rechts eingerückt

und mit

Umschalt + Tab komplett nach links ausgerückt

werden. Außerdem kann man sich zu einer Blockklammer das Gegenstück anzeigen lassen:



4.3.9 Konstanten

Für einen im Programm benötigten festen Wert, der schon *zur Übersetzungszeit* feststeht und nie geändert werden soll, empfiehlt sich die Definition einer *Konstanten*, die dann im Quellcode über ihren Namen angesprochen werden kann, denn:

- Bei einer späteren Änderung des Wertes ist nur die Quellcodezeile mit der Konstantendeklaration betroffen.
- Der Quellcode ist leichter zu lesen, wenn Variablennamen an Stelle von „magischen Zahlen“ stehen.

Im folgenden Beispiel wird für eine in Sekunden gemessene Dauer die Anzahl der vergangenen Tage bestimmt (ein Tag dauert 86400 Sekunden):

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  const int secDay = 86400;`
`  int duration = 176000;`
`  Console.WriteLine($"Vergangene ganze Tage: {duration / secDay}");`
`  }`
`}`
	`2`

Im Vergleich zu einer Variablen weist eine Konstante folgende Besonderheiten auf:

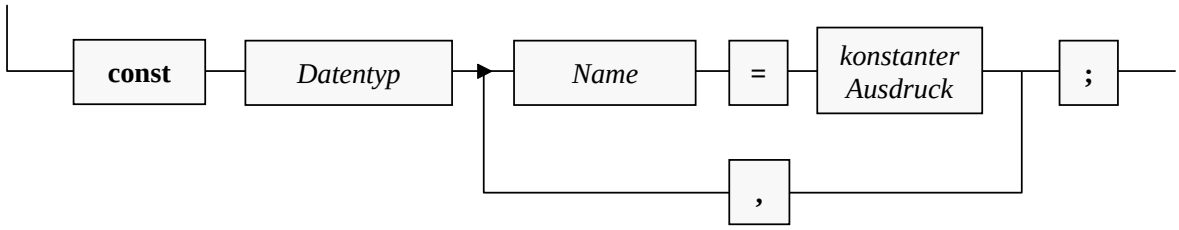
- Ihre Deklaration beginnt mit dem Modifikator **const** und *muss* eine Initialisierung enthalten, wobei ein *konstanter* Ausdruck zu verwenden ist, der nur Literale (siehe Abschnitt 4.3.10) und andere Konstanten enthält.
- Ihr Wert kann im Programm *nicht* geändert werden, sodass insbesondere irrtümliche Wertveränderungen ausgeschlossen sind.

Manche Programmierer verwenden im Namen einer Konstanten ausschließlich Großbuchstaben und verbessern bei einem Mehrwortnamen die Lesbarkeit durch trennende Unterstriche (z. B.: `SEC_DAY`). Nach der aktuell dominierenden Meinung sollte sich in C# der Modifikator **const** jedoch *nicht* auf die Benennung auswirken, sodass die im Abschnitt 4.1.5 beschriebenen Konventionen auch für Konstanten anwendbar sind.[1]

Als Datentypen sind für Variablen mit **const**-Deklaration ausschließlich die elementaren Datentypen und der Typ **String** erlaubt bzw. sinnvoll.[2]

Das Syntaxdiagramm zur Deklaration einer *konstanten* lokalen Variablen:

Deklaration von konstanten lokalen Variablen



Neben lokalen Variablen können auch Felder einer Klasse als konstant deklariert werden. Bei Feldern ist auch der Modifikator **readonly** erlaubt. Während **const** eine Wertfixierung zur Übersetzungszeit bewirkt, kann mit **readonly** eine Wertfixierung im Programmablauf nach der Initialisierung per Konstruktor vereinbart werden (siehe Abschnitt 5.2.5).

---

[1] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/constants
https://google.github.io/styleguide/csharp-style.html

[2] Begründung:
- Nicht-elementare Strukturen (siehe unten) sind als Datentypen verboten, weil deren Konstruktor zur Laufzeit ausgeführt wird, was dem Prinzip der Berechenbarkeit zur Übersetzungszeit widerstrebt.
- Referenztypen außer **String** sind zwar erlaubt, aber sinnlos, weil bei der obligatorischen Initialisierung nur das Referenzliteral **null** möglich ist.

4.3.10 Literale

Die im Programmcode auftauchenden expliziten Werte bezeichnet man als *Literale*. Wie Sie aus dem Abschnitt 4.3.9 wissen, ist es oft sinnvoll, Literale innerhalb von Konstanten-Deklarationen zu verwenden, z. B.:

```
const int secDay = 86400;
```

Aber auch andere Einsatzorte kommen in Frage (z. B. Variableninitialisierung, Aktualparameter).

Auch die Literale besitzen in C# stets einen **Datentyp**, wobei einige Regeln zu beachten sind, die gleich erläutert werden.

In diesem Abschnitt haben viele Passagen Nachschlagecharakter, sodass man beim ersten Lesen nicht jedes Detail aufnehmen muss bzw. kann.

4.3.10.1 Ganzzahlliterale

Ganzzahlliterale können im dezimalen, im binären oder im hexadezimalen Zahlensystem geschrieben werden. Bei Verwendung des seit C# 7.0 unterstützten binären Zahlensystems (mit der Basis 2 und den Ziffern 0, 1) ist das Präfix **0b** oder **0B** zu verwenden. Die Anweisungen:

```
int i = 11, j = 0b11;
Console.WriteLine($"i = {i}, j = {j}");
```

liefern die Ausgabe:

```
i = 11, j = 3
```

Für das Ganzzahlliteral `0b11` ergibt sich der dezimale Wert 3 aufgrund der Stellenwertigkeiten im Binärsystem folgendermaßen:

$$11_{bin} = 1 \cdot 2^1 + 1 \cdot 2^0 = 1 \cdot 2 + 1 \cdot 1 = 3$$

Bei Verwendung des hexadezimalen Zahlensystems (mit der Basis 16 und den Ziffern 0, 1, …, 9, A, B, C, D, E, F) ist das Präfix **0x** oder **0X** zu verwenden. Die Anweisungen:

```
int i = 11, j = 0x11;
Console.WriteLine($"i = {i}, j = {j}");
```

liefern die Ausgabe:

```
i = 11, j = 17
```

Für das Ganzzahlliteral `0x11` ergibt sich der dezimale Wert 17 aufgrund der Stellenwertigkeiten im Hexadezimalsystem folgendermaßen:

$$11_{hex} = 1 \cdot 16^1 + 1 \cdot 16^0 = 1 \cdot 16 + 1 \cdot 1 = 17$$

Eventuell fragen Sie sich, wozu man sich mit dem Binärsystem oder dem Hexadezimalsystem plagen sollte. Gelegentlich ist ein ganzzahliger Wert (z. B. als Methodenparameter) anzugeben, den man (z. B. aus einer Tabelle) nur in binärer oder hexadezimaler Darstellung kennt. In diesem Fall spart man sich durch die Verwendung dieser Darstellung die Wandlung in das Dezimalsystem.

Seit C# 7.0 dürfen dezimale Ganzzahlliterale mit dem Unterstrich als **Zifferntrennzeichen** geschrieben werden, z. B.:

```
long i = 9_000_000_000_000_000_000;
```

Seit C# 7.2 ist dies auch bei Ganzzahlliteralen im binären oder im hexadezimalen Zahlensystem erlaubt, wobei der Unterstrich auch zwischen dem Zahlensystempräfix und der Zahl stehen darf, z. B.:

```
int i127 = 0b_0111_1111;
```

Der Datentyp eines Ganzzahlliterals hängt von seinem Wert und einem eventuell vorhandenen Suffix ab:

- Ist *kein* Suffix vorhanden, hat das Ganzzahlliteral den ersten Typ aus der folgenden Serie, der seinen Wert aufnehmen kann:

  int, **uint**, **long**, **ulong**

  Beispiele:

Literale	Typ
`2147483647`	**int**
`2147483648`	**uint**
`9223372036854775807`	**long**
`9223372036854775808`	**ulong**

- Ein Ganzzahlliteral mit Suffix **u** oder **U** (*unsigned*, ohne Vorzeichen) hat den ersten Typ aus der folgenden Serie, der seinen Wert aufnehmen kann:

  uint, **ulong**

  Beispiele:

Literale	Typ
`2147483647U`	**uint**
`9223372036854775807u`	**ulong**

- Ein Ganzzahlliteral mit Suffix **l** oder **L** (*Long*) hat den ersten Typ aus der folgenden Serie, der seinen Wert aufnehmen kann:

  long, **ulong**

  Beispiele:

Literale	Typ
`2147483647L`	**long**
`9223372036854775808L`	**ulong**

  Der Kleinbuchstabe **l** ist leicht mit der Ziffer **1** zu verwechseln und daher als Suffix ungeeignet.

- Ein Ganzzahlliteral mit Suffix **ul**, **lu**, **UL**, **LU**, **uL**, **Lu**, **Ul**, oder **lU** hat den Typ **ulong**.

- Kann ein Wert von keinem Datentyp aufgenommen werden, warnt das Visual Studio, z. B.

```
static void Main() {
    Console.WriteLine(18446744073709551616);
}
```

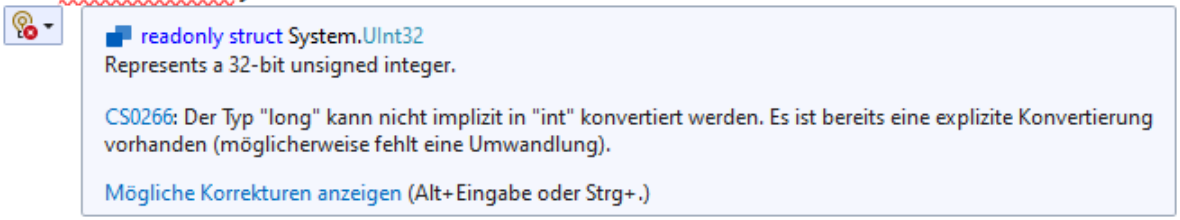
  struct System.Int32
  Stellt eine 32-Bit-Ganzzahl mit Vorzeichen dar. Um den .NET Framework-Quellcode für diesen Typ zu durchsuchen, finden Sie unter der Reference Source.

  CS1021: Die integrale Konstante ist zu groß.

  Obwohl die Fehlermeldung es nicht vermuten lässt, hat die Entwicklungsumgebung *alle* in Frage kommenden Datentypen (inkl. **ulong**) daraufhin untersucht, ob sie den Wert aufnehmen können.

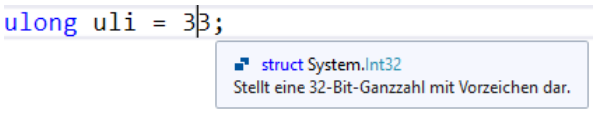
Per Ganzzahlliteral können nur nicht-negative Zahlen dargestellt werden, und im folgenden Beispiel

```
int l = -2147483649;
```



entsteht auf der rechten Seite des Zuweisungszeichens ein Ausdruck vom Typ **long** mit negativem Wert, indem die Vorzeichenumkehr-Operation auf ein Literal vom Typ **uint** angewendet wird.[1]

Weil der Compiler einige Intelligenz bei der Verwendung von Ganzzahlliteralen zeigt, sind die eben erwähnten Suffixe oft überflüssig. Man kann z. B. einer vorzeichenlosen Ganzzahlvariablen ein Literal mit geeignetem Wert zuweisen, obwohl dieses Literal formal zu einem vorzeichenbehafteten Typ gehört:



4.3.10.2 Gleitkommaliterale

Zahlen mit Dezimalpunkt oder Exponent sind in C# vom Typ **double**, wenn nicht per Suffix ein alternativer Typ erzwungen wird:

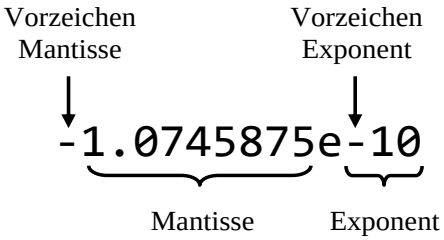
- Durch das Suffix **F** oder **f** wird der Datentyp **float** erzwungen, z. B.:
  `9.78f`
- Durch das Suffix **M** oder **m** wird der Datentyp **decimal** erzwungen, z. B.:
  `1.3m`

Mit dem Suffix **D** oder **d** wird auch bei einer Zahl *ohne* Dezimalpunkt oder Exponent der Datentyp **double** erzwungen. Warum das Suffix **d** im folgenden Beispiel für das erwartete Rechenergebnis sorgt, erfahren Sie im Zusammenhang mit dem Unterschied zwischen der Ganzzahl- und der Gleitkommaarithmetik (siehe Abschnitt 4.5.1):

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  Console.WriteLine(5/2);`
`  Console.WriteLine(5d/2);`
`  }`
`}`
	`2`
	`2,5`

Neben der alltagsüblichen Schreibweise (mit dem *Punkt* als Dezimaltrennzeichen) erlaubt C# bei Gleitkommaliteralen auch die wissenschaftliche Exponentialnotation (mit der Basis 10), z. B. bei der Zahl -0,00000000010745875):

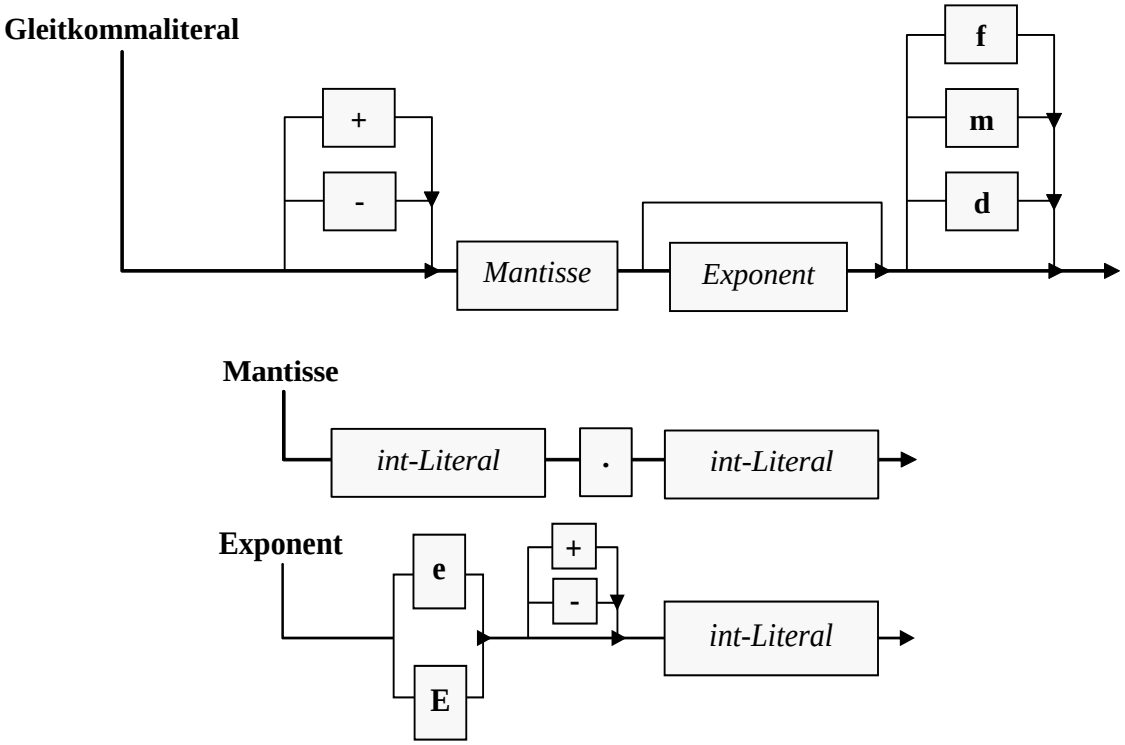


Eine Veränderung des Exponenten lässt das Dezimaltrennzeichen gleiten und macht somit die Bezeichnung *Gleitkommaliteral* (engl.: *floating-point literal*) plausibel.

In den folgenden Syntaxdiagrammen werden die wichtigsten Regeln für Gleitkommaliterale beschrieben:

---

[1] Dass hinter dem elementaren Datentyp **uint** die vordefinierte BCL-Struktur **System.UInt32** steckt, erfahren Sie im Abschnitt 6.1.2.



Es sind folgende Regeln zu beachten:

- Die in der Mantisse und im Exponenten auftretenden Ganzzahlliterale müssen das dezimale Zahlensystem verwenden und den Datentyp **int** besitzen, sodass die im Abschnitt 4.3.10.1 beschriebenen Präfixe (**0x**, **0X**) und Suffixe (z. B. **L**, **U**) verboten sind.
- Zifferntrennstriche sind erlaubt.
- Der Exponent wird zur Basis 10 verstanden.

Der Einfachheit halber unterschlagen die Syntaxdiagramme die im letzten Beispielprogramm benutzte Konstruktion eines Gleitkommaliterals über das Suffix **d**, z. B.:

```
Console.WriteLine(5d/2);
```

Der Compiler achtet bei Wertzuweisungen unter Verwendung von Gleitkommaliteralen streng auf die Typkompatibilität. Z. B. führt die folgende Deklarationsanweisung:

```
float pf = 1.3;
```

zu der Fehlermeldung:

```
Literale des Typs "Double" können nicht implizit in den Typ "float" konvertiert werden.
Verwenden Sie ein F-Suffix, um ein Literal mit diesem Typ zu erstellen.
```

Um das Problem zu lösen, muss ein Suffix an das Gleitkommaliteral angehängt werden:

```
float pf = 1.3f;
```

Wie im Abschnitt 4.3.5.1 über die binäre Gleitkommadarstellung erläutert wird, steckt in `1.3f` ein größerer Darstellungsfehler als in `1.3` bzw. in `pf`, und der Compiler verhindert eine Genauigkeitsreduzierende Zuweisung.

4.3.10.3 bool-Literale

Als Literale vom Typ **bool** sind nur die beiden reservierten Schlüsselwörter **true** und **false** erlaubt, z. B.:

```
bool cond = true;
```

Die **bool**-Literale sind mit *kleinem* Anfangsbuchstaben zu schreiben, obwohl sie in der Konsolenausgabe anders erscheinen, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  bool b = false;`
`  Console.WriteLine(b);`
`  }`
`}`
	`False`

4.3.10.4 char-Literale

char-Literale werden in C# durch *einfache* Hochkommata begrenzt. Es sind erlaubt:

- **Einfache Zeichen**
  Beispiel:
  ```
  const char a = 'a';
  ```
  Das einfache Hochkomma kann allerdings auf diese Weise ebenso wenig zum **char**-Literal werden wie der Rückwärts-Schrägstrich (\). In diesen Fällen benötigt man eine sogenannte *Escape-Sequenz*:
- **Escape-Sequenzen**
  Hier dürfen einem einleitenden Rückwärts-Schrägstrich u. a. folgen:
  - ein Steuerzeichen, z. B.:
    Neue Zeile `\n`
    Tabulator `\t`
  - das Hochkomma sowie der Rückwärts-Schrägstrich:
    `\'`
    `\\`
  - das Null-Zeichen:
    `\0`
    Das Zeichen mit der Nummer 0 im Unicode-Zeichensatz wird in der Programmiersprache C (*nicht* in C#) dazu verwendet, das Ende einer Zeichenfolge zu signalisieren.

  Beispiel:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  char rs = '\\';`
`  Console.WriteLine("Inhalt von rs: " + rs);`
`  }`
`}`
	`Inhalt von rs: \`

- **Unicode-Escape-Sequenzen**
  Eine Unicode-Escape-Sequenz enthält eine Unicode-Zeichennummer (vorzeichenlose Ganzzahl mit 16 Bits, also im Bereich von 0 bis 2^{16}-1 = 65535) in hexadezimaler, vierstelliger Schreibweise (ggf. links mit Nullen aufgefüllt) nach der Einleitung durch **\u** oder **\x**. So lassen sich Zeichen ansprechen, die per Tastatur nicht einzugeben sind.
  Beispiel:
  ```
  const char alpha = '\u03b3';
  ```
  Im Konsolenfenster werden die Unicode-Zeichen oberhalb von \u00ff in der Regel als Fragezeichen dargestellt. In einem GUI-Fenster erscheinen alle Unicode-Zeichen in voller Pracht (siehe nächsten Abschnitt).

4.3.10.5 Zeichenfolgenliterale

Zeichenfolgenliterale werden (im Unterschied zu **char**-Literalen) durch *doppelte* Hochkommata begrenzt. Hinsichtlich der erlaubten Zeichen und der Escape-Sequenzen gelten die Regeln für **char**-Literale analog, wobei das einfache und das doppelte Hochkomma ihre Rollen tauschen, z. B.:

```
string name = "Otto's Welt";
```

Zeichenfolgenliterale sind vom Datentyp **string**, und später wird sich herausstellen, dass es sich bei diesem Typ um eine Klasse aus dem Namensraum **System** handelt. Das (klein geschriebene!) Schlüsselwort **string** steht in C# als Aliasname für die Klassenbezeichnung **String** zur Verfügung. Wenn der Namensraum **System** (wie bei den meisten Quellcodedateien üblich) per **using**-Direktive importiert worden ist, kann die obige Variablendeklaration auch so geschrieben werden:

```
String name = "Otto's Welt";
```

Wegen der speziellen Unterstützung der sehr wichtigen Klasse **String** durch den Compiler ist hier ausnahmsweise die Groß-/Kleinschreibung irrelevant.

Um ein doppeltes Hochkomma in ein Zeichenfolgenliteral aufzunehmen, ist eine Escape-Sequenz erforderlich, z. B.:

```
string name = "\"Eumel\"";
```

Während ein **char**-Literal stets *genau ein* Zeichen enthält, kann ein Zeichenfolgenliteral aus beliebig vielen Zeichen bestehen oder auch leer sein, z. B.:

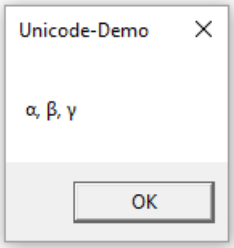
```
string name = "";
```

Das folgende Programm verwendet einen Aufruf der statischen Methode **Show()** der Klasse **MessageBox** aus dem Namensraum **System.Windows** zur Anzeige eines Zeichenfolgenliterals, das drei Unicode-Escape-Sequenzen enthält:[1]

```
class Prog {
   static void Main() {
      System.Windows.MessageBox.Show("\u03b1, \u03b2, \u03b3", "Unicode-Demo");
   }
}
```

Beim Programmstart erscheint das folgende Meldungsfenster:

[1] Für eine erfolgreiche Übersetzung des Programms ist ein Verweis auf das Assembly **PresentationFramework.dll** erforderlich. Wie man im Visual Studio Verweise setzt, wird im Abschnitt 3.3.6.1 beschrieben.



Um die Bedeutung des Rückwärts-Schrägstrichs als Escape-Zeichen abzuschalten, stellt man einer Zeichenfolge das @-Zeichen voran, z. B.:

```
Console.WriteLine(@"Pfad: C:\Program Files\Map\bin");
```

Weil nun das Escape-Zeichen fehlt, sind in manchen Fällen Ersatzlösungen erforderlich:

- Um ein doppeltes Hochkomma in die Zeichenfolge einzufügen, muss man es zweimal schreiben, z. B.:

```
string s = @"Auftretenshäufigkeit für das Wort ""Resonanz"":";
```

- Die Escape-Sequenz **\n** ist durch einen Zeilenumbruch im Quellcode zu ersetzen, was sich allerdings nachteilig auf die Formatierung des Quellcodes auswirkt, z. B.:

Quellcode	Ausgabe

`class Prog {`
`  static void Main() {`
`  string s = @"Literal`
`mit Zeilenwechsel";`
`  System.Console.WriteLine(s);`
`  }`
`}`
	`Literal`
	`mit Zeilenwechsel`

Das @-Zeichen darf mit dem im Abschnitt 4.2.2.2 beschriebenen Interpolationspräfix ($) kombiniert werden, z. B.:

```
string s = "Map";
Console.WriteLine($@"Pfad: C:\Program Files\{s}\bin");
```

Bis C# 7.3 muss die Reihenfolge **$@** eingehalten werden; ab C# 8.0 ist die Reihenfolge der beiden Präfixzeichen beliebig.

4.3.10.6 Referenzliteral null

Einer Referenzvariablen kann das Referenzliteral **null** zugewiesen werden, z. B.:

```
Bruch b1 = null;
```

Damit ist sie nicht undefiniert, sondern zeigt explizit auf nichts.

Zeigt eine Referenzvariable aktuell auf ein existentes Objekt, kann man diese Referenz per **null**-Zuweisung aufheben. Sofern im Programm keine andere Referenz auf dasselbe Objekt vorliegt, ist es zum Abräumen durch den Garbage Collector freigegeben.

Da C# eine streng typisierte Programmiersprache ist, und das Literal **null** einen Ausdruck darstellt (vgl. Abschnitt 4.5), muss es einen Datentyp besitzen. Es ist der **Nulltyp** (engl.: *null type*). Weil es in C# keinen Bezeichner für den Nulltyp gibt, kann man keine Variable von diesem Typ deklarieren. Außer dem **null**-Literal gibt es keinen weiteren Ausdruck mit Nulltyp, sodass man ihn im Programmieralltag getrost vergessen kann.

4.3.11 Übungsaufgaben zum Abschnitt 4.3

1) Wieso klagt der Compiler über ein unbekanntes Symbol, obwohl die Variable `i` deklariert worden ist?

Quellcode	Fehlermeldung

`class Prog {`
`  static void Main() {`
`  {`
`  int i = 2;`
`  }`
`  System.Console.WriteLine(i);`
`  }`
`}`
	`Prog.cs(6,28): error CS0103: Der Name "i" ist im aktuellen Kontext nicht vorhanden.`

2) Beseitigen Sie bitte alle Fehler im folgenden Programm:

```
class Prog {
  static void Main() {
    float pi = 3,141593;
    double radius = 2,0;
    System.Console.WriteLine('Der Flächeninhalt beträgt: {0:f3}',
                             pi * radius * radius);
  }
}
```

3) Schreiben Sie bitte ein Programm, das die folgende Ausgabe produziert:

```
Dies ist ein Zeichenfolgenliteral:
        "Hallo"
```

4) Im folgenden Programm erhält eine **char**-Variable das Zeichen 'c' als Wert. Anschließend wird dieser Inhalt auf eine **int**-Variable übertragen, und bei der Konsolenausgabe erscheint schließlich die Zahl 99:

Quellcode	Ausgabe

`class Prog {`
`  static void Main() {`
`  char zeichen = 'c';`
`  int nummer = zeichen;`
`  System.Console.WriteLine("zeichen = " + zeichen +`
`  "\nnummer = " + nummer);`
`  }`
`}`
	`zeichen = c`
	`nummer = 99`

Warum hat der ansonsten sehr pingelige C# - Compiler nichts dagegen, einer **int**-Variablen den Wert einer **char**-Variablen zu übergeben? Wie kann man das Zeichen 'c' über eine Unicode-Escape-Sequenz ansprechen?

5) Der im Abschnitt 3.3.4 erstellte Währungskonverter arbeitet mit dem Datentyp **double**, während für monetäre Anwendungen nachdrücklich der Typ **decimal** vorgeschlagen wird (siehe Abschnitte 4.3.4 und 4.3.5.2). Sorgen Sie daher durch einen Wechsel auf den Datentyp **decimal** für mehr Genauigkeit.

Die praktische Bedeutung dieser Maßnahme ist übrigens im Beispiel begrenzt. Man muss schon mit Billionen hantieren, damit sich nach der Division der Eingabe durch 1,95583 eine Abweichung im Cent-Bereich zeigt. Wie im Abschnitt 4.3.5.2 an einem anderen Beispiel zu sehen ist, wächst der Fehler mit der Anzahl der ausgeführten Operationen.

Außerdem sollte das Programm das Ergebnis auf volle Cent-Beträge runden. Das gelingt durch die Verwendung der statischen Methode **Format()** aus der Klasse **String**. Diese Methode arbeitet mit derselben Formatierungszeichenfolge und mit derselben Parameterliste wie die **Console**-Methode **WriteLine()** (siehe Abschnitt 4.2.2.1). In der folgenden Anweisung liefert **Format()**

```
ausgabe.Content = String.Format("{0:f2}", betrag / 1.95583m);
```

das Rechenergebnis `betrag / 1.95583m` mit 2 Nachkommastellen als Zeichenfolge.

4.4 Einfache Techniken für Benutzereingaben

Für unsere Übungsprogramme brauchen wir gelegentlich eine einfache Möglichkeit, Benutzereingaben entgegenzunehmen.

4.4.1 Via Konsole

In der `Frage()` - Methode der Klasse `Bruch` aus dem Einleitungsbeispiel (siehe Abschnitt 1) wird die folgende Anweisung genutzt, um einen **int**-Wert via Konsole zu erfragen:

```
zaehler = Convert.ToInt32(Console.ReadLine());
```

Von der statischen Methode **ReadLine()** der Klasse **Console** wird eine vom Benutzer per **Enter**-Taste abgeschickte Zeile von der Konsole gelesen. Diese Zeichenfolge dient anschließend als Argument für die statische Methode **ToInt32()** aus der Klasse **Convert**, die wie **Console** zum Namensraum **System** gehört. Sofern sich die übergebene Zeichenfolge als ganze Zahl im **int**-Wertebereich interpretieren lässt, liefert der **ToInt32()** - Aufruf diese Zahl zurück, und sie landet schließlich im **int**-Feld `zaehler`.

Hier liegt eine Verschachtelung zweier Methodenaufrufe vor, die bei Programmierern der kompakten Schreibweise wegen beliebt ist. Die folgende Variante ist für Einsteiger leichter zu verstehen, verursacht aber mehr Schreibarbeit:

```
string eingabe;
eingabe = Console.ReadLine();
zaehler = Convert.ToInt32(eingabe);
```

Die gerade vorgestellte Datenerfassungstechnik hat ein Problem mit weniger kooperativen Benutzern: Wird eine nicht konvertierbare Zeichenfolge abgeschickt, endet ein betroffenes Programm mit einem unbehandelten Ausnahmefehler, z. B.:

Quellcode	Ausgabe (Eingaben **fett**)

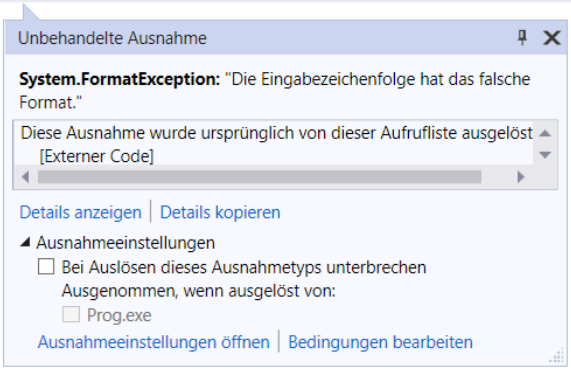
`using System;`
`class Prog {`
`  static void Main() {`
`  Console.Write("Ihre Lieblingszahl? ");`
`  int zahl = Convert.ToInt32(Console.ReadLine());`
`  Console.WriteLine("Verstanden: " + zahl);`
`  }`
`}`
	`Ihre Lieblingszahl?` **`drei`**
	
	`Unbehandelte Ausnahme:`
	`System.FormatException: Die Eingabezeichenfolge hat das falsche Format.`

Startet man das Programm aus dem Visual Studio im sogenannten *Debug-Modus*, indem man ...

- die Taste **F5** betätigt
- oder auf den Schalter ▶ klickt,

dann führt der Ausnahmefehler zur folgenden Anzeige:

```
using System;
class Prog {
    static void Main() {
        Console.Write("Ihre Lieblingszahl? ");
        int zahl = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Verstanden: " + zahl);
    }
}
```



Um in dieser Lage das Debuggen zu beenden, kann eine von den folgenden Bedienmöglichkeiten verwendet werden:

- Tastenkombination **Umschalt+F5**
- Symbolschalter ■
- Menübefehl **Debuggen > Debugging beenden**

Um einen Programmabsturz durch fehlerhafte Eingaben zu verhindern, sind Programmiertechniken erforderlich, mit denen wir uns momentan noch nicht beschäftigen wollen, z. B.:

Quellcode	Ausgabe (Eingaben **fett**)

`using System;`
`class Prog {`
`  static void Main() {`
`  int zahl;`
`  bool ok;`
`  do {`
`  Console.Write("Ihre Lieblingszahl? ");`
`  try {`
`  zahl = Convert.ToInt32(Console.ReadLine());`
`  Console.WriteLine("Verstanden: " + zahl);`
`  ok = true;`
`  } catch {`
`  ok = false;`
`  Console.WriteLine("Falsche Eingabe!");`
`  }`
`  } while (!ok);`
`  }`
`}`
	`Ihre Lieblingszahl?` **`drei`**
	`Falsche Eingabe!`
	`Ihre Lieblingszahl?` **`3`**
	`Verstanden: 3`

Hier wird ein Ausnahmefehler per **catch**-Klausel abgefangen, damit er nicht zur Beendigung des Programms führt. In einer **do**-Schleife (siehe Abschnitt 4.7.3.3.2) wird die Frage nach der Lieblingszahl so lange wiederholt, bis der Benutzer eine gültige Antwort geliefert hat.

Auch die folgende Lösung mit Hilfe der statischen Methode **TryParse()** des Typs **Int32** benötigt ebenfalls unbekannte Programmiertechniken und etliche Zeilen:

```
using System;
class Prog {
    static void Main() {
        bool ok = false;
        do {
            Console.Write("Ihre Lieblingszahl? ");
            if (ok = Int32.TryParse(Console.ReadLine(), out int zahl))
                Console.WriteLine("Verstanden: " + zahl);
            else
                Console.WriteLine("Falsche Eingabe!");
        } while (!ok);
    }
}
```

In den Übungs- bzw. Demoprogrammen verwenden wir der Einfachheit halber ungesicherte **ToInt32** - Aufrufe bzw. analoge Varianten für andere Datentypen.

4.4.2 Via InputBox

Wer eine simple Anwendung mit grafischer Bedienoberfläche erstellen möchte, ohne die Komplexität einer WPF-Anwendung (vgl. Abschnitt 3.3.4) in Kauf nehmen zu müssen, der kann ...

- die statische Methode **Show()** der Klasse **MessageBox** benutzen, um Informationen anzuzeigen,
- die statische Methode **InputBox()** der Klasse **Interaction** aus dem Namensraum **Microsoft.VisualBasic** benutzen, um Benutzereingaben per Dialogbox entgegenzunehmen.

Die Klasse **Interaction** ist als Migrationshilfe für Visual Basic 6 - Programmierer gedacht, kann aber auch in C# - Programmen genutzt werden. In der folgenden GUI-Alternative zum Beispielprogramm im Abschnitt 4.4.1 wird außerdem die Klasse **MessageBox** aus dem Namensraum **System.Windows** verwendet:

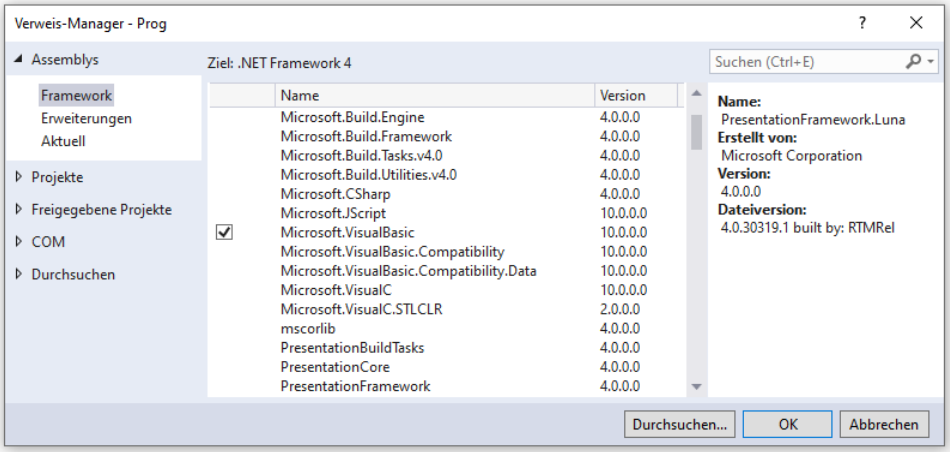
```
using System;
using System.Windows;
using Microsoft.VisualBasic;

class InputBox {
    static void Main() {
        string eingabe = Interaction.InputBox("Ihre Lieblingszahl?", "InputBox", "", -1, -1);
        int zahl = Convert.ToInt32(eingabe);
        MessageBox.Show("Verstanden: " + zahl);
    }
}
```

Ein- und Ausgabe werden per Dialogbox erledigt:

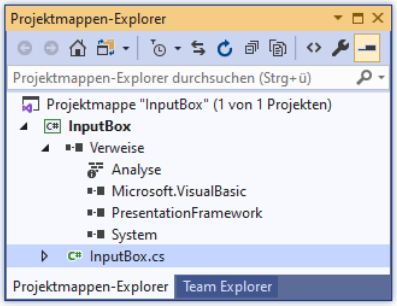
Damit das Projekt unter .NET 5.0 erstellt werden kann, muss lediglich das Element **UseWPF** in die Projektdefinitionsdatei eingefügt werden:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net5.0-windows</TargetFramework>
    <UseWPF>true</UseWPF>
  </PropertyGroup>

</Project>
```

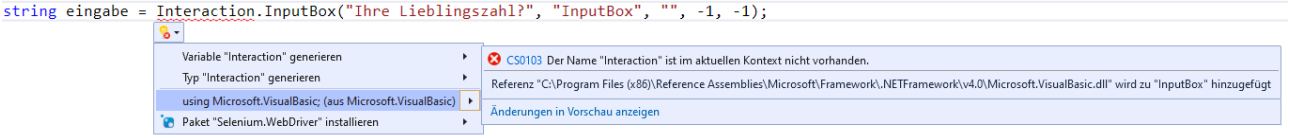
Bei einer Anwendung für das .NET Framework muss für die Klassen **Interaction** und **MessageBox** jeweils eine Referenz auf das implementierende Assembly (**microsoft.visualbasic.dll** bzw. **PresentationFramework.dll**) in die Projektdefinition aufgenommen werden. Wie Sie aus dem Abschnitt 3.3.6 wissen, kann man unserer Entwicklungsumgebung bequem erklären, welche Assembly-Referenzen beim Übersetzen eines Projekts erforderlich sind. Man wählt im Projektmappen-Explorer aus dem Kontextmenü zum Projekt-Knoten **Verweise** die Option **Verweis hinzufügen**, um dann im folgenden Dialog das gesuchte Assembly zu lokalisieren und im markierten Zustand per **OK** in die Verweisliste des Projekts aufzunehmen, z. B.:



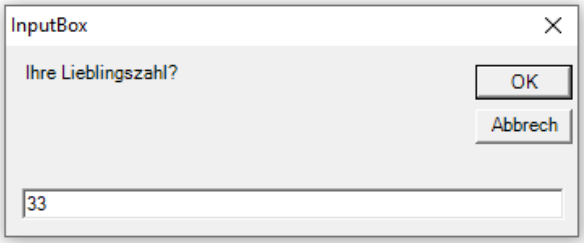
Anschließend werden die ergänzten Referenzen im Projektmappen-Explorer angezeigt, z. B.:



Über die Quick Actions der Entwicklungsumgebung klappt die Aufnahme einer Assembly-Referenz sogar noch einfacher:



Wird das Programm im .NET Framework erstellt, zeigt sich ein Schönheitsfehler:

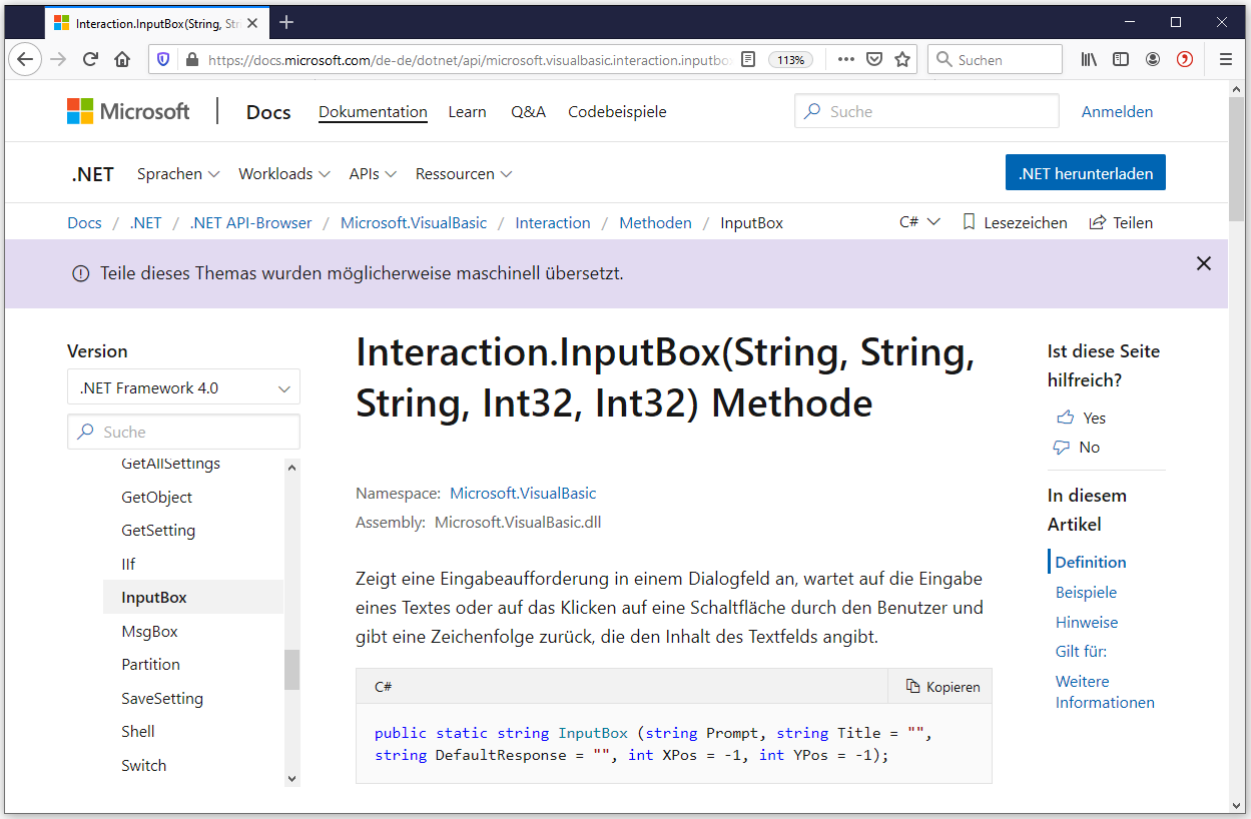


Wenn wir einen eigenen Dialog entwerfen, wird es uns natürlich *nicht* passieren, ein Bedienelement (z. B. einen Schalter) so klein zu dimensionieren, dass die Beschriftung nach der Übersetzung in eine alternative Sprache nicht mehr hineinpasst. Im InputBox-Fenster hat **Cancel** offenbar gepasst, **Abbrechen** aber nicht.

Über die Parameter (Argumente) und den Rückgabewert des **InputBox()** - Methodenaufrufs kann man sich z. B. über die Hilfefunktion der Entwicklungsumgebung informieren. Bei markiertem Methodennamen

```
string eingabe = Interaction.InputBox("Ihre Lieblingszahl?", "InputBox", "", -1, -1);
```

ruft ein Druck auf die Taste **F1** die folgende Webseite auf:



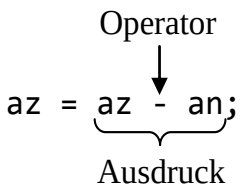
Zwar sieht die Ein- bzw. Ausgabe per GUI attraktiver aus, jedoch lohnt sich der erhöhte Aufwand bei Demo- bzw. Übungsprogrammen zu elementaren Sprachelementen kaum, sodass wir in der Regel darauf verzichten werden.

Die **Convert**-Methode **ToInt32()** reagiert natürlich auch im optisch aufgewerteten Programm auf ungeschickte Benutzereingaben mit einer Ausnahme (siehe Abschnitt 4.4.1). Später werden wir das Problem per Ausnahmebehandlung lösen.

4.5 Operatoren und Ausdrücke

Im Zusammenhang mit der Variablendeklaration und der Wertzuweisung haben wir das Sprachelement *Ausdruck* ohne Erklärung benutzt, und diese soll nun nachgeliefert werden. Im aktuellen Abschnitt 4.5 werden wir Ausdrücke als wichtige Bestandteile von C# - Anweisungen detailliert betrachten. Dabei lernen Sie elementare Datenverarbeitungs-Möglichkeiten kennen, die von sogenannten *Operatoren* mit ihren Argumenten realisiert werden, z. B. von den arithmetischen Operatoren (**+**, **-**, *****, **/**) für die Grundrechenarten. Im Verlauf des aktuellen Abschnitts werden Ihre Kenntnisse über die Datenverarbeitung mit C# erheblich wachsen. Der dabei zu investierende Aufwand lohnt sich, weil ein sicherer Umgang mit Operatoren und Ausdrücken eine unabdingbare Voraussetzung für das erfolgreiche Implementieren von Methoden und Eigenschaften ist. Dort werden die Handlungskompetenzen von Klassen bzw. Objekten realisiert.

Während die Variablen zur *Speicherung* von Werten dienen, geht es bei den **Operatoren** darum, aus vorhandenen Variableninhalten und/oder anderen Argumenten neue Werte zu berechnen. Den zur Berechnung eines Werts geeigneten, aus Operatoren und zugehörigen Argumenten aufgebauten Teil einer Anweisung bezeichnet man als **Ausdruck**, z. B. in der folgenden Wertzuweisung:[1]



Durch diese Anweisung aus der `Kuerze()` - Methode unserer Klasse `Bruch` (siehe Abschnitt 1) wird der lokalen **int**-Variablen `az` der Wert des Ausdrucks `az - an` zugewiesen. Wie in diesem Beispiel landen die Werte von Ausdrücken oft in Variablen, wobei Ausdruck und Variable typkompatibel sein müssen. Den Datentyp eines Ausdrucks bestimmen im Wesentlichen die Datentypen der Argumente, manchmal beeinflusst aber auch der Operator den Typ des Ausdrucks (z.B. bei einem Vergleichsoperator).

Schon bei einem Literal, einer Variablen oder einem Methodenaufruf haben wir es mit einem Ausdruck zu tun.[2]

Beispiel: `1.5`
Dieses Gleitkommaliteral ist ein Ausdruck mit dem Typ **double** und dem Wert 1,5 (vgl. Abschnitt 4.3.10.2).

Mit Hilfe diverser Operatoren entstehen komplexere Ausdrücke, deren Typen und Werte von den Argumenten und den Operatoren abhängen.

Beispiele:

- `2 * 1.5`
  Hier resultiert der **double**-Wert 3,0.
- `2 * 3`
  Hier resultiert der **int**-Wert 6.
- `2 > 1.5`
  Hier resultiert der **bool**-Wert **true**.

In der Regel beschränken sich die Operatoren darauf, aus ihren Argumenten (Operanden) einen Wert zu ermitteln und diesen für die weitere Verarbeitung zur Verfügung zu stellen. Einige Operatoren haben jedoch zusätzlich einen **Nebeneffekt** auf eine als Argument fungierende *Variable*, z. B.:

---

[1] Im Abschnitt 4.5.8 werden Sie eine Möglichkeit kennenlernen, diese Anweisung etwas kompakter zu formulieren.

[2] Besteht ein Ausdruck aus einem Methodenaufruf mit dem Pseudorückgabetyp **void**, dann liegt allerdings *kein* Wert vor.

```
int i = 12;
int j = i++;
```

In der zweiten Anweisung des Beispiels tritt der **Postinkrementoperator** `++` mit der **int**-Variablen `i` als Argument auf (siehe Abschnitt 4.5.1). Der Ausdruck `i++` hat den Typ **int** und den Wert 12, welcher in der Zielvariable `j` landet. Außerdem wird die Argumentvariable `i` beim Auswerten des Ausdrucks durch den Postinkrementoperator auf den neuen Wert 13 gebracht.

Die meisten Operatoren verarbeiten *zwei* Operanden (Argumente) und heißen daher **zweistellig** bzw. **binär**. Im folgenden Beispiel ist der **Additionsoperator** zu sehen, der zwei numerische Argumente erwartet:

```
a + b
```

Manche Operatoren verarbeiten nur *ein* Argument und heißen daher **einstellig** bzw. **unär**. Als Beispiel haben wir eben schon den Postinkrementoperator kennengelernt. Ein weiteres ist der **Negationsoperator**, der durch ein Ausrufezeichen dargestellt wird, *ein* Argument vom Typ **bool** erwartet und dessen Wahrheitswert umdreht (**true** und **false** vertauscht):

```
!cond
```

Wir werden auch noch einen *dreistelligen* (*ternären*) Operator kennenlernen.

Weil Ausdrücke von passendem Ergebnistyp als Argumente einer Operation erlaubt sind, können beliebig komplexe Ausdrücke aufgebaut werden. Unübersichtliche Exemplare sollten jedoch als potentielle Fehlerquellen vermieden werden.

4.5.1 Arithmetische Operatoren

Weil die arithmetischen Operatoren für die Grundrechenarten zuständig sind, müssen ihre Operanden (Argumente) einen numerischen Typ haben (**sbyte**, **short**, **int**, **long**, **byte**, **ushort**, **uint**, **ulong**, **char**, **float, double** oder **decimal**).

Die resultierenden **arithmetischen Ausdrücke** übernehmen ihren Ergebnistyp von den beiden Argumenten. Am Beispiel des Additionsoperators soll eine genauere Beschreibung geliefert werden. C# kennt sieben vordefinierte Additionsoperator-Varianten (siehe C# - Sprachspezifikation in ECMA 2017, Abschnitt 12.9.5). In der folgenden Auflistung tritt eine ungewohnte Funktions- bzw. Methodennotation für die Additionsoperator-Varianten auf, doch sind die Datentypen von Argumenten und Funktionswerten gut zu erkennen:

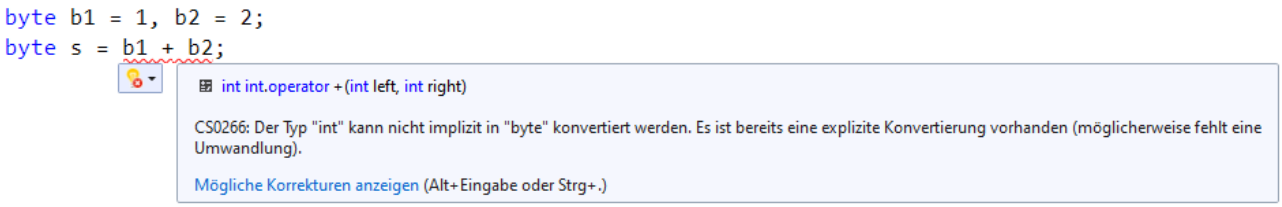
- Ganzzahladdition
  - **int operator +(int x, int y)**
  - **uint operator +(uint x, uint y)**
  - **long operator +(long x, long y)**
  - **ulong operator +(ulong x, ulong y)**
- Binäre Fließkommaaddition
  - **float operator +(float x, float y)**
  - **double operator +(double x, double y)**
- Dezimale Fließkommaaddition
  - **decimal operator +(decimal x, decimal y)**

Wie man sieht, ...

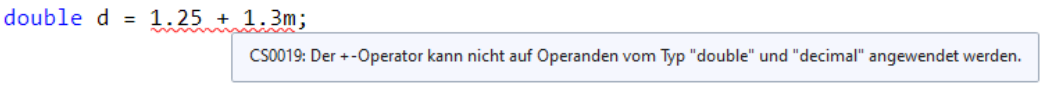
- gibt es nicht für jeden numerischen Datentyp eine spezielle Additionsoperator-Variante,
- müssen die beiden Argumente stets denselben Datentyp haben.

Wenn bei einem Argument oder bei beiden Argumenten der Datentyp nicht passt, findet nach Möglichkeit eine automatische (implizite) Typanpassung „nach oben“ statt (vgl. Abschnitt 4.5.7). Bevor

z. B. ein **int**-Argument zu einem **double**-Wert addiert werden kann, muss es in den Typ **double** konvertiert werden. Sind z. B. beide Argumente einer Additionsoperation vom Typ **byte**, werden sie vor der Addition in den Typ **int** gewandelt, den auch die Summe erhält. Der Compiler lehnt es ab, diesen **int**-Wert in eine **byte**-Variable zu schreiben:



Ist keine automatische Typanpassung möglich, beschwert sich der Compiler, z. B.:



Es hängt von den Datentypen der Argumente ab, ob die **Ganzzahl**-, oder die **Gleitkommaarithmetik** zum Einsatz kommt. Besonders auffällig sind die Unterschiede im Verhalten des Divisionsoperators, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  int i = 2, j = 3;`
`  double a = 2.0, b = 3.0;`
`  Console.WriteLine(i / j);`
`  Console.WriteLine(a / b);`
`  }`
`}`
	`0`
	`0,666666666666667`

Bei der Ganzzahldivision werden die Stellen nach dem Dezimaltrennzeichen abgeschnitten, was gelegentlich durchaus erwünscht ist. Im Zusammenhang mit dem Verhalten beim Überschreiten des Wertebereichs (dem sogenannten *Überlauf*, siehe Abschnitt 4.6) werden Sie noch weitere Unterschiede zwischen der Ganzzahl- und der Gleitkommaarithmetik kennenlernen.

In der nächsten Tabelle sind alle arithmetischen Operatoren beschrieben, wobei die Platzhalter *Num*, *Num1* und *Num2* für Ausdrücke mit einem numerischen Typ stehen, und *Var* eine numerische *Variable* vertritt:

Operator	**Bedeutung**	**Beispiel**	
		Quellcode-Fragment	**Ausgabe**
-Var	**Vorzeichenumkehr**	`int i = 2 , j = -3;` `Console.WriteLine("{0} {1}",-i,-j);`	`-2 3`
Num1 + Num2	**Addition**	`Console.WriteLine(2 + 3);`	`5`
Num1 – Num2	**Subtraktion**	`Console.WriteLine(2.6 - 1.1);`	`1,5`
*Num1 * Num2*	**Multiplikation**	`Console.WriteLine(4 * 5);`	`20`
Num1 / Num2	**Division**	`Console.WriteLine(8.0 / 5);` `Console.WriteLine(8 / 5);`	`1,6` `1`
Num1 % Num2	**Modulo (Divisionsrest)** Sei *GAD* der ganzzahlige Anteil aus dem Ergebnis der Division (*Num1 / Num2*). Dann ist *Num1 % Num2* def. durch *Num1 - GAD · Num2*	`Console.WriteLine(19 % 5);` `Console.WriteLine(-19 % 5.25);`	`4` `-3,25`
++Var	**Präinkrement bzw.**	`int i = 4;` `double a = 1.2;`	

Operator	Bedeutung	Beispiel	
		Quellcode-Fragment	Ausgabe
−−Var	**-dekrement** Als Argument ist nur eine Variable erlaubt. *++Var* erhöht *Var* um 1 *und* liefert *Var* + 1 *−−Var* reduz. *Var* um 1 *und* liefert *Var* − 1	`Console.WriteLine(++i + "\n" + --a);`	`5` `0,2`
Var++ *Var−−*	**Postinkrement bzw. -dekrement** Als Argument ist nur eine Variable erlaubt. *Var++* liefert *Var* erhöht *Var* um 1 *Var−−* liefert *Var* reduziert *Var* um 1	`int i = 4;` `Console.WriteLine(i++ + "\n" + i);`	`4` `5`

Bei den Inkrement- und den Dekrementoperatoren ist zu beachten, dass sie *zwei* Effekte haben:

- Das Argument wird ausgelesen, um den Wert des Ausdrucks zu ermitteln.
- Die als Argument fungierende numerische Variable wird verändert (vor oder nach dem Auslesen). Wegen dieses **Nebeneffekts** sind Prä- und Postinkrement- bzw. -dekrementausdrücke im Unterschied zu sonstigen arithmetischen Ausdrücken bereits vollständige *Anweisungen* (vgl. Abschnitt 4.7), wenn man ein Semikolon dahinter setzt, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  int i = 12;`
`  i++;`
`  Console.WriteLine(i);`
`  }`
`}`
	`13`

Ein (De)inkrementoperator bietet keine eigenständige mathematische Funktion, sondern eine vereinfachte Schreibweise. So ist z. B. die folgende Anweisung

```
j = ++i;
```

mit den beiden **int**-Variablen `i` und `j` äquivalent zu:

```
i = i + 1;
j = i;
```

Für den eventuell bei manchen Lesern noch wenig bekannten Modulo-Operator gibt es einige sinnvolle Anwendungen, z. B.:

- Man kann für eine ganze Zahl bequem feststellen, ob sie gerade (durch 2 teilbar) ist. Dazu prüft man, ob der Rest aus der Division durch 2 gleich 0 ist:

Quellcode-Fragment	Ausgabe
`int i = 19;` `Console.WriteLine(i % 2 == 0);`	`False`

- Man kann bei einer Gleitkommazahl den gebrochenen Anteil ermitteln bzw. abspalten:[1]

Quellcode-Fragment	Ausgabe

`double a = 7.1248239;`
`double rest = a % 1.0;`
`double ganz = a - rest;`
`Console.WriteLine($"{a} = {ganz,1:f0} + {rest}");`
	`7,1248239 = 7 + 0,1248239`

4.5.2 Methodenaufruf

Obwohl Ihnen eine gründliche Behandlung der Methoden noch bevorsteht, haben Sie doch schon einige Erfahrung mit diesen Handlungskompetenzen von Klassen bzw. Objekten gewonnen:

- Die Arbeitsweise einer Methode kann von Argumenten (Parametern) abhängen.
- Viele Methoden liefern ein Ergebnis an den Aufrufer. Die im Abschnitt 4.4.1 vorgestellte Methode **Convert.ToInt32()** liefert z. B. einen **int**-Wert, sofern die als Parameter übergebene Zeichenfolge als ganze Zahl im **int**-Wertebereich (siehe Tabelle im Abschnitt 4.3.4) interpretierbar ist. Bei einer Methodendefinition ist der Datentyp der Rückgabe anzugeben (siehe Syntaxdiagramm im Abschnitt 4.1.2.2).
- Liefert eine Methode dem Aufrufer *kein* Ergebnis, dann ist in der Definition der Pseudo-Rückgabetyp **void** anzugeben.
- Neben der Wertrückgabe hat ein Methodenaufruf oft weitere Effekte, z. B. auf die Merkmalsausprägungen eines handelnden Objekts oder auf die Konsolenausgabe.

In syntaktischer Hinsicht halten wir fest, dass ein Methodenaufruf einen **Ausdruck** darstellt, wobei seine Rückgabe den Datentyp und den Wert des Ausdrucks bestimmt. Wir nehmen auch zur Kenntnis, dass es sich bei dem die Parameterliste begrenzenden Paar runder Klammern um den **() - Operator** handelt.[2] Jedenfalls sollten Sie sich nicht darüber wundern, dass der Methodenaufruf in Tabellen mit den C# - Operatoren auftaucht und dort eine (ziemlich hohe) Bindungskraft (Priorität) besitzt (vgl. Abschnitt 4.5.10).

Bei passendem Rückgabetyp darf ein Methodenaufruf auch als Argument für komplexere Ausdrücke oder für übergeordnete Methodenaufrufe verwendet werden (siehe Abschnitt 5.3.1.2). Bei einer Methode *ohne* Rückgabewert resultiert ein Ausdruck vom Typ **void**, der nicht als Argument für Operatoren oder andere Methoden taugt.

Ein Methodenaufruf mit angehängtem Semikolon stellt eine **Anweisung** dar (vgl. Abschnitt 4.7), was Sie z. B. bei den zahlreichen Einsätzen der statischen Methode **WriteLine()** aus der Klasse **Console** in unseren Beispielprogrammen beobachten konnten.

Mit den arithmetischen Operatoren lassen sich nur elementare mathematische Probleme lösen. Darüber hinaus stellt das .NET - Framework eine große Zahl mathematischer Standardfunktionen (z. B. Potenzfunktion, Logarithmus, Wurzel, trigonometrische Funktionen) über statische Methoden der Klasse **Math** im Namensraum **System** zur Verfügung (siehe BCL-Dokumentation). Im folgenden Programm wird die Methode **Pow()** zur Berechnung der allgemeinen Potenzfunktion (b^e) genutzt:

---

[1] Der (gerundete) ganzzahlige Anteil eines **double**- oder **decimal**-Werts lässt sich auch über die statische Methode **Round()** bzw. **Truncate()** aus der Klasse **Math** bzw. aus der Struktur **Decimal** ermitteln.

[2] Diese Bezeichnung wird auch in Microsofts C# - Referenz verwendet, siehe z. B. http://msdn.microsoft.com/en-us/library/0z4503sa.aspx

Quellcode	Ausgabe

`using System;`
`class Prog{`
`  static void Main(){`
`  Console.WriteLine(Math.Pow(2.0, 3.0));`
`  }`
`}`
	`8`

Alle **Math**-Methoden sind als **static** definiert, werden also von der Klasse selbst ausgeführt.

Im Beispielprogramm liefert die Methode **Math.Pow()** einen Rückgabewert vom Typ **double**, der gleich als Argument der Methode **Console.WriteLine()** Verwendung findet. Solche Verschachtelungen sind bei Programmierern wegen ihrer Kompaktheit ähnlich beliebt wie die Inkrement- bzw. Dekrementoperatoren. Ein etwas umständliches, aber für Einsteiger leichter nachvollziehbares Äquivalent zum obigen **WriteLine()** - Aufruf könnte z. B. so aussehen:

```
double d = Math.Pow(2.0, 3.0);
Console.WriteLine(d);
```

4.5.3 Vergleichsoperatoren

Durch die Anwendung eines *Vergleichsoperators* auf zwei komparable (miteinander vergleichbare) Ausdrücke entsteht ein **Vergleich**. Dies ist ein einfacher **logischer Ausdruck** (vgl. Abschnitt 4.5.5). Folglich kann ein Vergleich die booleschen Werte **true** (wahr) und **false** (falsch) annehmen und zur Formulierung einer Bedingung verwendet werden. Das folgende Beispiel dürfte verständlich sein, obwohl die **if**-Anweisung noch nicht behandelt wurde:

```
if (arg > 0)
  Console.WriteLine(Math.Log(arg));
```

In der folgenden Tabelle mit den von C# unterstützten Vergleichsoperatoren stehen ...

- *Expr1* und *Expr2* für miteinander vergleichbare Ausdrücke
  Hier ein Beispiel für *fehlende* Vergleichbarkeit:

```
Console.WriteLine(2.4 == "2.4");
```

- *Num1* und *Num2* für numerische Ausdrücke

Operator	Bedeutung	Beispiel	
		Programmfragment	Ausgabe
Expr1 == *Expr2*	**gleich**	`string s = "2.4";` `Console.WriteLine(s == "2.4");`	`True`
Expr1 != *Expr2*	**ungleich**	`Console.WriteLine(2 != 3);`	`True`
Num1 > *Num2*	**größer**	`Console.WriteLine(3 > 2);`	`True`
Num1 < *Num2*	**kleiner**	`Console.WriteLine(3 < 2);`	`False`
Num1 >= *Num2*	**größer oder gleich**	`Console.WriteLine(3 >= 3);`	`True`
Num1 <= *Num2*	**kleiner oder gleich**	`Console.WriteLine(3 <= 2);`	`False`

Achten Sie unbedingt darauf, dass der Identitätsoperator durch **zwei** „=“ - Zeichen ausgedrückt wird. Ein nicht ganz seltener C# - Programmierfehler besteht darin, beim Identitätsoperator das zweite Gleichheitszeichen zu vergessen. Dabei muss nicht unbedingt ein harmloser Syntaxfehler entstehen, der nach dem Studium einer Compiler-Fehlermeldung leicht zu beseitigen ist, sondern es kann auch ein mehr oder weniger unangenehmer Semantikfehler resultieren, also ein irreguläres

Verhalten des Programms (vgl. Abschnitt 3.2.4 zur Unterscheidung von Syntax- und Semantikfehlern). Im ersten **WriteLine()** - Aufruf des folgenden Programms wird das Ergebnis eines Vergleichs auf die Konsole geschrieben:[1]

Quellcode	Ausgabe

`using System;`
`class Prog{`
`  static void Main(){`
`  int i = 1;`
`  Console.WriteLine(i == 2);`
`  Console.WriteLine(i);`
`  }`
`}`
	`False`
	`1`

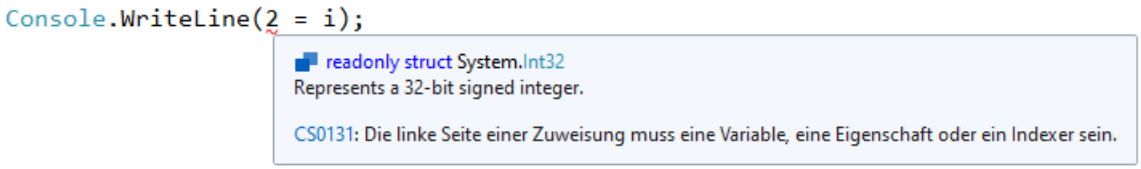
Durch Weglassen eines Gleichheitszeichens wird aus dem Vergleich jedoch ein *Wertzuweisungsausdruck* (siehe Abschnitt 4.5.8) mit dem Typ **int** und dem Wert 2:

Quellcode	Ausgabe

`using System;`
`class Prog{`
`  static void Main(){`
`  int i = 1;`
`  Console.WriteLine(i = 2);`
`  Console.WriteLine(i);`
`  }`
`}`
	`2`
	`2`

Die versehentlich entstandene Zuweisung sorgt nicht nur für eine unerwartete Ausgabe, sondern verändert auch den Wert der Variablen `i`, was im weiteren Verlauf eines Programms unangenehme Folgen haben kann.

Bei einem intendierten Vergleich einer Variablen mit einem Literal schützt das Vertauschen der Operanden vor dem Fehler durch ein vergessenes Gleichheitszeichen, weil die versehentlich zustande gekommene Zuweisung (vgl. Abschnitt 4.5.8) als linken Operanden eine Variable erwartet und folglich von der Entwicklungsumgebung bzw. vom Compiler als fehlerhaft erkannt wird:[2]



[1] Wir wissen schon aus dem Abschnitt 4.2.1, dass **WriteLine()** einen Ausdruck mit beliebigem Typ verarbeiten kann, wobei automatisch eine Zeichenfolgen-Repräsentation erstellt wird.

[2] Diese schlaue Idee stammt von Marian Peters.

4.5.4 Identitätsprüfung bei Gleitkommawerten

Bei den *binären* Gleitkommatypen (**float** und **double**) sind simple Identitätstests wegen technisch bedingter Abweichungen von der reinen Mathematik unbedingt zu unterlassen, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  const double usd = 1.0e-14; // Unterschieds-Schwelle Double`
`  double d1 = 10.0 - 9.9;`
`  double d2 = 0.1;`
`  Console.WriteLine(d1 == d2);`
`  Console.WriteLine(10.0m - 9.9m == 0.1m);`
`  Console.WriteLine(Math.Abs((d1 - d2)/d1) < usd);`
`  }`
`}`
	`False`
	`True`
	`True`

Der Vergleich

```
10.0 - 9.9 == 0.1
```

führt trotz des Datentyps **double** (mit mindestens 15 signifikanten Dezimalstellen) zum Ergebnis `False`. Wenn man die im Abschnitt 4.3.5.1 beschriebenen Genauigkeitsprobleme bei der Speicherung von binären Gleitkommazahlen berücksichtigt, ist das Vergleichsergebnis *nicht* überraschend. Im Kern besteht das Problem darin, dass mit der binären Gleitkommatechnik auch relativ „glatte“ rationale Zahlen (wie z. B. 9,9) nicht exakt gespeichert werden können:[1]

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  double d = 9.9f;`
`  Console.WriteLine("{0:f14}", d);`
`  }`
`}`
	`9,89999961853027`

Im zwischengespeicherten Berechnungsergebnis 10,0 - 9,9 steckt ein anderer Fehler als im Speicherabbild der Zahl 0,1. Weil die Vergleichspartner nicht Bit für Bit identisch sind, meldet der Identitätsoperator das Ergebnis **false**.

Für Anwendungen im Bereich der Finanzmathematik wurde schon im Abschnitt 4.3.5 der dezimale Gleitkommadatentyp **decimal** empfohlen. Mit der *dezimalen* Gleitkommaarithmetik und -speichertechnik resultiert beim Vergleich

```
10.0m - 9.9m == 0.1m
```

das korrekte Ergebnis **true**. Allerdings eignen sich **decimal**-Variablen wegen des relativ kleinen Wertebereichs, des großen Speicherbedarfs und des hohen Rechenzeitaufwands nicht für alle Anwendungen.

Den etwas anstrengenden Rest des Abschnitts kann überspringen, wer aktuell keinen Algorithmus mit auf Identität zu prüfenden **double**-Werten zu implementieren hat.

Um eine praxistaugliche Identitätsbeurteilung von **double**-Werten zu erhalten, sollte eine an der Rechen- bzw. Speichergenauigkeit orientierte **Unterschiedlichkeitsschwelle** verwendet werden.

---

[1] Um dies auch im .NET Framework demonstrieren zu können, wird ein **float**-Wert (erzwungen per Literal-Suffix **f**) in einer **double**-Variablen abgelegt und dann mit 14 Dezimalstellen ausgegeben. Mit einer **float**-Variablen lässt sich das Problem im .NET Framework nicht demonstrieren, weil hier die Ungenauigkeit bei der Ausgabe weggerundet wird.

Nach diesem Vorschlag werden zwei **normalisierte** (also insbesondere von null verschiedene) **double**-Werte d_1 und d_2 (vgl. Abschnitt 4.3.5.1) dann als numerisch identisch betrachtet, wenn der relative Abweichungsbetrag kleiner als $1{,}0 \cdot 10^{-14}$ ist:

$$\left|\frac{d_1 - d_2}{d_1}\right| < 1{,}0 \cdot 10^{-14}$$

Die Vergabe der d_1-Rolle, also die Wahl des Nenners, ist beliebig. Um das Verfahren vollständig festzulegen, wird die Verwendung der betragsmäßig größeren Zahl vorgeschlagen.

Ein Vorschlag zur Definition der *numerischen Identität* von zwei **double**-Werten muss die *relative* Differenz zugrunde legen, weil die technisch bedingten Mantissenfehler bei zwei **double**-Variablen mit eigentlich identischem Wert in Abhängigkeit vom Exponenten zu sehr unterschiedlichen Gesamtfehlern in der Differenz führen können. Vom gelegentlich anzutreffenden Vorschlag, die betragsmäßige Differenz

$$\left|d_1 - d_2\right|$$

mit einer Schwelle zu vergleichen, ist daher abzuraten. Dieses Verfahren ist (bei geeignet gewählter Schwelle) nur tauglich für Zahlen in einem engen Größenbereich. Bei einer Änderung der Größenordnung muss die Schwelle angepasst werden.

Zu einer Schwelle für die relative Abweichung $\left|\frac{d_1 - d_2}{d_1}\right|$ gelangt man durch Betrachtung von zwei normalisierten **double**-Variablen d_1 und d_2, die bis auf ihre durch begrenzte Speicher- und Rechengenauigkeit bedingten Mantissenfehler e_1 bzw. e_2 denselben Wert $(1 + m)\, 2^k$ enthalten:

$$d_1 = (1 + m + e_1)\, 2^k \quad \text{und} \quad d_2 = (1 + m + e_2)\, 2^k$$

Bei einem normalisierten **double**-Wert (mit 52 Mantissen-Bits) kann aufgrund der begrenzten Speichergenauigkeit als maximaler absoluter Mantissenfehler ε der halbe Abstand zwischen zwei benachbarten Mantissenwerten auftreten:

$$\varepsilon = 2^{-53} \approx 1{,}1 \cdot 10^{-16}$$

Für den Betrag der technisch bedingten relativen Abweichung von zwei eigentlich identischen normalisierten Werten (mit einer Mantisse im Intervall [1, 2)) gilt die Abschätzung:

$$\left|\frac{d_1 - d_2}{d_1}\right| = \left|\frac{e_1 - e_2}{1 + m + e_1}\right| \le \frac{|e_1| + |e_2|}{|1 + m + e_1|} \le \frac{2 \cdot \varepsilon}{|1 + m + e_1|} \le 2 \cdot \varepsilon \quad (\text{wegen} (1 + m + e_1) \in [1, 2))$$

Die oben vorgeschlagene Schwelle $1{,}0 \cdot 10^{-14}$ berücksichtigt über den Speicherfehler hinaus auch noch eingeflossene Rechnungsungenauigkeiten. Mit welcher Fehlerkumulation bzw. -verstärkung zu rechnen ist, hängt vom konkreten Algorithmus ab, sodass die Unterschiedlichkeitsschwelle eventuell angehoben werden muss. Immerhin hängt sie (anders als bei einem Kriterium auf Basis des Betrags der einfachen Differenz $|d_1 - d_2|$) nicht von der Größenordnung der Zahlen ab.

An der vorgeschlagenen Identitätsbeurteilung mit Hilfe einer Schwelle für den relativen Abweichungsbetrag ist u. a. zu bemängeln, dass eine Verallgemeinerung für die mit geringerer Genauigkeit gespeicherten *denormalisierten* Werte (Betrag kleiner als 2^{-1022} beim Typ **double**, siehe Abschnitt 4.3.5.1) benötigt wird.

Dass die definierte Indifferenzrelation nicht transitiv ist, muss hingenommen werden. Für drei **double**-Werte *a*, *b* und *c* kann also das folgende Ergebnismuster auftreten:

- *a* numerisch identisch mit *b*
- *b* numerisch identisch mit *c*
- *a nicht* numerisch identisch mit *c*

Für den Vergleich einer **double**-Zahl *a* mit dem Wert null ist eine Schwelle für die *absolute* Abweichung (statt der relativen) sinnvoll, z. B.:

$$|a| < 1{,}0 \cdot 10^{-14}$$

Die besprochenen Genauigkeitsprobleme sind auch bei den gerichteten Vergleichen (<, <=, >, >=) relevant.

Bei vielen naturwissenschaftlichen oder technischen Problemen ist es generell wenig sinnvoll, zwei Größen auf exakte Übereinstimmung zu testen, weil z. B. schon aufgrund von Messungenauigkeiten eine Abweichung von der theoretischen Identität zu erwarten ist. Bei Verwendung einer anwendungslogisch gebotenen Unterschiedsschwelle dürften die technischen Beschränkungen der Gleitkommatypen keine große Rolle mehr spielen. Präzisere Aussagen zur Computer-Arithmetik finden sich z. B. bei Strey (2005).

4.5.5 Logische Operatoren

Aus dem Abschnitt 4.5.3 wissen wir, dass jeder Vergleich (z. B. `arg > 0`) bereits ein logischer Ausdruck ist, also die Werte **true** und **false** annehmen kann. Durch Anwendung der logischen Operatoren (Negation, logisches UND, logisches ODER, exklusives ODER) auf bereits vorhandene logische Ausdrücke kann man neue, komplexere logische Ausdrücke erstellen. Die Wirkungsweise der logischen Operatoren wird in **Wahrheitstafeln** beschrieben (*La1* und *La2* seien logische Ausdrücke):

Argument *La1*	**Negation** **!***La1*
true	**false**
false	**true**

Argument 1 *La1*	**Argument 2** *La2*	**Logisches UND** *La1* **&&** *La2* *La1* **&** *La2*	**Logisches ODER** *La1* **||** *La2* *La1* **|** *La2*	**Exklusives ODER** *La1* **^** *La2*
true	**true**	**true**	**true**	**false**
true	**false**	**false**	**true**	**true**
false	**true**	**false**	**true**	**true**
false	**false**	**false**	**false**	**false**

Es folgt eine Tabelle mit Erläuterungen und Beispielen zu den logischen Operatoren:

Operator	Bedeutung	Beispiel	
		Programmfragment	**Ausgabe**
!*La1*	**Negation** Der Wahrheitswert wird umgekehrt.	`bool erg = true;` `Console.WriteLine(!erg);`	`False`

La1 && *La2*	**Logisches UND (mit bedingter Auswertung)**
	La1 && *La2* ist genau dann wahr, wenn beide Argumente wahr sind. Ist *La1* falsch, wird *La2* **nicht** ausgewertet.
		`int i = 3;`
		`bool erg = false && i++ > 3;`
		`Console.WriteLine(erg + "\n" + i);`
		
		`erg = true && i++ > 3;`
		`Console.WriteLine(erg + "\n" + i);`
			`False`
			`3`
			
			`False`
			`4`

La1 & *La2*	**Logisches UND (mit unbedingter Auswertung)**
	La1 & *La2* ist genau dann wahr, wenn beide Argumente wahr sind. Es werden auf jeden Fall **beide** Ausdrücke ausgewertet.
		`int i = 3;`
		`bool erg = false & i++ > 3;`
		`Console.WriteLine(erg + "\n" + i);`
			`False`
			`4`

La1 || *La2*	**Logisches ODER (mit bedingter Auswertung)**
	La1 || *La2* ist genau dann wahr, wenn mindestens ein Argument wahr ist. Ist *La1* wahr, wird *La2* **nicht** ausgewertet.
		`int i = 3;`
		`bool erg = true || i++ == 3;`
		`Console.WriteLine(erg + "\n" + i);`
		
		`erg = false || i++ == 3;`
		`Console.WriteLine(erg + "\n" + i);`
			`True`
			`3`
			
			`True`
			`4`

La1 | *La2*	**Logisches ODER (mit unbedingter Auswertung)**
	La1 | *La2* ist genau dann wahr, wenn mindestens ein Argument wahr ist. Es werden auf jeden Fall **beide** Ausdrücke ausgewertet.
		`int i = 3;`
		`bool erg = true | i++ > 3;`
		`Console.WriteLine(erg + "\n" + i);`
			`True`
			`4`

La1 ^ *La2*	**Exklusives logisches ODER** *La1* ^ *La2* ist genau dann wahr, wenn **genau ein** Argument wahr ist, wenn also die Argumente verschiedene Wahrheitswerte haben.	`Console.WriteLine(true ^ true);`	`False`

Der Unterschied zwischen den beiden logischen UND-Operatoren **&&** und **&** bzw. zwischen den beiden logischen ODER-Operatoren || und | ist für Einsteiger vielleicht wenig beeindruckend, weil man spontan den nicht ausgewerteten logischen Ausdrücken keine Bedeutung beimisst. Allerdings ist es in C# nicht ungewöhnlich, „Nebeneffekte“ in einen logischen Ausdruck einzubauen, z. B.:

```
b & i++ > 3
```

Hier erhöht der Postinkrementoperator beim Auswerten des rechten **&**-Arguments den Wert der Variablen `i`. Eine solche Auswertung wird jedoch in der folgenden Variante des Beispiels (mit **&&**-Operator) unterlassen, wenn bereits nach Auswertung des linken **&&**-Arguments das Gesamtergebnis **false** feststeht:

```
b && i++ > 3
```

Man spricht hier von einer *Kurzschlussauswertung* (engl.: *short-circuiting*). Das vom Programmierer nicht erwartete Ausbleiben einer Auswertung (z. B. bei `i++`) kann erhebliche Auswirkungen auf die Programmausführung haben.

Dank der beim Operator **&&** realisierten bedingten Auswertung kann man sich im rechten Operanden darauf verlassen, dass der linke Operand den Wert **true** besitzt, was im folgenden Beispiel aus-

genutzt wird. Dort prüft der linke Operand die Existenz und der rechte Operand die Länge einer Zeichenfolge:

```
str != null && str.Length < 10
```

Wenn die Referenzvariable `str` vom Typ der Klasse **String** keine Objektadresse enthält, darf der rechte Ausdruck nicht ausgewertet werden, weil eine Längenanfrage (per Eigenschaftszugriff) an ein nicht existentes Objekt zu einem Laufzeitfehler führen würde.

Mit der Entscheidung, grundsätzlich die unbedingte Operatorvariante zu verwenden, verzichtet man auf die eben beschriebene Option, im rechten Ausdruck den Wert **true** des linken Ausdrucks voraussetzen zu können, und man nimmt (mehr oder weniger relevante) Leistungseinbußen durch überflüssige Auswertungen des rechten Ausdrucks in Kauf. Eher empfehlenswert ist der Verzicht auf Nebeneffekt-Konstruktionen im Zusammenhang mit bedingt arbeitenden Operatoren.

Wie der Tabelle auf Seite 180 zu entnehmen ist, unterscheiden sich die beiden UND-Operatoren **&&** und **&** bzw. die beiden ODER-Operatoren || und | auch hinsichtlich der Bindungskraft auf Operanden (Auswertungspriorität).

Um die Verwirrung noch ein wenig zu steigern, werden die Zeichen **&** und | auch für *bitorientierte* Operatoren verwendet (siehe Abschnitt 4.5.6). Diese Operatoren erwarten zwei *integrale* Argumente (z. B. Datentyp **int**), während die logischen Operatoren den Datentyp **bool** voraussetzen. Folglich kann der Compiler erkennen, ob ein logischer oder ein bitorientierter Operator gemeint ist.

4.5.6 Bitorientierte Operatoren

Über unseren momentanen Bedarf hinausgehend bietet C# einige Operatoren zur bitweisen Analyse und Manipulation von Variableninhalten. Solche Techniken werden nur bei speziellen Anwendungen benötigt, sodass der Abschnitt beim ersten Lesen des Manuskripts übersprungen werden kann.

Statt einer systematischen Darstellung der verschiedenen Operatoren (siehe C# - Sprachspezifikation in ECMA 2017, Kapitel 12) beschränken wir uns auf ein Beispielprogramm, das zudem nützliche Einblicke in die Speicherung von **char**-Daten im Arbeitsspeicher eines Computers erlaubt. Allerdings sind Beispiel und zugehörige Erläuterungen mit einigen technischen Details belastet. Wenn Ihnen der Sinn momentan nicht danach steht, können Sie den aktuellen Abschnitt ohne Sorge um den weiteren Kurserfolg an dieser Stelle verlassen.

Das Programm `CharBits` liefert die Unicode-Codierung zu einem vom Benutzer erfragten Zeichen. Dabei kommt die statische Methode **ToChar()** der Klasse **Convert** aus dem Namensraum **System** zum Einsatz. Außerdem wird mit der **for**-Schleife eine Wiederholungsanweisung verwendet, die erst im Abschnitt 4.7.3.1 offiziell vorgestellt wird. Im Beispiel startet die **int**-Indexvariable `i` mit dem Wert 15, der am Ende jedes Schleifendurchgangs um eins dekrementiert wird (`i--`). Ob es zum nächsten Schleifendurchgang kommt, hängt von der Fortsetzungsbedingung ab (`i >= 0`):

Quellcode	Ausgabe (Eingaben **fett**)

`using System;`
`class CharBits {`
`  static void Main() {`
`  char cbit;`
`  Console.Write("Zeichen: ");`
`  cbit = Convert.ToChar(Console.ReadLine());`
`  Console.Write("Unicode: ");`
`  for (int i = 15; i >= 0; i--)`
`  Console.Write((1 << i & cbit) >> i);`
`  }`
`}`
	`Zeichen: x`
	`Unicode: 0000000001111000`

Der **Links-Shift - Operator** << im Ausdruck:

```
1 << i
```

verschiebt die Bits in der binären Repräsentation der Ganzzahl Eins um `i` Stellen nach links, wobei am linken Rand `i` Ziffern verworfen werden und auf der rechten Seite `i` Nullen nachrücken. Von den 32 Bits, die ein **int**-Wert insgesamt belegt (siehe Abschnitt 4.3.4), interessieren im Augenblick nur die rechten 16. Bei der Eins erhalten wir:

```
0000000000000001
```

Im 10. Schleifendurchgang (`i = 6`) geht dieses Muster z. B. über in:

```
0000000001000000
```

Nach dem Links-Shift- kommt der **bitweise UND-Operator** zum Einsatz:

```
1 << i & cbit
```

Das Operatorzeichen **&** wird in C# leider in doppelter Bedeutung verwendet: Wenn beide Argumente vom Typ **bool** sind, wird **&** als *logischer* Operator interpretiert (siehe Abschnitt 4.5.5). Sind jedoch (wie im vorliegenden Fall) beide Argumente von integralem Typ, was auch für den Typ **char** zutrifft, dann wird **&** als UND-Operator für Bits aufgefasst. Er erzeugt dann ein Bitmuster, das genau dann an der Stelle *k* eine 1 enthält, wenn *beide* Argumentmuster an dieser Stelle eine 1 besitzen. Hat in einem Programmablauf die **char**-Variable `cbit` z. B. den Wert `'x'` erhalten (dezimale Unicode-Zeichensatznummer 120), dann ist dieses Bitmuster

```
0000000001111000
```

im Spiel, und `1 << i & cbit` liefert z. B. bei `i = 6` das Muster:

```
0000000001000000
```

Per **Rechts-Shift - Operator** >> werden die Bits um `i` Stellen nach rechts verschoben, wobei am rechten Rand `i` Ziffern verworfen werden und auf der linken Seite `i` Nullen nachrücken:

```
(1 << i & cbit) >> i
```

Die signifikante Ziffer an der Position `i` wird zum rechten Rand befördert, sodass insgesamt ein Bitmuster vom Typ **int** entsteht, das den Wert 0 oder 1 besitzt.[1] Der Wert 1 resultiert genau dann, wenn das zum aktuellen `i`-Wert korrespondierende Bit in der Binärdarstellung des untersuchten Zeichens den Wert 1 hat.

4.5.7 Typumwandlung (Casting) bei elementaren Datentypen

Wie Sie aus dem Abschnitt 4.3.1 wissen, ist in C# der Datentyp einer Variablen in der Regel unveränderlich, und dieses Prinzip wird im aktuellen Abschnitt keineswegs aufgeweicht. Es gibt aber gelegentlich einen Grund dafür, z. B. den Inhalt einer **int**-Variablen in eine **double**-Variable zu übertragen. Aufgrund der abweichenden Speichertechniken ist dann eine Typanpassung fällig. Das geschieht manchmal automatisch auf Initiative des Compilers, kann aber auch vom Programmierer explizit angefordert werden.

---

[1] Die runden Klammern sind erforderlich, um die korrekte Auswertungsreihenfolge zu erreichen (siehe Abschnitt 4.5.10).

4.5.7.1 Automatische erweiternde Typanpassung

Bei der Auswertung des Ausdrucks

```
2.3 / 7
```

trifft der Divisionsoperator auf ein **double**- und ein **int**-Argument. Der C# - Compiler kennt sieben vordefinierte Divisionsoperator-Varianten (siehe C# - Sprachspezifikation in ECMA 2017, Abschnitt 12.9.3):

- Ganzzahldivision
  - **int operator /(int x, int y)**
  - **uint operator /(uint x, uint y)**
  - **long operator /(long x, long y)**
  - **ulong operator /(ulong x, ulong y)**
- Binäre Fließkommadivision
  - **float operator /(float x, float y)**
  - **double operator /(double x, double y)**
- Dezimale Fließkommadivision
  - **decimal operator /(decimal x, decimal y)**

Wenn bei einem Argument oder bei beiden Argumenten der Datentyp nicht passt, findet nach Möglichkeit eine automatische (implizite) Typanpassung „nach oben“ statt. Im Beispiel wird das **int**-Argument in den Datentyp **double** gewandelt und eine binäre Fließkommadivision durchgeführt.

In vergleichbaren Situationen (z. B. bei Wertzuweisungen) kommt es automatisch zu den folgenden **erweiternden Typanpassungen**:[1]

Der Typ …	**wird nach Bedarf automatisch konvertiert in:**
sbyte	short, int, long, float, double, decimal
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long	float, double, decimal
char	ushort, int, uint, long, ulong, float, double, decimal
float	double
ulong	float, double, decimal

Bei den Konvertierungen von **int**, **uint** oder **long** in **float** sowie von **long** in **double** kann es zu einem Verlust an Genauigkeit kommen, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  long i = 9223372036854775313;`
`  double d = i;`
`  Console.WriteLine(i);`
`  Console.WriteLine("{0:f}", d);`
`  }`
`}`
	`9223372036854775313`
	`9223372036854780000,00`

[1] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/conversions#implicit-conversions

Eine Abweichung von 4687 (z. B. Meter) kann durchaus unerfreuliche Konsequenzen haben (z. B. beim Versuch, auf einem Astroiden zu landen).

Weil eine **char**-Variable die Unicode-Nummer eines Zeichens speichert, macht ihre Konvertierung in numerische Typen kein Problem, z. B.:

Quellcode	Ausgabe

`class Prog {`
`  static void Main() {`
`  System.Console.WriteLine("x/2 \t= " + 'x' / 2);`
`  System.Console.WriteLine("x*0,27 \t= " + 'x' * 0.27);`
`  }`
`}`
	`x/2  = 60`
	`x*0,27  = 32,4`

4.5.7.2 Explizite Typumwandlung

Gelegentlich gibt es gute Gründe, über den sogenannten **Casting-Operator** eine *explizite* Typumwandlung zu erzwingen. Im nächsten Beispielprogramm wird mit

```
(int)'x'
```

die **int**-erpretation des (aus dem Abschnitt 4.5.6 bekannten) Bitmusters zum kleinen „x“ vorgenommen, damit Sie nachvollziehen können, warum das Beispielprogramm im vorigen Abschnitt beim „Halbieren“ dieses Zeichens auf den Wert 60 kam:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  Console.WriteLine((int)'x');`
`  double a = 3.7615926;`
`  int i = (int)a;`
`  Console.WriteLine(i);`
`  Console.WriteLine((int)(a + 0.5));`
`  a = 214748364852.13;`
`  Console.WriteLine((int)a);`
`  }`
`}`
	`120`
	`3`
	`4`
	`-2147483648`

Manchmal ist es erforderlich, einen Gleitkommawert in eine Ganzzahl zu wandeln, weil z. B. bei einem Methodenaufruf für einen Parameter ein ganzzahliger Datentyp benötigt wird. Dabei werden die Nachkommastellen abgeschnitten.

Bei nicht-negativen Gleitkommawerten resultiert ein Runden auf die nächstgelegene ganze Zahl, wenn man vor der Typkonvertierung 0,5 addiert.

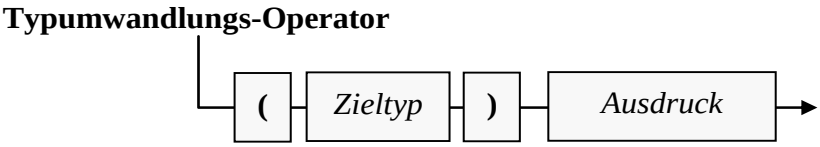
Es ist auf jeden Fall zu beachten, dass dabei eine **einschränkende Konvertierung** stattfindet, und dass die zu erwartende Gleitkommazahl im Wertebereich des Ganzzahltyps liegen muss. Wie die letzte Ausgabe zeigt, sind kapitale Programmierfehler möglich, wenn die Wertebereiche der beteiligten Variablen bzw. Datentypen nicht beachtet werden und bei der Zielvariablen ein Überlauf auftritt (vgl. Abschnitt 4.6.1). So soll die Explosion der europäischen Weltraumrakete Ariane 5 am 4. Juni 1996 (Schaden: ca. 500 Millionen Dollar)

durch die Konvertierung eines **double**-Werts (mögliches Maximum: $1{,}7976931348623157 \cdot 10^{308}$) in einen **short**-Wert (mögliches Maximum: $2^{15}-1 = 32767$) verursacht worden sein. Die kritische Typumwandlung hatte bei der langsameren Rakete Ariane 4 noch keine Probleme gemacht. Offenbar sind profunde Kenntnisse über elementare Sprachelemente unverzichtbar für eine erfolgreiche Raketenforschung und -entwicklung.

Später wird sich zeigen, dass auch zwischen Referenztypen gelegentlich eine explizite Wandlung erforderlich ist.

Welche expliziten Typkonvertierungen in C# erlaubt sind, ist der C# - Sprachspezifikation zu entnehmen (ECMA 2017, Abschnitt 11.3).

Die C# - Syntax zur expliziten Typumwandlung:



Am Rand soll noch erwähnt werden, dass die Wandlung in einen Ganzzahltyp keine sinnvolle Technik ist, um die Nachkommastellen in einem Gleitkommawert zu entfernen oder zu extrahieren. Dazu kann man z. B. den Modulo-Operator verwenden (vgl. Abschnitt 4.5.1), ohne ein Wertebereichsproblem befürchten zu müssen, z. B.:[1]

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  double a = 2147483648.13, b;`
`  int i = (int)a;`
`  b = a - a % 1;`
`  Console.WriteLine("{0}\n{1}\n{2}", a, i, b);`
`  }`
`}`
	`2147483648,13`
	`-2147483648`
	`2147483648`

4.5.8 Zuweisungsoperatoren

Bei den ersten Erläuterungen zur Wertzuweisung (vgl. Abschnitt 4.3.6) blieb aus didaktischen Gründen unerwähnt, dass eine Wertzuweisung ein *Ausdruck* ist, dass wir es also mit dem binären (zweistelligen) Operator „=“ zu tun haben, für den die folgenden Regeln gelten:

- Auf der linken Seite muss eine Variable oder eine Eigenschaft stehen.
- Auf der rechten Seite muss ein Ausdruck mit kompatiblem Typ stehen.
- Der zugewiesene Wert stellt auch den Ergebniswert des Ausdrucks dar.

---

[1] Der (gerundete) ganzzahlige Anteil eines **double**-Wertes lässt sich auch über die statische Methode **Round()** bzw. **Truncate()** aus der Klasse **Math** ermitteln.

Wie beim Inkrement- bzw. Dekrementoperator sind auch beim Zuweisungsoperator zwei Effekte zu unterscheiden:

- Die als linkes Argument fungierende Variable oder Eigenschaft erhält einen neuen Wert.
- Es wird ein Wert für den Ausdruck produziert.

Im folgenden Beispiel fungiert ein Zuweisungsausdruck als Parameter für einen **WriteLine()** - Methodenaufruf:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  int ivar = 13;`
`  Console.WriteLine(ivar = 4711);`
`  Console.WriteLine(ivar);`
`  }`
`}`
	`4711`
	`4711`

Beim Auswerten des Ausdrucks `ivar = 4711` entsteht der an **WriteLine()** zu übergebende Wert (identisch mit dem zugewiesenen Wert), *und* die Variable `ivar` wird verändert.

Selbstverständlich kann eine Zuweisung auch als Operand in einen übergeordneten Ausdruck integriert werden, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  int i = 2, j = 4;`
`  i = j = j * i;`
`  Console.WriteLine(i + "\n" + j);`
`  }`
`}`
	`8`
	`8`

Beim mehrfachen Auftreten des Zuweisungsoperators ist seine **Rechts-Assoziativität** relevant (vgl. Tabelle im Abschnitt 4.5.10). Sie bewirkt, dass im Beispielprogramm die Anweisung

```
i = j = j * i;
```

folgendermaßen ausgeführt wird:

- Weil der Multiplikationsoperator eine höhere Bindungskraft besitzt als der Zuweisungsoperator (siehe Abschnitt 4.5.10.1), wird zuerst der Ausdruck `j * i` ausgewertet, was zum Zwischenergebnis 8 (mit Datentyp **int**) führt.
- Die Rechts-Assoziativität des Zuweisungsoperators führt zu der anschließend durch runde Klammern hervorgehobenen Zuordnung der Operanden:

  ```
  i = (j = 8)
  ```

  Folglich wird nach der Multiplikation die *rechte* Zuweisung ausgeführt. Der folgende Ausdruck mit Wert 8 und Typ **int**

  ```
  j = 8
  ```

  verschafft der Variablen `j` einen neuen Wert.
- In der zweiten Zuweisung (bei Betrachtung von rechts nach links) wird der Wert des Ausdrucks `j = 8` an die Variable `i` übergeben.

Anweisungen der Art

```
i = j = k;
```

stammen nicht aus einem Kuriositätenkabinett, sondern sind in C# - Programmen oft anzutreffen, weil im Vergleich zur Alternative

```
j = k;
i = k;
```

Schreibaufwand gespart wird.

Wie wir seit dem Abschnitt 4.3.6 wissen, stellt ein Zuweisungsausdruck bereits eine vollständige **Anweisung** dar, sobald man ein Semikolon dahinter setzt. Dies gilt auch für die die Prä- und Post-inkrementausdrücke (vgl. Abschnitt 4.5.1) sowie für Methodenaufrufe, jedoch *nicht* für die anderen Ausdrücke, die im Abschnitt 4.5 vorgestellt werden.

Für die häufig benötigten Zuweisungen nach dem Muster

```
j = j * i;
```

(eine Variable erhält einen neuen Wert, an dessen Konstruktion sie selbst mitwirkt), bietet C# spezielle Zuweisungsoperatoren, die als *Aktualisierungsoperatoren* oder als *Verbundzuweisungs-Operatoren* (engl.: *compound assignment operators*) bezeichnet werden. Im Beispiel werden durch die Verwendung des Operators **`*=`**

```
j *= i;
```

zwei Vorteile erzielt:

- Der Code wird kürzer.
- Die Variable `j` muss nur *einmal* ausgewertet werden.

In der folgenden Tabelle steht *Var* für eine numerische Variable und *Expr* für einen typkompatiblen Ausdruck:

Operator	**Bedeutung**	**Beispiel**	
		Programmfragment	**Neuer Wert von `i`**
Var **`+=`** *Expr*	*Var* erhält den neuen Wert *Var* + *Expr*.	`int i = 2;` `i += 3;`	`5`
Var **`-=`** *Expr*	*Var* erhält den neuen Wert *Var* − *Expr*.	`int i = 10, j = 3;` `i -= j * j;`	`1`
Var **`*=`** *Expr*	*Var* erhält den neuen Wert *Var* * *Expr*.	`int i = 2;` `i *= 5;`	`10`
Var **`/=`** *Expr*	*Var* erhält den neuen Wert *Var* / *Expr*.	`int i = 10;` `i /= 5;`	`2`
Var **`%=`** *Expr*	*Var* erhält den neuen Wert *Var* % *Expr*.	`int i = 10;` `i %= 5;`	`0`

Eine Marginalie: Während für zwei **byte**-Variablen

```
byte b1 = 1, b2 = 2;
```

die folgende Zuweisung

```
b1 = b1 + b2;
```

verboten ist, weil der Ausdruck `(b1 + b2)` den Typ **int** besitzt (vgl. Abschnitt 4.5.1), akzeptiert der Compiler den äquivalenten Ausdruck mit Aktualisierungsoperator:

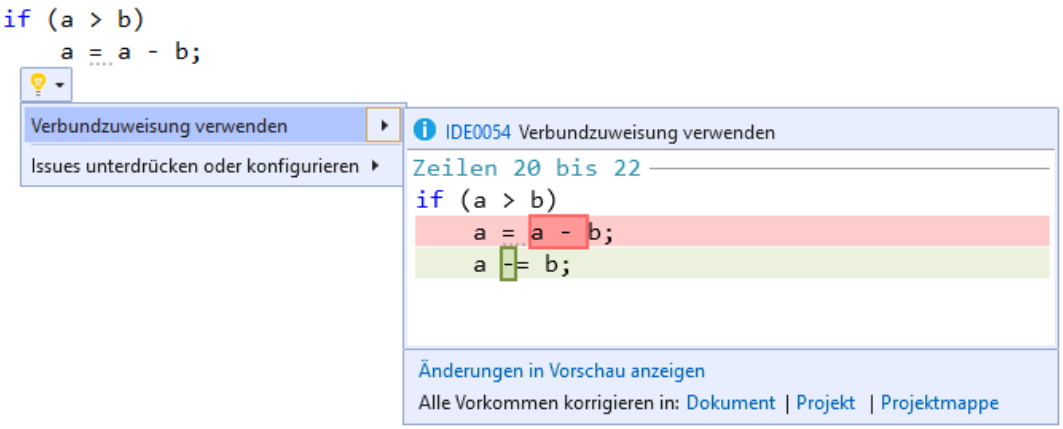
```
b1 += b2;
```

Allerdings soll diese Randbemerkung nicht als Geheimtipp für cleveres Programmieren verstanden werden, weil sich das Überlaufrisiko (vgl. Abschnitt 4.6.1) erhöht:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  byte b1 = 200, b2 = 200;`
`  b1 += b2;`
`  Console.WriteLine(b1);`
`  }`
`}`
	`144`

Wer sich dafür entscheidet, auf die Aktualisierungsoperatoren zu verzichten, wird vom Visual Studio per Voreinstellung zur kompakten Schreibweise aufgefordert, z. B.:



4.5.9 Konditionaloperator

Der **Konditionaloperator** erlaubt eine sehr kompakte Schreibweise, wenn beim neuen Wert für eine Zielvariable bedingungsabhängig zwischen zwei Ausdrücken zu entscheiden ist, z. B.

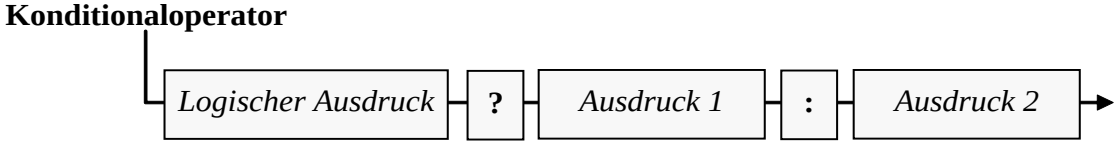
$$i = \begin{cases} i + j & \text{falls } k > 0 \\ i - j & \text{sonst} \end{cases}$$

In C# ist für diese Zuweisung mit Fallunterscheidung nur eine einzige Zeile erforderlich:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  int i = 2, j = 1, k = 7;`
`  i = k>0 ? i+j : i-j;`
`  Console.WriteLine(i);`
`  }`
`}`
	`3`

Eine Besonderheit des Konditionaloperators besteht darin, dass er *drei* Argumente verarbeitet, welche durch die Zeichen **?** und **:** getrennt werden:



Ist der logische Ausdruck *wahr*, liefert der Konditionaloperator den Wert von *Ausdruck 1*, anderenfalls den Wert von *Ausdruck 2*.

Die Frage nach dem Typ eines Konditionalausdrucks ist etwas knifflig, und in der C# - Sprachspezifikation werden etliche Fälle unterschieden (ECMA 2017, Abschnitt 12.15). Es liegt an Ihnen, sich auf den einfachsten und wichtigsten Fall zu beschränken: Wenn der zweite und der dritte Operand denselben Typ haben, ist dies auch der Typ des Konditionalausdrucks.

4.5.10 Auswertungsreihenfolge

Bisher haben wir zusammengesetzte Ausdrücke mit *mehreren* Operatoren und das damit verbundene Problem der Auswertungs*reihenfolge* nach Möglichkeit gemieden. Wie sich gleich zeigen wird, sind für Schwierigkeiten und Fehler bei der Verwendung zusammengesetzter Ausdrücke die folgenden Gründe hauptverantwortlich:

- Komplexität des Ausdrucks (Anzahl der Operatoren, Schachtelungstiefe)
- Operatoren mit Nebeneffekten

Um Problemen aus dem Weg zu gehen, sollte man also eine übertriebene Komplexität vermeiden und auf Nebeneffekte weitgehend verzichten.

4.5.10.1 Regeln

In diesem Abschnitt werden die Regeln vorgestellt, nach denen der C# - Compiler einen Ausdruck mit mehreren Operatoren auswertet.

1) Runde Klammern

Wenn aus den anschließend erläuterten Regeln zur Bindungskraft und Assoziativität der beteiligten Operatoren nicht die gewünschte Operandenzuordnung bzw. Auswertungsreihenfolge resultiert, dann greift man mit runden Klammern steuernd ein, wobei auch eine Schachtelung erlaubt ist. Durch Klammern werden Terme zu *einem* Operanden zusammengefasst, sodass die *internen* Operationen ausgeführt sind, bevor der Klammerausdruck von einem *externen* Operator verarbeitet wird.

2) Bindungskraft (Priorität)

Steht ein Operand (ein Ausdruck) zwischen zwei Operatoren, dann wird er dem Operator mit der stärkeren Bindungskraft (siehe Tabelle im Abschnitt 4.5.10.2) zugeordnet. Mit den numerischen Variablen `a`, `b` und `c` als Operanden wird z. B. der Ausdruck

```
a + b * c
```

nach der Regel „*Punktrechnung geht vor Strichrechnung*“ interpretiert als

```
a + (b * c)
```

In der Konkurrenz um die Zuständigkeit für den Operanden `b` hat der Multiplikationsoperator Vorrang gegenüber dem Additionsoperator.

Die implizite Klammerung kann durch eine explizite Klammerung dominiert werden:

```
(a + b) * c
```

3) Assoziativität (Orientierung)

Steht ein Operand zwischen zwei Operatoren mit *gleicher* Bindungskraft, dann entscheidet deren Assoziativität (Orientierung) über die Zuordnung des Operanden:

- Mit Ausnahme der Zuweisungsoperatoren sind alle binären Operatoren *links-assoziativ*. Z. B. wird

  ```
  x - y - z
  ```

  ausgewertet als

  ```
  (x - y) - z
  ```

  Diese implizite Klammerung kann durch eine explizite Klammerung dominiert werden:

  ```
  x - (y - z)
  ```

- Die Zuweisungsoperatoren sind *rechts-assoziativ*. Z. B. wird

  ```
  a += b -= c = d
  ```

  ausgewertet als

  ```
  a += (b -= (c = d))
  ```

  Diese implizite Klammerung kann *nicht* durch eine explizite Klammerung geändert werden, weil der linke Operand einer Zuweisung eine Variable oder eine Eigenschaft sein muss.

In C# ist dafür gesorgt, dass Operatoren mit gleicher Bindungskraft stets auch die gleiche Assoziativität besitzen, z. B. die im letzten Beispiel enthaltenen Operatoren `+=`, `-=` und `=`.

Für manche Operationen gilt das mathematische Assoziativgesetz, sodass die Reihenfolge der Auswertung irrelevant ist, z. B.:

$$(3 + 2) + 1 = 6 \quad = \quad 3 + (2 + 1)$$

Anderen Operationen fehlt diese Eigenschaft, z. B.:

$$(3 - 2) - 1 = 0 \quad \neq \quad 3 - (2 - 1) = 2$$

Während sich die Addition und die Multiplikation von *Ganzzahltypen* in C# tatsächlich assoziativ verhalten, gilt das aus technischen Gründen für die Addition und die Multiplikation von *Gleitkommatypen* nur approximativ.

4) Links vor rechts bei der Auswertung der Argumente eines binären Operators

Bevor ein Operator ausgeführt werden kann, müssen erst seine Argumente (Operanden) ausgewertet sein. Bei jedem binären Operator ist in C# sichergestellt, dass erst der linke Operand ausgewertet wird, dann der rechte.[1] Im folgenden Beispiel tritt der Ausdruck `++ivar` als rechter Operand einer Multiplikation auf. Die hohe Bindungskraft (Priorität) des Präinkrementoperators (siehe Tabelle im Abschnitt 4.5.10.2) führt *nicht* dazu, dass sich der Nebeneffekt des Ausdrucks `++ivar` auf den linken Operanden der Multiplikation auswirkt:

Quellcode	Ausgabe

`class Prog {`
`  static void Main() {`
`  int ivar = 2;`
`  int erg = ivar * ++ivar;`
`  System.Console.WriteLine("{0} {1}", erg, ivar);`
`  }`
`}`
	`6 3`

[1] In den folgenden Fällen unterbleibt die Auswertung des rechten Operanden:

- Bei der Auswertung des linken Operanden kommt es zu einem Ausnahmefehler (siehe unten).
- Bei den logischen Operatoren mit *bedingter* Ausführung (**&&**, ||) verhindert ein bestimmter Wert des linken Operanden die Auswertung des rechten Operanden (siehe Abschnitt 4.5.5).

Die Auswertung des Ausdrucks `ivar * ++ivar` verläuft so:

- Zuerst wird der linke Operand der Multiplikation ausgewertet (Ergebnis: 2)
- Dann wird der rechte Operand der Multiplikation ausgewertet:
  - Die Präinkrementoperation hat einen Nebeneffekt auf die Variable `ivar`.
  - Der Ausdruck `++ivar` hat den Wert 3.
- Die Ausführung der Multiplikationsoperation liefert schließlich das Endergebnis 6.

Das Beispiel zeigt auch, dass der Begriff der *Bindungskraft* gegenüber dem Begriff der *Priorität* zu bevorzugen ist. Weil sich kein Operand zwischen den Operatoren ***** und ++ befindet, können deren Bindungskräfte offensichtlich keine Rolle spielen. Der Begriff der *Priorität* suggeriert aber trotzdem, dass der Präinkrementoperator einen Vorrang bei der Ausführung hätte.

Wie eine leichte Variation des letzten Beispiels zeigt, kann sich ein Nebeneffekt im *linken* Operanden einer binären Operation auf den rechten Operanden auswirken:

Quellcode	Ausgabe

`class Prog {`
`  static void Main() {`
`  int ivar = 2;`
`  int erg = ivar++ * ivar;`
`  System.Console.WriteLine("{0} {1}", erg, ivar);`
`  }`
`}`
	`6 3`

Im folgenden Beispiel stehen *a*, *b* und *c* für beliebige numerische Operanden (z. B. `++ivar`). Für den Ausdruck

$$a + b * c$$

resultiert aus der Bindungskraftregel die folgende Zuordnung der Operanden:

$$a + (b * c)$$

Zusammen mit der Links-vor-rechts - Regel ergibt sich für die Auswertung der Operanden bzw. Ausführung der Operatoren die folgende Reihenfolge:

$$a, b, c, *, +$$

Wenn als Operanden numerische Literale oder Variablen auftreten, wird bei der „Auswertung“ eines Operanden lediglich sein Wert ermittelt, und die Reihenfolge der Operandenauswertungen ist belanglos. Im letzten Beispiel eine falsche Auswertungsreihenfolge zu unterstellen (z. B. *b*, *c*, *, *a*, +), bleibt ungestraft. Wenn Operanden *Nebeneffekte* enthalten (Zuweisungen, In- bzw. Dekrementoperationen oder Methodenaufrufe), dann ist die Reihenfolge der Operandenauswertungen jedoch relevant, und eine falsche Vermutung kann gravierende Fehler verursachen. Im folgenden Beispiel

Quellcode	Ausgabe

`class Prog {`
`  static void Main() {`
`  int ivar = 2;`
`  System.Console.WriteLine(ivar++ + ivar * 2);`
`  }`
`}`
	`8`

resultiert für den Ausdruck `ivar++ + ivar * 2` der Wert 8, denn:

- Zuerst wird der linke Operand der Addition ausgewertet:
  - Der Ausdruck `ivar++` hat den Wert 2.
  - Die Postinkrementoperation hat einen Nebeneffekt auf die Variable `ivar`.
- Dann wird der linke Operand der Multiplikation ausgewertet (Ergebnis: 3).
- Dann wird der rechte Operand der Multiplikation ausgewertet (Ergebnis: 2).

- Dann wird die Multiplikation ausgeführt (Ergebnis: 6).
- Dann wird die Addition ausgeführt (Ergebnis: 8).

Auch bei einem rechts-assoziativen Operator wird der linke Operand vor dem rechten ausgewertet, sodass im folgenden Beispiel mit der Variablen `alfa`

```
alfa += ++alfa
```

diese Auswertungs- bzw. Ausführungsreihenfolge resultiert:

```
alfa, ++alfa, +=
```

Als neuer Wert von `alfa` entsteht:

```
alfa + (alfa + 1)
```

Die oft anzutreffende Behauptung, Klammerausdrücke würden generell zuerst ausgewertet, ist falsch, wie das folgende Beispiel zeigt:

Quellcode	Ausgabe

`class Prog {`
`  static void Main() {`
`  int ivar = 2;`
`  int erg = ivar * (++ivar + 5);`
`  System.Console.WriteLine(erg);`
`  }`
`}`
	`16`

Die Auswertung des Ausdrucks `ivar * (++ivar + 5)` verläuft so:

- Wegen Regel 4 (links-vor-rechts bei der Auswertung der Operanden eines binären Operators) wird zuerst der linke Operand der Multiplikation ausgewertet (Ergebnis: 2)
- Dann wird der rechte Operand der Multiplikation ausgewertet (also der Klammerausdruck).
- Hier ist mit der Addition eine weitere binäre Operation vorhanden, und nach der Links-vor-rechts - Regel wird zunächst deren linker Operand ausgewertet (Ergebnis: 3, Nebeneffekt auf die Variable `ivar`). Dann wird der rechte Operand der Addition ausgewertet (Ergebnis: 5). Die Ausführung der Additionsoperation liefert für den Klammerausdruck den Wert 8.
- Schließlich führt die Multiplikation zum Endergebnis 16.

4.5.10.2 Operatorentabelle

In der folgenden Tabelle sind die bisher behandelten Operatoren mit absteigender Bindungskraft (Priorität) aufgelistet. Gruppen von Operatoren mit gleicher Bindungskraft sind durch eine horizontale Linie voneinander getrennt. In der **Operanden**-Spalte werden die zulässigen Datentypen der Argumentausdrücke mit Hilfe der folgenden Platzhalter beschrieben:

N	Ausdruck mit numerischem Datentyp (**sbyte**, **short**, **int**, **long**, **byte**, **ushort**, **uint**, **ulong**, **char**, **float**, **double**, **decimal**)
I	Ausdruck mit ganzzahligem (integralem) Datentyp (**sbyte**, **short**, **int**, **long**, **byte**, **ushort**, **uint**, **ulong**, **char**)
L	logischer Ausdruck (Typ **bool**)
K	Ausdruck mit kompatiblem Datentyp
S	**String** (Zeichenfolge)
V	Variable mit kompatiblem Datentyp
V_n	Variable mit kompatiblem, numerischem Datentyp (**sbyte**, **short**, **int**, **long**, **byte**, **ushort**, **uint**, **ulong**, **char**, **float**, **double**, **decimal**)

Operator	**Bedeutung**	**Operanden**
Methode(*Parameter*)	Methodenaufruf	
x++, *x*--	Postinkrement bzw. -dekrement	V_n
-*x*	Vorzeichenumkehr	*N*
!	Negation	*L*
++*x*, --*x*	Präinkrement bzw. -dekrement	V_n
(*Typ*)*x*	Typumwandlung	*K*
*, /	Multiplikation, Division	*N*, *N*
%	Modulo (Divisionsrest)	*N*, *N*
+, -	Addition, Subtraktion	*N*, *N*
+	String-Verkettung	*S*, *K* oder *K*, *S*
<<, >>	Links- bzw. Rechts-Shift	*I*, *I*
>, <, >=, <=	Vergleichsoperatoren	*N*, *N*
==, !=	Gleichheit, Ungleichheit	*K*, *K*
&	Bitweises UND	*I*, *I*
&	Logisches UND (mit unbedingter Auswertung)	*L*, *L*
^	Exklusives logisches ODER	*L*, *L*
|	Bitweises ODER	*I*, *I*
|	Logisches ODER (mit unbedingter Auswertung)	*L*, *L*
&&	Logisches UND (mit bedingter Auswertung)	*L*, *L*
||	Logisches ODER (mit bedingter Auswertung)	*L*, *L*
? :	Konditionaloperator	*L*, *K*, *K*
=	Wertzuweisung	*V*, *K*
+=, -=, *=, /=, %=	Wertzuweisung mit Aktualisierung	V_n, *N*

Im Anhang A finden Sie eine erweiterte Version dieser Tabelle, die zusätzlich alle Operatoren enthält, die im weiteren Verlauf des Kurses noch behandelt werden.

4.5.11 Übungsaufgaben zum Abschnitt 4.5

1) Welche Werte und Datentypen besitzen die folgenden Ausdrücke?

```
6/4*2.0
(int)6/4.0*3
(int)(6/4.0*3)
3*5+8/3%4*5
```

2) Welche Werte haben die Variablen `erg1` und `erg2` am Ende des folgenden Programms?

```
using System;
class Prog {
  static void Main() {
    int i = 2, j = 3, erg1, erg2;
    erg1 = (i++ == j ? 7 : 8) % 3;
    erg2 = (++i == j ? 7 : 8) % 2;
    Console.WriteLine("erg1 = {0}\nerg2 = {1}", erg1, erg2);
  }
}
```

3) Welche Wahrheitswerte erhalten im folgenden Programm die booleschen Variablen `la1` bis `la3`?

```
using System;
class Prog {
    static void Main() {
        bool la1;
        la1 = 3 > 2 && 2 == 2 ^ 1 == 1;
        Console.WriteLine(la1);

        bool la2;
        la2 = ((2 > 3) && (2 == 2)) ^ (1 == 1);
        Console.WriteLine(la2);

        bool la3;
        int i = 3;
        char c = 'n';
        la3 = !(i > 0 || c == 'j');
        Console.WriteLine(la3);
    }
}
```

Tipp: Die Negation von zusammengesetzten Ausdrücken ist etwas unangenehm. Mit Hilfe der Regeln von **De Morgan** kommt man zu äquivalenten Ausdrücken, die leichter zu interpretieren sind:

!(*la1* && *la2*) = !*la1* || !*la2*
!(*la1* || *la2*) = !*la1* && !*la2*

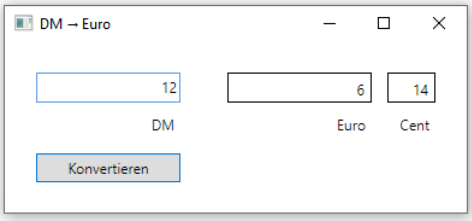
4) Erstellen Sie ein Programm, das den Exponentialfunktionswert e^x (mit e als der Eulerschen Zahl) zu einer vom Benutzer eingegebenen Zahl *x* bestimmt und ausgibt, z. B.:

Eingabe: `Argument: 1`
Ausgabe: `exp(1) = 2,71828182845905`

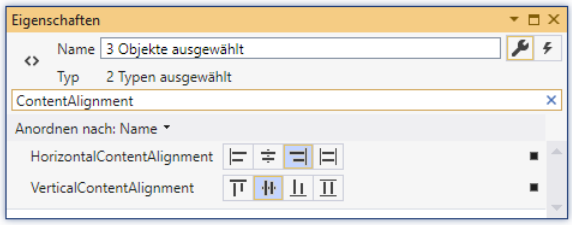
Hinweise:

- Suchen Sie mit Hilfe der BCL-Dokumentation zur Klasse **Math** im Namensraum **System** eine passende Methode.
- Verwenden Sie zum Einlesen des Arguments eine Variante der im Abschnitt 4.4 beschriebenen Technik, wobei die **Convert**-Methode **ToInt32()** geeignet zu ersetzen ist.

5) Entwickeln Sie den Währungskonverter aus den Abschnitten 3.3.4 und 4.3.11 so weiter, dass ganzzahlige, korrekt gerundete Werte für Euro und Cent erscheinen, z. B.:



Damit die Zahlen rechtsbündig und vertikal zentriert angezeigt werden, sollten die Eigenschaften **HorizontalContentAlignment** und **VerticalContentAlignment** angepasst werden, was für die drei betroffenen Steuerelemente gemeinsam geschehen kann:

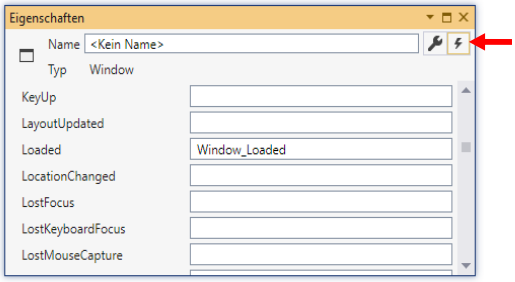


Es wäre nett, wenn nach dem Start unseres Programms das Textfeld für den DM-Betrag den Tastaturfokus hätte, sodass der Benutzer unmittelbar mit der Eingabe beginnen könnte, ohne zuvor den Tastaturfokus (z. B. per Maus) setzen zu müssen. Eine Lösungsmöglichkeit besteht darin, das zu privilegierende Steuerelement über die Methode **Focus()** der Klasse **UIElement** aufzufordern, den Fokus zu übernehmen, z. B.:

```
eingabe.Focus();
```

Damit diese Anweisung beim Laden des Anwendungsfensters ausgeführt wird, stecken wir sie in eine Ereignisbehandlungsmethode zum **Loaded**-Ereignis der Fensterklasse:

- Markieren Sie im WPF-Designer das Anwendungsfenster.
- Wechseln Sie im Eigenschaftenfenster zu den Ereignissen:

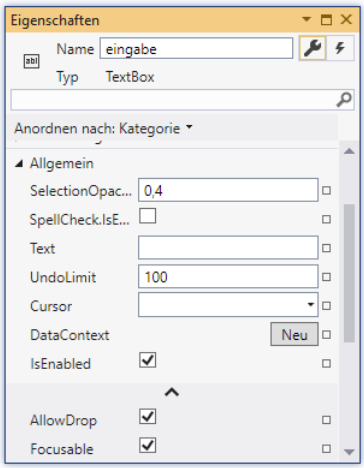


- Setzen Sie einen Doppelklick auf das Texteingabefeld zum Ereignis **Loaded**.
- Daraufhin wird in der Quellcodedatei **MainWindow.xaml.cs** die private Instanzmethode `Window_Loaded()` zu unserer Fensterklasse `MainWindow` angelegt:

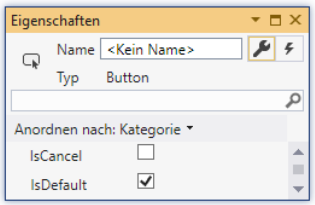
  ```
  private void Window_Loaded(object sender, RoutedEventArgs e) {

  }
  ```

  Diese Methode wird ausgeführt, wenn das Anwendungsfenster das Ereignis **Loaded** feuert.
- Ergänzen Sie im Rumpf dieser Methode den oben beschriebenen **Focus()** - Aufruf.

Außerdem muss die Eigenschaft **Focusable** des **TextBox**-Steuerelements den Wert **true** besitzen, was per Voreinstellung der Fall ist:



Wir haben (über den Wert **true** für die Eigenschaft **IsDefault**) bereits dafür gesorgt, dass sich das Klick-Ereignis des **Button**-Objekts per **Enter**-Taste auslösen lässt:



4.6 Über- und Unterlauf bei numerischen Datentypen

Wie Sie inzwischen wissen, haben die numerischen Datentypen jeweils einen bestimmten Wertebereich (siehe Tabelle im Abschnitt 4.3.4). Dank strenger Typisierung kann der Compiler verhindern, dass einer Variablen ein Ausdruck mit „zu großem Typ“ zugewiesen wird. So kann z. B. einer **int**-Variablen kein Wert vom Typ **long** zugewiesen werden. Bei der Auswertung eines Ausdrucks kann jedoch „unterwegs“ ein Wertebereichsproblem (z. B. ein Überlauf) auftreten. Im betroffenen Programm ist mit einem mehr oder weniger gravierenden Fehlverhalten zu rechnen, sodass Wertebereichsprobleme unbedingt vermieden bzw. rechtzeitig diagnostiziert werden müssen.

4.6.1 Überlauf bei Ganzzahltypen

Wird z. B. zu einer ganzzahligen Variablen, die bereits den maximalen Wert ihres Typs besitzt, eine positive Zahl addiert, kann das Ergebnis nicht mehr korrekt abgespeichert werden. Ohne besondere Vorkehrungen stellt ein C# - Programm im Fall eines solchen Ganzzahlüberlaufs keinesfalls seine Tätigkeit ein (z. B. mit einem Ausnahmefehler), sondern arbeitet munter weiter. Das folgende Programm

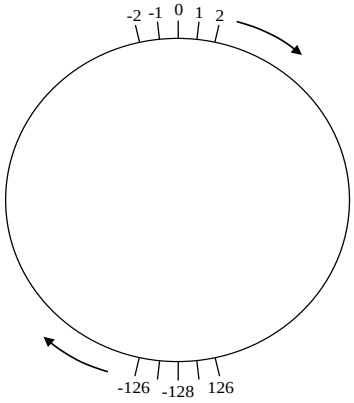
```
using System;
class Prog {
  static void Main() {
    int i = 2_147_483_647, j = 5, k;
    k = i + j;    // Überlauf!
    Console.WriteLine(i + " + " + j + " = " + k);
  }
}
```

liefert ohne jede Warnung das sinnlose Ergebnis:

```
2147483647 + 5 = -2147483644
```

Um das Auftreten eines negativen „Ergebniswerts“ zu verstehen, machen wir einen kurzen Ausflug in die Informatik. Die Werte der vorzeichenbehafteten Ganzzahltypen (mit positiven und negativen

Werten) sind nach dem **Zweierkomplementprinzip** auf einem Zahlenkreis angeordnet, und nach der größten positiven Zahl beginnt der Bereich der negativen Zahlen (mit abnehmendem Betrag), z. B. beim Typ **sbyte**:



Speziell bei der Steuerung von Raketenmotoren (vgl. Abschnitt 4.5.7) ist also Vorsicht geboten, weil ansonsten das Kommando „Mr. Spock, please push the engine.“ zum heftigen Rückwärtsschub führen könnte.[1] Es zeigt sich erneut, dass eine erfolgreiche Raketenforschung und -entwicklung ohne die sichere Beherrschung der elementaren Sprachelemente nicht möglich ist.

Natürlich kann nicht nur der positive Rand eines Ganzzahlwertebereichs überschritten werden, sondern auch der negative Rand, indem z. B. vom kleinstmöglichen Wert eine positive Zahl subtrahiert wird:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  public static void Main(String[] args) {`
`  int i = -2147483648, j = 5, k;`
`  k = i - j;`
`  Console.WriteLine(i + " - " + j + " = " + k);`
`  }`
`}`
	`-2147483648 - 5 = 2147483643`

Bei Wertebereichsproblemen durch eine betragsmäßig zu große Zahl wird im Manuskript generell von einem *Überlauf* gesprochen. Unter einem *Unterlauf* soll später das Verlassen eines Gleitkommawertebereichs in Richtung null durch eine betragsmäßig zu kleine Zahl verstanden werden (vgl. Abschnitt 4.6.4).

Oft lässt sich ein Überlauf durch die Wahl eines **geeigneten Datentyps** verhindern. Mit den Deklarationen

```
long i = 2_147_483_647, j = 5, k;
```

kommt es in der Anweisung

```
k = i + j;
```

nicht zum Überlauf, weil neben `i`, `j` und `k` nun auch der Ausdruck `i+j` den Typ **long** besitzt. Die Anweisung

```
Console.WriteLine(i + " + " + j + " = " + k);
```

liefert das korrekte Ergebnis:

```
2147483647 + 5 = 2147483652
```

[1] Mr. Spock arbeitete jahrelang als erster Offizier auf dem Raumschiff Enterprise.

Im Beispiel genügt es *nicht*, für die Zielvariable k den beschränkten Typ **int** durch **long** zu ersetzen, weil der Überlauf beim Berechnen des Ausdrucks („unterwegs“) auftritt. Mit den Deklarationen

```
int i = 2_147_483_647, j = 5;
long k;
```

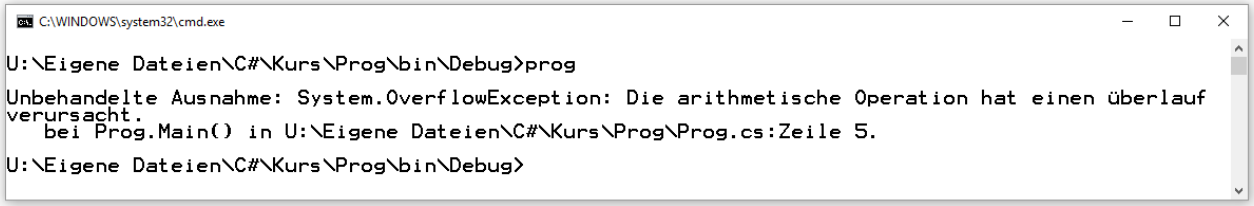
bleibt das Ergebnis falsch, denn …

- In der Anweisung
  ```
  k = i + j;
  ```
  wird zunächst der Ausdruck `i + j` berechnet.
- Weil beide Operanden vom Typ **int** sind, erhält auch der Ausdruck diesen Typ, und die Summe kann nicht korrekt berechnet bzw. zwischenspeichert werden.
- Schließlich wird der **long**-Variablen k das falsche Ergebnis zugewiesen.

In C# steht im Unterschied zu vielen anderen Programmiersprachen (z. B. Java) mit dem **checked**-Operator eine Möglichkeit bereit, den Überlauf bei Ganzzahlvariablen abzufangen. Eine gesicherte Variante des ursprünglichen Beispielprogramms

```
using System;
class Prog {
  static void Main() {
    int i = 2_147_483_647, j = 5, k;
    k = checked(i + j);
    Console.WriteLine(i + " + " + j + " = " + k);
  }
}
```

rechnet nach einem Überlauf nicht mit „Zufallszahlen“ weiter, sondern bricht mit einem Ausnahmefehler ab:[1]



Anstelle des **checked**-Operators bietet C# noch weitere Möglichkeiten, die Überlaufdiagnose für Ganzzahltypen einzuschalten:

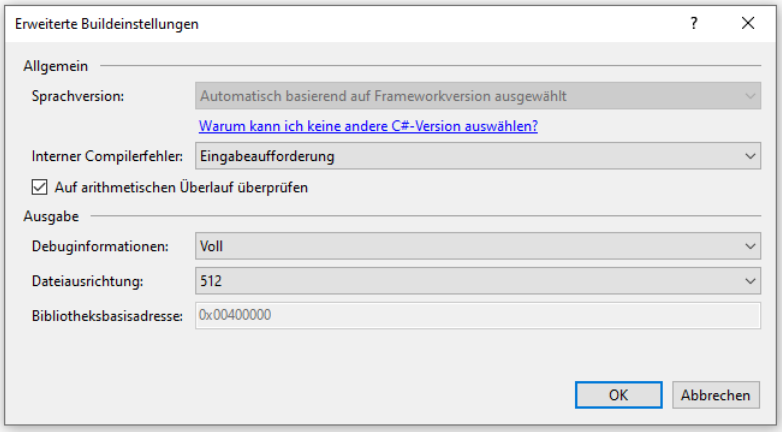
- **checked**-Anweisung
  Man kann die Überwachung für einen kompletten Anweisungsblock einschalten.
  Beispiel:
  ```
  checked {
    . . .
      k = i + j;
    . . .
  }
  ```

[1] Im Kapitel über Ausnahmebehandlung werden Sie lernen, Ausnahmefehler abzufangen, damit diese nicht mehr zum Abbruch des Programms führen.

- **checked** - Compiler-Option
  Man kann die Überwachung per Compiler-Option für das gesamte erstellte Assembly einschalten. Die Option wirkt sich auf alle Ganzzahlarithmetik-Operationen aus, die sich nicht im Gültigkeitsbereich eines **unchecked**-Operators (siehe unten) befinden, z. B.:

  ```
  >csc -checked Prog.cs
  ```

  Im Visual Studio vereinbart man diese Compiler-Option für ein Projekt folgendermaßen:
  - Menübefehl **Projekt > Eigenschaften**
  - Registerkarte **Build**
  - Schalter **Erweitert**
  - Kontrollkästchen **Auf arithmetischen Überlauf überprüfen**



Als Argument gegen die naheliegende Entscheidung, die Überlaufdiagnose für Ganzzahltypen *generell* einzuschalten, kommt nur der zeitliche Mehraufwand in Betracht. Bei der Aufgabe, mit 0 beginnend 2147483647 mal den **int**-Wert 1 zu addieren,

$$\sum_{i=1}^{2147483647} 1$$

ergaben sich in der *Release*-Konfiguration des Projekts basierend auf jeweils 50 Tests mit schwankenden Einzelergebnissen die folgenden mittleren Laufzeiten in Millisekunden:[1]

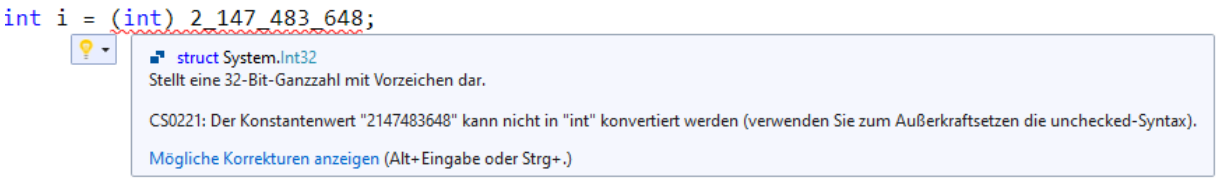
Statistiken

		Checked	Unchecked
N	Gültig	50	50
	Fehlend	0	0
Mittelwert		2206,0800	1470,6200
Median		2181,5000	1464,0000

Der zeitliche Mehraufwand aufgrund der Überlaufdiagnose beträgt immerhin ca. 50%, sodass man *nicht* empfehlen kann, die Überlaufkontrolle für Ganzzahloperationen generell einzuschalten.

Der Vollständigkeit halber soll an dieser Stelle noch der **unchecked**-Operator erwähnt werden, mit dem sich eine per Compiler-Option aktivierte Überlaufdiagnose und andere Kontrollfunktionen des Compilers abschalten lassen. Auch ohne **checked**-Instruktion verhindert der Compiler z. B., dass ein Ganzzahlliteral mit Typ **long** in den Typ **int** gewandelt wird. Das Visual Studio verrät, wie man diese Sperre umgehen kann:

[1] Berechnet mit IBM SPSS Statistics 27



Wird der „Tipp“ umgesetzt, resultiert ein fehlerhafter Wert:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  int i = unchecked((int) 2_147_483_648);`
`  Console.WriteLine(i);`
`  }`
`}`
	`-2147483648`

Wenn der **long**-Wertebereich für eine Aufgabenstellung nicht reicht, müssen wir uns trotzdem nicht auf die Problemdiagnose per Ausnahmefehler beschränken, sondern haben mit der Struktur **BigInteger** aus dem Namensraum **System.Numerics** noch einen Datentyp für beliebig große ganze Zahlen zur Verfügung (zu Strukturen siehe Abschnitt 6.1). Dass eine Variable vom Typ **BigInteger** erheblich mehr Speicher- und Rechenzeitaufwand verursacht als eine **int**-Variable, würde nur beim massenhaften Auftreten von **BigInteger**-Variablen ins Gewicht fallen. Wie das folgende Beispiel zeigt, erzielt man mit **BigInteger** noch sinnvolle Ergebnisse, wenn es bei den elementaren Ganzzahltypen zum Überlauf kommt:

Quellcode	Ausgabe

`using System;`
`using System.Numerics;`
 	
`class Prog {`
`  static void Main() {`
`  int i = 2_147_483_647, j = 5, k;`
`  k = i + j;`
`  Console.WriteLine("Rechnung mit int:");`
`  Console.WriteLine($"{i} + {j} = {i+j}");`
 	
`  BigInteger bigInt = i;`
`  bigInt += j;`
`  Console.WriteLine("\nRechnung mit BigInteger:");`
`  Console.WriteLine($"{i} + {j} = {bigInt}");`
`  }`
`}`
	`Rechnung mit int:`
	`2147483647 + 5 = -2147483644`
	
	`Rechnung mit BigInteger:`
	`2147483647 + 5 = 2147483652`

4.6.2 Unendliche und undefinierte Werte bei den Typen float und double

Auch bei den binären Gleitkommatypen **float** und **double** kann ein Überlauf auftreten, obwohl die unterstützten Wertebereiche hier erheblich größer sind. Dabei kommt es aber weder zu einem sinnlosen Zufallswert noch zu einem Ausnahmefehler, sondern zu den speziellen Gleitkommawerten +/- **Unendlich**, mit denen anschließend sogar weitergerechnet werden kann. Das folgende Programm:

```
using System;
class Prog {
  static void Main() {
    double bigd = Double.MaxValue;
    Console.WriteLine("Double.MaxValue =\t" + bigd);
    bigd *= 10.0;
    Console.WriteLine("Double.MaxValue * 10 =\t" + bigd);
    Console.WriteLine("Unendl. + 10 =\t\t" + (bigd + 10));
    Console.WriteLine("Unendl. * -13 =\t\t" + (bigd * -13));
    Console.WriteLine("13.0/0.0 =\t\t" + (13.0 / 0.0));
  }
}
```

liefert im .NET Framework 3.5 die Konsolenausgabe:

```
Double.MaxValue =       1,79769313486232E+308
Double.MaxValue * 10 =  +unendlich
Unendl. + 10 =          +unendlich
Unendl. * -13 =         -unendlich
13.0/0.0 =              +unendlich
```

Ab .NET Framework 4.0 und in .NET 5.0 sieht die Konsolenausgabe desselben Programms verblüffend anders aus:

```
Double.MaxValue =       1,79769313486232E+308
Double.MaxValue * 10 =  8
Unendl. + 10 =          8
Unendl. * -13 =         -8
13.0/0.0 =              8
```

Offenbar hatte jemand die kreative Idee, das in der Mathematik übliche Symbol ∞ für Unendlich durch die Ziffer 8 darzustellen.

Mit Hilfe der Unendlich-Werte „gelingt“ bei der Gleitkommaarithmetik sogar die Division durch null, während bei der Ganzzahlarithmetik ein solcher Versuch zu einem Laufzeitfehler führt.[1]

Bei den folgenden „Berechnungen“

$$\text{Unendlich} - \text{Unendlich}$$

$$\frac{\text{Unendlich}}{\text{Unendlich}}$$

$$\text{Unendlich} \cdot 0$$

$$\frac{0}{0}$$

resultiert der spezielle Gleitkommawert **NaN** (*Not a Number*), wie das folgende Programm zeigt:

[1] Es wird dann ein Objekt der Klasse **DivideByZeroException** „geworfen“. Mit der Ausnahmebehandlung werden wir uns im Kapitel 13 beschäftigen.

```
using System;
class Prog {
  static void Main() {
    double bigd = Double.MaxValue * 10.0;
    Console.WriteLine("Unendlich – Unendlich =\t" + (bigd - bigd));
    Console.WriteLine("Unendlich / Unendlich =\t" + (bigd / bigd));
    Console.WriteLine("Unendlich * 0.0 =\t" + (bigd * 0.0));
    Console.WriteLine("0.0 / 0.0 =\t\t" + (0.0 / 0.0));
  }
}
```

Es liefert im .NET-Framework 3.5 die Konsolenausgabe:

```
Unendlich - Unendlich = n. def.
Unendlich / Unendlich = n. def.
Unendlich * 0.0 =       n. def.
0.0 / 0.0 =             n. def.
```

Ab .NET Framework 4.0 und in .NET 5.0 sieht die Konsolenausgabe desselben Programms durch die Verwendung der üblichen Bezeichnung *NaN* sinnvoller aus:

```
Unendlich - Unendlich = NaN
Unendlich / Unendlich = NaN
Unendlich * 0.0 =       NaN
0.0 / 0.0 =             NaN
```

Zu den letzten Beispielprogrammen ist noch anzumerken, dass man über das öffentliche Feld **MaxValue** der Struktur[1] **Double** aus dem Namensraum **System** den größten Wert in Erfahrung bringt, der in einer **double**-Variablen gespeichert werden kann.[2]

Über die statischen **Double**-Methoden

- **public static bool IsPositiveInfinity(double** *d***)**
- **public static bool IsNegativeInfinity(double** *d***)**
- **public static bool IsNaN(double** *d***)**

mit einem Parameter vom Typ **double** und einer Rückgabe vom Typ **bool** lässt sich für einen Ausdruck vom Typ **double** prüfen, ob er einen unendlichen oder undefinierten Wert besitzt, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  double d = 0.0 / 0.0;`
`  Console.WriteLine(Double.IsNaN(d));`
`  }`
`}`
	`True`

Eben wurde eine Technik zur Beschreibung von *vorhandenen Bibliotheksmethoden* im Manuskript erstmals benutzt, die auch in der BCL-Dokumentation in ähnlicher Form Verwendung findet, z. B.:

---

[1] Bei den später noch ausführlich zu behandelnden *Strukturen* handelt es sich um Werttypen mit starker Verwandtschaft zu den Klassen. Insbesondere wird sich zeigen, dass die elementaren Datentypen (z. B. **double**) auf Strukturtypen aus dem Namensraum **System** abgebildet werden (z. B. **Double**).

[2] Die öffentliche **MaxValue**-Verfügbarkeit stellt übrigens *keinen* Verstoß gegen das Datenkapselungs-Prinzip der objektorientierten Programmierung dar, weil das Feld als *konstant* deklariert ist (vgl. Abschnitt 4.3.9):

```
public const double MaxValue = 1.7976931348623157E+308;
```

C# Kopieren

```
public static bool IsPositiveInfinity (double d);
```

Dabei wird die Benutzung einer Methode erläutert durch Angabe von:

- Modifikatoren (z. B. für den Zugriffsschutz)
- Rückgabetyp
- Methodenname
- Parameterliste (mit Angabe der Parametertypen)

Für besonders neugierige Leser sollen abschließend noch die **float**-Darstellungen der speziellen Gleitkommawerte angegeben werden (vgl. Abschnitt 4.3.5.1):

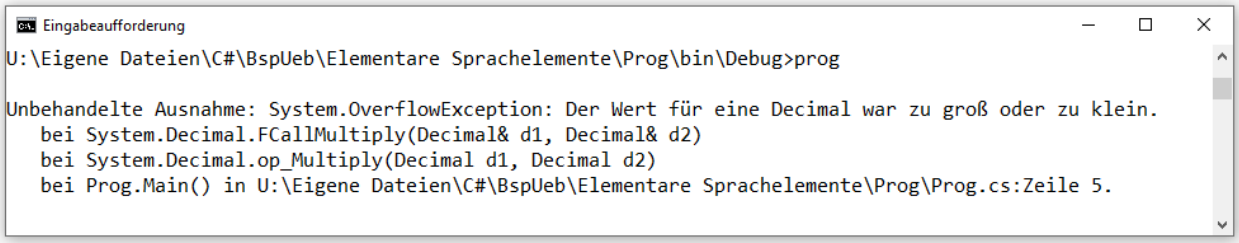
Wert	float-Darstellung		
	Vorz.	Exponent	Mantisse
+unendlich	0	11111111	00000000000000000000000
-unendlich	1	11111111	00000000000000000000000
NaN	0	11111111	10000000000000000000000

4.6.3 Überlauf beim Typ decimal

Beim Typ **decimal** wird im Fall eines Überlaufs *nicht* mit dem speziellen Wert Unendlich weitergearbeitet. Stattdessen wird ein Ausnahmefehler gemeldet, der unbehandelt zur Beendigung des Programms führt. Im Unterschied zu den Ganzzahltypen (vgl. Abschnitt 4.6.1) muss die Überlaufdiagnose *nicht* per **checked**-Operator, -Anweisung oder -Compiler-Option angeordnet werden. Das Beispielprogramm

```
using System;
class Prog {
  static void Main() {
    decimal d = Decimal.MaxValue;
    d = d * 10;
    Console.WriteLine(d);
  }
}
```

wird (bei Ausführung im .NET - Framework 4.8) mit der folgenden Meldung abgebrochen:



4.6.4 Unterlauf bei den Gleitkommatypen

Bei den binären Gleitkommatypen **float**, **double** und **decimal** ist auch ein Unterlauf möglich, wobei eine Zahl mit einem sehr kleinen Betrag (bei **double**: $< 4{,}94065645841247 \cdot 10^{-324}$) nicht mehr dargestellt werden kann. In diesem Fall rechnet ein C# - Programm mit dem Wert 0,0 weiter, was in der Regel akzeptabel ist, z. B.:

Quellcode	Ausgabe .NET Framework 4.8	Ausgabe .NET 5.0

`using System;`
`class Prog {`
`  static void Main() {`
`  double smalld = Double.Epsilon;`
`  Console.WriteLine(smalld);`
`  smalld /= 10.0;`
`  Console.WriteLine(smalld);`
`  }`
`}`
	`4,94065645841247E-324`
	`0`
		`5E-324`
		`0`

Das öffentliche Feld **Double.Epsilon** enthält den kleinsten Betrag, der in einer **double**-Variablen gespeichert werden kann (vgl. Abschnitt 4.3.5.1 zu denormalisierten Werten bei den binären Gleitkommatypen **float** und **double**).

Kommt es bei der Berechnung eines Ausdrucks *unterwegs* zu einem Unterlauf, ist allerdings ein falsches Endergebnis zu erwarten. Das folgende Programm kommt bei der Rechnung

$$10^{-323} * 10^{308} * 10^{16}$$

dem korrekten Ergebnis 10 recht nahe. Wird aber der Gesamtfaktor Eins (1,0 = 0,1·10,0) unglücklich in den Rechenweg eingebaut, führt ein irreversibler Unterlauf zum falschen Ergebnis 0:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  double a = 1e-323;`
`  double b = 1e308;`
`  double c = 1e16;`
`  Console.WriteLine(a * b * c);`
`  Console.WriteLine(a * 0.1 * b * 10.0 * c);`
`  }`
`}`
	`9,88131291682493`
	`0`

Im folgenden Beispielprogramm führt der Versuch, den kleinsten positiven **decimal**-Wert (10^{-28}) zu halbieren, zum Ergebnis 0:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  decimal smallDec = 1e-28m;`
`  Console.WriteLine(smallDec);`
`  smallDec /= 2m;`
`  Console.WriteLine(smallDec);`
`  }`
`}`
	`0,0000000000000000000000000001`
	`0`

4.7 Anweisungen (zur Ablaufsteuerung)

Wir haben uns im Kapitel 4 zunächst mit (lokalen) **Variablen** und elementaren **Datentypen** vertraut gemacht. Dann haben wir gelernt, aus Variablen, Literalen und Methodenaufrufen mit Hilfe von **Operatoren** mehr oder weniger komplexe **Ausdrücke** zu bilden. Diese wurden meist mit der **Console**-Methode **WriteLine()** auf dem Bildschirm ausgegeben oder in Wertzuweisungen verwendet.

In den meisten Beispielprogrammen traten nur wenige Sorten von Anweisungen auf (Variablendeklarationen, Wertzuweisungen und Methodenaufrufe). Nun werden wir uns systematisch mit dem

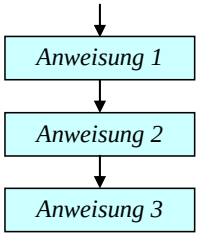
allgemeinen Begriff einer C# - Anweisung befassen und vor allem die wichtigen Anweisungen zur Ablaufsteuerung (Verzweigungen und Schleifen) kennenlernen.

4.7.1 Überblick

Ausführbare Programmteile, die in C# nach unserem bisherigen Kenntnisstand als Methoden oder Eigenschaften von Klassen zu realisieren sind, bestehen aus Anweisungen (engl. *statements*).

Am Ende des Abschnitts 4.7 werden Sie die folgenden Sorten von Anweisungen kennen:

- **Deklarationsanweisung für lokale Variablen**
  Die Deklarationsanweisung für lokale Variablen wurde schon im Abschnitt 4.3.6 eingeführt.
  Beispiel: `int i = 1, k;`
- **Ausdrucksanweisungen**
  Folgende Ausdrücke werden zu Anweisungen, sobald man ein Semikolon dahinter setzt:
  - **Wertzuweisung** (vgl. Abschnitte 4.3.6 und 4.5.8)
    Beispiel: `k = i + j;`
  - **Prä- bzw. Postinkrement- oder -dekrementoperation**
    Beispiel: `i++;`
    Im Beispiel ist nur der „Nebeneffekt“ des Ausdrucks `i++` von Bedeutung (vgl. Abschnitt 4.5.1). Sein Wert bleibt ungenutzt.
  - **Methodenaufruf**
    Beispiel: `Console.WriteLine(cond);`
    Besitzt die im Rahmen einer eigenständigen Anweisung aufgerufene Methode einen Rückgabewert, wird dieser ignoriert.
- **Leere Anweisung**
  Beispiel: `;`
  Die durch ein einsames (nicht anderweitig eingebundenes) Semikolon ausgedrückte *leere* Anweisung hat keinerlei Effekte und kommt gelegentlich zum Einsatz, wenn syntaktisch eine Anweisung erforderlich ist, aber nichts geschehen soll.
- **Blockanweisung**
  Eine Folge von Anweisungen, die durch ein Paar geschweifter Klammern zusammengefasst bzw. abgegrenzt werden, bildet eine **Verbund-** bzw. **Blockanweisung.** Wir haben uns bereits im Abschnitt 4.3.8 im Zusammenhang mit dem Sichtbarkeitsbereich von lokalen Variablen mit Anweisungsblöcken beschäftigt. Wie gleich näher erläutert wird, fasst man z. B. *dann* mehrere Abweisungen zu einem Block zusammen, wenn diese Anweisungen unter einer gemeinsamen Bedingung ausgeführt werden sollen. Es wäre sehr unpraktisch, dieselbe Bedingung für jede betroffene Anweisung wiederholen zu müssen.
- **Anweisungen zur Ablaufsteuerung**
  Die Methoden der bisherigen Beispielprogramme im Kapitel 4 bestanden meist aus einer *Sequenz* von Anweisungen, die bei jedem Programmeinsatz komplett durchlaufen wurde:



  Oft möchte man jedoch ...

- die Ausführung einer Anweisung (bzw. eines Anweisungsblocks) von einer *Bedingung* abhängig machen
- oder eine Anweisung (bzw. einen Anweisungsblock) *wiederholt* ausführen lassen.

Für solche Zwecke enthält C# etliche Anweisungen zur Ablaufsteuerung, die bald ausführlich behandelt werden (**bedingte Anweisung**, **Fallunterscheidung**, **Schleifen**).

Blockanweisungen sowie Anweisungen zur Ablaufsteuerung enthalten andere Anweisungen und werden daher auch als **zusammengesetzte Anweisungen** bezeichnet.

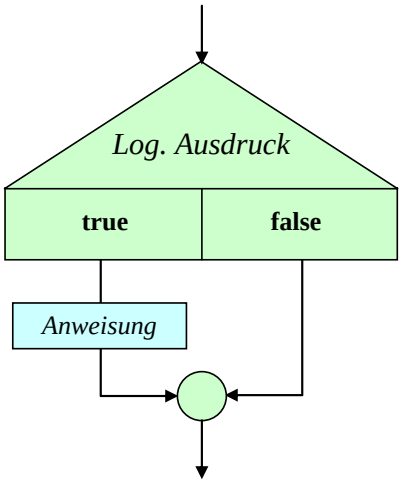
Anweisungen werden durch ein **Semikolon** abgeschlossen, sofern sie nicht mit einer schließenden Blockklammer enden.

4.7.2 Bedingte Anweisung und Fallunterscheidung

Oft ist es erforderlich, dass eine Anweisung nur unter einer bestimmten Bedingung ausgeführt wird. Etwas allgemeiner formuliert geht es darum, dass viele Algorithmen *Fallunterscheidungen* benötigen, also an bestimmten Stellen in Abhängigkeit vom Wert eines steuernden Ausdrucks in unterschiedliche Pfade verzweigen müssen.

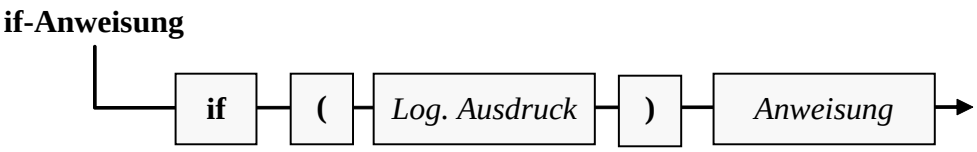
4.7.2.1 if-Anweisung

Nach dem folgenden **Programmablaufplan** (**PAP**) bzw. **Flussdiagramm** soll eine Anweisung nur dann ausgeführt werden, wenn ein logischer Ausdruck den Wert **true** besitzt:



Wir werden diese Darstellungstechnik ab jetzt verwenden, um einen Algorithmus oder einen Programmablauf zu beschreiben. Die verwendeten Symbole sind hoffentlich anschaulich, entsprechen aber keiner strengen Normierung.

Während der Programmablaufplan den Zweck (die Semantik) eines Sprachbestandteils erläutert, beschreibt das vertraute Syntaxdiagramm, wie zulässige Exemplare des Sprachbestandteils zu bilden sind. Das folgende Syntaxdiagramm beschreibt die zur Realisation einer bedingten Ausführung dienende **if**-Anweisung:



Die eingebettete (bedingt auszuführende) Anweisung darf keine Variablendeklaration (im Sinn von Abschnitt 4.3.6) sein. Ein Block ist aber selbstverständlich erlaubt, und darin dürfen auch lokale Variablen definiert werden.

Es ist übrigens nicht vergessen worden, ein Semikolon ans Ende des **if**-Syntaxdiagramms zu setzen. Dort wird eine eingebettete Anweisung verlangt, wobei konkrete Beispiele oft mit einem Semikolon enden, manchmal aber auch mit einer schließenden geschweiften Klammer.

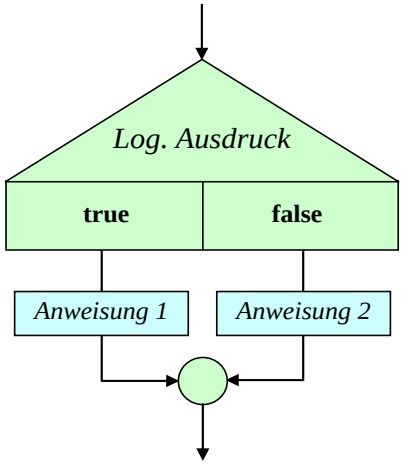
Im folgenden Beispiel wird eine Meldung ausgegeben, wenn die Variable `anz` einen Wert kleiner oder gleich Null besitzt:

```
if (anz <= 0)
  Console.WriteLine("Die Anzahl muss > 0 sein!");
```

Der Zeilenumbruch zwischen dem logischen Ausdruck und der eingebetteten Anweisung dient nur der Übersichtlichkeit und ist für den Compiler irrelevant.

4.7.2.2 if-else - Anweisung

Soll auch etwas passieren, wenn der steuernde logische Ausdruck den Wert **false** besitzt,



dann erweitert man die **if**-Anweisung um eine **else**-Klausel.

Zur Beschreibung der **if-else** - Anweisung wird an Stelle eines Syntaxdiagramms eine alternative Darstellungsform gewählt, die sich am typischen C# - Quellcode-Layout orientiert:

if (*Logischer Ausdruck***)**
  Anweisung 1
else
  Anweisung 2

Wie bei den Syntaxdiagrammen gilt auch für diese Form der Syntaxbeschreibung:

- Für **terminale Sprachbestandteile**, die exakt in der angegebenen Form in konkreten Quellcode zu übernehmen sind, wird **fette** Schrift verwendet.
- *Platzhalter* sind durch *kursive* Schrift gekennzeichnet.

Während die Syntaxbeschreibung im Quellcode-Layout relativ einfache Bildungsregeln (mit einer einzigen zulässigen Sequenz) sehr anschaulich beschreibt, kann das manchmal weniger anschauliche Syntaxdiagramm bei einer komplizierten und variantenreichen Syntax alle zulässigen Sequenzen kompakt und präzise dokumentieren.

Bei den eingebetteten Anweisungen (*Anweisung 1* bzw. *Anweisung 2*) darf es sich nicht um Variablendeklarationen (im Sinn von Abschnitt 4.3.6) handeln. Wird ein *Block* als eingebettete Anweisung verwendet, sind darin aber auch Variablendeklarationen erlaubt.

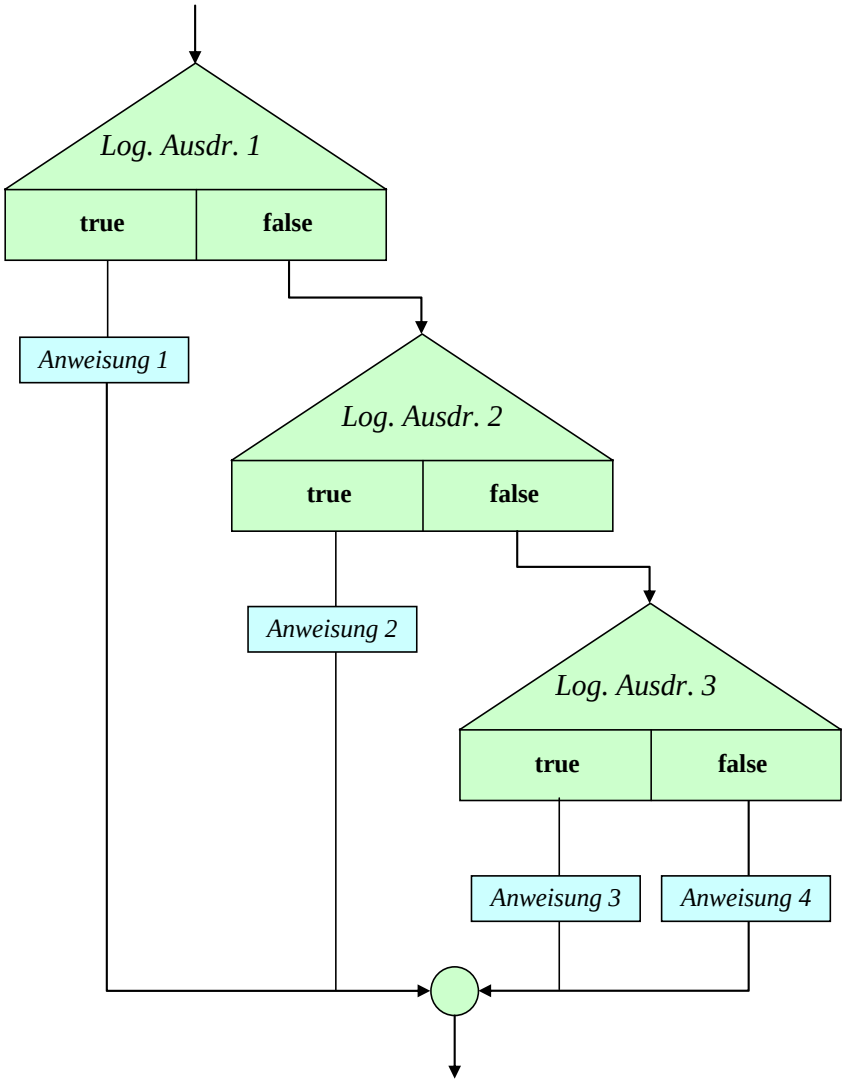
Im folgenden **if-else** - Beispiel wird der natürliche Logarithmus zu einer Zahl geliefert, falls diese positiv ist. Anderenfalls erscheint eine Fehlermeldung. Das Argument wird vom Benutzer über eine **ToDouble() - ReadLine()** - Konstruktion erfragt (vgl. Abschnitt 4.4.1).

Quellcode	Ausgabe (Eingaben **fett**)

```
using System;
class Prog {
  static void Main() {
    Console.Write("Argument: ");
    double arg = Convert.ToDouble(Console.ReadLine());
    if (arg > 0)
      Console.WriteLine("ln(" + arg + ") = "
                        + Math.Log(arg));
    else
      Console.WriteLine("Argument <= 0");
  }
}
```

```
Argument: 2
ln(2) = 0,693147180559945
```

Eine bedingt auszuführende Anweisung darf durchaus wiederum vom **if**- bzw. **if-else** - Typ sein, sodass sich mehrere, *hierarchisch geschachtelte* Fälle unterscheiden lassen. Den folgenden Programmablauf mit „sukzessiver Restaufspaltung“



realisiert z. B. eine **if-else** - Konstruktion nach diesem Muster:

if (*Logischer Ausdruck 1***)**
  Anweisung 1
else if (*Logischer Ausdruck 2***)**
  Anweisung 2
  . . .
  . . .
else if (*Logischer Ausdruck k***)**
  Anweisung k
else
  Default-Anweisung

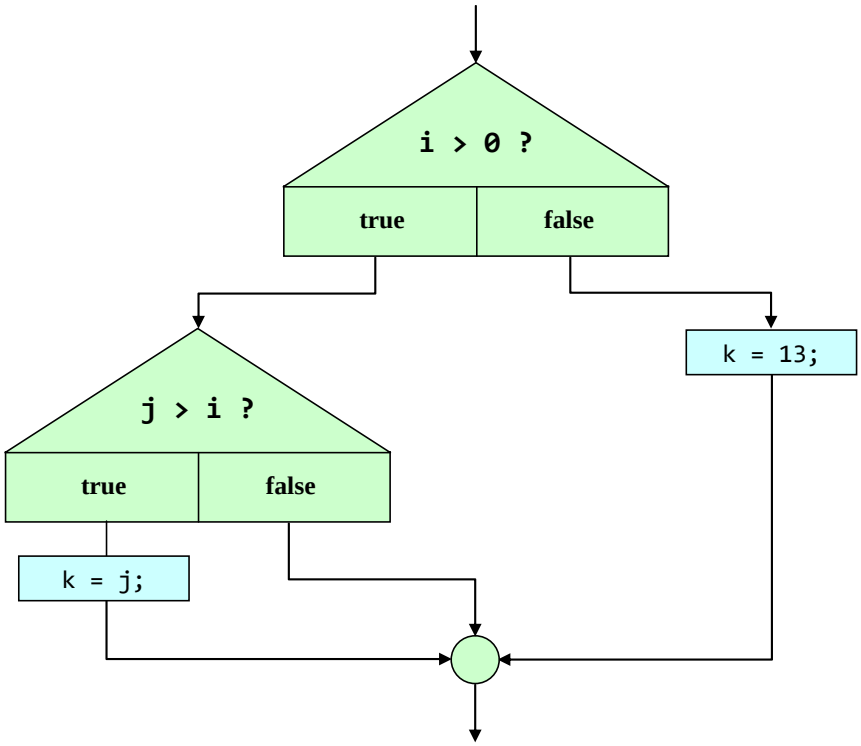
Wenn alle logischen Ausdrücke den Wert **false** annehmen, dann wird die **else**-Klausel zur letzten **if**-Anweisung ausgeführt.

Bei einer Mehrfallunterscheidung ist die im Abschnitt 4.7.2.3 vorzustellende **switch**-Anweisung gegenüber einer verschachtelten **if-else** - Konstruktion zu bevorzugen, wenn die Fallzuordnung über die verschiedenen Werte *eines* Ausdrucks (z. B. vom Typ **int**) erfolgen kann.

Beim Schachteln von bedingten Anweisungen kann es zum sogenannten **dangling-else** - Problem[1] kommen, wobei ein Missverständnis zwischen Compiler und Programmierer hinsichtlich der Zuordnung einer **else**-Klausel besteht. Im folgenden Codefragment

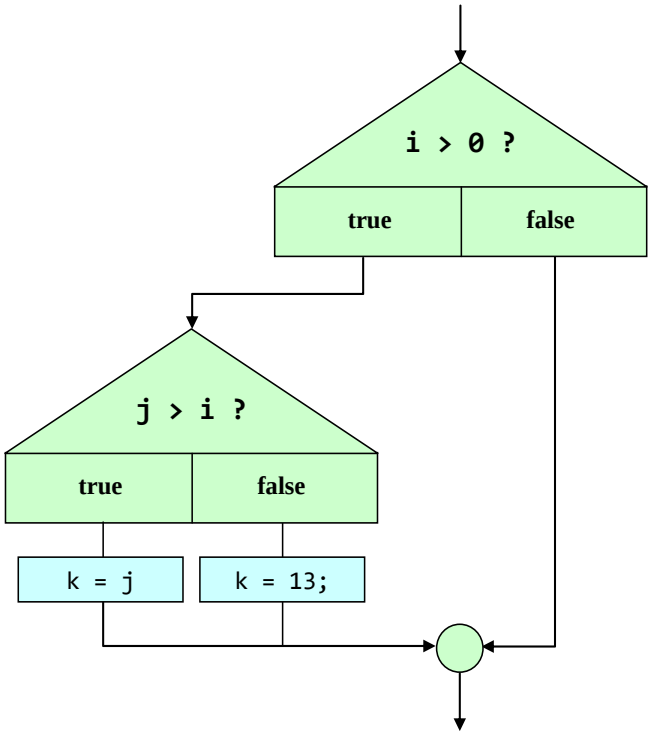
```
if (i > 0)
    if (j > i)
        k = j;
else
    k = 13;
```

lassen die Einrücktiefen vermuten, dass der Programmierer die **else**-Klausel auf die *erste* **if**-Anweisung bezogen zu haben glaubt:



[1] Deutsche Übersetzung von *dangling*: *baumelnd*.

Der Compiler ordnet eine **else**-Klausel jedoch dem in Aufwärtsrichtung nächstgelegenen **if** zu, das nicht durch Blockklammern abgeschottet ist und noch keine **else**-Klausel besitzt. Im Beispiel bezieht er die **else**-Klausel also auf die *zweite* **if**-Anweisung, sodass de facto der folgende Programmablauf resultiert:



Bei `i` ≤ 0 geht der Programmierer vom neuen `k`-Wert 13 aus, der beim tatsächlichen Programmablauf jedoch *nicht* unbedingt zu erwarten ist.

Mit Hilfe von Blockklammern kann die gewünschte Zuordnung erzwungen werden:

```
if (i > 0)
    {if (j > i)
        k = j; }
else
    k = 13;
```

Alternativ kann man dem zweiten **if** eine **else**-Klausel spendieren und dabei eine leere Anweisung verwenden:

```
if (i > 0)
    if (j > i)
        k = j;
    else
        ;
else
    k = 13;
```

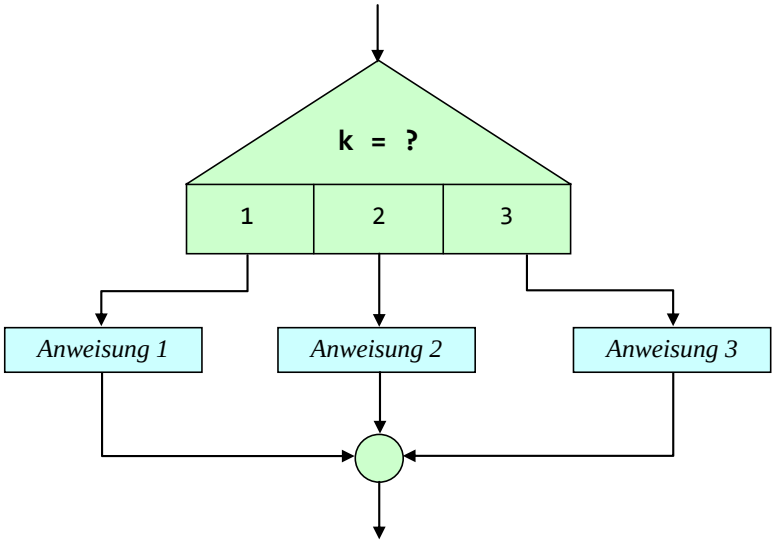
Gelegentlich kommt als Alternative zu einer simplen **if-else** - Anweisung, die zur Berechnung eines Wertes bedingungsabhängig zwei unterschiedliche Ausdrücke benutzt, der Konditionaloperator (vgl. Abschnitt 4.5.9) in Frage, z. B.:

if-else - Anweisung	Konditionaloperator

`double arg = 3.0, d;`
`if (arg > 1)`
`  d = arg * arg;`
`else`
`  d = arg;`
	`double arg = 3.0, d;`
	`d = arg > 1 ? arg * arg : arg;`

4.7.2.3 switch-Anweisung

Wenn eine Fallunterscheidung mit mehr als zwei Alternativen in Abhängigkeit vom Wert *eines* Ausdrucks vorgenommen werden soll,



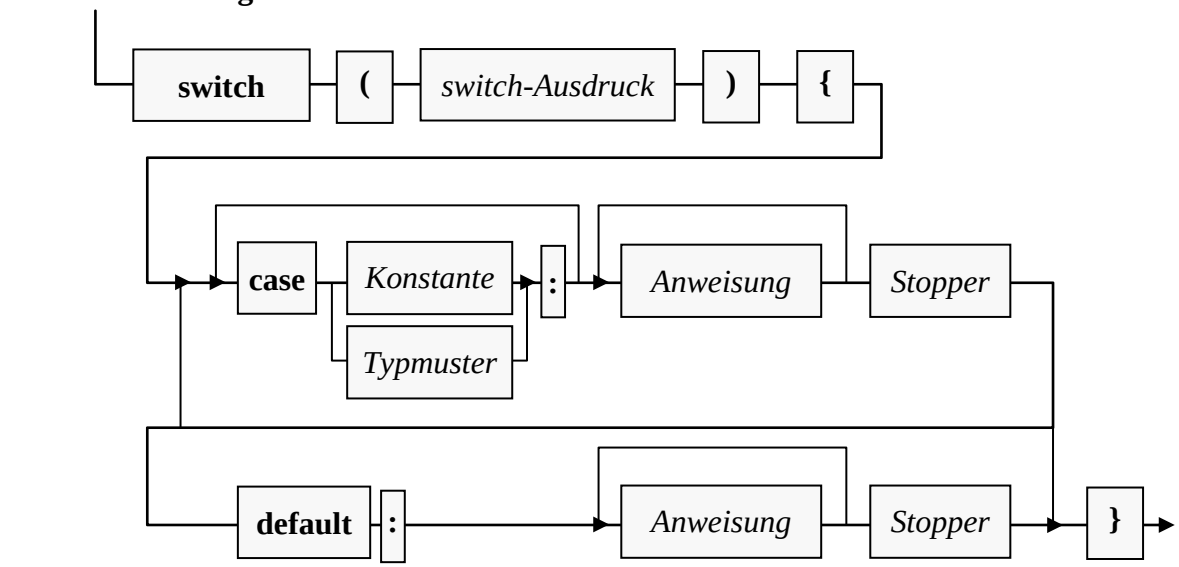
dann ist eine **switch**-Anweisung weitaus handlicher als eine verschachtelte **if**-**else** - Konstruktion.

Als Datentyp des steuernden Ausdrucks sind erlaubt:

- In C# 6.0:
  - Ganzzahlige (integrale) numerische Datentypen (**sbyte**, **short**, **int**, **long**, **byte**, **ushort**, **uint**, **ulong**, **char**). Dazu gehört auch der Datentyp **char** (vgl. Abschnitt 4.3.4).
  - **bool**
  - Aufzählungstypen (siehe unten)
  - Zeichenfolgen (Datentyp **string**)
- Seit C# 7.0 ist ein beliebiger Ausdruck erlaubt.

Der Genauigkeit halber wird die **switch**-Anweisung mit einem Syntaxdiagramm beschrieben. Wer die Syntaxbeschreibung im Quellcode-Layout bevorzugt, kann ersatzweise einen Blick auf die gleich folgenden Beispiele werfen.

switch-Anweisung



4.7.2.3.1 Erläuterung der Grundlogik unter Aussparung der Erweiterungen ab C# 7.0

Zur Erläuterung der **switch**-Grundlogik ignorieren wir aus didaktischen Gründen zunächst die im Abschnitt 4.7.2.3.3 beschriebenen **switch**-Erweiterungen, die C# ab Version 7.0 anbietet (beliebiger **switch**-Ausdruck, **case**-Deklaration mit Typmuster).

In den **case**-Deklarationen werden (auf dem Sprachniveau von C# 6.0) nur *konstante* Ausdrücke verwendet, deren Ergebnis schon der Compiler ermitteln kann (z. B. Literale, Konstanten oder mit konstanten Argumenten gebildete Ausdrücke).

Stimmt bei der Ausführung einer Methode der Wert des **switch**-Ausdrucks mit einem **case**-Wert überein, dann werden die zugehörigen Anweisungen ausgeführt, ansonsten (falls vorhanden) die **default**-Anweisungen.

Sollen für mehrere Werte des **switch**-Ausdrucks dieselben Anweisungen ausgeführt werden, dann setzt man die zugehörigen **case**-Deklarationen hintereinander und lässt die Anweisungen auf die letzte Deklaration folgen. Wer in einem **case** eine *Serie* von Fällen durch Angabe der Randwerte (z. B. von *a* bis *z*) behandeln möchte, wird sich über die ab C# 7.0 verfügbaren **switch**-Erweiterungen freuen (siehe Abschnitt 4.7.2.3.3).

Jeder Fall *muss* mit einem **Stopper** abgeschlossen werden, auch der terminale **default** - Fall:

```
        default:
            Console.WriteLine("Restkategorie");
    }
```

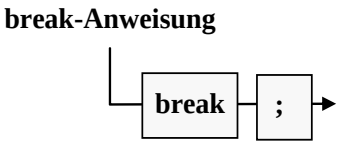
CS8070: Die Steuerung kann nicht von der abschließenden case-Bezeichnung ("default:") aus dem switch-Ausdruck übergeben werden.

Der aus anderen Programmiersprachen (z. B. C, C++, Java) bekannte „Durchfall“ zu den Anweisungen „tieferer“ Fälle ist verboten, womit viele Programmierfehler vermieden werden.

Als Stopper sind erlaubt:

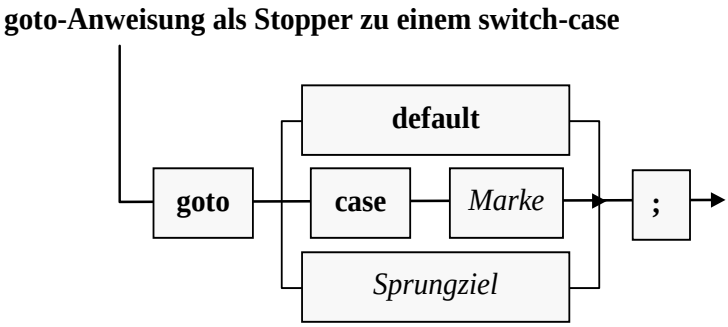
- **break**-Anweisung
- **goto**-Anweisung
- **return**-Anweisung

Meist stoppt man mit der **break**-Anweisung,



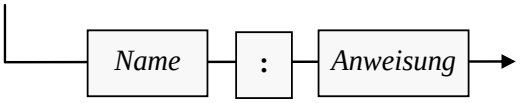
wobei die Methode hinter der **switch**-Anweisung fortgesetzt wird.

Per **goto**-Anweisung



kann eine **case**-Deklaration innerhalb der **switch**-Anweisung (inklusive **default**) oder ein Sprungziel an anderer Stelle innerhalb der Methode angesteuert werden:

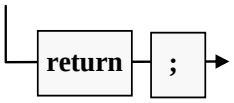
Sprungziel



Das gute (böse) alte **goto**, als Inbegriff rückständiger Programmierung („Spaghetti-Code“) aus vielen modernen Programmiersprachen verbannt, ist also in C# erlaubt. Es ist in manchen Situationen durchaus brauchbar, z. B. um den aus anderen Programmiersprachen bekannten **switch**-Durchfall in C# trotz Stopper-Vorschrift zu realisieren.

Mit der (später noch ausführlich zu behandelnden) **return**-Anweisung wird die gesamte Methode beendet. Bei einer Methode mit dem Rückgabetyp **void** ist die folgende Syntax zu verwenden:

return-Anweisung ohne Rückgabe



Weil im nächsten Abschnitt ein praxisnahes (und damit auch etwas kompliziertes) Beispiel folgt, ist hier ein ebenso einfaches wie sinnfreies Beispiel zur Erläuterung der Syntax angemessen:

```
using System;
class Prog {
  static void Main() {
    int zahl = 2;
    switch (zahl) {
      case 1:
        Console.WriteLine("Fall 1, mit break-Stopper");
        break;
      case 2:
        Console.WriteLine("Fall 2, mit goto-Stopper (Durchfall)");
        goto case 3;
      case 3:
      case 4:
        Console.WriteLine("Fälle 3, 4, mit goto-Stopper (Sprungziel)");
        goto Fertig;
      default:
        Console.WriteLine("Restkategorie, mit return-Stopper");
        return;
    }
  Console.WriteLine("Hinter break");
  Fertig:
    Console.WriteLine("Fertig");
  }
}
```

Es produziert die folgende Ausgabe:

```
Fall 2, mit goto-Stopper (Durchfall)
Fälle 3, 4, mit goto-Stopper (Sprungziel)
Fertig
```

4.7.2.3.2 Praxisnahes switch-Beispiel und Erstkontakt mit Befehlszeilenargumenten

Im folgenden Beispielprogramm wird die Persönlichkeit des Benutzers mit Hilfe seiner Farb- und Zahlpräferenz analysiert. Während bei einer Vorliebe für Rot oder Schwarz die Diagnose sofort feststeht, wird bei den restlichen Farben auch die Lieblingszahl berücksichtigt:

```
using System;
class PerST {
   static void Main(String[] args) {
      if (args.Length < 2) {
         Console.WriteLine("Bitte Lieblingsfarbe und -zahl angeben!");
         return;
      }
      String farbe = args[0].ToLower();
      int zahl = Convert.ToInt32(args[1]);
      switch (farbe) {
         case "rot":
            Console.WriteLine("Sie sind durchsetzungsfreudig und impulsiv.");
            break;
         case "schwarz":
            Console.WriteLine("Nehmen Sie nicht alles so tragisch.");
            break;
         default:
            Console.WriteLine("Ihre Emotionalität ist unauffällig.");
            if (zahl % 2 == 0)
               Console.WriteLine("Sie haben einen geradlinigen Charakter.");
            else
               Console.WriteLine("Sie machen wohl gerne krumme Touren.");
            break;
      }
   }
}
```

Das Programm `PerST` demonstriert nicht nur die **switch**-Anweisung, sondern auch die Verwendung von **Befehlszeilenargumenten**. Benutzer des Programms sollen ihre bevorzugte Farbe sowie ihre Lieblingszahl über Befehlszeilenargumente (Kommandozeilenparameter) angeben. Wer z. B. die Farbe Blau und die Zahl 17 bevorzugt, sollte das Programm (bis auf die beliebige Groß-/Kleinschreibung) folgendermaßen starten:

```
perst Blau 17
```

Im Quellcode wird jeweils nur *eine* Anweisung benötigt, um die (durch Leerzeichen getrennten) Befehlszeilenargumente auszuwerten und das Ergebnis in eine **String**- bzw. **int**-Variable zu befördern. Solche Anweisungen werden Sie mit Leichtigkeit selbst formulieren, sobald Methoden-Parameter sowie Arrays und Zeichenketten behandelt worden sind. An dieser Stelle greifen wir späteren Erläuterungen mal wieder etwas vor (hoffentlich mit motivierendem Effekt):

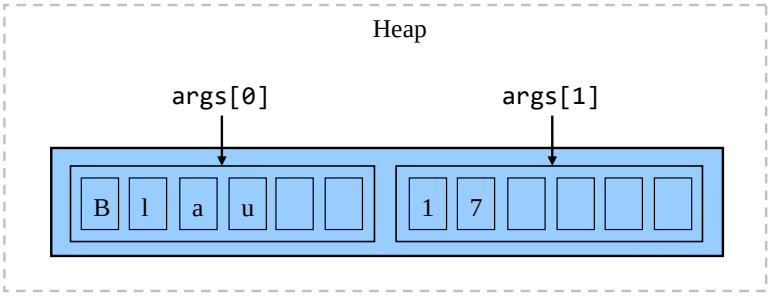
- Bei einem **Array** handelt es sich um ein Objekt, das eine Serie von Elementen desselben Typs aufnimmt, auf die man per Index, d.h. durch die mit eckigen Klammern begrenzte Elementnummer, zugreifen kann.
- In unserem Beispiel kommt ein Array mit Elementen vom Datentyp **String** zum Einsatz, wobei es sich um Zeichenfolgen handelt. Literale mit diesem Datentyp sind uns schon öfter begegnet (z. B. `"Hallo"`).
- Über die Parameterliste kann man eine Methode mit Daten versorgen und/oder ihre Arbeitsweise beeinflussen.
- Besitzt die **Main()** - Methode einer Startklasse einen Parameter vom Datentyp **String[]** (Array mit **String**-Elementen), dann übergibt das .NET - Laufzeitsystem dieser Methode als Elemente des **String**-Arrays die Spezifikationen, die der Anwender beim Start hinter den Programmnamen in die Kommandozeile, jeweils durch Leerzeichen getrennt, geschrieben hat. Der Datentyp des Parameters ist fest vorgegeben, sein Name ist jedoch frei wählbar (im Beispiel: `args`). Damit die Methode **Main()** weiterhin zum Starten des Programms taugt, darf kein weiterer Parameter vorhanden sein. In der Methode **Main()** kann man auf den Array `args` genauso zugreifen wie auf eine lokale Variable.

- Das erste Befehlszeilenargument landet im ersten Element des **String**-Arrays `args` und wird mit `args[0]` angesprochen, weil Array-Elemente mit 0 beginnend nummeriert sind. Als Objekt der Klasse **String** wird `args[0]` aufgefordert, die Methode **ToLower()** auszuführen. Diese Methode erstellt ein neues **String**-Objekt, das im Unterschied zum angesprochenen Original auf Kleinschreibung normiert ist, was die spätere Verwendung im Rahmen der **switch**-Anweisung erleichtert. Die Adresse dieses Objekts landet als **ToLower()** -Rückgabe in der lokalen **String**-Referenzvariablen `farbe`.
- Das zweite Element des Zeichenketten-Arrays `args` (mit der Nummer 1) enthält das zweite Befehlszeilenargument. Zumindest bei kooperativen Benutzern kann man aus dieser Zeichenfolge mit der Klassenmethode **ToInt32()** der Klasse **Convert** eine Zahl vom Datentyp **int** gewinnen und anschließend der lokalen Variablen `zahl` zuweisen.

Nach einem Programmstart mit dem Aufruf

```
perst Blau 17
```

kann man sich den **String**-Array `args`, der als Objekt im Heap-Bereich des programmeigenen Speichers abgelegt wird, ungefähr so vorstellen:[1]



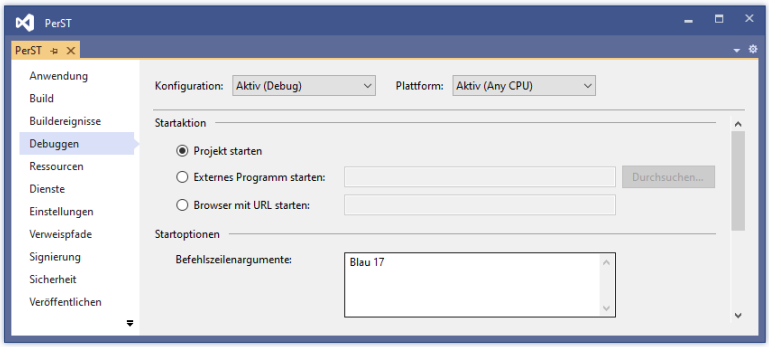
Ansonsten ist im Beispielprogramm noch die **return**-Anweisung von Interesse, welche die **Main()** -Methode und damit das Programm in Abhängigkeit von einer Bedingung sofort beendet:

```
return;
```

Wir haben die **return**-Anweisung eben schon als Stopper im Rahmen der **switch**-Anweisung kennengelernt. Ihre „offizielle“ Behandlung erfolgt später im Zusammenhang mit den Methoden.

Für den Programmstart aus dem Visual Studio kann man die Befehlszeilenargumente folgendermaßen vereinbaren:

- Menübefehl **Projekt > Eigenschaften**
- Registerkarte **Debuggen**
- **Befehlszeilenargumente** eintragen, z. B.:

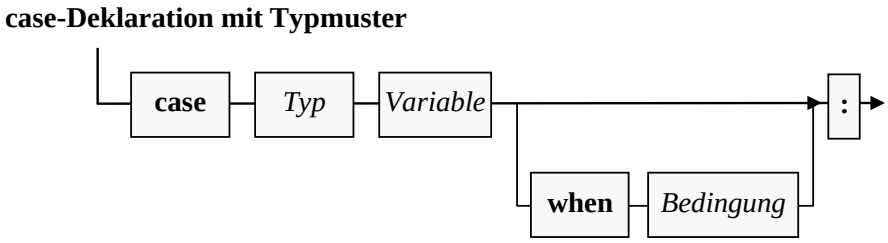


[1] Hier wird aus didaktischen Gründen ein wenig gemogelt: Die beiden Zeichenfolgen sind selbst Objekte und liegen „neben“ dem Array-Objekt auf dem Heap. Die Array-Elemente sind Referenzen, die auf die zugehörigen **String**-Objekte zeigen.

4.7.2.3.3 case-Deklaration mit Typmuster

Die seit C# 7.0 verfügbare **case**-Deklaration mit Typmuster bringt einige Vorteile, die teilweise auch für Programmiereinsteiger nachvollziehbar und nützlich sind. Hier ist vor allem die Möglichkeit von Interesse, in einem **case** einen kompletten Wertbereich behandeln zu können, ohne die Werte einzeln auflisten zu müssen (siehe Beispiel am Ende des Abschnitts). Allerdings sind manche Details der **case**-Deklaration mit Typmuster für Einsteiger schwer zu verdauen. Es ist daher vertretbar, wenn Sie sich vorläufig auf die traditionelle **switch**-Syntax beschränken und Ihre C# - Kompetenz später um die sehr empfehlenswerten **switch**-Neuerungen erweitern.

Das Syntaxdiagramm zur **case**-Deklaration mit Typmuster ist recht übersichtlich:



Insgesamt können folgende Konstellationen dazu führen, dass zur Laufzeit eine **case**-Deklaration mit Typmuster für den aktuellen Wert des **switch**-Ausdrucks als passend beurteilt wird:[1]

- Der zur Übersetzungszeit feststehende Typ des **switch**-Ausdrucks ist der **case**-Typ oder ein daraus abgeleiteter Typ.
- Der zur Übersetzungszeit feststehende Typ des **switch**-Ausdrucks ist eine Basisklasse des **case**-Typs, und der Laufzeittyp des **switch**-Ausdrucks ist vom **case**-Typ oder von einem daraus abgeleiteten Typ. Erst im Zusammenhang mit der Vererbung werden wir offiziell zur Kenntnis nehmen, dass eine Referenz nicht nur auf ein Objekt vom eigenen Typ, sondern auch auf ein Objekt aus einer abgeleiteten Klasse zeigen kann.
- Ist als **case**-Typ eine *Schnittstelle* (siehe sehr weit unten) angegeben, dann muss der Laufzeittyp des **switch**-Ausdrucks diese Schnittstelle implementieren.

Ist eine **when**-Klausel vorhanden, dann muss auch deren Bedingung erfüllt sein, bevor die **case**-Deklaration als passend bewertet wird.

Bei diesem Verfahren können sich *mehrere* **case**-Deklarationen als passend erweisen. Dann gewinnt die *erste*, sodass die Reihenfolge der **case**-Deklarationen relevant wird. Dies ist ein gravierender Unterschied zu der **switch**-Anweisung auf dem Sprachniveau von C# 6.0. Dort werden in den **case**-Deklarationen ausschließlich Konstanten verwendet, sodass stets maximal eine **case**-Deklaration passt, und die Reihenfolge der Deklarationen irrelevant ist.

Der Compiler erlaubt in einer **switch**-Anweisung einen Mix aus **case**-Deklarationen mit einer Konstanten bzw. mit einem Typmuster und verhindert außerdem nach Möglichkeit, dass eine **case**-Deklaration unter einer allgemeineren steht und daher nie erreicht werden kann. Im folgenden Beispiel bemerkt der Compiler aber nicht, dass die **case**-Deklaration

```
case string s when s.Contains("ha")
```

unerreichbar ist, weil die allgemeinere Deklaration

```
case string s when s.Length > 0:
```

darüber steht:

---

[1] https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/switch

Quellcode	Ausgabe

```
using System;
class Prog {
    static void Main() {
        string str = "hallo";
        switch (str) {
            case "hi":
                Console.WriteLine("string-Literal");
                break;
            case string s when s.Length > 0:
                Console.WriteLine("Typ String, Länge > 0");
                break;
            case string s when s.Contains("ha"):
                Console.WriteLine("Typ String, enthält \"ha\"");
                break;
            case string s:
                Console.WriteLine("Typ String");
                break;
            case null:
                Console.WriteLine("null");
                break;
        }
    }
}
```

```
Typ String, Länge > 0
```

Per **when**-Klausel lässt sich eine **case**-Deklaration mit Bereichsangabe realisieren, die in der traditionellen **switch**-Syntax (bis C# 6.0) oft vermisst wurde, z. B.:

Quellcode	Ausgabe

```
using System;
class Prog {
    static void Main() {
        int i= 5;
        switch (i) {
            case 3:
                Console.WriteLine("3");
                break;
            case int ia when ia > 3  &&  ia <= 10:
                Console.WriteLine("von 4 bis 10");
                break;
            case 99:
                Console.WriteLine("99");
                break;
            default:
                Console.WriteLine("default");
                break;
        }
    }
}
```

```
von 4 bis 10
```

Seit C# 8 sind die Optionen zum Mustervergleich erheblich erweitert worden, und seit der Version 9 ist das neue Relationsmuster eine bessere Alternative zu mancher **when**-Klausel (siehe Abschnitt 15.1).

4.7.2.4 switch-Ausdruck

Zur Ergänzung der **switch**-Anweisung (siehe Abschnitt 4.7.2.3) wurde in C# 8.0 der **switch**-Ausdruck eingeführt.[1] Wie die im Abschnitt 4.7.2.3.3 beschriebene **case**-Deklaration mit Typmuster ist auch der **switch**-Ausdruck eher für fortgeschrittene Programmierer zu empfehlen. Daher beschränken wir uns an dieser Stelle auf eine elementare Beschreibung und liefern im Abschnitt 15.1 eine vollständige Behandlung nach.

Die Verwendung eines **switch**-Ausdrucks bietet sich z. B. an, wenn eine Abbildung von einem Ausgangstyp in einen Ergebnistyp stattfinden soll. Im folgenden Beispiel wird eine Eingabe vom Typ **int** in eine Ausgabe vom Typ **String** übersetzt. In einer Personendatenbank sei der Charakter unter Verwendung der vier Temperamentstypen des griechischen Philosophen Hippokrates (melancholisch, cholerisch, phlegmatisch, sanguinisch) durch eine **int**-Variable gespeichert.[2] Per **switch**-Ausdruck sollen die Zahlen in aussagekräftige Zeichenfolgen übersetzt werden:

Quellcode	Ausgabe

`using System;`
`class Prog5 {`
`  static void Main() {`
`  int charCode = 3;`
`  String charLabel = charCode switch {`
`  1 => "melancholisch",`
`  2 => "cholerisch",`
`  3 => "phlegmatisch",`
`  4 => "sanguinisch",`
`  _ => "undefiniert"`
`  };`
`  Console.WriteLine($"Charakter: {charLabel}");`
`  }`
`}`
	`Charakter: phlegmatisch`

Den Ergebnistyp des **switch**-Ausdrucks ermittelt der Compiler aus der Verwendung des Ausdrucks im Rahmen einer Wertzuweisung oder einer **return**-Anweisung.

Syntaxregeln:

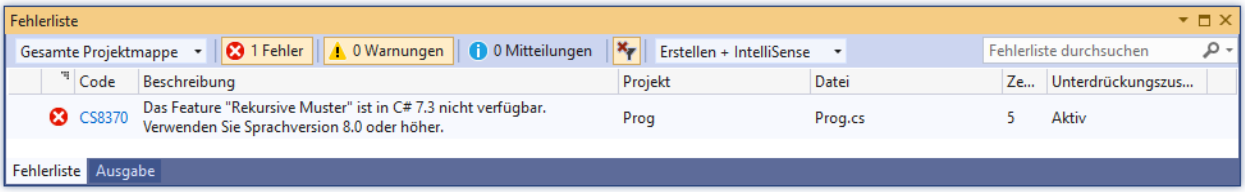
- Zu den Fällen gehören keine Anweisungen, sondern Ausdrücke, die syntaktisch elegant mit dem Symbol => zugewiesen werden.
- Es muss *kein Durchfall* (z. B. per **break**) verhindert werden.
- Per Unterstrich können alle sonstigen Werte des steuernden Ausdrucks angesprochen werden. Der Unterstrich wird in C# oft zur Kennzeichnung von Fällen verwendet, die ausgeschlossen bzw. verworfen werden sollen (engl. *discarded*).
- Damit unter Verwendung eines **switch**-Ausdrucks eine vollständige Anweisung entsteht (z. B. eine Wertzuweisung), muss ein Semikolon am Ende der Anweisung (also hinter der schließenden Klammer des **switch**-Blocks) stehen.

Um **switch**-Ausdrücke verwenden zu können, muss mindestens die C# - Version 8.0 zur Verfügung stehen. Das ist z. B. in einem Projekt mit .NET 5.0 als Zielframework der Fall. Der Versuch, einen **switch**-Ausdruck im Rahmen eines Projekts mit .NET 4.8 als Zielframework zu verwenden, führt zu einer Fehlermeldung:

---

[1] Wir befinden uns gerade im Abschnitt über *Anweisungen*, und die **switch**-Ausdrücke hätten eigentlich im Abschnitt 4.5 behandelt werden müssen. Trotz dieses Arguments ist eine Behandlung der **switch**-Ausdrücke *nach* den traditionellen (und noch lange stark verbreiteten) **switch**-Anweisungen didaktisch aber sehr viel sinnvoller.

[2] Hippokrates lebte ca. von 460 bis 370 v. Chr.



Im Vergleich zur obigen Lösung per **switch**-Ausdruck führt die traditionelle **switch**-Anweisung zu einem höheren Aufwand und zu einem weniger übersichtlichen Ergebnis:

```
using System;
class Prog {
    static void Main() {
        int charCode = 4;
        String charLabel;
        switch (charCode) {
            case 1:
                charLabel = "melancholisch";
                break;
            case 2:
                charLabel = "cholerisch";
                break;
            case 3:
                charLabel = "phlegmatisch";
                break;
            case 4:
                charLabel = "sanguinisch";
                break;
            default:
                charLabel = "undefiniert";
                break;
        };
        Console.WriteLine($"Chrakter: {charLabel}");
    }
}
```

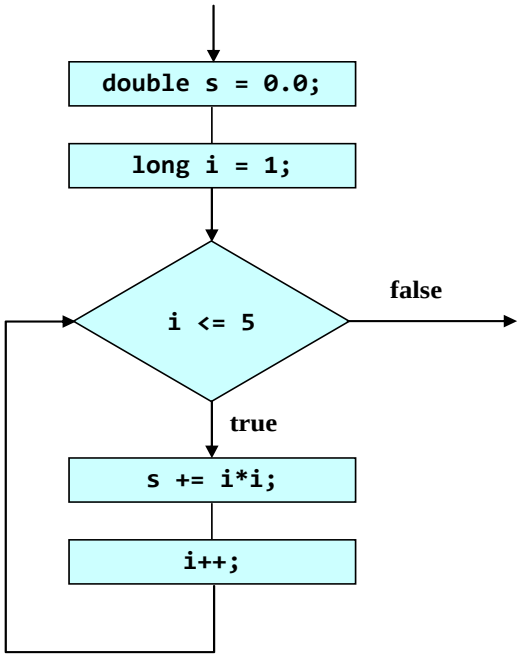
4.7.3 Wiederholungsanweisungen

Eine Wiederholungsanweisung (oder schlicht: *Schleife*) kommt zum Einsatz, wenn eine (Verbund-)Anweisung *mehrfach* ausgeführt werden soll, wobei sich in der Regel schon der Gedanke daran verbietet, die Anweisung entsprechend oft in den Quelltext zu schreiben.

Im folgenden Flussdiagramm ist ein iterativer Algorithmus zu sehen, der die Summe der quadrierten natürlichen Zahlen von 1 bis 5 berechnet:[1]

[1] Das Verzweigungssymbol sieht aus darstellungstechnischen Gründen etwas anders aus als im Abschnitt 4.7.2, was aber keine Verwirrung stiften sollte. Obwohl im Beispiel eine Steigerung der Laufgrenze für die Variable `i` kaum in Frage kommt, soll an dieser Stelle das Thema *Ganzzahlüberlauf* (vgl. Abschnitt 4.6.1) kurz in Erinnerung gerufen werden. Weil die Variable `i` vom Typ **long** ist, kann der Algorithmus bis zur Laufgrenze 3037000499 verwendet werden. Für größere `i`-Werte tritt beim Ausdruck `i*i` ein Überlauf auf, und das Ergebnis ist unbrauchbar. Eine einfache Möglichkeit zur Steigerung der maximalen sinnvollen Laufgrenze besteht darin, für eine Berechnung der Summanden per Gleitkommaarithmetik zu sorgen:

```
(double)i * i
```



C# bietet verschiedene Wiederholungsanweisungen, die sich bei der Ablaufsteuerung unterscheiden. Wir werden sie gleich im Detail betrachten und als Beispiel jeweils den Algorithmus aus dem obigen Flussdiagramm implementieren. Zunächst sollen die Optionen zur Schleifensteuerung bei leicht vereinfachender Beschreibung im Überblick präsentiert werden:

- **Zählergesteuerte Schleife** (**for**)
  Die Anzahl der Wiederholungen steht typischerweise schon vor Schleifenbeginn fest. Bei der Ablaufsteuerung kommt eine Zählvariable zum Einsatz, die *vor dem ersten* Schleifendurchgang initialisiert und *am Ende jedes* Durchlaufs aktualisiert (z. B. inkrementiert) wird. Die zur Schleife gehörige (Verbund-)Anweisung wird ausgeführt, solange die Zählvariable einen festgelegten Grenzwert nicht über- bzw. unterschritten hat.
- **Iterieren über die Elemente einer Kollektion** (**foreach**)
  Mit der **foreach**-Schleife bietet C# die Möglichkeit, eine Anweisung für jedes Element eines Arrays, einer Zeichenfolge oder einer anderen Kollektion (siehe unten) auszuführen.
- **Bedingungsabhängige Schleife** (**while**, **do**)
  Bei jedem Schleifendurchgang wird eine Bedingung überprüft, und das Ergebnis entscheidet über das weitere Vorgehen:
  - **true**: Die zur Schleife gehörige Anweisung wird ein weiteres Mal ausgeführt.
  - **false**: Die Schleife wird beendet.

  Bei der *kopf*gesteuerten **while**-Schleife wird die Bedingung *vor Beginn* eines Durchgangs geprüft, bei der *fuß*gesteuerten **do**-Schleife hingegen *am Ende*. Weil man z. B. *nach* dem 3. Schleifendurchgang in keiner anderen Lage ist als *vor* dem 4. Schleifendurchgang, geht es bei der Entscheidung zwischen Kopf- und Fußsteuerung lediglich darum, ob auf jeden Fall ein *erster* Schleifendurchgang stattfinden soll (**do**-Schleife) oder nicht (**while**-Schleife).

Die gesamte Konstruktion aus Schleifensteuerung und (Verbund-)anweisung stellt in syntaktischer Hinsicht *eine zusammengesetzte* Anweisung dar.

4.7.3.1 Zählergesteuerte Schleife (for)

Die Anweisung einer **for**-Schleife wird ausgeführt, solange eine Bedingung erfüllt ist, die normalerweise auf eine ganzzahlige Laufvariable Bezug nimmt.

Auf das Schlüsselwort **for** folgt die von runden Klammern umgebene Schleifensteuerung, wo die Vorbereitung der Laufvariablen (nötigenfalls samt Deklaration), die Fortsetzungsbedingung und die Aktualisierungsvorschrift untergebracht werden. Danach folgt die wiederholt auszuführende (Block-)Anweisung:

for (*Vorbereitung***;** *Bedingung***;** *Aktualisierung***)**
    Anweisung

Zu den drei Bestandteilen der Schleifensteuerung sind einige Erläuterungen erforderlich, wobei anschließend etliche weniger sinnvolle Möglichkeiten weggelassen werden:

- ***Vorbereitung***
  In der Regel wird man sich auf *eine* Laufvariable beschränken und dabei einen Ganzzahl-Typ wählen. Somit kommen im Vorbereitungsteil der **for**-Schleifensteuerung in Frage:
  - eine Variablendeklaration mit Initialisierung, z. B.
    ```
    long i = 1
    ```
  - eine Wertzuweisung für eine bereits deklarierte Variable, z. B.:
    ```
    i = 1
    ```

  Im folgenden Programm, das die Summe der quadrierten natürlichen Zahlen von 1 bis 5 berechnet, findet sich die erste Variante:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  double s = 0.0;`
`  for (long i = 1; i <= 5; i++)`
`  s += i*i;`
`  Console.WriteLine("Quadratsumme = " + s);`
`  }`
`}`
	`Quadratsumme = 55`

  Der Vorbereitungsteil wird *vor dem ersten Durchlauf* ausgeführt. Eine hier deklarierte Variable ist *lokal* bzgl. der **for**-Schleife, also nur in deren Anweisung(sblock) sichtbar. Eine möglichst eingeschränkte Sichtbarkeit mindert das Risiko von Programmierfehlern (siehe Abschnitt 4.3.8).

- ***Bedingung***
  Üblicherweise wird eine Ober- oder Untergrenze für die Laufvariable gesetzt, doch erlaubt C# beliebige logische Ausdrücke. Die Bedingung wird *vor jedem Schleifendurchgang* geprüft. Resultiert der Wert **true**, dann wird die eingebettete Anweisung (der Schleifenrumpf) ausgeführt, anderenfalls wird die **for**-Schleife verlassen. Folglich kann es auch passieren, dass überhaupt kein Schleifendurchgang zustande kommt.

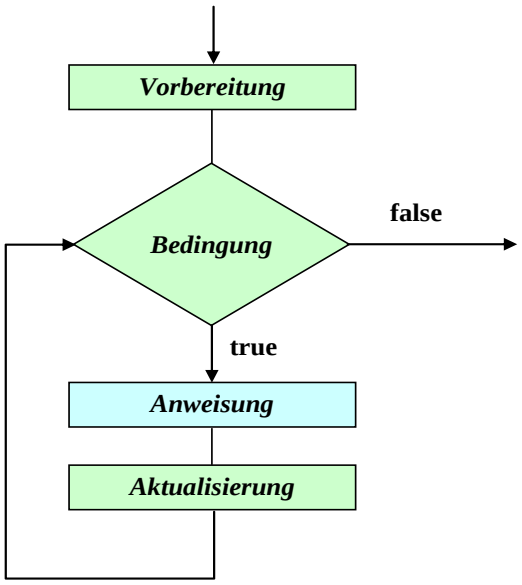
- ***Aktualisierung***
  Am Ende jedes Schleifendurchgangs (nach der Ausführung der Anweisung) wird die Aktualisierung ausgeführt. Dabei wird meist die Laufvariable in- oder dekrementiert. In der C# - Sprachreferenz werden einige Alternativen für die meist als Aktualisierungsvorschrift verwendete In- oder Dekrementoperation beschrieben, z. B.:[1]
  - Wertzuweisung
  - Methodenaufruf

  Im folgenden Beispiel sorgt ein **WriteLine()** - Aufruf mit Postinkrementoperation als Argument dafür, dass die Laufvariable protokolliert und inkrementiert wird:

```
for (long i = 1; i <= 5; Console.WriteLine(i++))
    s += i * i;
```

  Während es für das Ergebnis der **for**-Schleife irrelevant ist, ob die Aktualisierung eine Prä- oder eine Postinkrementoperation enthält, hat die Wahl einen Effekt auf die **WriteLine()** - Ausgabe.

Im folgenden Flussdiagramm ist das Ablaufverhalten der **for**-Schleife dargestellt, wobei die Bestandteile der Schleifensteuerung an der grünen Farbe zu erkennen sind:



Zu den (zumindest stilistisch) bedenklichen Konstruktionen, die der Compiler klaglos akzeptiert, gehören **for**-Schleifenköpfe ohne Vorbereitung oder ohne Aktualisierung, wobei die trennenden Strichpunkte im Schleifenkopf trotzdem zu setzen sind. In solchen Fällen ist die Umlaufzahl einer **for**-Schleife natürlich nicht mehr aus dem Schleifenkopf abzulesen. Dies gelingt auch dann nicht, wenn …

- eine Indexvariable in der Schleifenanweisung modifiziert wird,
- die Schleife per **break**-, **goto**- oder **return**-Anweisung verlassen wird (siehe Abschnitt 4.7.3.5).

[1] https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/for

4.7.3.2 Iterieren über die Elemente einer Kollektion (foreach)

Obwohl wir uns bisher nur anhand von Beispielen mit *Kollektionen* wie Arrays oder Zeichenfolgen beschäftigt haben, soll die einfach aufgebaute **foreach**-Schleife doch hier im Kontext mit den übrigen Schleifen behandelt werden. Konzentrieren Sie sich also auf das gleich präsentierte, leicht nachvollziehbare Beispiel, und lassen Sie sich durch die zu später behandelten Themen gehörenden Begriffe *Array*, *Kollektion* und *Interface* nicht beunruhigen.

Die Steuerungslogik der **foreach**-Schleife:

- Es ist eine Kollektion mit einer Anzahl von Elementen vorhanden, z. B. eine Zeichenfolge mit acht Zeichen.
- Die Anweisung der **foreach**-Schleife wird nacheinander für jedes Element der Kollektion ausgeführt.
- Im Schleifenkopf wird eine Iterationsvariable vom Datentyp der Kollektionselemente deklariert, über die in der Schleifenanweisung das aktuelle Element angesprochen werden kann.

Die Syntax der **foreach**-Schleife:

foreach (*Elementtyp Iterationsvariable* **in** *Kollektion***)**
    Anweisung

Als Kollektion erlaubt der Compiler:[1]

- einen Array
- eine Instanz eines Typs, der das Interface **IEnumerable** im Namensraum **System.Collections** oder das Interface **IEnumerable<T>** im Namensraum **System.Collections.Generic** implementiert.

Das Beispielprogramm `PerST` im Abschnitt 4.7.2.3.2 hat demonstriert, wie man über einen Parameter der Methode **Main()** auf die Zeichenfolgen zugreifen kann, welche der Benutzer beim Start eines Konsolenprogramms hinter den Assembly-Namen geschrieben hat. Im folgenden Programm wird durch zwei geschachtelte **foreach**-Schleifen für jedes Element im **string**-Array `args` mit den Befehlszeilenargumenten folgendes getan:

- In der äußeren **foreach**-Schleife wird die aktuelle Zeichenfolge komplett ausgegeben.
- In der inneren **foreach**-Schleife wird jedes Zeichen der aktuellen Zeichenfolge separat ausgegeben.

---

[1] Genaugenommen ...

- muss das Interface **IEnumerable** bzw. **IEnumerable <T>** nicht *explizit* implementiert werden (siehe Kapitel **9**). Es genügt, die Methode **GetEnumerator()** mit einer Rückgabe vom Typ **IEnumerator** bzw. **IEnumerator<T>** zu implementieren (siehe Abschnitt 9.7), z. B.:

```
class EnumeratorFrom0To4 {
  public System.Collections.Generic.IEnumerator<int> GetEnumerator() {
    for (int i = 0; i < 5; i++) yield return i;
  }
}
class Prog {
  public static void Main() {
    var from0T4  = new EnumeratorFrom0To4 ();
    foreach (int i in from0T4) System.Console.WriteLine(i);
  }
}
```

- hätte die **foreach**-Tauglichkeit von Arrays nicht explizit genannt werden müssen, weil jeder Array ein Objekt aus einer Klasse ist, welche die Schnittstelle **IEnumerable** implementiert.

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main(string[] args) {`
`  foreach (string s in args) {`
`  Console.WriteLine(s);`
`  foreach (char c in s)`
`  Console.WriteLine(" " + c);`
`  Console.WriteLine();`
`  }`
`  }`
`}`
	`eins`
	` e`
	` i`
	` n`
	` s`
	
	`zwei`
	` z`
	` w`
	` e`
	` i`

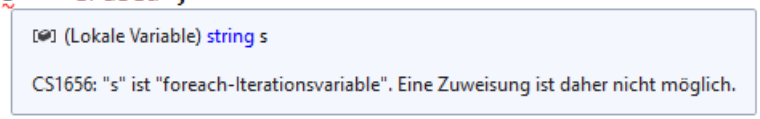
Beim Array mit den Befehlszeilenargumenten haben wir es mit einer Kollektion zu tun, die als Elemente wiederum Kollektionen enthält (nämlich **String**-Objekte).[1]

Bloch (2018, S. 264ff) empfiehlt nachdrücklich, wegen der folgenden Vorteile die **foreach** - Schleife nach Möglichkeit gegenüber der traditionellen **for**-Schleife zu bevorzugen:

- Oft werden die Flexibilität (und der Aufwand) bei der Initialisierung, Überprüfung und Aktualisierung der Indexvariablen nicht benötigt. In der **foreach** - Syntax beschränkt man sich auf die Elementzugriffsvariable und erhält einen besser lesbaren Quellcode.
- Durch den Verzicht auf eine Indexvariable entfallen Fehlermöglichkeiten.
- Weil in der **foreach** - Schleife für Arrays und Kollektionen dieselbe Syntax verwendet wird, macht der Wechsel der Container-Architektur wenig Aufwand.

Eine wichtige Einschränkung der **foreach**-Schleife besteht darin, dass man in ihrer Anweisung über die Iterationsvariable nur *lesend* auf die Kollektionselemente zugreifen kann, sodass z. B. der folgende Versuch zum „Löschen“ der Elemente im **string**-Array `args` misslingt:

```
foreach (string s in args)
    s = "erased";
```



Der Grund für dieses Verhalten besteht darin, dass die Iterationsvariable eine *Kopie* des aktuellen Kollektionselements enthält, sodass eine Änderung sinnlos wäre.[2]

Über eine **for**-Schleife ist der Plan aus dem letzten Beispiel durchaus zu realisieren, z. B.:

```
for (int i = 0; i < args.Length; i++)
    args[i] = "erased";
```

4.7.3.3 Bedingungsabhängige Schleifen

Wie die Erläuterungen zur **for**-Schleife gezeigt haben, ist die Überschrift zu diesem Abschnitt nicht sehr trennscharf, weil bei der **for**-Schleife ebenfalls eine beliebige Terminierungsbedingung angegeben werden darf. In vielen Fällen ist es eine Frage des persönlichen Geschmacks, welche Wiederholungsanweisung man zur Lösung eines konkreten Iterationsproblems einsetzt. Unter der aktuellen Abschnittsüberschrift werden traditionsgemäß die **while**- und die **do**-Schleife diskutiert.

---

[1] Als syntaktische Besonderheit kennt C# für den Klassennamen **String** einen Alias mit kleinem Anfangsbuchstaben.

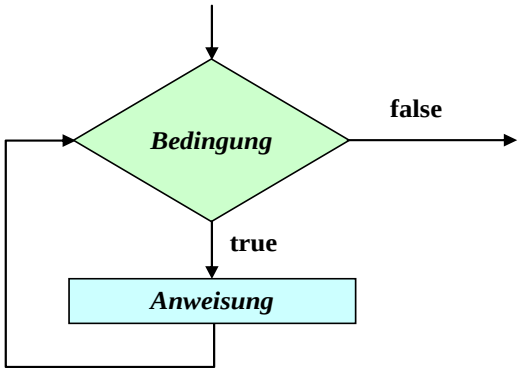
[2] Weitere Details sind hier zu finden:
http://stackoverflow.com/questions/4004755/why-is-foreach-loop-read-only-in-c-sharp

4.7.3.3.1 while-Schleife

Die **while**-Anweisung kann als vereinfachte **for**-Anweisung beschrieben werden: Wer im Kopf einer **for**-Schleife auf Vorbereitung und Aktualisierung verzichten möchte, ersetzt besser das Schlüsselwort **for** durch **while** und erhält dann die folgende Syntax:

while (*Bedingung***)**
    Anweisung

Wie bei der **for**-Anweisung wird die Bedingung *vor Beginn* eines Schleifendurchgangs geprüft. Resultiert der Wert **true**, dann wird die Anweisung ausgeführt, anderenfalls wird die **while**-Schleife verlassen, eventuell noch vor dem ersten Durchgang:



Das im Abschnitt 4.7.3.1 vorgestellte Beispielprogramm zur Quadratsummenberechnung mit Hilfe einer **for**-Schleife kann leicht auf die Verwendung einer **while**-Schleife umgestellt werden:

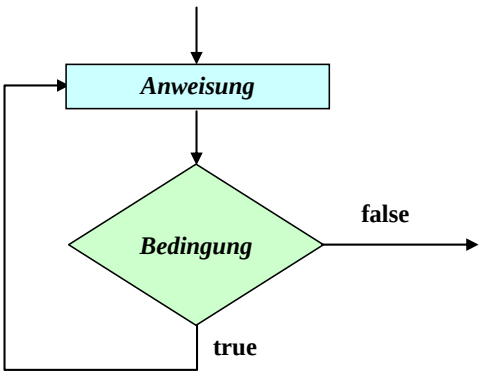
Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  double s = 0.0;`
`  long i = 1;`
`  while (i <= 5) {`
`  s += i * i;`
`  i++;`
`  }`
`  Console.WriteLine("Quadratsumme = " + s);`
`  }`
`}`
	`Quadratsumme = 55`

Ein Nachteil der im Beispiel verwendeten **while**-Schleife gegenüber der im Abschnitt 4.7.3.1 beschriebenen **for**-Schleife besteht darin, dass die Laufvariable `i` *außerhalb* der **while**-Schleife deklariert werden muss, was zu einem unnötig großen Gültigkeitsbereich für diese lokale Variable führt. Außerdem sind bei der **while**-Lösung der Schreibaufwand höher und die Lesbarkeit schlechter.

4.7.3.3.2 do-Schleife

Bei der **do**-Schleife wird die Fortsetzungsbedingung *am Ende* der Schleifendurchläufe geprüft, sodass mindestens *ein* Durchlauf stattfindet:



Das Schlüsselwort **while** tritt auch in der Syntax der **do**-Schleife auf:

do
Anweisung
while (*Bedingung***);**

do-Schleifen werden seltener benötigt als **while**-Schleifen, sind aber z. B. dann von Vorteil, wenn man vom Benutzer eine Eingabe mit bestimmten Eigenschaften einfordern möchte. Im folgenden Programm wird mit der statischen Methode **Console.ReadLine()** eine per **Enter** - Taste quittierte Zeile von der Konsole gelesen. Weil dieser Methodenaufruf ein Ausdruck vom Typ **String** ist, kann das erste Zeichen (mit der Nummer 0) per Indexzugriff (mit Hilfe der eckigen Klammern) angesprochen werden:

Quellcode	Ein-/Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  char antwort;`
`  do {`
`  Console.Write("Einverstanden? (j/n)? ");`
`  antwort = Console.ReadLine()[0];`
`  } while (antwort != 'j' && antwort != 'n');`
`  }`
`}`
	`Einverstanden? (j/n)? r`
	`Einverstanden? (j/n)? 4`
	`Einverstanden? (j/n)? j`

Bei einer **do**-Schleife mit Anweisungs*block* sollte man die **while**-Klausel unmittelbar hinter die schließende Blockklammer setzen (in dieselbe Zeile), um sie optisch von einer selbständigen **while**-Anweisung abzuheben (siehe Beispiel).

4.7.3.4 Endlosschleifen

Bei einer **for**-, **while**- oder **do**-Schleife kann es in Abhängigkeit von der Fortsetzungsbedingung passieren, dass der Anweisungsteil so lange wiederholt wird, bis das Programm von außen abgebrochen wird. Solche Endlosschleifen sind in seltenen Fällen intendiert, meist aber das Ergebnis eines gravierenden Programmierfehlers. Befindet sich ein Programm in diesem Zustand muss es mit Hilfe des Betriebssystems abgebrochen werden, bei unseren Konsolenanwendungen z. B. über die Tastenkombination **Strg+C**.

Im folgenden Beispiel resultiert eine Endlosschleife aus einer ungeschickten Identitätsprüfung bei **double**-Werten (vgl. Abschnitt 4.5.3):

```
using System;
class Prog {
  static void Main() {
    long i = 0;
    double d = 1.0;
    // besser: while (d > 1.0e-14) {
    while (d != 0.0) {
      i++;
      d = d - 0.1;
      Console.WriteLine("i = {0}, d = {1}", i, d);
    }
    Console.WriteLine("Fertig!");
  }
}
```

4.7.3.5 Schleifen(durchgänge) vorzeitig beenden

Mit der **break**-Anweisung, die uns schon im Abschnitt 4.7.2.3 als Bestandteil der **switch**-Anweisung begegnet ist, kann eine Schleife vorzeitig verlassen werden, wobei die Methode hinter der Schleife fortgesetzt wird.

Mit der **continue**-Anweisung erreicht man, dass der Anweisungsblock des aktuellen Schleifendurchgangs verlassen und die Bearbeitung folgendermaßen fortgesetzt wird:

- Bei der **for**-Schleife wird …
  - die Aktualisierung zum aktuellen Schleifendurchgang ausgeführt,
  - die Fortsetzungsbedingung geprüft und ggf. mit dem nächsten Schleifendurchgang begonnen.
- Bei der **while**- und der **do**-Schleife wird die Fortsetzungsbedingung geprüft und ggf. mit dem nächsten Schleifendurchgang begonnen.[1]

Im folgenden Beispielprogramm zur (relativ primitiven) Primzahlendiagnose wird die schon im Abschnitt 4.4.1 erwähnte Ausnahmebehandlung per **try-catch** - Anweisung eingesetzt, die später in einem eigenen Kapitel ausführlich behandelt wird. Wir leisten uns den Vorgriff, weil sich störende Fälle (negative, zu große oder nicht interpretierbare Eingaben) bequem abfangen lassen, und ein sinnvoller **continue**-Einsatz resultiert:

```
using System;
class Primitiv {
  static void Main() {
    bool tg;
    ulong i, mtk, zahl;
    Console.WriteLine("Einfacher Primzahlendetektor\n");
    while (true) {
      Console.Write("Zu untersuchende ganze Zahl von 2 bis 2^64-1 oder 0 zum Beenden: ");
      try {
        zahl = Convert.ToUInt64(Console.ReadLine());
      } catch {
        Console.WriteLine("Keine ganze Zahl (im zulässigen Bereich)!\n");
        continue;
      }
      if (zahl == 1) {
        Console.WriteLine("1 ist per Definition keine Primzahl.\n");
        continue;
      }
```

[1] Dass bei der **do**-Schleife die Fortsetzungsbedingung des abzubrechenden Schleifendurchgangs trotzdem geprüft wird, war in früheren Ausgaben dieses Skripts übersehen worden. Der Fehler wurde von Marian Peters entdeckt und gemeldet.

```
        if (zahl == 0)
          break;

        tg = false;
        mtk = (ulong) (Math.Sqrt(zahl) + 0.5); //Maximaler Teilerkandidat
        for (i = 2; i <= mtk; i++)
          if (zahl % i == 0) {
            tg = true;
            break;
          }

        if (tg)
          Console.WriteLine(zahl + " ist keine Primzahl (Teiler: " + i + ").\n");
        else
          Console.WriteLine(zahl + " ist eine Primzahl.\n");
      }
      Console.WriteLine("\nVielen Dank für den Einsatz dieser Software!");
      Console.ReadLine();
    }
  }
```

Bei einer irregulären, nicht als ganze Zahl im **ulong**-Wertebereich interpretierbaren Eingabe „wirft“ die **Convert**-Methode **ToUInt64()** eine Ausnahme. Weil sich der Methodenaufruf in einem **try**-Block befindet, wird im Ausnahmefall der zugehörige **catch**-Block ausgeführt. Im Beispielprogramm erscheint dann eine Fehlermeldung auf dem Bildschirm, und der aktuelle Durchgang der **while**-Schleife wird per **continue** verlassen.

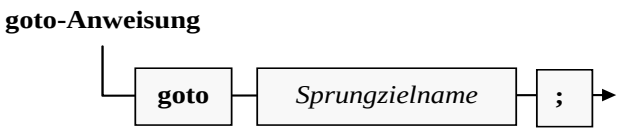
Auch nach der Eingabe 1 wird der aktuelle Schleifendurchgang per **continue** verlassen, weil keine Prüfung erforderlich ist. Der Benutzer wird darüber informiert, dass die 1 per Definition keine Primzahl ist.

Durch Eingabe der Zahl 0 kann das Beispielprogramm beendet werden, wobei die absichtlich konstruierte **while** - „Endlosschleife“ per **break** verlassen wird.

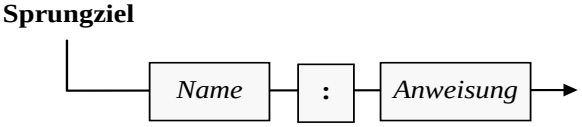
Man hätte die **continue**- und die **break**-Anweisungen zwar vermeiden können (siehe Übungsaufgabe im Abschnitt 4.7.4), doch werden beim vorgeschlagenen Verfahren lästige Sonderfälle (unzulässige Werte, 1 als Kandidat, 0 als Terminierungssignal) auf besonders übersichtliche Weise abgehakt, bevor der Kernalgorithmus startet.

Neben der **break**-Anweisung stehen weitere, seltener benötigte Anweisungen zum vorzeitigen Verlassen einer Schleife zur Verfügung:

- **goto**-Anweisung
  Über die bereits im Zusammenhang mit der **switch**-Verzweigung (siehe Abschnitt 0) beschriebenen **goto**-Anweisung

  kann ein Sprungziel

  außerhalb der Wiederholungsanweisung (aber innerhalb der Methode) angesteuert werden. Das gute (böse) alte **goto**, als Inbegriff rückständiger Programmierung („Spaghetti-Code“) aus vielen modernen Programmiersprachen verbannt, ist also in C# erlaubt.

- **return**-Anweisung
  Über die bereits mehrfach verwendete, aber noch nicht offiziell behandelte **return**-Anweisung (siehe Abschnitt 5.3.1.2) wird die Methode verlassen, was im Fall der Methode **Main()** einer Beendigung des Programms gleichkommt.

Zum Kernalgorithmus der Primzahlendiagnose sollte vielleicht noch erläutert werden, warum die Suche nach einem Teiler des Primzahlkandidaten bei seiner Wurzel enden kann (genauer: bei der größten ganzen Zahl ≤ Wurzel):

Sei *d* (≥ 2) ein echter Teiler der positiven, ganzen Zahl *z*, d. h. es gebe eine Zahl *k* (≥ 2) mit

$$z = k \cdot d$$

Dann ist auch *k* ein echter Teiler von *z*, und es gilt:

$$d \leq \sqrt{z} \quad \text{oder} \quad k \leq \sqrt{z}$$

Anderenfalls wäre das Produkt $k \cdot d$ größer als *z*. Wir haben also folgendes Ergebnis: Wenn eine Zahl *z* einen echten Teiler hat, dann besitzt sie auch einen echten Teiler kleiner oder gleich $\sqrt{z}$. Wenn man *keinen* echten Teiler kleiner oder gleich $\sqrt{z}$ gefunden hat, kann man die Suche einstellen, und *z* ist eine Primzahl.

Ist z. B. die Primzahl 23 mit der Wurzel 4,796... zu untersuchen, dann kann die Suche mit dem Teilerkandidaten 4 enden (größte ganze Zahl ≤ Wurzel). Es sind also nur die Teilerkandidaten 2, 3 und 4 zu untersuchen, sodass viel sinnloser Aufwand eingespart wird. Nach dem Teilerkandidaten 2 auch noch ein Vielfaches dieser Zahl (z. B. 4) zu untersuchen, ist natürlich nicht sehr intelligent. Daher trägt das Programm den Namen *Primitiv*.

Zur Berechnung der Quadratwurzel verwendet das Beispielprogramm die Methode **Sqrt()** aus der Klasse **Math**, über die man sich bei Bedarf in der BCL-Dokumentation informieren kann. Hinter der übertrieben aufwändig wirkenden Anweisung

```
mtk = (ulong) (Math.Sqrt(zahl) + 0.5);
```

stecken folgende Überlegungen:

- Der Laufindex `i` (Datentyp **ulong**) in der anschließenden **for**-Schleife wird wiederholt mit `mtk` verglichen. Damit dabei keine implizite Typumwandlung erforderlich ist, hat auch `mtk` den Typ **ulong** erhalten, sodass für das **Sqrt()** - Ergebnis eine explizite Typanpassung erforderlich ist. Dabei kann es *nicht* zum Ganzzahlüberlauf kommen, weil das **Sqrt()** - Argument eine **ulong**-Zahl ist.
- Ist eine ganze Zahl das Quadrat einer Primzahl, dann ist sie selbst keine Primzahl, sondern eine sogenannte *Sekundzahl*. Bei der Berechnung der Quadratwurzel aus einer Zahl im **ulong** - Wertebereich per Gleitkommaarithmetik kommt es in der Regel zu einer Abweichung vom mathematisch korrekten Ergebnis (vgl. Abschnitte 4.3.5 und 4.5.7.1). Bei einer Sekundzahl könnte das **Sqrt()** - Ergebnis knapp unter dem korrekten ganzzahligen Wert liegen, was bei der Wandlung in **ulong** (durch Abschneiden der Nachkommastellen) zu einem Fehler führen würde. Infolgedessen würde die Sekundzahl falsch als Primzahl erkannt. Um dies zu verhindern, wird vor der Wandlung in den Typ **ulong** der Wert 0,5 addiert und so eine Rundung erzwungen.

4.7.4 Übungsaufgaben zum Abschnitt 4.7

1) Bei einer Lotterie soll der folgende Gewinnplan gelten:

- Durch 13 teilbare Losnummern gewinnen 100 Euro.
- Losnummern, die nicht durch 13 teilbar sind, gewinnen immerhin noch einen Euro, wenn sie durch 7 teilbar sind.

Wird im folgenden Codesegment für Losnummern in der **int**-Variablen `losNr` der richtige Gewinn ermittelt?

```
if (losNr % 13 != 0)
   if (losNr % 7 == 0)
      Console.WriteLine("Das Los gewinnt einen Euro!");
else
   Console.WriteLine("Das Los gewinnt 100 Euro!");
```

2) Warum liefert dieses Programm widersprüchliche Auskünfte über die boolesche Variable `b`?

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  bool b = false;`
`  if (b = false)`
`  Console.WriteLine("b ist False");`
`  else`
`  Console.WriteLine("b ist True");`
`  Console.WriteLine("Kontr.ausg. von b: " + b);`
`  }`
`}`
	`b ist True`
	`Kontr.ausg. von b: False`

3) Erstellen Sie eine Variante des Primzahlen-Diagnoseprogramms aus dem Abschnitt 4.7.3.5, die ohne **break** und **continue** auskommt.

4) Wie oft wird die folgende **while**-Schleife ausgeführt?

```
using System;
class Prog {
    static void Main() {
      int i = 0;
      while (i < 100);
      {
         i++;
         Console.WriteLine(i);
      }
   }
}
```

5) Wegen der beschränkten Genauigkeit bei der Speicherung von Gleitkommazahlen (siehe Abschnitt 4.3.4) kann ein Rechner die **double**-Werte 1,0 und $1{,}0 + 2^{-i}$ ab einem bestimmten Exponenten *i* nicht mehr voneinander unterscheiden. Bestimmen Sie mit einem Testprogramm den größten ganzzahligen Exponenten *i*, für den man noch erhält:[1]

$$1{,}0 + 2^{-i} > 1{,}0$$

6) In dieser Aufgabe sollen Sie zwei Varianten von Euklids Algorithmus zur Bestimmung des größten gemeinsamen Teilers (GGT) zu zwei natürlichen Zahlen *u* und *v* implementieren und die Laufzeitunterschiede messen. Verwenden Sie als ersten Kandidaten den im Einführungsbeispiel zum Kürzen von Brüchen (Methode `Kuerze()` der Klasse `Bruch`) benutzten Algorithmus (siehe Abschnitt 1.3). Sein Problem besteht darin, dass bei stark unterschiedlichen Zahlen *u* und *v* sehr viele Subtraktions-Operationen erforderlich sind. In der meist benutzten Variante des Euklidischen Ver-

[1] Im Abschnitt 4.3.5.1, der allerdings für Programmiereinsteiger nur bedingt geeignet ist, finden sich Hintergrundinformationen zur Aufgabe.

fahrens wird dieses Problem vermieden, indem an Stelle der Subtraktion die Modulo-Operation zum Einsatz kommt, basierend auf dem folgenden Satz der mathematischen Zahlentheorie:

> Für zwei natürliche Zahlen *u* und *v* (mit *u* > *v*) ist der GGT gleich dem GGT von *u* und *u* % *v* (*u* modulo *v*).

Begründung (analog zu Abschnitt 1.4): Für natürliche Zahlen *u* und *v* mit *u* > *v* gilt:

x ist gemeinsamer Teiler von *u* und *v*
⇔
x ist gemeinsamer Teiler von *v* und *u* % *v*

Der GGT-Algorithmus per Modulo-Operation läuft für zwei natürliche Zahlen *u* und *v* (*u* ≥ *v* > 0) folgendermaßen ab:

> Es wird geprüft, ob *u* durch *v* restfrei teilbar ist.
> Trifft dies zu, ist *v* der GGT.
> Anderenfalls ersetzt man:
>
> > *u* durch *v*
> > *v* durch *u* % *v*
>
> Das Verfahren startet neu mit den kleineren Zahlen.

Die Voraussetzung *u* ≥ *v* ist nicht wesentlich, weil beim Start mit *u* < *v* der erste Algorithmusschritt die beiden Zahlen vertauscht.

Zur Messung des Zeitaufwands eignet sich z. B. ein Objekt der Klasse **Stopwatch** aus dem Namensraum **System.Diagnostics** (implementiert im Assembly **System.dll**). Ein Messvorgang wird ...

- mit der Methode **Start()** gestartet
  public void Start()
- und mit der Methode **Stop()** beendet
  public void Stop()

Für die Beispielwerte *u* = 999000999 und *v* = 36 liefern beide Euklid-Varianten sehr verschiedene Laufzeiten (CPU: Intel Core i3 550, Betriebssystem: Windows 10 (64 Bit)):

```
GGT-Bestimmung mit Euklid (Differenz)

Erste Zahl:     999000999

Zweite Zahl:    36

GGT:            9

Benöt. Zeit:    127 Millisek.
```

```
GGT-Bestimmung mit Euklid (Modulo)

Erste Zahl:   999000999

Zweite Zahl:  36

GGT:          9

Benöt. Zeit:  0 Millisek.
```

5 Klassen und Objekte

Software-Entwicklung mit C# besteht nach unserem bisherigen Kenntnisstand im Wesentlichen aus der Definition von **Klassen**, die aufgrund der vorangegangenen objektorientierten Analyse und Modellierung …

- als Baupläne für Objekte
- und/oder als Akteure

konzipiert werden. Wenn ein spezieller Akteur im Programm nur *einfach* benötigt wird, kann eine handelnde Klasse diese Rolle übernehmen.[1] Sind hingegen mehrere Individuen einer Gattung erforderlich (z. B. mehrere Brüche in einem Bruchrechnungsprogramm oder mehrere Fahrzeuge in der Speditionsverwaltung), dann ist eine Klasse mit Bauplancharakter gefragt.

Für eine Klasse und/oder ihre Objekte werden **Merkmale** (Felder), **Eigenschaften**, **Handlungskompetenzen** (Methoden) und weitere (bisher noch nicht behandelte) Bestandteile deklariert bzw. definiert. Diese werden als **Member** der Klasse bezeichnet (dt.: *Mitglieder*).

In den Methoden eines Programms werden Aufträge ausgeführt bzw. Algorithmen realisiert. Ein agierendes (eine Methode ausführendes) Objekt bzw. eine agierende Klasse muss nicht alles selbst erledigen, sondern kann vordefinierte (z. B. der BCL entstammende) oder im Programm definierte Klassen einspannen, z. B.:

- Eine Klasse aus der Standardbibliothek wird beauftragt:
  ```
  int az = Math.Abs(zaehler);
  ```
- Ein explizit im Programm erstelltes Objekt aus einer im Programm definierten Klasse wird beauftragt:
  ```
  Bruch b1 = new Bruch();
  b1.Frage();
  ```

Mit dem „Beauftragen“ eines Objekts oder einer Klasse bzw. mit dem „Zustellen einer Botschaft“ ist nichts anderes gemeint als ein Methodenaufruf.

Unsere vorläufige, auch im aktuellen Kapitel 5 zugrundeliegende Vorstellung von einem Computer-Programm lässt sich so beschreiben:

- Ein Programm besteht aus Klassen, die als Baupläne für Objekte und/oder als Akteure dienen.
- Die Akteure (Objekte und Klassen) haben jeweils einen Zustand (abgelegt in Feldern).
- Sie können Botschaften empfangen und senden (Methoden ausführen und aufrufen).

In der Hoffnung, dass die bisher präsentierten Eindrücke von der objektorientierten Programmierung (OOP) neugierig gemacht und nicht abgeschreckt haben, kommen wir nun zur systematischen Behandlung dieser Software-Technologie. Für die im Kapitel 1 speziell für größere Projekte empfohlene objektorientierte Analyse und Modellierung, z. B. mit Hilfe der Unified Modeling Language (UML), ist dabei leider keine Zeit vorhanden (siehe z. B. Balzert 2011; Booch et al. 2007).

---

[1] Eine nur einfach zu besetzende Rolle von einer Klasse übernehmen zu lassen, ist keinesfalls in jeder Situation eine ideale Design-Entscheidung und wird im Manuskript hauptsächlich der Einfachheit halber bevorzugt. In einer späteren Phase auf dem Weg zum professionellen Entwickler sollte man sich unbedingt mit dem sogenannten *Singleton-Pattern* beschäftigen (siehe z. B. Bloch 2018, S. 17ff). Dabei geht es um Klassen, von denen innerhalb einer Anwendung garantiert nur ein Objekt entsteht. Hier fungiert also ein Objekt statt einer Klasse als Solist, was etliche Vorteile bietet, z. B.:

- Die Adresse des Solo-Objekts kann an Methoden als Parameter übergeben werden.
- Die Vererbungstechnik der OOP wird besser unterstützt (inkl. Polymorphie, siehe Kapitel 7).
- Eine Singleton-Klasse kann Interfaces implementieren (siehe Kapitel 9).

5.1 Überblick, historische Wurzeln, Beispiel

5.1.1 Einige Kernideen und Vorzüge der OOP

Lahres & Rayman (2009, Kapitel 2) nennen in ihrem *Praxisbuch Objektorientierung* unter Berufung auf **Alan Kay**, der den Begriff *Objektorientierte Programmierung* geprägt und die objektorientierte Programmiersprache *Smalltalk* entwickelt hat, als unverzichtbare OOP-Grundelemente:

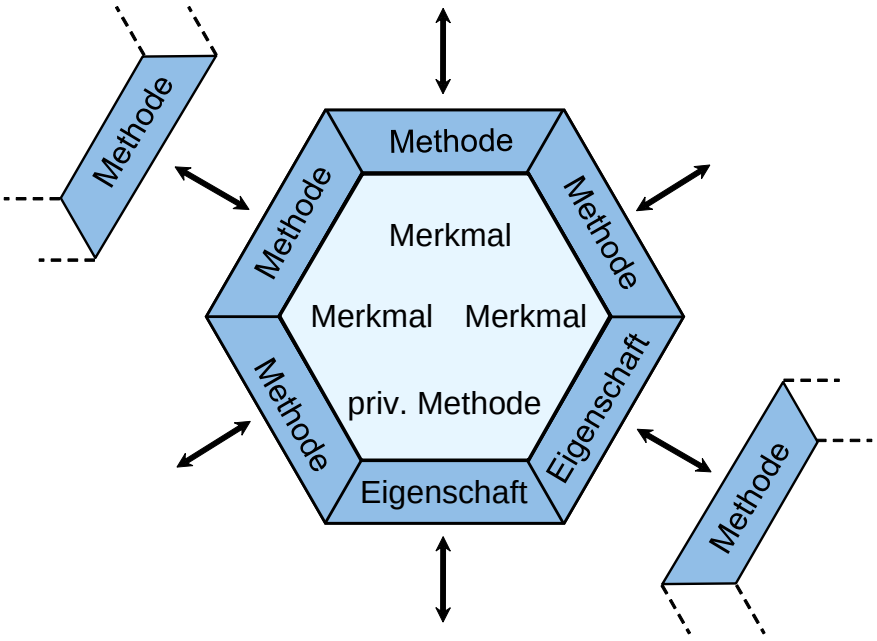
- **Datenkapselung**
  Eine Klasse erlaubt in der Regel fremden Klassen keinen direkten Zugriff auf ihre Zustandsdaten. So wird das Risiko für das Auftreten inkonsistenter Zustände reduziert. Außerdem kann der Klassendesigner Implementierungsdetails ohne Nebenwirkungen auf andere Klassen ändern. Mit der Datenkapselung haben wir uns schon im Kapitel 1 beschäftigt.
- **Vererbung**
  Aus einer vorhandenen Klasse lassen sich zur Lösung neuer Aufgaben spezialisierte Klassen ableiten, die alle Member der Basisklasse erben. Hier findet eine Wiederverwendung von Software ohne lästiges und fehleranfälliges Kopieren von Quellcode statt. Beim Design der abgeleiteten Klasse kann man sich darauf beschränken, neue Member zu definieren oder bei manchen Erbstücken (z. B. Methoden) Modifikationen zur Anpassung an die neue Aufgabe vorzunehmen.
- **Polymorphie**
  Über Referenzvariablen vom Typ einer Basisklasse lassen sich auch Objekte von abgeleiteten Klassen verwalten, wobei selbstverständlich nur solche Methoden aufgerufen werden dürfen, die schon in der Basisklasse definiert sind. Ist eine solche Methode in abgeleiteten Klassen unterschiedlich implementiert, führt jedes per Basisklassenreferenz angesprochene Objekt sein angepasstes Verhalten aus. Derselbe Methodenaufruf hat also unterschiedliche (polymorphe) Verhaltensweisen zur Folge. Welche Methode ausgeführt wird, entscheidet sich erst zur Laufzeit (späte Bindung). Dank Polymorphie ist eine lose Kopplung von Klassen möglich, und die Wiederverwendbarkeit von vorhandenem Code wird verbessert.

C# bietet sehr gute Voraussetzungen zur Nutzung dieser Konstruktionsprinzipien beim Entwurf von stabilen, wartungsfreundlichen, anpassungsfähigen und auf Wiederverwendung angelegten Software-Systemen, kann aber keinen Entwickler zur Realisation der Prinzipien zwingen.

5.1.1.1 Datenkapselung und Modularisierung

In der objektorientierten Programmierung (OOP) wird die vorher übliche Trennung von Daten und Operationen aufgegeben. Ein objektorientiertes Programm besteht aus **Klassen**, die durch **Felder (also Daten) und Methoden (also Operationen)** sowie weitere Bestandteile definiert sind. Wie Sie bereits aus dem Einleitungsbeispiel wissen, steht in C# eine **Eigenschaft** für ein Paar von Zugriffsmethoden zum Lesen bzw. Verändern eines Feldes.[1] Eine Klasse wird in der Regel ihre Felder gegenüber anderen Klassen verbergen (**Datenkapselung**, engl.: **information hiding**) und so vor ungeschickten Zugriffen schützen. Die meisten Methoden und Eigenschaften einer Klasse sind hingegen von außen ansprechbar und bilden ihre **Schnittstelle** bzw. ihr API. Dies kommt in der folgenden Abbildung zum Ausdruck, die Sie im Wesentlichen schon aus dem Abschnitt 1.2 kennen:

[1] Diese Darstellung ist etwas vereinfachend. Hinter einer Eigenschaft muss nicht unbedingt ein einzelnes Feld stehen.



Es kann aber auch *private Methoden* für den ausschließlich internen Gebrauch geben. Ebenso sind *öffentliche Felder* möglich, die damit zur Schnittstelle einer Klasse gehören. Solche Felder sind oft als *konstant* deklariert (siehe Abschnitt 5.2.5) und damit vor Veränderungen geschützt. Ein Beispiel ist das **double**-Feld **PI** für die trigonometrische Konstante π (≈ 3.1416) in der BCL-Klasse **Math**.

Klassen mit Datenkapselung realisieren besser als frühere Software-Technologien (siehe Abschnitt 5.1.2) das Prinzip der **Modularisierung**, das schon Julius Cäsar (100 v. Chr. - 44 v. Chr.) bei seiner beruflichen Tätigkeit als römischer Kaiser und Feldherr erfolgreich einsetzte (*Divide et impera!*).[1] Die Modularisierung ist ein probates, ja unverzichtbares Mittel der Software-Entwickler zur Bewältigung von umfangreichen Projekten.

Zugunsten einer häufigen und erfolgreichen Wiederverwendung sind Klassen mit hoher Komplexität (vielfältigen Aufgaben) und auch Methoden mit hoher Komplexität zu vermeiden. Als eine Leitlinie für den Entwurf von Klassen findet das von **Robert C. Martin**[2] erstmals formulierte Prinzip einer **einzigen Verantwortung** (engl.: ***S**ingle **R**esponsibility **P**rinciple*, SRP) bei den Vordenkern der objektorientierten Programmierung breite Zustimmung (siehe z. B. Lahres & Rayman 2009, Abschnitt 3.1). Multifunktionale Klassen tendieren zu stärkeren Abhängigkeiten von anderen Klassen, wobei die Wahrscheinlichkeit einer erfolgreichen Wiederverwendung sinkt. Ein negatives Beispiel wäre eine Klasse aus einem Personalverwaltungsprogramm, die sich sowohl um Gehaltsberechnungen als auch um die Interaktion mit dem Benutzer über eine grafische Bedienoberfläche kümmert.[3]

Aus der Datenkapselung und anderen Prinzipien der Modularisierung (z. B. Klassendesign nach dem Prinzip einer einzigen Verantwortung) ergeben sich gravierende Vorteile für die Software-Entwicklung:

---

[1] Deutsche Übersetzung: Teile und herrsche!

[2] Der als *Uncle Bob* bekannte Software-Berater und Autor erläutert auf der folgenden Webseite seine Vorstellungen von objektorientiertem Design: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

[3] In einem sehr kleinen Programm ist es angemessen, wenn eine einzige Klasse für die „Geschäftslogik“ und die Benutzerinteraktion zuständig ist (wie im Beispielprogramm aus dem Abschnitt 3.3.4).

- **Vermeidung von Fehlern**
  Direkte Schreibzugriffe auf die Felder einer Klasse bleiben den klasseneigenen Methoden vorbehalten, die vom Designer der Klasse sorgfältig entworfen wurden. Damit sollten Programmierfehler nur sehr selten auftreten. In unserer Beispielklasse `Bruch` haben wir verhindert, dass Anwender der Klasse den Nenner eines Bruchs auf den Wert 0 setzen. Fremde Klassen können einen Nenner einzig über die Eigenschaft `Nenner` verändern, die aber den Wert 0 nicht akzeptiert. Bei einer anderen Klasse mag es wichtig sein, dass für eine *Gruppe* von Feldern bei jeder Änderung gewisse Konsistenzbedingungen eingehalten werden.
- **Günstige Voraussetzungen für das Testen und die Fehlerbereinigung**
  Treten in einem Programm trotz Datenkapselung pathologische Variablenausprägungen auf, ist die Ursache relativ leicht aufzuklären, weil nur wenige Methoden verantwortlich sein können. Zur Sicherstellung wichtiger Bedingungen kann es sinnvoll sein, auch Feldzugriffe durch *klasseneigene* Methoden über die zuständigen Eigenschaften vorzunehmen. Bei der Software-Entwicklung im professionellen Umfeld spielt das systematische Testen eines Programms (**Unit Testing**) eine entscheidende Rolle. Ein objektorientiertes Softwaresystem mit Datenkapselung und guter Modularisierung bietet günstige Voraussetzungen für eine möglichst umfassende Testung.
- **Innovationsoffenheit durch gekapselte Details der Klassenimplementation**
  Verborgene Details einer Klassenimplementation kann der Designer ändern, ohne die Kooperation mit anderen Klassen zu gefährden.
- **Produktivität durch wiederholt und bequem verwendbare Klassen**
  Selbständig agierende Klassen, die ein Problem ohne überflüssige Abhängigkeiten von anderen Klassen lösen, sind potenziell in vielen Projekten zu gebrauchen (Wiederverwendbarkeit). Wer als Programmierer eine Klasse verwendet, braucht sich um deren inneren Aufbau nicht zu kümmern, sodass neben dem Fehlerrisiko auch der Einarbeitungsaufwand sinkt. Man kann z. B. in einem GUI-Programm einen kompletten Rich-Text-Editor über eine Klasse aus der Standardbibliothek integrieren, ohne wissen zu müssen, wie Text und Textauszeichnungen intern verwaltet werden.
- **Erfolgreiche Teamarbeit durch abgeschottete Verantwortungsbereiche**
  In großen Projekten können mehrere Programmierer nach der gemeinsamen Definition von Schnittstellen relativ unabhängig an verschiedenen Klassen arbeiten.

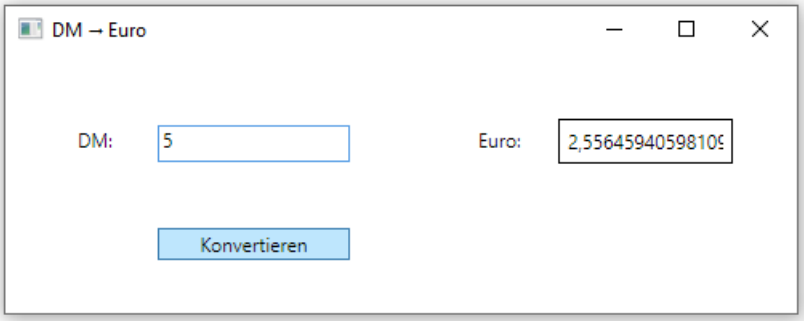
Durch die objektorientierte Programmierung werden auf vielfältige Weise **Kosten reduziert**:

- Vermeidung bzw. schnelle Aufklärung von Programmierfehlern
- gute Chancen für die Wiederverwendung von Software
- gute Voraussetzungen für die Kooperation in Teams

5.1.1.2 Vererbung

Zu den Vorzügen der „super-modularen“ Klassenkonzeption gesellt sich in der OOP ein Vererbungsverfahren, das beste Voraussetzungen für die Erweiterung von Software-Systemen bei rationeller **Wiederverwendung** der bisherigen Code-Basis schafft: Bei der Definition einer *abgeleiteten Klasse* können alle Merkmale und Handlungskompetenzen (Methoden, Eigenschaften) der *Basisklasse* übernommen werden. Es ist also leicht möglich, ein Software-System um neue Klassen mit speziellen Leistungen zu erweitern. Durch systematische Anwendung des Vererbungsprinzips entstehen mächtige Klassenhierarchien, die in zahlreichen Projekten einsetzbar sind. Neben der direkten Nutzung vorhandener Klassen (über erzeugte Objekte oder statische Methoden) bietet die OOP mit der Vererbungstechnik eine weitere Möglichkeit zur Wiederverwendung von Software.

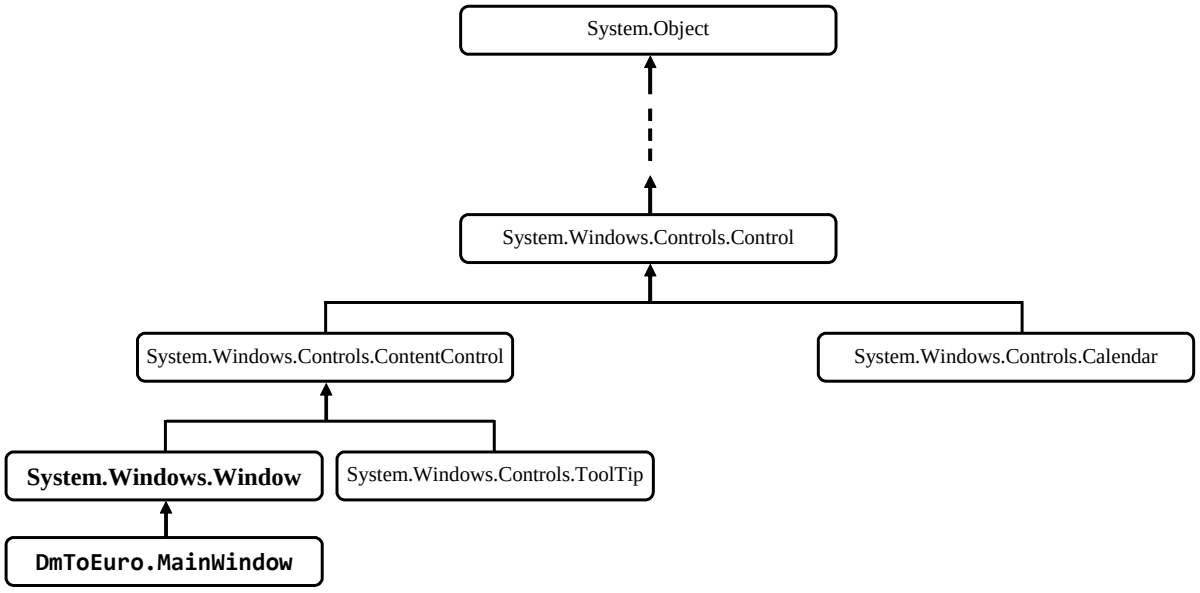
Bei dem im Abschnitt 3.3.4 erstellten Währungskonverter mit grafischer Benutzerschnittstelle



haben wir mit Hilfe der Entwicklungsumgebung die Klasse `MainWindow` definiert als Ableitung der Klasse **Window**:[1]

```
public partial class MainWindow : Window
{
   public MainWindow()
   {
      InitializeComponent();
   }

   private void Button_Click(object sender, RoutedEventArgs e)
   {
      double betrag = Convert.ToDouble(eingabe.Text);
      ausgabe.Content = Convert.ToString(betrag / 1.95583);
   }
}
```

Die Basisklasse **Window** ist selbst Bestandteil eines komplexen Stammbaums, von dem anschließend nur ein kleiner Ausschnitt zu sehen ist:



Im .NET - Typsystem wird das Vererbungsprinzip sogar auf die Spitze getrieben: Alle Klassen (und auch die Strukturen, siehe unten) stammen von der Urahnklasse **Object** ab, die an der Spitze des

---

[1] Die Klasse ist als **partial** gekennzeichnet, weil ihre Implementation auf zwei Dateien verteilt ist, um die Kooperation zwischen Entwicklungsumgebung und Programmierer zu erleichtern:

- Die erste Quellcodedatei ist für den Programmierer zugänglich.
- Den restlichen Quellcode verwaltet die Entwicklungsumgebung in einer versteckten Datei.

Den Quellcode einer Klasse auf zwei Dateien aufzuteilen, ist im Normalfall nicht sinnvoll.

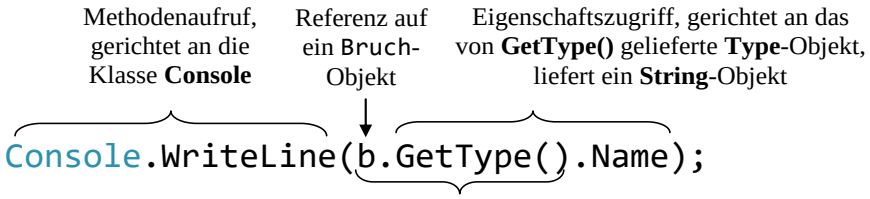
hierarchischen .NET - Klassensystems steht, das man seiner Universalität wegen als **Common Type System** (CTS) bezeichnet.

Weil sich im Handlungsrepertoire der Urahnklasse u. a. auch die Methode **GetType()** befindet, kann man beliebige .NET-Objekte und -Strukturinstanzen (siehe unten) nach ihrem Datentyp fragen. Im folgenden Programm wird ein `Bruch`-Objekt (vgl. Abschnitt 1) um die entsprechende Auskunft gebeten:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  Bruch b = new Bruch();`
`  Console.WriteLine(b.GetType().Name);`
`  }`
`}`
	`Bruch`

Von **GetType()** erhält man als Rückgabewert eine Referenz auf ein Objekt der Klasse **Type**. Über diese Referenz wird das **Type**-Objekt gebeten, den Wert seiner Eigenschaft **Name** mitzuteilen. Diese Zeichenfolge mit dem Typnamen (ein Objekt der Klasse **String**[1]) bildet schließlich den Parameter des **WriteLine()** - Aufrufs und landet auf der Konsole. In unserem Kursstadium ist es angemessen, die komplexe Anweisung unter Beteiligung von vier Klassen (**Console**, `Bruch`, **Type**, **String**), zwei Methoden (**WriteLine()**, **GetType()**), einer Eigenschaft (**Name**), einer expliziten Referenzvariablen (`b`) und einer impliziten Referenz (**GetType()** - Rückgabewert) genau zu erläutern:[2]



Durch die technischen Details darf nicht der Blick auf das wesentliche Thema des aktuellen Abschnitts verstellt werden: Eine abgeleitete Klasse erbt die Merkmale und Handlungskompetenzen ihrer Basisklasse. Wenn diese Basisklasse ihrerseits abgeleitet wurde, kommen indirekt erworbene Erbstücke hinzu. Die als Beispiel betrachtete Klasse `Bruch` stammt direkt von der Klasse **Object** ab, und ihre Instanzen beherrschen dank Vererbung die Methode **GetType()**, obwohl in der `Bruch`-Klassendefinition davon nichts zu sehen ist.

5.1.1.3 Polymorphie

Obwohl in unseren bisherigen Beispielen die Polymorphie noch nicht zum Einsatz kam, soll doch versucht werden, die Kernidee hinter diesem Begriff schon jetzt zu vermitteln. In diesem Abschnitt sind einige Vorgriffe auf später ausführlich behandelte Begriffe erforderlich. Wer sich jetzt noch nicht stark für den Begriff der Polymorphie interessiert, kann den Abschnitt ohne Risiko für den weiteren Kursverlauf auslassen.

---

[1] Eine Besonderheit der Klasse **String** ist der aus Bequemlichkeitsgründen vom Compiler unterstützte Aliasname **string** (mit kleinem Anfangsbuchstaben).

[2] Der **Name**-Eigenschaftszugriff ist im Beispiel eigentlich nicht erforderlich, weil die Methode **WriteLine()** auch mit Objekten (z. B. aus der Klasse **Type**) umzugehen weiß und deren garantiert vorhandene, weil bereits in der Urahnklasse **Object** definierte, Methode **ToString()** nutzt, um sich ein ausgabefähiges **String**-Objekt zu besorgen.

Beim Klassendesign ist generell das **Open-Closed - Prinzip** beachtenswert:[1]

- Eine Klasse soll **offen** sein für die Verwendung zur Lösung neuer Aufgaben.
- Dabei darf es nicht erforderlich werden, vorhandenen Code zu verändern. Er soll **abgeschlossen** bleiben, möglichst für immer. In ungünstigen Fällen zieht eine Änderung am Quellcode weitere nach sich, sodass eine Kaskade von Anpassungen (eventuell unter Beteiligung von anderen Klassen) resultiert. Dadurch verursacht die Anpassung einer Klasse an neue Aufgaben hohe Kosten und oft ein fehlerhaftes Ergebnis.

Einen exzellenten Beitrag zur Erstellung von änderungsoffenem und doch abgeschlossenem Code leistet schon die Vererbungstechnik der OOP. Zur Modellierung einer neuen, spezialisierten Rolle kann man oft auf eine Basisklasse zurückgreifen und muss nur die zusätzlichen Merkmale und/oder Verhaltenskompetenzen ergänzen.

In C# können über eine Referenzvariable Objekte vom deklarierten Typ *und von jedem abgeleiteten Typ* angesprochen werden. In einer abgeleiteten Klasse können nicht nur zusätzliche Methoden erstellt, sondern auch geerbte überschrieben werden, um das Verhalten an spezielle Einsatzbereiche anzupassen. Ergeht ein Methodenaufruf an Objekte aus verschiedenen abgeleiteten Klassen, die jeweils die Methode überschrieben haben, unter Verwendung von Basisklassenreferenzen, dann zeigen die Objekte ihr artgerechtes Verhalten. Obwohl alle Objekte mit einer Referenz vom selben Basisklassentyp angesprochen werden und denselben Methodenaufruf erhalten, agieren sie unterschiedlich. Welche Methode tatsächlich ausgeführt wird, entscheidet sich erst zur Laufzeit (späte Bindung). Genau in dieser Situation spricht man von *Polymorphie*, und diese Software-Technik leistet einen wichtigen Beitrag zur Realisation des Open-Closed - Prinzips.

Wird z. B. in einer Klasse zur Verwaltung von geometrischen Objekten eine Referenzvariable vom relativ allgemeinen Typ `Figur` deklariert und beim Aufruf der Methode `MeldeInhalt()` verwendet, dann führt das angesprochene Objekt, das bei einem konkreten Programmeinsatz z. B. aus der abgeleiteten Klasse `Kreis` oder `Rechteck` stammt, seine spezifischen Berechnungen durch. Die Klasse zur Verwaltung von geometrischen Objekten kann ohne Quellcodeänderungen mit beliebigen, eventuell sehr viel später definierten `Figur`-Ableitungen kooperieren.

Weil in der allgemeinen Klasse `Figur` keine Inhaltsberechnungsmethode realisiert werden kann, wird hier die Methode `MeldeInhalt()` zwar deklariert, aber nicht implementiert, sodass eine sogenannte *abstrakte* Methode entsteht. Enthält eine Klasse mindestens eine abstrakte Methode, ist sie ihrerseits abstrakt und kann nicht zum Erzeugen von Objekten genutzt werden. Eine abstrakte Klasse ist aber gleichwohl als Datentyp erlaubt und spielt eine wichtige Rolle bei der Realisation von Polymorphie.[2]

Dank Polymorphie ist eine lose Kopplung von Klassen möglich, und die Wiederverwendbarkeit von vorhandenem Code wird verbessert. Um die Offenheit für neue Aufgaben zu ermöglichen, verwendet man beim Klassendesign für Felder und Methodenparameter mit Referenztyp einen möglichst allgemeinen Datentyp, der die benötigten Verhaltenskompetenzen vorschreibt, aber keine darüberhinausgehende Einschränkung enthält.

---

[1] Das Open-Closed - Prinzip wird von Robert C. Martin (*Uncle Bob*) in einem Text erläutert, der über folgende Web-Adresse zu beziehen ist: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

[2] Neben den abstrakten Klassen, die mindestens *eine* abstrakte Methode (Definitionskopf ohne Implementation) enthalten, spielen bei der Polymorphie auch die sogenannten *Schnittstellen* eine wichtige Rolle als Datentypen für ein veränderungsoffenes Design. Eine Schnittstelle kann näherungsweise als Klasse mit *ausschließlich* abstrakten Methoden charakterisiert werden. Abstrakte Klassen und Schnittstellen (Interfaces) werden später ausführlich behandelt.

Dank Vererbung und Polymorphie kann objektorientierte Software **anpassungs- und erweiterungsfähig** bei weitgehend fixiertem Bestands-Code, also unter Beachtung des Open-Closed - Prinzips, gestaltet werden.

5.1.1.4 Realitätsnahe Modellierung

Klassen sind nicht nur ideale Bausteine für die rationelle Konstruktion von Software-Systemen, sondern sie erlauben auch eine gute Modellierung des Anwendungsbereichs. In der zentralen Projektphase der objektorientierten Analyse und Modellierung sprechen Software-Entwickler und Auftraggeber dieselbe Sprache, sodass Kommunikationsprobleme weitgehend vermieden werden.

Neben den Klassen zur Modellierung von Akteuren oder Ereignissen des realen Anwendungsbereichs sind bei einer typischen Anwendung aber auch zahlreiche Klassen beteiligt, die Akteure oder Ereignisse aus der Welt des Computers repräsentieren (z. B. Bildschirmfenster, Ereignisse).

5.1.2 Strukturierte Programmierung und OOP

In vielen älteren Programmiersprachen (z. B. C, Fortran, Pascal) sind zur Strukturierung von Programmen zwei Techniken verfügbar, die in weiterentwickelter Form auch bei der OOP genutzt werden:

- **Unterprogramme**
  Man zerlegt ein Gesamtproblem in mehrere Teilprobleme, die jeweils in einem eigenen *Unterprogramm* gelöst werden. Wird die von einem Unterprogramm erbrachte Leistung wiederholt (an verschiedenen Stellen eines Programms) benötigt, muss jeweils nur ein Aufruf mit dem Namen des Unterprogramms und passenden Parametern eingefügt werden. Durch diese Strukturierung ergeben sich kompakte und übersichtliche Programme, die leicht erstellt, analysiert, korrigiert und erweitert werden können. Praktisch alle älteren Programmiersprachen unterstützen solche Unterprogramme (Subroutinen, Funktionen, Prozeduren), und meist stehen auch umfangreiche Bibliotheken mit fertigen Unterprogrammen für diverse Standardaufgaben zur Verfügung. Beim Einsatz einer Unterprogrammsammlung klassischer Art muss der Programmierer passende Daten bereitstellen, auf die dann vorgefertigte Routinen losgelassen werden. Der Programmierer hat also seine Datensammlung *und* das Arsenal der verfügbaren Unterprogramme (aus fremden Quellen oder selbst erstellt) zu verwalten und zu koordinieren.
- **Problemadäquate Datentypen**
  Zusammengehörige Daten unter *einem* Variablennamen ansprechen zu können, vereinfacht das Programmieren erheblich. Über das Schlüsselwort **struct** der Programmiersprache C oder das analoge Schlüsselwort **record** der Programmiersprache Pascal lassen sich problemadäquate Datentypen mit mehreren Bestandteilen definieren, die jeweils einen beliebigen, bereits bekannten Typ haben dürfen. So eignet sich etwa für ein Programm zur Adressenverwaltung ein neu definierter Datentyp mit Variablen für Name, Vorname, Telefonnummer etc. Alle Adressinformationen zu einer Person lassen sich dann in *einer* Variablen vom selbst definierten Typ speichern. Dies vereinfacht z. B. das Lesen, Kopieren oder Schreiben solcher Daten.

Die problemadäquaten Datentypen der älteren Programmiersprachen werden in der OOP durch *Klassen* ersetzt, wobei diese Datentypen nicht nur durch eine Anzahl von *Merkmalen* (Feldern) beliebigen Typs charakterisiert sind, sondern auch *Handlungskompetenzen* (Methoden, Eigenschaften) besitzen, welche die Aufgaben der Funktionen bzw. Prozeduren der älteren Programmiersprachen übernehmen.

Im Vergleich zur strukturierten Programmierung bietet die OOP u. a. folgende Vorteile:

- Optimierte Modularisierung mit Zugriffsschutz
  Die Daten sind sicher in Objekten gekapselt, während sie bei traditionellen Programmiersprachen entweder als globale Variablen allen Missgriffen ausgeliefert sind oder zwischen Unterprogrammen „wandern“ (Goll & Heinisch 2016), was bei Fehlern zu einer aufwändigen Suche entlang der Verarbeitungskette führen kann.
- Gute Voraussetzungen für die Teamarbeit
  Durch die optimierte Modularisierung wird die (vor allem in großen Projekten wichtige) Kooperation in Entwicklungs-Teams erleichtert.
- Rationelle (Weiter-)Entwicklung von Software nach dem Open-Closed - Prinzip durch Vererbung und Polymorphie
- Bessere Abbildung des Anwendungsbereichs
  Das erleichtert die Kommunikation zwischen dem Auftraggeber bzw. Anwender einerseits und dem Software-Architekten bzw. -Entwickler andererseits.
- Mehr Komfort für Bibliotheksbenutzer
  Jede rationelle Software-Produktion greift in hohem Maß auf Bibliotheken mit bereits vorhandenen Lösungen zurück. Dabei sind die Klassenbibliotheken der OOP einfacher zu verwenden als klassische Funktionsbibliotheken.
- Erleichterte Wiederverwendung
  Die komfortable Nutzung von Lösungsbibliotheken sowie die rationelle Weiterentwicklung von Software durch Vererbung und Polymorphie führen zu einer erleichterten Wiederverwendung von vorhandener Software.

Dass objektorientierte Programmiersprachen im Vergleich zu ihren strukturierten Vorgängern etwas mehr Speicherplatz und CPU-Leistung verbrauchen, spielt schon lange keine Rolle mehr.

5.1.3 Auf-`Bruch` zu echter Klasse

In den Beispielprogrammen von Kapitel 4 wurde mit der Klassendefinition lediglich eine in C# unausweichliche formale Anforderung an Programme erfüllt. Die im Kapitel 1 vorgestellte Klasse `Bruch` realisiert hingegen wichtige Prinzipien der objektorientierten Programmierung. Sie wird nun wieder aufgegriffen und in verschiedenen Varianten bzw. Ausbaustufen als Beispiel verwendet. Auf der Klasse `Bruch` basierende Programme sollen Schüler beim Erlernen der Bruchrechnung unterstützen. Eine objektorientierte Analyse der Problemstellung hat ergeben, dass in elementaren Bruchrechnungsprogrammen lediglich eine Klasse zur Repräsentation von Brüchen benötigt wird. Später sind weitere Klassen zu ergänzen (z. B. Aufgabe, Übungsaufgabe, Testaufgabe, Schüler, Lernepisode, Testepisode, Fehler).

Wir nehmen nun bei der `Bruch`-Klassendefinition im Vergleich zur Variante im Kapitel 1 einige Verbesserungen vor:

- Als zusätzliches Feld erhält jeder Bruch ein `etikett` vom Datentyp der Klasse **String**. Damit wird eine beschreibende Zeichenfolge verwaltet, die z. B. beim Aufruf der Methode `Zeige()` zusätzlich zu den Feldern `zaehler` und `nenner` auf dem Bildschirm erscheint. Objekte der erweiterten Klasse `Bruch` besitzen also auch eine Instanzvariable mit *Referenztyp* (neben den Feldern `zaehler` und `nenner` vom elementaren Typ **int**).
- Weil die Klasse `Bruch` ihre Merkmale kapselt, also fremden Klassen keine *direkten* Zugriffe erlaubt, stellt sie auch für das Feld `etikett` eine Eigenschaft namens `Etikett` (mit großem Anfangsbuchstaben) für das Lesen und das (kontrollierte) Verändern des Felds zur Verfügung.

- Wir erlauben uns einen erneuten Vorgriff auf die später noch ausführlich zu diskutierende Ausnahmebehandlung per **try-catch** - Anweisung, um in der `Bruch`-Methode `Frage()` sinnvoll auf fehlerhafte Benutzereingaben reagieren zu können. Die kritischen Aufrufe der **Convert**-Methode **ToInt32()** finden nun innerhalb eines **try**-Blocks statt. Bei einem Ausnahmefehler aufgrund einer irregulären Eingabe wird daher *nicht* mehr das Programm beendet, sondern der zugehörige **catch**-Block ausgeführt. Damit die Methode `Frage()` den Aufrufer über eine reibungslose oder verpatzte Ausführung informieren kann, wechselt der Rückgabetyp von **void** zu **bool**. Mit der **return**-Anweisung in der verbesserten `Frage()` - Variante wird nach einer erfolgreichen Ausführung der Wert **true** bzw. nach dem Auftreten einer Ausnahme der Wert **false** zurückgemeldet.
- In der Methode `Kuerze()` wird die performante Modulo-Variante von Euklids Algorithmus zur Bestimmung des größten gemeinsamen Teilers von zwei ganzen Zahlen verwendet (siehe Übungsaufgabe im Abschnitt 4.7.4).

Im folgenden Quellcode der erweiterten Klasse `Bruch` sind die unveränderten Methoden bzw. Eigenschaften gekürzt wiedergegeben:

```
using System;
public class Bruch {
   int zaehler,             // zaehler wird automatisch mit 0 initialisiert
       nenner = 1;
   string etikett = "";     // die Ref.typ-Init. auf null wird ersetzt, siehe Text

   public int Zaehler {
      . . .
   }

   public int Nenner {
      . . .
   }

   public string Etikett {
      get {
         return etikett;
      }
      set {
         if (value.Length <= 40)
            etikett = value;
         else
            etikett = value.Substring(0, 40);
      }
   }

   public void Kuerze() {
      // größten gemeinsamen Teiler mit Euklids Algorithmus bestimmen
      // (performante Variante mit Modulo-Operator)
      if (zaehler != 0) {
         int rest;
         int ggt = Math.Abs(zaehler);
         int divisor = Math.Abs(nenner);
         do {
            rest = ggt % divisor;
            ggt = divisor;
            divisor = rest;
         } while (rest > 0);
         zaehler /= ggt;
         nenner /= ggt;
      } else
         nenner = 1;
   }
```

```
    public void Addiere(Bruch b) {
       . . .
    }

    public bool Frage() {
       try {
          Console.Write("Zähler: ");
          int z  = Convert.ToInt32(Console.ReadLine());
          Console.Write("Nenner : ");
          int n = Convert.ToInt32(Console.ReadLine());
          Zaehler = z;
          Nenner = n;
          return true;
       } catch {
          return false;
       }
    }

    public void Zeige() {
       string luecke = "  ";
       string glz;
       for (int i = 1; i <= etikett.Length; i++)
          luecke += " ";
       if (etikett.Length == 0)
          glz = "";
       else {
          glz = " = ";
          luecke += "   ";
       }
       Console.WriteLine($" {luecke}{zaehler}\n {etikett}{glz}-----\n {luecke}{nenner}\n");
    }
}
```

Im Unterschied zur Präsentation im Kapitel 1 wird die Definition der Klasse `Bruch` anschließend gründlich erläutert. Dabei machen die im Abschnitt 5.2 behandelten Instanzvariablen (Felder) relativ wenig Mühe, weil wir viele Details schon von den *lokalen* Variablen her kennen. Bei den Methoden gibt es mehr Neues zu lernen, sodass wir uns im Abschnitt 5.3 auf elementare Themen beschränken und später noch wichtige Ergänzungen vornehmen.

Jede .NET - Anwendung benötigt bekanntlich eine Startklasse mit statischer **Main()** - Methode, die von der CLR (*Common Language Runtime*) beim Programmstart aufgerufen wird. Für die bei diversen Demonstrationen in den folgenden Abschnitten verwendeten Startklassen (mit jeweils spezieller Implementation) werden wir generell den Namen `Bruchrechnung` verwenden, z. B.:

```
using System;
class Bruchrechnung {
   static void Main() {
      Bruch b = new Bruch();
      b.Frage();
      b.Kuerze();
      b.Etikett = "b";
      b.Zeige();
   }
}
```

Im gerade präsentierten Beispielprogramm zum Kürzen von Brüchen wird ein Objekt aus der Klasse `Bruch` erzeugt und mit Aufträgen versorgt.

Die Instanzvariablen `zaehler` und `nenner` der Klasse `Bruch` haben bei der Renovierung den Datentyp **int** behalten und sind daher nach wie vor mit einem potentiellen Überlaufproblem (vgl. Abschnitt 4.6.1) belastet, das im folgenden Programm demonstriert wird:

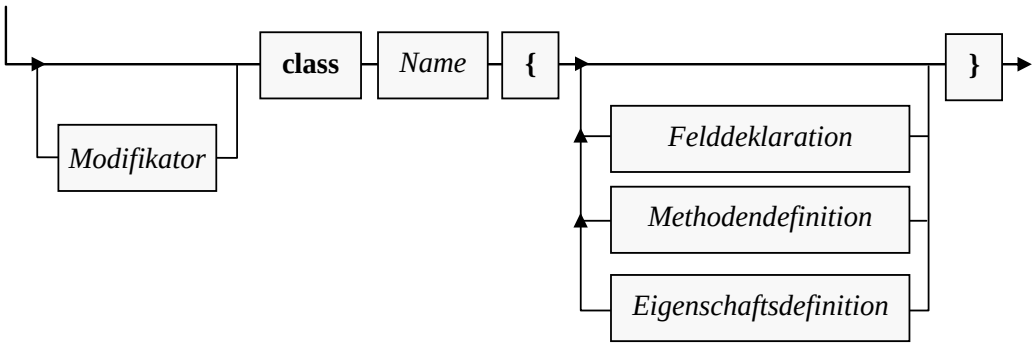
Quellcode	Ausgabe

`using System;`
`class Bruchrechnung {`
`  public static void Main(String[] args) {`
`  Bruch b1 = new Bruch(), b2 = new Bruch();`
`  b1.Zaehler = 2147483647; b1.Nenner = 1;`
`  b2.Zaehler = 2; b2.Nenner = 1;`
`  b1.Addiere(b2);`
`  b1.Zeige();`
`  }`
`}`
	`  -2147483647`
	`-----`
	`  1`

Wie das Problem zu beheben ist, wurde im Abschnitt 4.6.1 beschrieben.

Wir arbeiten im Wesentlichen weiterhin mit dem aus dem Abschnitt 4.1.2.1 bekannten Syntaxdiagramm zur Klassendefinition, das aus didaktischen Gründen einige Vereinfachungen enthält:

Klassendefinition



Am Ende der Klassendefinition (hinter der schließenden geschweiften Klammer des Rumpfs) ist ein Semikolon erlaubt, aber nicht erforderlich.

Bemerkungen zum Kopf einer Klassendefinition:

- Die Klasse `Bruch` ist als **public** definiert, damit sie uneingeschränkt von anderen Klassen (aus beliebigen Assemblies) genutzt werden kann. Weil bei der startfähigen Klasse `Bruchrechnung` eine solche Nutzung nicht in Frage kommt, wird hier auf den (zum Starten durch die CLR nicht erforderlichen) Zugriffsmodifikator **public** verzichtet.
- Klassennamen beginnen einer allgemein akzeptierten C# - Konvention folgend mit einem Großbuchstaben. Besteht ein Name aus mehreren Wörtern (z. B. `ArithmeticException`), dann schreibt man der besseren Lesbarkeit wegen die Anfangsbuchstaben aller Wörter groß (Pascal Casing, siehe Abschnitt 4.1.5).

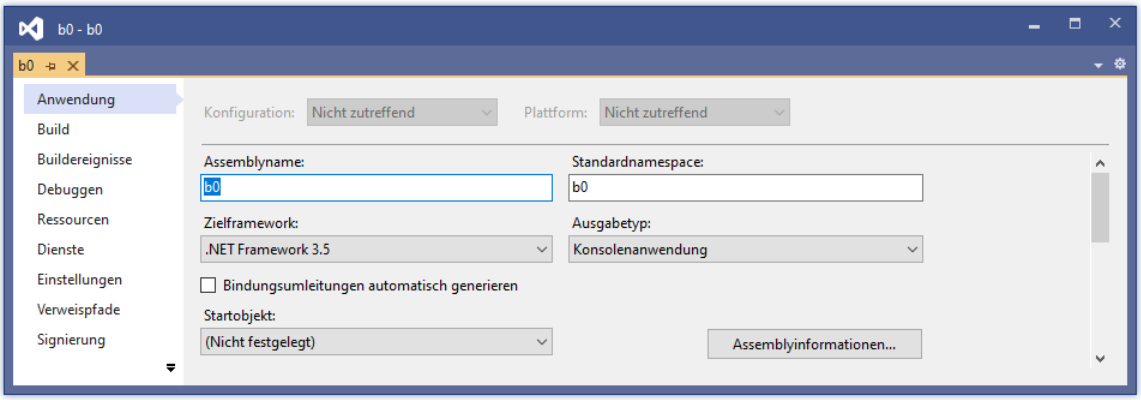
Zur Dateiverwaltung wird vorgeschlagen:

- Die Klassendefinitionen (z. B. `Bruch` und `Bruchrechnung`) sollten jeweils in einer eigenen Datei gespeichert werden.
- Den Namen dieser Datei sollte man aus dem Klassennamen durch Anhängen der Erweiterung **.cs** bilden.

Beim gemeinsamen Übersetzen der Quellcode-Dateien **Bruch.cs** und **Bruchrechnung.cs** entsteht *ein* Exe-Assembly, dessen Namen man im Visual Studio nach dem Menübefehl

Projekt > Eigenschaften

ändern kann, wenn die vom Assistenten für neue Projekte vergebene Voreinstellung nicht (mehr) gefällt, z. B.:



Im Beispiel entsteht das ausführbare Assembly **b0.exe**.

5.2 Instanzvariablen (Felder)

Die Instanzvariablen (bzw. -felder) einer Klasse besitzen viele Gemeinsamkeiten mit den *lokalen* Variablen, die wir im Kapitel 4 über elementare Sprachelemente ausführlich behandelt haben, doch gibt es auch wichtige Unterschiede, die im Mittelpunkt des aktuellen Abschnitts stehen. Unsere Klasse `Bruch` besitzt nach der Erweiterung um ein beschreibendes Etikett die folgenden Instanzvariablen:

- `zaehler` (Datentyp **int**)
- `nenner` (Datentyp **int**)
- `etikett` (Datentyp **String**)

Zu den beiden Feldern `zaehler` und `nenner` vom elementaren Datentyp **int** ist das Feld `etikett` mit dem Referenzdatentyp **String** dazugekommen. Jedes nach dem `Bruch`-Bauplan geschaffene Objekt erhält seine eigene Ausstattung mit diesen Variablen.

5.2.1 Sichtbarkeitsbereich, Existenz und Ablage im Hauptspeicher

Von den lokalen Variablen einer Methode unterscheiden sich die Instanzvariablen (Felder) einer Klasse vor allem bei der *Zuordnung* (vgl. Abschnitt 4.3.3):

- lokale Variablen gehören zu einer *Methode* (oder *Eigenschaft*)
- Instanzvariablen gehören zu einem *Objekt*

Daraus ergeben sich gravierende Unterschiede in Bezug auf den Sichtbarkeitsbereich, die Lebensdauer und die Ablage im Hauptspeicher:

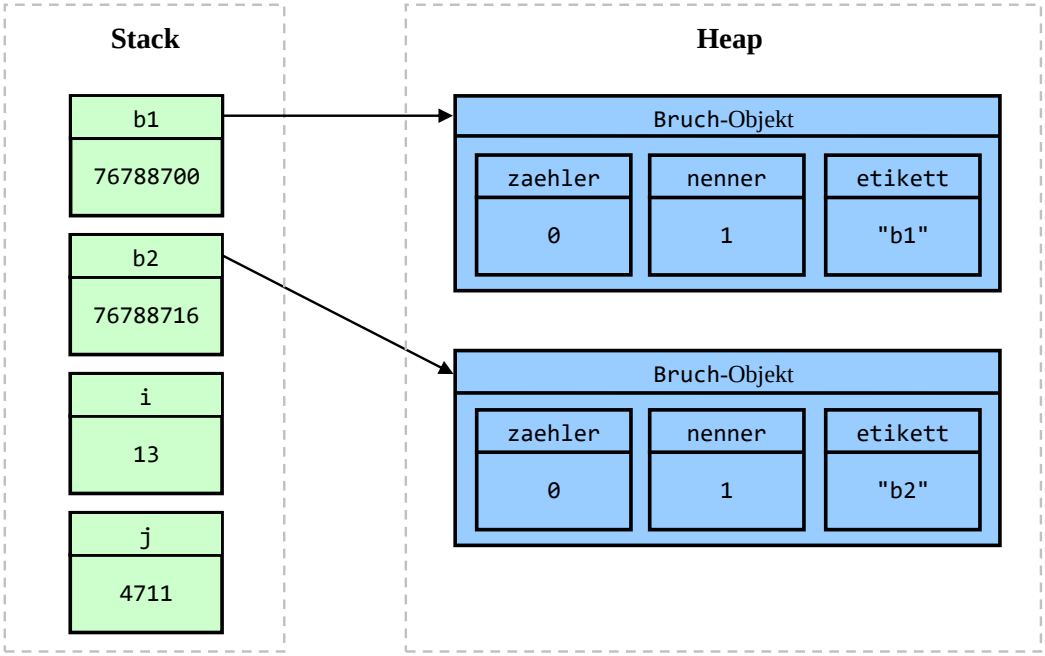
	lokale Variable	Instanzvariable
Sichtbarkeit, Gültigkeit	Eine lokale Variable ist nur in ihrer eigenen Methode sichtbar (gültig). Nach der Deklarationsanweisung bleibt sie ansprechbar bis zur schließenden Klammer des Blocks, in dem sie deklariert worden ist. Ein eingeschachtelter Block gehört zum Sichtbarkeitsbereich des umgebenden Blocks.	Die Instanzvariablen eines existenten Objekts sind in einer Methode sichtbar, wenn ... • der Zugriff erlaubt ist (siehe Abschnitt 5.11) • eine Referenz zum Objekt vorhanden ist Instanzvariablen werden in klasseneigenen Instanzmethoden durch gleichnamige lokale Variablen überlagert, können in dieser Situation jedoch über das vorgeschaltete Schlüsselwort **this** weiter angesprochen werden (siehe Abschnitt 5.2.4).
Lebensdauer	Sie existiert nur während der Ausführung der zugehörigen Methode.	Für jedes neue Objekt wird ein Satz mit allen Instanzvariablen seiner Klasse erzeugt. Die Instanzvariablen existieren bis zum Ableben des Objekts. Ein Objekt wird zur Entsorgung freigegeben, sobald keine Referenz auf das Objekt mehr im Programm vorhanden ist.
Ablage im Speicher	Sie wird auf dem sogenannten **Stack** (deutsch: *Stapel*) abgelegt. Dieses Segment des programmeigenen Speichers dient zur Durchführung von Methodenaufrufen.	Die Objekte landen mit ihren Instanzvariablen in einem Bereich des programmeigenen Speichers, der als **Heap** (deutsch: *Haufen*) bezeichnet wird.

Während die folgende **Main()** - Methode

```
class Bruchrechnung {
  static void Main() {
    Bruch b1 = new Bruch(), b2 = new Bruch();
    int i = 13, j = 4711;
    b1.Etikett = "b1";
    b2.Etikett = "b2";
    . . .
  }
}
```

ausgeführt wird, befinden sich auf dem Stack die lokalen Variablen `b1`, `b2`, `i` und `j`. Die beiden `Bruch`-Referenzvariablen (`b1`, `b2`) zeigen jeweils auf ein `Bruch`-Objekt auf dem Heap, das einen kompletten Satz der `Bruch`-Instanzvariablen besitzt:[1]

[1] Hier wird aus didaktischen Gründen ein wenig gemogelt: Die beiden Etiketten sind selbst Objekte und liegen „neben“ den `Bruch`-Objekten auf dem Heap. In jedem `Bruch`-Objekt befindet sich eine Referenz-Instanzvariable namens `etikett`, die auf das zugehörige **String**-Objekt zeigt.



5.2.2 Deklaration mit Modifikatoren für den Zugriffsschutz und für andere Zwecke

Während lokale Variablen in einer Methode (oder Eigenschaft) deklariert werden, erscheinen die Deklarationen der Instanzvariablen in der Klassendefinition *außerhalb* jeder Methoden- oder Eigenschaftsdefinition, z. B. in der Definition der Klasse `Bruch`:

```
using System;
public class Bruch {
   int zaehler, nenner = 1;
   string etikett = "";

   public int Zaehler {
      . . .
   }
   . . .
   public void Kuerze() {
      . . .
   }
   . . .
}
```

Man sollte die Instanzvariablen der Übersichtlichkeit halber am Anfang der Klassendefinition deklarieren, wenngleich der Compiler auch ein späteres Erscheinen akzeptiert.[1]

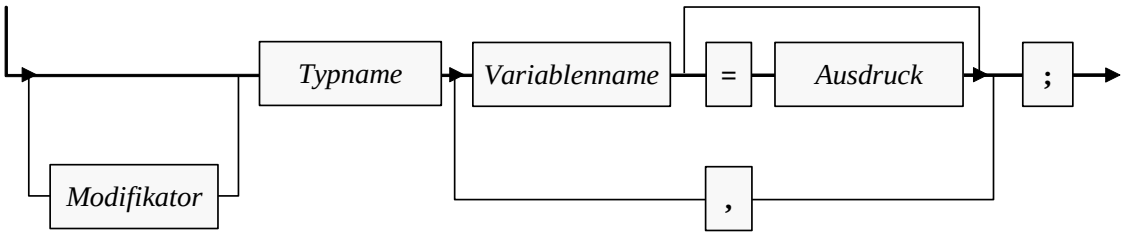
Für die Deklaration einer lokalen Variablen haben wir **const** als einzigen Modifikator kennengelernt und in einem speziellen Syntaxdiagramm beschrieben (siehe Abschnitt 4.3.9). Dieser Modifikator ist auch bei Instanzvariablen erlaubt (siehe Abschnitt 5.2.5). Außerdem kommen hier weitere Modifikatoren in Frage, die z. B. zur Spezifikation der **Sichtbarkeit** (in anderen Klassen bzw. Assemblies) dienen. Insgesamt ist es sinnvoll, in das Syntaxdiagramm zur Deklaration von Instanzvariablen den allgemeinen Begriff des Modifikators aufzunehmen:[2]

---

[1] Anders als bei lokalen Variablen von Methoden hat der Deklarationsort bei Instanzvariablen keinen Einfluss auf den Sichtbarkeitsbereich.

[2] Es ist sinnlos und verboten, einen Modifikator *mehrfach* auf eine Instanzvariable anzuwenden. Im Syntaxdiagramm zur Instanzvariablendeklaration wird der Einfachheit halber darauf verzichtet, die Mehrfachvergabe durch eine aufwändige Darstellungstechnik zu verhindern.

Deklaration von Instanzvariablen



In C# besitzen alle Instanzvariablen per Voreinstellung die Sichtbarkeit **private**, sodass sie nur in klasseneigenen Methoden bzw. Eigenschaften angesprochen werden können. Weil bei den `Bruch`-Feldern diese voreingestellte Datenkapselung erwünscht ist, kommen hier die Felddeklarationen ohne Modifikatoren aus:

```
int zaehler,
    nenner = 1;
string etikett = "";
```

Um fremden Klassen trotzdem einen (allerdings kontrollierten) Zugang zu den `Bruch`-Instanzvariablen zu ermöglichen, ist in der Klassendefinition jeweils eine zugehörige Eigenschaft vorhanden.

Auf den ersten Blick scheint die Datenkapselung nur beim Nenner eines Bruchs relevant zu sein, doch auch bei den restlichen Instanzvariablen bringt sie potentiell Vorteile:

- Zugunsten einer übersichtlichen Bildschirmausgabe soll das Etikett auf 40 Zeichen beschränkt bleiben. Mit Hilfe der Eigenschaft `Etikett()` kann dies auf einfache Weise gewährleistet werden.
- Abgeleitete (erbende) Klassen (siehe unten) können in die Eigenschaften `Zaehler` und `Nenner` neben der Null-Überwachung für den Nenner noch weitere Intelligenz einbauen und z. B. mit speziellen Aktionen reagieren, wenn der Wert auf eine Primzahl gesetzt wird. Ein zwingendes Argument für die Eigenschaft `Zaehler` würde aus der Entscheidung resultieren, beim Zähler (und beim Nenner) eines `Bruch`-Objekts negative Werte zu verbieten (vgl. Abschnitt 1.2).

Trotz ihrer überzeugenden Vorteile soll die Datenkapselung nicht zum Dogma erhoben werden. Sie verliert an Bedeutung, wenn ...

- bei einem Feld Lese- und Schreibzugriffe uneingeschränkt erlaubt sein sollen, wenn es also insbesondere nicht erforderlich ist, die möglichen Werte zu restringieren.
- es nicht von Interesse ist, auf bestimmte Wertzuweisungen zu reagieren, um z. B. bestimmte Objekteigenschaften (man sagt auch: *Invarianten*) sicherzustellen.

Um allen Klassen den Direktzugriff auf ein Feld zu erlauben, wird in seiner Deklaration der Modifikator **public** angegeben, z. B.:

```
public int Zaehler;
```

Im Abschnitt 5.11 finden Sie eine Tabelle mit allen Sichtbarkeitsstufen und zugehörigen Modifikatoren.

Bei *konstanten* Instanzvariablen (siehe Abschnitt 5.2.5), ist keine Datenkapselung als Schutz gegen irreguläre Wertzuweisungen erforderlich. Die folgenden Deklarationen stammen aus der BCL-Klasse **Math** im Namensraum **System**:

```
public const double PI = 3.14159265358979323846;
public const double E = 2.7182818284590452354;
```

Für die Benennung von Instanzvariablen werden folgende Regeln empfohlen (vgl. Mössenböck 2019, S. 14):

- Für Felder mit den Schutzstufen **private**, **protected** oder **internal** wird das *Camel Casing* verwendet (z. B. `currentSpeed`). Oft tritt eine *private* Instanzvariable (z. B. `nenner`), die einen Namen mit *kleinem* Anfangsbuchstaben besitzt, als Hintergrund zu einer *öffentlichen* Eigenschaft (z. B. `Nenner`) auf, die einen Namen mit *großem* Anfangsbuchstaben besitzt.
- Für die Felder mit der Schutzstufe **public** wird das *Pascal Casing* verwendet (z. B. `MinValue`).

Um private Instanzvariablen gut von lokalen Variablen und Formalparametern (siehe Abschnitt 5.3.1.3) unterscheiden zu können, verwenden manche Programmierer ein Präfix, z. B.:

```
int m_nenner; //das m steht für "Member"
int _nenner;
```

5.2.3 Automatische Initialisierung auf den Voreinstellungswert

Während bei lokalen Variablen der Programmierer für die Initialisierung verantwortlich ist, erhalten die Instanzvariablen eines neuen Objekts automatisch die folgenden Voreinstellungswerte, wenn der Programmierer nicht eingreift:

Datentyp	**Voreinstellungswert**
sbyte, **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**	0
float, **double**, **decimal**	0.0
char	\0 (Zeichen mit der Nummer 0 im Unicode)
bool	**false**
Referenztyp	**null**

In der Klasse `Bruch`

```
int zaehler,
    nenner = 1;
string etikett = "";
```

wird nur die automatische `zaehler`-Initialisierung unverändert übernommen, denn:

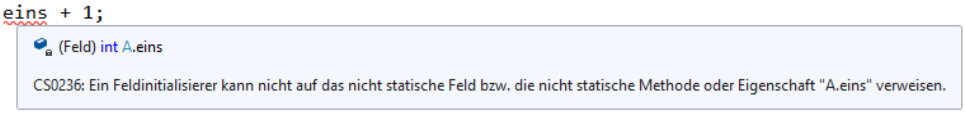
- Beim `nenner` eines Bruches wäre die Initialisierung auf 0 bedenklich, weshalb eine explizite Initialisierung auf den Wert 1 vorgenommen wird.
- Wie noch näher zu erläutern sein wird, ist **string** in C# *kein* elementarer Datentyp, sondern eine *Klasse*, wobei der Compiler als Typbezeichner neben dem Klassennamen **String** auch den Alias **string** (mit kleinem Anfangsbuchstaben) akzeptiert. Variablen von diesem Typ können einen Verweis auf ein Objekt aus dieser Klasse aufnehmen. Solange kein zugeordnetes Objekt existiert, hat eine **String**-Instanzvariable den Initialisierungswert **null**, zeigt also auf nichts. Weil der `etikett`-Wert **null** z. B. beim Aufruf der `Bruch`-Methode `Zeige()` einen Laufzeitfehler (**NullReferenceException**) zur Folge hätte, wird ein **String**-Objekt mit einer leeren Zeichenfolge erstellt und zur `etikett`-Initialisierung verwendet.[1] Das Erzeugen des **String**-Objekts erfolgt *implizit* (ohne **new**-Operator, siehe unten), indem der **String**-Variablen `etikett` ein Zeichenfolgen-Literal zugewiesen wird.

---

[1] Seit C# 8.0 bietet der C# - Compiler eine optional zuschaltbare Unterstützung zur Vermeidung der **NullReferenceException** an, die aus didaktischen Gründen vorläufig ausgespart bleibt.

Bei der optionalen Initialisierung von Instanzvariablen im Rahmen ihrer Deklaration ist es nicht erlaubt, auf andere Instanzvariablen desselben Objekts zuzugreifen, z. B.:

```
class A {
    int eins = 1;
    int zwei = eins + 1;
```



Im Rahmen eines Konstruktors (siehe Abschnitt 5.4.3) ist eine solche Initialisierung hingegen möglich.

5.2.4 Zugriff in klasseneigenen und fremden Methoden

In den Instanzmethoden einer Klasse können die Instanzvariablen des *aktuellen* (die Methode ausführenden) Objekts direkt über ihren Namen angesprochen werden, was z. B. in der `Bruch`-Methode `Zeige()` geschieht:

```
Console.WriteLine($" {luecke}{zaehler}\n {etikett}{glz}-----\n {luecke}{nenner}\n");
```

Im Beispiel zeigt sich syntaktisch kein Unterschied zwischen dem Zugriff auf die Instanzvariablen (`zaehler`, `nenner`, `etikett`) und dem Zugriff auf die lokalen Variablen (`luecke`, `glz`).

Gelegentlich kann es sinnvoll oder erforderlich sein, einem Instanzvariablennamen über das Schlüsselwort **this** (vgl. Abschnitt 5.4.5.3) eine Referenz auf das handelnde Objekt voranzustellen, wobei das Schlüsselwort und der Feldname durch den **Punktoperator** zu trennen sind:

- Das kann optional der Klarheit halber geschehen, z. B.:
  ```
  Console.WriteLine($" {luecke}{this.zaehler}\n {this.etikett}{glz}-----\n" +
                    $" {luecke}{this.nenner}\n");
  ```
- Instanzvariablen werden durch gleichnamige lokale Variablen oder Methodenparameter (siehe Abschnitt 5.3) überlagert, können jedoch in dieser (besser zu vermeidenden) Situation über das vorgeschaltete Schlüsselwort **this** weiter angesprochen werden (siehe Abschnitt 5.4.5.3).

Beim Zugriff auf eine Instanzvariable eines *anderen* Objekts derselben Klasse muss dem Variablennamen eine Referenz auf das Objekt vorangestellt werden, wobei die Bezeichner durch den Punktoperator zu trennen sind. In der folgenden Anweisung aus der `Bruch`-Methode `Addiere()` greift das handelnde Objekt lesend auf die Instanzvariablen eines anderen `Bruch`-Objekts zu, das über die Referenzvariable `b` angesprochen wird:

```
zaehler = zaehler * b.nenner + b.zaehler * nenner;
```

In einer *statischen* Methode der eigenen Klasse muss zum Zugriff auf eine Instanzvariable eines konkreten Objekts natürlich eine Referenz auf dieses Objekt vorhanden sein und dem Instanzvariablennamen vorangestellt werden (getrennt durch den Punktoperator).

Direkte Zugriffe auf die Instanzvariablen eines Objekts durch Methoden *fremder* Klassen sind zwar nicht grundsätzlich verboten, verstoßen aber gegen das Prinzip der Datenkapselung, das in der OOP von zentraler Bedeutung ist. Würden die `Bruch`-Instanzvariablen mit dem Modifikator **public**, also ohne Datenkapselung deklariert, dann könnte z. B. der Nenner eines Bruches in der **Main**() - Methode der fremden Klasse `Bruchrechnung` direkt angesprochen werden:

```
Console.WriteLine(b.nenner);
b.nenner = 0;
```

In der von uns tatsächlich realisierten `Bruch`-Definition werden solche Zu- bzw. Fehlgriffe jedoch vom Compiler verhindert, z. B.:

```
Der Zugriff auf "Bruch.nenner" ist aufgrund der Sicherheitsebene nicht
möglich
```

5.2.5 Wertfixierung zur Übersetzungszeit oder nach der Initialisierung per Konstruktor

Neben der Schutzstufenwahl gibt es weitere Anlässe für den Einsatz von Modifikatoren in Felddeklarationen. Mit dem Modifikator **const** können nicht nur lokale Variablen (siehe Abschnitt 4.3.9) sondern auch Felder einer Klasse als konstant deklariert werden. Der Wert wird zur Übersetzungszeit unveränderlich festgelegt, sodass eine neue Übersetzung erforderlich ist, wenn der Wert doch einmal geändert werden soll. Damit eignet sich der Modifikator **const** z. B. nicht für eine Variable, die zur Speicherung des Mehrwertsteuersatzes dient.

Als Datentypen sind für Felder mit **const**-Deklaration ausschließlich die elementaren Datentypen und der Typ **String** erlaubt bzw. sinnvoll.[1]

Besonderheiten bei den konstanten Feldern einer Klasse:

- Sie sind grundsätzlich *statisch*, wobei der überflüssige Modifikator **static** *nicht* angegeben werden darf. Genau genommen passt die Behandlung des Feld-Modifikators **const** also nicht in den Abschnitt 5.2 über Instanzvariablen.
- Sie werden *nicht* automatisch (mit dem typspezifischen Nullwert) initialisiert. Bei der somit erforderlichen Deklaration mit expliziter Initialisierung ist ein Ausdruck zu verwenden, der schon *zur Übersetzungszeit* berechnet werden kann.

Soll eine Instanzvariable *zur Laufzeit* in einem sogenannten Konstruktor (siehe Abschnitt 5.4.3) initialisiert und danach fixiert werden, verwendet man in der Deklaration den Modifikator **readonly**, z. B.:

Quellcode	Ausgabe

`using System;`
 	
`class ReadonlyFraction {`
`  public readonly int Num, Denom;`
 	
`  public ReadonlyFraction(int n, int d) { // Konstruktor`
`  Num = n;`
`  Denom = d != 0 ? d : 1;`
`  }`
`}`
 	
`class Prog {`
`  static void Main() {`
`  ReadonlyFraction b = new ReadonlyFraction(1, 7);`
`  Console.WriteLine(b.Denom);`
`  // b.Denom = 13; // verboten`
`  }`
`}`
	`7`

Bei den Konstruktoren handelt es sich um spezielle Methoden, die den Namen ihres Typs tragen und keinen Rückgabetyp besitzen (auch nicht **void**).

---

[1] Begründung:

- Nicht-elementare Strukturen (siehe unten) sind als Datentypen verboten, weil deren Konstruktor zur Laufzeit ausgeführt wird, was dem Prinzip der Berechenbarkeit zur Übersetzungszeit widerstrebt.
- Referenztypen außer **String** sind zwar erlaubt, aber sinnlos, weil bei der obligatorischen Initialisierung nur das Referenzliteral **null** möglich ist.

Unterschiede zwischen der **const**- und der **readonly**-Deklaration von Variablen:

	const	**readonly**
Erlaubte bzw. sinnvolle Datentypen	Elementare Datentypen und **String**	Alle Datentypen Bei einer **readonly**-deklarierten Referenzvariablen ist zu beachten, dass der Variableninhalt (die Objektadresse) nach der Initialisierung schreibgeschützt ist, während das referenzierte Objekt im Rahmen bestehender Zugriffsrechte durchaus verändert werden kann.
Möglicher Bezug	Lokale Variable oder statisches Feld	Instanzvariable oder statisches Feld

Im Zusammenhang mit dem OOP-Prinzip der Datenkapselung sind *wertfixierte und öffentliche* Felder eine akzeptable Möglichkeit, um fremden Klassen ohne Risiko einen syntaktisch einfachen *Lese*zugriff zu gestatten. In der BCL-Klasse **Math** (Namensraum **System**) ist z. B. das **double**-Feld **PI** als **public** (allgemein verfügbar) und **const** (und damit implizit auch als **static**) deklariert:

```
public const double PI = 3.14159265358979323846;
```

In vielen Büchern über die professionelle Software-Entwicklung (z. B. Bloch 2018) wird nachdrücklich empfohlen, Klassen nach Möglichkeit als unveränderlich (engl.: *immutable*) zu entwerfen, sodass ihre Objekte nach der Initialisierung nicht mehr geändert werden können. Als Programmiereinsteiger sollte bzw. muss man sich mit diesem anspruchsvollen Thema nicht unbedingt beschäftigen. Weil aber die Empfehlung zur Definition von unveränderlichen Klassen so oft anzutreffen ist, soll das Thema doch hier angesprochen werden, um möglichen Irritationen vorzubeugen.

Komplett unveränderliche Klassen spielen in der Software-Entwicklung generell eine wichtige Rolle. In C# haben sie mit der zunehmenden Unterstützung der sogenannten *funktionalen Programmierung* an Bedeutung gewonnen (z. B. Erweiterung um die Lambda-Syntax in C# 3). Code mit vielen veränderlichen Variablen ist fehleranfällig, relativ schwer zu verstehen und schlecht zu parallelisieren, d.h. auf mehrere Prozessorkerne zu verteilen. Unveränderliche Klassen in der BCL sind z. B.:

- **String**
- **DateTime**

Bloch (2018, S. 82ff) nennt folgende Vorteile unveränderlicher Klassen:

- Wird bei der Kreation eines Objekts für einen gültigen Zustand gesorgt, ist die Gültigkeit während der gesamten Lebenszeit garantiert, was die Handhabung von Objekten sicher und einfach macht.
- Ein unveränderliches Objekt kann ohne Synchronisierungsaufwand von mehreren Threads genutzt werden (siehe Kapitel 17). Es wird also auf besonders einfache Weise Thread-Sicherheit erzielt.
- Unveränderliche Objekte können an mehreren Stellen einer Anwendung wiederverwendet werden, statt jeweils ein neues Objekt zu erzeugen (z. B. ein **String**-Objekt mit dem Inhalt „`N.N.`“).

Die Klasse `Bruch` folgt dem modernen Trend hin zu unveränderlichen Objekten noch *nicht*, was vermutlich Programmiereinsteigern entgegenkommt. Sobald die sichere Verwendung der Klasse in einer Multithreading-Umgebung relevant wird, sollte eine unveränderliche Neukonzeption erwogen werden.

Dem funktionalen Programmierstil verpflichtete Methoden verändern nicht den Zustand von Objekten (z. B. die Koordinaten von Positionen), sondern produzieren nach Bedarf neue Objekte (z. B. neue Positionen). Man kann sich leicht vorstellen, dass die funktionale Programmierung nicht für

alle Aufgabenstellungen eine angemessene Modellierung erlaubt. Sehr viele Objekte zu erstellen, verursacht einen hohen Zeitaufwand und bei großen Objekten auch einen hohen Speicherbedarf.

Traditionelle, veränderliche Klassen sind für viele Aufgaben weiterhin unverzichtbar. Wir werden als Alternative zur Klasse **String**, die für unveränderliche Zeichenfolgen optimiert ist, später die Klasse **StringBuilder** kennenlernen, die für variable Zeichenfolgen konzipiert ist. In einem Testprogramm wird sich zeigen, dass die unveränderliche Klasse **String** für bestimmte Algorithmen nicht geeignet ist.

5.3 Instanzmethoden

Durch eine Bauplan-Klassendefinition werden Objekte mit einer Anzahl von Verhaltenskompetenzen entworfen, die sich über Methodenaufrufe nutzen lassen. Objekte sind also Dienstleister, die eine Reihe von Nachrichten interpretieren und mit passendem Verhalten beantworten können.

Ihre Instanzvariablen sind bei konsequenter Datenkapselung für fremde Klassen unsichtbar (*information hiding*). Um anderen Klassen trotzdem (kontrollierte) Zugriffe auf ein Feld zu ermöglichen, definiert man in C# in der Regel eine zugehörige *Eigenschaft*, die der Compiler letztlich in Zugriffsmethoden übersetzt (siehe Abschnitt 5.5).

Beim Aufruf einer Methode werden oft durch sogenannte **Parameter** erforderliche Daten und/oder Anweisungen zur Steuerung der Arbeitsweise an die Methode übergeben, und von vielen Methoden wird dem Aufrufer ein **Rückgabewert** geliefert (z. B. mit der angeforderten Information).

Ziel einer typischen Klassendefinition sind kompetente, einfach und sicher einsetzbare Objekte, die oft auch noch *reale* Objekte aus dem Aufgabenbereich der Software repräsentieren. Wenn ein anderer Programmierer z. B. ein Objekt aus unserer Klasse `Bruch` verwendet, dann kann er es mit einem Aufruf der Methode `Addiere()` veranlassen, einen per Parameter benannten zweiten Bruch zum eigenen Wert zu addieren, wobei das Ergebnis auch noch gekürzt wird:

```
public void Addiere(Bruch b) {
   zaehler = zaehler*b.nenner + b.zaehler*nenner;
   nenner = nenner*b.nenner;
   Kuerze();
}
```

Weil diese Methode auch für fremde Klassen verfügbar sein soll, wird per Modifikator die Schutzstufe **public** gewählt.

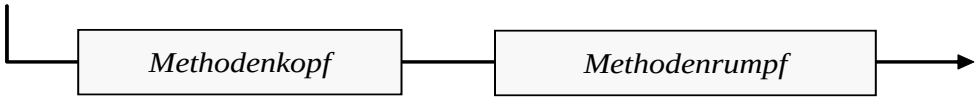
Da es vom Verlauf der Auftragserledigung nichts zu berichten gibt, liefert `Addiere()` keinen Rückgabewert. Folglich wird im Kopf der Methodendefinition der Rückgabetyp **void** angegeben.

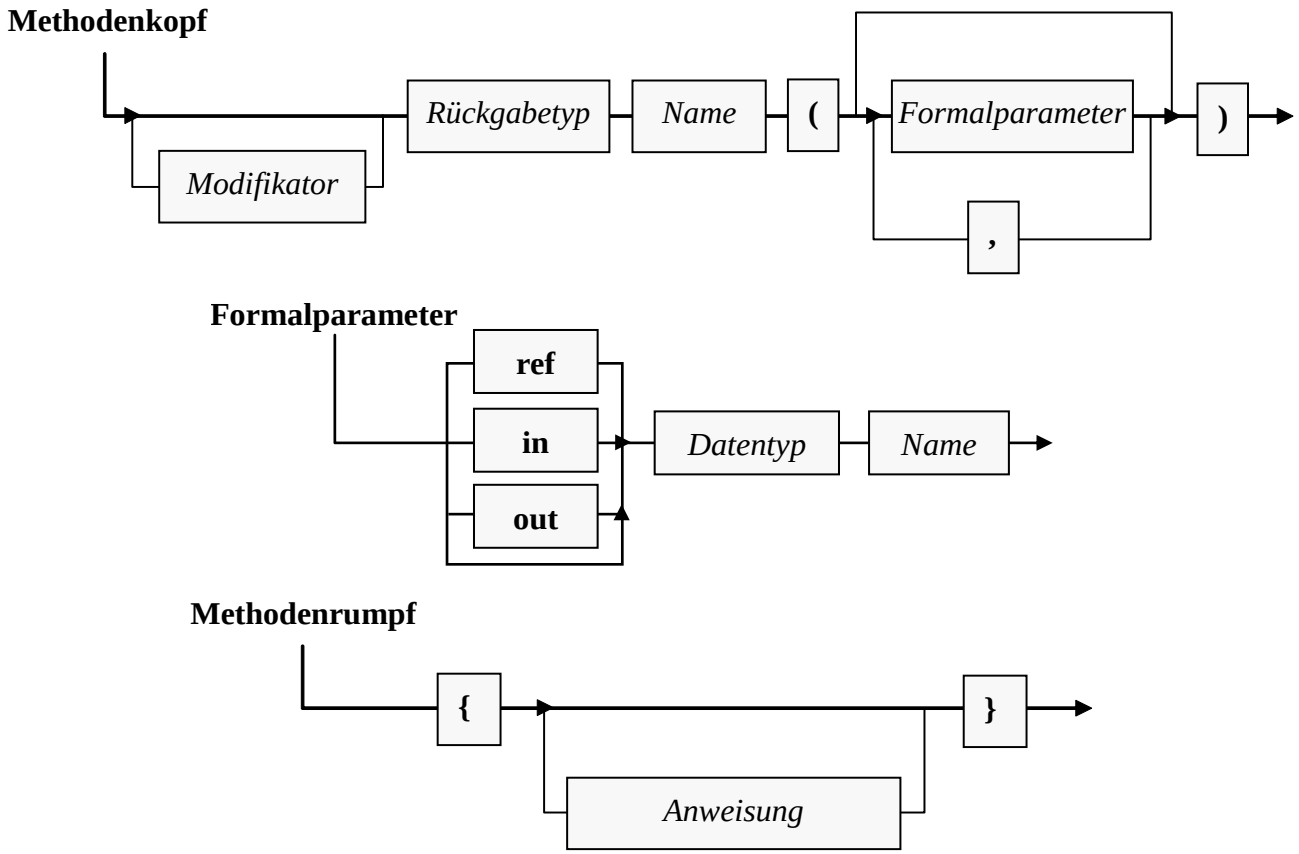
Während jedes Objekt einer Klasse seine eigenen Instanzvariablen auf dem Heap besitzt, ist der IL-Code der Instanzmethoden nur *einmal* im Speicher vorhanden und wird von allen Objekten der Klasse verwendet.

5.3.1 Methodendefinition

Die folgende Serie von Syntaxdiagrammen zur Methodendefinition unterscheidet sich von der Variante im Abschnitt 4.1.2.2 durch eine genauere Erklärung der (im Abschnitt 5.3.1.3 behandelten) Formalparameter:

Methodendefinition





Anschließend werden die (mehr oder weniger) neuen Bestandteile dieser Syntaxdiagramme erläutert. Dabei werden Methodendefinition und -aufruf keinesfalls so sequentiell und getrennt dargestellt, wie es die Abschnittsüberschriften vermuten lassen. Schließlich ist die Bedeutung mancher Details der Methodendefinition am besten am Effekt auf den Methodenaufruf zu erkennen.

Während sich in C# bei Feldnamen die Groß-/Kleinschreibung des Anfangsbuchstabens nach einer weithin akzeptierten Konvention an der Schutzstufe orientiert (Camel Casing für Felder mit den Schutzstufen **private**, **protected** oder **internal**, Pascal Casing für öffentliche Felder, vgl. Abschnitt 5.2.2), starten die Namen von Methoden in der Regel unabhängig von der Schutzstufe mit einem Großbuchstaben (Pascal Casing).

5.3.1.1 Modifikatoren

Bei einer Methodendefinition kann per Modifikator der voreingestellte **Zugriffsschutz** verändert werden. In C# gilt für Methoden wie für Instanzvariablen:

- Voreingestellt ist die Schutzstufe **private**, sodass eine Methode nur in anderen Methoden (oder Eigenschaften) derselben Klasse aufgerufen werden darf.
- Soll eine Methode *allen* Klassen zur Verfügung stehen, ist in ihrer Definition der Modifikator **public** anzugeben.

Später werden noch weitere Optionen zur Zugriffssteuerung vorgestellt.

Während man bei Instanzvariablen die Voreinstellung **private** meist belässt, ist sie bei allen Methoden zu ändern, die zur Schnittstelle einer Klasse gehören sollen. In unserer Beispielklasse `Bruch` haben *alle* Methoden den Zugriffsmodifikator **public**.

Später (z. B. im Zusammenhang mit der Vererbung) werden uns noch Methoden-Modifikatoren begegnen, die anderen Zwecken als der Zugriffsregulation dienen (z. B. **sealed**, **abstract**).

5.3.1.2 Rückgabewert und return-Anweisung

Für den Informationstransfer von einer Methode an ihren Aufrufer kann neben **ref**- und **out**-Parametern (siehe Abschnitt 5.3.1.3.2) auch ein Rückgabewert genutzt werden. Hier ist man auf einen *einzigen* Wert (von beliebigem Typ) beschränkt, doch lässt sich die Übergabe sehr elegant in den Programmablauf integrieren. Wir haben schon im Abschnitt 4.5.2 erfahren, dass ein Methodenaufruf einen Ausdruck darstellt und als Argument von komplexeren Ausdrücken oder von Methodenaufrufen verwendet werden darf, sofern die Methode einen Wert von passendem Typ liefert.

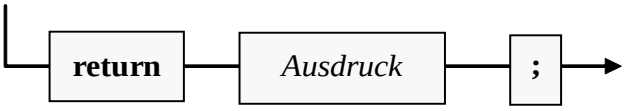
Bei der Definition einer Methode muss festgelegt werden, von welchem Datentyp ihr Rückgabewert ist. Erfolgt *keine* Rückgabe, ist der Ersatztyp **void** anzugeben.

Als Beispiel betrachten wir die aktuelle Variante der `Bruch`-Methode `Frage()` (mit einem Vorgriff auf die Ausnahmebehandlung per **try-catch** - Anweisung), die den Aufrufer durch einen Rückgabewert vom Datentyp **bool** darüber informiert, ob der Benutzer zwei ganze Zahlen im **int**-Wertebereich als Eingaben geliefert hat (**true**) oder nicht (**false**):

```
public bool Frage() {
   try {
      Console.Write("Zähler: ");
      int z  = Convert.ToInt32(Console.ReadLine());
      Console.Write("Nenner : ");
      int n = Convert.ToInt32(Console.ReadLine());
      Zaehler = z;
      Nenner = n;
      return true;
   } catch {
      return false;
   }
}
```

Ist der Rückgabetyp einer Methode von **void** verschieden, dann *muss* im Rumpf dafür gesorgt werden, dass jeder mögliche (nicht durch einen unbehandelten Ausnahmefehler abgebrochene) Ausführungspfad mit einer **return**-Anweisung endet, die einen Wert von kompatiblem Typ liefert.

return-Anweisung für Methoden *mit* Rückgabewert



In der `Bruch`-Methode `Frage()` wird am Ende eines störungsfrei durchlaufenen **try**-Blocks der boolesche Wert **true** zurückgemeldet. Tritt im **try**-Block eine Ausnahme auf (z. B. beim Versuch, eine irreguläre Benutzereingabe zu konvertieren), dann wird der **catch**-Block ausgeführt, und die dortige **return**-Anweisung sorgt für eine Terminierung mit dem Rückgabewert **false**.

Beim bedenklichen Wunsch des Anwenders, den Nenner auf 0 zu setzen, überlässt es die Methode `Frage()`der Eigenschaft `Nenner`, angemessen darauf zu reagieren.

Wenn die **Main()** - Methode eines Programms ihrem Aufrufer (also der CLR bzw. dem Betriebssystem) per **return**-Anweisung eine ganze Zahl als Information über den (Miss)erfolg ihrer Tätigkeit liefern möchte (z. B. 0: alles gut gegangen, 1: Beendigung mit Fehler), dann ist in der **Main()** - Methodendefinition der Rückgabetyp **int** anzugeben. Nach einem beendeten Programmeinsatz in einem per **cmd.exe** gestarteten Konsolenfenster befindet sich der **return**-Wert in der Pseudo-Umgebungsvariablen **errorlevel**, die mit dem **echo**-Kommando angezeigt werden kann, z. B.:

```
>echo %errorlevel%
0
```

Bei Methoden *ohne* Rückgabewert ist die **return**-Anweisung nicht unbedingt erforderlich, kann jedoch (in der Variante *ohne* Ausdruck) dazu verwendet werden, um die Methode vorzeitig zu beenden (z. B. im Rahmen einer bedingten Anweisung):

return-Anweisung für Methoden *ohne* Rückgabewert

Beispiel:

```
if (euro <= 0)
    return;
```

Die mit C# 7.0 eingeführten **ref**-Rückgabewerte sind vor allem im Zusammenhang mit Strukturen als Rückgabetypen relevant (siehe Abschnitt 15.5).

5.3.1.3 Formalparameter

Parameter wurden bisher leicht vereinfachend als Daten und/oder Informationen beschrieben, die einer Methode beim Aufruf übergeben werden. Tatsächlich kennt C# verschiedene Parameterarten, um den Informationsaustausch zwischen einer rufenden und einer aufgerufenen Methode in *beide* Richtungen zu unterstützen.

Im Kopf der **Methodendefinition** werden über sogenannte **Formalparameter** Daten von bestimmtem Typ spezifiziert, die entweder den Ablauf der Methode beeinflussen, und/oder durch die Methode verändert werden, um Informationen an den Aufrufer zu übergeben. Beim späteren **Aufruf** der Methode sind korrespondierende **Aktualparameter** zu übergeben (siehe Abschnitt 5.3.2), wobei je nach Parameterart Variablen oder Ausdrücke in Frage kommen.

In den Anweisungen des Methodenrumpfs werden die Formalparameter wie lokale Variablen verwendet, die teilweise (je nach Parameterart) mit den beim Aufruf übergebenen Aktualparameterwerten initialisiert worden sind.

Für jeden Formalparameter sind folgende Angaben zu machen:

- **Transfermodus**
  In Bezug auf die zum Datentransfer benutzte Technik sind Wert- und Verweisparameter zu unterscheiden. Bei den Verweisparametern wird durch die Schlüsselwörter **ref**, **in** und **out** festgelegt, in welche Richtung(en) Informationen übertragen werden können.
- **Datentyp**
  Es sind beliebige Typen erlaubt. Man muss den Datentyp eines Formalparameters auch dann explizit angeben, wenn er mit dem Typ des linken Nachbarn übereinstimmt.
- **Name**
  Nach den Empfehlungen aus dem Abschnitt 4.1.5 ist bei Parameternamen das Camel Casing (mit kleinem Anfangsbuchstaben) zu verwenden. Um Namenskonflikte zu vermeiden, hängen manche Programmierer an Parameternamen ein Suffix an, z. B. *par* oder einen Unterstrich. Weil Formalparameter im Methodenrumpf wie lokale Variablen funktionieren, …

  - müssen sich die Parameternamen von den Namen der (anderen) lokalen Variablen unterscheiden,
  - werden namensgleiche Instanz- bzw. Klassenvariablen überlagert.
    Diese bleiben jedoch über ein geeignetes Präfix weiter ansprechbar:
    - **this** bei Instanzvariablen
    - Klassenname bei Klassenvariablen

- **Position**
  Die Position eines Formalparameters ist natürlich nicht gesondert anzugeben, sondern liegt durch die Methodendefinition fest. Sie wird hier als relevante Eigenschaft erwähnt, weil die beim späteren Aufruf der Methode übergebenen *unbenannten* Aktualparameter gemäß ihrer Reihenfolge den Formalparametern zugeordnet werden.

5.3.1.3.1 Wertparameter

Über einen Wertparameter werden Informationen in eine Methode *kopiert*, um diese mit Daten zu versorgen oder ihre Arbeitsweise zu steuern. Als Beispiel betrachten wir die folgende Variante der `Bruch`-Methode `Addiere()`. Das beauftragte Objekt soll den über **int**-Parameter (`zpar`, `npar`) übergebenen Bruch zu seinem eigenen Wert addieren und optional (**bool**-Parameter `autokurz`) das Resultat gleich kürzen:

```
public bool Addiere(int zpar, int npar, bool autokurz) {
   if (npar != 0) {
      zaehler = zaehler * npar + zpar * nenner;
      nenner = nenner * npar;
      if (autokurz)
         Kuerze();
      return true;
   } else
      return false;
}
```

Bei der Definition eines formalen Wertparameters ist vor dem Datentyp *kein* Schlüsselwort anzugeben. Innerhalb der Methode verhält sich ein Wertparameter wie eine lokale Variable, die durch den beim Aufruf übergebenen Wert initialisiert worden ist.

Selbstverständlich sind auch methodeninterne Änderungen eines Wertparameters möglich, die *ohne* Effekt auf eine als Aktualparameter fungierende Variable der rufenden Programmeinheit bleiben. Im folgenden Beispiel übersteht die lokale Variable `i` der Methode **Main()** den Einsatz als Wertaktualparameter beim Aufruf der Methode `WertParDemo()` ohne Folgen:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  void WertParDemo(int ipar) {`
`  Console.WriteLine(++ipar);`
`  }`
 	
`  static void Main() {`
`  int i = 4711;`
`  Prog p = new Prog();`
`  p.WertParDemo(i);`
`  Console.WriteLine(i);`
`  }`
`}`
	`4712`
	`4711`

Die Demoklasse `Prog` ist startfähig, besitzt also eine Methode **Main()**. Dort wird ein Objekt der Klasse `Prog` erzeugt und beauftragt, die Instanzmethode `WertParDemo()` auszuführen. Mit dieser

auch in den folgenden Abschnitten anzutreffenden, etwas umständlich wirkenden Konstruktion wird es vermieden, im aktuellen Abschnitt 5.3.1 über Details bei der Definition von *Instanzmethoden* zur Demonstration *statische* Methoden verwenden zu müssen. Bei den Parametern und beim Rückgabetyp gibt es allerdings keine Unterschiede zwischen den Instanzmethoden und den Klassenmethoden (siehe Abschnitt 5.6.3).

Als Wertaktualparameter sind nicht nur Variablen erlaubt, sondern beliebige (z. B. auch aus einem Literal bestehende) Ausdrücke mit einem Typ, der nötigenfalls erweiternd in den Typ des zugehörigen Formalparameters gewandelt werden kann.

5.3.1.3.2 Verweisparameter

C# bietet die Möglichkeit, beim Methodenaufruf das Kopieren eines Aktualparameterwerts zu ersetzen durch die Übergabe der Speicheradresse des Originals, wobei es sich um eine Variable handeln muss. Dieses Verfahren lohnt sich vor allem bei umfangreichen Strukturen, also bei den später zu behandelnden klassenähnlichen Werttypen. Es wird übrigens auf lokale Variablen von Methoden verwiesen, d.h. die Adressen befinden sich auf dem Stack.

Nach der Richtung des möglichen Informationstransfers sind drei Arten von Verweisparametern zu unterscheiden, die jeweils durch ein Schlüsselwort zu kennzeichnen sind:

- **ref**-Parameter
  Sie ermöglichen den Informationstransfer in beide Richtungen (siehe Abschnitt 5.3.1.3.2.1).
- **out**-Parameter
  Sie ermöglichen der aufgerufenen Methode einen *Schreibzugriff* auf eine Variable der rufenden Methode (siehe Abschnitt 5.3.1.3.2.2).
- **in**-Parameter
  Sie ermöglichen der aufgerufenen Methode einen *Lesezugriff* auf eine Variable der rufenden Methode (siehe Abschnitt 5.3.1.3.2.3).

In einer Klasse dürfen keine Methoden koexistieren, die sich lediglich bei der Transferrichtung eines Verweisparameters unterscheiden. Das folgende Quellcodesegment kann also *nicht* übersetzt werden:

```
using System;
class Prog {
    public void M(ref int i) { Console.WriteLine("ref"); }
    public void M(in int i) { Console.WriteLine("in"); }

    static void Main() {
      . . .
    }
}
```

Die beiden folgenden Methoden dürfen allerdings seit C# 7.2 in einer Klasse koexistieren, weil `ipar` in der ersten Variante ein Wert- und in der zweiten Variante ein Verweisparameter ist:

```
public void M(int ipar) { Console.WriteLine("Standard"); }
public void M(in int ipar) { Console.WriteLine("in"); }
```

Im Abschnitt 5.3.5 zum Überladen von Methoden wird erläutert, wann sich die sogenannten *Signaturen* von zwei Instanzmethoden unterscheiden, sodass sie in derselben Klasse definiert werden dürfen.

5.3.1.3.2.1 ref-Parameter

Ein **ref**-Parameter ermöglicht es der aufgerufenen Methode, auf eine als Aktualparameter übergebene Variable lesend und schreibend zuzugreifen.[1] Die Methode erhält beim Aufruf keine *Kopie* des Variableninhalts, sondern die Speicheradresse des Originals. Alle methodenintern über den Formalparameternamen vorgenommenen Modifikationen wirken sich direkt auf das Original aus.

Als **ref**-Aktualparameter sind nur *Variablen* erlaubt (keine Literale), weil eine Übergabe per Verweis stattfindet. Eine als **ref**-Aktualparameter fungierende Variable muss außerdem initialisiert sein und den *exakten* Formalparametertyp besitzen. Es findet also keine implizite Typanpassung statt. Auf den garantiert definierten Wert eines **ref**-Aktualparameters kann die gerufene Methode (vor einer möglichen Veränderung) auch lesend zugreifen. Im Unterschied zu den Wertparametern und den gleich vorzustellenden **in**- bzw. **out**-Parametern ermöglichen die **ref**-Parameter einen Informationsfluss in *beide* Richtungen.

Das kennzeichnende Schlüsselwort **ref** ist in der Methodendefinition *und* beim Methodenaufruf anzugeben.

Im folgenden Programm (nach dem aus dem Abschnitt 5.3.1.3.1 bekannten Strickmuster) tauscht eine Instanzmethode die Werte zwischen den als **ref**-Aktualparameter übergebenen Variablen:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  void Tausche(ref int a, ref int b) {`
`  int temp = a;`
`  a = b;`
`  b = temp;`
`  }`
`  static void Main() {`
`  Prog p = new Prog();`
`  int x = 1, y = 2;`
`  Console.WriteLine("Vorher:  x = {0}, y = {1}", x, y);`
`  p.Tausche(ref x, ref y);`
`  Console.WriteLine("Nachher: x = {0}, y = {1}", x, y);`
`  }`
`}`
	`Vorher:  x = 1, y = 2`
	`Nachher: x = 2, y = 1`

Es folgt ein Satz für Begriffsakrobaten: Wird eine *Referenzvariable* (hat als Inhalt eine Objektadresse) als **ref**-Aktualparameter übergeben, kann man methodenintern nicht nur auf das Objekt zugreifen (z. B. auf seine Instanzvariablen bei entsprechenden Zugriffsrechten), sondern auch den Inhalt der übergebenen Referenzvariablen ändern, sodass sie z. B. anschließend auf ein anderes Objekt zeigt. Es ist nur selten sinnvoll, bei einer Methodendefinition einen **ref**-Parameter mit Referenztyp (z. B. vom Typ einer Klasse) zu verwenden. Erlaubt ist diese für Anfänger recht verwirrende Doppelreferenz jedoch. Ein möglicher Einsatzzweck ist eine methodenintern zu verändernde und zum Aufrufer zurück zu transportierende Zeichenfolge. Wie sich im Abschnitt 6.3.1.1 zeigen wird, lässt sich ein **String**-Objekt nicht ändern, sondern nur durch ein neues **String**-Objekt ersetzen. Ein **ref**-Parameter vom Typ **String** bietet die Möglichkeit, die Adresse des alten Strings in eine Methode hinein und die Adresse des neuen Strings zum Aufrufer zurückzutransportieren:

[1] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  void RefRefParDemo(ref String spar) {`
`  spar = "NEU";`
`  }`
 	
`  static void Main() {`
`  string s = "ALT";`
`  Prog p = new Prog();`
`  p.RefRefParDemo(ref s);`
`  Console.WriteLine(s);`
`  }`
`}`
	`NEU`

Ein weiteres Beispiel für die sinnvolle Verwendung des **ref**-Modifikators bei einem Parameter mit Referenztyp liefert die im Abschnitt 6.2.2 vorgestellte statische Methode **Resize()** der Klasse **Array**.

5.3.1.3.2.2 out-Parameter

Über einen **out**-Parameter kann man einer Methode die Veränderung einer als Aktualparameter übergebenen Variable ermöglichen.[1] Alle methodenintern über den Formalparameternamen vorgenommenen Modifikationen wirken sich auf das Original aus.

Als **out**-Aktualparameter sind nur *Variablen* erlaubt (keine Literale), weil eine Übergabe per Verweis stattfindet. Eine als **out**-Aktualparameter fungierende Variable muss außerdem den *exakten* Formalparametertyp haben. Es findet also keine implizite Typanpassung statt. Der Compiler interessiert sich *nicht* dafür, ob die als Aktualparameter fungierenden Variablen beim Methodenaufruf initialisiert sind. Stattdessen stellt er sicher, dass jedem **out**-Parameter vor dem Verlassen der Methode ein Wert zugewiesen wird.[2]

Das kennzeichnende Schlüsselwort **out** ist in der Methodendefinition *und* beim Methodenaufruf anzugeben, z. B.:

---

[1] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/out-parameter-modifier

[2] Außerdem ist ein Lesezugriff auf den **out**-Parameter natürlich erst nach einer Methoden-internen Wertzuweisung möglich.

Quellcode	Ausgabe (Eingaben **fett**)

`using System;`
`class Prog {`
`  void Lies(out int x, out int y) {`
`  Console.Write("x = ");`
`  x = Convert.ToInt32(Console.ReadLine());`
`  Console.Write("\ny = ");`
`  y = Convert.ToInt32(Console.ReadLine());`
`  }`
 	
`  static void Main() {`
`  Prog p = new Prog();`
`  int x, y;`
`  p.Lies(out x, out y);`
`  Console.WriteLine("\nx % y = " + (x % y));`
`  }`
`}`
	`x = 29`
	
	`y = 5`
	
	`x % y = 4`

Seit C# 7.0 kann ein **out**-Aktualparameter durch eine sogenannte **Ausschuss-Variable** (engl.: *discard variable*) ersetzt werden, wenn man am Ergebniswert des Parameters nicht interessiert ist. Statt eines Variablennamens ist der Unterstrich zu setzen, z. B. in der folgenden Variante des letzten Beispiels:

Quellcode	Ausgabe (Eingaben **fett**)

`using System;`
`class Prog {`
`  void Lies(out int x, out int y) {`
`  . . .`
`  }`
 	
`  static void Main() {`
`  Prog p = new Prog();`
`  int x;`
`  p.Lies(out x, out _);`
`  Console.WriteLine("\nx = " + x);`
`  }`
`}`
	`x = 4`
	
	`y = 7`
	
	`x = 4`

Eine ebenfalls mit C# 7.0 eingeführte Syntax-Vereinfachung ist die **Inline-Variablendeklaration** bei **out**-Aktualparametern, die in einer weiteren Variante des aktuellen Beispiels demonstriert wird:

```
static void Main() {
    Prog p = new Prog();
    p.Lies(out int x, out int y);
    Console.WriteLine("\nx % y = " + (x % y));
}
```

Wenn die als **out**-Aktualparameter fungierenden lokalen Variablen erstmals im Methodenaufruf benötigt werden, können sie in der Parameterliste deklariert werden, wobei der Typ anzugeben ist. Ihre Gültigkeit erstreckt sich wie bei anderen lokalen Variablen bis zum Ende des Blocks mit der Deklaration.

5.3.1.3.2.3 *in-Parameter*

Über einen **in**-Parameter kann man einer Methode den Lesezugriff auf eine als Aktualparameter übergebene Variable ermöglichen.[1] Die Methode erhält beim Aufruf keine *Kopie* des Variableninhalts, sondern die Speicheradresse des Originals. C# unterstützt **in**-Parameter ab Version 7.2.

Als **in**-Aktualparameter sind nur *Variablen* erlaubt (keine Literale), weil eine Übergabe per Verweis stattfindet. Eine als **in**-Aktualparameter fungierende Variable muss außerdem initialisiert sein und den *exakten* Formalparametertyp besitzen. Es findet also keine implizite Typanpassung statt. Der Compiler stellt sicher, dass die als Aktualparameter fungierenden Variablen beim Methodenaufruf initialisiert sind und verhindert einen Methoden-internen Schreibzugriff auf einen **in**-Parameter.

Das kennzeichnende Schlüsselwort **in** *muss* in der Methodendefinition und *sollte* auch beim Methodenaufruf angegeben werden, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  public void M(int i) { Console.WriteLine("Standard"); }`
`  public void M(in int i) { Console.WriteLine("in"); }`
 	
`  static void Main() {`
`  Prog p = new Prog();`
`  int i = 13;`
`  p.M(in i);`
`  p.M(i);`
`  }`
`}`
	`in`
	`Standard`

Im Fall einer Methodenüberladung gibt nämlich das Schlüsselwort **in** vor dem Aktualparameter den Ausschlag für die Wahl der auszuführenden Methode.

Während die Parameter-Schlüsselwörter **ref** und **out** eine Rolle spielen beim Informationstransfer von der gerufenen Methode an den Aufrufer, besteht der Nutzen des Schlüsselworts **in** darin, durch Adressübergabe einen Kopiervorgang einzusparen. Das lohnt sich aber nur, wenn der Aktualparameterwert größer ist als eine Speicheradresse. Die Größe einer Speicheradresse ist aus der statischen Eigenschaft **IntPtr.Size** abzulesen. Im Beispiel wird also nur die Technik der **in**-Parameter demonstriert, aber kein Nutzen erzielt:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  int i;`
`  Console.WriteLine("Größe in Bytes:\n Speicheradresse: {0} "+`
`  "\n int-Variable:  {1}", IntPtr.Size, sizeof(int));`
`  }`
`}`
	`Größe in Bytes:`
	` Speicheradresse: 8`
	` int-Variable:  4`

5.3.1.3.3 Serienparameter

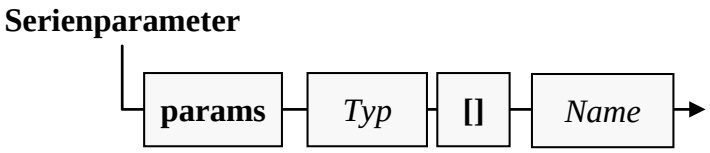
Vielleicht haben Sie sich schon darüber gewundert, dass man beim Aufruf der Methode **Console.WriteLine()** hinter einer geeigneten Formatierungszeichenfolge unterschiedlich viele Ausdrücke durch jeweils ein Komma getrennt als Aktualparameter übergeben darf, z. B.:

---

[1] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/in-parameter-modifier

```
  Console.WriteLine("x = {0} ", x);
  Console.WriteLine("x = {0}, y = {1} ", x, y);
```

Diese Variabilität wird durch einen Array-Parameter ermöglicht, der an *letzter* Stelle stehen und in der Definition durch das Schlüsselwort **params** gekennzeichnet werden muss. Im Syntaxdiagramm zum Methodenkopf wurde diese Option der Einfachheit halber weggelassen. Hier wird das Syntaxdiagramm zum Serienparameter nachgeliefert:



Als Transfermodus ist beim Serienparameter nur das Kopieren erlaubt. Das Schlüsselwort **params** ist also nicht mit den Verweisparameter-Schlüsselwörtern **ref**, **out** und **in** kombinierbar.

Zwar haben wir uns bisher kaum mit Array-Datentypen beschäftigt, doch kann das folgende Beispiel hoffentlich trotzdem die Verwendung von Serienparametern hinreichend klären:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  void PrintSum(params double[] args) {`
`  double summe = 0.0;`
`  foreach (double arg in args)`
`  summe += arg;`
`  Console.WriteLine("Die Summe ist = " + summe);`
`  }`
`  static void Main() {`
`  Prog p = new Prog();`
`  p.PrintSum(1.2, 1.0);`
`  p.PrintSum(1.2, 1.0, 3.6);`
`  p.PrintSum();`
`  double[] da = {1.1, 2.2, 3.9 };`
`  p.PrintSum(da);`
`  }`
`}`
	`Die Summe ist = 2,2`
	`Die Summe ist = 5,8`
	`Die Summe ist = 0`
	`Die Summe ist = 7,2`

Beim Methodenaufruf wird das Schlüsselwort **params** *nicht* angegeben. Beim Aufruf darf die Liste der Aktualparameter eine beliebige Länge haben, sogar die Länge null. Es ist erlaubt, aber keineswegs erforderlich, als Aktualparameter eine Variable mit Array-Datentyp (siehe unten) zu liefern.

5.3.1.4 Methodenrumpf

Über die Blockanweisung, die den Rumpf einer Methode bildet, haben Sie bereits erfahren:

- Hier werden die Formalparameter wie lokale Variablen verwendet.
- Wert-, **ref**- und **in**-Parameter werden von der aufrufenden Methode initialisiert. Damit kann die aufrufende Methode den Ablauf der gerufenen Methode beeinflussen.
- Über **ref**- und **out**-Parameter können Variablen der aufrufenden Methode (auf dem Stack) verändert werden.
- Die **return**-Anweisung dient zur Rückgabe eines Wertes an den Aufrufer und/oder zum Beenden der Methodenausführung.

Ansonsten können beliebige Anweisungen unter Verwendung von elementaren und objektorientierten Sprachelementen eingesetzt werden, um den Zweck der Methode zu realisieren.

Im letzten Satz war bewusst von *dem Zweck* einer Methode die Rede und nicht von *den Zwecken*. Durch Mehrzweckmethoden verschlechtern sich Lesbarkeit und Wartungsfreundlichkeit, während das Fehlerrisiko steigt, weil z. B. die für eine Teilaufgabe benötigten Variablen auch im Codesegment anderer Teilaufgaben gültig sind und durch Tippfehler unverhofft ins Spiel kommen oder in Mitleidenschaft gezogen werden können. Um diese Nachteile zu vermeiden, sollte für jede Aufgabe bzw. Aktivität eine eigene Methode definiert werden (Bloch 2018, S. 263).

Weil in einer Methode häufig andere Methoden aufgerufen werden, kommt es in der Regel zu mehrstufig verschachtelten Methodenaufrufen, wobei die Höhe des Stacks (Stapelspeichers) zur Verwaltung der Methodenaufrufe entsprechend wächst (siehe Abschnitt 5.3.4).

5.3.1.5 Lokale (eingeschachtelte) Methoden

Seit C# 7.0 ist es möglich, eine lokale Methode innerhalb einer ausführbaren Programmeinheit (z. B. Methode oder Eigenschaft) zu definieren. Dies ist eine sinnvolle Option, wenn es sich um eine ausschließlich lokal benötigte Hilfsmethode handelt:

- Im Quellcode kommt die eingeschränkte Verwendung der Hilfsmethode zum Ausdruck.
- Weil sich die Methode nicht auf Klassenebene befindet, ist keine Absprache mit anderen Programmierern erforderlich, die an derselben Klasse arbeiten.
- Weil die lokale Methode auf die lokalen Variablen der umgebenden Methode zugreifen kann, müssen diese Variablen weder auf Klassenebene veröffentlicht noch über Aufrufparameter transferiert werden. Seit C# 8.0 kann eine lokale Methode aber als **static** definiert werden, um den Zugriff auf die lokalen Variablen der umgebenden Methode zu verbieten (siehe unten).

Im folgenden Programm wird innerhalb der Methode `M()` die lokale Methode `LM()` definiert und verwendet:

Quellcode	Ausgabe (Eingaben **fett**)

`using System;`
`class LMDemo {`
`  static readonly int staticField = 1;`
`  readonly int field = 2;`
 	
`  int M() {`
`  int local = 3;`
`  int LM() => staticField + field + local;`
`  return LM();`
`  }`
 	
`  static void Main() {`
`  LMDemo p = new LMDemo();`
`  Console.WriteLine(p.M());`
`  }`
`}`
	`6`

Das Verschachteln von Methoden lässt sich auf mehreren Ebenen fortsetzen. Es ist also auch in einer lokalen Methode erlaubt, eingeschachtelte Methoden zu definieren.

Als Modifikatoren sind für lokale Methoden ausschließlich erlaubt:

- **static**
  Seit C# 8.0 kann eine lokale Methode als statisch definiert werden. Dann kann sie auf statische Felder zugreifen, aber nicht auf Instanzvariablen und lokale Variablen der umgebenden Methode, z. B.:

```
using System;
class LMDemo {
    static readonly int staticField = 1;
    readonly int field = 2;

    int M() {
        int local = 3;
        static int SLM() => staticField + field + local;
        return SLM();
    }

    static void Main() {
        LMDemo p = new LMDemo();
        Console.WriteLine(p.M());
    }
}
```

- **async**
  Durch den Modifikator **async** wird eine Methode als *asynchron* deklariert. Mit asynchronen Methoden werden wir uns im Zusammenhang mit Multithreading beschäftigen.
- **unsafe**
  Durch den Modifikator **unsafe** muss unsicherer Code (z. B. unter Verwendung von Zeigeroperationen oder mit eingeschränkter Portierbarkeit) markiert werden. Mit dieser selten benötigten und nach Möglichkeit zu vermeidenden Programmiertechnik werden wir uns im Kurs nicht beschäftigen.

5.3.2 Methodenaufruf und Aktualparameter

Beim Aufruf einer Instanzmethode, z. B.:

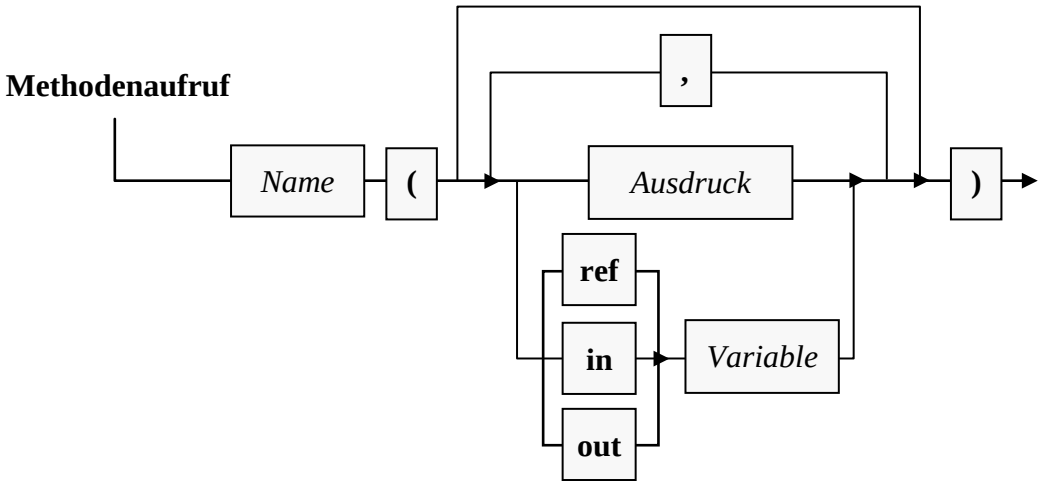
```
b1.Zeige();
```

wird nach objektorientierter Denkweise eine *Botschaft* an ein Objekt geschickt:

„b1, zeige dich!“.

Als Syntaxregel ist festzuhalten, dass zwischen dem Objektnamen (genauer: dem Namen der Referenzvariablen, die auf das Objekt zeigt) und dem Methodennamen der **Punktoperator** zu stehen hat.

Beim Aufruf einer Methode folgt ihrem Namen die in runde Klammern eingeschlossene Liste mit den **Aktualparametern**, wobei es sich im Normalfall (bei Verzicht auf benannte und optionale Parameter, siehe Abschnitt 5.3.3) um eine analog zur Formalparameterliste geordnete Serie von Ausdrücken bzw. Variablen kompatiblen Typs handeln muss.



Es ist grundsätzlich eine Parameterliste anzugeben, ggf. eine leere.

Als Beispiel betrachten wir einen Aufruf der im Abschnitt 5.3.1.3.1 vorgestellten Variante der `Bruch`-Methode `Addiere()`:

```
b1.Addiere(1, 3, true);
```

Einem **ref**- oder **out**-Parameter muss auch beim Aufruf das Schlüsselwort **ref** bzw. **out** vorangestellt werden, z. B.:

```
p.Tausche(ref x, ref y);
```

Es wird empfohlen, auch bei einem **in**-Parameter analog zu verfahren.

Liefert eine Methode einen Wert zurück, dann kann der aus ihrem Aufruf bestehende **Ausdruck** als Argument in komplexeren Ausdrücken verwenden werden, z. B.:

```
do
  Console.WriteLine("Welchen Bruch möchten Sie kürzen?");
while (!b1.Frage());
```

Durch ein angehängtes Semikolon wird jeder Methodenaufruf zur vollständigen **Anweisung**, wobei ein Rückgabewert ggf. ignoriert wird, z. B.:

```
b1.Frage();
```

Soll in einer Methodenimplementierung vom aktuell handelnden Objekt eine andere Instanzmethode ausgeführt werden, so muss beim Aufruf *keine* Objektbezeichnung angegeben werden. In beiden Varianten der `Bruch`-Methode `Addiere()` soll das beauftragte Objekt den via Parameterliste übergebenen Bruch zum eigenen Wert addieren und das Resultat (bei der Variante aus dem Abschnitt 5.3.1.3.1 parametergesteuert) gleich kürzen. Zum Kürzen kommt natürlich die entsprechende `Bruch`-Methode zum Einsatz. Weil sie vom gerade agierenden Objekt auszuführen ist, wird keine Objektbezeichnung benötigt, z. B.:

```
public void Addiere(Bruch b) {
   zaehler = zaehler*b.nenner + b.zaehler*nenner;
   nenner = nenner*b.nenner;
   Kuerze();
}
```

Wer auch solche Methodenaufrufe nach dem Schema

Empfänger.*Botschaft*

realisieren möchte, kann mit dem Schlüsselwort **this** das aktuell handelnde Objekt ansprechen, z. B.:

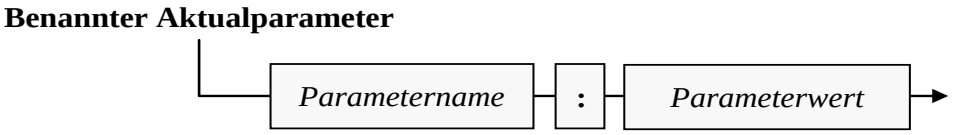
```
this.Kuerze();
```

5.3.3 Benannte und optionale Parameter

In diesem Abschnitt werden syntaktische Varianten zur Definition und Verwendung von Parametern nachgeliefert, die bisher der Übersichtlichkeit halber weggelassen wurden.

5.3.3.1 Benannte Aktualparameter

Statt beim Aufruf einer Methode mit zwei oder mehr Parametern eine analog zur Formalparameterliste geordnete Serie von Ausdrücken bzw. Variablen passenden Typs zu liefern, kann man benannte Aktualparameter verwenden (engl. Bezeichnung: *named arguments*):



Der folgende Aufruf der im Abschnitt 5.3.1.3.1 vorgestellten Variante der `Bruch`-Methode `Addiere()`

```
b1.Addiere(1, 3, true);
```

lässt sich äquivalent auch so formulieren:

```
b1.Addiere(npar: 3, zpar: 1, autokurz: true);
```

Als Vorteile der benannten Aktualparameter sind zu nennen:

- Die Lesbarkeit des Quellcodes wird verbessert.
- In Kombination mit den im Abschnitt 5.3.3.2 behandelten optionalen Parametern ist es möglich, für manche optionale Parameter einen Wert anzugeben und bei anderen die Voreinstellung beizubehalten.
- Man muss sich nicht an die Definitionsreihenfolge der Parameter halten.
- Die benannten Parameter erleichtern zusammen mit den im Abschnitt 5.3.3.2 behandelten optionalen Parametern die COM-Interoperabilität.[1]

Es ist erlaubt, auf eine Serie von Positionsparametern benannte Parameter folgen zu lassen, z. B.:

```
b1.Addiere(1, 3, autokurz: true);
```

Seit C# 7.2 dürfen Positionsparameter auf Namensparameter folgen, sofern die Namensparameter an ihrer korrekten Position stehen, z. B.:

```
b1.Addiere(zpar: 1, 2, true);
```

5.3.3.2 Optionale Parameter

In einer Methodendefinition kann seit C# 4.0 für einen Formalparameter ein Voreinstellungswert angegeben werden, sodass ein *optionaler* Parameter entsteht, der beim Aufruf im Unterschied zu den bisher behandelten *obligatorischen* Parametern weggelassen werden darf, wobei der Voreinstellungswert zum Einsatz kommt. Bei der im Abschnitt 5.3.3.1 als Beispiel betrachteten `Addiere()` -

---

[1] Gelegentlich ist es erforderlich, Komponenten mit der COM-Architektur (*Component Object Model*) in einem .NET-Programm anzusprechen. Über solche Komponenten macht z. B. Microsoft Office seine Funktionalität für andere Programme nutzbar.

Überladung aus der Klasse `Bruch` könnte z. B. der dritte Parameter einen Voreinstellungswert erhalten und somit zum optionalen Parameter werden:

```
public bool Addiere(int zpar, int npar, bool autokurz = true) {
  . . .
}
```

Ein Aufruf mit gewünschter Ergebniskürzung könnte dann wie im folgenden Beispiel formuliert werden:

```
b1.Addiere(1, 3);
```

Bei optionalen Parametern sind folgende Regeln zu beachten:

- Als Voreinstellungswert ist ein konstanter Ausdruck erlaubt, dessen Wert schon zur Übersetzungszeit feststeht, z. B.:
  ```
  void TesOpt(int i = Int32.MaxValue - 1)  {
    . . .
  }
  ```
- Auf einen optionalen Formalparameter darf kein obligatorischer mehr folgen.

Optionale Parameter sind auch bei Konstruktoren und Indexern erlaubt (siehe Abschnitt 5.8).

Beim Aufruf können optionale Parameter weggelassen werden. Um eine Teilmenge der optionalen Parameter mit Werten zu versorgen, arbeitet man mit benannten Aktualparametern (siehe Abschnitt 5.3.3.1), z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  void PrintSum(int a = 1, int b = 2, int c = 3) {`
`  Console.WriteLine("a = " + a);`
`  Console.WriteLine("b = " + b);`
`  Console.WriteLine("c = " + c + '\n');`
`  }`
 	
`  static void Main() {`
`  Prog p = new Prog();`
`  p.PrintSum(13, 4711);`
`  p.PrintSum(7);`
`  p.PrintSum();`
`  p.PrintSum(c: 999);`
`  }`
`}`
	`a = 13`
	`b = 4711`
	`c = 3`
	
	`a = 7`
	`b = 2`
	`c = 3`
	
	`a = 1`
	`b = 2`
	`c = 3`
	
	`a = 1`
	`b = 2`
	`c = 999`

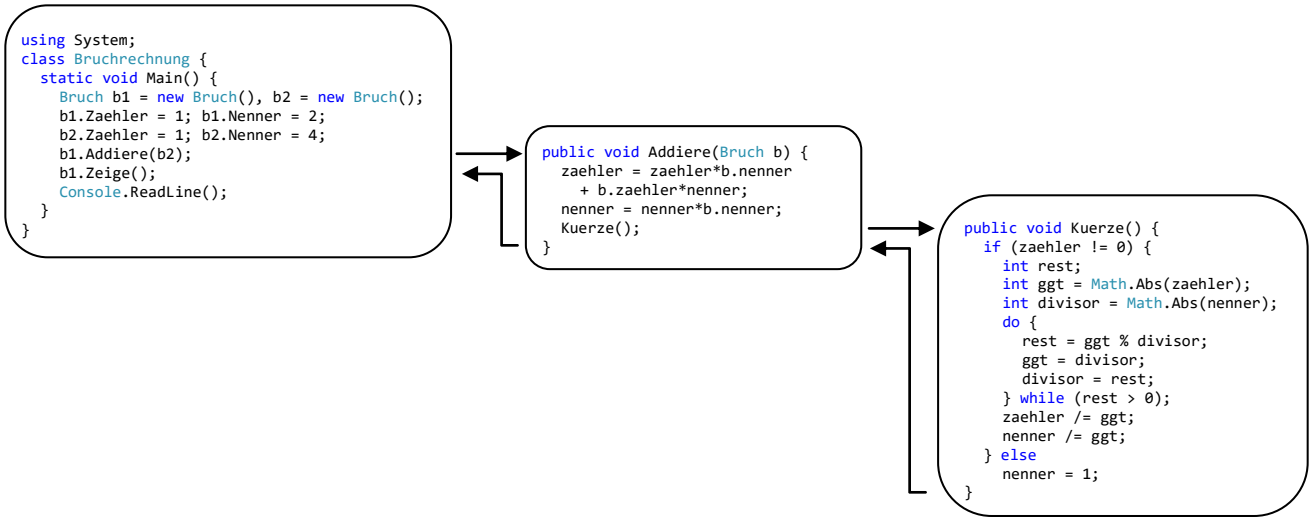
5.3.4 Debug-Einsichten zu (verschachtelten) Methodenaufrufen

Verschachtelte Methodenaufrufe stellen keine Besonderheit dar, sondern den selbstverständlichen Normalfall. Anhand der folgenden Bruchrechnungsstartklasse

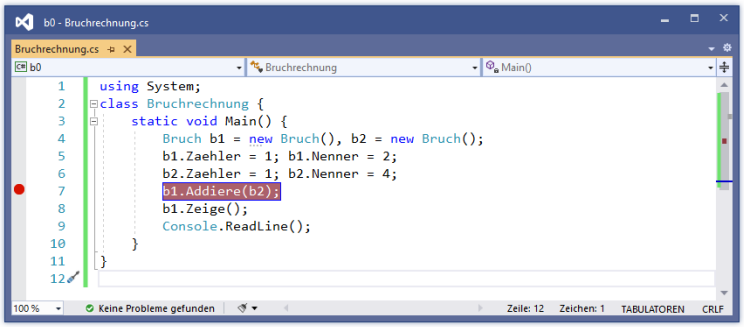
```
using System;
class Bruchrechnung {
  static void Main() {
    Bruch b1 = new Bruch(), b2 = new Bruch();
    b1.Zaehler = 1; b1.Nenner = 2;
    b2.Zaehler = 1; b2.Nenner = 4;
    b1.Addiere(b2);
    b1.Zeige();
    Console.ReadLine();
  }
}
```

soll mit Hilfe unserer Entwicklungsumgebung untersucht werden, was bei folgender Aufrufverschachtelung geschieht:

- Die statische Methode **Main()** der Klasse `Bruchrechnung` ruft die `Bruch`-Instanzmethode `Addiere()` auf.
- Die `Bruch`-Instanzmethode `Addiere()` ruft die `Bruch`-Instanzmethode `Kuerze()` auf.



Wir verwenden dabei die zur Fehlersuche konzipierte Debug-Technik unserer Entwicklungsumgebung. Das Bruchrechnungsprogramm soll an mehreren Stellen durch einen sogenannten **Halte-** bzw. **Unterbrechungspunkt** (engl. *breakpoint*) gestoppt werden, sodass wir jeweils die Lage im Hauptspeicher inspizieren können. Um einen Haltepunkt zu setzen oder wieder zu entfernen, setzt man im Quellcode-Editor einen Mausklick in die grau hinterlegte linke Randspalte neben der betroffenen Anweisung, z. B.



Befindet sich die Einfügemarke in der betroffenen Zeile, hat die Taste **F9** denselben Effekt.

Setzen Sie weitere Unterbrechungspunkte …

- in der Methode **Main()** vor den `Zeige()` - Aufruf,
- in der `Bruch`-Methode `Addiere()` vor den `Kuerze()` - Aufruf,
- in der `Bruch`-Methode `Kuerze()` vor die Anweisung

  ```
  ggt = divisor;
  ```

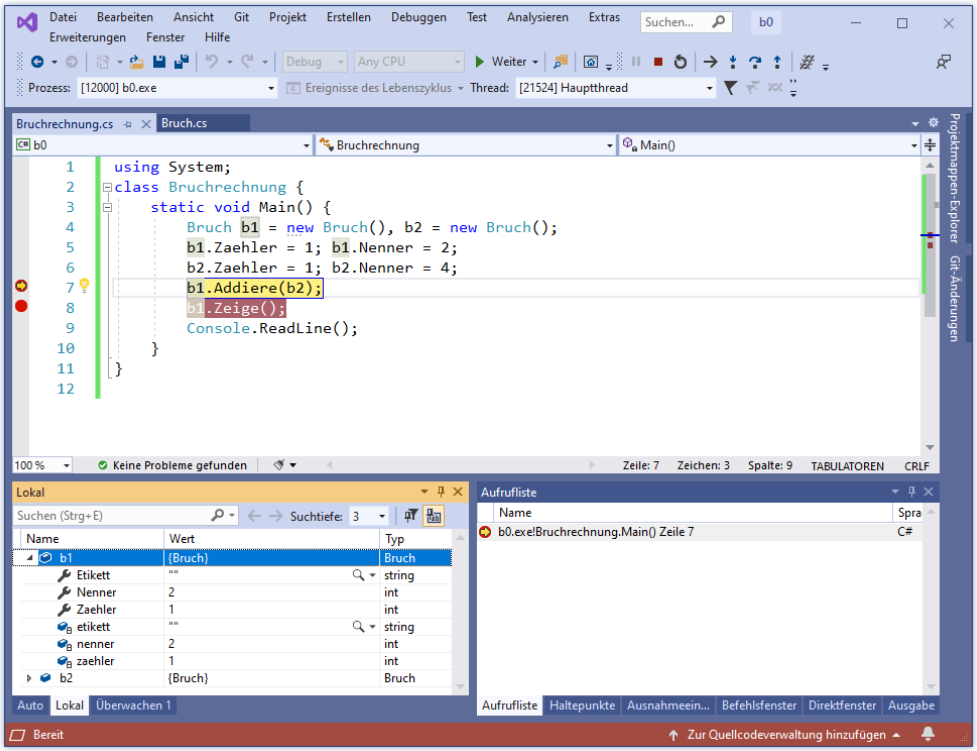
  im Block der **do-while** - Schleife.

Starten Sie das Programm über den Schalter ▶, die Funktionstaste **F5** oder den Menübefehl

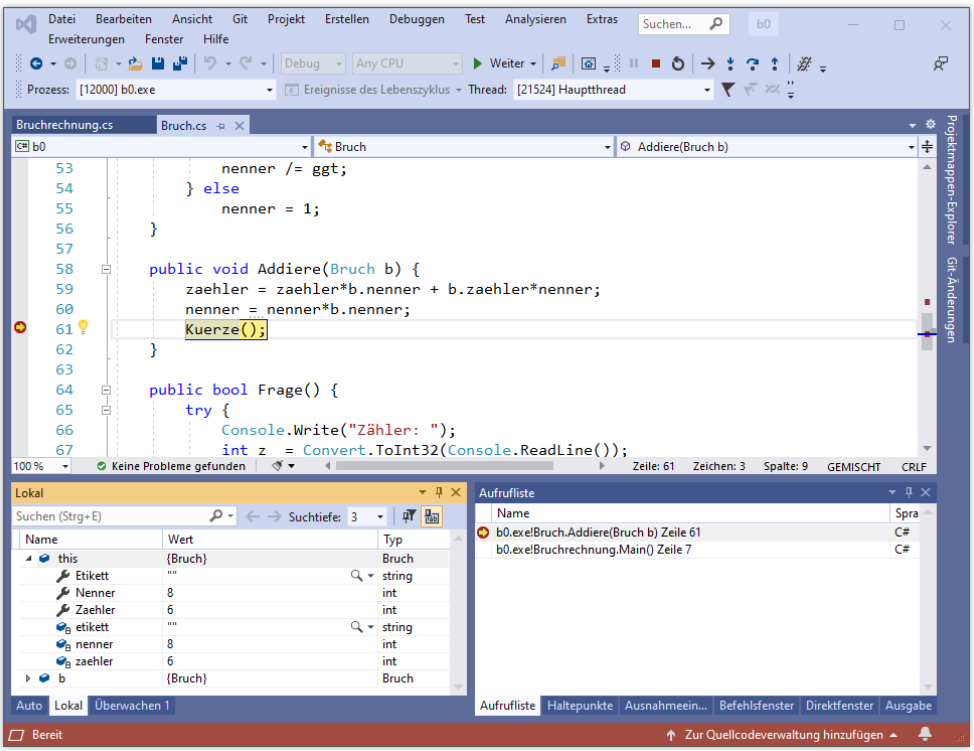
Debuggen > Debugging starten

Die Entwicklungsumgebung stoppt das Programm beim ersten Unterbrechungspunkt und zeigt im Quellcode-Editor den erreichten Programmfortschritt an. Unten links zeigt die mit **Lokal** betitelte

Registerkarte die beiden lokalen Referenzvariablen der **Main()** - Methode (`b1`, `b2`) und auf Wunsch (nach einem Mausklick auf den ▸ - Schalter neben einer Referenzvariablen) das Innenleben des referenzierten Objekts. Das mit **Aufrufliste** betitelte Fenster unten rechts zeigt, dass aktuell nur die Startmethode **Main()** aktiv ist (mit Daten im Stack-Bereich des Speichers):



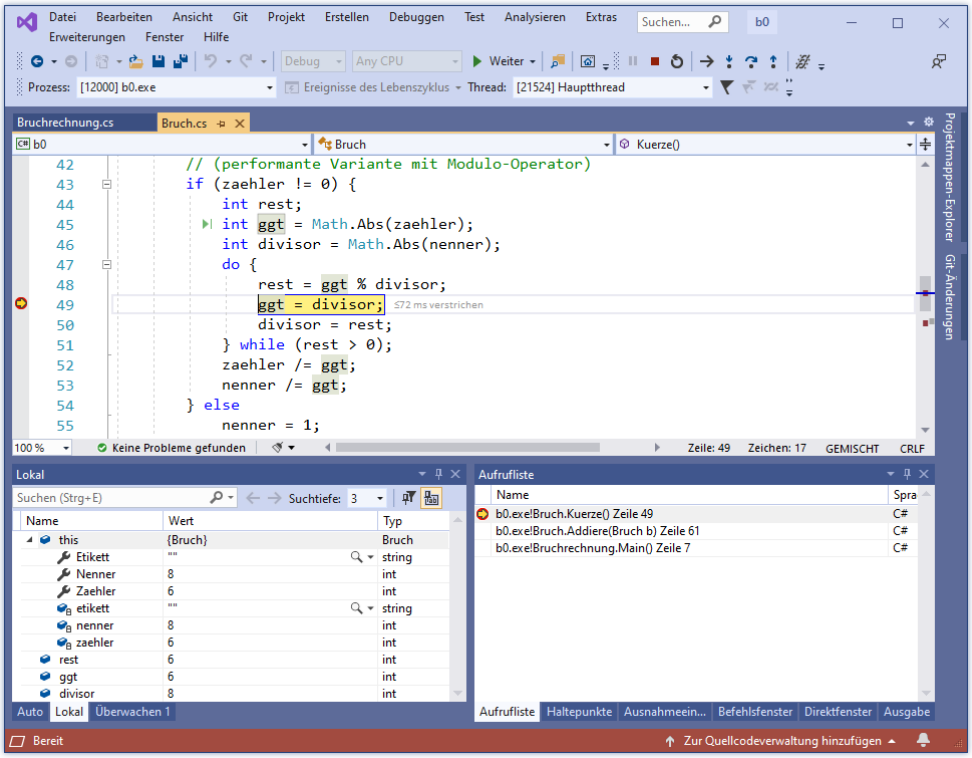
Veranlassen Sie mit dem Schalter ▶ oder mit der Funktionstaste **F5** die Fortsetzung des Programms. Beim Erreichen des zweiten Haltepunkts (Anweisung „`Kuerze();`“ in der Methode `Addiere()`) liegen auf dem Stack die Daten und Verwaltungsinformationen (die *Stack Frames*) der Methoden `Addiere()` und **Main()** übereinander:



Das Fenster **Lokal** zeigt als lokale Variablen der Methode `Addiere()`:

- **this** (Referenz auf das handelnde `Bruch`-Objekt)
  Erwartungsgemäß ist der Bruch noch nicht gekürzt.
- Parameter `b` (Referenz auf das zu addierende `Bruch`-Objekt)

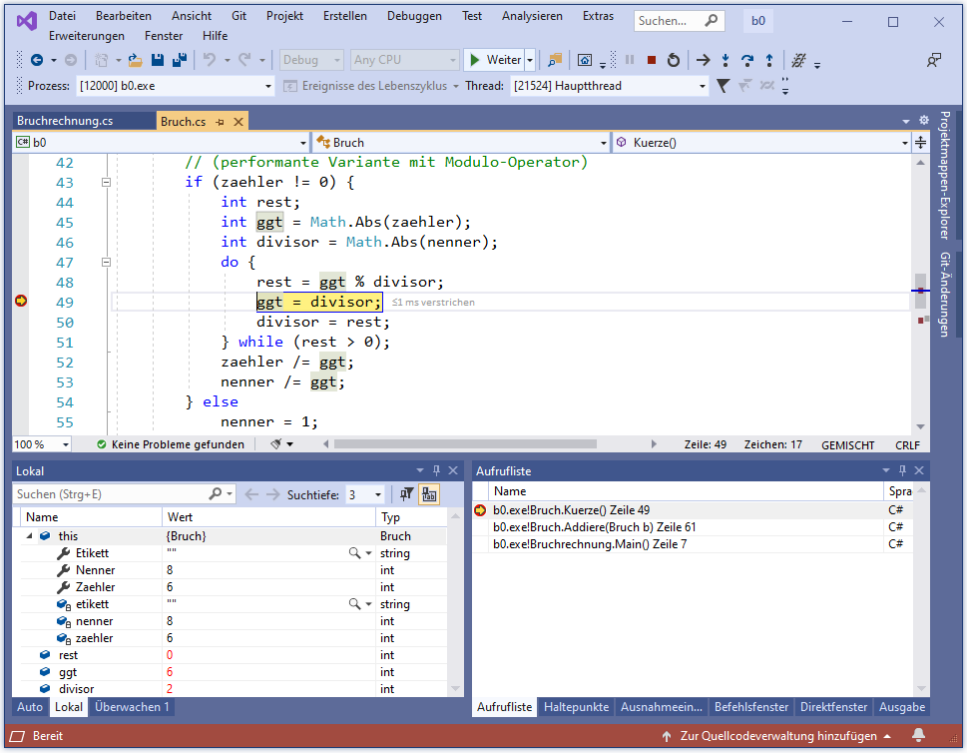
Beim Erreichen des dritten Haltepunkts (Anweisung `ggt = divisor;` in der Methode `Kuerze()`) liegen die Stack Frames der Methoden `Kuerze()`, `Addiere()` und **Main()** übereinander:



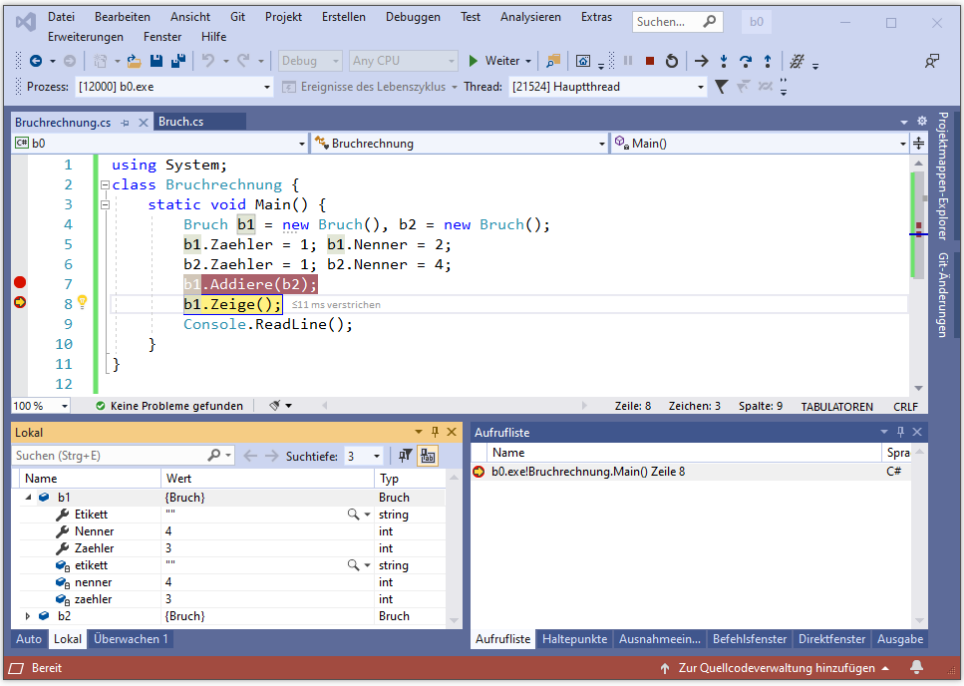
Das Fenster **Lokal** zeigt als lokale Variablen der Methode `Kuerze()`:

- **this** (Referenz auf das handelnde `Bruch`-Objekt)
- die lokalen (im Block zur **if**-Anweisung deklarierten) Variablen `rest`, `ggt` und `divisor`

Weil sich der dritte Unterbrechungspunkt in einer **do** - Schleife befindet, sind mehrere Fortsetzungsbefehle bis zum Verlassen der Methode `Kuerze()` erforderlich, wobei die Werte der lokalen Variablen den Verarbeitungsfortschritt erkennen lassen, z. B.:

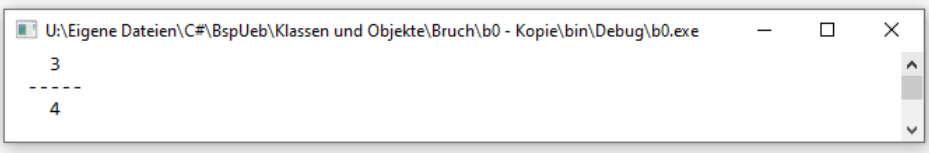


Beim Erreichen des letzten Haltepunkts (Anweisung „`b1.Zeige();`“ in **Main()**) ist nur noch der Stack Frame der Methode **Main()** vorhanden:



Die anderen Stack Frames sind verschwunden, und die dort ehemals vorhandenen lokalen Variablen existieren nicht mehr.

Nach einem weiteren Fortsetzungsklick zeigt sich das `Bruch`-Objekt `b1` im Konsolenfester:



Zum Beseitigen eines Haltepunkts klickt man ihn erneut an. Das simultane Entfernen *aller* Haltepunkte gelingt über den folgenden Menübefehl:

Debuggen > Alle Haltepunkte löschen

Ferner lassen sich per **Debuggen**-Menü alle Haltepunkte **deaktivieren** bzw. **aktivieren**.

Weil der verfügbare Speicher endlich ist, kann es bei einer Aufrufverschachtelung und der damit verbundenen Stapelung von Stack Frames zu einem Laufzeitfehler vom Typ **StackOverflow-Exception** kommen. Das wird aber nur bei einem schlecht entworfenen bzw. fehlerhaften Algorithmus passieren.

5.3.5 Methoden überladen

Die im Abschnitt 5.3.1.3.1 vorgestellte `Addiere()` - Methode kann problemlos in der `Bruch`-Klassendefinition mit der dort bereits vorhandenen `Addiere()` - Variante koexistieren, weil beide Methoden unterschiedliche Parameterlisten besitzen. Besitzt eine Klasse mehrere Methoden mit demselben Namen, liegt eine sogenannte *Überladung* von Methoden vor.

Eine Überladung ist erlaubt, wenn sich die **Signaturen** der beteiligten Methoden unterscheiden. Zwei Methoden besitzen genau dann *dieselbe* Signatur, was *innerhalb einer Klasse* verboten ist, wenn die folgenden Bedingungen erfüllt sind:[1]

- Die Namen der Methoden sind identisch.
- Die Formalparameterlisten sind gleich lang.
- Positionsgleiche Parameter haben denselben Datentyp.
- Positionsgleiche Parameter haben denselben Transfermodus, d.h.:
  - Beide sind Wertparameter,
  - oder beide sind Verweisparameter.

  Bei zwei Verweisparametern sorgen unterschiedliche Richtungsangaben (**ref**, **in**, **out**) *nicht* für abweichende Signaturen.

Für die Signatur einer Methode sind irrelevant:

- Der Rückgabetyp
  Die fehlende Signaturrelevanz des Rückgabetyps resultiert daraus, dass der Rückgabewert einer Methode in Anweisungen oft keine Rolle spielt (ignoriert wird). Folglich muss generell unabhängig vom Rückgabetyp für den Compiler entscheidbar sein, welche Methode aus einer Überladungsfamilie zu verwenden ist.
- Modifikatoren
- Die *Namen* der Formalparameter
- Das beim letzten Formalparameter erlaubte **params**-Schlüsselwort (vgl. ECMA 2017, S. 46ff).

Ist bei einem Methodenaufruf die angeforderte Überladung nicht eindeutig zu bestimmen, meldet der Compiler einen Fehler, was aber bei einem sinnvollen Entwurf von überladenen Methoden nur sehr selten passiert.

Von einer Methode unterschiedlich parametrisierte Varianten in eine Klassendefinition aufzunehmen, lohnt sich z. B. in den folgenden Situationen:

---

[1] Bei den später zu behandelnden *generischen* Methoden (siehe Abschnitt 8.5) muss die Liste mit den Kriterien für die Identität von Signaturen erweitert werden.

- Für verschiedene Datentypen (z. B. **double** und **int**) werden analog arbeitende Methoden benötigt. So besitzt z. B. die Klasse **Math** im Namensraum **System** u. a. folgende Methoden, um den Betrag einer Zahl zu berechnen:

  public static decimal Abs(decimal *value*)
  public static double Abs(double *value*)
  public static float Abs(float *value*)
  public static int Abs(int *value*)
  public static long Abs(long *value*)

  Seit der .NET - Version 2.0 bieten allerdings *generische Methoden* (siehe unten) eine elegantere Lösung für die Unterstützung verschiedener Datentypen.[1]
- Für eine Methode sollen unterschiedlich umfangreiche Parameterlisten angeboten werden, sodass zwischen einer bequem aufrufbaren Standardausführung (z. B. mit leerer Parameterliste) und einer individuell gestalteten Ausführungsvariante gewählt werden kann. Über die seit C# 4.0 verfügbaren optionalen Parameter lässt sich die beschriebene Aufgabenstellung allerdings oft auch mit einer einzigen Methodendefinition lösen (siehe Abschnitt 5.3.3).

Zum Schluss noch ein Beispiel aus dem Kuriositäten- bzw. Gruselkabinett. Wenn zu einer Methode mit optionalen Parametern (siehe Abschnitt 5.3.3.2) eine Überladung mit einer kürzeren Parameterliste existiert, wobei mindestens ein optionaler Parameter fehlt, dann wird bei einem Aufruf mit der kürzeren Aktualparameterliste die Methode *ohne* optionale Parameter aufgerufen. In dieser Konstellation ist die Definition von Voreinstellungswerten wirkungslos, z. B.:[2]

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  void Test(int i = 13)  {`
`  Console.WriteLine("Opt. Par. = " + i);`
`  }`
 	
`  void Test() {`
`  Console.WriteLine("Ohne Parameter");`
`  }`
 	
`  static void Main() {`
`  Prog p = new Prog();`
`  p.Test();`
`  }`
`}`
	`Ohne Parameter`

Hat ein Anwender der Klasse die kurze Parameterliste im Vertrauen auf den Voreinstellungswert des weggelassenen Parameters gewählt, dann können durchaus ernste Konsequenzen resultieren (z. B. ein Raketenabsturz). Zu der unglücklichen Konstellation kann es z. B. kommen, wenn mehrere Programmierer an einer Klassendefinition arbeiten, oder wenn ein Programmierer mit größeren zeitlichen Unterbrechungen an einer Klassendefinition arbeitet.

---

1 So tauscht z. B. die folgende statische Methode die Inhalte von zwei **ref**-Parametern mit beliebigem (natürlich identischem) Datentyp:

```
static void Tausche<T>(ref T a, ref T b) {
  T temp = a;
  a = b;
  b = temp;
}
```

2 Diese Konstellation wurde aufgeklärt von Jens Weber (Universität Trier).

5.4 Objekte

Im Abschnitt 5.4 geht es darum, wie Objekte erzeugt und im obsoleten Zustand wieder aus dem Speicher entfernt werden.

5.4.1 Referenzvariablen deklarieren

Um irgendein Objekt aus der Klasse `Bruch` ansprechen zu können, benötigen wir eine **Referenzvariable** mit dem Datentyp `Bruch`. In der folgenden Anweisung wird eine solche Referenzvariable definiert und auch gleich initialisiert:

```
Bruch b = new Bruch();
```

Um die Wirkungsweise dieser Anweisung Schritt für Schritt zu untersuchen, beginnen wir mit einer einfacheren Variante *ohne* Initialisierung:

```
Bruch b;
```

Hier wird die **Referenzvariable** `b` mit dem Datentyp `Bruch` deklariert, der man folgende Werte zuweisen kann:

- die Adresse eines `Bruch`-Objekts
  In der Variablen wird kein komplettes `Bruch`-Objekt mit sämtlichen Instanzvariablen abgelegt, sondern ein **Verweis** (eine **Referenz**) auf einen Ort im Heap-Bereich des programmeigenen Speichers, wo sich ein `Bruch`-Objekt befindet.
  Sollte einmal eine Ableitung der Klasse `Bruch` definiert werden, können deren Objekte ebenfalls über `Bruch`-Referenzvariablen verwaltet werden. Von der Vererbungstechnik der objektorientierten Programmierung haben Sie schon einiges gehört, doch steht die gründliche Behandlung noch aus.
- **null**
  Wird einer Variablen dieses Referenzliteral zugewiesen, dann ist die Variable nicht undefiniert, sondern zeigt explizit auf nichts.[1]

Wir nehmen nunmehr offiziell und endgültig zur Kenntnis, dass *Klassen als Datentypen* verwendet werden können und haben damit bislang folgende Datentypen zur Verfügung (vgl. Abschnitt 4.3.2):

- Elementare Typen (**bool**, **char**, **byte**, **double**, ...)
  Hier handelt es sich um Werttypen.
- Klassen (Referenztypen)
  Ist eine Variable vom Typ einer Klasse, dann kann sie (neben **null**) die Adresse eines Objekts aus dieser Klasse oder aus einer daraus abgeleiteten Klasse aufnehmen.

Später kommen mit den *Strukturen* noch Werttypen hinzu, deren Instanzen (wie die Objekte von Klassen) problemadäquat mit beliebig vielen Feldern ausgestattet werden können.

5.4.2 Objekte erzeugen

Damit z. B. der folgendermaßen deklarierten Referenzvariablen `b` vom Datentyp `Bruch`

```
Bruch b;
```

ein Verweis auf ein `Bruch`-Objekt als Wert zugewiesen werden kann, muss ein solches Objekt erst erzeugt werden, was per **new**-Operator geschieht, z. B. im folgenden Ausdruck:

```
new Bruch()
```

---

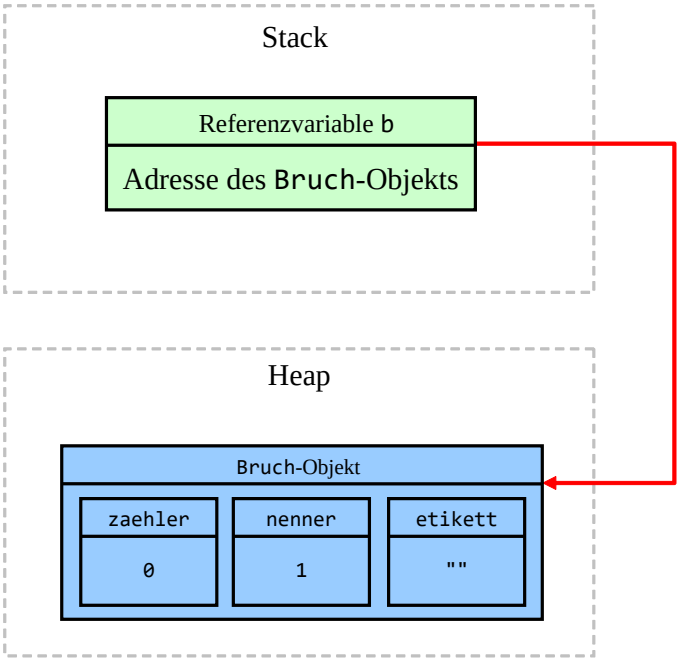
[1] Seit C# 8.0 bietet der C# - Compiler eine optional zuschaltbare Unterstützung zur Vermeidung der **NullReferenceException** an, die aus didaktischen Gründen vorläufig ausgespart bleibt. Man kann den Compiler so konfigurieren, dass einer **Bruch**-Referenzvariablen der Wert **null** *nicht* mehr zugewiesen werden darf.

Als Operanden erwartet der **new**-Operator einen Klassennamen, dem eine Parameterliste zu folgen hat, weil der **new**-Operand als Name eines *Konstruktors* fungiert (siehe Abschnitt 5.4.3). Die involvierte Klasse legt den Typ des Ausdrucks fest. Als Wert resultiert eine Referenz, die (im Rahmen bestehender Rechte) einen Zugriff auf das neue Objekt (seine Methoden, Eigenschaften, etc.) ermöglicht. Auch der Konstruktor ist ein Klassenmitglied, dessen Verfügbarkeit für andere Klassen per Zugriffsmodifikator geregelt wird.

In der **Main()** - Methode der folgenden Startklasse

```
class Bruchrechnung {
  static void Main() {
    Bruch b = new Bruch();
     . . .
  }
}
```

wird die vom **new**-Operator gelieferte Adresse mit dem Zuweisungsoperator in die lokale Referenzvariable `b` geschrieben. Es resultiert die folgende Situation im programmeigenen Arbeitsspeicher:[1]



Während lokale Variablen im **Stack**-Bereich des Arbeitsspeichers abgelegt werden, entstehen Objekte mit ihren Instanzvariablen auf dem **Heap**.

In einem Programm können durchaus *mehrere* Referenzvariablen auf *dasselbe* Objekt zeigen, z. B.:

[1] Hier wird aus didaktischen Gründen ein wenig gemogelt. Die Instanzvariable `etikett` ist vom Typ der Klasse **String**, zeigt also auf ein **String**-Objekt, das „neben" dem `Bruch`-Objekt auf dem Heap liegt. In der `Bruch`-Referenzinstanzvariablen `etikett` befindet sich die Adresse des **String**-Objekts.

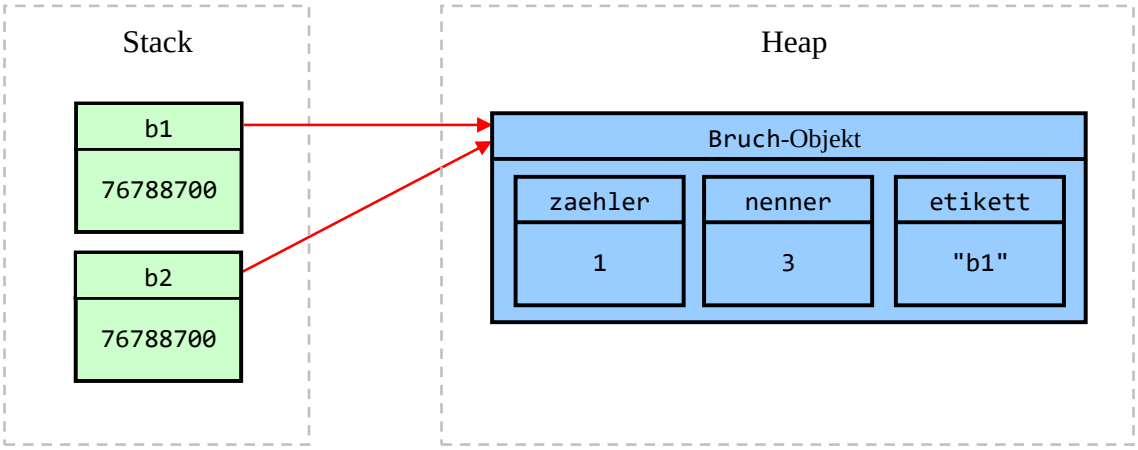
Quellcode	Ausgabe

`using System;`
`class Bruchrechnung {`
`  static void Main() {`
`  Bruch b1 = new Bruch();`
`  b1.Zaehler = 1;`
`  b1.Nenner = 3;`
`  b1.Etikett = "b1";`
`  Bruch b2 = b1;`
`  b2.Etikett = "b2";`
`  b1.Zeige();`
`  }`
`}`
	`  1`
	`b2 = -----`
	`  3`

In der Anweisung

```
Bruch b2 = b1;
```

wird die neue Referenzvariable `b2` vom Typ `Bruch` angelegt und mit dem Inhalt von `b1` (also mit der Adresse des bereits vorhandenen `Bruch`-Objekts) initialisiert. Es resultiert die folgende Situation im Speicher des Programms:



Hier sollte nur die Möglichkeit der Mehrfachreferenzierung demonstriert werden. Bei einer ernsthaften Anwendung des Prinzips befinden sich die alternativen Referenzen an verschiedenen Stellen des Programms, z. B. in Instanzvariablen verschiedener Objekte. In einem Speditionsverwaltungsprogramm kennen z. B. alle Objekte zu einzelnen Fahrzeugen die Adresse des Planerobjekts, dem sie besondere Ereignisse wie Pannen melden.

5.4.3 Objekte initialisieren

5.4.3.1 Konstruktoren

In diesem Abschnitt werden mit den sogenannten *Konstruktoren* spezielle Methoden behandelt, die beim Erzeugen von neuen Objekten zum Einsatz kommen, um deren Instanzvariablen zu initialisieren und/oder andere Arbeiten zu verrichten (z. B. Öffnen einer Datei). Ziel der Konstruktor-Tätigkeit ist es, ein neues Objekt in einen validen Zustand zu bringen und für seinen Einsatz vorzubereiten. Wie Sie bereits wissen, wird zum Erzeugen von Objekten der **new**-Operator verwendet. Als Operand ist ein Konstruktor der gewünschten Klasse anzugeben.

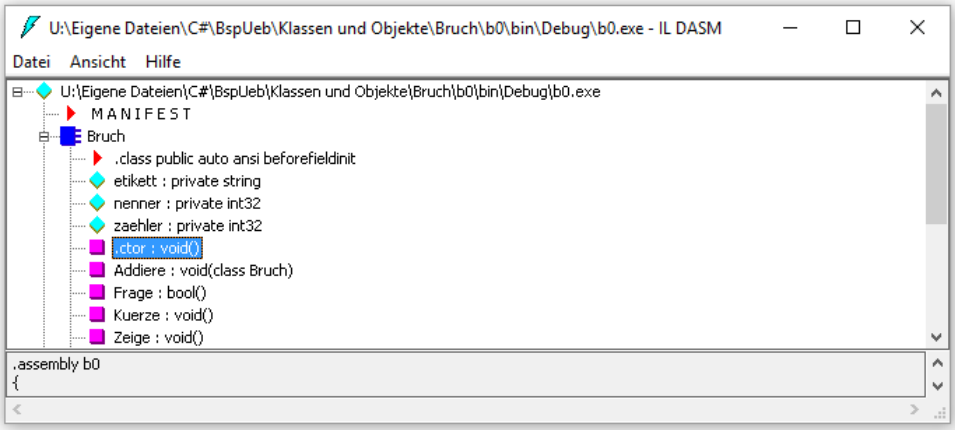
Hat der Programmierer zu einer Klasse *keinen* Konstruktor definiert, dann erhält sie automatisch einen **Standardkonstruktor** (engl.: *default constructor*). Weil dieser keine Parameter besitzt, ergibt sich sein Aufruf aus dem Klassennamen durch Anhängen einer leeren Parameterliste, z. B.:

```
Bruch b = new Bruch();
```

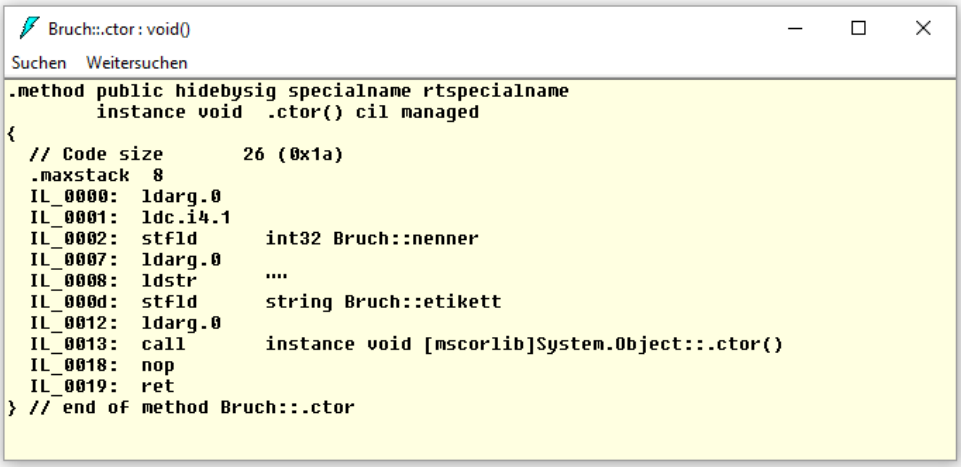
Bei der Deklaration einer *lokalen* Variablen mit Initialisierung kann über das Schlüsselwort **var** und die implizite Typisierung etwas Schreibaufwand gespart und die doppelte Nennung des Klassennamens vermieden werden (vgl. Abschnitt 4.3.6):

```
var b = new Bruch();
```

Inspiziert man die Klasse `Bruch` mit dem Hilfsprogramms **ILDasm**, dann entdeckt man den Standardkonstruktor unter dem Namen **.ctor** (Abkürzung für *constructor*):[1]



In diese Methode fügt der Compiler automatisch IL-Code für die im Rahmen ihrer Deklaration initialisierten Instanzvariablen ein (betroffen: `nenner` und `etikett`):[2]



Für eine automatische Null-Initialisierung (vgl. Abschnitt 5.2.3) ist hingegen kein IL-Code erforderlich.

Am Ende (!) des IL-Codes zum Standardkonstruktor wird der parameterlose Konstruktor der Basisklasse aufgerufen, wobei unsere Klasse `Bruch` direkt von der Urahnklasse **Object** im Namensraum **System** abstammt.

---

[1] Mit dem Visual Studio 2019 wird das Programm **ildasm.exe** unter Windows 10 (64 Bit) im folgenden Ordner installiert:

C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.8 Tools

Wie der Speicherort vermuten lässt, kann das Programm nur Assemblies für das .NET Framework verarbeiten.

[2] Das Programm **ildasm.exe** zeigt den CIL-Code einer Methode nach einem Doppelklick auf ihren Namen an.

Der Standardkonstruktor hat die Schutzstufe **public**, ist also allgemein verfügbar (ECMA 2017, S, 333).[1]

Beim Klassendesign ist es oft erforderlich, Konstruktoren *explizit* zu definieren, um das individuelle Initialisieren der Instanzvariablen von neuen Objekten unter Beachtung von Konsistenzbedingungen zu ermöglichen. Dabei sind folgende Regeln zu beachten:

- Ein Konstruktor trägt denselben Namen wie die Klasse.
- In der Definition wird *kein* Rückgabetyp angegeben.
- Als Modifikatoren sind nur solche erlaubt, welche die Sichtbarkeit des Konstruktors (den Zugriffsschutz) regeln (z. B. **public**, **private**), sodass pro Konstruktor maximal *ein* Modifikator verwendet werden kann.
- Während der Standardkonstruktor die Schutzstufe **public** besitzt, haben explizit definierte Konstruktoren wie gewöhnliche Methoden die voreingestellte Schutzstufe **private**. Wenn sie für beliebige fremde Klassen (in beliebigen Assemblies) zur Objektkreation verfügbar sein sollen, ist also in der Definition der Modifikator **public** anzugeben.
- Wie bei einer gewöhnlichen Methodendefinition ist eine Parameterliste anzugeben, ggf. eine leere. Parameter erlauben das individuelle Initialisieren der Instanzvariablen von neuen Objekten.
- Sobald man einen expliziten Konstruktor definiert hat, steht der Standardkonstruktor *nicht* mehr zur Verfügung. Ist weiterhin ein parameterfreier Konstruktor erwünscht, so muss dieser *zusätzlich* explizit definiert werden.
- Der Compiler fügt bei *jedem* Konstruktor automatisch IL-Code für die im Rahmen der Deklaration initialisierten Instanzvariablen ein (siehe oben), z. B. auch bei einem Konstruktor mit leerem Anweisungsteil.
- Es sind beliebig viele Konstruktoren möglich, die alle denselben Namen und jeweils eine individuelle Parameterliste haben müssen. Das Überladen von Methoden (vgl. Abschnitt 5.3.5) ist also auch bei Konstruktoren erlaubt.
- Konstruktoren können (bis auf zwei Ausnahmen, siehe unten) nicht direkt aufgerufen, sondern nur als Argument des **new**-Operators verwendet werden.

Für die Klasse `Bruch` eignet sich z. B. der folgende Konstruktor mit Parametern zur Initialisierung aller Instanzvariablen:

```
public Bruch(int zpar, int npar, string epar) {
  Zaehler = zpar;
  Nenner  = npar;
  Etikett = epar;
}
```

Weil die „beantragten“ Initialisierungswerte nicht direkt den Feldern zugewiesen, sondern durch die Eigenschaften `Zaehler`, `Nenner` und `Etikett` geschleust werden, bleibt die Datenkapselung erhalten. Wie jede andere Methode einer Klasse muss ein Konstruktor so entworfen sein, dass die Objekte der Klasse unter allen Umständen konsistent und funktionstüchtig sind. In der Klassendokumentation sollte darauf hingewiesen werden, dass dem Wunsch, den Nenner eines neuen `Bruch`-

[1] Abstrakte Klassen, mit denen wir uns später im Zusammenhang mit der Vererbung beschäftigen werden, haben allerdings einen Standardkonstruktor mit der Zugriffsebene **protected**.

Objekts per Konstruktor auf den Wert 0 zu setzen, *nicht* entsprochen wird, und dass stattdessen der Wert 1 resultiert.[1]

Wenn weiterhin auch ein parameterfreier Konstruktor verfügbar sein soll, dann muss dieser explizit definiert werden, z. B. mit leerem Anweisungsteil:

```
public Bruch() {}
```

Im folgenden Programm werden beide Konstruktoren eingesetzt:

Quellcode	Ausgabe

`using System;`
`class Bruchrechnung {`
`  static void Main() {`
`  var b1 = new Bruch(1, 2, "b1");`
`  var b2 = new Bruch();`
`  b1.Zeige();`
`  b2.Zeige();`
`  }`
`}`
	`  1`
	`b1 =  -----`
	`  2`
	
	`  0`
	`-----`
	`  1`

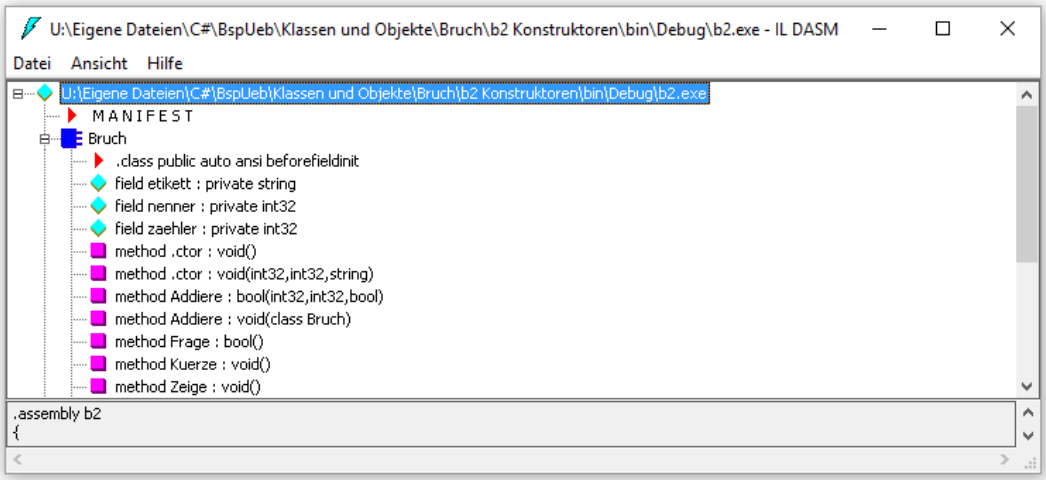
Als Ausnahme von der Regel, dass Konstruktoren nur über den **new**-Operator genutzt werden können, darf man zwischen der Parameterliste und dem Anweisungsblock eines Konstruktors einen anderen Konstruktor derselben Klasse über das Schlüsselwort **this** aufrufen, z. B.:

```
public Bruch() : this(0, 1, "unbekannt") {}
```

Die Anweisungen des gerufenen Konstruktors werden zuerst ausgeführt. Im Beispiel beschränkt sich der rufende Konstruktor darauf, dem (aufwändiger parametrisierten) Kollegen die Arbeit zu überlassen.

Im Zusammenhang mit der Vererbung werden wir noch die Möglichkeit kennenlernen, in einem Konstruktor über das Schlüsselwort **base** einen Konstruktor der Basisklasse aufzurufen.

Wie das Hilfsprogramm **ILDasm** für die aktuelle Ausbaustufe des Bruchrechnungs-Assemblies zeigt, führen die Konstruktoren einer Klasse unter dem Namen **.ctor** die Liste der Instanzmethoden an:



[1] Eine sinnvolle Reaktion auf den Versuch, ein defektes Objekt zu erstellen, kann darin bestehen, im Konstruktor eine sogenannte *Ausnahme* zu werfen und dadurch den Aufrufer über das Scheitern seiner Absicht zu informieren. Mit der Kommunikation über Ausnahmeobjekte werden wir uns später beschäftigen.

5.4.3.2 Zieltypisierte new-Ausdrücke

Wie bereits im Abschnitt 4.3.7.2 im Zusammenhang mit lokalen Variablen erwähnt, kann seit C# 9.0 bei der initialisierenden Deklaration einer Referenzvariablen zur Aufnahme einer Objektadresse die doppelte Nennung der Klassennamens vermieden werden. Die folgende Anweisung

```
Bruch b = new Bruch();
```

lässt sich also vereinfachen zu:

```
Bruch b = new();
```

Während die im Abschnitt 4.3.7.1 beschriebene Schreibvereinfachung durch Verwendung des Schlüsselworts **var** an Stelle des Datentyps in einer Felddeklaration verboten ist, ist ein zieltypisierter **new**-Ausdruck in dieser Situation erlaubt.

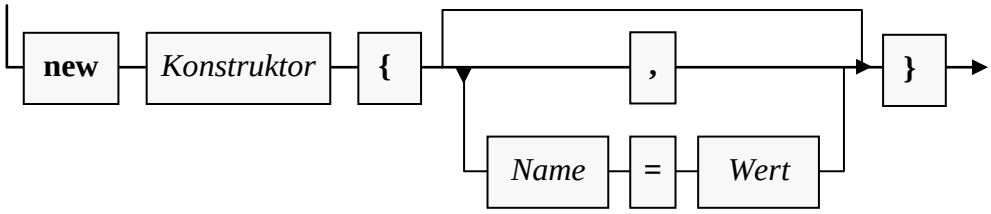
Dass der Name des Konstruktors weggelassen werden darf, gilt nicht nur für den parameterfreien Standardkonstruktor, sondern für jede Konstruktorüberladung, z. B.:

```
Bruch b = new(1, 2, "1/2");
```

5.4.3.3 Objektinitialisierer

Öffentliche Felder und Eigenschaften (siehe Abschnitt 5.5) können seit der C# - Version 3.0 bei der Objektkreation auch ohne spezielle Konstruktordefinition initialisiert werden. Dazu wird hinter den Konstruktoraufruf eine durch geschweifte Klammern begrenzte Liste von Name-Wert - Paaren gesetzt, die man als *Objektinitialisierer* bezeichnet.

Objektinitialisierer



In der `Bruch`-Klassendefinition könnte man auf den parametrisierten Konstruktor

```
public Bruch(int zpar, int npar, string epar) {
   Zaehler = zpar;
   Nenner  = npar;
   Etikett = epar;
}
```

verzichten und stattdessen Objektinitialisierer verwenden, wobei allerdings der Schreibaufwand für die Anwender der Klasse `Bruch` steigen würde, z. B.:

Quellcode	Ausgabe

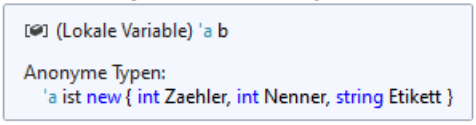
`using System;`
`class Bruchrechnung {`
`  static void Main() {`
`  var b = new Bruch() { Zaehler = 3,`
`  Nenner = 7,`
`  Etikett = "b" };`
`  b.Zeige();`
`  }`
`}`
	`  3`
	`b = -----`
	`  7`

Auch bei Verwendung eines Objektinitialisierers kommt ein Konstruktor zum Einsatz, wobei meist der *parameterfreie* Konstruktor verwendet wird. Dessen leere Parameterliste darf dann weggelassen werden, z. B.:

```
var b = new Bruch { Zaehler = 3, Nenner = 7, Etikett = "b" };
```

Gibt man *keinen* Konstruktor an, dann resultiert ein Objekt einer sogenannten *anonymen Klasse* (siehe Abschnitt 6.5), z. B.

```
var b = new  { Zaehler = 3, Nenner = 7, Etikett = "b" };
```



Um den Wert einer Eigenschaft (bzw. eines Felds) in einem Objektinitialisierer festzulegen, darf statt eines Literals auch ein Ausdruck verwendet werden, z. B.:

```
Bruch b1 = new Bruch(1, 2, "b1");
Bruch b2 = new Bruch { Zaehler = b1.Zaehler, Nenner = 7, Etikett = "b2" };
```

Das Beispiel illustriert auch die Vor- und Nachteile bei der *positionsorientierten* bzw. *namensorientierten* Initialisierung von Objekten (engl.: *positional* vs. *nominal creation*).

Durch die Objektinitialisierer werden explizite Konstruktoren keinesfalls überflüssig, denn die können ...

- beliebige Initialisierungsarbeiten ausführen und dazu Methoden aufrufen,
- Konsistenzbedingungen sicherstellen.

5.4.4 Abräumen überflüssiger Objekte durch den Garbage Collector

Wenn keine Referenz mehr auf ein Objekt zeigt, dann wird es bei passender Gelegenheit (z. B. bei Speichermangel) vom **Garbage Collector** (Müllsammler) der CLR entsorgt, und der belegte Speicher wird freigegeben.

Eine lokale Referenzvariable wird beim Verlassen ihres Deklarationsbereichs ungültig, also spätestens beim Beenden der Methode. Man kann eine Referenzvariable aktiv von einem Objekt „entkoppeln“, indem man ihr den Wert **null** (Verweis auf nichts) oder aber ein alternatives Referenzziel zuweist. Es ist jedoch durchaus möglich (und normal), dass ein Objekt die erzeugende Methode überlebt, weil eine Referenz nach Außen transportiert worden ist (z. B. per Rückgabewert, vgl. Abschnitt 5.4.5.2).

Zur Tätigkeit des Garbage Collectors können einige Anlässe führen, die von Richter (2012, Kap. 20) beschrieben werden (z. B. Speichermangel). In jedem Fall ist es für Entwickler kaum vorhersehbar, wann und in welcher Reihenfolge obsolete Objekte entsorgt werden. Es ist noch nicht einmal garantiert, dass der Garbage Collector tatsächlich tätig wird.[1]

Vermutlich sind Programmiereinsteiger vom Garbage Collector nicht sonderlich beeindruckt. Schließlich war im Manuskript noch nie die Rede davon, dass man sich um den belegten Speicher nach Gebrauch kümmern müsse. Der in einer Methode von lokalen Variablen belegte Speicher wird bei *jeder* Programmiersprache freigegeben, wenn die Ausführung der Methode endet. Demgegenüber muss der von überflüssig gewordenen *Objekten* belegte Speicher bei älteren Programmiersprachen (z. B. C++) nach Gebrauch explizit wieder freigegeben werden. In Anbetracht der Objektmassen, die manche Programme (z. B. ein Grafikeditor) benötigen, ist einiger Aufwand erforderlich, um eine Verschwendung von Speicherplatz zu verhindern. Mit seinem automatischen Garbage Collector vermeidet C# lästigen Aufwand und zwei kritische Fehlerquellen:

---

[1] https://ericlippert.com/2015/05/18/when-everything-you-know-is-wrong-part-one/

- Weil der Programmierer keine Verpflichtung (und Berechtigung) zum Entsorgen von Objekten hat, kann es nicht zu Programmabstürzen durch den Zugriff auf voreilig entsorgte Objekte kommen.
- Es entstehen keine **Speicherlöcher** (engl.: *memory leaks*) durch versäumte Speicherfreigaben bei überflüssig gewordenen Objekten.

Sollen die Objekte einer Klasse vor dem Entsorgen noch spezielle Aufräumaktionen durchführen, ist als Gegenstück zu den Konstruktoren ein sogenannter **Finalisierer** (alias: **Destruktor**) zu definieren, der ggf. vom Garbage Collector aufgerufen wird. Er trägt denselben Namen wie die Klasse, wobei das Tilde-Zeichen (~) voranzustellen ist.

Eine wichtige Tätigkeit von Destruktoren ist die Freigabe von Ressourcen, die *nicht* von der CLR verwaltet werden (z. B. Datei-, Netzwerk- oder Datenbankverbindungen). Weil wir solche Ressourcen noch nicht verwendet haben, beschränken wir uns auf ein inhaltsfreies Beispiel, das die Tätigkeit des Garbage Collectors am Programmende dokumentiert. Es ist allerdings nicht garantiert, dass der Garbage Collector tätig wird und die Destruktoren aufruft. In der **Main()** - Methode der Startklasse `KD` wird ein Objekt der Klasse `K2` erzeugt, die von der Klasse `K1` abstammt:

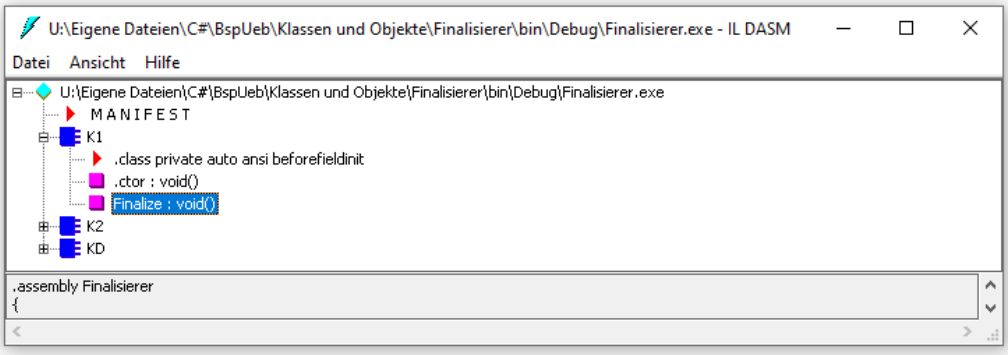
Quellcode	Ausgabe

```
using System;

class K1 {
   ~K1() {
      Console.WriteLine("K1-Finalisierer");
   }
}

class K2 : K1 {
   ~K2() {
      Console.WriteLine("K2-Finalisierer");
   }
}

class KD {
   static void Main() {
      K2 k2 = new K2();
   }
}
```

```
K2-Finalisierer
K1-Finalisierer
```

Es werden alle Finalisierer entlang des Stammbaums aufgerufen. Vom Finalisierer der Urahnklasse **Object** ist in der Ausgabe des Beispielprogramms nichts zu sehen, weil er keine Ausgabe macht (und auch sonst nichts tut).

Bei einer Assembly-Inspektion mit dem Hilfsprogramm **ILDasm** stellt man fest, dass der Finalisierer eigentlich den Namen **Finalize()** trägt:



In C# ist aber die Finalisierer-Syntax mit einer Tilde und dem Klassennamen zu verwenden.

Solange eine Klasse nur Speicher belegt, muss kein Finalisierer definiert werden. Verwendet eine Klasse hingegen auch sogenannte *unverwaltete Ressourcen* (z. B. Datei-, Netzwerk- oder Datenbankverbindungen), dann muss sich der Klassendesigner um die Freigabe dieser Ressourcen kümmern. Bei der von Microsoft empfohlenen Vorgehensweise kommt die Definition eines eigenen Finalisierers nur als letzter Ausweg vor.[1] In der folgenden Skizze der Empfehlungslage ist ein Vorgriff auf das Thema *Interfaces* erforderlich. Auch aus anderen Gründen klingen die Erläuterungen für Einsteiger sehr kompliziert, sodass Sie sich beim ersten Lesen des Manuskripts nicht allzu intensiv damit beschäftigen sollten:

- Verwendet eine Klasse unverwaltete Ressourcen (z. B. Datei-, Netzwerk- oder Datenbankverbindungen), dann sollte sie das Interface **IDisposable** implementieren, d.h. die von der Schnittstelle **IDisposable** geforderte Methode namens **Dispose()** implementieren. In dieser Methode sollten die von einem Objekt belegten unverwalteten Ressourcen freigegeben werden. Nutzer der Klasse sollten die **Dispose()** - Methode aufrufen, wenn ein Objekt nicht mehr benötigt wird und so für eine sofortige und zuverlässige Freigabe der Ressourcen sorgen (ohne Abhängigkeit vom Garbage Collector).
- Um für den Fall vorzusorgen, dass es ein Benutzer seiner Klasse versäumt, die Methode **Dispose()** aufzurufen, hat der Klassendesigner zwei Optionen, wobei unweigerlich eine Finalisierungsmethode durch den Garbage Collector aufgerufen wird. Die Objekt-Finalisierung ist u. a. deshalb komplex und fehleranfällig, weil der Garbage Collector nichtdeterministisch arbeitet und selbständig über Zeitpunkt und Reihenfolge seiner Aufräumaktionen entscheidet.
    - Bei der nach Möglichkeit zu bevorzugenden Technik sollten unverwaltete Ressourcen in ein Objekt aus einer Ableitung der Klasse **SafeHandle** im Namensraum **System.Runtime.InteropServices** verpackt werden, weil diese Klassen über eine robuste, auch unter ungünstigen Bedingungen korrekt arbeitende Finalisierungsmethode verfügen. Ein Klassendesigner kann sich in seiner eigenen **Dispose()** - Implementierung darauf beschränken, die **Dispose()** - Methode der **SafeHandle** - Ableitung aufzurufen. Wird für ein Objekt der neu entwickelten Klasse vom Nutzer der **Dispose()** - Aufruf versäumt, dann ruft der Garbage Collector die (besonders robuste) Finalisierungsmethode der **SafeHandle** -Ableitung auf.
    - Nur wenn keine **SafeHandle** - Verpackung für unverwaltete Ressourcen möglich ist, sollte ein Klassendesigner eine eigene Finalisierungsmethode implementieren und darin **Dispose()** aufrufen.

Im weiteren Kursverlauf wird der korrekte Aufruf der **Dispose()** - Methoden von BCL-Klassen mehrfach ein wichtiges Thema sein, während sich keine Notwendigkeit zur Definition eines Finalisierers zu einer eigenen Klasse ergeben wird. Wer doch einmal in diese Verlegenheit kommt, muss folgende Regeln beachten:[2]

- Ein Finalisierer trägt denselben Namen wie die Klasse, wobei das Tilde-Zeichen (~) voranzustellen ist.
- Ein Finalisierer liefert grundsätzlich *keinen* Rückgabewert, und es wird bei der Definition *kein* Typ angegeben.
- Pro Klasse ist nur *ein* Finalisierer erlaubt. Dieser muss eine leere Parameterliste haben.
- Finalisierer können nicht direkt aufgerufen werden. Es ist ausschließlich der automatische Aufruf durch den Garbage Collector vorgesehen.

---

1 https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/unmanaged

2 https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/destructors

- Es ist nicht garantiert, dass der Garbage Collector tätig wird und einen Finalizer aufruft.
- Bei der Definition eines Finalisierers sind Modifikatoren überflüssig und verboten.
- Finalisierer werden nicht vererbt.
- Es ist nicht garantiert, dass ein Finalisierer tatsächlich aufgerufen wird.

Der C# - Insider Eric Lippert warnt auf einer Stackoverflow-Seite:[1]

> Implementing a destructor correctly is one of the hardest things to do in C# in all but the most trivial cases.

5.4.5 Objektreferenzen verwenden

In diesem Abschnitt geht es um Wertparameter und Methodenrückgaben mit Referenztyp sowie um das Schlüsselwort **this**, mit dem man in einer Methode das aktuell handelnde Objekt ansprechen kann.

5.4.5.1 Objektreferenzen als Wertparameter

Wir haben schon festgehalten, dass die formalen Wertparameter einer Methode wie *lokale Variablen* funktionieren, die beim Methodenaufruf mit den Werten der Aktualparameter initialisiert worden sind. Methodeninterne Änderungen bei den Werten dieser lokalen Variablen wirken sich *nicht* auf die eventuell als Wertaktualparameter verwendeten Variablen der rufenden Methode aus. Bei einem Wertparameter mit *Referenztyp* wird ebenfalls der Wert des Aktualparameters (eine Objektreferenz) beim Methodenaufruf in eine lokale Variable *kopiert*. Es wird jedoch keinesfalls eine Kopie des referenzierten Objekts (auf dem Heap) erstellt, sodass die aufgerufene Methode über ihre lokale Referenzvariable auf das Originalobjekt zugreift und dort ggf. Veränderungen vornimmt.[2]

Von den beiden `Addiere()` - Überladungen der Klasse `Bruch` verfügt die ältere Variante über einen Wertparameter mit Referenztyp:

```
public void Addiere(Bruch b) {
   zaehler = zaehler*b.nenner + b.zaehler*nenner;
   nenner = nenner*b.nenner;
   Kuerze();
}
```

Mit dem Aufruf dieser Methode wird ein Objekt beauftragt, den via Parameter spezifizierten Bruch zum eigenen Wert zu addieren und das Resultat gleich zu kürzen.

Zähler und Nenner des „fremden“ `Bruch`-Objekts können per Parametername und Punktoperator trotz Schutzstufe **private** direkt angesprochen werden, weil der Zugriff in einer `Bruch`-Methode stattfindet. Hier liegt *kein* Verstoß gegen das Prinzip der Datenkapselung vor, weil der Zugriff durch eine klasseneigene Methode erfolgt, die vom Klassendesigner gut konzipiert sein sollte.

Dass in einer `Bruch`-Methodendefinition ein Parameter vom Typ `Bruch` verwendet wird, ist übrigens weder „zirkulär“ noch ungewöhnlich; schließlich sollen Brüche auch mit ihresgleichen interagieren können.

---

[1] https://stackoverflow.com/questions/8174347/inheritance-and-destructors-in-c-sharp

[2] Wertparameter mit Referenztyp arbeiten also analog zu den Referenzparametern (vgl. Abschnitt 5.3.1.3.2), insofern beide einer Methode Einwirkungen auf die Außenwelt ermöglichen. Die Referenzparameter sind in C# vor allem deshalb aufgenommen worden, um den Zeit und Speicherplatz sparenden *call by reference* auch bei den Instanzen von Werttypen zu ermöglichen. Wir werden mit den sogenannten Strukturen noch Werttypen kennenlernen, die ähnlich umfangreich sein können wie Klassen.

In obiger `Addiere()` - Überladung bleibt das per Parameter ansprechbare `Bruch`-Objekt unverändert. Sofern entsprechende Zugriffsrechte vorliegen, was bei Parametern vom Typ des agierenden Objekts stets der Fall ist, kann eine Methode das Parameter-Objekt aber auch verändern. Als Beispiel erweitern wir die `Bruch`-Klasse um die Methode `DuplWerte()`, die ein Objekt beauftragt, seinen Zähler und Nenner auf ein anderes `Bruch`-Objekt zu übertragen, das per Wertparameter vom Typ `Bruch` bestimmt wird:

```
public void DuplWerte(Bruch bc) {
   bc.zaehler = zaehler;
   bc.nenner = nenner;
}
```

Im folgenden Programm wird das `Bruch`-Objekt `b1` beauftragt, die `DuplWerte()` - Methode auszuführen, wobei als Parameter eine Referenz auf das Objekt `b2` übergeben wird:

Quellcode	Ausgabe

`using System;`
`class Bruchrechnung {`
`  static void Main() {`
`  var b1 = new Bruch(1, 2, "b1");`
`  var b2 = new Bruch(5, 6, "b2");`
 	
`  b1.Zeige();`
`  b2.Zeige();`
`  b1.DuplWerte(b2);`
`  Console.WriteLine("Nach DuplWerte():\n");`
`  b2.Zeige();`
`  }`
`}`
	`  1`
	`b1 =  -----`
	`  2`
	
	`  5`
	`b2 =  -----`
	`  6`
	
	`Nach DuplWerte():`
	
	`  1`
	`b2 =  -----`
	`  2`

5.4.5.2 Rückgabewerte mit Referenztyp

Soll ein methodenintern erzeugtes Objekt das Ende der Methodenausführung überleben, muss eine Referenz außerhalb der Methode geschaffen werden, was z. B. über einen Rückgabewert mit Referenztyp geschehen kann.

Zur Demonstration des Verfahrens erweitern wir die Klasse `Bruch` um die Methode `Klone()`, welche ein Objekt beauftragt, einen neuen Bruch anzulegen, mit den Werten der eigenen Instanzvariablen zu initialisieren und die Adresse an den Aufrufer zu übergeben:[1]

```
public Bruch Klone() {
   return new Bruch(zaehler, nenner, etikett);
}
```

Im folgenden Beispiel wird das durch `b2` referenzierte `Bruch`-Objekt in der von `b1` ausgeführten Methode `Klone()` erstellt:

[1] Bei einer für die breitere Öffentlichkeit gedachten Klasse sollte auch eine die Schnittstelle **ICloneable** (siehe Kapitel 9) implementierende Vervielfältigungsmethode angeboten werden, obwohl diese Schnittstelle durch semantische Unklarheit von begrenztem Wert ist, was im Kapitel 9 über Schnittstellen (Interfaces) näher erläutert wird.

Quellcode	Ausgabe

`using System;`
`class Bruchrechnung {`
`  static void Main() {`
`  var b1 = new Bruch(1, 2, "b1");`
`  b1.Zeige();`
`  var b2 = b1.Klone();`
`  b2.Zeige();`
`  }`
`}`
	`  1`
	`b1 =  -----`
	`  2`
	
	`  1`
	`b1 =  -----`
	`  2`

5.4.5.3 this als Referenz auf das aktuelle Objekt

Gelegentlich ist es sinnvoll oder erforderlich, dass ein handelndes Objekt sich selbst ansprechen bzw. seine eigene Adresse als Methodenaktualparameter verwenden kann. Das ist mit dem Schlüsselwort **this** möglich, das innerhalb einer Instanzmethode wie eine Referenzvariable funktioniert. Im folgenden Beispiel ermöglicht die **this**-Referenz die Verwendung von Formalparameternamen, die mit den Namen von Instanzvariablen übereinstimmen:

```
public bool Addiere(int zaehler, int nenner, bool autokurz) {
  if (nenner != 0) {
    this.zaehler = this.zaehler * nenner + zaehler * this.nenner;
    this.nenner = this.nenner * nenner;
    if (autokurz)
      this.Kuerze();
    return true;
  } else
    return false;
}
```

Außerdem wird beim `Kuerze()` - Aufruf durch die (nicht erforderliche) **this**-Referenz verdeutlicht, dass die Methode vom aktuell handelnden Objekt ausgeführt wird. Später werden Sie noch weit relevantere **this**-Verwendungsmöglichkeiten kennenlernen.

5.5 Eigenschaften

5.5.1 Syntaktisch elegante Zugriffsmethoden

Sollen fremde Klassen Lese- und/oder Schreibzugriff auf ein gekapseltes (also privates) Feld erhalten, sind entsprechende Zugriffsmethoden zu definieren. Im `Bruch`-Beispiel könnte man z. B. für das `nenner`-Feld die folgenden Methoden definieren:

```
public int GibNenner() {
  return nenner;
}

public void SetzeNenner(int value) {
  if (value != 0)
    nenner = value;
}
```

Infolgedessen sähe der klassenfremde Zugriff auf einen Nenner z. B. so aus:

```
b1.SetzeNenner(2);
Console.WriteLine(b1.GibNenner());
```

Mit den *Eigenschaften* (engl.: *properties*), die wegen ihrer großen Bedeutung schon in der ersten Variante des `Bruch`-Beispiels genutzt wurden, bietet C# die Möglichkeit, Zugriffe auf gekapselte

Felder syntaktisch zu vereinfachen. Aus den beiden obigen Methodendefinitionen wird die folgende Eigenschaftsdefinition:

```
public int Nenner {
  get {
    return nenner;
  }
  set {
    if (value != 0)
      nenner = value;
  }
}
```

Für den Klassen*designer* ändert sich nicht allzu viel: Im Rahmen einer Eigenschaftsdefinition mit Modifikatoren, Datentyp und Namen sind ein **get**- und ein **set**-Block mit naheliegender Syntax zu implementieren (siehe Syntaxdiagramm im Abschnitt 4.1.2.3).[1]

Erwähnenswert ist, dass im **set**-Block der vom Aufrufer übergebene neue Wert ohne Formalparameterdefinition über das Schlüsselwort **value** angesprochen wird.

Für den Klassen*anwender* ändert sich mehr, weil eine Eigenschaft intuitiver zu verwenden ist als korrespondierende Zugriffsmethoden, z. B.:

```
b1.Nenner = 2;
Console.WriteLine(b1.Nenner);
```

Sogar die Aktualisierungs-Operatoren (vgl. Abschnitt 4.5.8) werden unterstützt, z. B.:

```
b1.Nenner += 2;
```

Um aus der Perspektive fremder Klassen eine **get-only** - oder **set-only** - **Eigenschaft** zu realisieren, verzichtet man einfach auf die **set**- bzw. **get**-Implementation. Durch die folgende Variante der `Nenner`-Definition aus der Klasse `Bruch` entsteht eine Eigenschaft, die nur den lesenden Zugriff erlaubt, wobei die klasseneigenen Methoden nach wie vor auf die private Instanzvariable `nenner` schreibend zugreifen dürfen:

```
public int Nenner {
    get {
        return nenner;
    }
}
```

Es ist möglich, den für eine Eigenschaft gültigen Zugriffsschutz entweder für den **get**- oder für den **set**-Zugriff zu verschärfen, aber nicht für beide gleichzeitig. Durch die folgende Variante der `Nenner`-Definition aus der Klasse `Bruch` entsteht eine Eigenschaft, die den Lesezugriff für beliebige Klassen erlaubt, den Schreibzugriff hingegen auf die klasseneigenen Methoden beschränkt:

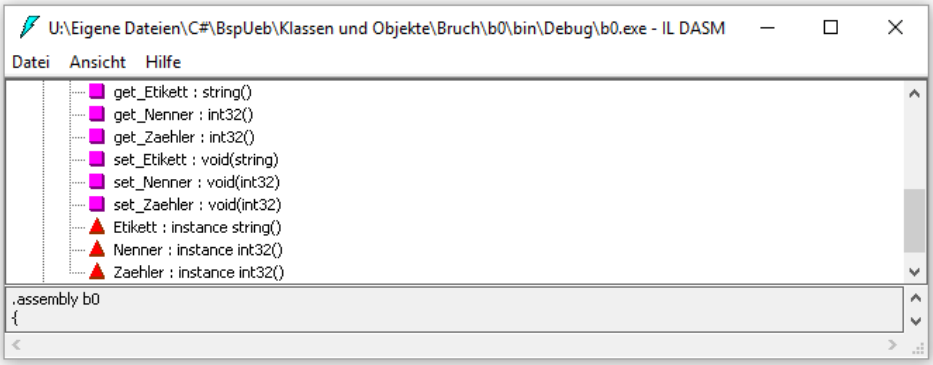
---

[1] Das Syntaxdiagramm im Abschnitt 4.1.2.3 verschweigt die Möglichkeit, in einem **get**-Block neben der obligatorischen **return**-Anweisung noch weitere Anweisungen unterzubringen und dabei sogar den Zustand des handelnden Objekts zu verändern, z. B.:

```
public double X { get { x = 13.0; return x; } set { x = value; } }
```

```
public int Nenner {
    get {
        return nenner;
    }
    private set {
        if (value != 0)
            nenner = value;
    }
}
```

Wie eine Assembly-Inspektion mit dem Hilfsprogramm **ILDasm** zeigt, erstellt der Compiler zu den Eigenschaften unserer Klasse `Bruch` jeweils ein Paar von Zugriffsmethoden:



Wenngleich die Eigenschaften praktisch sind, handelt es sich doch eher um *syntactic sugar* (Mössenböck 2019, S. 3) als um einen essentiellen Vorteil gegenüber anderen Programmiersprachen wie Java oder C++.[1]

5.5.2 Automatisch implementierte Eigenschaften

5.5.2.1 Routinearbeit an den Compiler delegieren

Bei Eigenschaften, die lediglich ein privates Feld kapseln und auf jeden Eingriff beim Lesen und Schreiben verzichten kommt man seit der C# - Version 3.0 mit einem minimalen Definitionsaufwand aus. Man kann sich auf Modifikatoren, Typ, Namen sowie die Schlüsselwörter **get** und **set** beschränken. Die **get**- bzw. **set**-Implementation kann man ebenso dem Compiler überlassen wie die Deklaration des gekapselten Felds. Bei der `Zaehler`-Eigenschaft unserer Klasse `Bruch` könnten also die Deklaration

```
int zaehler;
```

und die Definition

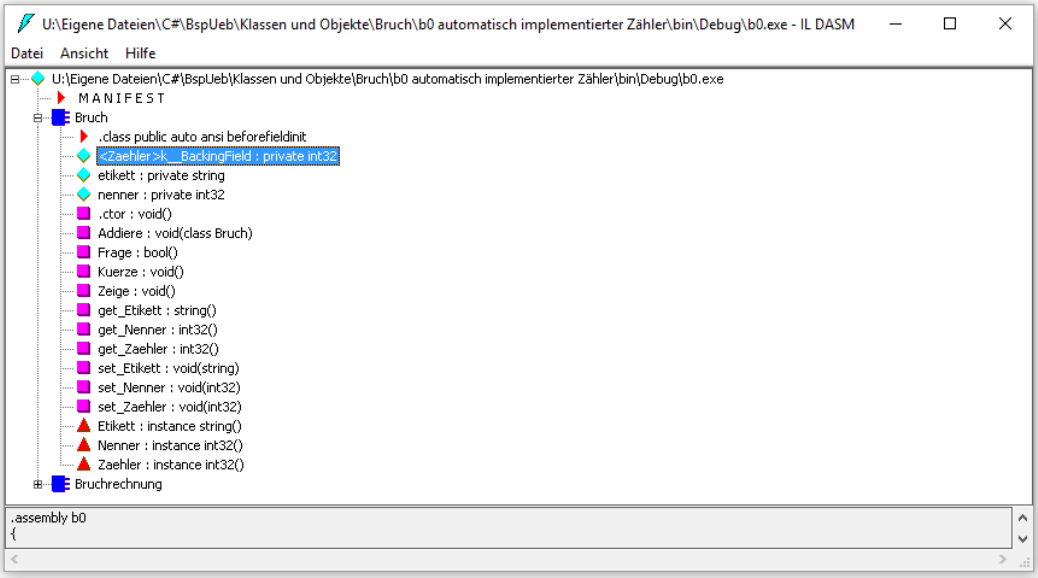
[1] Beim schreibenden Zugriff geht im Vergleich zu einer traditionellen **set**-Methode die Möglichkeit verloren, durch eine Rückgabe vom Typ **bool** zu signalisieren, ob die gewünschte Wertzuweisung durchgeführt wurde. Allerdings hat eine **set**-Methode per Konvention den Rückgabetyp **void**, und eine **bool**-Rückgabe wird von fremden Programmierern eventuell ignoriert. Als Warnung vor einer nicht ausgeführten Wertzuweisung sollte eher eine Ausnahme geworfen werden (siehe unten), was natürlich auch in der **set**-Implementation einer Eigenschaft möglich ist.

```
public int Zaehler {
  get {
    return zaehler;
  }
  set {
    zaehler = value;
  }
}
```

äquivalent ersetzt werden durch die Definition:

```
public int Zaehler {get; set;}
```

Wie das Hilfsprogramm **ILDasm** zeigt, ergänzt der Compiler automatisch ein privates **backing field** mit passendem Typ:



Dieses Feld ist ausschließlich über die Eigenschaft ansprechbar, sodass bei einem Fehler relativ gute Voraussetzungen für die Ursachensuche bestehen.

Mit der C# - Version 6.0 sind für automatisch implementierte Eigenschaften (engl. Bezeichnung *Auto-Implemented Properties*) zwei Verbesserungen eingeführt worden:

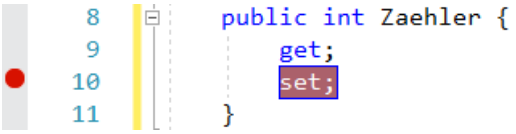
- Man kann einen Initialisierungswert vereinbaren, z. B.:
  ```
  public int Auto { get; set; } = 4711;
  ```
- Man kann sich auf den **get**-Zugriff beschränken, z. B.:
  ```
  public int Auto { get; } = 4711;
  ```
  Die in diesem Fall meist unverzichtbare Initialisierung kann bei der Deklaration erfolgen (siehe Beispiel) oder in einem Konstruktor, z. B.:
  ```
  public Bsp() {
      Auto = 4711;
  }
  ```
  Bei einer Eigenschaft ohne **set** ist nach der Initialisierung kein weiterer Schreibzugriff mehr möglich.

Im `Bruch`-Einstiegsbeispiel wird der Klarheit halber auf automatisch implementierte Eigenschaften verzichtet.

5.5.2.2 Automatisch implementierte Eigenschaft im Vergleich zu Feldern

Auf den ersten Blick unterscheidet sich eine automatisch implementierte öffentliche Eigenschaft mit **get**- *und* **set**-Methode nur durch den leicht erhöhten Schreibaufwand von einem öffentlichen Feld. Eine genauere Analyse fördert aber doch Vorteile der Eigenschaft zu Tage:[1]

- Eine automatisch implementierte Eigenschaft erleichtert die Fehlersuche. Weil auf ihr Backing Field nicht direkt zugegriffen werden kann, genügt ein einziger Haltepunkt, um alle Schreibzugriffe beobachten zu können, z. B.:



  Wenn hingegen mehrere Methoden auf ein Feld schreibend zugreifen, sind entsprechend viele Haltepunkte erforderlich.
  Ab .NET Core 3.0 (und auch in .NET 5) können im Visual Studio 2019 allerdings im Debug-Modus **Datenhaltepunkte** (engl.: *data break points*) gesetzt werden, die das Programm stoppen, sobald bei einem bestimmten Objekt ein bestimmtes Feld geändert wird. Dazu wählt man im **Auto**- oder **Lokal**-Fenster aus dem Kontextmenü zu einem Feld eines Objekts das Item **Bei Werteänderungen unterbrechen**. Man hat zwei Vorteile gegenüber einem Code-Haltepunkt:
  - Der Stopp ist nicht an eine Quellcode-Stelle gebunden, sondern an eine Instanzvariable. Es ist also gleichgültig, welche Anweisung die Wertveränderung ausgelöst hat.
  - Sind mehrere Objekte einer Klasse vorhanden, dann stoppt ein **set**-Haltepunkt bei *jedem* Objekt, während per Datenhaltepunkt ein konkretes Objekt beobachtet wird.

  Allerdings verursacht ein Datenhaltepunkt beim Debugging einen höheren Zeitaufwand als ein Code-Haltepunkt.
- Bei einem späteren Wechsel von einem öffentlichen Feld zu einer öffentlichen Eigenschaft müssten kooperierende Klassen neu übersetzt werden. Ein derartiges Innovations-Hindernis besteht nicht bei einer späteren Modifikation einer Eigenschaft.
- In einer Schnittstelle (siehe Kapitel 9) kann von einem implementierenden Typ die Existenz von öffentlichen Eigenschaften verlangt werden, aber nicht die Existenz von öffentlichen Feldern.

5.5.3 Objektinitialisierer und init - Setter

Für Eigenschaften mit **set**-Methode ist kein Konstruktor-Parameter zur Initialisierung bei der Objektkreation erforderlich, weil der Objektinitialisierer dazu verwendet werden kann (siehe Abschnitt 5.4.3.3), z. B.:

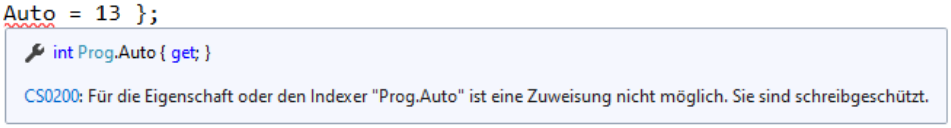
```
using System;
public class Prog {
    public int Auto { get; set; }
    static void Main() {
        Prog p = new Prog() { Auto = 13 };
         . . .
    }
}
```

Soll eine Eigenschaft nach der Initialisierung *unveränderlich* sein, scheidet diese Konstruktor-freie Initialisierung aber aus, weil der Objektinitialisierer nur veränderliche Eigenschaften versorgen kann.

---

[1] Die aufgelisteten und weitere Vorteilen von Eigenschaften werden hier beschrieben: https://stackoverflow.com/questions/1180860/public-fields-versus-automatic-properties

```
public int Auto { get; }
static void Main() {
    var p = new Prog() { Auto = 13 };
```



Seit der C# - Version 9.0 ist dieses Problem behoben durch die **init-only - Eigenschaften**, die mit Hilfe des Schlüsselworts **init** deklariert werden. Im Beispiel klappt nun die Objektkreation mit Initialisierung, während spätere Wertänderungen verhindert werden:

```
public int Auto { get; init; }
static void Main() {
    var p = new Prog() { Auto = 13 };
    p.Auto = 4711;
```

Das nächste Beispiel definiert die Eigenschaften `FirstName` und `LastName` auf traditionelle Art und verwendet dabei jeweils eine **get**- und eine **init**-Methode:

```
public class Person {
    readonly string firstName = "unbekannt";
    readonly string lastName = "unbekannt";
    readonly bool nameInit;

    public string FirstName {
        get { return firstName; }
        init {
            if (value != null)
                firstName = value;
        }
    }
    public string LastName {
        get { return lastName; }
        init {
            if (value != null)
                lastName = value;
            nameInit = true;
        }
    }
}
```

Das Beispiel demonstriert, dass (analog zu **get** und **set**) auch hinter **init** eine Methode steckt, die potentiell mehrere Anweisungen enthalten kann. Weil diese Methode nur bei der Initialisierung einer Eigenschaft zum Einsatz kommt, darf sie auf (beliebige) **readonly**-Instanzvariablen zugreifen.

5.5.4 Zeitaufwand bei Eigenschafts- und Feldzugriffen

Microsoft verspricht, dass Eigenschaftszugriffe aufgrund von Optimierungen des Just-In-Time - Compilers der CLR in der Regel *nicht* aufwändiger sind als Feldzugriffe.[1]

> Generally, because of just-in-time optimizations, properties are no more expensive than fields.

Vom JIT-Compiler der CLR ist zu erwarten, dass er den (meist sehr kleinen) Maschinencode an jeder Aufrufstelle einsetzt, um bei Eigenschaftszugriffen den Aufwand eines gewöhnlichen Methodenaufrufs zu vermeiden. Diese auch von anderen Compilern eingesetzte Technik zur Optimierung von Funktions- bzw. Methodenaufrufen bezeichnet man als **Inlining**.

---

[1] Siehe z. B. http://msdn.microsoft.com/en-us/library/65zdfbdt.aspx

Um die obige Aussage von Microsoft zu überprüfen, wurden im Bruchrechnungsbeispiel für den lesenden und schreibenden Eigenschafts- bzw. Feldzugriff auf den Zähler bzw. Nenner eines Bruchs Vergleichsmessungen unter den folgenden Bedingungen ausgeführt:

- .NET-Framework 4.8
- Windows 10 (64 Bit)
- Rechner mit Intel-CPU Core i3 550
- Erstellungskonfiguration: Release
  Zum Unterschied zwischen Release- und Debug-Konfiguration siehe Abschnitt 3.3.4.4.

Die Messergebnisse bestätigen Microsofts Aussage:

a) Eigenschafts- bzw. Feldzugriff auf den Zähler eines `Bruch`-Objekts:

Zugriffsart	Zeitaufwand für 100 Millionen Zugriffe in Millisekunden	
	via Eigenschaft	direkter Feldzugriff
Lesen	63,0092	66,9489
Schreiben	63,9913	64,0449

b) Eigenschafts- bzw. Feldzugriff auf den Nenner eines `Bruch`-Objekts:

Zugriffsart	Zeitaufwand für 100 Millionen Zugriffe in Millisekunden	
	via Eigenschaft	direkter Feldzugriff
Lesen	64,0446	64,9971
Schreiben	64,9988	64,0166

In der Debug-Konfiguration dauern lesende und schreibende Zugriffe erheblich länger:

- über Eigenschaften ca. 600 Millisekunden
- über direkte Feldzugriffe ca. 300 Millisekunden

Schreibende Feldzugriffe auch in klasseneigenen Methoden durch Eigenschaftszugriffe zu ersetzen, bringt ohne Performanz-Beeinträchtigung (in der Release-Konfiguration) einige Vorteile:

- Maßnahmen zur Sicherung der Objektkonsistenz sind leichter realisierbar.
- Im Multi-Threading - Betrieb ist die Koordination von mehreren, schreibend zugreifenden Threads leichter realisierbar (siehe Abschnitt 17.1.4.1.1).
- Beim Debugging ist man nicht auf einen Datenhaltepunkt angewiesen, sondern kann einen (weniger aufwändigen) Code-Haltepunkt verwenden.

Es wird daher explizit empfohlen, schreibende Feldzugriffe auch in klasseneigenen Methoden durch Eigenschaftszugriffe zu ersetzen.[1]

In mehreren Methoden der Klasse `Bruch` sind aktuell noch schreibende Feldzugriffe zu finden, z. B.:

```
public void Addiere(Bruch b) {
   zaehler = zaehler*b.nenner + b.zaehler*nenner;
   nenner = nenner*b.nenner;
   Kuerze();
}
```

In der folgenden Variante der obigen `Addiere()` - Methode sind die schreibenden Feldzugriffe durch Eigenschaftszugriffe ersetzt:

[1] Es kann leider nicht garantiert werden, dass im Manuskript ab jetzt keine schreibenden Feldzugriffe mehr zu sehen sind.

```
public void Addiere(Bruch b) {
   Zaehler = zaehler*b.nenner + b.zaehler*nenner;
   Nenner = nenner*b.nenner;
   Kuerze();
}
```

5.6 Statische Member und Klassen

Neben den *objektbezogenen* Feldern, Eigenschaften, Methoden und Konstruktoren unterstützt C# auch *klassenbezogene* Varianten. Syntaktisch werden diese Member in der Deklaration bzw. Definition durch den Modifikator **static** gekennzeichnet, und man spricht oft von *statischen* Feldern, Methoden, Eigenschaften etc. Ansonsten gibt es bei der Deklaration bzw. Definition kaum Unterschiede zwischen einem Instanz-Mitglied und dem analogen statischen Mitglied.

Auch bei den statischen Mitgliedern gilt für den Zugriffsschutz:

- Voreingestellt ist die Schutzstufe **private**, sodass eine Verwendung nur in klasseneigenen Methoden erlaubt ist.
- Durch Modifikatoren kann eine alternative Schutzstufe festgelegt werden (z. B. **public**).

5.6.1 Statische Felder und Eigenschaften

In unserem Bruchrechnungsbeispiel soll ein statisches Feld die Anzahl der bisher im aktuellen Programmlauf erzeugten `Bruch`-Objekte aufnehmen:

```
using System;
public class Bruch {
  int zaehler,
      nenner = 1;
  string etikett = "";

  static int anzahl;

  . . .

  public static int Anzahl {
    get {
      return anzahl;
   }
    private set {
      anzahl = value;
    }
  }

  public Bruch(int zpar, int npar, String epar) {
    Zaehler = zpar;
    Nenner  = npar;
    Etikett = epar;
    Anzahl++;
  }

  public Bruch() {
    Anzahl++;
  }

  . . .

}
```

Ein statisches Feld kann in klasseneigenen Methoden (objektbezogen oder statisch) direkt angesprochen werden. Sofern Methoden *fremder* Klassen (durch den Modifikator **public**) der direkte Zugriff auf eine Klassenvariable gewährt wird, müssen diese dem Variablennamen einen Vorspann aus Klassennamen und Punktoperator voranstellen, z. B.:

```
Console.WriteLine("Bisher wurden " + Bruch.anzahl + " Brüche erzeugt");
```

In unserem Beispiel wird das statische Feld `anzahl` aber *ohne* **public**-Modifikator deklariert, sodass der direkte Zugriff klasseneigenen Methoden vorbehalten bleibt.

Damit im erweiterten Bruchrechnungsbeispiel fremde Klassen trotz Datenkapselung die Anzahl der bisher erzeugten `Bruch`-Objekte in Erfahrung bringen können, wird eine statische Eigenschaft namens `Anzahl` mit **public**-Zugriff ergänzt. Weil die **set**-Funktionalität als **private** deklariert ist, können fremde Klassen den `anzahl`-Wert zwar ermitteln, aber nicht verändern.

Es sprechen einige Argumente dafür, auch in den Methoden der Klasse `Bruch` den privaten **setter** der Eigenschaft `Anzahl` statt direkter Feldzugriffe zu verwenden (siehe Abschnitt 5.5.4). Im Beispiel wird die (automatisch auf 0 initialisierte) Klassenvariable `anzahl` in den beiden Instanzkonstruktoren über die Eigenschaft `Anzahl` inkrementiert.

Während jedes Objekt einer Klasse über einen eigenen Satz mit allen Instanzvariablen verfügt, existiert eine klassenbezogene Variable nur *einmal*. Sie wird beim Laden der Klasse angelegt und erhält per Voreinstellung dieselbe Null-Initialisierung wie eine Instanzvariable (vgl. Abschnitt 5.2.3). Alternative Initialisierungen können in der Variablendeklaration oder im statischen Konstruktor (siehe Abschnitt 5.6.4) vorgenommen werden.

In der folgenden Tabelle werden wichtige Unterschiede zwischen Klassen- und Instanzvariablen zusammengestellt:

	Instanzvariablen	**Klassenvariablen**
Deklaration	ohne Modifikator **static**	mit Modifikator **static**
Zuordnung	Jedes Objekt besitzt einen eigenen Satz mit allen Instanzvariablen.	Klassenbezogene Variablen sind nur *einmal* vorhanden.
Existenz	Instanzvariablen werden beim Erzeugen des Objektes angelegt und initialisiert. Sie werden ungültig, wenn das Objekt nicht mehr referenziert ist.	Klassenvariablen werden beim Laden der Klasse angelegt und initialisiert.[1]

5.6.2 Wiederholung zur Kategorisierung von Variablen

Mittlerweile haben wir verschiedene Variablensorten kennengelernt, wobei die Sortenbezeichnung unterschiedlich motiviert war. Um einer möglichen Verwirrung vorzubeugen, folgt nun eine Zusammenfassung bzw. Wiederholung. Die folgenden Begriffe sollten Ihnen keine Probleme mehr bereiten:

[1] Entladen kann man eine Klasse nur zusammen mit ihrer sogenannten *Anwendungsdomäne*. Das wird im Manuskript nicht behandelt.

- **Lokale Variablen ...**
  werden in Methoden deklariert,
  landen auf dem Stack,
  werden **nicht** automatisch initialisiert,
  sind gültig von der Deklaration bis zum Ende des Blocks, der die Deklaration enthält.
- **Instanzvariablen ...**
  werden außerhalb jeder Methode deklariert,
  landen (als Bestandteile von Objekten) auf dem Heap,
  werden automatisch mit dem typspezifischen Nullwert initialisiert,
  sind verwendbar, wo eine Referenz zum Objekt vorliegt und Zugriffsrechte bestehen.
- **Klassenvariablen ...**
  werden außerhalb jeder Methode mit dem Modifikator **static** deklariert,
  werden automatisch mit dem typspezifischen Nullwert initialisiert,
  sind verwendbar, wo Zugriffsrechte bestehen.
- **Referenzvariablen ...**
  zeichnen sich durch ihren speziellen *Inhalt* aus (Referenz auf ein Objekt). Es kann sich sowohl um lokale Variablen (z. B. `b1` in der **Main()** - Methode von `Bruchrechnung`) als auch um Instanzvariablen (z. B. `etikett` in der `Bruch`-Definition) oder um Klassenvariablen handeln.

Die Variablen in C# kann man einteilen nach ...

- **Datentyp**
  Es sind vor allem zu unterscheiden:
  - Werttypen (z. B. **int**, **double**, **bool**)
  - Referenztypen (mit Objektreferenzen als Inhalt).
- **Zuordnung**
  Eine Variable kann zu einem Objekt (Instanzvariable), zu einer Klasse (statische Variable) oder zu einer Methode (lokale Variable) gehören. Damit sind weitere Eigenschaften wie Ablageort, Lebensdauer, Sichtbarkeitsbereich und Initialisierung festgelegt.

5.6.3 Statische Methoden

Es ist in vielen Situationen sinnvoll oder sogar unvermeidlich, einer *Klasse* Handlungskompetenzen (Methoden) zu verschaffen. So muss z. B. beim Programmstart die **Main()** - Methode der Startklasse ausgeführt werden, bevor irgendein Objekt existiert. Das Erzeugen von Objekten gehört zu den typischen Aufgaben der statischen Methode **Main()**, wobei es sich nicht unbedingt um Objekte der eigenen Klasse handeln muss.

Wie eine statische (und öffentliche) Methode von fremden Klassen genutzt werden kann, ist Ihnen längst bekannt, weil die statische Methode **WriteLine()** der Klasse **Console** bisher in fast jedem Konsolenprogramm zum Einsatz kam, z. B.:

```
Console.WriteLine("Hallo");
```

Vor den Namen der gewünschten Methode setzt man (durch den Punktoperator getrennt) den Namen der angesprochenen Klasse, der eventuell durch den Namensraumbezeichner vervollständigt werden muss, je nach Namensraumzugehörigkeit der Klasse und vorhandenen **using**-Direktiven am Anfang des Quellcodes.

Trotz Ihrer Erfahrung mit diversen **Main()** - Methoden soll auch im Kontext unserer Klasse `Bruch` das Definieren einer statischen Methode geübt werden. Zur Vereinfachung von Anweisungsfolgen nach dem folgenden Muster

```
var b = new Bruch(0, 1, "Startwert");
b.Frage();
b.Kuerze();
```

definieren wir eine Klassenmethode namens `BenDef()`, die eine Referenz auf ein neues `Bruch`-Objekt mit benutzerdefinierten und gekürzten Werten liefert:

```
public static Bruch BenDef(string e) {
  var b = new Bruch(0, 1, e);
  if (b.Frage()) {
    b.Kuerze();
    return b;
  } else
    return null;
}
```

Bei fehlerhaften Benutzereingaben liefert die Methode den Referenzwert **null** zurück. Mit Hilfe der neuen Methode kann die obige Sequenz durch eine einzelne Anweisung ersetzt werden:

Quellcode	Eingaben (fett) und Ausgabe

`using System;`
`class Bruchrechnung {`
`  static void Main() {`
`  var b = Bruch.BenDef("Startwert");`
`  if (b != null)`
`  b.Zeige();`
`  else`
`  Console.WriteLine("b zeigt auf null");`
`  }`
`}`
	`Zähler: 26`
	`Nenner : 39`
	`  2`
	`Startwert = -----`
	`  3`

Wird eine Klassenmethode von anderen Methoden der *eigenen* Klasse (objekt- oder klassenbezogen) verwendet, muss der Klassenname *nicht* angegeben werden. Es ist aber erlaubt und kann der Klarheit dienen.

Weil der Modifikator **static** nicht Signatur-relevant ist (vgl. Abschnitt 5.3.5) kann es in einer Klasse zu einer Instanzmethode keine statische Methode mit demselben Namen und derselben Parameterliste geben.

In früheren Abschnitten waren mit *Methoden* stets *objektbezogene* Methoden (*Instanzmethoden*) gemeint. Dies soll auch weiterhin so gelten.

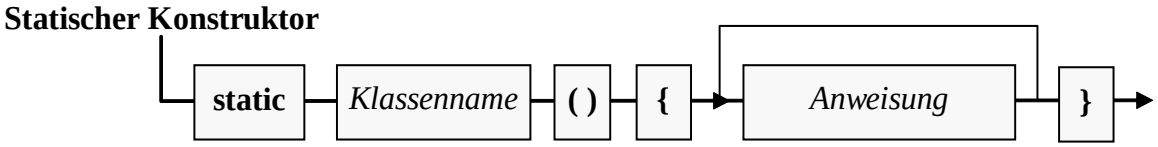
5.6.4 Statische Konstruktoren

Analog zu den Instanzkonstruktoren (siehe Abschnitt 5.4.3), die beim Erzeugen eines Objekts ausgeführt werden und das Objekt auf den Einsatz vorbereiten (z. B. durch die Initialisierung von Instanzvariablen), kann für jede Klasse ein statischer Konstruktor zur Initialisierung von Klassenvariablen und andere Arbeiten definiert werden. Er wird beim Laden der Klasse (also beim Erstellen des ersten Objekts oder beim ersten Zugriff auf ein statisches Mitglied) automatisch ausgeführt und kann nirgends explizit aufgerufen werden.

Eine statische Variable (ohne **const**-Modifikator, siehe Abschnitt 5.2.5) durchläuft folgende Initialisierungen:

- Zunächst erhält sie den typspezifischen Nullwert (vgl. Abschnitt 5.2.3).
- Ggf. wird anschließend die Initialisierung aus der Deklarationsanweisung vorgenommen.
- Schließlich wird der statische Konstruktor ausgeführt.[1]

Naheliegenderweise ist pro Klasse nur *ein* statistischer Konstruktor erlaubt, und seine Parameterliste muss leer bleiben. In der Definition ist dem Klassennamen der Modifikator **static** voranzustellen, während andere Modifikatoren verboten sind. Insbesondere dürfen keine Zugriffsmodifikatoren angegeben werden. Diese werden auch nicht benötigt, weil ein statischer Konstruktor ohnehin nur vom Laufzeitsystem aufgerufen wird. Insgesamt erhalten wir das folgende Syntaxdiagramm:



In einer etwas künstlichen Erweiterung des `Bruch`-Beispiels soll der parameterfreie Instanzkonstruktor zufallsabhängige, aber pro Programmeinsatz identische Werte zur Initialisierung der Felder `zaehler` und `nenner` verwenden:

```
public Bruch() {
  Zaehler = zaehlerVoreinst;
  Nenner = nennerVoreinst;
  Anzahl++;
}
```

Dazu erhält die `Bruch`-Klasse private statische Felder, die vom statischen Konstruktor beim Laden der Klasse auf Zufallswerte gesetzt werden sollen:

```
static readonly int zaehlerVoreinst;
static readonly int nennerVoreinst;
```

Der Modifikator **readonly** sorgt dafür, dass die Felder nach der Initialisierung nicht mehr geändert werden können (vgl. Abschnitt 5.2.2). Im statischen Konstruktor wird ein Objekt der Klasse **Random** aus dem Namensraum **System** erzeugt und dann durch **Next()** - Methodenaufrufe mit der Produktion von **int**-Zufallswerten beauftragt:

```
static Bruch() {
  var zuf = new Random();
  zaehlerVoreinst = zuf.Next(1,7);
  nennerVoreinst = zuf.Next(zaehlerVoreinst, 9);
  Console.WriteLine("Klasse Bruch geladen");
}
```

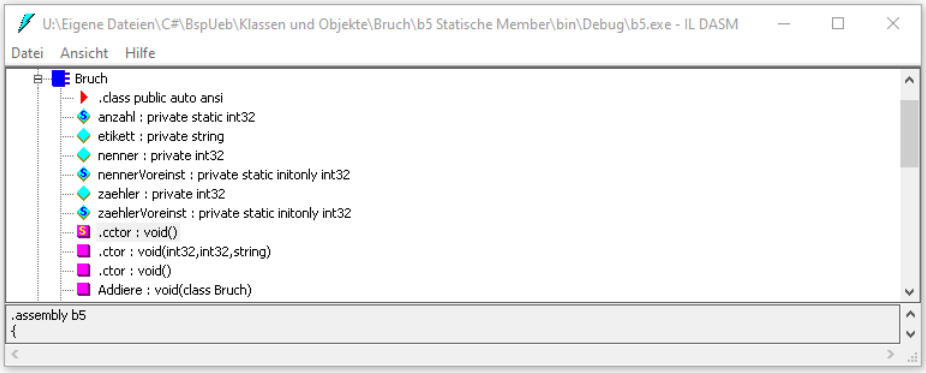
Außerdem protokolliert der statische Konstruktor noch das Laden der Klasse, z. B.:

Quellcode	Ausgabe

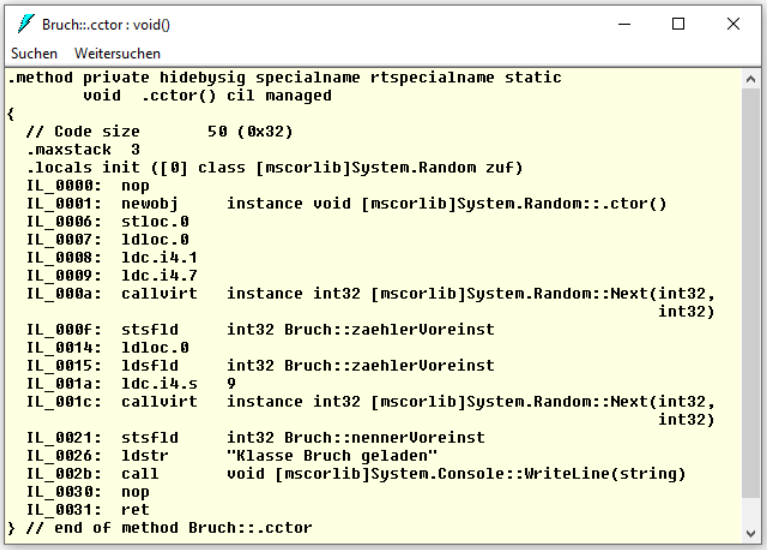
`using System;`
`class Bruchrechnung {`
`  static void Main() {`
`  var b1 = new Bruch();`
`  var b2 = new Bruch();`
`  b1.Zeige(); b2.Zeige();`
`  }`
`}`
	`Klasse Bruch geladen`
	`  1`
	`-----`
	`  5`
	
	`  1`
	`-----`
	`  5`

[1] Genaugenommen führt die vorhandene Initialisierung einer statischen Variablen im Rahmen der Deklarationsanweisung bereits zur automatischen Erstellung eines statischen Konstruktors, der die Initialisierung vornimmt.

Eine Inspektion mit dem Werkzeug **ILDasm** zeigt, dass der statische Konstruktor einer Klasse im Assembly den Namen **.cctor** trägt, z. B.:



Nach einem Doppelklick auf den statischen Konstruktor sieht man seinen IL-Code, z. B.:



5.6.5 Statische Klassen

Besitzt eine Klasse *ausschließlich* statische Member, ist das Erzeugen von Objekten nicht sinnvoll. Man kann es mit dem Modifikator **static** in der Klassendefinition verhindern, z. B.

```
public static class Service {
  . . .
}
```

Außerdem lässt sich eine statische Klasse nicht beerben.

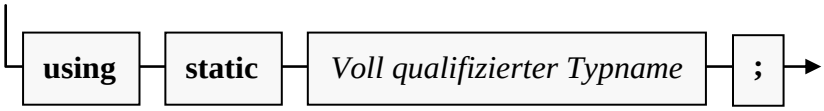
Auch die BCL enthält etliche Klassen, die ausschließlich statische Member enthalten und damit nicht zum Erzeugen von Objekten konzipiert sind. Mit der Klasse **Math** aus dem Namensraum **System** haben wir ein wichtiges Beispiel bereits kennengelernt.

5.6.6 Statische Member eines Typs in eine Quellcode-Datei importieren

Die im Abschnitt 2.6 beschriebene und in praktisch jedem unserer Programme verwendete **using**-Direktive dient dazu, einen Namensraum zu importieren, sodass die dort befindlichen Typen im Programm ohne Namensraumpräfix angesprochen werden können. Seit der C# - Version 6.0 ist

zusätzlich eine **using**-Variante verfügbar, welche die statischen Member eines Typs importiert, sodass im Programm beim Zugriff weder der Namensraum noch die Klasse anzugeben ist:

Import der statischen Member eines Typs



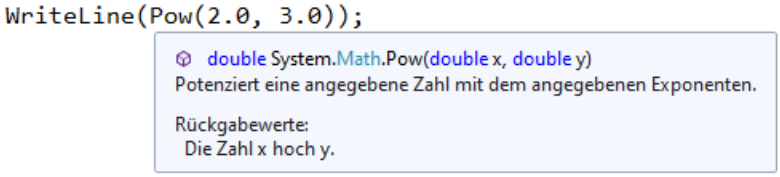
Im folgenden Beispielprogramm aus dem Abschnitt 4.5.2 können aufgrund einer Verwendung der vertrauten **using**-Variante die beiden Klassen **Console** und **Math** ohne Namensraumpräfix angesprochen werden:

```
using System;
class Prog{
  static void Main(){
    Console.WriteLine(Math.Pow(2.0, 3.0));
  }
}
```

Importiert man mit der in C# 6.0 hinzu gekommenen **using**-Variante die statischen Member aus den beiden Klassen, so lassen sie sich wie statische Member der Klasse `Prog` verwenden:

```
using static System.Console;
using static System.Math;
class Prog {
    static void Main() {
        WriteLine(Pow(2.0, 3.0));
    }
}
```

Mit der statischen **using**-Variante lässt sich zwar Schreibarbeit sparen, doch wird gleichzeitig das Verständnis des Quellcodes erschwert, weil verborgen bleibt, welche Klasse bei einer Methode am Werk ist. Das Visual Studio kompensiert den Nachteil teilweise, indem es die Klassenzugehörigkeit anzeigt, sobald der Mauszeiger über einem Methodennamen verharrt:



5.7 Vertiefungen zum Thema Methoden

5.7.1 Rekursive Methoden

Innerhalb einer Methode darf man selbstverständlich nach Belieben *andere* Methoden aufrufen. Es ist aber auch zulässig, dass eine Methode *sich selbst* aufruft. Solche *rekursive* Aufrufe erlauben eine elegante Lösung für ein Problem, das sich sukzessive auf stets einfachere Probleme desselben Typs reduzieren lässt, bis man schließlich zu einem direkt lösbaren Problem gelangt.

Als Beispiel betrachten wir die Ermittlung des größten gemeinsamen Teilers (GGT) zu zwei natürlichen Zahlen, die z. B. in der `Bruch`-Methode `Kuerze()` benötigt wird. Sie haben bereits zwei *iterative* (mit einer Schleife arbeitende) Realisierungen des Euklidischen Lösungsverfahrens kennengelernt: Im Kapitel 1 wurde ein sehr einfacher Algorithmus benutzt, den Sie später in einer Übungsaufgabe (siehe Abschnitt 4.7.4) durch einen effizienteren Algorithmus (unter Verwendung des Modulo-Operators) ersetzt haben. Im aktuellen Abschnitt betrachten wir noch einmal die effizientere Variante, wobei zur Vereinfachung der Darstellung der GGT-Algorithmus vom restlichen

Kürzungsverfahren getrennt und in eine eigene (private) Methode namens `GGTi()` ausgelagert wird:

```
int GGTi(int a, int b) {
   int rest;
   do {
      rest = a % b;
      a = b;
      b = rest;
   } while (rest > 0);
   return a;
}

public void Kuerze() {
   if (zaehler != 0) {
      int teiler = GGTi(Math.Abs(zaehler), Math.Abs(nenner));
      Zaehler /= teiler;
      Nenner /= teiler;
   } else
      Nenner = 1;
}
```

Die mit einer **do-while** - Schleife operierende Methode `GGTi()` kann durch die folgende rekursive Variante `GGTr()` ersetzt werden:

```
int GGTr(int a, int b) {
   int rest = a % b;
   if (rest == 0)
      return b;
   else
      return GGTr(b, rest);
}
```

Statt eine Schleife zu benutzen, arbeitet die rekursive Methode nach folgender Logik:

- Ist der Parameter `a` durch den Parameter `b` restfrei teilbar, dann ist `b` der GGT, und der Algorithmus ist beendet:

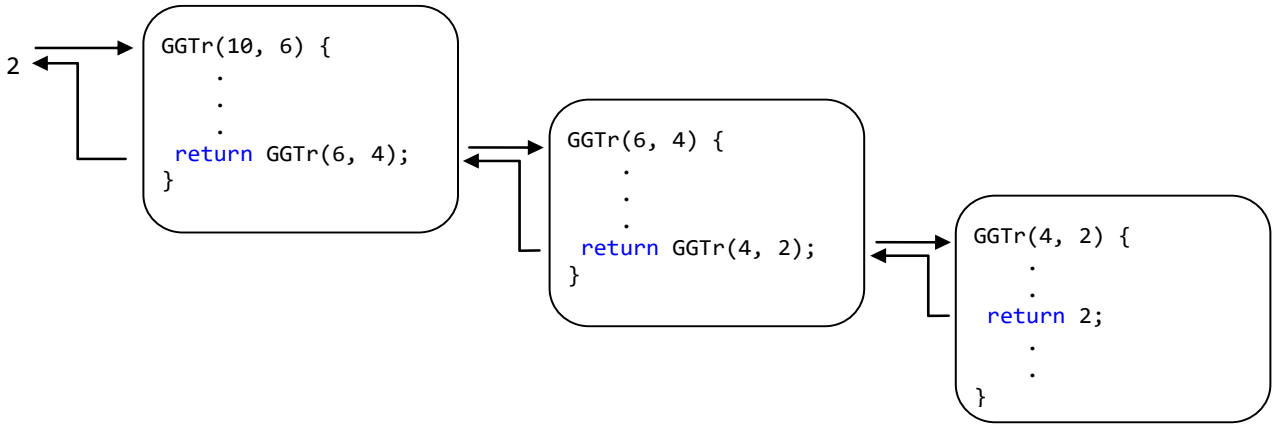
  ```
  return b;
  ```

- Anderenfalls wird das Problem, den GGT von `a` und `b` zu finden, auf das einfachere Problem zurückgeführt, den GGT von `b` und `(a % b)` zu finden, und die Methode `GGTr()` ruft sich selbst mit neuen Aktualparametern auf. Dies geschieht elegant im Ausdruck der **return**-Anweisung:

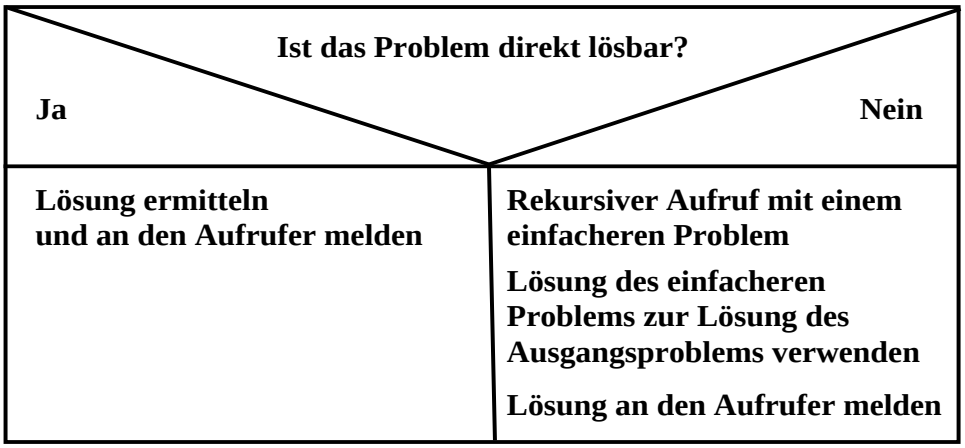
  ```
  return GGTr(b, rest);
  ```

Im iterativen Algorithmus wird übrigens derselbe Trick zur Reduktion des Problems verwendet. Den zugrunde liegenden Satz der mathematischen Zahlentheorie kennen Sie schon aus der oben erwähnten Übungsaufgabe im Abschnitt 4.7.4.

Wird die Methode `GGTr()` z. B. mit den Argumenten 10 und 6 aufgerufen, dann kommt es zur folgenden Aufrufverschachtelung:



Generell läuft eine rekursive Methode mit Lösungsübermittlung per Rückgabewert nach der im folgenden **Struktogramm** beschriebenen Logik ab:



Im Beispiel ist die Lösung des einfacheren Problems identisch mit der Lösung des ursprünglichen Problems.

Wird bei einem fehlerhaften Algorithmus der linke Zweig nie oder zu spät erreicht, dann erschöpfen die geschachtelten Methodenaufrufe die Stack-Kapazität, und es kommt zu einem Ausnahmefehler:

```
Process is terminated due to StackOverflowException.
```

Zu einem rekursiven Algorithmus (per Selbstaufruf einer Methode) existiert stets ein äquivalenter *iterativer* Algorithmus (per Wiederholungsanweisung). Rekursive Algorithmen lassen sich zwar oft eleganter formulieren als die iterativen Alternativen, benötigen aber durch die hohe Zahl von Methodenaufrufen in der Regel mehr Rechenzeit und mehr Speicher.

5.7.2 Operatoren überladen

In C# können nicht nur Methoden überladen werden (siehe Abschnitt 5.3.5), sondern auch die *Operatoren* (**+, *** etc.), was z. B. in der Klasse **String** mit dem „+“ - Operator geschehen ist, sodass wir Zeichenfolgen bequem per „+“ - Operator verketten können. Generell geht es beim Überladen von Operatoren darum, dem Anwender einer Klasse syntaktisch elegante Lösungen für Aufgaben anzubieten, die letztlich einen Methodenaufruf erfordern. Statt die Argumente in einer Aktualparameterliste anzugeben, können sie bei reduziertem Syntaxaufwand um ein Operatorzeichen gruppiert werden. Dieses Zeichen erhält eine neue, zusätzliche Bedeutung für Argumente aus der betroffenen (mit der Operatorüberladung ausgestatteten) Klasse.

Mit den aktuell in der Klasse `Bruch` vorhandenen Methoden lässt sich nur umständlich ein neues Objekt `b3` als Summe von zwei vorhandenen Objekten `b1` und `b2` erzeugen, z. B.:

```
Bruch b3 = new Bruch(b1.Zaehler*b2.Nenner+b1.Nenner*b2.Zaehler, b1.Nenner*b2.Nenner, "");
b3.Kuerze();
```

Es wäre eleganter, wenn derselbe Zweck mit der folgenden Anweisung erreicht werden könnte:

```
Bruch b3 = b1 + b2;
```

Um dies zu ermöglichen, definieren wir eine neue statische `Bruch`-Methode mit dem merkwürdigen Namen **operator+**, die ausgeführt werden soll, wenn das Pluszeichen zwischen zwei `Bruch`-Objekten auftaucht:

```
public static Bruch operator+(Bruch b1, Bruch b2) {
  Bruch temp = new Bruch(b1.zaehler * b2.nenner + b1.nenner * b2.zaehler,
                         b1.nenner * b2.nenner,
                         nameof(b1) + " + " + nameof(b2));
  temp.Kuerze();
  return temp;
}
```

Beim Überladen von Operatoren sind u. a. folgende Regeln zu beachten:

- Es ist grundsätzliche eine *statische* Definition erforderlich.
- Als Namen verwendet man das Schlüsselwort **operator** mit dem jeweiligen Operationszeichen als Suffix.

Nähere Hinweise finden sich z. B. bei Mössenböck (2019, S. 78ff).

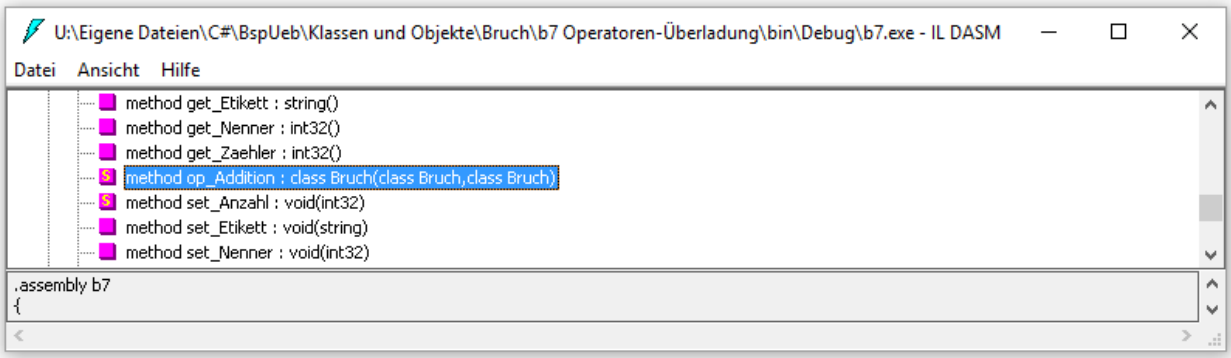
Im Beispiel wird mit Hilfe des **nameof**-Operators dafür gesorgt, dass die Summe automatisch ein passendes Etikett erhält. Der Operator verarbeitet im Beispiel eine Variable als Operanden, und er liefert als Wert des Ausdrucks den Variablennamen.

Mit dem überladenen „+“ - Operator lassen sich `Bruch`-Additionen nun sehr übersichtlich formulieren:

Quellcode	Ausgabe

`using System;`
`class Bruchrechnung {`
`  static void Main() {`
`  var b1 = new Bruch(1, 2, "b1");`
`  b1.Zeige();`
`  var b2 = new Bruch(1, 4, "b2");`
`  b2.Zeige();`
`  var b3 = b1 + b2;`
`  b3.Zeige();`
`  }`
`}`
	`  1`
	`b1 = -----`
	`  2`
	
	`  1`
	`b2 = -----`
	`  4`
	
	`  3`
	`b1 + b2 = -----`
	`  4`

Wie das Hilfsprogramm **ILDasm** zeigt, resultiert aus unserer Operatorenüberladung im Assembly die statische Methode **op_Addition()**:

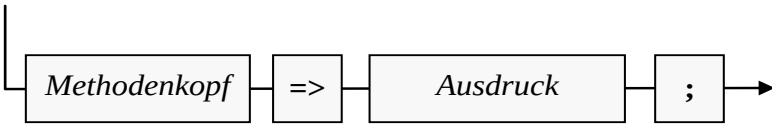


5.7.3 Methoden- und Eigenschaftsdefinition mit Lambda-Operator

Die mit C# 3.0 zur Unterstützung der LINQ-Technik eingeführte *Lambda-Syntax* macht es möglich, eine Funktionalität, die bei bestimmten Gelegenheiten ausgeführt werden soll, auf syntaktisch einfache Weise zu definieren. Die Lambda-Syntax wurde zunächst dazu verwendet, um ein Objekt eines sogenannten *Delegaten* zu realisieren. Mit diesem Datentyp werden wir uns im Kapitel 10 ausführlich beschäftigen. Seit C# 6.0 lässt sich der **Lambda-Operator** (=>) auch zur Methoden- oder Eigenschaftsdefinition verwenden, sofern dabei als Anweisungsblock nur ein einzelner Ausdruck auftritt (engl. Bezeichnung: *expression bodied members*). Obwohl die Leser im aktuellen Kapitel 5 schon mit zahlreichen Details belastet worden sind, hat es auch noch die Methodendefinition mit Lambda-Operator ins Manuskript geschafft, weil die Technik in der von Microsoft im Internet angebotenen Dokumentation zu diversen Themen verwendet wird. Das Verständnis dieser Dokumentation sollte nicht am Lambda-Operator scheitern.

Das folgende Syntaxdiagramm zeigt die Definition einer Methode mit Hilfe des Lambda-Operators:

Methodendefinition per Lambda-Operator



Auf den Lambda-Operator (=>) muss ein Ausdruck folgen mit einem Typ, der zum Rückgabetyp der Methode passt. Die im folgenden Beispiel definierte Methode `Max()` liefert das Maximum von zwei **int**-Argumenten:

```
static int Max(int a, int b) => a >= b ? a : b;
```

Ob diese Notation einen Fortschritt darstellt im Vergleich zur Standardvariante,

```
static int Max(int a, int b) {return a >= b ? a : b;}
```

die nur unwesentlich länger und dabei leichter zu lesen ist, scheint fraglich.

Bei einer Methode mit dem Rückgabetyp **void** muss bei Verwendung des Lambda-Stils auch der Ausdruck diesen Typ haben, z. B.:[1]

```
static void WriteModulo(int a, int b) =>
   Console.WriteLine($"{a} modulo {b} = {a % b}");
```

Das folgende Programm zeigt die beiden eben per Lambda-Operator definierten Methoden in Aktion:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static int Max(int a, int b) => a >= b ? a : b;`
 	
`  static void WriteModulo(int a, int b) =>`
`  Console.WriteLine($"{a} modulo {b} = {a % b}");`
 	
`  static void Main() {`
`  Console.WriteLine(Max(3, 4));`
`  WriteModulo(17, 3);`
`  }`
`}`
	`4`
	`17 modulo 3 = 2`

[1] Weil gemäß Abschnitt 4.5.2 auch ein Methodenaufruf einen Ausdruck darstellt, ist das Beispiel eine korrekte Konkretisierung des Syntaxdiagramms zur Methodendefinition per Lambda-Ausdruck.

Mittlerweile können neben Methoden auch weitere ausführbare Klassenmitglieder per Lambda-Syntax definiert werden:[1]

Mitglied	Möglich ab C# - Version
Methode	6.0
Konstruktor	7.0
Finalisierer	7.0
Get einer Eigenschaft	6.0
Set einer Eigenschaft	7.0
Indexer[2]	7.0

Wir beschränken uns darauf, nach der Methodendefinition auch die Eigenschaftsdefinition per Lambda-Operator vorzuführen. In unserer Klasse `Bruch` kommen bei der Eigenschaft `Zaehler` der **get**- und der **set**-Teil mit einem Ausdruck aus:

```
public int Zaehler {
   get {
      return zaehler;
   }
   set {
      zaehler = value;
   }
}
```

Folglich könnten wir die Eigenschaft `Zaehler` zu einem *Ausdruckskörpermember* (Originalbezeichnung von Microsoft) umgestalten:

```
public int Zaehler {
   get => zaehler;
   set => zaehler = value;
}
```

5.8 Indexer

Bei Arrays und vielen Klassen (z. B. **String**) hat sich der Indexzugriff auf die Elemente eines Objekts per [ ] - Operator als sehr nützlich bis unverzichtbar erwiesen (siehe z. B. Abschnitt 4.7.2.3.2). Um denselben Komfort für eine eigene Klasse (oder Struktur, siehe unten) zu realisieren, die zur Verwaltung von zahlreichen Elementen desselben Typs dient, muss man ihr einen sogenannten **Indexer** spendieren. Analog zur Situation bei einer Eigenschaft handelt es sich auch bei diesem Member letztlich um ein *Paar von Methoden* für den lesenden bzw. schreibenden Zugriff auf ein Element per Indexsyntax.

Ein Indexer kommt also dann in Frage, wenn ein selbst definierter Typ eine Kollektion (z. B. eine Liste) von Elementen verwaltet und ein wahlfreier Zugriff auf die Elemente ermöglicht werden soll.

---

[1] https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/statements-expressions-operators/expression-bodied-members

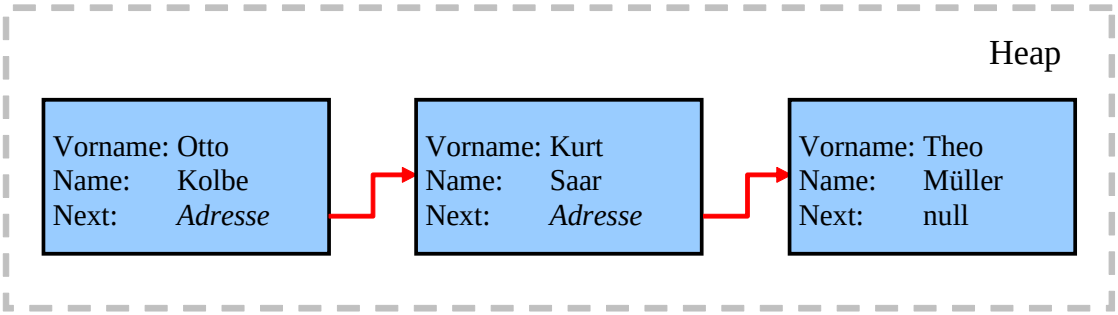
[2] Indexer werden im Abschnitt 5.8 behandelt.

5.8.1 Definition am Beispiel einer Klasse zur Verwaltung einer verketteten Liste

Als Beispiel betrachten wir eine „Datenbank“, die mit einer sogenannten *verketteten Liste* von Objekten der folgenden Klasse `Person` arbeitet:

```
class Person {
  public string Vorname;
  public string Name;
  public Person Next;
  public Person(string vorname, string name) {
    Vorname = vorname;
    Name = name;
  }
}
```

Wir verzichten der Kürze halber bei der Klasse `Person` auf die hier durchaus empfehlenswerte Datenkapselung. Jedes `Person`-Objekt besitzt eine Instanzvariable `Next` zur Aufnahme der Adresse seines Nachfolgers. Wenn man beim Erzeugen eines neuen Objekts dessen Adresse in das `Next`-Feld des bisher letzten Objekts schreibt, entsteht eine (einfach) verkettete Liste. Durch das beharrliche Verfolgen der `Next`-Referenzen lässt sich jedes Serienelement erreichen, sofern eine Referenz auf das *erste* Element verfügbar ist. In der folgenden Abbildung ist eine Kette aus drei `Person`-Objekten zu sehen:



Ein Objekt der folgenden Klasse `PersonDB` verwaltet eine verkettete Liste von `Person`-Objekten:

```
class PersonDB {
  int n;
  Person first, last;

  public int Count {
    get {
      return n;
    }
  }

  public void Add(Person neu) {
    if (neu == null)
      return;
    if (n == 0)
      first = last = neu;
    else {
      last.Next = neu;
      last = neu;
    }
    n++;
  }
```

```
    public Person this[int i] {
      get {
        if (i >= 0 && i < n) {
          Person sel = first;
          for (int j = 0; j < i; j++)
              sel = sel.Next;
          return sel;
        } else
          return null;
      }
      set {
        if (i >= 0 && i < n && value != null) {
          if (i == 0) {
            value.Next = first.Next; // Next-Wert des Neulings zeigt auf den Nachfolger
            first = value; // Der Neuling wird zum Startelement
          } else {
            Person pre = first;
            for (int j = 0; j < i - 1; j++)
              pre = pre.Next;
            value.Next = pre.Next.Next; // Next-Wert des Neulings zeigt auf den Nachfolger
            pre.Next = value; // Next-Wert des Vorgängers zeigt auf den Neuling
          }
        }
      }
    }
}
```

Für den lesenden oder schreibenden Zugriff auf das *i*-te Listenelement stellt `PersonDB` einen Indexer mit dem Parameterdatentyp **int** zur Verfügung, der eine `Person`-Referenz liefert (**get**) oder das *i*-te Listenelement durch eine andere `Person`-Referenz ersetzt (**set**).

Einige Regeln für die Indexer-Definition:

- Nach den optionalen Modifikatoren wird der Datentyp angegeben (im Beispiel: `Person`).
- Der Name lautet stets **this**.
- Hinter dem Schlüsselwort **this** wird *eckig* eingeklammert der Indexparameter mit Datentyp und Namen deklariert.
- Der **set**-Methode wird wie bei Eigenschaften ein impliziter Parameter namens **value** übergeben.

Vom nicht ganz trivialen `PersonDB`-Aufbau merkt ein Anwender dieser Klasse nichts, z. B.:

```
using System;
class PersonDbDemo {
  static void Main() {
    var pdb = new PersonDB();
    pdb.Add(new Person("Otto", "Kolbe"));
    pdb.Add(new Person("Kurt", "Saar"));
    pdb.Add(new Person("Theo", "Müller"));
    for (int i = 0; i < pdb.Count; i++)
      Console.WriteLine($"Nummer {i}: {pdb[i].Vorname} {pdb[i].Name}");
    Console.WriteLine();
    pdb[1] = new Person("Ilse", "Golter");
    for (int i = 0; i < pdb.Count; i++)
      Console.WriteLine($"Nummer {i}: {pdb[i].Vorname} {pdb[i].Name}");
  }
}
```

Das Programm liefert die folgende Ausgabe:

```
Nummer 0: Otto Kolbe
Nummer 1: Kurt Saar
Nummer 2: Theo Müller

Nummer 0: Otto Kolbe
Nummer 1: Ilse Golter
Nummer 2: Theo Müller
```

In den **WriteLine()** - Aufrufen des letzten Beispielprogramms wird übrigens die im Abschnitt 4.2.2.2 beschriebene Zeichenfolgeninterpolation verwendet.

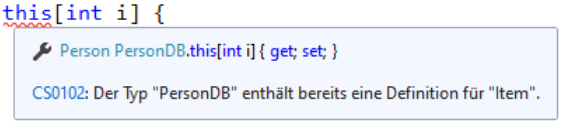
Statt eine eigene Klasse `PersonDB` mit Listenkonstruktion und Indexer zu entwerfen, wird man in der Praxis eine solche Aufgabe weit ökonomischer unter Verwendung einer generischen Kollektionsklasse aus dem Namensraum **System.Collections.Generic** realisieren (siehe Abschnitt 11.3.2). Allerdings gehört der Eigenbau einer verketteten Liste zu einer soliden Programmierer-Grundausbildung, sodass sich der Aufwand des `PersonDB`-Beispiels wohl doch lohnt.

Die Definition eines Indexers hat zur Folge, dass automatisch eine Eigenschaft mit dem Namen **Item** entsteht.[1] Auf diese Eigenschaft ist im Quellcode kein Zugriff möglich, und man könnte sie komplett ignorieren, wenn nicht die Definition eines Indexers scheitern würde, sobald man selbst eine Eigenschaft mit dem Namen `Item` definiert. Würde z.B. die Klasse `PersonDB` eine explizit definierte Eigenschaft namens `Item` enthalten,

```
class PersonDB {
  int n;
  Person first, last;
  int Item { get; set; }
 . . .
}
```

dann könnte in `PersonDB` kein Indexer definiert werden:

```
public Person this[int i] {
```



Wenn die explizit definierte Eigenschaft namens `Item` unbedingt benötigt wird, dann kann man den Indexer durch ein angeheftetes **IndexerNameAttribute** umbenennen.[2]

5.8.2 Indexer überladen

Wenn die Elemente einer Kollektion alternativ durch *mehrere* Variablen angesprochen bzw. identifiziert werden können (z. B. Landkreise durch eine laufende Nummer und einen Namen), dann kommt das von C# ermöglichte Überladen des Indexers durch die Verwendung verschiedener Parametertypen gelegen. Im Beispiel könnte man eine Indexer-Überladung mit **string**-Parameter ergänzen, welche die (nach Listenposition) erste Person liefert, deren Vorname mit dem Aktualparameter übereinstimmt:

---

[1] https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/indexers/using-indexers

[2] Mit Attributen werden wir uns im Kapitel 14 beschäftigen.

```
public Person this[string vn] {
   get {
        for (int j = 0; j < n; j++)
            if (this[j].Vorname == vn)
                return this[j];
      return null;
   }
}
```

Hier ist ein Einsatz der Indexer-Überladung mit dem Vornamensparameter zu sehen:

```
String s = "Ilse";
Console.WriteLine($"\nName der ersten Person mit dem Vornamen \"{s}\": {pdb[s].Name}");
```

5.8.3 Mehrdimensionale Indexer

Um auch die mehrdimensionalen Indexer behandeln zu können, wird die Serie der Vorgriffe auf den Abschnitt 6.2 über Arrays noch etwas erweitert, indem auch zweidimensionale Arrays verwendet werden. Im weiteren Verlauf des Kurses werden mehrdimensionale Indexer allerdings nicht erneut auftauchen, sodass der aktuelle Abschnitt beim ersten Lesen übersprungen werden kann.

Wenn die Elemente einer Kollektion über eine Kombination von Variablenausprägungen identifiziert werden, dann kommt ein mehrdimensionaler Indexer in Frage. Im folgenden Beispiel wird die Definition einer Klasse namens `FloatMatrix` zur Verwaltung einer zweidimensionalen Matrix mit **float**-Elementen angedeutet. Zur Datenablage kommt ein zweidimensionaler **float**-Array zum Einsatz. Diese elementare, von der Programmiersprache C# angebotene Kollektion wird von der Klasse `FloatMatrix` um zusätzliche Verhaltenskompetenzen erweitert. So kann ein Objekt der Klasse `FloatMatrix` z. B. seine Elemente transponieren:

```
using System;
public class FloatMatrix {
   float[,] Data;

    public FloatMatrix(int z_, int s_) {
        Data = new float[z_, s_];
        Zeilen = z_;
        Spalten = s_;
   }

    public int Zeilen { get; private set; }
    public int Spalten { get; private set; }

    public float this[int z, int s] {
        get { return Data[z, s]; }
        set { Data[z, s] = value; }
    }

    public void Anzeigen() {
       . . .
   }

   public void Transponieren() {
       . . .
   }
}
```

Mit Hilfe des mehrdimensionalen Indexers kann die von zweidimensionalen C# - Arrays gewohnte Syntax beim Zellzugriff auch bei der Matrix mit zusätzlichen Verhaltenskompetenzen verwendet werden:

```
using System;
class FloatMatrixDemo {
   static void Main() {
      const int zeilen = 3, spalten = 5;
      var matrix = new FloatMatrix(zeilen, spalten);
      for (int i = 0; i < zeilen; i++)
         for (int j = 0; j < spalten; j++)
            matrix[i,j] = i * spalten + j + 1;
      matrix.Transponieren();
      Console.WriteLine(matrix[2,0]);
   }
}
```

Der zweidimensionale Indexer der Klasse `FloatMatrix` verwendet natürlich im Wesentlichen die Zugriffstechnik des zweidimensionalen C# - Arrays.

5.9 Komposition

Bei den Feldern einer Klasse sind beliebige Datentypen zugelassen, auch Referenztypen. Z. B. ist in der aktuellen `Bruch`-Definition eine Instanzvariable vom Referenztyp **String** vorhanden. Es ist also möglich, vorhandene Klassen als Bestandteile von neuen, komplexeren Klassen zu verwenden. Neben der Vererbung und der Polymorphie ist diese *Komposition* eine weitere Technik zur Wiederverwendung von vorhandenen Typen bei der Definition von neuen Typen. Außerdem ist sie im Sinne einer realitätsnahen Modellierung unverzichtbar, denn auch ein reales Objekt (z. B. eine Firma) enthält andere Objekte[1] (z. B. Mitarbeiter, Kunden), die ihrerseits wiederum Objekte enthalten (z. B. ein Gehaltskonto und einen Terminkalender bei den Mitarbeitern) usw.

Man kann den Standpunkt einnehmen, dass die Komposition eine selbstverständliche, wenig spektakuläre Angelegenheit sei, eigentlich nur ein neuer Begriff für eine längst vertraute Situation (Instanzvariablen mit Referenztyp). Es ist tatsächlich für den weiteren Lernerfolg im Kurs unkritisch, wenn Sie den Rest des aktuellen Abschnitts mit dem recht länglichen Beispiel zur Komposition überspringen.

Wir erweitern das Bruchrechnungsprogramm um eine Klasse namens `Aufgabe`, die Trainingssitzungen unterstützen soll und dazu mehrere `Bruch`-Objekte verwendet. In der `Aufgabe`-Klassendefinition tauchen vier Instanzvariablen vom Typ `Bruch` auf:

```
using System;

public class Aufgabe {
   Bruch b1, b2, lsg, antwort;
   char op;

   public Aufgabe(char op_, int b1Z, int b1N, int b2Z, int b2N) {
      op = op_;
      b1 = new Bruch(b1Z, b1N, "1. Argument:");
      b2 = new Bruch(b2Z, b2N, "2. Argument:");
      lsg = new Bruch(b1Z, b1N, "Resultat");
      antwort = new Bruch();
      Init();
   }
```

[1] Die betroffenen Personen mögen den Fachterminus *Objekt* nicht persönlich nehmen.

```
    private void Init() {
       switch (op) {
          case '+':  lsg.Addiere(b2);
                     break;
          case '*':  lsg.Multipliziere(b2);
                     break;
       }
    }

    public bool Korrekt {
       get {
          Bruch temp = antwort.Klone();
          temp.Kuerze();
          if (lsg.zaehler == temp.zaehler && lsg.nenner == temp.nenner)
             return true;
          else
             return false;
       }
    }

    public void Zeige(int was) {
       switch (was) {
          case 1:  Console.WriteLine("   " + b1.zaehler +
                                     "                " + b2.zaehler);
                   Console.WriteLine(" ------     " + op + "     -----");
                   Console.WriteLine("   " + b1.nenner +
                                     "                " + b2.nenner);
                   break;
          case 2: lsg.Zeige(); break;
          case 3: antwort.Zeige(); break;
       }
    }

    public void Frage() {
       Console.WriteLine("\nBerechne bitte:\n");
       Zeige(1);
       Console.Write("\nWelchen Zähler hat Dein Ergebnis:      ");
       antwort.Zaehler = Convert.ToInt32(Console.ReadLine());
       Console.WriteLine("                                    ------");
       Console.Write("Welchen Nenner hat Dein Ergebnis:      ");
       antwort.Nenner = Convert.ToInt32(Console.ReadLine());
    }

    public void Pruefe() {
       Frage();
       if (Korrekt)
          Console.WriteLine("\n Gut!");
       else {
          Console.WriteLine("\n Leider falsch!\n");
          Zeige(2);
       }
    }

    public void NeueWerte(char op_, int b1Z, int b1N, int b2Z, int b2N) {
       op = op_;
       b1.Zaehler = b1Z; b1.Nenner = b1N;
       b2.Zaehler = b2Z; b2.Nenner = b2N;
       lsg.Zaehler = b1Z; lsg.Nenner = b1N;
       Init();
    }
}
```

Die vier `Bruch`-Objekte in einer `Aufgabe` dienen den folgenden Zwecken:

- `b1` und `b2` werden dem Anwender (in der `Aufgabe`-Methode `Frage()`) im Rahmen einer Aufgabenstellung vorgelegt, z. B. zum Addieren.
- In `antwort` landet der Lösungsversuch des Anwenders.
- In `lsg` steht das korrekte Ergebnis.

In der Klasse `Bruch` wird die Instanzmethode `Multipliziere()` nachgerüstet, die analog zur Methode `Addiere()` arbeitet:

```
public void Multipliziere(Bruch b) {
   Zaehler = zaehler * b.zaehler;
   Nenner = nenner * b.nenner;
   Kuerze();
}
```

Im folgenden Programm wird die Klasse `Aufgabe` für ein Bruchrechnungstraining verwendet:

```
using System;
class Bruchrechnung {
   static void Main() {
      Aufgabe auf = new Aufgabe('*', 3, 4, 2, 3);
      auf.Pruefe();
      auf.NeueWerte('+', 1, 2, 2, 5);
      auf.Pruefe();
   }
}
```

Man kann immerhin schon ahnen, wie die praxistaugliche Endversion des Programms einmal arbeiten wird:

```
Berechne bitte:

   3               2
 ------     *     -----
   4               3

Welchen Zaehler hat Dein Ergebnis:     6
                                      ------
Welchen Nenner hat Dein Ergebnis:      12

 Gut!


Berechne bitte:

   1               2
 ------     +     -----
   2               5

Welchen Zaehler hat Dein Ergebnis:     3
                                      ------
Welchen Nenner hat Dein Ergebnis:      7

 Leider falsch!
              9
 Resultat = -----
              10
```

5.10 Innere (eingeschachtelte) Klassen

Eine Klasse darf neben Feldern, Methoden etc. auch Klassendefinitionen enthalten, wobei *innere (eingeschachtelte) Klassen* entstehen. Eine Klasse innerhalb einer umgebenden Klasse zu definieren, ist z. B. dann sinnvoll, wenn die innere Klasse nicht allgemein benötigt wird, sondern nur zur Modellierung von speziellem Zubehör der umgebenden Klasse dient. Bei der voreingestellten Schutzstufe **private** ist die innere Klasse im restlichen Programm nicht sichtbar, kann also dort nicht verwendet werden und auch keine Namenskollision verursachen.

Im folgenden Beispiel werden innerhalb der Klasse `Familie` die Klassen `Tochter` und `Sohn` definiert:

```
using System;

public class Familie {
  string name;
  Tochter t;
  Sohn s;

  public Familie(string name_, string nato, int alto, string naso, int also) {
    name = name_;
    t = new Tochter(this, nato, alto);
    s = new Sohn(this, naso, also);
  }

  public void Informiere() {
    Console.WriteLine("Die Kinder von Familie {0}:\n", name);
    t.Informiere();
    s.Informiere();
  }

  class Tochter {
    Familie f;
    string name;
    int alter;

    public Tochter(Familie f_, string name_, int alt_) {
      f = f_;
      name = name_;
      alter = alt_;
    }

    public void Informiere() {
      Console.WriteLine("  Ich bin die {0}-jährige Tochter {1} von Familie {2}",
                        alter, name, f.name);
    }
  }

  public class Sohn {
    Familie f;
    string name;
    int alter;

    public Sohn(Familie f_, string name_, int alt_) {
      f = f_;
      name = name_;
      alter = alt_;
    }

    public void Informiere() {
      Console.WriteLine("  Ich bin der {0}-jährige Sohn {1} von Familie {2}",
                        alter, name, f.name);
    }
  }
}
```

Die **Main()** - Methode der Testklasse

```
class FamilienTest {
   static void Main() {
      var f = new Familie("Müller", "Lea", 7, "Leo", 4);
      f.Informiere();
   }
}
```

liefert folgende Ausgabe:

```
Die Kinder von Familie Müller:

  Ich bin die 7-jährige Tochter Lea von Familie Müller.
  Ich bin der 4-jährige Sohn Leo von Familie Müller.
```

Für das Schachteln von Klassendefinitionen gelten u. a. folgende Regeln:

- Die Methoden der inneren Klasse dürfen auf die als **private** oder **protected** deklarierten Member der umgebenden Klasse zugreifen, d.h.:[1]
  - auf die Member eines Objekts der umgebenden Klasse, sofern eine Referenz vorhanden ist, die meist per Konstruktor bekannt gemacht wird (wie in den inneren Klassen `Tochter` und `Sohn`)
  - auf die statischen Member der umgebenden Klasse

  In der folgenden Anweisung aus der `Tochter`-Methode `Informiere()` erfolgt ein Zugriff auf die private Instanzvariable `name` aus der Klasse `Familie`):

  ```
  Console.WriteLine("  Ich bin die {0}-jährige Tochter {1} von Familie {2}",
                    alter, name, f.name);
  ```

- Umgekehrt hat die umgebende Klasse *keine* Zugriffsrechte für die privaten Member einer inneren Klasse, weshalb im Beispiel die Konstruktoren und die `Informiere()` - Methoden der inneren Klassen als **public** definiert wurden.
- Innere Klassen besitzen wie die sonstigen Klassen-Member (Felder, Methoden, Eigenschaften etc.) die voreingestellte Schutzstufe **private**.
- Erhält eine innere Klasse die Schutzstufe **public**, dann können in beliebigen Klassen Objekte von diesem Typ erstellt werden, wobei dem Namen der inneren Klasse der Name ihrer Hüllenklasse voranzustellen ist, z. B.:

  ```
  Familie.Sohn faso = new Familie.Sohn(f, "Leo", 4);
  ```

5.11 Verfügbarkeit von Klassen und Klassenmitgliedern

Nachdem die Datenkapselung mehrfach als wesentlicher Vorzug bzw. als Kernidee der objektorientierten Programmierung herausgestellt wurde und wiederholt Angaben zur Verfügbarkeit von Klassen bzw. Klassenbestandteilen an verschiedenen Stellen eines .NET - Programms gemacht wurden, sollen die Regeln zum Zugriffsschutz nun zusammengestellt werden, obwohl dabei noch einige Vorgriffe auf das Thema *Vererbung* nötig sind.

Bei einer äußeren (nicht eingeschachtelten) Klasse kennt C# die folgenden Stufen der Verfügbarkeit (vgl. ECMA 2017, S. 276):

[1] Zum Zugriffsmodifikator **protected** siehe Abschnitt 5.11.

Modifikator	Die Klasse ist verfügbar …	
	im eigenen Assembly	überall
ohne oder **internal**	ja	nein
public	ja	ja

Unsere als **public** definierte Beispielklasse `Bruch` kann in beliebigen .NET-Programmen mit Zugang zur Assembly-Datei genutzt werden, was gleich im Abschnitt 5.12 demonstriert werden soll.

Für Klassen-Member (Felder, Methoden, Eigenschaften, innere Klassen etc.) unterstützt C# die folgenden Schutzstufen:[1]

Modifikator(en)	Der Zugriff ist erlaubt für ...				
	eigene Klasse (Abk. K) u. innere Klassen	nicht von K abgel. Klassen im eigenen Assembly	von K abgeleitete Klassen		sonstige Klassen
			im eigenen Assembly	in anderen Assemblies	
ohne oder **private**	ja	nein	nein	nein	nein
internal	ja	ja	ja	nein	nein
protected	ja	nein	geerbte Member	geerbte Member	nein
protected internal	ja	ja	ja	geerbte Member	nein
private protected	ja	nein	geerbte Member	nein	nein
public	ja	ja	ja	ja	ja

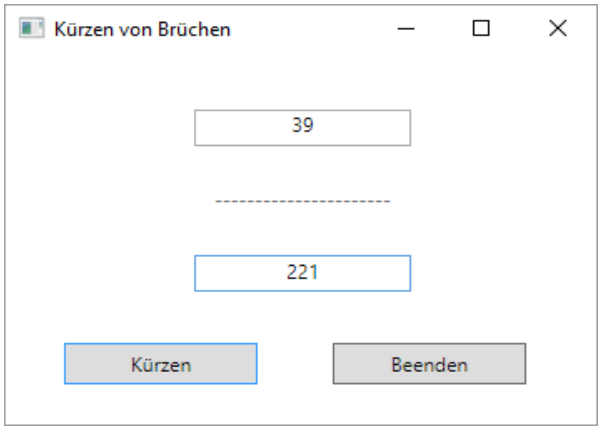
Die Zugriffsmodifikatorenkombination **private protected** ist seit C# 7.2 verfügbar.

Wir haben die Methoden und Eigenschaften unserer Beispielklasse `Bruch` als **public** definiert und bei den Feldern die voreingestellte Schutzstufe **private** beibehalten.

Die beschriebenen Zugriffsregeln für Klassen und Member gelten analog auch bei später vorzustellenden Typen (Strukturen, Enumerationen, Schnittstellen und Delegaten).

5.12 Bruchrechnungsprogramm mit WPF-Bedienoberfläche

Nachdem Sie nun wesentliche Teile der objektorientierten Programmierung mit C# kennengelernt haben, ist vielleicht ein weiterer Ausblick auf die nicht mehr allzu ferne Entwicklung von Windows-Programmen mit grafischer Bedienoberfläche als Belohnung und Motivationsquelle angemessen. Schließlich gilt es in diesem Kurs auch die Erfahrung zu vermitteln, dass man beim Programmieren Erfolg und damit Spaß haben kann. Wir erstellen nun das schon im Abschnitt 1.6 präsentierte Bruchkürzungsprogramm mit grafischer Bedienoberfläche:



[1] https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/access-modifiers
https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers

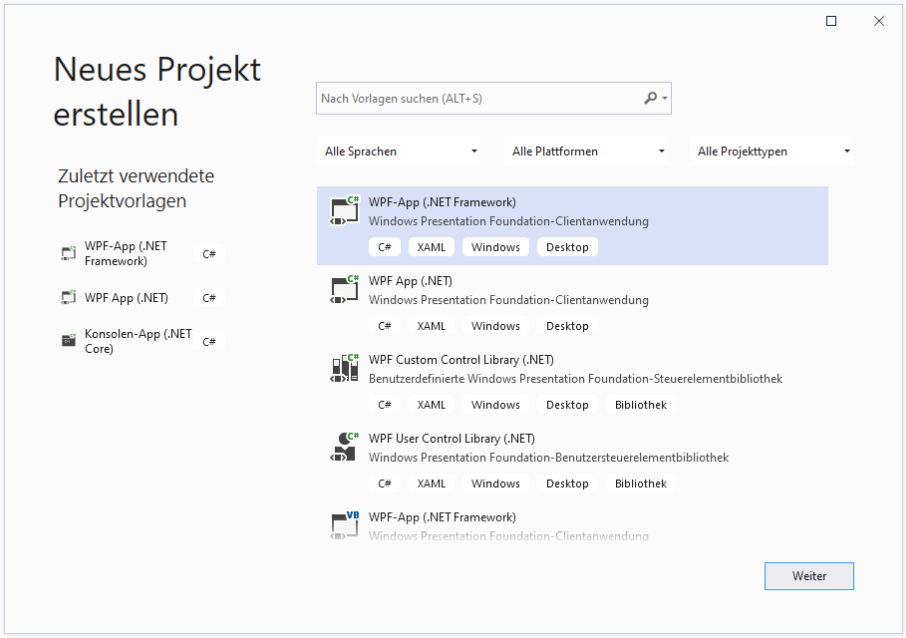
Bei dieser Gelegenheit werden wir unser Wissen über die Erstellung einer WPF-Anwendung (*Windows Presentation Foundation*) erweitern, um bei der „offiziellen“ Behandlung des Themas im Kapitel 12 schon einige Erfahrungen einbringen zu können. Wir werden im Kurs die WPF - GUI-Technik aus später noch im Detail zu diskutierenden Gründen gegenüber den Alternativen **WinForms** (veraltet) und **UWP** (*Universal Windows Platform*, fragliche Zukunftsaussichten) bevorzugen.

5.12.1 Projekt anlegen

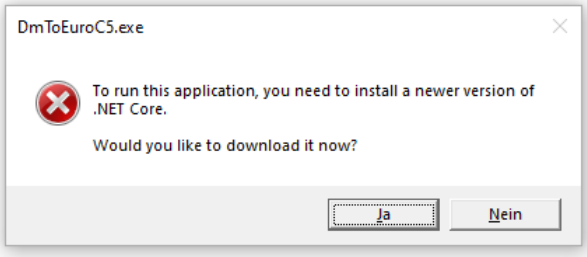
Verwenden Sie im Visual Studio nach

Datei > Neu > Projekt

für ein neues Projekt mit dem Namen `BruchKürzenGui` die Vorlage **WPF-App (.NET Framework)**:



Wählt man die Projektvorlage **WPF App (.NET)**, kann man zwar C# 9.0 benutzen, doch läuft das resultierende Programm auf einem Rechner mit Windows 10 nicht ohne Zusatzinstallation:



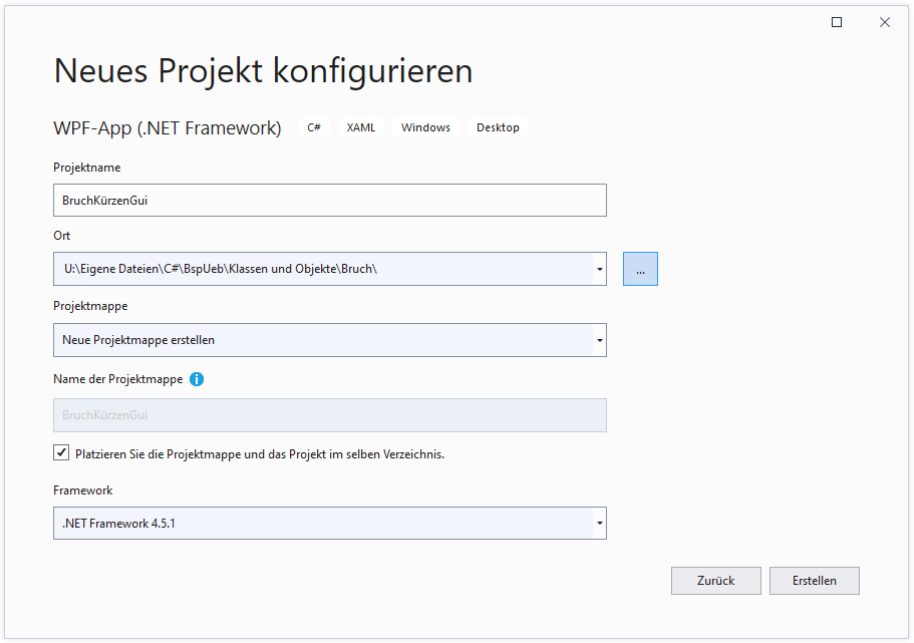
Um das Programm nutzen zu können, muss ein Kunde (z. B. über den Schalter **Ja** im Fehlermeldungsfenster) die **.NET Desktop Runtime 5** (z. B. in der Datei **windowsdesktop-runtime-5.0.1-win-x64.exe**) von der folgenden Webseite

https://dotnet.microsoft.com/download/dotnet/5.0

herunterladen und installieren (vgl. Abschnitt 2.1.1.2).

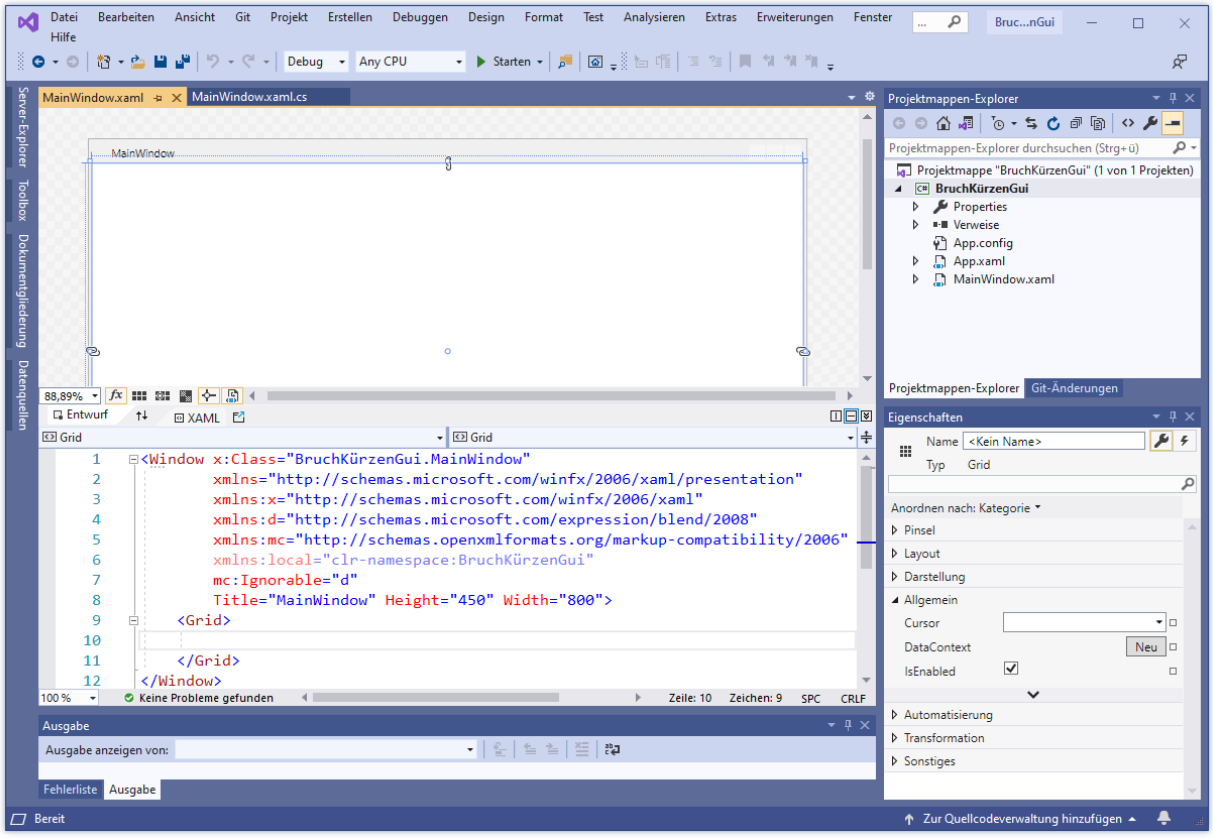
Da unser geplantes Programm mit C# 7.3 auskommt, wählen wir die Projektvorlage **WPF-App (.NET Framework)** und erhalten die maximale Kompatibilität, indem wir im nächsten Dialog des

Assistenten für neue Projekte das in Windows 8.1 und Windows 10 ohne Zusatzinstallation unterstützte **.NET Framework 4.5.1** wählen:



Weil wir ein Assembly mit der Klasse `Bruch` einbinden wollen, darf das im dortigen Projekt eingestellte Kompatibilitätsniveau nicht unterschritten werden.

Nach einem Mausklick auf **Erstellen** präsentiert die Entwicklungsumgebung im **WPF-Designer** einen Rohling für das Fenster der entstehenden Anwendung:



In der oberen Designer-Zone können wir die Bedienoberfläche unseres Programms mit Hilfe von grafischen Werkzeugen erstellen und dazu konfigurierbare Komponenten (Steuerelemente) aus der

Toolbox (siehe Abschnitt 5.12.3) übernehmen. Wie gleich zu sehen sein wird, definieren wir dabei die neue Klasse `MainWindow` im Namensraum `BruchKürzenGui`.

5.12.2 Deklaration der Bedienoberfläche per XAML

Ein zentrales Merkmal der WPF-Technologie besteht darin, das GUI-Design einer Anwendung durch eine XML-Spezialisierung (*eXtensible Markup Language*) namens XAML (*eXtensible Application Markup Language*) zu deklarieren. Zu unserem Anwendungsfenster gehört eine XAML-Datei, die im unteren Teil der Designer-Zone erscheint und initial so aussieht:

```
<Window x:Class="BruchKürzenGui.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:BruchKürzenGui"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>

    </Grid>
</Window>
```

Das **Window**-Element, das mit dem Start-Tag

```
<Window  . . . >
```

startete und mit dem End-Tag

```
</Window>
```

endet, definiert ein Fenster, also das Erscheinungsbild eines Objekts aus einer anwendungseigenen Klasse, die von der BCL-Klasse **Window** abstammt. Im Beispiel enthält das **Window**-Element initial im Start-Tag etliche Attribute:

- Die **xmlns**-Attribute deklarieren die XML-Namensräume, aus denen die verwendeten XAML-Bestandteile stammen. Von den 5 Namensräumen erhalten 4 ein Präfix, das den aus einem Namensraum stammenden XAML-Bestandteilen vorangestellt werden muss (z. B. **x:Class**).
- Das Attribut **x:Class** nennt die zum **Window**-Element gehörende Klasse samt .NET - Namensraum, in unserem Fall also `BruchKürzenGui.MainWindow`.
- Das Attribut **mc:Ignorable** mit dem Wert "**d**" ist nur bei Verwendung des GUI-Designers *Blend for Visual Studio* relevant.[1]
- Durch das Attribut **Title** wird die Fensterbeschriftung festgelegt.
- Die Attribute **Height** und **Width** sind für die initiale Fenstergröße zuständig.

Außerdem enthält das **Window**-Element initial ein **Grid**-Kindelement, das als Container für Steuerelemente dient und später noch ausführlich besprochen wird.

Der grafische Fenster-Designer ist lediglich ein (sehr praktisches!) Werkzeug zur Bearbeitung der XAML-Datei.

---

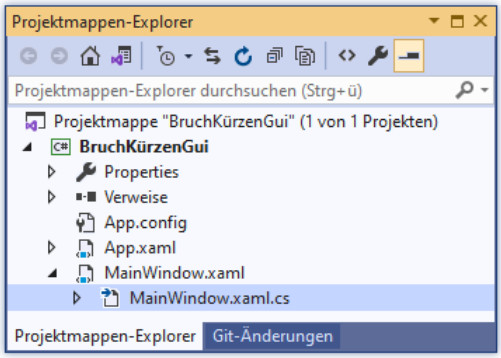
[1] Es sorgt dafür, dass alle Elemente und Attribute mit dem Namensraumpräfix **d** zur Laufzeit ignoriert werden. Um diese Bestandteile kümmert sich das für Animationen und andere aufwändige GUI-Techniken zuständige Programm *Blend for Visual Studio* (frühere Bezeichnung: *Expression Blend*). Dieses Programm ist primär für Designer gedacht, die zusammen mit Entwicklern an großen Projekten arbeiten. Dasselbe Projekt kann alternativ von einem Entwickler mit dem Visual Studio und von einem Designer mit Blend for Visual Studio geöffnet werden. Blend for Visual Studio kann zusammen mit dem Visual Studio 2019 installiert werden. Wir haben auf die Installation verzichtet (siehe Abschnitt 3.1.3) und werden Blend im Kurs nicht verwenden.

Die XAML-Datei zu unserem Anwendungsfenster heißt **MainWindow.xaml** und beschreibt die Oberfläche von Objekten der Klasse `MainWindow`. Zu dieser Klasse gehören auch zwei Quellcode-Dateien mit den Deklarationen bzw. Definitionen der üblichen Klassen-Member (Felder, Methoden, Eigenschaften etc.):

- **MainWindow.xaml.cs**
  Hier werden unsere Beiträge zur Funktionalität der Klasse `MainWindow` landen.
- **MainWindow.g.cs**
  Diese Datei wird automatisch durch das Übersetzen der XAML - Deklarationen in C# - Quellcode generiert und vor unseren Blicken relativ gut verborgen, weil direkte Änderungen durch Programmierer *nicht* vorgesehen sind.

Weitere Details zu den beiden C# - Dateien folgen im Abschnitt 5.12.6.

Der Projektmappen-Explorer zeigt zum Anwendungsfenster die XAML-Datei und die für Programmierer relevante C# - Datei:



Über die Fensterklasse hinaus benötigt eine WPF-Anwendung auch noch eine **Anwendungsklasse**, die von der BCL-Klasse **Application** abstammt. Erneut sind eine XAML-Deklarationsdatei und zwei C# - Quellcodedateien beteiligt (siehe Abschnitt 5.12.6 für Details). Bei unserem geplanten Beispielprogramm müssen wir uns um diese Dateien *nicht* kümmern. Wer die XAML-Datei **App.xaml** neugierig per Doppelklick auf ihren Eintrag im Projektmappen-Explorer öffnet, kann z. B. im **Application**-Element feststellen, dass im Attribut **StartupUri** das beim Programmstart anzuzeigende Fenster (über seine XAML-Datei) festgelegt wird:

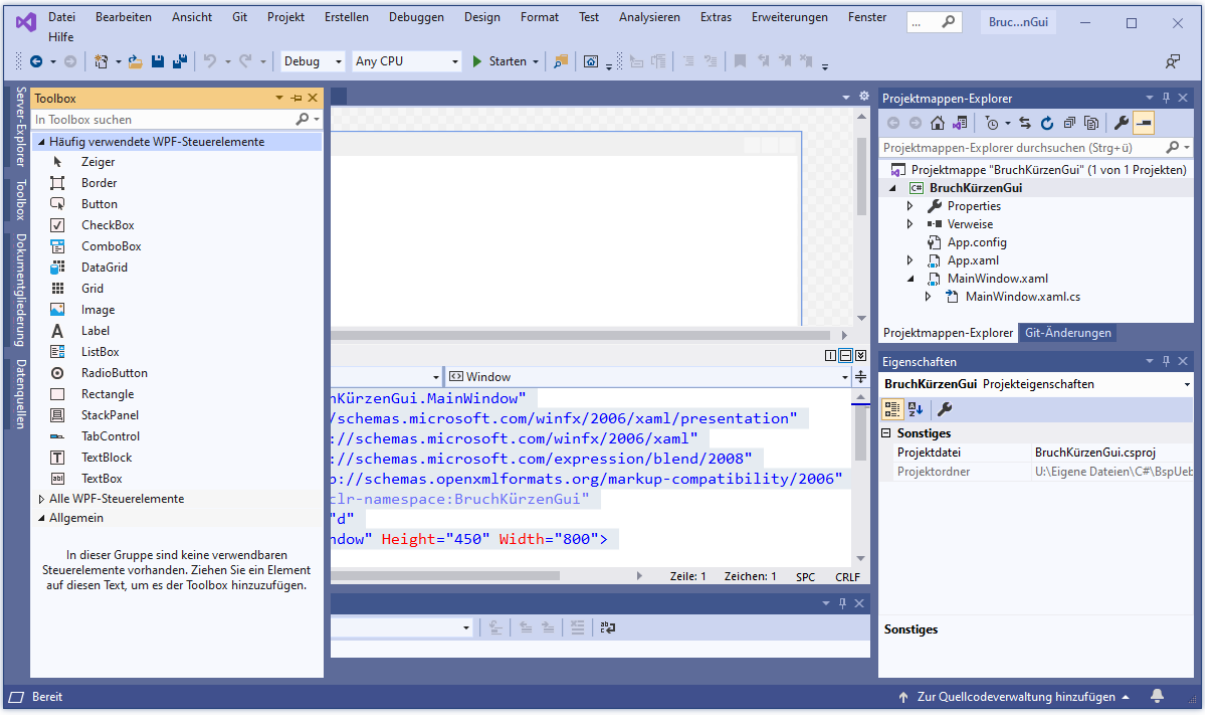
```
<Application x:Class="BruchKürzenGui.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:BruchKürzenGui"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

5.12.3 Steuerelemente aus der Toolbox übernehmen

Öffnen Sie das **Toolbox**-Fenster mit der Tastenkombination **Strg-Alt-X**, mit dem Menübefehl

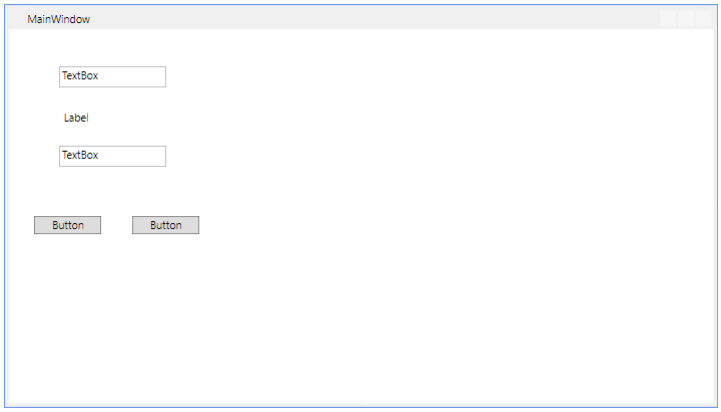
Ansicht > Toolbox

oder per Mausklick auf die **Toolbox**-Schaltfläche am linken Fensterrand:



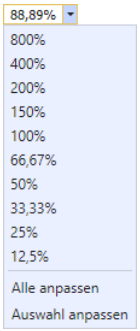
Erweitern Sie nötigenfalls im **Toolbox**-Fenster die Liste mit den **häufig verwendeten WPF-Steuerelementen**, und erstellen Sie auf dem Anwendungsfenster zwei **TextBox**-Objekte, ein **Label**-Objekt sowie zwei **Button**-Objekte per Drag & Drop (Ziehen und Ablegen), indem Sie einen linken Mausklick auf den jeweiligen **Toolbox**-Eintrag setzen, den Mauszeiger mit gedrückter Taste zum Ziel bewegen und dort die Taste wieder loslassen.

Das Ergebnis sollte ungefähr so aussehen:

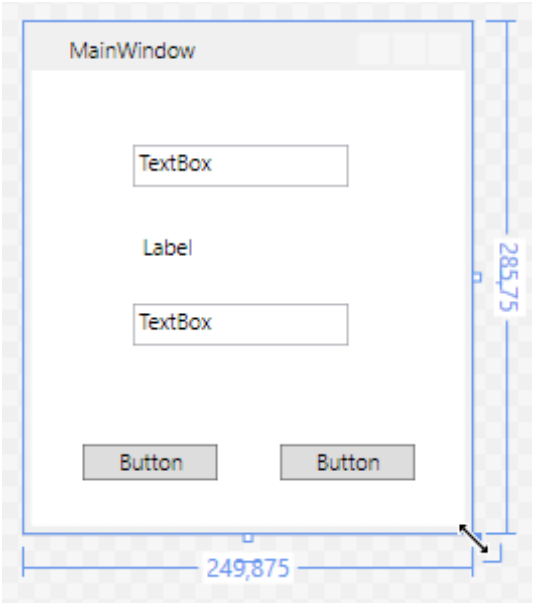


5.12.4 Positionen und Größen der Steuerelemente gestalten

Nun können Sie wie in einem Grafikprogramm die Positionen und Größen der Fensterbestandteile verändern, um das gewünschte Layout zu erzielen. Zur Erleichterung der Arbeit lässt sich mit dem **Zoom-Werkzeug** des WPF-Designers (unter der Fenster-Vorschau, am linken Rand) eine passende Ansicht einstellen:

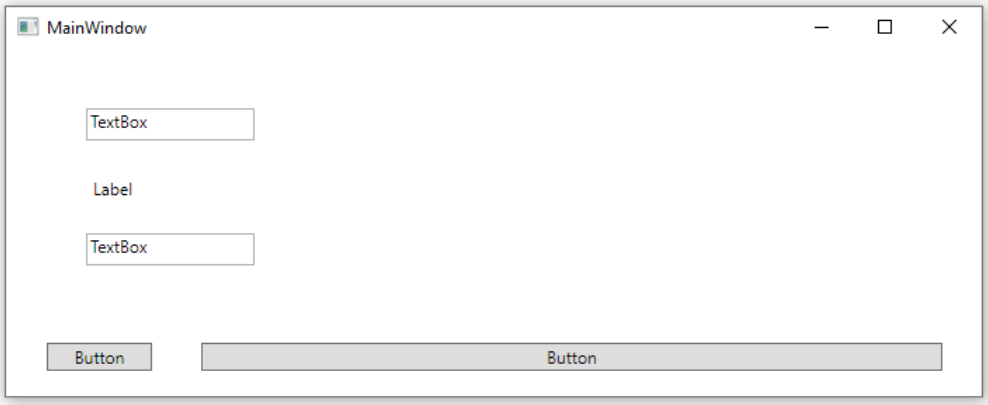


Wählen Sie zunächst für das Hauptfenster eine Größe über den Anfasser unten rechts:

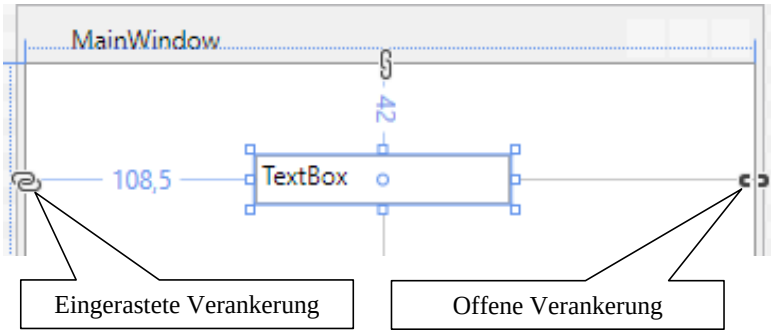


Die aktuelle Breite und Höhe in geräteunabhängigen Pixeln wird numerisch angezeigt. Achten Sie darauf, dass Sie wirklich das Hauptfenster erwischen (ein Objekt der von **Window** abstammenden Klasse `MainWindow`) und nicht etwa den darin enthaltenen und aus didaktischen Gründen vorläufig ignorierten Container (ein Objekt aus der Klasse **Grid**).

Ändert der Benutzer im laufenden Programm die Fenstergröße, dann hängt das Orts- und Größenverhalten eines Steuerelements davon ab, zu welchen Seiten des umgebenden Containers es einen festen **Randabstand** einhält, z. B.:



Per Voreinstellung sind die Steuerelemente links und oben angedockt, was bei einem markierten Steuerelement durch Verankerungssymbole an den Fensterrändern angezeigt wird, z. B.:

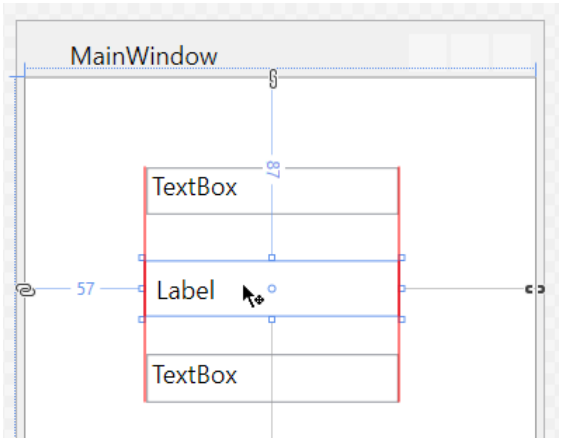


Um eine Verankerung vorzunehmen oder aufzuheben, klickt man auf den zugehörigen Verankerungspunkt. Ist beim Lösen einer Verankerung die gegenüberliegende Seite gerade frei, springt die Verankerung dorthin.

Soll ein Steuerelement z. B. vom horizontalen Größenwachstum des Hauptfensters profitieren, auf vertikale Änderungen aber nicht reagieren,

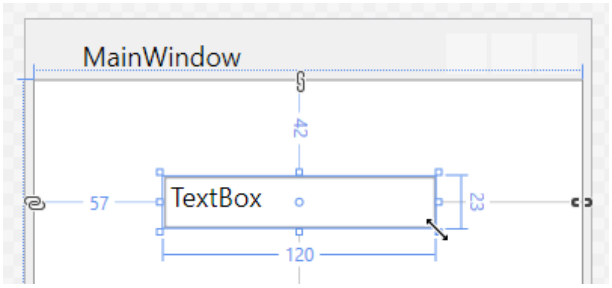
verankert man es oben, links und rechts.

Bei der Positions- und Größenanpassung von Steuerelementen helfen **Ausrichtungs- bzw. Führungslinien**, die auftauchen und dabei anziehend wirken, sobald die horizontale oder vertikale Position von potentiellen Kandidaten für eine Ausrichtung erreicht wird. Im folgenden Beispiel wurde das **Label**-Objekt bewegt:

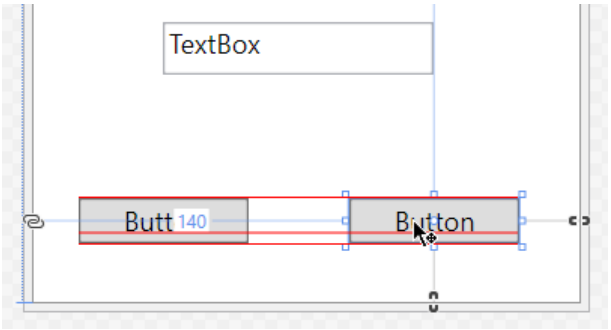


Man kann z. B. so vorgehen, um die Steuerelemente für das Beispielprogramm anzuordnen:

- Größe und Position für das obere Textfeld festlegen, z. B.:



  Breite und Höhe des Textfelds werden numerisch angezeigt.
- Das **Label**-Objekt und das untere Textfeld nacheinander ...
  - linksbündig mit ungefähr gleichem Abstand unter das jeweils darüber liegende Steuerelement setzen
  - und dieselbe Breite wählen
- Bei den **Button**-Objekten helfen die Führungslinien dabei, eine geeignete Position zu finden:

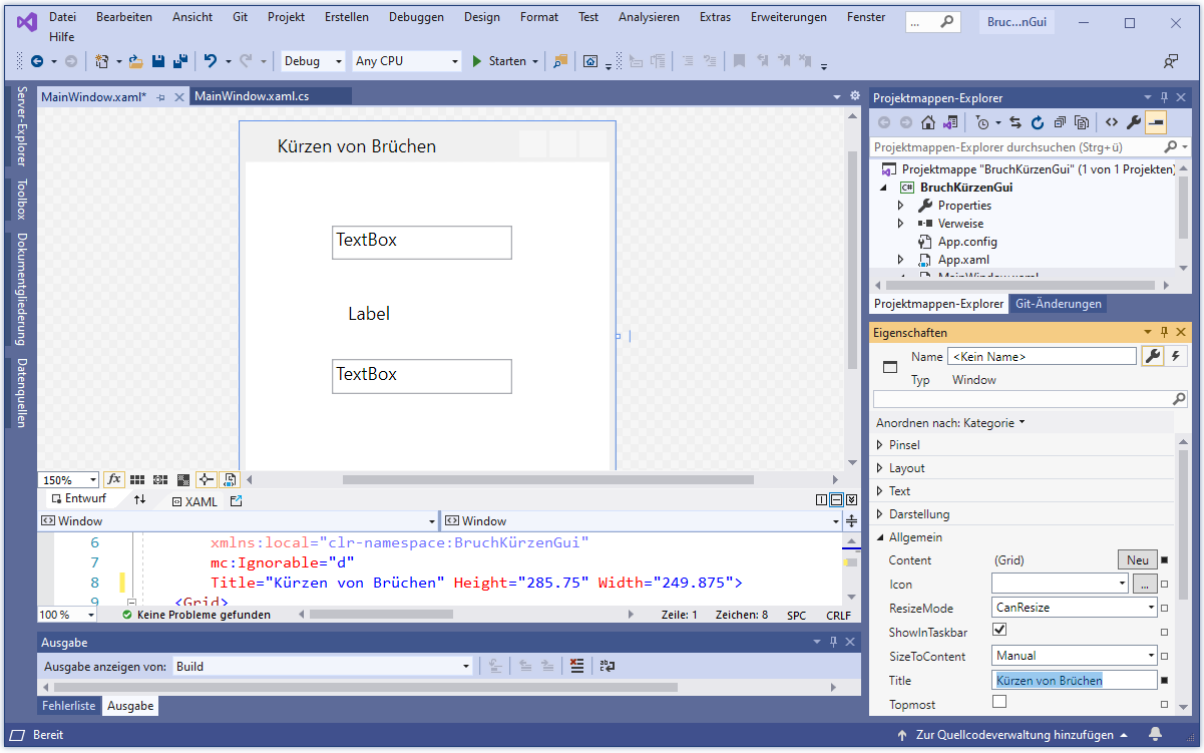


5.12.5 Eigenschaften der Steuerelemente ändern

Im Eigenschaftenfenster der Entwicklungsumgebung, das bei Bedarf mit der Funktionstaste **F4** oder mit dem Menübefehl

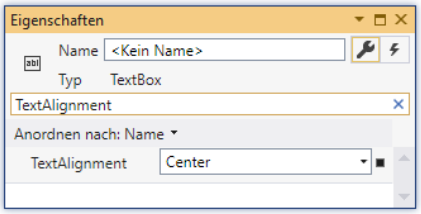
Ansicht > Eigenschaftenfenster

zu öffnen ist, lassen sich diverse Eigenschaften (im Sinne von Abschnitt 5.5!) der markierten Objekte festlegen, z. B. der **Title** des Fensters:

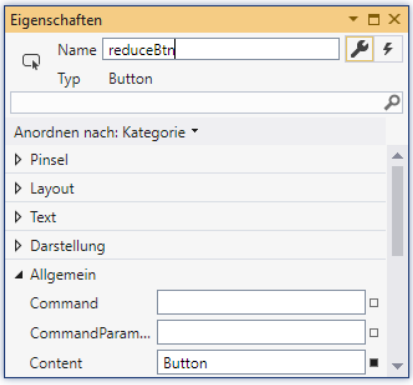


Wenn Sie eine Eigenschaft mit bekanntem Namen bei der voreingestellten **Anordnung nach Kategorie** nicht finden, können Sie ...

- die **Anordnung nach Namen** wählen
- oder das Suchfeld des Eigenschaftsfensters benutzen, z. B.:



Neben den Eigenschaftsmodifikationen ist insbesondere die Möglichkeit von Relevanz, den Instanzvariablennamen eines Steuerelements zu ändern, z. B.:



Wir ändern …

- die Namen der Instanzvariablen zu den **TextBox**- und den **Button**-Objekten auf `numTb`, `denomTb`, `reduceBtn` und `closeBtn`,
- die **Content**-Eigenschaft der Schaltflächen, sodass sinnvolle Beschriftungen entstehen (z. B. Kürzen, Beenden),
- die **Content**-Eigenschaft des Labels, um einen behelfsmäßigen Bruchstrich zu erzielen, und setzen die **HorizontalContentAlignment** des Labels auf den Wert **Center**,
- für beide **TextBox**-Objekte die Eigenschaft **Text** auf eine leere Zeichenfolge sowie die Eigenschaften **TextAlignment** und **VerticalContentAlignment** auf den Wert **Center**,
- die **Title**-Eigenschaft des **Window**-Objekts (z. B. auf „Kürzen von Brüchen“),
- die Eigenschaft **ResizeMode** des **Window**-Objekts auf den Wert **NoResize**, sodass sich die Fenstergrüße nicht verändern lässt,
- die **IsDefault**-Eigenschaft der linken Schaltfläche auf den Wert **True**, sodass diese Schaltfläche im laufenden Programm per **Enter**-Taste angesprochen werden kann.[1]

Beim Verschieben eines Steuerelements im WPF-Designer (siehe Abschnitt 5.12.4) haben wir übrigens seine **Margin**-Eigenschaft verändert und so für jede Seite einen Randabstand festgelegt.

Zum Markieren eines Steuerelements (im WPF-Designer und im XAML-Code) stehen folgende Techniken bereit:

- Mausklick auf das Steuerelement im WPF-Designer oder im XAML-Fenster
- Wenn der Designer den Tastaturfokus besitzt, kann man die Reihe der Steuerelemente per Tabulator-Taste vorwärts und bei zusätzlich gedrückter **Umschalt**-Taste rückwärts durchlaufen.

Um *mehrere* Steuerelemente zu markieren, kann man ...

- im WPF-Designer ein Markierungsrechteck um die gewünschten Teilnehmer ziehen,
- im WPF-Designer bei gedrückter **Strg**-Taste die Steuerelemente nacheinander anklicken

Die Eigenschaften von mehreren, gleichzeitig markierten Steuerelementen lassen sich in *einem* Arbeitsgang ändern.

Zwar ist eine Eigenschaftsmodifikation zur *Entwurfszeit* besonders bequem per Eigenschaftenfenster zu bewerkstelligen, doch muss man trotzdem wissen, wie der Eigenschaftszugriff per C# - Anweisung erfolgt, weil viele Steuerelementeigenschaften auch zur *Laufzeit* (dynamisch) geändert werden müssen.

5.12.6 Automatisch erstellter und gepflegter Quellcode

Aufgrund Ihrer kreativen Tätigkeit beim GUI-Design erzeugt die Entwicklungsumgebung im Hintergrund Quellcode zu einer neuen Klasse namens `MainWindow`, die von der Klasse **Window** im Namensraum **System.Windows** abstammt. Aber auch *Sie* werden signifikanten Quellcode zu dieser Klasse beisteuern. Damit sich die beiden Autoren nicht in die Quere kommen, wird der Quellcode der Klasse `MainWindow` auf zwei Dateien verteilt:

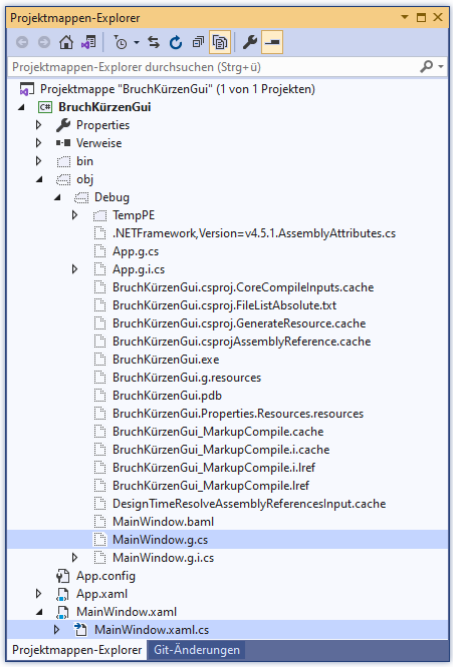
---

[1] Anders als in C# wird das boolesche Literal **True** in XAML groß geschrieben, z. B.:

```
<Button x:Name="reduceBtn" Content="Kürzen" ... IsDefault="True"/>
```

- **MainWindow.xaml.cs** im Projektordner
  Hier werden die von Ihnen erstellten Methoden landen (siehe unten).
- **MainWindow.g.cs** im Projektunterordner **...\obj\Debug**[1]
  Hier landet der durch Interpretation der XAML-Datei **MainWindow.xaml** erstellte C# - Quellcode. In dieser Datei dürfen Sie keine Änderungen vornehmen, um die Entwicklungsumgebung nicht aus dem Tritt zu bringen. Daran erinnert der Namensbestandteil *g* für *generated*.[2]

Wer die Datei **MainWindow.g.cs** lokalisieren und öffnen möchte, kann den Projektmappen-Explorer über den Symbolschalter dazu überreden, **alle Dateien anzuzeigen**, z. B.:



Dem C# - Compiler wird durch das Schlüsselwort **partial** in der Klassendefinition signalisiert, dass der Quellcode auf mehrere Dateien verteilt ist, z. B. in der Quellcodedatei **MainWindow.xaml.cs**:

```
using System;
using System.Collections.Generic;
   .  .  .
using System.Windows.Shapes;

namespace BruchKürzenGui {
   /// <summary>
   /// Interaktionslogik für MainWindow.xaml
   /// </summary>
   public partial class MainWindow : Window {
      public MainWindow() {
         InitializeComponent();
      }
   }
}
```

Der `MainWindow`-Quellcode in der Datei **MainWindow.xaml.cs** enthält einen Konstruktor, welcher die Methode **InitializeComponent()** aufruft. Diese wird in der Datei **MainWindow.g.cs** von der Entwicklungsumgebung implementiert.

---

[1] Das gilt bei Verwendung der Debug-Konfiguration (vgl. Abschnitt 3.3.4.4).

[2] Warum es neben **MainWindow.g.cs** im selben Ordner noch eine inhaltsidentische Datei mit dem Namen **MainWindow.g.i.cs** gibt, müssen wir nicht erforschen.

Aufgrund Ihrer Tätigkeit im WPF-Designer enthält die Fensterklasse `MainWindow` im Sinne der im Abschnitt 5.9 behandelten Komposition mehrere Objekte anderer Klassen, die Steuerelemente der grafischen Bedienoberfläche repräsentieren. In der Datei **MainWindow.g.cs** finden sich die Deklarationen der zugehörigen Instanzvariablen, nachdem das Projekt erstellt worden ist:

```
internal System.Windows.Controls.TextBox numTb;
internal System.Windows.Controls.TextBox denomTb;
internal System.Windows.Controls.Label label;
internal System.Windows.Controls.Button reduceBtn;
internal System.Windows.Controls.Button closeBtn;
```

Über den Modifikator **internal** wird der Zugriff durch alle Klassen im eigenen Assembly erlaubt (vgl. Abschnitt 5.11).

Neben der Klasse `MainWindow` mit der XAML-Datei **MainWindow.xaml** und dem C# - Quellcode-Duo **MainWindow.xaml.cs** und **MainWindow.g.cs** enthält das Projekt noch die Klasse `App` mit ...

- der XAML-Datei **App.xaml**
- den Quellcodedateien **App.xaml.cs** und **App.g.cs**.

Wer die **Main()** - Methode zu unserer WPF-Anwendung vermisst, findet sie in der Datei **App.g.cs**:

```
public static void Main() {
    BruchKürzenGui.App app = new BruchKürzenGui.App();
    app.InitializeComponent();
    app.Run();
}
```

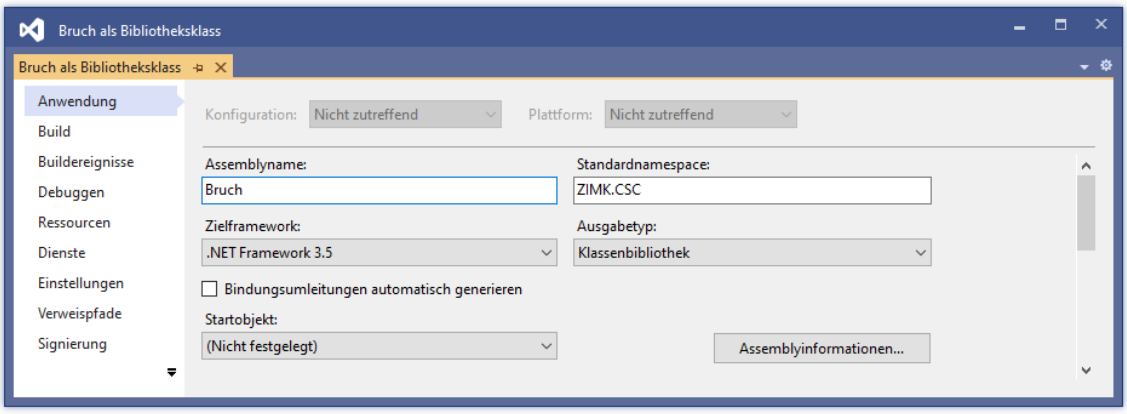
Weitere Details zu den vom Visual Studio bei WPF-Projekten gepflegten Quellcodedateien folgen im Abschnitt 12.3.4.

5.12.7 Bibliotheks-Assembly mit der Bruch-Klasse einbinden

Aus der letzten Ausbaustufe der `Bruch`-Klasse ist ein Bibliotheks-Assembly erstellt worden, indem im Visual Studio nach

Projekt > Eigenschaften > Anwendung

der Ausgabetyp geändert worden ist:



Außerdem wurde die Klasse `Bruch` in den Namensraum `ZIMK.CSC` aufgenommen:

```
   using System;

   namespace ZIMK.CSC {

       public class Bruch {
        . . .
       }
}
```

Das Bibliotheks-Assembly wurde mit der Release-Konfiguration erstellt.

Gehen Sie folgendermaßen vor, um Objekte unserer Beispielklasse `Bruch` im neuen Programm nutzen zu können:

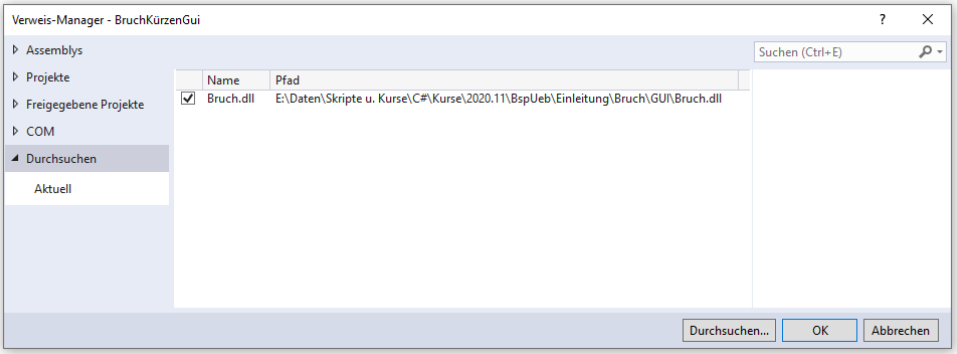
- Kopieren Sie das Bibliotheks-Assembly mit der Klasse `Bruch`

  …\BspUeb\Klassen und Objekte\Bruch\Bibliothek\bin\Release\Bruch.dll

  in den Ordner des neuen Projekts, also z. B. nach

  U:\Eigene Dateien\C#\BspUeb\Einleitungsbeispiel Bruchrechnen\GUI

  Beim Erstellen der Anwendung landet **Bruch.dll** im selben Ordner wie das Anwendungs-Assembly **BruchKürzenGui.exe**, was bei Projekt-lokalen Assemblies angemessen ist. Allgemein benötigte Bibliotheks-Assemblies sollten so abgelegt werden (z. B. im Global Assembly Cache, GAC, siehe Abschnitt 2.4.5), dass sie zur Nutzung in einem konkreten Projekt nicht kopiert werden müssen.
- Nehmen Sie per Projektmappen-Explorer das Bibliotheks-Assembly **Bruch.dll** in die Verweisliste des Projekts auf (vgl. Abschnitt 3.3.6.1), z. B.:



5.12.8 Ereignisbehandlungsmethoden anlegen

Zunächst erstellen wir zu den beiden Befehlsschaltern jeweils eine Methode, die durch das Betätigen des Schalters (z. B. per Mausklick) ausgelöst werden soll. Setzen Sie im Formulardesigner einen Doppelklick auf den Befehlsschalter `reduceBtn` (mit der Beschriftung *Kürzen*), sodass die Entwicklungsumgebung in der Quellcodedatei **MainWindow.xaml.cs** die private Instanzmethode `reduceBtn_Click()` der Klasse `MainWindow` mit leerem Rumpf anlegt

```
   private void reduceBtn_Click(object sender, RoutedEventArgs e) {

   }
```

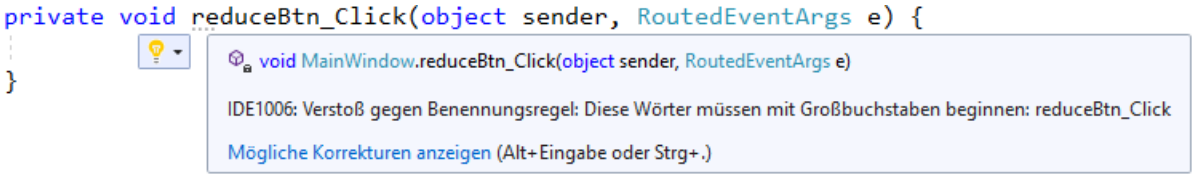
und die Datei im Editor öffnet.

Außerdem wird die Methode in der XAML-Datei zum Anwendungsfenster

```
   <Button x:Name="reduceBtn" Content="Kürzen" HorizontalAlignment="Left"
      Margin="70,201,0,0" VerticalAlignment="Top" Width="75" IsDefault="True"
      Click="reduceBtn_Click"/>
```

dem **Button**-Ereignis **Click** zugeordnet.

Lässt man den Mauszeiger auf der punktierten Unterstreichung des Methoden-Namensanfangs verharren, äußert das Visual Studio eine Kritik an der eigenen Produktion:



Für die Namen von Methoden sollte in C# grundsätzlich das Pascal Casing (mit großem Anfangsbuchstaben) benutzt werden (vgl. Abschnitte 4.1.5 und 5.3.1). Weil es sich nur um eine Konvention handelt, verzichten wir auf eine Korrektur der Assistentenproduktion, die uns bei konsequenter Umsetzung zu ständigen Eingriffen zwingen würde. Man kann das Problem auch aus der Welt schaffen durch die Entscheidung, bei Namen von Feldern mit Zugriffebene **internal** (abweichend von den Empfehlungen im Abschnitt 5.2.2) das Pascal Casing zu verwenden.

Mit Hilfe eines Objekts aus unserer Klasse `Bruch` ist die benötigte Funktionalität leicht zu implementieren, z. B.:

```
private void reduceBtn_Click(object sender, RoutedEventArgs e) {
    var b = new Bruch();
    try {
        b.Zaehler = Convert.ToInt32(numTb.Text);
        b.Nenner = Convert.ToInt32(denomTb.Text);
        b.Kuerze();
        numTb.Text = b.Zaehler.ToString();
        denomTb.Text = b.Nenner.ToString();
    } catch {
        MessageBox.Show("Eingabefehler", "Fehler", MessageBoxButton.OK,
                        MessageBoxImage.Error);
    }
}
```

Damit die Klasse Bruch ohne Namensraum-Präfix angesprochen werden kann, wird der Namensraum `ZIMK.CSC` in die Quellcodedatei importiert:

```
using ZIMK.CSC;
```

Der Bequemlichkeit halber wird bei jedem Aufruf von `reduceBtn_Click()` ein neues Objekt erzeugt (vgl. Übungsaufgabe im Abschnitt 5.13).

Tritt im **try**-Block der **try-catch** - Anweisung eine Ausnahme auf (z. B. beim Versuch, eine irreguläre Benutzereingabe zu konvertieren), dann wird der **catch**-Block ausgeführt, und es erscheint eine Fehlermeldung. Im Aufruf der **MessageBox**-Methode **Show**() sorgen Werte der Enumerationen (siehe unten) **MessageBoxButton** bzw. **MessageBoxImage** als Aktualparameter für die gewünschte Ausstattung der Meldungsdialogbox mit Schaltflächen und einem Symbol. Bei einem gelungenen Ablauf wandern Informationen zwischen den **Text**-Eigenschaften der beiden **TextBox**-Objekte (Datentyp **String**) und den `Bruch`-Instanzvariablen `zaehler` und `nenner` (Datentyp: **int**) hin und her.

Erstellen Sie nun per Doppelklick auf den Befehlsschalter `closeBtn` (mit der Beschriftung *Beenden*) den Rohling für seine Klickereignisbehandlungsmethode, und ergänzen Sie einen Aufruf der **Window**-Methode **Close()**, die das Hauptfenster schließt und damit das Programm beendet, z. B.:

```
private void closeBtn_Click(object sender, RoutedEventArgs e) {
    Close();
}
```

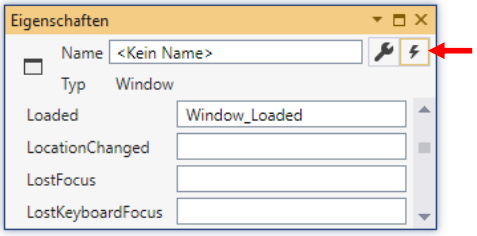
Es wäre nett, wenn nach dem Start unseres Programms das Zähler-Textfeld den Tastaturfokus hätte, sodass der Benutzer unmittelbar mit der Werteingabe beginnen könnte, ohne zuvor den Tastaturfokus (z. B. per Maus) setzen zu müssen. Eine Lösungsmöglichkeit besteht darin, das zu privilegie-

rende Steuerelement über die Methode **Focus()** der Klasse **UIElement** aufzufordern, den Fokus zu übernehmen, z. B.:

```
numTb.Focus();
```

Damit diese Anweisung beim Laden des Anwendungsfensters ausgeführt wird, stecken wir sie in eine Ereignisbehandlungsmethode zum **Loaded**-Ereignis der Fensterklasse:

- Markieren Sie im WPF-Designer das Anwendungsfenster.
- Wechseln Sie im Eigenschaftenfenster zur Liste mit den Ereignissen, die von der Klasse **Window** angeboten werden.



- Setzen Sie einen Doppelklick auf das Texteingabefeld zum Ereignis **Loaded**. Wenn das Anwendungsfenster das Ereignis **Loaded** „feuert“, ist der richtige Moment für unsere Initialisierungsarbeiten gekommen.
- Daraufhin wird in der Quellcodedatei **MainWindow.xaml.cs** die private Instanzmethode `Window_Loaded()` zu unserer Fensterklasse `MainWindow` angelegt:

```
private void Window_Loaded(object sender, RoutedEventArgs e) {

}
```

  Außerdem wird die Methode in der XAML-Datei zum Hauptfenster

```
<Window x:Class="BruchKürzenGui.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   . . .
    Title="Kürzen von Brüchen" Height="302" Width="349" Loaded="Window_Loaded">
```

  dem Ereignis **Loaded** zugeordnet.
- Ergänzen Sie im Rumpf dieser Methode den oben beschriebenen **Focus()** - Aufruf.

Veranlassen Sie mit der Tastenkombination **Strg+F5** das Übersetzen und die Ausführung der fertigen Anwendung.

Die Assembly-Datei **BruchKürzenGui.exe** findet sich im Projektunterordner **…\bin\Debug**, weil die Erstellungskonfiguration **Debug** verwendet wurde (vgl. Abschnitt 3.3.4.4).

Zum Installieren des Programms genügt es, den Programmordner **…\bin\Debug** zu kopieren. Dabei darf die automatisch beim Erstellen des Projekts dorthin beförderte Assembly-Datei **Bruch.dll** mit der Klasse `Bruch` nicht vergessen werden.

5.13 Übungsaufgaben zum Kapitel 5

1) Welche von den folgenden Aussagen sind richtig?

1. Alle Instanzvariablen einer Klasse müssen von elementarem Typ sein.
2. In einer Klasse können mehrere Methoden mit demselben Namen existieren.
3. Bei der Definition eines Konstruktors ist der Rückgabetyp **void** anzugeben.
4. Mit der Datenkapselung wird verhindert, dass ein Objekt auf die Instanzvariablen anderer Objekte derselben Klasse zugreifen kann.
5. Als Wertaktualparameter sind nicht nur Variablen erlaubt, sondern beliebige Ausdrücke mit kompatiblem (erweiternd konvertierbarem Typ).
6. Ändert man den Rückgabetyp einer Methode, dann ändert sich auch ihre Signatur.

2) Erläutern Sie den Unterschied zwischen einem **readonly** - deklarierten Feld und einer Eigenschaft ohne **set** - Implementierung, die man auch als *get-only* - Eigenschaft bezeichnen könnte.

3) Im folgenden Programm soll die statische `Bruch`-Eigenschaft `Anzahl` ausgelesen werden:

```
using System;
class Bruchrechnung {
  static void Main() {
    var b1 = new Bruch();
    var b2 = new Bruch();
    Console.WriteLine("Jetzt sind wir " + Bruch.Anzahl);
  }
}
```

Es liegt die folgende die Eigenschaftsdefinition zugrunde:

```
public static int Anzahl {
  get {
    return Anzahl;
  }
}
```

Statt der erwarteten Auskunft:

```
Jetzt sind wir 2
```

erhält man jedoch (beim Programmstart im Konsolenfenster) die Fehlermeldung:

```
Process is terminated due to StackOverflowException.
```

Offenbar hat sich ein Fehler in die Eigenschaftsdefinition eingeschlichen, den der Compiler nicht bemerkt.

4) Für diese Aufgabe werden Grundbegriffe der linearen Algebra benötigt, wobei die beteiligten Formeln in der Aufgabenbeschreibung enthalten sind: Erstellen Sie eine Klasse für Vektoren im $\mathbb{R}^2$ (in der der reellen Zahlenebene), die mindestens über Methoden bzw. Eigenschaften mit den folgenden Leistungen verfügt:

- **Länge** ermitteln

  Der Betrag eines Vektors $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ ist definiert durch:

  $$|x| := \sqrt{x_1^2 + x_2^2}$$

  Verwenden Sie die Klassenmethode **Math.Sqrt()**, um die Quadratwurzel aus einer **double**-Zahl zu berechnen.

- Vektor auf Länge Eins **normieren**
  Dazu dividiert man beide Komponenten durch die Länge des Vektors, denn mit
  $\tilde{x} := (\tilde{x}_1, \tilde{x}_2)$ sowie $\tilde{x}_1 := \frac{x_1}{\sqrt{x_1^2 + x_2^2}}$ und $\tilde{x}_2 := \frac{x_2}{\sqrt{x_1^2 + x_2^2}}$ gilt:

$$|\tilde{x}| = \sqrt{\tilde{x}_1^2 + \tilde{x}_2^2} = \sqrt{\left(\frac{x_1}{\sqrt{x_1^2 + x_2^2}}\right)^2 + \left(\frac{x_2}{\sqrt{x_1^2 + x_2^2}}\right)^2} = \sqrt{\frac{x_1^2}{x_1^2 + x_2^2} + \frac{x_2^2}{x_1^2 + x_2^2}} = 1$$

- Vektoren (komponentenweise) **addieren**
  Die Summe der Vektoren $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ und $y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ ist definiert durch:

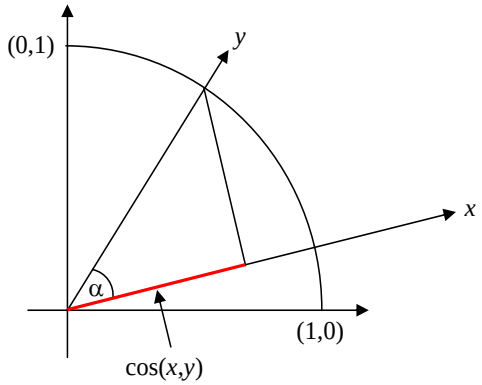
$$\begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \end{pmatrix}$$

- **Skalarprodukt** zweier Vektoren ermitteln
  Das Skalarprodukt der Vektoren $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ und $y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ ist definiert durch:

$$x \cdot y := x_1 y_1 + x_2 y_2$$

- **Winkel** zwischen zwei Vektoren in Grad ermitteln
  Für den Kosinus des Winkels, den zwei Vektoren *x* und *y* im mathematischen Sinn (links herum) einschließen, gilt:[1]

$$\cos(x, y) = \frac{x \cdot y}{|x||y|}$$



  Um aus cos(*x*, *y*) den Winkel α in Grad zu ermitteln, können Sie folgendermaßen vorgehen:
  - mit der Klassenmethode **Math.Acos()** den Winkel im Bogenmaß ermitteln
  - das Bogenmaß (*rad*) nach folgender Formel in Grad umrechnen (*deg*):

$$deg = \frac{rad}{2\pi} \cdot 360$$

[1] Dies folgt aus dem Additionstheorem für den Kosinus.

- **Rotation** eines Vektors um einen bestimmten Winkelgrad
  Mit Hilfe der Rotationsmatrix

$$D := \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}$$

  kann der Vektor *x* um den Winkel α (im Bogenmaß!) gedreht werden:

$$x' = D\,x = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \cos(\alpha)\,x_1 - \sin(\alpha)\,x_2 \\ \sin(\alpha)\,x_1 + \cos(\alpha)\,x_2 \end{pmatrix}$$

  Zur Berechnung der trigonometrischen Funktionen stehen die Klassenmethoden **Math.Cos()** und **Math.Sin()** bereit. Winkelgrade (*deg*) müssen nach folgender Formel in das von **Cos()** und **Sin()** benötigte Bogenmaß (*rad*) umgerechnet werden:

$$rad = \frac{deg}{360} \cdot 2\pi$$

Erstellen Sie ein Demonstrationsprogramm, das Ihre Vektorklasse verwendet und ungefähr den folgenden Programmablauf ermöglicht (Eingabe fett):

```
Vektor 1:               ( 1,00;  0,00)
Vektor 2:               ( 1,00;  1,00)

Länge von Vektor 1:      1,00

Länge von Vektor 2:      1,41

Winkel:                 45,00 Grad

Um wie viel Grad soll Vektor 2 gedreht werden: 45

Neuer Vektor 2          ( 0,00;  1,41)
Neuer Vektor 2 normiert ( 0,00;  1,00)

Summe der Vektoren      ( 1,00;  1,00)
```

5) Erstellen Sie eine Klasse mit einer statischen Methode zur Berechnung der Fakultät über einen rekursiven Algorithmus. Erstellen Sie eine Testklasse, welche die rekursive Fakultätsmethode benutzt. Diese Aufgabe dient dazu, an einem einfachen Beispiel mit rekursiven Methodenaufrufen zu experimentieren. Für die Praxis ist die rekursive Fakultätsberechnung weniger geeignet.

Wer Spaß an der Programmierpraxis zur Lösung mathematischer Aufgaben gefunden hat, kann auch noch ein rekursives Verfahren zur Berechnung der Fibonacci-Zahlen entwerfen. Die Fibonacci-Folge entsteht, wenn ausgehend von den Zahlen 0 und 1 alle weiteren Folgenglieder als Summe der beiden Vorgänger berechnet werden:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Die Idee zu dieser Übungsaufgabe stammt samt Lösungsvorschlag von Paul Frischknecht.

6) Ersetzen Sie beim GUI-Bruchkürzungsprogramm im Abschnitt 5.12 die lokale `Bruch`-Referenzvariable in der Klick-Ereignisbehandlungsmethode zum Befehlsschalter `reduceBtn` (mit der Beschriftung *Kürzen*) durch eine Instanzvariable der Hauptfensterklasse `MainWindow`. So wird vermieden, dass bei jedem Methodenaufruf ein neues `Bruch`-Objekt entsteht, das nach Beenden der Methode dem Garbage Collector überlassen wird.

7) Lokalisieren Sie bitte im folgenden Quellcode mit einer Kurzform der Klasse `Bruch`

```
using System;
public class Bruch {
   int zaehler, nenner = 1;                                          1
   string etikett = "";
   static int anzahl;                                                2

   public Bruch(int zpar, int npar, String epar) {
      Zaehler = zpar; Nenner = npar;
      Etikett = epar; Anzahl++;
   }
   public Bruch() {Anzahl++;}                                        3

   public int Zaehler {  .  .  .  }
   public int Nenner {                                               4
      get {return nenner;}
      set {
         if (value != 0)
            Nenner = value;
      }
   }
   public string Etikett {  .  .  .  }                               5

   public void Addiere(Bruch b) {
      Zaehler = zaehler*b.nenner + b.zaehler*nenner;                 6
      Nenner = nenner*b.nenner;
      Kuerze();
   }
   public Bruch Klone() {                                            7
      return new Bruch(zaehler, nenner, etikett);
   }
   public void Kuerze() {  .  .  .  }
   public bool Frage() {  .  .  .  }
   public void Zeige() {
      string luecke = "";                                            8
      for (int i=1; i <= etikett.Length; i++)
         luecke = luecke + " ";
      Console.WriteLine(" {0}   {1}\n {2} -----\n {0}   {3}\n",
                        luecke, zaehler, etikett, nenner);
   }
   public static Bruch operator+ (Bruch b1, Bruch b2) {              9
      Bruch temp = new Bruch(b1.zaehler * b2.nenner + b1.nenner * b2.zaehler,
         b1.nenner * b2.nenner, "");
      temp.Kuerze();
      return temp;
   }
   public static Bruch BenDef(string e) {                            10
      Bruch b = new Bruch(0, 1, e);                                  11
      if (b.Frage()) {
         b.Kuerze();
         return b;
      } else
         return null;
   }
   public static int Anzahl {                                        12
      get {return anzahl;}
      private set {anzahl = value;}
   }
}
```

12 Begriffe der objektorientierten Programmierung, und tragen Sie die Positionen in die folgende Tabelle ein

Begriff	Pos.
Definition einer Instanzmethode mit Referenzrückgabe	
Deklaration lokale Variable	
Def. einer Instanzmeth. mit Wert-parameter vom Typ einer Klasse	
Deklaration von Instanzvariablen	
Methodenaufruf	
Definition einer statischen Eigenschaft	

Begriff	Pos.
Konstruktordefinition	
Deklaration einer Klassenvariablen	
Objekterzeugung	
Definitionskopf einer Klassenmethode	
Definition einer Instanzeigenschaft	
Operatorüberladung	

Zum Eintragen benötigen Sie nicht unbedingt eine gedruckte Variante des Manuskripts, sondern können auch das interaktive PDF-Formular

...\BspUeb\Klassen und Objekte\Bruch\Begriffe lokalisieren.pdf

benutzen. Die Idee zu dieser Übungsaufgabe stammt aus Mössenböck (2019).

6 Weitere .NETte Typen

Nachdem wir uns ausführlich mit elementaren Datentypen und mit Klassen beschäftigt haben, wird in diesem Kapitel Ihr Wissen über das *Common Type System* (CTS) der .NET - Plattform abgerundet. Sie lernen u. a. die folgenden Datentypen kennen:

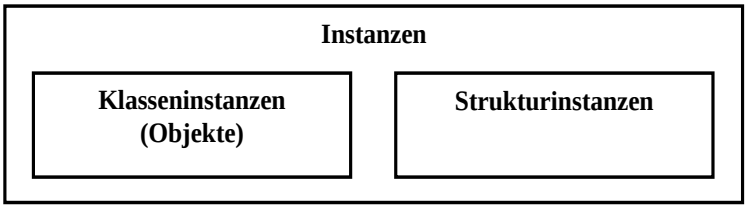
- Strukturen als Klassenalternative mit Wertsemantik
- Arrays als Container für eine feste Anzahl von Elementen desselben Datentyps
- Klassen zur Verwaltung von Zeichenketten (**String**, **StringBuilder**)
- Aufzählungstypen (Enumerationen)
- Tupel als syntaktisch bequeme Strukturen zur Verwaltung von Elementen mit unterschiedlichen Datentypen

6.1 Strukturen

Klassen und Objekte haben ohne Zweifel einen sehr hohen Nutzen, verursachen aber auch Kosten, z. B. beim Erzeugen von Objekten, bei der Referenzverwaltung und bei der Entsorgung per Garbage Collector. Daher stellt die .NET - Plattform mit den *Strukturen* auch *Werttypen* zur Verfügung, die in manchen Situationen bei geringerem Ressourcen-Verbrauch „echte“ Klassen ersetzen und somit die Performanz der Software steigern können.

Beim Design eines Strukturtyps können im Wesentlichen dieselben Member eingesetzt werden wie bei einer Klassendefinition (Felder beliebigen Typs, Methoden, Eigenschaften, Konstruktoren, ein statischer Konstruktor, usw.).[1] Eine Variable vom Typ einer Struktur enthält jedoch keine *Referenz* auf ein Heap-Objekt, sondern alle Feldinhalte ihres Typs. Wie bei Variablen mit einem elementaren Datentyp liegt keine Referenz-, sondern eine **Wertsemantik** vor.

Für ein Individuum nach dem Bauplan einer Struktur soll *nicht* der Begriff *Objekt*, sondern der allgemeinere Begriff *Instanz* verwendet werden.



Von *Instanzvariablen* und *-methoden* sprechen wir bei Objekten *und* bei Strukturinstanzen.

Eine Struktur eignet sich als Datentyp bei folgender Konstellation:

- Kleine Instanzen
  Der Typ ist relativ einfach aufgebaut, hat also nur wenige Instanzvariablen. Microsoft empfiehlt für Strukturinstanzen eine Gesamtgröße von weniger als 16 Bytes.[2]
- Wertsemantik erwünscht
  Bei einem Methodenaufruf wird eine als Wertaktualparameter verwendete Strukturinstanz komplett kopiert, während von einem Parameter*objekt* nur die Adresse übergeben wird. Das Original der kopierten Strukturinstanz wird durch den Methodenaufruf nicht verändert, was beim Parameterobjekt hingegen möglich ist.

---

[1] Es ist allerdings kein Destruktor erlaubt.

[2] http://msdn.microsoft.com/en-us/library/ms229017.aspx

- Unveränderliche Instanzen
  Es wird empfohlen, Strukturinstanzen als **unveränderlich** (engl. *immutable*) zu konzipieren, so z. B. auf einer Microsoft Webseite mit .NET - Designrichtlinien:[1]

  DO NOT define mutable value types.

  Aufgrund der Wertsemantik kann es bei einer Struktur nämlich leicht passieren, dass man lediglich eine *Kopie* modifiziert an Stelle der eigentlich zu ändernden Instanz (siehe z. B. Griffiths 2013, S. 97). Um die Unveränderlichkeit eines Typs zu erreichen, muss man die Datenkapselung realisieren (also die Instanzvariablen schützen) und außerdem auf Methoden und Eigenschaften verzichten, die Instanzvariablen ändern. Wie gleich an Beispielen zu sehen ist, nimmt Microsoft die eigene Unveränderlichkeitsempfehlung allerdings nicht immer ernst.
- Nullinitialisierung akzeptabel
  Bei einer Struktur muss sichergestellt sein, dass die Nullinitialisierung aller Felder zu einer regulären Instanz führt (siehe Abschnitt 6.1.1).
- Keine Vererbung erforderlich
  Bei Strukturen fehlt die Möglichkeit, per Vererbung (siehe Kapitel 7) eine Hierarchie spezialisierter Typen aufzubauen. Das Implementieren von Schnittstellen (siehe Kapitel 9) ist aber möglich.

Werden von einem Typ viele Instanzen benötigt, kann sich die Vermeidung von Objektkreationen durch die Verwendung eines Strukturtyps lohnen. Typische Anwendungsbeispiele für Strukturen:

- Punkte in einem zwei- oder dreidimensionalen Raum
  Ein Beispiel ist die von Microsoft definierte BCL-Struktur **Point** im Namensraum **System.Windows**. Sie besitzt allerdings abweichend von einer obigen Empfehlung öffentliche Eigenschaften mit **set**-Methode für die Koordinaten und ist folglich veränderlich.
- Komplexe Zahlen[2]
  Bei der Struktur **Complex** im Namensraum **System.Numerics** hat sich Microsoft an die eigene Empfehlung gehalten, Strukturen als unveränderlich zu konzipieren. Hier besitzen z. B. die öffentlichen Eigenschaften **Real** und **Imaginary** für den Zugriff auf den Real- bzw. Imaginärteil einer komplexen Zahl nur eine **get**-Methode.

Empfehlungen aus Griffiths (2013, S. 97) für die Entscheidung zwischen einer Klasse und einer Struktur bei der Realisation eines neuen Datentyps:

- Wenn es für eine Instanz erforderlich ist, ihren Zustand zu ändern, so ist dies ein deutliches Indiz dafür, dass eine Klasse gegenüber einer Struktur bevorzugt werden sollte.
- Im Zweifel sollte man eine Klasse definieren.

Wie das Beispiel der außerordentlich wichtigen Klasse **String** zeigt (siehe Abschnitt 6.3.1), ist bei einem Typ mit unveränderlichen Instanzen noch lange nicht entschieden, dass eine Struktur definiert werden sollte.

---

[1] https://docs.microsoft.com/de-de/dotnet/standard/design-guidelines/struct

[2] Dieser mathematische Begriff meint Paare aus reellen Zahlen, für die spezielle Rechenregeln gelten. Wer nicht mathematisch vorbelastet ist, kann das Beispiel ignorieren.

6.1.1 Detailvergleich von Klassen und Strukturen

Um den Unterscheid zwischen der *Referenzsemantik* der Klassen und der *Wertsemantik* der Strukturen zu demonstrieren, definieren wir sowohl eine *Klasse* als auch eine *Struktur* zur Repräsentation von Punkten der reellen Zahlenebene ($\mathbb{R}^2$). Für die beiden Koordinaten eines Punkts werden Felder mit dem elementaren Datentyp **double** verwendet.

Eine Strukturdefinition unterscheidet sich von der gewohnten Klassendefinition auf den ersten Blick nur durch das neue Schlüsselwort **struct**, das an Stelle von **class** verwendet wird:

```
public struct Punkt {
   double x, y;

   public Punkt(double x_, double y_) {
      x = x_;
      y = y_;
   }

// Nutzlose bzw. gefährlich irreführende Methode:
   public static void Bewegen(Punkt p, double hor, double vert) {
      p.x += hor;
      p.y += vert;
   }

   public void Bewegen(double hor, double vert) {
      x += hor;
      y += vert;
   }

   public string Pos() {
      return "("+x+";"+y+")";
   }
}
```

Wir *ignorieren* die Empfehlung, Strukturtypen als unveränderlich zu konzipieren, und definieren zwei Methoden namens `Bewegen()`:

- Die statische Variante ist ungeschickt und kann wegen der Wertsemantik bei Strukturen die Position der im ersten Parameter benannten `Punkt`-Instanz *nicht* verändern. Um die Macke zu beseitigen, müsste man allerdings lediglich den ersten Parameter zum Referenzparameter machen (vgl. Abschnitt 5.3.1.3.2).
- Die Instanzmethode `Bewegen()` erfüllt ihren Zweck.

Wie die Objekte von Klassen werden auch Strukturinstanzen per **new**-Operator unter Verwendung eines Konstruktors erstellt. Beim folgenden Einsatz der `Punkt`-Struktur wird die Variable `p1` über den expliziten Konstruktor mit den Koordinaten (1, 2) initialisiert. Der Punkt `p2` erhält (vom *nicht* verloren gegangenen!) Standardkonstruktor eine Initialisierung mit den Koordinaten (0, 0). Der Punkt `p3` erhält eine *Kopie* von `p1` (mit allen Instanzvariablen).

Quellcode	Ausgabe (mit `Punkt` als Struktur)

`using System;`
`class PunktDemo {`
`  static void Main() {`
`  var p1 = new Punkt(1, 2);`
`  var p2 = new Punkt();`
`  var p3 = p1;`
`  Punkt.Bewegen(p2, 1, 1); // nutzlos`
`  p1.Bewegen(1, 0); // klappt, aber ohne Effekt auf p3`
`  Console.WriteLine("p1 = " + p1.Pos()+`
`  "\np2 = " + p2.Pos() +`
`  "\np3 = " + p3.Pos());`
`  }`
`}`
	`p1 = (2;2)`
	`p2 = (0;0)`
	`p3 = (1;2)`

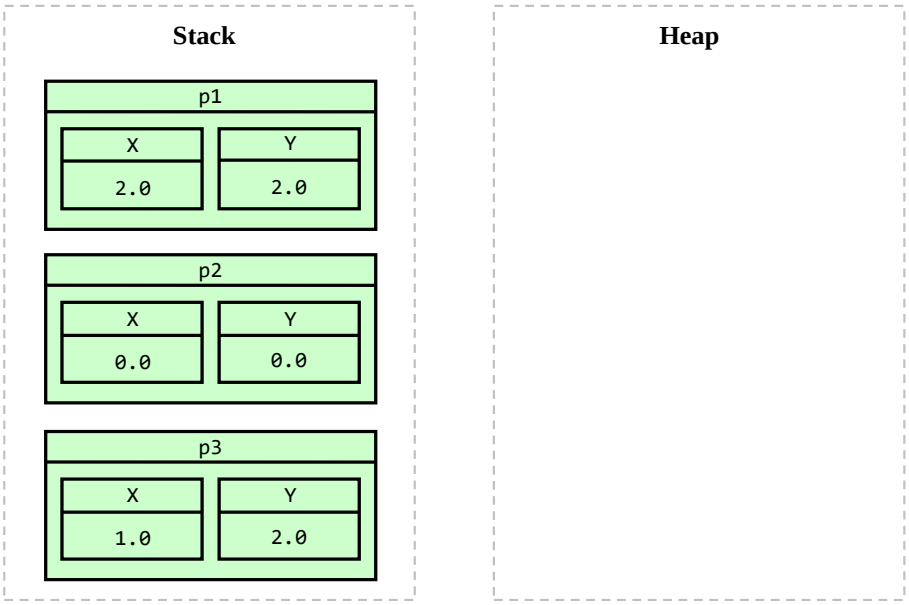
Die Wertsemantik von Strukturen ist im Verhalten des Programms vor allem an zwei Stellen zu beobachten:

- Mit der statischen `Bewegen()` - Überladung kann die Position des ersten Aktualparameters *nicht* verändert werden. Die fehlerhaft implementierte Methode ändert lediglich die per Wertparameter erhaltene Kopie.
- Die Änderung von `p1` bleibt ohne Effekt auf `p3`,weil durch die Anweisung
  `var p3 = p1;`
  nicht die Adresse von `p1` in `p3` kopiert wird, sondern eine vollständige Strukturinstanz. Eine spätere Veränderung des Originals hat natürlich keinen Effekt auf die Kopie.

Nach der Anweisung

```
p1.Bewegen(1, 0);
```

haben wir die folgende Situation im Speicher des Programms:



Aus der `Punkt`-Struktur wird durch wenige Quellcode-Änderungen eine *Klasse*:

```
public class Punkt {
   double x, y;

   public Punkt(double x_, double y_) {
      x = x_;
      y = y_;
   }

   public Punkt() { }

   public static void Bewegen(Punkt p, int hor, int vert) {
      p.x = p.x + hor;
      p.y = p.y + vert;
   }

   public void Bewegen(int hor, int vert) {
      x = x + hor;
      y = y + vert;
   }

   public string Pos() {
      return "("+x+";"+y+")";
   }
}
```

Weil bei Klassen (im Unterschied zu Strukturen) der Standardkonstruktor verloren geht, sobald ein expliziter Konstruktor vorhanden ist (vgl. Abschnitt 5.4.3), hat die Klasse `Punkt` auch einen parameterlosen expliziten Konstruktor erhalten.

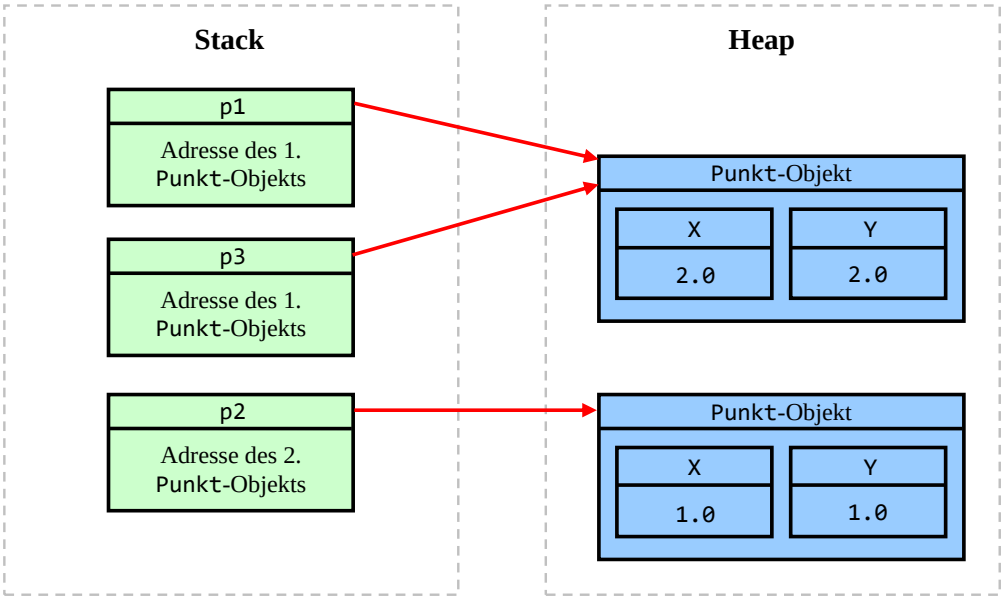
Das obige **Main()** - Programm muss beim Wechsel von der `Punkt`-Struktur zur `Punkt`-Klasse *nicht* geändert werden, verhält sich aber anders:

Quellcode	Ausgabe (mit `Punkt` als Klasse)

`using System;`
`class PunktDemo {`
`  static void Main() {`
`  var p1 = new Punkt(1, 2);`
`  var p2 = new Punkt();`
`  var p3 = p1;`
`  Punkt.Bewegen(p2, 1, 1);`
`  p1.Bewegen(1, 0);`
`  Console.WriteLine("p1 = " + p1.Pos()+`
`  "\np2 = " + p2.Pos() +`
`  "\np3 = " + p3.Pos());`
`  }`
`}`
	`p1 = (2;2)`
	`p2 = (1;1)`
	`p3 = (2;2)`

Die in der statischen Variante der Methode `Bewegen()` vorgenommene Ortsveränderung wirkt sich auf das *Objekt* mit der im ersten Aktualparameter übergebenen Adresse aus.

`p1`, `p2` und `p3` sind nun lokale Referenzvariablen, die auf insgesamt *zwei* Objekte zeigen:



Das erste `Punkt`-Objekt kann über die Referenzvariablen `p1` *und* `p3` angesprochen (und z. B. verändert) werden.

Während ein Objekt *eigenständig* auf dem *Heap* existiert (persistiert) und verfügbar ist, solange irgendwo im Programm eine Referenz (Kommunikationsmöglichkeit) vorhanden ist, kann eine Strukturinstanz nur als Objekt-Member das Ende der erzeugenden Methode überstehen (so wie auch die Variablen mit elementarem Typ). Eine *lokale* Variable mit Strukturtyp wird auf dem *Stack* abgelegt und beim Verlassen der Methode gelöscht. Bei entsprechender Methodendefinition kann dem Aufrufer via Rückgabewert eine *Kopie* der Strukturinstanz übergeben werden.

Auch bei Verwendung einer Eigenschaft oder eines Indexers findet ein verkappter Methodenaufruf statt, und hier können sich Programmierfehler durch veränderliche Strukturen besonders leicht einschleichen. Im folgenden Beispielprogramm besitzt die Klasse `PunktDemo` eine (automatisch implementierte und initialisierte) Eigenschaft namens `Position` vom Typ `Punkt`:

```
using System;
class PunktDemo {
   public Punkt Position {get; set;} = new Punkt();
   static void Main() {
      var pd = new PunktDemo();
      pd.Position.Bewegen(1, 2);
      Console.WriteLine("Neue Position: " + pd.Position.Pos());
   }
}
```

Ist `Punkt` eine Klasse, dann liefert der Lesezugriff auf die Eigenschaft `Position` des `PunktDemo`-Objekts die *Adresse* des `Punkt`-Objekts hinter der Eigenschaft. Dieses Objekt wird durch die Methode `Bewegen()` erfolgreich zu einer Ortsveränderung veranlasst, und es resultiert die Ausgabe:

```
Neue Position: (1;2)
```

Ist `Punkt` hingegen eine Struktur, wird keine Positionsveränderung gemeldet:

```
Neue Position: (0;0)
```

Die **get**-Methode der Eigenschaft `Position` liefert eine *Kopie* der `Punkt`-Struktur hinter der Eigenschaft, und diese Kopie wird anschließend per `Bewegen()` - Aufruf verschoben. Das Backing Field der Eigenschaft (vgl. Abschnitt 5.5.2) wird dabei *nicht* verändert. Um solche Missverständnisse mit beliebig gravierenden Konsequenzen auszuschließen, sollten Strukturen als unveränderlich konzipiert werden.

Neben der eigenständigen Speicherpersistenz fehlt den Strukturen auch die Vererbungstechnik, die wir erst in einem späteren Kapitel gründlich behandeln werden. *Jede* Struktur stammt direkt von der Klasse **ValueType** im Namensraum **System** ab, die wiederum direkt von der .NET - Urahnklasse **Object** erbt (siehe Abbildung im Abschnitt 6.1.2). Somit können keine Strukturhierarchien entstehen.

Die Struktur- und die Klassendefinition haben viele Gemeinsamkeiten, doch gibt es auch Unterschiede:

- Wie bei Klassen wird die Verfügbarkeit eines Strukturtyps über Modifikatoren geregelt (Voreinstellung: **internal**, Alternative: **public**, vgl. Abschnitt 5.11).
- Das Schlüsselwort **class** wird durch das Schlüsselwort **struct** ersetzt.
- Seit C# 7.2 kann man in einer Strukturdefinition dem Compiler durch den Modifikator **readonly** mitteilen, dass die oben empfohlene Unveränderlichkeit der Strukturinstanzen realisiert werden soll, z. B.:

```
readonly public struct Strecke {
    public int Li { get; }
    public int Re { get; }

    public Strecke(int li, int re) {
        Li = li; Re = re;
    }
}
```

  Daraufhin besteht der Compiler darauf, dass ...
  - alle Felder den Modifikator **readonly** erhalten
  - alle Eigenschaften schreibgeschützt (get-only) sind.

  Instanzen einer **readonly**-Struktur sollten aus Performanzgründen als Aktualparameter mit dem **in**-Schlüsselwort gekennzeichnet werden (siehe Abschnitt 5.3.1.3.2.3), wenn sie größer sind als eine Speicheradresse.[1]
- Wie bei Klassen können auch bei Strukturen die Felder den Modifikator **readonly** erhalten, sodass nach der Initialisierung keine Wertveränderungen mehr möglich sind.
- Seit C# 8.0 kann bei Strukturen auch eine instanzbezogene Methode oder Eigenschaft den Modifikator **readonly** erhalten. Daraufhin stellt der Compiler sicher, dass die Methode bzw. Eigenschaft den Zustand der *agierenden Instanz* nicht ändert. Eine *andere* Instanz derselben Struktur ist aber *nicht* vor Veränderungen geschützt, z. B.:

```
struct Struktur {
    public int Temp;
    public readonly void RoMo(ref Struktur p) {
        Temp = 13; // verboten
        p.Temp = 13; // erlaubt
    }
}
```

- Bei der Deklaration von Instanzfeldern ist *keine explizite Initialisierung* erlaubt. Verboten ist also z. B.:

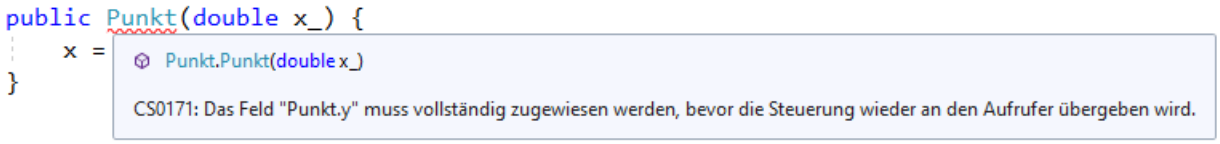
```
double delta = 1.0; // verboten!
```

  Felder von Strukturinstanzen werden jedoch wie die Felder von Klasseninstanzen (Objekten) per Voreinstellung mit der typspezifischen Null initialisiert.
- Es sind beliebig viele Konstruktorüberladungen erlaubt.
- Der (parameterfreie) Standardkonstruktor geht durch die Definition eines expliziten Konstruktors *nicht* verloren.

---

[1] Die Größe einer Speicheradresse ist aus der statischen Eigenschaft **IntPtr.Size** abzulesen.

- Bei einer Struktur darf kein expliziter parameterloser Konstruktor definiert werden, sodass man das Verbot einer Deklaration mit Initialisierung (siehe oben) auf Nicht-Nullwerte nicht über einen parameterlosen Konstruktor aushebeln kann. Folglich muss bei einer Struktur sichergestellt sein, dass die Nullinitialisierung zu einer regulären Instanz führt.
- Ein Struktur-Konstruktor muss *alle* Instanzvariablen initialisieren, z. B.:



  Von einem Klassen-Konstruktor wird das *nicht* verlangt.
- Bei Strukturen ist *keine Vererbung* möglich, was einige zusätzliche Regeln zur Folge hat (siehe ECMA 2017, S. 346).
- Die Methode **Equals()** basiert bei Strukturinstanzen auf einem Vergleich der *Inhalte*, nicht (wie bei Objekten) auf einem Vergleich der *Speicheradressen*.[1]
- Auch bei Strukturen ist eine Datenkapselung möglich. Bei der Deklaration bzw. Definition der Member kann der Zugriffsschutz über die Modifikatoren **private** (Voreinstellung), **public** und **internal** reguliert werden. Weil keine Vererbung unterstützt wird, ist der Modifikator **protected** verboten.
- Am Ende der Strukturdefinition darf optional ein Semikolon stehen.

In der Regel enthält eine Struktur nur Felder mit einem Werttyp (z. B. mit einem elementaren Datentyp). Felder mit Referenztyp sind ungewöhnlich, aber erlaubt, wobei die zugehörigen Objekte unveränderlich sein sollten, was z. B. bei **String**-Objekten der Fall ist (siehe Abschnitt 6.3.1).

Das folgende Programm vergleicht den Zeit- und Speicheraufwand für Objekte und Strukturinstanzen. Es wird jeweils in einer Methode ein Array (vgl. Abschnitt 6.2) mit 1.000.000 Punkten angelegt. Zur Ermittlung des vom Programm belegten Speichers wird die statische Methode **GetTotalMemory()** der Klasse **GC** verwendet:

public static long GetTotalMemory(bool *forceFullCollection***)**

Mit dem Parameter *forceFullCollection* legt man fest, ob die Methode vor der Rückkehr einen Einsatz des Garbage Collectors abwarten soll (Aktualparameterwert **true**). Das Beispielprogramm aktiviert in den **GetTotalMemory()** - Aufrufen *nach* der Rückkehr der Punkterstellungs-Methoden den Müllsammler, sodass die Array-Objekte abgeräumt werden. Während alle Strukturinstanzen zusammen mit dem Array untergehen, müssen die Objekte einzeln gelöscht werden.

```
using System;

class Aufwandsvergleich {
  const int anzahl = 1_000_000;

  static void MakeStructArray() {
    Console.WriteLine("{0,-50}{1,20}","MB vor Struct-Array-Erstellung:",
      GC.GetTotalMemory(false) / 1048576);
    long st = DateTime.Now.Ticks;
    SPunkt[] arc = new SPunkt[anzahl];
    for (int i = 0; i < anzahl; i++)
      arc[i] = new SPunkt(i, i);
    st = DateTime.Now.Ticks - st;
    Console.WriteLine("{0,-50}{1,20}", "Benötigte Zeit für den Struktur-Array:",
                      (st / 1.0e4) + " Millisek.");
    Console.WriteLine("{0,-50}{1,20}", "MB nach Struct-Array-Erst., noch in Methode:",
                       GC.GetTotalMemory(false) / 1048576);
  }
```

[1] Siehe Richter 2012 (Kapitel 5) für eine Erläuterung der Methode **Equals()**.

```
  static void MakeClassArray() {
    Console.WriteLine("\n{0,-50}{1,20}", "MB vor Class-Array-Erstellung:",
                      GC.GetTotalMemory(false) / 1048576);
    long ct = DateTime.Now.Ticks;
    CPunkt[] arc = new CPunkt[anzahl];
    for (int i = 0; i < anzahl; i++)
      arc[i] = new CPunkt(i, i);
    ct = DateTime.Now.Ticks - ct;
    Console.WriteLine("{0,-50}{1,20}", "Benötigte Zeit für den Klassen-Array:",
                      (ct / 1.0e4) + " Millisek.");
    Console.WriteLine("{0,-50}{1,20}", "MB nach Class-Array-Erstellung, noch in Methode:",
                      GC.GetTotalMemory(true) / 1048576);
  }

  static void Main() {
    long wzeit = DateTime.Now.Ticks; // Verhindert einen verzerrten Wert der 1. Messung
    MakeStructArray();
    Console.WriteLine("{0,-50}{1,20}", "MB nach Struct-Array-Erstellung, nach Rückkehr:",
                      GC.GetTotalMemory(true) / 1048576);
    MakeClassArray();
    Console.WriteLine("{0,-50}{1,20}", "MB nach Class-Array-Erstellung, nach Rückkehr:",
                      GC.GetTotalMemory(true) / 1048576);
  }
}
```

In der Release-Konfiguration ist der Zeitaufwand für die Objekte erheblich höher als für die Strukturinstanzen, wobei der Aufwand für das Abräumen durch den Garbage Collector noch nicht einmal berücksichtigt ist:[1]

```
MB vor Struct-Array-Erstellung:                              0
Benötigte Zeit für den Struktur-Array:             17,999 Millisek.
MB nach Struct-Array-Erst., noch in Methode:                15
MB nach Struct-Array-Erstellung, nach Rückkehr:              0

MB vor Class-Array-Erstellung:                               0
Benötigte Zeit für den Klassen-Array:            142,9908 Millisek.
MB nach Class-Array-Erstellung, noch in Methode:            38
MB nach Class-Array-Erstellung, nach Rückkehr:               0
```

Der höhere Speicherbedarf für die Objekte überrascht nicht, denn ein Array mit 1.000.000 Strukturinstanzen ergibt auf dem Heap ein einziges Objekt mit einem zusammenhängenden Speicherbereich. Ein Array mit 1.000.000 Objekten ergibt hingegen 1000001 Objekte auf dem Heap. Der Array enthält nur die Adressen der 1.000.000 separaten `Punkt`-Objekte.

Wir verzichten darauf, von der Klasse `Bruch` eine Strukturalternative zu erstellen, denn:

- Weil bei der Deklaration von Strukturfeldern keine Initialisierung erlaubt ist, und außerdem der parameterfreie Konstruktor nicht verändert werden darf, könnte bei parameterfrei konstruierten `Bruch`-Strukturinstanzen nicht verhindert werden, dass der Nenner den Wert 0 erhält.
- Um die dringende Empfehlung der Unveränderlichkeit von Strukturinstanzen umzusetzen, müsste das Design der Klasse `Bruch` erheblich geändert werden. Es wäre z. B. nicht zulässig/akzeptabel, dass man eine `Bruch`-Strukturinstanz auffordern kann, sich zu kürzen.

---

[1] Der Vorabzugriff auf die **Ticks**-Eigenschaft in der folgenden, scheinbar überflüssigen Anweisung verhindert, dass die erste Laufzeitmessung einen überhöhten Wert liefert:

```
long wzeit = DateTime.Now.Ticks;
```

Das Problem einer verzerrten ersten Messung besteht bei der Klasse **Stopwatch** aus dem Namensraum **System.Diagnostics** (vgl. Abschnitt 4.7.4) übrigens *nicht*.

- Es ist nicht damit zu rechnen, dass `Bruch`-Objekte derart massenhaft auftreten, dass ein Spareffekt durch die Verwendung von Strukturen spürbar wird.
- Wir würden die Möglichkeit verlieren, per Vererbung aus dem Typ `Bruch` spezialisierte Varianten abzuleiten.

6.1.2 Strukturen im allgemeinen Typsystem der .NET - Plattform

Aus den bisherigen Ausführungen zu folgern, dass Strukturen wohl eher exotisch und nur für leistungskritische Anwendungen interessant seien, wäre schon deshalb grundverkehrt, weil es sich bei allen elementaren Datentypen um Strukturen handelt. Die früher als Typbezeichner eingeführten C# - Schlüsselwörter sind lediglich Aliasnamen für Strukturen aus dem BCL-Namensraum **System**:

Aliasname	Struktur
sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32

Aliasname	Struktur
long	System.Int64
ulong	System.UInt64
char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

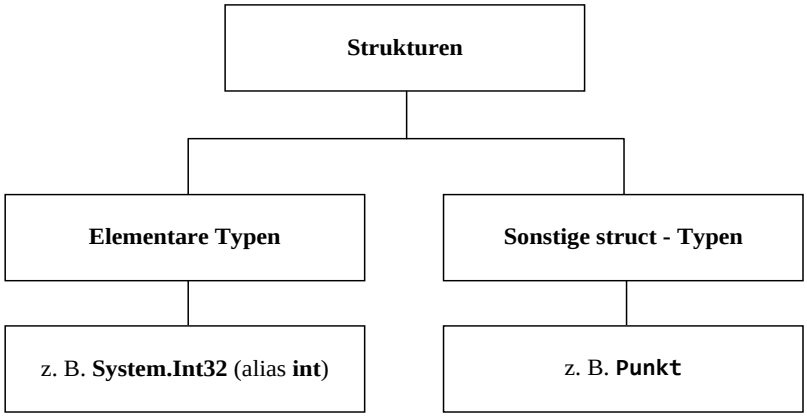
Nun wird z. B. klar, warum die **Convert**-Methode zur Wandlung von Zeichenfolgen in **int**-Werte den Namen **ToInt32()** trägt.

Für die elementaren Datentypen bietet C# im Vergleich zu den sonstigen Strukturtypen neben den reservierten Wörtern als Typaliasnamen noch weitere Vorzugsbehandlungen, z. B. die Erzeugung von Werten über Literale (vgl. ECMA 2017, S. 63f). Wie z. B. die Definition des BCL-Typs **Int32** zeigt,[1]

```
public struct Int32 : IComparable, IFormattable, IConvertible { ... }
```

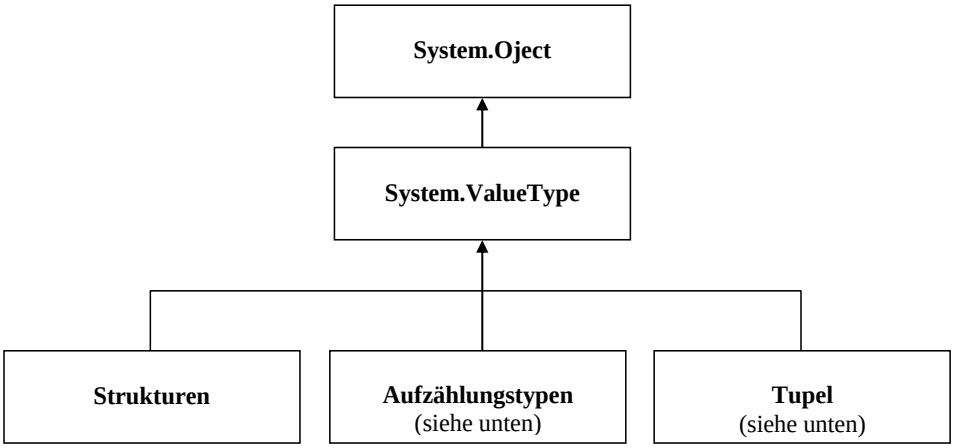
sind die elementaren Datentypen ansonsten reguläre Strukturen.

Bei den Strukturen im CTS (Common Type System) können wir also zwischen den elementaren Datentypen und sonstigen Strukturtypen differenzieren:



Jeder Strukturtyp stammt von der Klasse (!) **ValueType** im Namensraum **System** ab, die wiederum direkt von der .NET - Urahnklasse **Object** erbt:

[1] Hinter dem Doppelpunkt im Definitionskopf sind *Schnittstellen* aufgelistet, denen bald ein Kapitel gewidmet wird.



Im folgenden Beispielprogramm wird beim Ganzzahlliteral 13 (vom Strukturtyp **Int32**) über die von **System.Object** geerbte Methode **GetType()** erfolgreich der Laufzeit-Datentyp erfragt:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  Console.WriteLine(13.GetType());`
`  }`
`}`
	`System.Int32`

Trotz des obigen Stammbaums *ist* eine Strukturinstanz kein Objekt, aber aufgrund einer im nächsten Abschnitt zu beschreibenden Raffinesse der .NET - Plattform kann eine Strukturinstanz jederzeit so behandelt werden, als *wäre* sie ein Objekt.

6.1.3 Boxing und Unboxing

Wie bald im Kapitel 7 über die Vererbung noch näher erläutert wird, kann eine Variable vom Typ **Object** Referenzen auf Objekte aus beliebigen Klassen aufnehmen, weil alle Klassen direkt oder indirekt von **Object** abstammen. Damit das *Common* Type System seinem Namen gerecht wird, muss diese Zuweisungskompatibilität für *beliebige Typen* gelten, also auch für Werttypen. Wie Sie wissen, kann eine Referenzvariable vom Typ **Object** nur die Adresse eines Heap-Objekts aufnehmen. Was soll nun aber geschehen, wenn einer solchen Referenzvariablen z. B. ein Wert vom elementaren Typ **int** zugewiesen wird?

Eine analoge Situation liegt vor, wenn bei einem Methodenaufruf eine Strukturinstanz als Aktualparameter auftritt, obwohl gemäß Methodendefinition eine Objektreferenz (die Adresse eines Heap-Objekts) benötigt wird. Wir haben uns längst daran gewöhnt, dass der Compiler Werte von beliebigem Typ als **Object**-Instanzen akzeptiert. Z. B. werden im folgenden Programm

```
using System;
class Prog {
  static void Main() {
    int i = 7, j = 3;
    Console.WriteLine("{0} % {1} = {2}", i, j, i % j);
  }
}
```

der Methode **WriteLine()** Aktualparameter vom Werttyp **int** an Positionen mit dem Formalparametertyp **Object** übergeben:

public static void WriteLine(String *format***, params Object[]** *arg***)**

Diese Zuweisungskompatibilität wird möglich durch ein als **Boxing** bezeichnetes Prinzip: Es sorgt dafür, dass ein Wert bei Bedarf automatisch in ein Objekt einer passenden Hilfs- bzw. Hüllenklasse

verpackt wird. Somit existiert ein Heap-Objekt und der **Console**-Methode **WriteLine()** kann die benötigte Adresse übergeben werden.

Im folgenden Programm mit einer Boxing-Trockenübung wird die **int**-Variable `i` einer Referenzvariablen vom Typ **object** (Aliasname für **System.Object**) zugewiesen und dabei automatisch in ein Objekt der zugehörigen Hilfsklasse gesteckt:[1]

```
using System;
class Prog {
   static void Main() {
      int i = 4711;
      object iBox = i;
      int j = (int) iBox;
   }
}
```

Wie die letzte Anweisung im Beispielprogramm zeigt, benötigt der Compiler beim **Unboxing**, also beim Auspacken eines Wertes aus einem Hüllenobjekt, eine explizite Typumwandlung mit Angabe des Zieltyps:

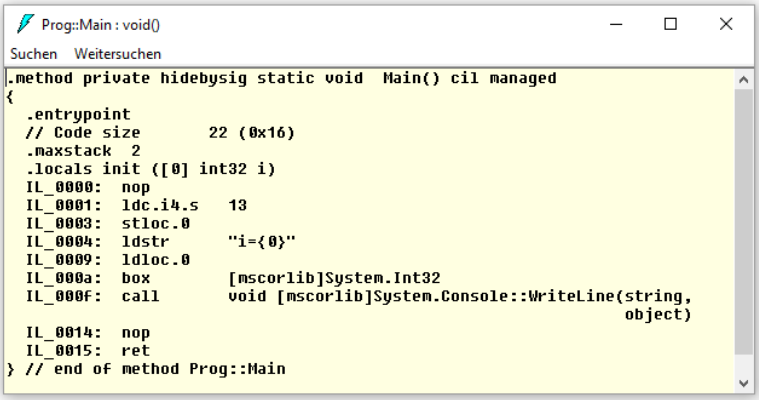
```
int j = (int)iBox;
```

Weil das Boxing durch die damit verbundene Objektkreation relativ aufwändig ist, sollte man die Verwendung dieser Technik auf das notwendige Maß beschränken.

Wer sich vergewissern möchte, dass beim Methodenaufruf

```
Console.WriteLine("i={0}", i);
```

mit der **int**-Variablen `i` als zweitem Aktualparameter tatsächlich eine Boxing-Operation stattfindet, kann mit dem Hilfsprogramm **ILDasm** einen Blick auf den IL-Code werfen. Hier wird die Objektkreation zur Verpackung eines **System.Int32** - Werts mit dem OpCode **box** veranlasst:



Anschließend wird die Methode **Console.WriteLine()** mit Aktualparametern vom Typ **String** bzw. **Object** aufgerufen.

Auch zur Ausführung des **GetType()** - Aufrufs in der folgenden Anweisung

```
Console.WriteLine(13.GetType());
```

wird per **box**-OpCode ein Objekt erzeugt. Wie im Abschnitt 6.1.2 zu sehen war, liefert **GetType()** aber nicht den Typ des Hüllenobjekts, sondern den Typ der verpackten Strukturinstanz.

---

[1] Neben dem *impliziten* Boxing, das auch als **Autoboxing** bezeichnet wird, ist auch ein *explizites* Boxing möglich, aber nie erforderlich, z. B.

```
int i = 4711;
object iBox = (object) i;
```

Die Frage, ob sich in C# ein per Boxing verpackter Wert ändert lässt, ist „überwiegend“ zu verneinen. Wie Richter (2006, S. 165f) zeigt, geht es mit Hilfe einer implementierten Schnittstelle aber doch. Das folgende Programm kann erst nach der Lektüre von Kapitel 9 vollständig verstanden werden:

```
using System;

interface IChangeBoxedPoint {
    void Change(int x, int y);
}

struct Point : IChangeBoxedPoint {
    int m_x, m_y;
    public void Change(int x, int y) {
        m_x = x; m_y = y;
    }
    public override String ToString() {
        return String.Format("({0}, {1})", m_x.ToString(), m_y.ToString());
    }
}

class Prog {
    static void Main() {
        Point p = new Point();
        Object o = p;
        Console.WriteLine("Boxed Point: " + o);
        ((IChangeBoxedPoint)o).Change(5, 5);
        Console.WriteLine("Boxed Point moved by interface method: " + o);
    }
}
```

Seine Ausgabe zeigt, dass die Änderung eines verpackten Werts per Interface-Referenz gelingt:

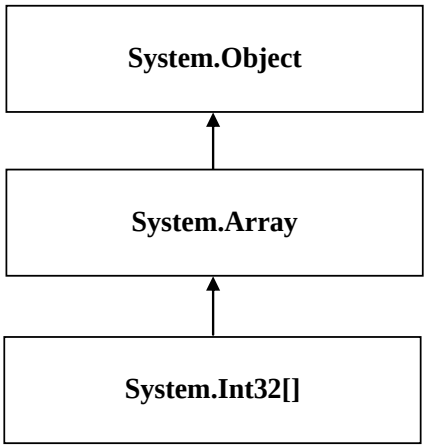
```
Boxed Point: (0, 0)
Boxed Point moved by interface method: (5, 5)
```

6.2 Arrays

Ein Array ist ein Objekt, das eine feste Anzahl von Elementen desselben Datentyps als Instanzvariablen enthält, die in einem zusammenhängenden Speicherbereich hintereinander abgelegt werden. Man kann den kompletten Array ansprechen (z. B. als Aktualparameter an eine Methode übergeben), oder auf einzelne Elemente über einen durch eckige Klammern begrenzten Index zugreifen.

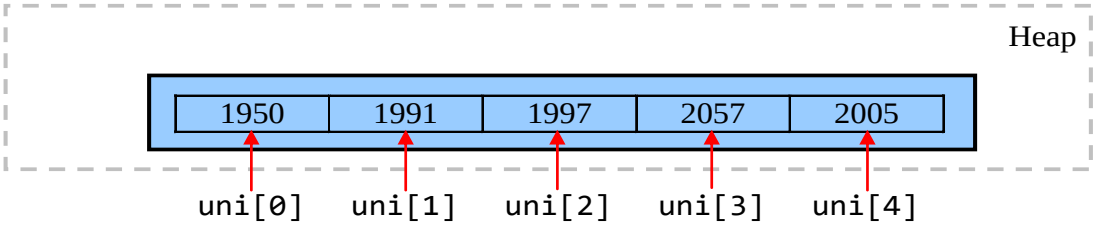
Arrays werden in vielen Programmiersprachen auch *Felder* genannt. In C# bezeichnet man jedoch recht einheitlich die (Instanz-)variablen einer Klasse oder Struktur als *Felder*, sodass der Name hier nicht mehr zur Verfügung steht.

Wir beschäftigen uns erst *jetzt* mit den zur Grundausstattung praktisch jeder Programmiersprache gehörenden Arrays, weil diese Datentypen in C# als *Klassen* realisiert sind und folglich zunächst entsprechende Grundlagen zu erarbeiten waren. Obwohl wir die wichtige Vererbungsbeziehung zwischen Klassen noch nicht offiziell behandelt haben, können Sie vermutlich schon den Hinweis verdauen, dass alle Array-Klassen von der Basisklasse **Array** im Namensraum **System** abstammen, z. B. die Klasse der eindimensionalen Arrays mit Elementen vom Strukturtyp **Int32** (alias **int**):



Weil alle Arrays von der Klasse **System.Array** abstammen, implementieren sie wichtige Schnittstellen (vgl. Kapitel 9), z. B. **ICollection**, **IEnumerable**, **IList**. Dieser Satz wird sich später als wichtig erweisen, kann und muss aber jetzt noch nicht verstanden werden.

Hier ist als konkretes Objekt aus der Klasse **int[]** ein Array namens `uni` mit fünf Elementen zu sehen:



Neben den Array-Elementen enthält das Objekt noch Verwaltungsdaten (z. B. die per **Length**-Eigenschaft ansprechbare Anzahl seiner Elemente).

Beim Zugriff auf ein *einzelnes Element* gibt man nach dem Array-Namen den durch eckige Klammern begrenzten Index an, wobei die Nummerierung mit 0 beginnt und bei *n* Elementen folglich mit *n* - 1 endet. Technisch gesehen liegt ein Array-Zugriffsausdruck mit dem Operator [] vor.

Im Vergleich zur Verwendung einer entsprechenden Anzahl von Einzelvariablen ermöglichen Arrays eine gravierende Vereinfachung der Programmierung:

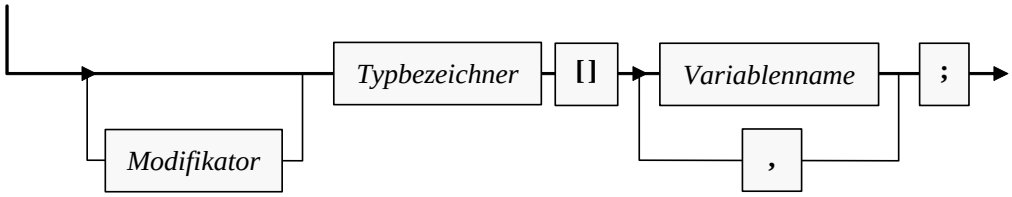
- Weil der Index auch durch einen *Ausdruck* (z. B. durch eine Variable) geliefert werden kann, sind Arrays im Zusammenhang mit den Wiederholungsanweisungen äußerst praktisch.
- Man kann die *gemeinsame* Verarbeitung *aller* Elemente (z. B. bei der Ausgabe in eine Datei) per Methodenaufruf mit Array-Aktualparameter veranlassen.
- Viele Algorithmen arbeiten mit Vektoren und Matrizen. Zur Modellierung dieser mathematischen Objekte sind Arrays unverzichtbar.

Wir befassen uns zunächst mit *eindimensionalen* Arrays, behandeln später aber auch den mehrdimensionalen Fall.

6.2.1 Array-Referenzvariablen deklarieren

Im Vergleich zu der bisher bekannten Variablendeklaration (ohne Initialisierung) ist bei Array-Variablen hinter dem Typbezeichner zusätzlich ein Paar eckiger Klammern anzugeben:

Deklaration einer Array-Variablen



Welche Modifikatoren zulässig bzw. erforderlich sind, hängt davon, ob die Variable zu einer Methode, zu einer Klasse oder zu einer Instanz gehört. Die Array-Variable `uni` aus dem einleitend beschriebenen Beispiel gehört zu einer Methode (siehe unten) und ist folgendermaßen zu deklarieren:

```
int[] uni;
```

Bei der Deklaration entsteht nur eine Referenzvariable, jedoch noch kein Array-Objekt. Daher ist auch keine Array-Größe (Anzahl der Elemente) anzugeben.

Einer Array-Referenzvariablen kann als Wert die Adresse eines Arrays mit Elementen vom vereinbarten Typ oder das Referenzliteral **null** (Zeiger auf nichts) zugewiesen werden.

6.2.2 Array-Objekte erzeugen

Mit Hilfe des **new**-Operators erzeugt man ein Array-Objekt mit einem bestimmten Elementtyp und einer bestimmten Größe auf dem Heap. In der folgenden Anweisung entsteht ein Array mit (`max+1`) **int**-Elementen, und seine Adresse landet in der Referenzvariablen `uni`:

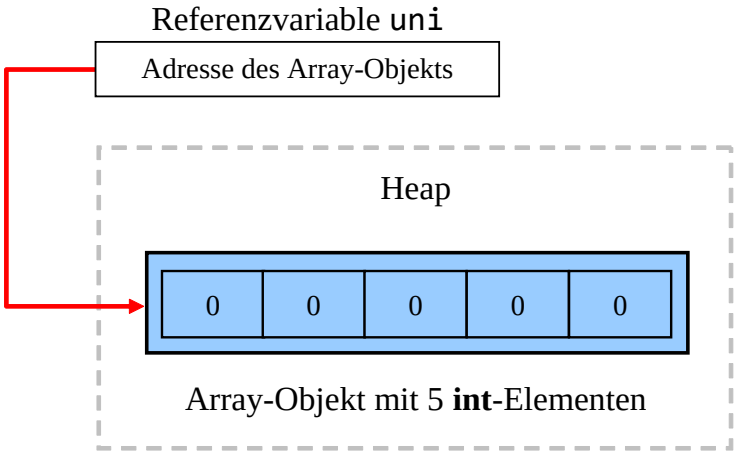
```
uni = new int[max+1];
```

Im **new**-Operanden *muss* hinter dem Datentyp zwischen eckigen Klammern die Anzahl der Elemente festgelegt werden, wobei ein beliebiger Ausdruck mit ganzzahligem Wert (≥ 0) erlaubt ist. Man kann also die Länge eines Arrays *zur Laufzeit* festlegen, z. B. in Abhängigkeit von einer Benutzereingabe.

Die Deklaration einer Array-Referenzvariablen *und* die Erstellung des Array-Objekts lassen sich natürlich auch in *einer* Anweisung erledigen, z. B.:

```
int[] uni = new int[5];
```

Mit der Verweisvariablen `uni` und dem referenzierten Array-Objekt auf dem Heap haben wir insgesamt die folgende Situation im Speicher:



Weil es sich bei den Array-Elementen um Instanzvariablen eines *Objekts* handelt, erfolgt eine automatische Null-Initialisierung nach den Regeln von Abschnitt 5.2.3. Die **int**-Elemente im Beispiel erhalten folglich den Startwert 0.

Als Objekt wird ein Array vom Garbage Collector entsorgt, wenn keine Referenz mehr vorliegt (vgl. Abschnitt 5.4.4). Um eine Referenzvariable aktiv von einem Array-Objekt zu „entkoppeln“, kann man ihr z. B. das Referenzliteral **null** oder aber ein alternatives Referenzziel zuweisen. Es ist ohne weiteres möglich, dass mehrere Referenzvariablen auf dasselbe Array-Objekt zeigen, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  int[] x = new int[3], y;`
`  x[0] = 1; x[1] = 2; x[2] = 3;`
`  y = x; //y zeigt nun auf das selbe Array-Objekt wie x`
`  y[0] = 99;`
`  Console.WriteLine(x[0]);`
`  }`
`}`
	`99`

Nachdem ein Array-Objekt erstellt worden ist, lässt sich seine Länge nicht mehr ändern. Seit der .NET - Version 2.0 erlaubt die statische Methode **Resize()** der Klasse **System.Array** scheinbar eine nachträgliche Längenkorrektur, z. B.:

```
Array.Resize(ref uni, 2 * max + 1);
```

Allerdings muss die Methode ein *neues* Objekt erzeugen und die Elemente des alten Arrays umkopieren, was einen erheblichen Aufwand bedeuten kann. Solche Aktionen werden bei der im Abschnitt 6.2.12 beschriebenen Kollektionsklasse **ArrayList** mit dynamischer Größenanpassung bei Bedarf im Hintergrund automatisch ausgeführt.

Weil die Methode **Resize()** einen neuen Array anlegen und dessen Adresse an den Aufrufer übergeben muss, hat der Parameter zum Array, der „vergrößert“ werden soll, den Modifikator **ref** erhalten.

6.2.3 Arrays benutzen

Der Zugriff auf die Elemente eines Array-Objektes geschieht über eine zugehörige Referenzvariable, an deren Namen zwischen eckigen Klammern ein passender Index angehängt wird. Als Index ist ein beliebiger Ausdruck mit einem nicht-negativen ganzzahligem Wert erlaubt, wobei natürlich die Feldgrenzen zu beachten sind. In der folgenden **for**-Schleife wird pro Durchgang ein zufällig gewähltes Element des **int**-Arrays inkrementiert,

```
for (int i = 0; i < drl; i++)
  uni[zzg.Next(5)]++;
```

auf den die Referenzvariable `uni` aufgrund der im Abschnitt 6.2.2 beschriebenen Deklaration und Initialisierung zeigt:

```
int[] uni = new int[5];
```

Den Indexwert liefert die **Random**-Methode **Next()** mit Rückgabetyp **int** (siehe unten). Die **for**-Anweisung stammt aus einer Methode, die im Abschnitt 6.2.5 vorgestellt wird. Dort sind die Variablen `i` und `drl` aus der **for**-Schleifensteuerung lokal definiert.

Wie in vielen anderen Programmiersprachen hat auch in C# das erste von *n* Array-Elementen die Nummer 0 und folglich das letzte die Nummer *n* - 1. Damit existiert z. B. nach der Anweisung

```
int[] uni = new int[5];
```

kein Element `uni[5]`. Ein Zugriffsversuch führt zum Laufzeitfehler vom Typ **IndexOutOfRangeException**, z. B.:

```
Unbehandelte Ausnahme: System.IndexOutOfRangeException: Der Index war
außerhalb des Arraybereichs.
   at UniRand.Main() in ...\BspUeb\Arrays\UniRand\UniRand.cs:line 7
```

Wenn das verantwortliche Programm einen solchen Ausnahmefehler nicht behandelt (siehe Kapitel 13), dann wird es vom Laufzeitsystem beendet. Man kann sich in C# generell darauf verlassen, dass jede Überschreitung von Feldgrenzen verhindert wird, sodass es nicht zur Verletzung anderer Speicherbereiche und den entsprechenden Folgen (Absturz mit Speicherschutzverletzung, unerklärliches Programmverhalten) kommt.

Die (z. B. durch eine Benutzerentscheidung zur Laufzeit festgelegte) Länge eines Array-Objekts lässt sich über seine (get-only -) Eigenschaft **Length** (vom Typ **int)** jederzeit ermitteln, z. B.:

Quellcode	Eingaben (**fett**) und Ausgaben

```
using System;
class Prog {
  static void Main() {
    Console.Write("Länge des Vektors: ");
    int[] wecktor =
      new int[Convert.ToInt32(Console.ReadLine())];

    Console.WriteLine();
    for (int i = 0; i < wecktor.Length; i++) {
      Console.Write("Wert von Element " + i + ": ");
      wecktor[i] = Convert.ToInt32(Console.ReadLine());
    }

    Console.WriteLine();
    for (int i = 0; i < wecktor.Length; i++)
      Console.WriteLine(wecktor[i]);
  }
}
```

```
Länge des Vektors: 3

Wert von Element 0: 7
Wert von Element 1: 13
Wert von Element 2: 4711

7
13
4711
```

Auch beim Entwurf von *Methoden* mit Array-Parametern ist es von Vorteil, dass die Länge eines übergebenen Arrays ohne entsprechenden Zusatzparameter in der Methode bekannt ist.

6.2.4 Maximale Array-Größe

Die Frage nach der maximalen Array-Größe (hinsichtlich Speichervolumen und Indexwert) treibt Programmiereinsteiger sicher nicht um, könnte aber irgendwann bei einer konkreten Anwendung relevant werden. Dieser exotisch erscheinende Fall wird sogar schon bald eintreten (siehe Übungsaufgabe im Abschnitt 6.9). Alle relevanten Fakten sind in der .NET - Dokumentation zur Klasse **Array** zusammengestellt.[1] Durch den Datentyp **int** (maximaler Wert: 2.147.483.647, siehe Abschnitt 4.3.4) der **Array**-Eigenschaft **Length** ist eine obere Schranke für die Länge eines Arrays gesetzt. Ist die .NET-Voreinstellung von 2 GB für die maximale Objektgröße in Kraft, ist der **int**-Wertebereich der Eigenschaft **Length** allerdings *nicht* der limitierende Faktor für die Array-Länge. Über das Element **gcAllowVeryLargeObjects** der Anwendungskonfigurationsdatei **app.config** lässt sich die 2 GB - Grenze für die maximale Objektgröße aufheben, was seit .NET 4.5 möglich und nur für 64-Bit - Systeme relevant ist, z. B.:

[1] https://docs.microsoft.com/en-us/dotnet/api/system.array

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7.2"/>
  </startup>
  <runtime>
    <gcAllowVeryLargeObjects enabled="true" />
  </runtime>
</configuration>
```

Dann gelten folgende Größenbeschränkungen für Arrays:

- Der maximale Index in einer Dimension beträgt 0X7FEFFFFF (0X7FFFFFC7, falls ein einzelnes Element nur 1 Byte belegt). Diese Zahlen liegen im **int**-Wertebereich.
- Ein mehrdimensionaler Array (siehe Abschnitt 6.2.10) darf insgesamt 4 Milliarden Elemente enthalten, wobei sich die Anzahl der Elemente über die Array-Eigenschaft **LongLength** mit Datentyp **long** ermitteln lässt.

Mit Hilfe unserer Entwicklungsumgebung kann z. B. auf folgende Weise eine Anwendungskonfigurationsdatei im Projektverzeichnis angelegt werden:

- Menübefehl: **Projekt > Hinzufügen > Neues Element**
- Im Fenster **Neues Element hinzufügen** wählen:
  Installiert > Visual C# > Allgemein > Anwendungskonfigurationsdatei
- Bei Bedarf den voreingestellten Namen ändern
- Klick auf **Hinzufügen**

6.2.5 Beispiel: Beurteilung des .NET - Pseudozufallszahlengenerators

Oben wurde am Beispiel des 5-elementigen **int**-Arrays `uni` demonstriert, dass die Array-Technik im Vergleich zur Verwendung einzelner Variablen den Aufwand bei der Deklaration und beim Zugriff deutlich verringert. Insbesondere beim Einsatz in einer Schleifenkonstruktion erweist sich die Ansprache der einzelnen Elemente über einen Index als überaus praktisch. Die oben zur Demonstration verwendeten Anweisungen lassen sich leicht zu einem Programm erweitern, das die Verteilungsqualität des .NET - **Pseudozufallszahlengenerators** überprüft. Dieser Generator produziert Folgen von Zahlen mit einem bestimmten Verteilungsverhalten. Obwohl eine Serie perfekt vom Initialisierungswert des Pseudozufallszahlengenerators abhängt, kann sie in der Regel echte Zufallszahlen ersetzen. Manchmal ist es sogar von Vorteil, eine Serie über einen festen Initialisierungswert reproduzieren zu können. Meist verwendet man aber variable Initialisierungen, z. B. abgeleitet aus einer Zeitangabe. Der Einfachheit halber spricht man oft von *Zufallszahlen* und lässt den Zusatz *Pseudo* weg.

Man kann übrigens mit moderner EDV-Technik unter Verwendung von physischen Prozessen auch *echte* Zufallszahlen produzieren, doch ist der Zeitaufwand im Vergleich zu Pseudozufallszahlen erheblich höher (siehe z. B. Lau 2009).

Nach der folgenden Anweisung zeigt die Referenzvariable `zzg` auf ein Objekt der Klasse **Random** aus dem Namensraum **System**, das als Pseudozufallszahlengenerator taugt:

```
Random zzg = new Random();
```

Durch die Verwendung des parameterfreien **Random**-Konstruktors entscheidet man sich für einen aus der Systemzeit abgeleiteten Initialisierungswert für den Pseudozufall.

Das angekündigte Programm zieht 10.000 Zufallszahlen aus der Menge {0, 1, ..., 4} und überprüft die empirische Verteilung dieser Stichprobe:

```
using System;
class UniRand {
   static void Main() {
      const int drl = 10_000;
      int[] uni = new int[5];
      Random zzg = new Random();
      for (int i = 0; i < drl; i++)
         uni[zzg.Next(5)]++;
      Console.WriteLine("Absolute Häufigkeiten:");
      for (int i = 0; i < 5; i++)
         Console.Write(uni[i] + " ");
      Console.WriteLine("\n\nRelative Häufigkeiten:");
      for (int i = 0; i < 5; i++)
         Console.Write((double)uni[i] / drl + " ");
   }
}
```

Die **Random**-Methode **Next()** liefert beim Aufruf mit dem Aktualparameterwert 5 als Rückgabe eine **int**-Zufallszahl aus der Menge {0, 1, 2, 3, 4}, wobei die möglichen Werte mit der gleichen Wahrscheinlichkeit 0,2 auftreten sollten. Im Programm dient der **Next()** - Rückgabewert als Array-Index dazu, ein zufällig gewähltes `uni`-Element zu inkrementieren. Wie das folgende Ergebnis-Beispiel zeigt, stellt sich die erwartete Gleichverteilung in guter Näherung ein:

```
Absolute Häufigkeiten:
1986 1983 1995 1995 2041

Relative Häufigkeiten:
0,1986 0,1983 0,1995 0,1995 0,2041
```

Ein χ^2-Signifikanztest mit der Gleichverteilung als Nullhypothese bestätigt durch eine Überschreitungswahrscheinlichkeit von 0,893 (weit oberhalb der kritischen Grenze 0,05), dass keine Zweifel an der Gleichverteilung bestehen:

uni

	Beobachtetes N	Erwartete Anzahl	Residuum
0	1986	2000,0	-14,0
1	1983	2000,0	-17,0
2	1995	2000,0	-5,0
3	1995	2000,0	-5,0
4	2041	2000,0	41,0
Gesamt	10000		

Statistik für Test

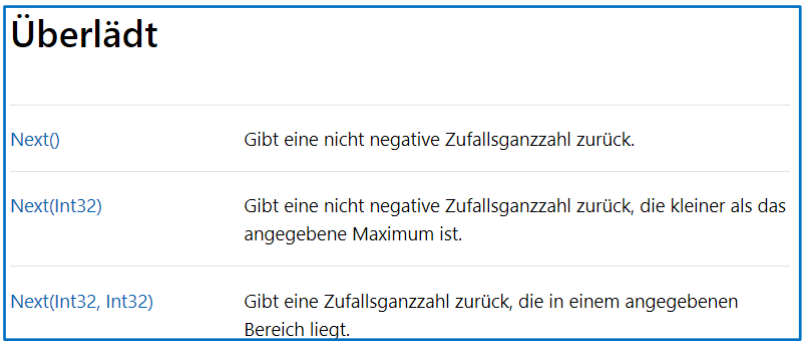
	uni
Chi-Quadrat	1,108[a]
df	4
Asymptotische Signifikanz	**,893**

a. Bei 0 Zellen (,0%) werden weniger als 5 Häufigkeiten erwartet. Die kleinste erwartete Zellenhäufigkeit ist 2000,0.

Über die im Beispielprogramm verwendete statische Methode **Next()** der Klasse **Random** liefert die BCL-Dokumentation ausführliche Informationen, die vom Visual Studio aus z. B. so zu erreichen sind:

- Einfügemarke auf den Methodennamen setzen
- Funktionstaste **F1** drücken

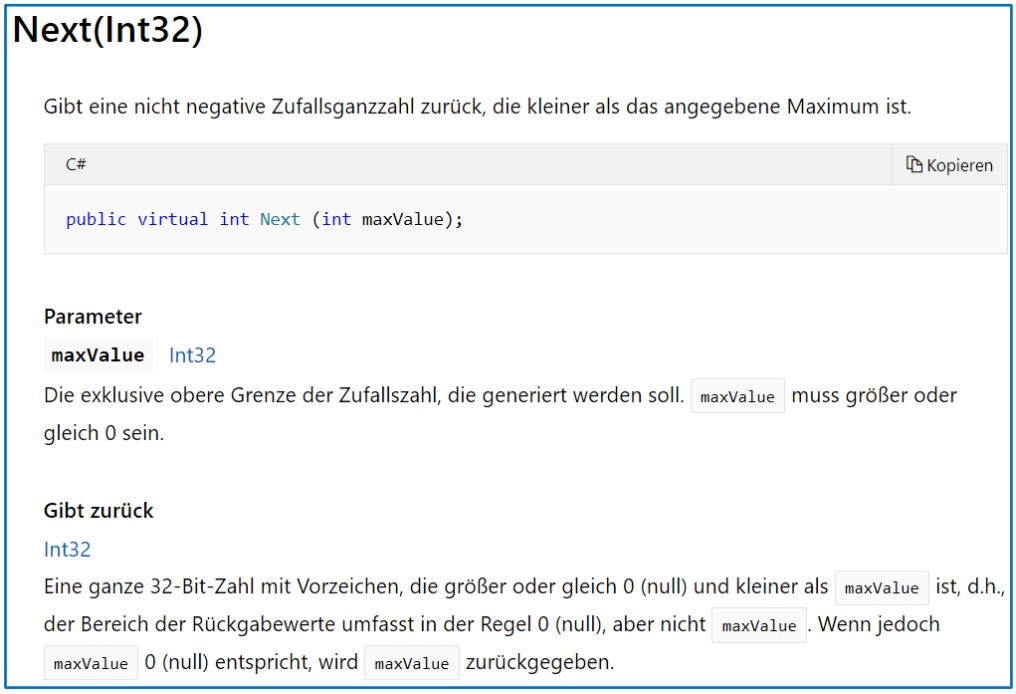
Der voreingestellte Browser zeigt ein HTML-Dokument mit den Überladungen der Methode:



Überladt

Next()	Gibt eine nicht negative Zufallsganzzahl zurück.
Next(Int32)	Gibt eine nicht negative Zufallsganzzahl zurück, die kleiner als das angegebene Maximum ist.
Next(Int32, Int32)	Gibt eine Zufallsganzzahl zurück, die in einem angegebenen Bereich liegt.

Über die im Beispiel verwendete mittlere Überladung erhält man die folgenden Informationen:



Next(Int32)

Gibt eine nicht negative Zufallsganzzahl zurück, die kleiner als das angegebene Maximum ist.

C# Kopieren

```
public virtual int Next (int maxValue);
```

Parameter

maxValue Int32

Die exklusive obere Grenze der Zufallszahl, die generiert werden soll. `maxValue` muss größer oder gleich 0 sein.

Gibt zurück

Int32

Eine ganze 32-Bit-Zahl mit Vorzeichen, die größer oder gleich 0 (null) und kleiner als `maxValue` ist, d.h., der Bereich der Rückgabewerte umfasst in der Regel 0 (null), aber nicht `maxValue`. Wenn jedoch `maxValue` 0 (null) entspricht, wird `maxValue` zurückgegeben.

Eine weitere Anleitung zur Nutzung der BCL-Dokumentation ist in diesem Manuskript sicher nicht mehr erforderlich.

6.2.6 Suchen und Sortieren

Die Klasse **Array** (Namensraum **System**) bietet mehrere statische Methoden, die einen Array nach dem ersten Auftreten eines Wertes durchsuchen. Im folgenden Programm wird die Methode **IndexOf()** verwendet, die den Index des ersten Treffers liefert oder -1, wenn kein Element mit dem angegebenen Wert gefunden wurde:[1]

[1] Im aktuellen Abschnitt wird (die Fähigkeit des Compilers zur Typinferenz ausnutzend) die generische Natur der Methoden **IndexOf<T>()**, **LastIndexOf<T>()**, **FindIndex<T>()** und **FindLastIndex<T>()** aus didaktischen Gründen unterschlagen.

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  const int len = 100_000;`
`  const int mxr = 50_000;`
`  const int ncand = 5;`
 	
`  int[] iar = new int[len];`
`  var zzg = new Random();`
`  for (int i = 0; i < len; i++)`
`  iar[i] = zzg.Next(mxr);`
 	
`  for (int i = 0; i < ncand; i++)`
`  Console.WriteLine("i=" + i + ", Index=" +`
`  Array.IndexOf(iar, zzg.Next(mxr)));`
`  }`
`}`
	`i=0, Index=9605`
	`i=1, Index=92345`
	`i=2, Index=21988`
	`i=3, Index=-1`
	`i=4, Index=13`

Bei alternativen Überladungen der Methode kann durch weitere Parameter festgelegt werden, ...

- dass die Suche an einer bestimmten Indexposition starten soll,
- dass nur eine bestimmte Anzahl von Elementen durchsucht werden soll.

Während die Methode **IndexOf()** einen Array in aufsteigender Ordnung durchsucht, arbeitet die Methode **LastIndexOf()** in umgekehrter Reihenfolge.

Während die beiden eben genannten Methoden nach einem festen Wert suchen, fahnden die Methoden **FindIndex()** und **FindLastIndex()** nach einem Wert mit einer bestimmten Eigenschaft. Im folgenden Beispiel wird der Index des ersten restfrei durch 3 teilbaren Elements in einem per Initialisierungsliste (vgl. Abschnitt 6.2.7) erstellten **int**-Array ermittelt:

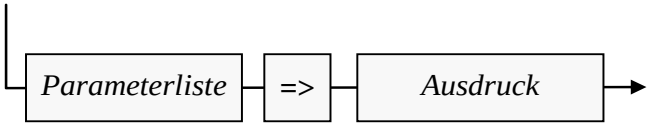
Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  int[] iar = { 2, 13, 4, 12, 7 }`
`  Console.WriteLine("Index: " +`
`  Array.FindIndex(iar, (int i) => i%3 == 0));`
`  }`
`}`
	`Index: 3`

Als zweiter Parameter wird im Beispiel an die Methode **FindIndex()** ein per Ausdrucks-Lambda realisiertes **Delegatenobjekt** (siehe Abschnitt 10.1.6) übergeben. Es führt eine Methode zur Beurteilung der Array-Elemente aus (Rückgabetyp **bool**). Wird die Methode **FindIndex()** nicht fündig, liefert sie die Rückgabe -1.

Die Syntax für ein per Ausdrucks-Lambda realisiertes Delegatenobjekt zeigt eine starke Ähnlichkeit zu der im Abschnitt 5.7.3 beschriebenen Methodendefinition per Lambda-Operator:

Delegatenobjekt per Ausdrucks-Lambda



Die eben erwähnten und weitere Suchmethoden der Klasse **Array** werden von Griffith (2013, S. 155ff) ausführlich beschrieben.

Beim Sortieren eines Arrays über eine von den zahlreichen Überladungen der statischen **Array**-Methode **Sort()** resultiert die Vielfalt u. a. daraus, dass

- entweder die natürliche Anordnung der Elemente, basierend auf der **CompareTo()** - Methode des Elementtyps benutzt wird,[1]
- oder ein die Schnittstelle **IComparer<T>** erfüllendes Objekt engagiert wird, das eine beliebig definierte Anordnung von zwei Elementen liefert.

Das folgende Programm sortiert **int**-Werte auf Basis der natürlichen Ordnung:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  int[] iar = new int[7];`
`  iar[0] = 7; iar[1] = 2; iar[2] = 4; iar[3] = 3;`
`  iar[4] = 5; iar[5] = 1; iar[6] = 6;`
`  Array.Sort(iar);`
`  foreach (int i in iar)`
`  Console.WriteLine(i);`
`  }`
`}`
	1
	2
	3
	4
	5
	6
	7

Zu der im Beispiel verwendeten mühseligen Array-Initialisierung wird gleich im Abschnitt 6.2.7 eine sinnvolle Alternative vorgestellt.

Ist ein Array in sortiertem Zustand, dann kann die Suche nach einem Element durch das in der statischen **Array**-Methode **BinarySearch()** implementierte **binäre Suchverfahren** im Vergleich zu den oben beschriebenen Suchmethoden erheblich beschleunigt werden:

- Im ersten Schritt wird der gesuchte Wert mit dem Element in der Mitte des Arrays verglichen:
  - Stimmen beide überein, ist die Suche erfolgreich beendet.
  - Ist der gesuchte Wert größer als das mittlere Array-Element, wird die Suche in der oberen Array-Hälfte fortgesetzt.
  - Ist der gesuchte Wert kleiner als das mittlere Array-Element, wird die Suche in der unteren Array-Hälfte fortgesetzt.
- Im zweiten Schritt wird in der relevanten Array-Hälfte das Element in der Mitte aufgesucht usw.

Das folgende Beispielprogramm nach Griffith (2013, S. 158) zeigt allerdings, dass der Zeitgewinn beim binären Suchen im Vergleich zum einfachen Suchen nur bei einer hohen Anzahl von Suchvorgängen den Aufwand des vorherigen Sortierens rechtfertigt:

```
using System;
class Prog {
   static void Main() {
      const int len = 1_000_000;
      const int mxr = 1_000_000;
      const int nCand = 10;

      int[] iar = new int[len];
      Random zzg = new Random();
      for (int i = 0; i < len; i++)
         iar[i] = zzg.Next(mxr);
```

[1] Eine mit **CompareTo()** beauftragte Instanz vergleicht sich mit dem Aktualparameter und liefert die Rückgaben -1, 0, oder 1, wenn sie kleiner, gleich oder größer als der Aktualparameter ist. Die Methode **CompareTo()** wird später noch ausführlich behandelt.

```
        long time = DateTime.Now.Ticks; // Verhindert einen verzerrten Wert der 1. Messung
        time = DateTime.Now.Ticks;
        for (int i = 0; i < nCand; i++)
           Console.WriteLine("i=" + i + ", Index=" +
              Array.IndexOf(iar, zzg.Next(mxr)));
        time = DateTime.Now.Ticks - time;
        Console.WriteLine("\nBenöt. Zeit für die einfache Suche:\t" +
                time / 1.0e4 + " Millisek.");

        time = DateTime.Now.Ticks;
        Array.Sort(iar);
        time = DateTime.Now.Ticks - time;
        Console.WriteLine("\nBenöt. Zeit für das Sortieren:\t\t" +
                     time / 1.0e4 + " Millisek.\n");

        time = DateTime.Now.Ticks;
        for (int i = 0; i < nCand; i++)
           Console.WriteLine("i=" + i + ", Index=" +
              Array.BinarySearch(iar, zzg.Next(mxr)));
        time = DateTime.Now.Ticks - time;
        Console.WriteLine("\nBenöt. Zeit für die binäre Suche:\t" +
                     time / 1.0e4 + " Millisek.");
    }
}
```

In einem Array mit 1 Milliarde **int**-Elementen mit Zufallswerten aus dem Bereich von 0 bis 1.000.000-1 werden 10 Werte gesucht:

- Vor dem Sortieren mit der **Array**-Methode **IndexOf()**
- Nach dem Sortieren mit der **Array**-Methode **BinarySearch()**

Bei **BinarySearch()** signalisiert eine negative Rückgabe eine gescheiterte Suche.[1] Die folgenden Messergebnisse wurden mit der Release-Konfiguration des Projekts ermittelt:

```
i=0, Index=-1
i=1, Index=-1
i=2, Index=506729
i=3, Index=571602
i=4, Index=-1
i=5, Index=35363
i=6, Index=983763
i=7, Index=-1
i=8, Index=623382
i=9, Index=512375

Benöt. Zeit für die einfache Suche:     12,0002 Millisek.

Benöt. Zeit für das Sortieren:          93,9926 Millisek.

i=0, Index=602865
i=1, Index=281577
i=2, Index=-652928
i=3, Index=819123
i=4, Index=-669718
i=5, Index=-970766
i=6, Index=-830349
i=7, Index=-372915
i=8, Index=-87305
i=9, Index=340960

Benöt. Zeit für die binäre Suche:       0,9996 Millisek.
```

[1] Zur genauen Bedeutung der negativen Rückgabewerte siehe: https://docs.microsoft.com/en-us/dotnet/api/system.array

Die binäre Suche ist also nur dann sinnvoll, wenn ein Array bereits sortiert vorliegt, oder wenn viele Suchvorgänge erforderlich sind.

6.2.7 Initialisierungslisten

Bei einem Array mit wenigen Elementen ist die Möglichkeit von Interesse, beim Deklarieren der Referenzvariablen eine Initialisierungsliste mit den Werten für die Elementvariablen anzugeben und das Array-Objekt dabei implizit (ohne Verwendung des **new**-Operators) zu erzeugen, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  int[] wecktor = {1, 2, 3};`
`  Console.WriteLine(wecktor[2]);`
`  }`
`}`
	`3`

Die Deklarations- und Initialisierungsanweisung

```
int[] wecktor = {1, 2, 3};
```

ist äquivalent zu:

```
int[] wecktor = new int[3];
wecktor[0] = 1;
wecktor[1] = 2;
wecktor[2] = 3;
```

Initialisierungslisten sind nicht nur bei der Deklaration erlaubt, sondern auch bei der Objektkreation per **new**-Operator, z. B.:

```
int[] wecktor;
 . . .
wecktor = new int[] {1, 2, 3};
```

Haben alle Elemente einer Initialisierungsliste denselben Datentyp, sodass keine erweiternde Typanpassung erforderlich ist, dann kann man die Typangabe weglassen und die Typinferenz des Compilers nutzend das eckige Klammernpaar direkt hinter das Schlüsselwort **new** schreiben, z. B.:

```
wecktor = new[] {1, 2, 3};
```

Die implizite Objektkreation (*ohne* Schlüsselwort **new**) akzeptiert der Compiler ausschließlich bei der Array-Deklaration mit Initialisierung, im folgenden Beispiel also *nicht*:

```
wecktor = {1, 2, 3}; // Nicht erlaubt
```

6.2.8 Objekte als Array-Elemente

Für die Elemente eines Arrays ist natürlich auch ein Referenztyp erlaubt. Im folgenden Beispiel wird ein Array mit `Bruch`-Objekten erzeugt:

Quellcode	Ausgabe

`using System;`
`class BruchRechnung {`
`  static void Main() {`
`  Bruch b1 = new Bruch(1, 2, "b1");`
`  Bruch b2 = new Bruch(5, 6, "b2");`
`  Bruch[] bruvek = {b1, b2};`
`  bruvek[1].Zeige();`
`  }`
`}`
	`  5`
	`b2 = -----`
	`  6`

6.2.9 Indizes und Bereiche

Für Arrays und andere Typen mit Indexzugriff (z. B. **String**) bietet C# seit der Version 8.0 Verbesserungen bei der Ansprache von einzelnen Elementen und von zusammenhängenden Bereichen.

Mit Hilfe des ^ - Operators lässt sich ein Element mit Bezug auf das *Ende* des Arrays ansprechen, wobei ^1 das letzte Element adressiert, ^2 das vorletzte usw. Im folgenden Beispiel werden Elemente in einem **char**-Array angesprochen:

Quellcode	Ausgabe

`using System;`
`public class Prog {`
`  static void Main() {`
`  char[] ca = new char[] { 'a', 'b', 'c', 'd' };`
`  Console.WriteLine($"{ca[^1]} {ca[^2]} {ca[^3]}");`
`  }`
`}`
	`d c b`

Weil beim Array `ca` der Ausdruck

```
ca[^i]
```

für

```
ca[ca.Length - i]
```

steht, existiert das Element `ca[^0]` nicht.

Mit dem Bereichsoperator **..** lässt sich der Bereich von einer Startposition (inklusive) bis zu einer Endposition (exklusive) ansprechen, z. B.:

```
var bc = ca[1 .. ^1]; // {'b', 'c'}
```

Bei einer fehlenden Startposition wird das erste Element angenommen und bei einer fehlenden Endposition das letzte Element, z. B.:

```
var abc = ca[.. ^1]; // {'a', 'b', 'c'}
var cd = ca[2 ..]; // {'c', 'd' }
```

6.2.10 Mehrdimensionale Arrays

In der linearen Algebra und in vielen anderen Anwendungsbereichen werden auch *mehrdimensionale* Arrays benötigt, z. B.:

Quellcode	Ausgabe

```
using System;
class Prog {
   static void Main() {
      int[,] matrix = new int[4, 3];
      int nrow = matrix.GetLength(0);
      int ncol = matrix.GetLength(1);
      int nelem = matrix.Length;
      Console.WriteLine("{0} Dimensionen,\n{1} Zeilen, {2} Spalten" +
                  "\n{3} Elemente", matrix.Rank, nrow, ncol, nelem);
      for (int i = 0; i < nrow; i++) {
         for (int j = 0; j < ncol; j++) {
            matrix[i, j] = (i + 1) * (j + 1);
            Console.Write("{0,3}", matrix[i, j]);
         }
         Console.WriteLine();
      }
   }
}
```

```
2 Dimensionen,
4 Zeilen, 3 Spalten
12 Elemente
  1  2  3
  2  4  6
  3  6  9
  4  8 12
```

Im Beispiel wird ein *zweidimensionaler* Array (eine Matrix) mit 4 Zeilen und 3 Spalten erzeugt, wobei sich die Zellen per Doppelindizierung ansprechen lassen. Bei der Erzeugung bzw. Verwendung eines mehrdimensionalen Arrays sind die in eckigen Klammern eingeschlossenen Dimensionsangaben bzw. Indexwerte jeweils durch ein Komma zu trennen.

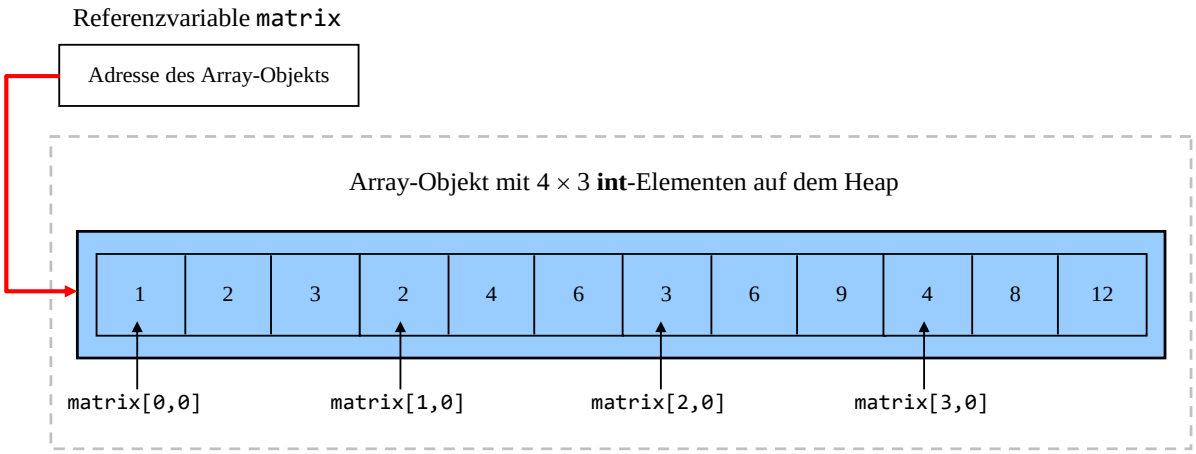
Auch im mehrdimensionalen Fall sind Initialisierungslisten erlaubt, wobei z. B. die Elemente einer (zweidimensionalen) Matrix zeilendominant anzugeben sind (als Liste von Zeilenvektoren):

```
int[,] matrix = {{1, 2, 3}, {2, 4, 6}, {3, 6, 9}, {4, 8, 12}};
```

Weil alle Arrays von der Basisklasse **Array** im Namensraum **System** abstammen, verfügen sie über entsprechende Methoden und Eigenschaften (siehe BCL-Dokumentation), z. B.:

- Die Eigenschaft **Rank** enthält die Anzahl der Dimensionen (den Rang).
- Über die Methode **GetLength()** erfährt man von einem Array-Objekt die Anzahl der Elemente in der per Parameter angegebenen Dimension.
- Die Eigenschaft **Length** mit dem Datentyp **int** liefert die Gesamtzahl der Array-Elemente, im Beispiel also $4 \cdot 3 = 12$.
- Hat ein mehrdimensionaler Array mehr als 2^{31}-1 Elemente, dann verwendet man statt **Length** die Eigenschaft **LongLength** mit dem Datentyp **long**, um die Gesamtzahl der Elemente zu ermitteln (vgl. Abschnitt 6.2.4).

Im Speicher liegen alle Elemente unmittelbar hintereinander, was einen schnellen Indexzugriff ermöglicht:



Es ist eine beliebige Anzahl von Dimensionen (ein beliebiger Rang) möglich, wobei aber meist nur der zweidimensionale Fall von Relevanz ist.

6.2.11 Array aus Arrays

C# unterstützt auch Arrays, die als Elemente wiederum Arrays enthalten. So lässt sich etwa eine zweidimensionale Matrix mit unterschiedlich langen Zeilen realisieren:

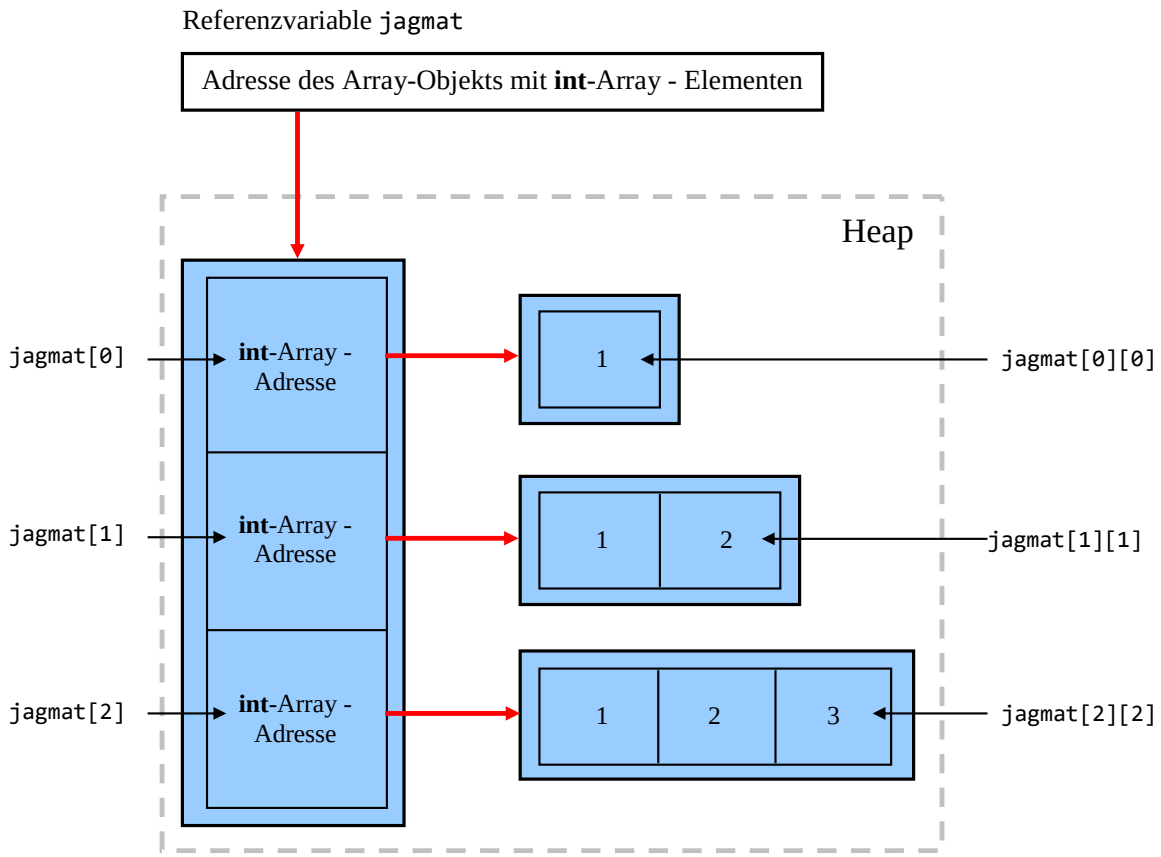
Quellcode	Ausgabe

```
using System;
class Prog {
  static void Main() {
    int[][] jagmat = new int[3][];
    jagmat[0] = new int[1] {1};
    jagmat[1] = new int[2] {1, 2};
    jagmat[2] = new int[3] {1, 2, 3};
    for (int i = 0; i < jagmat.Length; i++) {
      for (int j = 0; j < jagmat[i].Length; j++)
        Console.Write(jagmat[i][j] + " ");
      Console.WriteLine();
    }
  }
}
```

```
1
1 2
1 2 3
```

Man spricht hier auch von „ausgesägten“ Arrays (engl.: *jagged arrays*).

Im Unterschied zum **int[,]** - Objekt `matrix` aus dem Beispiel im Abschnitt 6.2.10, das doppelt indizierte **int**-Elemente enthält, handelt es sich bei den Elementen des **int[][]** - Objekts `jagmat` um Referenzen vom Typ **int[]**, die wiederum auf entsprechende Heap-Objekte (oder **null**) zeigen können. Während `matrix` ein *zwei*dimensionaler Array mit **int**-Elementen ist, handelt es sich bei `jagmat` um einen *ein*dimensionalen Array mit **int[]** - Elementen:



Beim Erzeugen des `jagmat`-Objekts darf nur die Elementzahl des äußeren Arrays angegeben werden:[1]

```
int[][] jagmat = new int[3][];
```

Anschließend erzeugt man die **int[]** - Objekte und legt ihre Adressen in den `jagmat`-Elementvariablen ab, z. B.:

```
jagmat[2] = new int[3] {1, 2, 3};
```

Auch bei einem ausgesägten Array lässt sich mit Hilfe einer Initialisierungsliste und durch Ausnutzung von Compiler-Intelligenz (automatische Ermittlung des Typs und der Elementzahl des äußeren Arrays) der Schreibaufwand reduzieren. Es folgt eine zulässige Alternative zu der aus didaktischen Gründen oben bevorzugten, ausführlichen Schreibweise:

```
int[][] jagmat = {new[] {1}, new[] {1, 2}, new[] {1, 2, 3}};
```

Im Unterschied zur Initialisierungsliste für einen mehrdimensionalen Array (vgl. Abschnitt 6.2.10) muss der **new**-Operator für jeden inneren Array wiederholt werden. Es entsteht ja nicht (wie im mehrdimensionalen Fall) *ein* Array, sondern es entstehen ...

- ein äußerer Array
- mehrere selbständige innere Arrays

---

[1] Nach Griffith (2013, S. 163f) sollte im **new**-Ausdruck die Elementzahl des äußeren Arrays (mit dem Elementtyp **int[]**) eigentlich durch das zweite Klammernpaar begrenzt werden. Die C# - Designer haben sich aber anders entschieden. Vermutlich hat man sich daran orientiert, dass die von `jagmat`-Elementen indizierten **int[]** - Arrays als *Zeilenvektoren* aufgefasst werden. In der mathematischen Matrix-Notation wird der Zeilenindex meist zuerst angegeben (Zeilendominanz).

6.2.12 Kollektionsklasse ArrayList

Im Namensraum **System.Collections** bietet die BCL etliche Klassen zur flexiblen Verwaltung von Datenbeständen *variablen* Umfangs, mit denen wir uns später noch detailliert beschäftigen werden. Dort findet sich u. a. die Klasse **ArrayList**, deren Objekte analog zu eindimensionalen Arrays genutzt werden können. Man legt jedoch beim Erzeugen eines **ArrayList**-Objekts keinen Umfang fest, sondern kann z. B. mit der Methode **Add()** nach Bedarf neue Elemente einfügen, z. B.:

Quellcode	Ausgabe (Eingaben fett)

```
using System;
using System.Collections;
class ArrayListDemo {
   static void Main() {
      var al = new ArrayList();
      string s;
      Console.WriteLine("Was fällt Ihnen zu C# ein?\n");
      do {
         Console.Write(": ");
         s = Console.ReadLine();
         if (s.Length > 0)
            al.Add(s);
         else
            break;
      } while (true);

      Console.WriteLine("\nIhre Anmerkungen:");
      for(int i = 0; i < al.Count; i++)
         Console.WriteLine(al[i]);
   }
}
```

```
Was fällt Ihnen zu C# ein?

: Tolle Sache
: Nicht ganz trivial
: Macht Spaß
:

Ihre Anmerkungen:
Tolle Sache
Nicht ganz trivial
Macht Spaß
```

Das Fassungsvermögen des Containers wird bei Bedarf automatisch erhöht, wobei eine leistungsoptimierende Logik dafür sorgt, dass diese Anpassungsmaßnahme möglichst selten erforderlich ist. Über die Eigenschaft **Capacity** kann die momentane Kapazität festgestellt und auch eingestellt werden. Mit der Methode **TrimToSize()** reduziert man die Größe auf den momentanen Bedarf, z. B. nach der voraussichtlich letzten Neuaufnahme.

Weil die Klasse **ArrayList** einen *Indexer* bietet (siehe Abschnitt 5.8), kann man per Indexsyntax (über einen durch rechteckige Klammern begrenzten ganzzahligen Ausdruck, siehe Beispiel) auf die Elemente zugreifen:

- Das erste Element hat den Indexwert 0.
- Das letzte Element hat den Indexwert (**Count** – 1), wobei die **Count**-Eigenschaft die Anzahl der Elemente angibt.

Als Datentyp für die **ArrayList**-Elemente dient **System.Object**, also die Klasse an der Spitze des allgemeinen Typsystems der .NET - Plattform. Folglich kann ein **ArrayList**-Container Daten beliebigen Typs aufnehmen, wobei dank Boxing-Technik (siehe Abschnitt 6.1.3) auch die Werttypen erlaubt sind, z. B.:

Quellcode	Ausgabe

`using System;`
`using System.Collections;`
`class Prog {`
`  static void Main() {`
`  var al = new ArrayList();`
`  al.Add("Wort");`
`  al.Add(3.14);`
`  al.Add(13);`
`  foreach (object o in al)`
`  Console.WriteLine(o);`
`  }`
`}`
	`Wort`
	`3,14`
	`13`

Ein solcher „Gemischtwarenladen“ (allerdings mit *fester* Länge) ist durch Wahl des Element-Datentyps **Object** übrigens auch mit einem einfachen Array zu realisieren.

Im Kapitel 8 über das typgenerische Programmieren wird sich herausstellen, dass dem Gemischtwarenladen **ArrayList** bei sehr vielen Anwendungen die generische Klasse **List<T>** überlegen ist. Sie bietet einen größendynamischen Container mit einem *festen*, beim Erstellen des Containers zu bestimmenden Elementtyp. Wenn jedoch tatsächlich ein Gemischtwarenladen benötigt wird, kommt die Klasse **ArrayList** weiterhin in Frage.

6.3 Klassen für Zeichenketten

C# bietet für die Verwaltung von Zeichenfolgen, die grundsätzlich aus Unicode-Zeichen bestehen, zwei Klassen an, die für unterschiedliche Einsatzzwecke optimiert wurden:

- **String** (im Namensraum **System**)
  String-Objekte können nach dem Erzeugen nicht mehr geändert werden. Momentan erscheinen Ihnen unveränderliche Objekte eventuell noch als eingeschränkt brauchbar. Im weiteren Verlauf des Kurses wird aber immer öfter von den Vorteilen unveränderlicher Objekte zu hören sein (z. B. im Zusammenhang mit dem Multithreading).
- **StringBuilder** (im Namensraum **System.Text**)
  Für *variable*, d.h. im Programmablauf häufig zu ändernde Zeichenketten sollte unbedingt die Klasse **StringBuilder** verwendet werden, weil deren Objekte nach dem Erzeugen noch modifiziert werden können.

6.3.1 Die Klasse String für unveränderliche Zeichenketten

Weil Objekte der Klasse **String** aus dem Namensraum **System** in C# - Programmen sehr oft benötigt werden, hat man diesem Datentyp das reservierte Wort **string** (mit *klein* geschriebenen Anfangsbuchstaben) als Aliasnamen spendiert, und das ist nicht die einzige syntaktische Vorzugsbehandlung gegenüber anderen Klassen. In der folgenden Deklarations- und Initialisierungsanweisung

```
string s1 = "abcde";
```

wird:

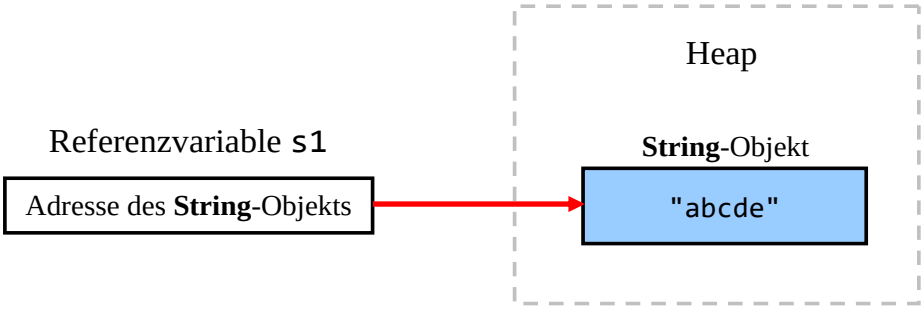
- eine **String**-Referenzvariable namens `s1` angelegt,
- ein neues **String**-Objekt mit dem Inhalt „abcde“ auf dem Heap erzeugt,
- die Adresse des neuen Heap-Objekts in der Referenzvariablen abgelegt.

Soviel objektorientierten Hintergrund sieht man der angenehm einfachen Anweisung auf den ersten Blick nicht an. In C# sind jedoch auch Zeichenketten*literale* als **String**-Objekte realisiert, sodass z. B.

```
"abcde"
```

einen Ausdruck darstellt, der als Wert einen Verweis auf ein **String**-Objekt auf dem Heap liefert.

Weil in der obigen Deklarations- und Initialisierungsanweisung kein **new**-Operator auftaucht, spricht man auch vom *impliziten* Erzeugen eines **String**-Objekts. Die Anweisung bewirkt im Hauptspeicher die folgende Situation:



6.3.1.1 String als WORM - Klasse

Nachdem ein **String**-Objekt auf dem Heap erzeugt wurde, ist es **unveränderlich** (engl.: *immutable*). In der Abschnittsüberschrift wird für diesen Sachverhalt eine Abkürzung aus der Elektronik ausgeliehen: WORM (***W**rite **O**nce **R**ead **M**any*). Eventuell werden Sie die Unveränderlichkeit von **String**-Objekten in Zweifel ziehen und ein Gegenbeispiel der folgenden Art vorbringen:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  String testr = "abc";`
`  Console.WriteLine("testr = " + testr);`
`  testr += "def";`
`  Console.WriteLine("testr = " + testr);`
`  }`
`}`
	`testr = abc`
	`testr = abcdef`

In der Zeile

```
testr += "def";
```

wird aber das per `testr` ansprechbare **String**-Objekt (mit dem Text „abc“) nicht geändert, sondern durch ein neues **String**-Objekt (mit dem Text „abcdef“) ersetzt. Das alte Objekt ist noch vorhanden, aber nicht mehr referenziert. Sobald das Laufzeitsystem Langeweile hat oder Speicher benötigt, wird das alte Objekt vom Garbage Collector eliminiert.

6.3.1.2 Methoden für String-Objekte

Von den zahlreichen Instanzmethoden und -eigenschaften der Klasse **String** werden in diesem Abschnitt nur die wichtigsten angesprochen. Für spezielle Anwendungen lohnt sich also ein Blick in die BCL-Dokumentation.

6.3.1.2.1 Verketten von Strings

Weil die Klasse **String** den „+“- Operator geeignet überladen hat (vgl. Abschnitt 5.7.2), taugt er zum Verketten von **String**-Objekten, wobei Operanden beliebiger Datentypen bei Bedarf automatisch in **String**-Objekte konvertiert werden. Wie Sie aus dem Abschnitt 6.3.1.1 wissen, entsteht beim Verketten von zwei Zeichenfolgen ein *neues* **String**-Objekt. Im ersten **WriteLine()** - Aufruf des folgenden Beispielprogramms wird mit Klammern dafür gesorgt, dass der Compiler die „+“ - Operatoren jeweils sinnvoll interpretiert (Verketten von Strings bzw. Addieren von Zahlen):

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  Console.WriteLine("4 + 3 = " + (4 + 3));`
`  Console.WriteLine((15+2) + " ist eine Primzahl.");`
`  }`
`}`
	`4 + 3 = 7`
	`17 ist eine Primzahl.`

Wie der zweite **WriteLine()** - Aufruf im Beispielprogramm zeigt, hängt die Interpretation des „+“ - Zeichens als Operator-Überladung der Klasse **String** nicht davon ab, ob das Zeichenfolgen-Argument links oder rechts vom Operatorsymbol steht.

6.3.1.2.2 Vergleichen von Strings

Angewandt auf **String**-Variablen vergleichen die auf **Identität** prüfenden Operatoren „==“ und „**!=**“ *nicht* (wie z. B. in Java) die in den Variablen abgelegten *Adressen*, sondern die *Inhalte* der referenzierten Zeichenfolgenobjekte, z. B.:

Das C#-Programm liefert **true**	Das Java-Programm liefert **false**

`using System;`
`class Prog {`
`  static void Main() {`
`  String s1 = "abcde";`
`  String s2 = "de";`
`  String s3 = "abc" + s2;`
`  Console.WriteLine(s1 == s3);`
`  }`
`}`
	`class Prog {`
	`  public static void main(String[] args) {`
	`  String s1 = "abcde";`
	`  String s2 = "de";`
	`  String s3 = "abc" + s2;`
	`  System.out.println(s1 == s3);`
	`  }`
	`}`

Mit der etwas umständlichen `s3`-Konstruktion wird verhindert, dass `s1` und `s3` auf dasselbe Objekt im internen **String**-Pool zeigen, weil dann Adress- und Inhaltsvergleich zum selben Ergebnis kämen. Wie im Abschnitt 6.3.1.3 demonstriert wird, ist bei einer großen Anzahl von **String**-Vergleichen durch das sogenannte Internalisieren und die Verwendung von Adressvergleichen eine Leistungssteigerung zu erzielen.

Zum Testen auf **lexikographische Priorität** (z. B. beim Sortieren) kann die **String**-Methode **CompareTo()**

public int CompareTo (string *verglString***)**

dienen:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  String a = "Müller, Anja", b = "Müller, Kurt",`
`  c = "Müller", d = ", Anja";`
`  c += d;`
`  Console.WriteLine("<  : " + a.CompareTo(b));`
`  Console.WriteLine("== : " + a.CompareTo(c));`
`  Console.WriteLine(">  : " + b.CompareTo(a));`
`  }`
`}`
	`<  : -1`
	`== : 0`
	`>  : 1`

CompareTo() liefert folgende **int**-Rückgabewerte:

		CompareTo() - Ergebnis
Die lexikographische Priorität des angesprochenen **String**-Objekts ist im Vergleich zum Parameterobjekt:	kleiner	-1
	gleich	0
	größer	1

6.3.1.2.3 Länge einer Zeichenkette

Über die Länge einer Zeichenkette informiert die **String**-Eigenschaft **Length**, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  Console.WriteLine("abc".Length);`
`  }`
`}`
	`3`

6.3.1.2.4 Zeichen(folgen) extrahieren, suchen oder ersetzen

Auf einzelne Zeichen eines Strings kann man per Indexsyntax zugreifen, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  string s = "abcd";`
`  Console.WriteLine(s[0]);`
`  Console.WriteLine(s.Substring(0, 2));`
`  Console.WriteLine(s.IndexOf("c"));`
`  Console.WriteLine(s.IndexOf("x"));`
`  Console.WriteLine(s.StartsWith("a"));`
`  Console.WriteLine(s.Replace('c', 'C'));`
`  }`
`}`
	`a`
	`ab`
	`2`
	`-1`
	`True`
	`abCd`

Über die Methode

public string Substring(int *start***, int** *anzahl***)**

erhält man von einem **String**-Objekt die *anzahl* Zeichen ab Position *start* (inklusive) als Kopie.

Mit der Methode

public int IndexOf(string *gesucht***)**

kann man einen **String** nach der Existenz einer anderen Zeichenfolge befragen. Als Rückgabewert erhält man ...

- nach erfolgreicher Suche: die (0-basierte) Startposition der ersten Trefferstelle
- nach vergeblicher Suche: -1

Mit der Methode

public bool StartsWith(string *start***)**

lässt sich feststellen, ob ein String mit einer bestimmten Zeichenfolge beginnt.

Mit den Überladungen der Methode

public string Replace(char *alt*, **char** *neu***)**

public string Replace(string *alt*, **string** *neu***)**

erhält man als Rückgabewert die Adresse eines neuen **String**-Objekts, das aus dem angesprochenen Original durch Ersetzen eines alten Zeichens (einer alten Zeichenfolge) durch ein neues Zeichen (eine neue Zeichenfolge) hervorgeht.

6.3.1.2.5 Groß-/Kleinschreibung normieren

Mit den Methoden

public String ToUpper()

bzw.

public String ToLower()

erhält man einen neuen **String**, der im Unterschied zum angesprochenen Original auf Groß- bzw. Kleinschreibung normiert ist, was vor Vergleichen oft sinnvoll ist, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  String a = "Otto", b = "otto";`
`  Console.WriteLine(a.ToUpper() == b.ToUpper());`
`  Console.WriteLine(a.ToUpper().IndexOf("T"));`
`  }`
`}`
	`True`
	`1`

In der letzten Anweisung des Beispiels ist der **WriteLine()** - Parameter etwas komplex geraten, sodass eine kurze Erklärung angemessen ist:

- Der linke Punktoperator wird zuerst ausgeführt. Dabei erzeugt der Methodenaufruf `a.ToUpper()` ein neues **String**-Objekt und liefert die zugehörige Referenz.
- Diese Referenz ermöglicht es, dem neuen Objekt Botschaften zu übermitteln, was durch den Methodenaufruf `IndexOf("T")` geschieht.

6.3.1.3 Interner String-Pool

Ein Zeichenkettenliteral ist ein Ausdruck vom Typ **String** mit einer Speicheradresse eines Objekts als Wert. Für wiederholt im Quellcode auftretende identische Zeichenkettenliterale lässt dieses Prinzip eine Verschwendung von Speicherplatz durch eine Vielzahl von Objekten mit identischem Inhalt befürchten. Um das zu verhindern, verwaltet die CLR eine als **interner String-Pool** bezeichnete Tabelle ...

- für die auf Literalen basierenden **String**-Objekte
- und für explizit internalisierte **String**-Objekte (siehe unten).

Wird zu einem Zeichenkettenliteral eine Objektreferenz benötigt, liefert die CLR nach Möglichkeit die Adresse eines bereits vorhandenen, inhaltsgleichen **String**-Objekts aus dem internen Pool. Schlägt die Suche fehl, wird ein neues Objekt erzeugt und im internen Pool registriert. Diese Vorgehensweise ist sinnvoll, weil sich vorhandene **String**-Objekte garantiert nicht mehr ändern. Im folgenden Beispiel zeigen eine Instanzvariable und eine lokale Variable vom Typ **String** auf dasselbe Objekt:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  string sf = "Dies ist ein Zeichenketten-Literal";`
`  static void Main() {`
`  string ls = "Dies ist ein Zeichenketten-Literal";`
`  Prog p = new Prog();`
`  Console.WriteLine(Object.ReferenceEquals(ls, p.sf));`
`  }`
`}`
	`True`

Dass die beiden Referenzvariablen tatsächlich auf dasselbe Objekt zeigen, wird durch die statische **Object**-Methode **ReferenceEquals()** nachgewiesen.

Über die statische **String**-Methode **Intern()** kann man den internen **String**-Pool zusätzlich bevölkern. Die Methode erwartet einen Aktualparameter vom Typ **String** und liefert einen Rückgabewert vom selben Typ:

- Ist ein Objekt im internen **String**-Pool inhaltsgleich mit dem Parameterobjekt, wird die Adresse dieses Pool-Objekts geliefert.
- Anderenfalls nimmt **Intern()** das Parameterobjekt in den internen **String**-Pool auf und liefert seine Adresse als Rückgabe.

Das Internalisieren kann nicht nur Speicherplatz sparen, wenn viele Strings mit identischem Inhalt zu erwarten sind, sondern es kann vor allem Identitätsvergleiche für Strings (vgl. Abschnitt 6.3.1.2.2) beschleunigen, weil bei Referenzvariablen zu internalisierten Strings aus der Gleichheit der Adressen bereits die Inhaltsgleichheit folgt. Im folgenden Programm werden `anz` Zufallszeichenfolgen der Länge `len` jeweils `vergl` mal mit einem zufällig gewählten Partner verglichen. Dies geschieht zunächst per Inhaltsvergleich und dann nach dem zwischenzeitlichen Internalisieren per Adressvergleich:

```
using System;
using System.Text;

class StringIntern {
   public static void Main() {
      const int anz = 50_000, len = 50, vergl = 5;
      var sb = new StringBuilder();
      var ran = new Random();
      String[] sar  = new String[anz];
      for (int i = 0; i < anz; i++) {
         for (int j = 0; j < len; j++)
            sb.Append((char) (65 + ran.Next(26)));
         sar[i] = sb.ToString();
         sb.Remove(0, len);
      }

      long start = DateTime.Now.Ticks; // DateTime außerhalb der Messung laden
      start = DateTime.Now.Ticks;
      int hits = 0;
      // anz * vergl Inhaltsvergleiche
      for (int i = 0; i < anz; i++)
         for (int n = 1; n <= vergl; n++)
            if (sar[i] == sar[ran.Next(anz)])
               hits++;
      Console.WriteLine((anz * vergl)+" Inhaltsvergleiche ("+hits+
            " hits) benötigen "+((DateTime.Now.Ticks - start)/1.0e4)+
            " Millisekunden");

      start = DateTime.Now.Ticks;
      hits = 0;
```

```
        // Internalisieren
        for (int j = 1; j < anz; j++)
           sar[j] = String.Intern(sar[j]);
        Console.WriteLine("Zeit für das Internalisieren: " +
              ((DateTime.Now.Ticks - start) / 1.0e4) + " Millisekunden");
        // anz * vergl Adressvergleiche
        for (int i = 0; i < anz; i++)
           for (int n = 1; n <= vergl; n++)
              if ((Object)sar[i] == sar[ran.Next(anz)])
                 hits++;
        Console.WriteLine((anz * vergl)+" Adressvergleiche ("+hits+
                          " hits) benötigen (inkl. Internalisieren) "+
                          ((DateTime.Now.Ticks - start)/1.0e4)+" Millisekunden");
     }
  }
```

Beim Erzeugen der Zufallszeichenfolgen kommt ein Objekt der Klasse **StringBuilder** zum Einsatz (siehe Abschnitt 6.3.2).

Um den Identitätsoperator zu einem Adressvergleich zu zwingen, wird ein Vergleichspartner als Instanz der Klasse **Object** behandelt:

```
(Object)sar[i] == sar[ran.Next(anz)]
```

Es hängt von den Aufgabenparametern `anz`, `len` und `vergl` ab, welche Vergleichsmethode überlegen ist:[1]

	Laufzeit in Millisekunden	
	Inhaltsvergleiche	Internalisieren + Adressvergl.
`anz` = 50000, `len` = 50, `vergl` = 5	35,9977	61,9962
`anz` = 50000, `len` = 50, `vergl` = 50	365,977	73,9943

Erwartungsgemäß ist das Internalisieren umso rentabler, je mehr Vergleiche anschließend mit den Zeichenfolgen angestellt werden. Bei **String**-Vergleichen sind sicher noch weitere Verbesserungen möglich, z. B. durch Ausnutzen der lexikographischen Ordnung.

Auch die statische **String**-Methode **IsInterned()** liefert die Adresse des Parameter-Strings, falls er sich im internen Pool befindet. Anderenfalls wird jedoch *kein* Pool-String erzeugt, sondern der Wert **null** abgeliefert.

Wird bei der Verwendung des internen **String**-Pools vor allem der Zweck verfolgt, Speicherplatz zu sparen, dann sollten folgende Punkte bedacht werden:[2]

- Im internen Pool befindliche **String**-Objekte persistieren in der Regel auch dann, wenn keine Referenzen mehr im Programm vorhanden sind. Sie sind zwar nicht grundsätzlich aus dem Aufgabenbereich des Garbage Collectors ausgeschlossen, existieren aber meist bis zum Prozessende.
- Bevor ein **String**-Objekt in den Pool gelangt, muss es zunächst als normales Heap-Objekt erstellt werden. Eine obsolet gewordene Heap-Instanz belegt Speicher, bis sie vom Garbage Collector abgeräumt wird.

---

[1] Die Ergebnisse wurden unter Verwendung der Release-Konfiguration der Entwicklungsumgebung auf einem PC mit Intel - CPU Core i3 550 (3,2 GHz) unter Windows 10 (64 Bit) ermittelt.

[2] https://docs.microsoft.com/en-us/dotnet/api/system.string.intern

6.3.2 Die Klasse StringBuilder für veränderliche Zeichenketten

Für häufig zu ändernde Zeichenketten sollte man statt der Klasse **String** unbedingt die Klasse **StringBuilder** aus dem Namensraum **System.Text** verwenden, weil hier beim Ändern einer Zeichenkette die relativ aufwändige Erzeugung eines neuen Objekts entfällt.

Ein **StringBuilder**-Objekt kann nicht *implizit* erzeugt werden, jedoch stehen bequeme Konstruktoren zur Verfügung, z. B.:

- **public StringBuilder**()
  Beispiel: `StringBuilder sb = new StringBuilder();`
- **public StringBuilder**(**String** *str*)
  Beispiel: `StringBuilder sb = new StringBuilder("abc");`

Im folgenden Programm wird eine Zeichenkette 100000-mal verlängert, zunächst mit Hilfe der Klasse **String**, dann mit Hilfe der Klasse **StringBuilder**:

```
using System;
using System.Text;

class StrBldrDemo {
  static void Main() {
    const int n = 100_000;
    String s = "*";
    long vorher = DateTime.Now.Ticks; // Laden von DateTime
    vorher = DateTime.Now.Ticks;
    for (int i = 0; i < n; i++)
      s += "*";
    long diff = DateTime.Now.Ticks - vorher;
    Console.WriteLine("Zeit für String-Manipulation:\t\t" +
                      diff/1.0e4 + "\tMillisekunden");

    var t = new StringBuilder("*");
    vorher = DateTime.Now.Ticks;
    for (int i = 0; i < n; i++)
      t.Append("*");
    diff = DateTime.Now.Ticks - vorher;
    Console.WriteLine("Zeit für StringBuilder-Manipulation:\t" +
                      diff/1.0e4 + "\tMillisekunden");
  }
}
```

Die (in Millisekunden gemessenen) Laufzeiten unterscheiden sich erheblich:[1]

```
Zeit für String-Manipulation:           2381,0253 Millisekunden
Zeit für StringBuilder-Manipulation:    1,0007  Millisekunden
```

Ein **StringBuilder**-Objekt kennt u. a. die folgenden Methoden und Eigenschaften (alle **public**):

[1] Gemessen auf einem Rechner mit der Intel-CPU Core i3 550 (3,2 GHz) unter Windows 10 (64 Bit).

StringBuilder-Member	Erläuterung
Length	enthält die Anzahl der Zeichen
Append()	Das **StringBuilder**-Objekt wird um die String-Repräsentation des Argumentes verlängert, z. B.: `t.Append("*");` Es sind **Append()** - Überladungen für zahlreiche Parameterdatentypen vorhanden.
Insert()	Die **String**-Repräsentation des Argumentes, das von nahezu beliebigem Typ sein kann, wird vom angesprochenen **StringBuilder**-Objekt an der vom ersten Parameter bestimmten Position eingefügt, z. B.: `sb.Insert(4, 3.14);`
Remove()	Ab einer Startposition wird eine Anzahl von Zeichen entfernt, z. B.: `sb.Remove(2, 5);`
Replace()	Ein Zeichen bzw. eine Zeichenfolge des **StringBuilder**-Objekts wird durch anderes Zeichen bzw. eine andere Zeichenfolge ersetzt, z. B.: `sb.Replace("alt", "neu");`
ToString()	Es wird ein **String**-Objekt mit dem Inhalt des **StringBuilder**-Objekts erzeugt. Dies ist z. B. erforderlich, um ein **StringBuilder**- und ein **String**-Objekt nach Inhalt vergleichen zu können: `Console.WriteLine(s == sb.ToString());`

6.4 Enumerationen

Angenommen, Sie entwerfen eine Klasse namens `Person` und wollen auch den Charakter einer Person erfassen. Dabei orientieren sie sich an den vier Temperamentstypen des griechischen Philosophen Hippokrates (ca. 460 - 370 v. Chr.): cholerisch, melancholisch, sanguinisch, phlegmatisch. Um dieses Merkmal mit seinen vier möglichen Ausprägungen in einer Instanzvariablen Ihrer Klasse `Person` zu speichern, kennen Sie bereits verschiedene Möglichkeiten, z. B.:

- Eine **String**-Variable zur Aufnahme der Temperamentsbezeichnung
  Dabei wird relativ viel Speicherplatz benötigt, und es drohen Fehler durch inkonsistente Schreibweisen, z. B.:

```
public string Temp;
  . . .
if (otto.Temp == "Flegmatisch") ...
```

- Eine **int**-Variable mit der Codierungsvorschrift 0 = cholerisch, 1 = melancholisch etc.
  Es wird wenig Speicher benötigt, allerdings ist der Quellcode nur für Eingeweihte zu verstehen, z. B.:

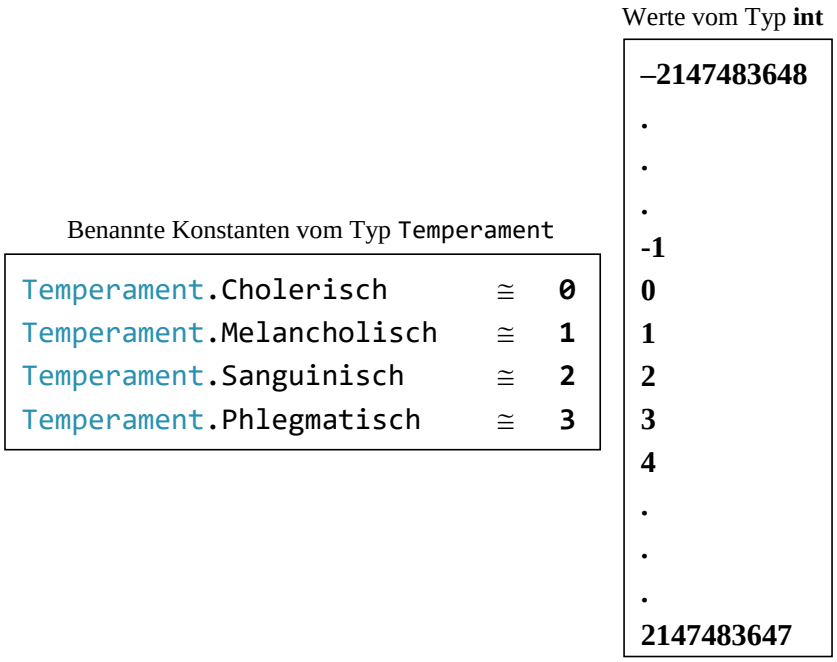
```
public int Temp;
  .  .  .
if (otto.Temp == 3) ...
```

Fehlerhafte Zuweisungen könnte und sollte man bei beiden Lösungsansätzen durch eine sorgfältige Eigenschaftsdefinition verhindern (Abweisen ungeeigneter Werte im **set**-Block, siehe Abschnitt 5.5).

C# bietet mit den **Enumerationen** (**Aufzählungstypen)** eine Lösung, die folgende Vorteile hat:

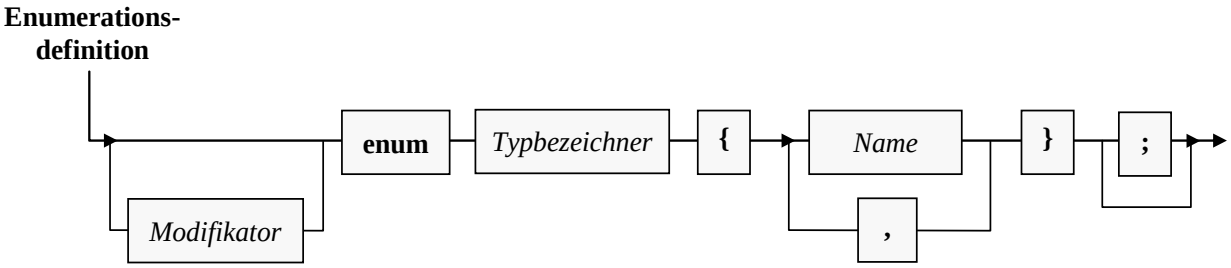
- Gut lesbarer Quellcode durch Klartextnamen für die Merkmalsausprägungen
- Abweisung von falsch geschriebenen Klartextnamen durch den Compiler
- Geringer Speicherbedarf

Eine Enumeration basiert auf einem zugrunde liegenden integralen Typ (meist **int**) und enthält eine (meist kleine) Menge von benannten Konstanten dieses Typs, z. B. die Enumeration `Temperament`:



Sofern man auf eine explizite Typumwandlung verzichtet (siehe unten), können einer Variablen des Enumerationstyps nur die definierten Konstanten zugewiesen werden. Dabei sind nicht die zugrunde liegenden Werte (per Voreinstellung 0, 1, 2, …) zu verwenden, sondern die vereinbarten Namen (z. B. `Temperament.Sanguinisch`).

Bei der Definition eines Aufzählungstyps folgt auf das Schlüsselwort **enum** und den Typbezeichner eine geschweift eingeklammerte Liste mit den Namen für die Konstanten:



Wie bei Klassen und Strukturen …

- wird die Verfügbarkeit einer Enumeration über Modifikatoren geregelt (Voreinstellung: **internal**, Alternative: **public**, vgl. Abschnitt 5.11),
- ist neben einer Top-Level-Definition auch eine eingeschachtelte Definition (innerhalb eines anderen Typs) möglich,
- kann optional hinter der schließenden Klammer der Enumerationsdefinition ein Semikolon stehen.

Als Beispiel betrachten wir eine von Hippokrates inspirierte Enumeration zur Erfassung des Charakters von Personen:

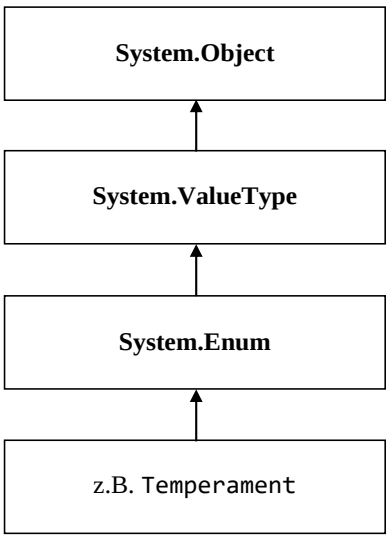
```
public enum Temperament {Cholerisch, Melancholisch, Sanguinisch, Phlegmatisch}
```

Per Voreinstellung stehen die Namen in der Enumerationsdefinition für eine 0-basierte Liste von **int**-Werten, doch lassen sich auch alternative Werte vergeben. Im folgenden Beispiel wird dafür gesorgt, dass die Werteliste bei 1 startet und dann die folgenden natürlichen Zahlen durchläuft. Im Visual Studio wird das numerische Äquivalent zu einer Enumerationskonstanten angezeigt, während sich der Mauszeiger über ihrem Namen befindet, z. B.:

```
public enum Temperament { Cholerisch=1, Melancholisch, Sanguinisch, Phlegmatisch }
```

Temperament.Melancholisch = 2

Die Enumerationen sind **Werttypen** und folgendermaßen in das CTS (Common Type System) der .NET - Plattform eingeordnet:



Alternative Abstammungen sind bei Enumerationen *nicht* möglich; insbesondere kann man eine Enumeration nicht beerben.

Weil Enumerationskonstanten stets mit dem Typnamen qualifiziert werden müssen, ist einige Tipparbeit erforderlich, die aber mit einem gut lesbaren Quellcode belohnt wird, z. B.[1]

```
public class Person {
   public string Vorname;
   public string Name;
   public int Alter;
   public Temperament Temp;

   public Person(string vorname, string name, int alter, Temperament temp) {
      Vorname = vorname; Name = name; Alter = alter; Temp = temp;
   }
}

class PersonTest {
   static void Main() {
      Person otto = new Person("Otto", "Hummer", 35, Temperament.Sanguinisch);
      if (otto.Temp == Temperament.Sanguinisch)
         System.Console.WriteLine("Lustiger Typ!");
   }
}
```

[1] Wir verzichten der Kürze halber bei der Klasse `Person` auf die hier durchaus empfehlenswerte Datenkapselung.

Einer Variablen mit einem Enumerationstyp können leider über eine explizite Typumwandlung neben den benannten Konstanten auch beliebige andere Werte des zugrunde liegenden Typs zugewiesen werden, z. B.:

```
otto.Temp = (Temperament)13;
```

Daher sollten Enumerations-Instanzvariablen (abweichend von dem obigen schlechten Beispiel) in der Regel gekapselt und nur über eine Eigenschaft mit überwachter Wertzuweisung zugänglich sein, z. B.:

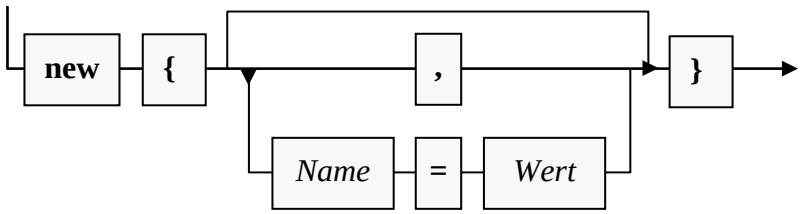
```
Temperament temp;

public Temperament Temp {
   get {
      return temp;
   }
   set {
      if (System.Enum.IsDefined(typeof(Temperament), value))
         temp = value;
   }
}
```

Zum Entscheid über die Gültigkeit eines Wertes lässt sich die statische Methode **IsDefined()** der Klasse **Enum** im Namensraum **System** verwenden. Gegen welchen Enumerationstyp geprüft werden soll, erfährt die Methode **IsDefined()** über das im ersten Aktualparameter zu übergebende **Type**-Objekt, das im Beispiel durch den Operator **typeof** mit dem Enumerationsnamen als Argument geliefert wird.[1]

6.5 Anonyme Klassen

Zu dem im Abschnitt 5.4.3.3 beschriebenen Objektinitialisierer existiert eine Variante *ohne* Klassenname, wobei ein Objekt aus einer anonymen Klasse entsteht:

Objekt einer anonymen Klasse



Im folgenden Beispiel wird es zur Initialisierung einer über das Schlüsselwort **var** implizit typisierten lokalen Variablen verwendet:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`   var a = new {Name = "Knut", Alter = 53 };`
`   Console.WriteLine(a.GetType());`
`  }`
`}`
	`<>f__AnonymousType0`2[System.String,System.Int32]`

Wie das Beispiel zeigt, hat die „anonyme“ Klasse durchaus einen (vom Compiler vergebenen) Namen, doch darf dieser Name im Quellcode nicht verwendet werden. Daher ist das Schlüsselwort

---

[1] Alternativ zum Operator **typeof** hätte die **Object**-Methode **GetType()** verwendet werden können:
`value.GetType()`

var, das wir bisher als Schreiberleichterung kennengelernt haben, hier erforderlich, um eine Variable mit dem anonymen Typ deklarieren zu können.

Als Innenausstattung enthält die anonyme Klasse eine öffentliche **get-only** - Eigenschaft für jedes Element in der Liste des Objektinitialisierers, sodass im Beispiel eine Klasse mit den Eigenschaften `Name` und `Alter` resultiert, wobei der Compiler die Datentypen (**String** und **Int32**) aus den zugewiesenen Werten ermittelt. Hinter jeder Eigenschaft steht ein privates Feld. Weil ein Objekt einer anonymen Klasse nur private Felder mit zugehörigen **get-only** - Eigenschaften besitzt, ist es *unveränderlich*, wie der folgende Fehlversuch demonstriert:

```
var a = new { Name = "Knut", Alter = 53 };
a.Alter = 54;
```

Eine anonyme Klasse erbt die Methoden ihrer Basisklasse **Object**. Aufgrund ihrer Entstehungsgeschichte können anonyme Klassen über das **Object**-Erbgut hinaus keine Handlungskompetenzen besitzen. Immerhin wird die **Object**-Methode **Equals()** intelligent überschrieben, wie das folgende Programm zeigt:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  object ob1 = new();`
`  object ob2 = new();`
`  Console.WriteLine(ob1.Equals(ob2));`
`  var ano1 = new { i = 13, d = 3.1 };`
`  var ano2 = new { i = 13, d = 3.1 };`
`  Console.WriteLine(ano1.Equals(ano2));`
`  }`
`}`
	`False`
	`True`

Während bei den Objekten `ob1` und `ob2` ein Adressvergleich das **Equals()** - Ergebnis **false** bewirkt, führt die **Equals()** - Methode der anonymen Klasse bei den Objekten `ano1` und `ano2` einen Inhaltsvergleich durch, der mit dem Ergebnis **true** endet.

An der Stelle eines Name-Wert - Paars kann im Initialisierer auch ein Variablen- oder Parametername der Umgebung stehen, der als Eigenschaftsname für die anonyme Klasse übernommen wird, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  string zf = "s";`
`  static void Main() {`
`  Prog p = new Prog();`
`  int alter = 53;`
`  var a = new { p.zf, alter };`
`  Console.WriteLine(a);`
`  }`
`}`
	`{ zf = s, alter = 53 }`

Um den Wert zu einer Eigenschaft einer anonymen Klasse festzulegen, kann man statt eines Literals auch einen Ausdruck verwenden, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`   int alter = 53;`
`   var ano = new { Name = "Otto",`
`                   Gen50plus = alter > 50 };`
`   Console.WriteLine(ano);`
`  }`
`}`
	`{ Name = Otto, Gen50plus = True }`

Ein Objekt einer anonymen Klasse dient als spontan, ohne den Aufwand einer expliziten Typdefinition erstellter Behälter für eine Anzahl von Werten.

Man verwendet ein solches nur innerhalb der erzeugenden Methode. Sollen enthaltene Daten außerhalb der erzeugenden Methode genutzt werden, dann ist zur Übergabe ein alternatives Verfahren zu verwenden (z. B. eine reguläre Klasse als Rückgabetyp).

Zwei durch anonyme Objektinitialisierer entstandene Objekte gehören nur dann zur selben Klasse, wenn bei den Initialisierungslisten die Namen und Typen der Eigenschaften sowie die Reihenfolgen übereinstimmen. In diesem Fall können die Referenzvariablen einander zugewiesen werden, z. B.:

```
var a = new {Name = "Knut", Alter = 45};
var b = new {Name = "Otto", Alter = 54};
  . . .
b = a;
```

Außerdem ist in dieser Situation ein Array mit Elementen vom Typ einer anonymen Klasse möglich, z. B.:

```
var duo = new[] {
    new { Name = "Knut", Alter = 45 },
    new { Name = "Otto", Alter = 54 }
};
```

Anonyme Klassen wurden zur Unterstützung der LINQ-Technik (*Language Integrated Query*, siehe Kapitel 19) eingeführt, werden aber auch für andere Zwecke verwendet.

6.6 Tupel

Mehrere Werte bzw. Variablen zu einem Typ zusammenzufassen und gemeinsam behandeln zu können (z. B. als Parameter oder Rückgabe einer Methode), ist eine unverzichtbare Option im Alltag der Software-Entwicklung. Selbst die älteren, strukturierten Programmiersprachen (wie C oder Pascal) bieten zu diesem Zweck Arrays und benutzerdefinierte Datenstrukturen an (siehe Abschnitt 5.1.2). In C# können Datencontainer über Arrays (vgl. Abschnitt 6.2), Klassen (siehe Kapitel 5) oder Strukturen (vgl. Abschnitt 6.1) realisiert werden. Unter den folgenden Voraussetzungen

- .NET Core (alle Versionen, inkl. 5) oder .NET Framework ab 4.7
- C# ab 7.0

ist mit den als *Strukturen realisierten Tupeln* eine neue Option mit Vorteilen gegenüber den traditionellen Lösungen verfügbar.[1]

Arrays haben den Nachteil, dass alle Elemente vom selben Typ sein müssen. Klassen und Arrays haben als Referenztypen den Nachteil, dass bei ihrer Verwendung Objekte auf dem Heap entstehen,

[1] Es gibt seit dem .NET Framework 4.0 als Klassen realisierte Tupel, die sich aber nicht bewährt haben (siehe Abschnitt 6.6.2). Wenn im Abschnitt 6.6 von *Tupeln* die Rede ist, dann sind die neuen *Struktur-Tupel* gemeint, sofern nicht explizit von *Objekt-Tupel* die Rede ist.

sodass bei einer großen Anzahl von Objekten Performanzprobleme entstehen können. Strukturen verursachen keinen Heap-Aufwand, doch haben sie (wie Klassen) folgende Nachteile:

- Es entsteht Aufwand durch die Notwendigkeit einer Definition.
- Strukturen und Klassen sind dazu vorgesehen, Daten *und* Handlungskompetenzen zu kombinieren). Ihre Verwendung als pure Datencontainer erschwert ein schnelles Verständnis der Programmlogik.

Die mit C# 7.0 eingeführten Struktur-Tupel ermöglichen die syntaktisch einfache Kreation einer Datensammlung mit Wertsemantik. Dies ist besonders nützlich bei Methoden, die mehr als einen Wert zurückliefern sollen. Die folgende Methode `AnalyzeName()` geht bei ihrem **String**-Parameter (in einer nicht praxistauglichen Vereinfachung) davon aus, dass ein Vor- und ein Nachname durch ein Leerzeichen getrennt enthalten sind. Als Rückgabe erhält der Aufrufer:

- die beiden Namensbestandteile,
- jeweils eine Beurteilung, ob es sich um ein Palindrom handelt.[1]
  Ein Palindrom besitzt in beiden Leserichtungen dieselbe Buchstabensequenz, z. B. Reittier.

Die Methode liefert die Bestandteile einzeln ab, sodass sie vom Aufrufer bequem verarbeitet werden können (siehe Abschnitt 6.6.5):

```
static (String First, String Last, bool FistPalin, bool LastPalin) AnalyzeName(String name) {
    int posSpace = name.IndexOf(" ");
    int lenLast = name.Length - (posSpace + 1);
    string fn = name.Substring(0, posSpace);
    string ln = name.Substring(posSpace + 1, lenLast);
    bool CheckPalin(string ins) {
        int len = ins.Length;
        var sb = new StringBuilder(len);
        for (int i = 0; i < len; i++)
            sb.Append(ins[len - i - 1]);
        return sb.ToString().ToUpper() == ins.ToUpper();
    }
    return (fn, ln, CheckPalin(fn), CheckPalin(ln));
}
```

Hier wird ohne nennenswerten syntaktischen Aufwand für die Rückgabe der Tupeltyp

(String First, String Last, bool FirstPalin, bool LastPalin)

definiert, der vier öffentlich zugängliche Felder mit unterschiedlichen Datentypen enthält.

Tupel eignen sich als leichtgewichtige Datencontainer:

- Es entsteht kein Heap-Aufwand, weil die Tupel als Werttypen realisiert sind.
- Die Tupeltypen können mit einer bequemen Syntax definiert werden.
- Tupeltypen besitzen nur rudimentäre, nicht erweiterbare Handlungskompetenzen, z. B.:
  - die Methoden **CompareTo()**, **Equals()** und **ToString()**)
  - die Eigenschaft **Length**

  Seit C# 7.3 sind Identitätsvergleiche mit Hilfe der Operatoren == und **!=** möglich.

Bequemlichkeit, Typsicherheit und Performanz sprechen für die Verwendung der Tupeltypen, sodass sie für den Klassen- oder Struktur-internen Einsatz sehr gut geeignet sind. Als Rückgabetypen für *öffentliche* Methoden werden die Tupeltypen der Transparenz halber jedoch *nicht* empfohlen: [2]

> Even so, they are most useful for utility methods that are `private`, or `internal`. Create user-defined types, either `class` or `struct` types when your public methods return a value that has multiple elements.

---

[1] https://de.wikipedia.org/wiki/Palindrom

[2] https://docs.microsoft.com/de-de/dotnet/csharp/tuples

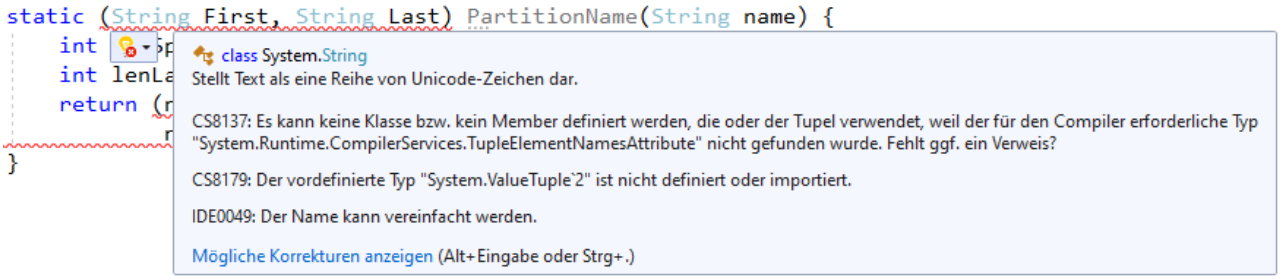
Während die Tupel für den erfahrenen C# - Entwickler eine willkommene Detailverbesserung darstellen, kann die erneute Erweiterung des Typenarsenals von Einsteigern als Informationsüberladung empfunden werden. Einsteigern wird daher empfohlen, sich vorläufig auf die im Abschnitt 6.6.5 beschriebene Verwendung von Tupeln als Rückgabetypen von Methoden zu beschränken. Diese Einsatzart ist der primäre Zweck von Tupeln (Albahari & Johannsen 2020, S. 197) und wird spontan als Verbesserung gegenüber alternativen Optionen (explizite Definition eines zusammengesetzten Rückgabetyps, **out**-Parameter) empfunden.

6.6.1 Voraussetzungen

Die bereits genannten Voraussetzungen für die Verwendung von Tupeln sind:

- .NET Core (alle Versionen, inkl. 5) oder .NET Framework ab 4.7
- C# ab 7.0

Wenn das Visual Studio bei einem .NET Framework - Projekt die Verwendung von Tupeltypen folgendermaßen kritisiert,



dann fehlt eine von den oben genannten Voraussetzungen. Sofern ein .NET Framework ab Version 4.7 installiert ist, kann das Problem leicht per Projektkonfiguration behoben werden:

Projekt > Eigenschaften > Anwendung > Zielframework

6.6.2 Tupel im CTS (Common Type System)

Das CTS (Common Type System) einer hinreichend aktuellen .NET - Implementation enthält zur Unterstützung der Tupeltypen acht generische Strukturen im Namensraum **System**:

- **public struct ValueTuple<T1>**
- **public struct ValueTuple<T1, T2>**
  . . .
- **public struct ValueTuple<T1, T2, T3, T4, T5, T6, T7>**
- **public struct ValueTuple<T1, T2, T3, T4, T5, T6, T7, TRest>**

Da wir uns noch nicht mit generischen Typen beschäftigt haben (siehe Kapitel 8), sollten Sie sich mit den durch spitze Klammern begrenzten Typformalparametern jetzt noch nicht beschäftigen. Eine Konkretisierung für den vier Felder enthaltenden Typ **ValueTuple<T1, T2, T3, T4>** ist der zu Beginn des Abschnitts 6.6 als Rückgabe der Methode `AnalyzeName()` verwendete Typ:

(String First, String Last, bool FirstPalin, bool LastPalin)

Die durch das Streichen der Feldnamen entstehende Tupel-Typbezeichnung

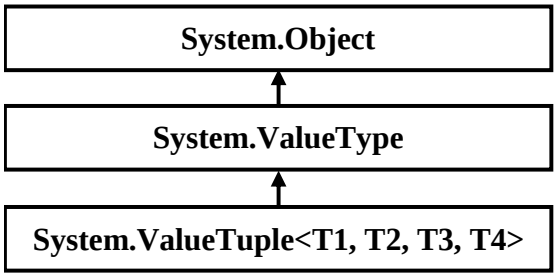
(String, String, bool, bool)

ist eigentlich nur ein Aliasname für

ValueTupel<String, String, bool, bool>

und die Feldnamen existieren nur im Quellcode, aber nicht im Übersetzungsergebnis.[1]

Hier ist der Stammbaum von **ValueTuple<T1, T2, T3, T4>** zu sehen:[2]



Die Realisation der Tupel als Strukturen hat wichtige Konsequenzen:

- Wird einer Tupel-Instanz `tia` eine andere Tupel-Instanz `tib` zugewiesen, dann erhält `tia` nicht die Adresse von `tib` , sondern eine vollständige Kopie aller Werte.
- Beim Vergleich von zwei Tupel-Instanzen durch die **Equals()** - Methode findet kein Adressvergleich statt, sondern ein Inhaltsvergleich.

Weitere Eigenschaften der Tupeltypen:

- Die Sichtbarkeit der Typen ist **public**.
- Alle Felder der Tupeltypen haben ebenfalls die Sichtbarkeit **public**.
- Die Felder der Tupeltypen erlauben Schreibzugriffe, während die Objekte einer anonymen Klasse unveränderlich sind.
- Die Felder können über ihre optional bei der Definition vergebenen Namen oder über die Standardnamen **Item1**, **Item2**, usw. angesprochen werden.
- Wie bei anderen Strukturen ist *keine* Vererbung möglich (siehe Abschnitt 6.1).

Es muss noch erwähnt werden, dass im .NET Framework ab Version 4.0 (und auch in .NET Core) auch Tupel-*Klassen* enthalten sind (Namensraum **System**):

- **public class Tuple<T1>**
- **public class Tuple <T1, T2>**
  . . .
- **public class Tuple <T1, T2, T3, T4, T5, T6, T7>**
- **public class Tuple <T1, T2, T3, T4, T5, T6, T7, TRest>**

Diese haben im Vergleich zu den Struktur-Tupeln folgende Nachteile:[3]

- Es handelt sich um Referenztypen, was sich zumindest bei Verwendung einer größeren Anzahl negativ auf die Performanz auswirken kann.
- Ihre Member können nur über die Standardnamen **Item1**, **Item2**, etc. angesprochen werden, was die Lesbarkeit des Quellcodes erschwert.
- Es fehlt die in C# 7.0 für die Wert-Tupel eingeführte bequeme Syntax.

Die Objekt-Tupel sind unveränderbar, was nicht generell als Nachteil zu werten ist. Microsoft hat sich bei der Erneuerung des Tupel-Konzepts aber für veränderbare Strukturen entschieden.

---

[1] Diese Aussage wird in Albahari & Johannsen (2020, S. 199) leicht korrigiert, wobei mit den Attributen (siehe Kapitel 14) ein weiterer noch nicht behandelter Sprachbestandteil ins Spiel kommt.

[2] https://docs.microsoft.com/de-de/dotnet/api/system.valuetuple-4

[3] https://docs.microsoft.com/de-de/dotnet/api/system.tuple

6.6.3 Variablen mit Tupeltyp deklarieren

Im ersten Beispiel deklarieren wir Variablen mit dem Tupeltyp (**String**, **String**, **int**) und verzichten dabei auf die Benennung der Felder. Die Fähigkeiten des Compilers zur Typinferenz ausnutzend kann man den Typbezeichner durch das Schlüsselwort **var** ersetzen:

Quellcode	Ausgabe

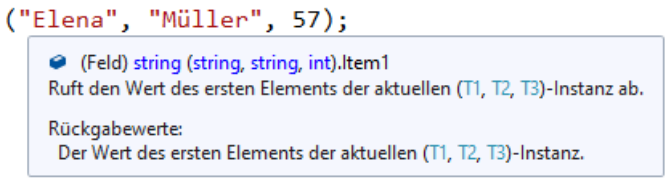
`using System;`
`class Prog {`
`  static void Main() {`
`  (String, String, int) ano1 = ("Elena", "Müller", 57);`
`  var ano2 = ("Otto", "Meyer", 34);`
`  Console.WriteLine(ano2.Item1);`
`  }`
`}`
	`Otto`

Um beim Feldzugriff nicht auf die Standardnamen **Item1**, **Item2**, usw. angewiesen zu sein, sollten für Tupeltypen explizite Feldnamen vereinbart werden, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  (String Vorname, String Nachname, int Alter) named = ("Elena", "Müller", 57);`
`  (String, String, int) unnamed = (Vorname: "Otto", Nachname: "Meyer", Alter:34);`
 	
`  Console.WriteLine(named.Vorname);`
`  Console.WriteLine(unnamed.Item2);`
`  }`
`}`
	`Elena`
	`Meyer`

Für die Variable `named` wird ein Tupeltyp mit benannten Feldern explizit vereinbart. Auf der rechten Seite der Zuweisung steht ein Tupeltyp mit unbenannten Feldern und Werten mit passenden Typen, sodass die Zuweisung gelingt:



Es resultiert eine Variable mit den deklarierten Feldnamen und den zugewiesenen Werten.

Für die Variable `unnamed` wird ein Tupeltyp vereinbart mit Feldern, die einen expliziten Typ erhalten, aber keinen Namen. Durch die im Tupel-Literal auf der rechten Seite der Zuweisung vorhandenen Feldnamen wird der unbenannte Zustand aus der Variablendeklaration von `unnamed` nicht geändert. Ersetzt man die Tupel-Definition mit expliziten Typangaben durch das Schlüsselwort **var**, sodass der Compiler zur Typinferenz gezwungen wird, dann kommt es zur Übernahme der Feldnamen, z. B.:

```
var named2 = (Vorname: "Otto", Nachname: "Meyer", Alter: 34);
Console.WriteLine(named2.Vorname);
```

Man kann sich darüber streiten, ob die expliziten Feldnamen in einem Tupeltyp das Pascal- oder das Camel-Casing verwenden sollten. Weil es sich um implizit **public** deklarierte Felder einer Struktur handelt, sollten die Namen m.E. mit einem Großbuchstaben beginnen. Die Standardnamen **Item1**, **Item2**, usw. sind konsistent zu diesem Vorschlag.

Wird die seit C# 7.1 bestehende Option genutzt, den Feldnamen für eine Tupeltyp-Instanz von einer bereits deklarierten Variablen elementaren Typs zu übernehmen, dann endet die Konsistenz, weil es

sich oft um eine lokale Variable handelt. Dann wird der mit einem kleinen Buchstaben startende Name einer lokalen Variablen zum Namen eines **public**-Feldes einer Tupeltyp-Instanz, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  int alter = 34;`
`  var named = (Vorname: "Otto", Nachname: "Meyer", alter);`
`  Console.WriteLine(named.alter);`
`  var named2 = (Vorname: "Otto", Nachname: "Meyer", Alter: alter);`
`  Console.WriteLine(named2.Alter);`
`  }`
`}`
	34
	34

Es ist allerdings erlaubt und sinnvoll, den übernehmbaren Namen durch einen explizit definierten Feldnamen zu dominieren (siehe Variable `named2`).

Die höhere Funktionalität der Tupeltypen im Vergleich zu den anonymen Typen, die ebenfalls spontan (ohne explizite Typdefinition) als Container für Daten mit unterschiedlichen Typen verwendbar sind (siehe Abschnitt 6.5), zeigt sich in der Eignung der Tupeltypen zur Konkretisierung von generischen Typen, z. B.:

```
List<(string Name, int Alter)> mitglieder;
```

Wie schon im Abschnitt 6.6.2 erläutert wurde, ist im IL - Übersetzungsergebnis (Intermediate Language) von den explizit vergebenen Feldnamen für Tupeltypen nichts zu sehen. Der Compiler ersetzt sie durch die Standardnamen **Item1**, **Item2**, usw.[1] Im Quellcode vorhandene anonyme Klassen werden vom C# - Compiler in neu definierte IL-Klassen übersetzt, wobei selbstverständlich die Eigenschafts- bzw. Feldnamen aus dem Quellcode Verwendung finden. Demgegenüber resultieren aus den Tupeltypen im Quellcode konkretisierte generische Strukturen, und diese enthalten die unveränderlichen Feldnamen **Item1**, **Item2**. Solange keine ambitionierten Programmiertechniken wie die Reflektion zum Einsatz kommen, spüren die Programmierer aber nichts von dieser Einschränkung.

6.6.4 Zuweisungen mit Tupel-Syntax

Eine Tupeltyp-Instanz kann einer Tupeltyp-Variablen zugewiesen werden, wenn ...

- gleich viele Felder vorhanden sind
- alle Feldtypen auf der rechten Seite der Zuweisung erweiternd in den zugehörigen Typ auf der linken Seite konvertiert werden können. Die Feldnamen spielen bei der Zuweisungskompatibilität keine Rolle.

Im folgenden Codesegment übernimmt die Variable `sint` vom rechten Teil der Zuweisung den Tupeltyp mit benannten Feldern:

```
var sint = (Vorname: "Otto", Alter: 34);
(string Vorname, long Alter) slong = ("Otto", 34);
```

Das zweite Feld von `sint` hat den erschlossenen Typ **int**, während das zweite Feld von `slong` den deklarierten Typ **long** hat, sodass die folgende Zuweisung scheitert:

```
sint = slong;
```

[1] https://docs.microsoft.com/de-de/dotnet/csharp/tuples

Im nächsten Beispielprogramm werden die beiden lokalen Variablen `ganz` und `gk` mit den elementaren Datentypen **int** bzw. **double** per Tupel-Syntax deklariert und initialisiert, wobei optional über das Schlüsselwort **var** die Typinferenz durch den Compiler in Anspruch genommen werden kann:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  (int ganz, double gk) = (13, 47.11);`
`  Console.WriteLine(ganz);`
`  (var ganzTi, var gkTi) = (13, 47.11);`
`  Console.WriteLine(ganzTi);`
 	
`  int ivar;`
`  double dvar;`
`  (ivar, dvar) = (13, 47.11);`
`  Console.WriteLine(dvar);`
`  }`
`}`
	`13`
	`13`
	`47,11`

Anschließend erhalten die bereits deklarierten Variablen `ivar` und `dvar` per Tupel-Syntax einen neuen Wert.

In allen Zuweisungen werden die beteiligten Tupel-Literale (z. B.: `(13, 47.11)`) zerlegt, und ihre Feldinhalte landen in einfachen lokalen Variablen. Man spricht hier von einer *Dekonstruktion*.

Gelegentlich sind bei der Zerlegung einer Tupeltyp-Instanz nicht alle Felder von Interesse. Dann kann per Unterstrich ein Feld ausgelassen werden, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`    static void Main() {`
`        string first;`
`         (first, _, _) = ("Otto", "Meyer", 34);`
`        Console.WriteLine(first);`
`    }`
`}`
	`Otto`

Dabei kommt die seit C# 7.0 zugelassene Ausschuss-Variable (engl.: *discard variable*) zum Einsatz, die wir im Zusammenhang mit **out**-Parametern kennengelernt haben (vgl. Abschnitt 5.3.1.3.2.2).

6.6.5 Tupel als Rückgabetypen von Methoden

Wie zu Beginn von Abschnitt 6.6 erwähnt, ist die Verwendung von Tupeltypen für die Rückgabe von Methoden besonders attraktiv. Die bereits vorgestellte Methode `AnalyzeName()` geht bei ihrem **String**-Parameter (in einer nicht praxistauglichen Vereinfachung) davon aus, dass ein Vor- und ein Nachname durch ein Leerzeichen getrennt enthalten sind. Für die beiden Namensbestandteile liefert die Methode:

- eine Zeichenfolge
- die Information, ob es sich um ein Palindrom handelt

Der Aufrufer erhält eine Instanz einer Konkretisierung der passenden generischen **Tuple**-Struktur. Er kann sie komplett verarbeiten oder auf einzelne Felder zugreifen, z. B.:

Quellcode	Ausgabe

```
using System;
using System.Text;

class Prog {
  static (string First, string Last, bool FirstPalin, bool LastPalin)
         AnalyzeName(string name) {
    int posSpace = name.IndexOf(" ");
    int lenLast = name.Length - (posSpace + 1);
    string fn = name.Substring(0, posSpace);
    string ln = name.Substring(posSpace + 1, lenLast);
    bool CheckPalin(string ins) {
      int len = ins.Length;
      var sb = new StringBuilder(len);
      for (int i = 0; i < len; i++)
        sb.Append(ins[len - i - 1]);
      return sb.ToString().ToUpper() == ins.ToUpper();
    }
    return (fn, ln, CheckPalin(fn), CheckPalin(ln));
  }

  static void Main() {
    var an = AnalyzeName("Otto Rentner");
    Console.WriteLine(an.First);
    var (first, last, firstPal, lastPal) = AnalyzeName("Otto Rentner");
    Console.WriteLine(firstPal);
  }
}
```

```
Otto
True
```

Beim ersten Aufruf von `AnalyzeName()` wird die Rückgabe in der lokalen Variablen `an` mit dem Tupeltyp

(string, string, bool, bool) bzw. **ValueTuple<string, string, bool, bool>**

abgelegt. Anschließend kann z. B. der Vorname über das öffentliche Feld `First` in der Variablen `an` angesprochen werden.

Im zweiten Aufruf von `AnalyzeName()` wird die im Abschnitt 6.6.4 beschriebene Dekonstruktion benutzt, um die Rückgabe auf einzelne lokale Variablen zu verteilen.

Verzichtet man in einer Methodendefinition mit Tupel-Rückgabetyp auf eine Benennung der Felder, dann müssen die Felder später über die Standardnamen **Item1**, **Item2**, usw. angesprochen werden, z. B.:

Quellcode	Ausgabe

```
using System;
using System.Text;

class Prog {
  static (string, string, bool, bool) AnalyzeName(string name) {
    . . .
  }

  static void Main() {
    var an = AnalyzeName("Otto Rentner");
    Console.WriteLine(an.Item1);
    Console.WriteLine(AnalyzeName("Otto Rentner").Item3);
  }
}
```

```
Otto
True
```

Man kann zwar auch mit **ref**- oder **out**-Parametern (siehe Abschnitt 5.3.1.3.2) mehrere Werte an den Aufrufer übertragen, verliert dabei aber z. B. ...

- die Möglichkeit, die erhaltenen Werte als Felder einer Instanz gemeinsam zu verarbeiten,
- syntaktische Eleganz.

6.6.6 Dekonstruktion für selbst definierte Typen

Eine Möglichkeit zur bequemen Übertragung von Instanzeigenschaften und -feldern in einzelne Variablen nach dem Muster der im Abschnitt 6.6.4 beschriebenen Dekonstruktion kann man auch für selbst definierte Typen realisieren. Man definiert eine öffentliche Methode namens **Deconstruct()** (bei Bedarf auch in mehreren Überladungen) und verwendet dabei **out**-Parameter für die auszuliefernden Instanzeigenschaften und -felder. Das folgende Beispiel wurde weitgehend unverändert aus der C# - Online-Dokumentation von Microsoft übernommen:[1]

```
using System;

public class Person {
    public string FirstName { get; }
    public string LastName { get; }

    public Person(string first, string last) {
        FirstName = first;
        LastName = last;
    }

    public void Deconstruct(out string firstName, out string lastName) {
        firstName = FirstName;
        lastName = LastName;
    }
}

class Prog {
    static void Main() {
        Person p = new Person("Otto", "Meyer");
        var (first, last) = p;
        Console.WriteLine(first);
    }
}
```

Als Ausgabe resultiert:

```
Otto
```

Eine **Deconstruct**-Methode ist strikt zu unterscheiden vom Finalisierer einer Klasse, der ggf. kurz vor der Beseitigung eines obsolet gewordenen Objekts vom Garbage Collector aufgerufen wird (siehe Abschnitt 5.4.4).

[1] https://docs.microsoft.com/de-de/dotnet/csharp/deconstruct

6.6.7 Identitätsvergleiche

Seit C# 7.3 sind für Tupeltyp-Instanzen Identitätsvergleiche mit Hilfe der Operatoren == und != möglich, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  var tti1 = (G: 13, D: 13.0);`
`  var tti2 = (13, 13.0);`
`  var tti3 = (13, 13);`
`  Console.WriteLine(tti1 == tti2);`
`  Console.WriteLine(tti2 == tti3);`
`  }`
`}`
	`True`
	`True`

Regeln:

- Explizite Feldnamen spielen beim Vergleich keine Rolle.
- Bei Bedarf werden erweiternde Typanpassungen vorgenommen.

6.7 Records

Insbesondere in Programmen mit mehreren Threads (Ausführungsfäden), die gemeinsame Daten verwenden, sind unveränderliche Objekte von Vorteil, weil die Thread-Synchronisation entfällt. Die in C# 9.0 bzw. .NET 5.0 eingeführten Records ...

- vereinfachen die Erstellung von *unveränderlichen* Referenztypen,
- bieten eine *inhaltsorientierte* Gleichheitsprüfung durch synthetisierte (vom Compiler erstellte) Methoden
- und unterstützen die *Vererbung*.

In der Bezeichnung *Record* (dt.: *Datensatz*) steckt eine klare Anwendungsempfehlung für den neuen Referenzdatentyp im Kontrast zur traditionellen Klasse. Wir haben uns seit dem Kapitel 1 mit der Vorstellung vertraut gemacht, dass Objekte über Kompetenzen, einen in ständiger Wandlung begriffenen Zustand und einen Verantwortungsbereich besitzen. Dies ist eine grundlegende Vorstellung im objektorientierten Programmier-Paradigma. Allerdings befinden sich im Aufgabenbereich mancher Programme auch Objekte, die eher als Datensatz denn als aktives Individuum zu beschreiben sind. Wenn dann ....

- Objekte (mit Heap-orientierter Speicherverwaltung und Vererbung) gegenüber Struktur-Instanzen zu bevorzugen sind,
- die Unveränderlichkeit der Objekte auf einfache Weise sichergestellt werden soll,
- und außerdem eine Inhalts-orientierte Gleichheitsprüfung gewünscht ist,

dann ist ein Record-Datentyp die beste Lösung.

6.7.1 Definition

6.7.1.1 Traditionelle oder positionsorientierte Eigenschaftsdeklaration

Die typischen Record-Member sind get-only - Eigenschaften. Wenn diese auf traditionelle Weise deklariert werden, dann unterscheidet sich eine Datensatzdefinition von einer analogen Klassende-

finition nur durch die Verwendung des Schlüsselworts **record** statt **class**. Das folgende Beispiel wurde im Wesentlichen von einer Microsoft-Webseite übernommen:[1]

```
public record Person {
   public string LastName { get; }
   public string FirstName { get; }

   public Person(string first, string last) {
      FirstName = first;
      LastName = last;
   }
}
```

Bei der Record-Definition kann alternativ zur traditionellen Eigenschaftsdeklaration eine an Konstruktordefinitionen erinnernde, als *positionsorientiert* bezeichnete Syntax verwendet werden (engl.: *positional record*). Im Beispiel kommt man dann mit einer Zeile aus:

```
public record Person(string FirstName, string LastName);
```

Die Eigenschaftsdeklarationen und den Konstruktor (mit einem Parameter für jede Eigenschaft) erstellt der Compiler automatisch. Zusätzlich generiert er über die für *alle* Record-Typen synthetisierten (automatisch erstellten) Methoden hinaus für Records mit positionsorientierter Eigenschaftsdeklaration auch noch eine **Deconstruct()** - Methode (siehe Abschnitt 6.7.2).

In einer Record-Definition darf man (bei traditioneller und bei positionsorientierter Eigenschaftsdeklaration) selbstverständlich geerbte Methoden ändern (siehe Kapitel 7 zum Unterschied zwischen dem Ersetzen und dem Überschreiben von geerbten Methoden) sowie zusätzliche Methoden definieren.

Es ist auch möglich, einen Record-Typ mit *änderbaren* Eigenschaften auszustatten, z. B.:

```
public record Person {
    public string LastName { get; }
    public string FirstName { get; }
    public int NoOfVisits { get; set; }

    public Person(string first, string last, int nov) {
        FirstName = first;
        LastName = last;
        NoOfVisits = nov;
    }
}
```

Da Records eingeführt wurden, um die Erstellung von *unveränderlichen* Objekten zu erleichtern, sind änderbare Eigenschaften sowie Methoden zur Änderung von Record-Instanzen allerdings nur in Ausnahmefällen sinnvoll.

Die in C# 9.0 eingeführten init-only - Eigenschaften (siehe Abschnitt 5.5.2) sind auch bei Records sehr nützlich, z. B.:

---

[1] https://docs.microsoft.com/de-de/dotnet/csharp/whats-new/csharp-9

```
public record Person {
    public string LastName { get; }
    public string FirstName { get; }
    public int YearOfBirth { get; init; }

    public Person(string first, string last) {
        FirstName = first;
        LastName = last;
    }
}
```

Ihr Wert kann nach einer Initialisierung per Konstruktor oder Objektinitialisierer nicht mehr geändert werden.

Im Vergleich zu einem Feld mit **readonly**-Modifikator (vgl. Abschnitt 5.2.5), das in einer Record-Definition ebenfalls erlaubt ist, hat eine init-only - Eigenschaft u. a. den Vorteil, dass die einmalige Wertzuweisung auch per Objektinitialisierer erfolgen kann, sodass zur Initialisierung kein Konstruktorparameter erforderlich ist.

Im folgenden Beispiel wird für die erste von zwei `Person`-Instanzen die Eigenschaft `YearOfBirth` per Objektinitialisierer auf einen gültigen Wert gesetzt, während die zweite Instanz den Wert 0 behält, der einen fehlenden Wert signalisiert:

```
class Prog {
    static void Main() {
        var p1 = new Person("Luise", "Schmit") { YearOfBirth = 2002 };
        var p2 = new Person("Otto", "Rempremerding");
        Console.WriteLine(p1);
        Console.WriteLine(p2);
    }
}
```

In der Ausgabe des Programms ist u. a. das Ergebnis der vom Compiler für den Record-Typ `Person` automatisch erstellten **ToString()** - Überschreibung zu sehen, die von **WriteLine()** aufgerufen wird:

```
Person { LastName = Schmit, FirstName = Luise, YearOfBirth = 2002 }
Person { LastName = Rempremerding, FirstName = Otto, YearOfBirth = 0 }
```

6.7.1.2 Vererbung

Bei der Beschreibung der Besonderheiten von Record-Typen im Zusammenhang mit der Vererbung werden einige, vermutlich leicht verdaulichen Vorgriffen auf das Kapitel 7 in Kauf genommen:

- Wird in einer Record-Definition kein Basistyp angegeben, dann stammt der neue Typ von **Object** ab.
- Alternativ kann in einer Record-Definition ein vorhandener Record-Typ als Basis angegeben werden, z. B.:

  ```
  public record Teacher : Person {
      public string Subject { get; }
      public Teacher(string first, string last, string sub) : base(first, last) {
          Subject = sub;
      }
  }
  ```

- Aus einem Record-Typ kann nur ein anderer Record-Typ abgeleitet werden.
- Mit dem Modifikator **sealed** lässt sich für einen Record-Typ die Definition von Ableitungen verhindern, z. B.:

  ```
  public sealed record Student : Person { ... }
  ```

6.7.2 Vom Compiler erstellte Methoden

Der Compiler erstellt zu einem Record-Typ mehrere Methoden, die zur inhaltsbasierten Gleichheitsprüfung und für andere Zwecke geeignet sind, wobei aber im Quellcode vorhandene, Signatur-gleiche Methoden *nicht* ersetzt werden:

- Eine Überschreibung der **Object**-Methode **Equals()**:

  public override bool Equals(object *obj***)**

  Zwei Variablen mit Record-Typ werden von der Methode **Equals()** und auch vom überladenen Operator == (siehe unten) als gleich beurteilt, wenn ...
  - sie auf Objekte vom selben Typ zeigen
  - alle Eigenschaftsausprägungen identisch sind

  Für zwei Objekte von einem Record-Typ wird also eine inhaltsbasierte Gleichheitsprüfung durchgeführt.
- Eine **Equals()** - Überladung mit einem Parameter vom eigenen Typ, die z. B. beim Record-Typ `Person` den folgenden Definitionskopf besitzt:

  public virtual bool Equals(`Person` *other***)**

  Weil ein Record-Typ eine solche Methode besitzt, kann er von sich behaupten, die generische Schnittstelle **System.IEquatable<T>** zu implementieren (siehe Kapitel 8 zu generischen Typen und Kapitel 9 zu Schnittstellen).
- Eine Überschreibung der **Object**-Methode **GetHashCode()**:

  public override int GetHashCode()

  Die Methode **GetHashCode()** ist z. B. relevant beim Befüllen eines Kollektionsobjekts vom Typ **HashSet<T>** (siehe Abschnitt 11.4.1). Sie muss mit der **Equals()** - Methode kompatibel sein. Sind z. B. zwei Objekte gleich im Sinne der **Equals()** - Methode, dann müssen sie bei einem **GetHashCode()** - Aufruf denselben Wert liefern.
- Auf der **Equals()** - Methode basierende Überladungen für die Operatoren == und **!=** (vgl. Abschnitt 5.7.2).
- Eine Überschreibung der **Object**-Methode **ToString()**, die für eine informative Selbstdarstellung sorgt (siehe Beispiel im Abschnitt 6.7.1.1):

  public override void ToString()

Wird bei der Record-Definition wie im folgenden Beispiel

```
public record Person(string FirstName, string LastName);
```

für Eigenschaften die positionsorientierte Deklaration verwendet (vgl. Abschnitt 6.7.1.1), dann erstellt der Compiler über die oben erwähnten Methoden hinaus eine **Deconstruct()** - Methode (vgl. Abschnitt 6.6.6), die einen **out**-Parameter für jede öffentliche Eigenschaft des Record-Typs besitzt, z. B.:

```
Person p = new Person("Otto", "rempremerding");
var (first, last) = p;
```

6.7.3 Kopierkonstruktoren und with-Ausdrücke

Für jeden Record-Typ existiert ein sogenannter *Kopierkonstruktor (*engl. *copy constructor*), den nötigenfalls der Compiler definiert.[1] Der Kopierkonstruktor hat einen Parameter vom eigenen Typ und liefert eine flache Kopie des Parameterobjekts. Im folgenden Beispiel wird der Record-Typ `Person` aus dem Abschnitt 6.7.1 erweitert:

```
public record Dog(string Name, string Breed);

public record Person(string FirstName, string LastName, Dog Dog) {
    public Person(Person p) {
        FirstName = p.FirstName;
        LastName = p.LastName;
        Dog = p.Dog;
    }
}
```

Es wird die Eigenschaft `Dog` vom gleichnamigen Record-Typ sowie ein explizit definierter und als **public** deklarierten Kopierkonstruktor ergänzt. Weil der Kopierkonstruktor als **public** deklariert ist, kann er auch in der Klasse `Prog` verwendet werden:

```
class Prog {
    static void Main() {
        Person p1 = new Person("Otto", "Rempremerding", new Dog("Emma", "Mix"));
        var p2 = new Person(p1);
        Console.WriteLine(p2.Dog.Name);
    }
}
```

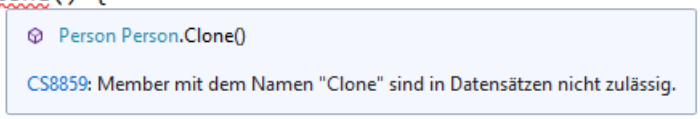
Wie das Beispiel zeigt, geht bei einem Record-Typ der vom Compiler ergänzte Standardkonstruktor durch die Definition eines Kopierkonstruktors *nicht* verloren.

Ein vom Compiler erstellter Kopierkonstruktor hätte im Beispiel den Zugriffsmodifikator **protected**, wäre aber ansonsten mit dem explizit erstellten Kopierkonstruktor identisch.

Solange die Eigenschaften eines Records einen Werttyp oder einen unveränderlichen Referenztyp (z. B. **String**) besitzen, ist die flache Kopie mit der tiefen Kopie äquivalent.

Wer in einem Record eine eigene Methode namens **Clone()** (z. B. zur Erstellung einer tiefen Kopie) erstellen möchte, der scheitert an einer Fehlermeldung des Compilers:

```
public Person Clone() {
```

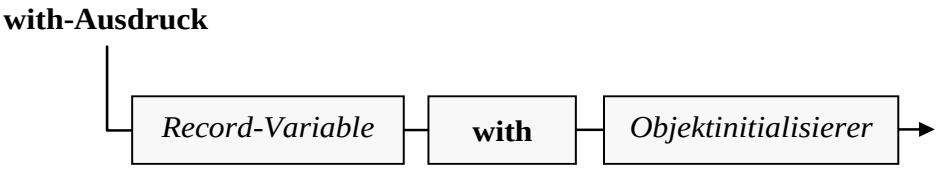


Das liegt letztlich an einer vom Compiler für jeden Record-Typ automatisch erstellten öffentlichen Methode mit dem ungewöhnlichen Namen **<Clone>$**. Weil dieser Name in C# verboten, in der IL aber erlaubt ist, kann die Methode vom Compiler benutzt, im Quellcode aber *nicht* verwendet werden.

Wird in einem Programm bevorzugt mit unveränderlichen Objekten gearbeitet, dann ist es oft erforderlich, zu einer Instanz eine Variante mit leicht abweichenden Eigenschaften zu erstellen. Für Record-Typen wird diese sogenannte *nicht-destruktive Mutation* (engl.: *non-destructive mutation*) in C# seit der Version 9.0 durch **with**-Ausdrücke unterstützt:

---

[1] Regeln für die Modifikatoren zu einem explizit oder implizit definierten Kopierkonstruktor sind hier zu finden:
- https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-9
- https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/records



Es wird per Kopierkonstruktor eine neue Record-Instanz erstellt und anschließend per Objektinitialisierer (siehe Abschnitt 5.4.3.3) modifiziert. Im folgenden Beispiel wird von der **Record**-Instanz `p1` eine Kopie erstellt, wobei mit Ausnahme des Nachnamens alle Eigenschaftsausprägungen beibehalten werden:

```
class Prog {
    static void Main() {
        Person p1 = new Person("Otto", "Rempremerding",  new Dog("Emma", "Mix"));
        var p2 = p1 with { LastName = "Brgl" };
        Console.WriteLine(p2.Dog.Name);
    }
}
```

Im Objektinitialisierer können natürlich auch *mehrere* Eigenschaften einen neuen Wert erhalten.

Es ist erlaubt und oft sinnvoll, *keine* zu verändernde Eigenschaft anzugeben, z. B.:

```
var p3 = p1 with { };
```

Dies ermöglicht die implizite Nutzung eines vom Compiler automatisch erstellten Kopierkonstruktors, der aufgrund seines Zugriffsmodifikators **protected** ansonsten nur in Methoden vom selben oder von einem abgeleiteten Record-Typ nutzbar wäre.

6.8 Ein RSS-Feed-Reader zur Motivationsstärkung

Eventuell waren die letzten Abschnitte nicht für alle Leser vergnüglich und motivationsfördernd. Damit kein Handtuch fliegt, schieben wir einen Abschnitt ein, der hoffentlich Entspannung bringt und neue Motivation zuführt. Wir erstellen in Anlehnung an einen Artikel von Hajo Schulz in der Computer-Zeitschrift c't (Ausgabe 2010.13, S. 138f) ein Anzeigeprogramm für RSS-Feeds mit frei wählbarer Adresse:

Als *RSS-Feed* (*Really Simple Syndication*) bezeichnet man eine im Internet angebotene und per URL (*Uniform Resource Locator*) ansprechbare Datei, die neue Beiträge zu einem Thema kurz beschreibt und jeweils einen Link zur Vollinformation bietet.[1] Das RSS-Dateiformat ist XML-basiert (*eXtensible Markup Language*) und hat (seit 2002) die aktuelle Version 2.0.

Eine RSS-Datei der Version 2.0 hat den folgenden Aufbau:[2]

```
<rss version="2.0">
  <channel>
    <item>
      <title>Titel des Eintrags</title>
      <description>Kurze Zusammenfassung des Eintrags</description>
      <link>Link zum vollständigen Eintrag</link>
      <author>Autor des Artikels, E-Mail-Adresse</author>
      <guid>Eindeutige Identifikation des Eintrages</guid>
      <pubDate>Datum des Items</pubDate>
    </item>
    <item>
      . . .
    </item>
  </channel>
</rss>
```

Von Interesse sind vor allem die `<item>` - Elemente, die jeweils einen Beitrag beschreiben und durch unser Programm formatiert aufgelistet werden sollen.

Zwar haben der Google-Browser Chrome und der Mozilla-Browser Firefox die Unterstützung für RSS-Feeds eingestellt, doch werden im Internet nach wie vor Millionen von attraktiven RSS-Feeds angeboten, sodass sich die Entwicklung eines RSS-Feed - Readers lohnt.

6.8.1 Projekt anlegen mit Vorlage *WPF - Anwendung*

Legen Sie über

Datei > Neu > Projekt

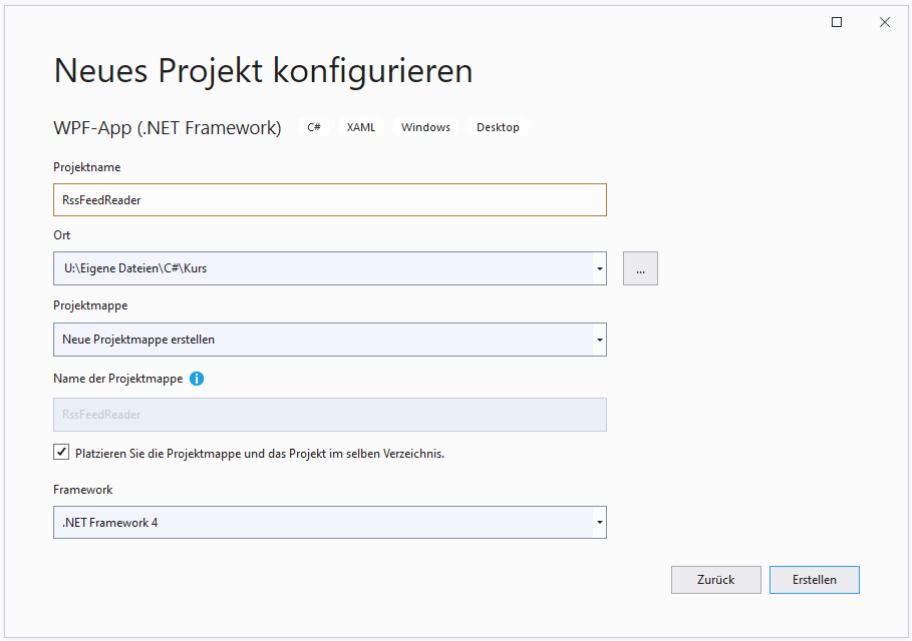
ein neues Projekt basierend auf der Vorlage **C# WPF - App (.NET Framework)**[3] mit dem Namen `RssFeedReader` an:

---

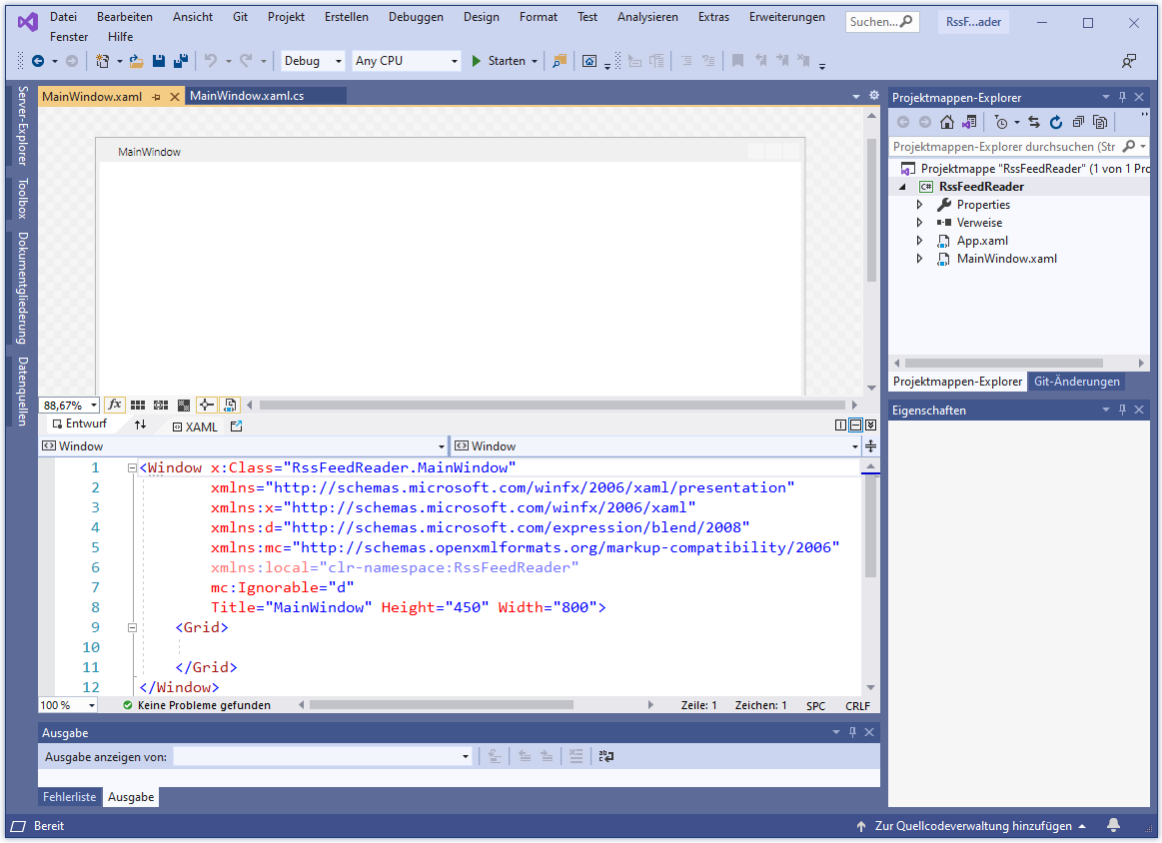
[1] Laut Wikipedia (https://de.wikipedia.org/wiki/Content-Syndication) ist mit *Content-Syndication* im Internet das Zusammenführen der Informationen von verschiedenen Webseiten gemeint.

[2] https://de.wikipedia.org/wiki/RSS_(Web-Feed)

[3] Im Kurs wird das GUI-Design (*Graphical User Interface*) per WPF (*Windows Presentation Foundation*) gegenüber den Alternativen WinForms (veraltet) und UWP (*Universal Windows Platform*, unsichere Zukunft) bevorzugt (zu Details siehe Kapitel 12).



Nach einem Mausklick auf **Erstellen** präsentiert die Entwicklungsumgebung im WPF- bzw. XAML-Designer einen Rohling für das Fenster der entstehenden Anwendung:



Wir gestalten mit dem WPF-Designer die Bedienoberfläche unseres Programms und übernehmen dazu konfigurierbare Komponenten (Steuerelemente) aus der **Toolbox** (siehe unten). Wie gleich zu sehen sein wird, definieren wir dabei die neue Klasse `MainWindow`, die von der BCL-Klasse **Window** abgeleitet wird.

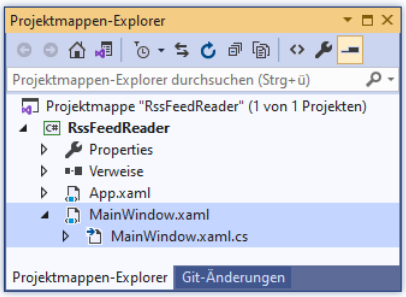
Wie Sie bereits aus dem Abschnitt 5.12 wissen, besteht ein zentrales Merkmal der WPF-Technologie darin, das GUI-Design einer Anwendung durch eine XML-Spezialisierung namens

XAML (*eXtensible Application Markup Language*) zu deklarieren. Zu unserem Anwendungsfenster gehört die XAML-Datei **MainWindow.xaml**, die im unteren Teil der Designer-Zone erscheint und initial so aussieht:

```
<Window x:Class="RssFeedReader.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:RssFeedReader"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>

    </Grid>
</Window>
```

Das **Window**-Wurzelelement definiert ein Anwendungsfenster, also das Erscheinungsbild eines Objekts aus der Klasse `MainWindow`. Die initialen Bestandteile des **Window**-Elements wurden schon im Abschnitt 5.12.2 erläutert. Der grafische Fenster-Designer ist ein Werkzeug zur bequemen Bearbeitung der XAML-Datei. Manchmal wird es sich aber als praktisch erweisen, den XAML-Code direkt zu editieren.

Zur Definition der Klasse `MainWindow` trägt auch eine Quellcode-Datei namens **MainWindow.xaml.cs** bei, für die wir als Entwickler verantwortlich sind. Der Projektmappen-Explorer zeigt die XAML- und die zugehörige Quellcode-Datei:



Außerdem trägt zur Definition der Klasse `MainWindow` noch eine zweite C# - Quellcodedatei bei, die vom Visual Studio aufgrund der GUI-Deklaration in der Datei **MainWindow.xaml** im Hintergrund gepflegt wird.

Über die Fensterklasse `MainWindow` hinaus benötigt unsere WPF-Anwendung auch noch eine Anwendungsklasse. Diese stammt von der BCL-Klasse **Application** im Namensraum **System.Windows** ab und trägt im Beispiel den Namen `App`. Analog zur Fensterklasse sind eine XAML-Deklarationsdatei (**App.xaml**) und eine C# - Quellcodedatei mit einer partiellen Klassendefinition (**App.xaml.cs**) beteiligt. Beim geplanten Beispielprogramm müssen wir uns um diese beiden Dateien *nicht* kümmern. Wer die XAML-Datei **App.xaml** neugierig per Doppelklick auf ihren Eintrag im Projektmappen-Explorer öffnet,

```
<Application x:Class="RssFeedReader.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:RssFeedReader"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

findet im **Application**-Wurzelelement u. a. die die folgenden Attribute:

- **x:Class**
  Es nennt die zugehörige Klasse `App` samt Namensraum `RssFeedReader`.
- **StartupUri**
  Es nennt die XAML-Datei zum Fenster, das beim Programmstart angezeigt werden soll.

6.8.2 Steuerelemente aus der Toolbox übernehmen

Holen Sie nötigenfalls im Editor die Datei **MainWindow.xaml** in den Vordergrund, und öffnen Sie das **Toolbox**-Fenster mit dem Menübefehl

Ansicht > Toolbox

oder per Mausklick auf die **Toolbox**-Schaltfläche am linken Fensterrand. Erweitern Sie nötigenfalls im **Toolbox**-Fenster die Liste mit den **Häufig verwendeten WPF-Steuerelementen**, und erstellen Sie auf dem Formular folgende Steuerelemente:

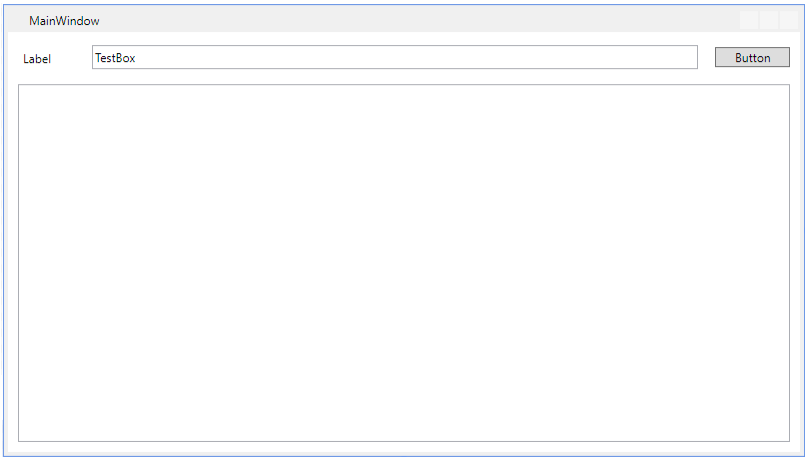
- ein **Label**-Objekt
  Es soll den im Texteingabefeld benötigten Inhalt beschreiben (*Feed-URL:*).
- ein **TextBox**-Objekt
  Hier können die Benutzer die Feed-Adresse eintragen.
- ein **Button**-Objekt
  Damit fordern die Benutzer das Laden einer RSS-Datei an.
- ein **ListBox**-Objekt
  Hier werden die RSS-Items formatiert angezeigt.

Die Übernahme eines Steuerelements aus der **Toolbox** gelingt ...

- per Doppelklick auf den jeweiligen **Toolbox**-Eintrag
- per Drag & Drop (Ziehen und Ablegen)

6.8.3 Positionen, Größen und sonstige Eigenschaften der Steuerelemente

Nun können Sie wie in einem Grafikprogramm die Positionen und Größen der Fensterbestandteile verändern, um das gewünschte Layout zu erzielen, z. B.:



Für alle Positionen und Größen wird im WPF-Designer die Maßeinheit **DIP** verwendet (*Device Independent Pixel*, dt.: *geräteunabhängige Pixel*). Ein DIP hat eine Breite und Höhe von 1/96 Zoll, sodass 96 DIP gerade einem Zoll (= 2,54 cm) entsprechen. Bei einer Bildschirmauflösung von 96 DPI (*Dots Per Inch*) ist ein geräteunabhängiges Pixel gerade genauso groß wie ein physisches Pixel. Positionen und Größen werden in Variablen vom Typ **double** gespeichert.

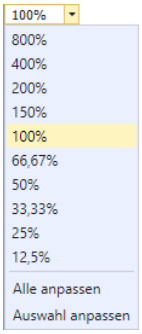
6.8.3.1 Arbeitshilfen

Wir erweitern unser Wissen über die vom WPF-Designer angebotenen Arbeitshilfen im Vergleich zum Abschnitt 5.12, ignorieren aber weiterhin die Bedienhilfen mit Bezug zu der noch nicht behandelten WPF-Containertechnik.

6.8.3.1.1 Zoom-Stufe

Über die Symbolleiste zwischen Entwurfs- und XAML-Ansicht

lässt sich eine Zoomstufe wählen:



Bei gedrückter **Strg**-Taste kann man die Zoom-Stufe auch per Mausrad ändern.

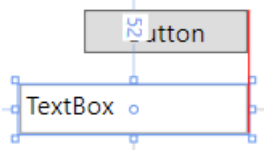
6.8.3.1.2 Andocken an Ausrichtungslinien

Wenn über die Symbolleiste zwischen Entwurfs- und XAML-Ansicht

das **Andocken an den Ausrichtungslinien** aktiviert ist, dann erscheint eine rote Linie, ...

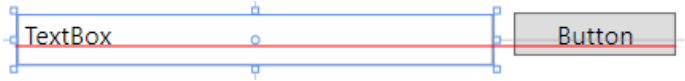
- wenn die *Ränder* von zwei Steuerelementen vertikal

oder horizontal



ausgerichtet sind,

- oder wenn die *Textbasislinien* von zwei Steuerelementen vertikal ausgerichtet sind:



6.8.3.1.3 Rasterpositionen und -linien

Wenn über die Symbolleiste zwischen Entwurfs- und XAML-Ansicht

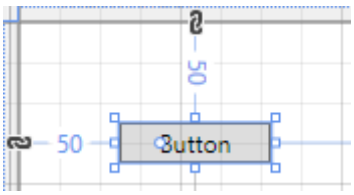
das **Ausrichten an den Rasterlinien** aktiviert ist, dann wird ...

- ein horizontal bewegtes Steuerelement von der nächsten vertikalen Rasterlinie angezogen,
- ein vertikal bewegtes Steuerelement von der nächsten horizontalen Rasterlinie angezogen.

Der Abstand zwischen den Rasterlinien beträgt 5 DIP.

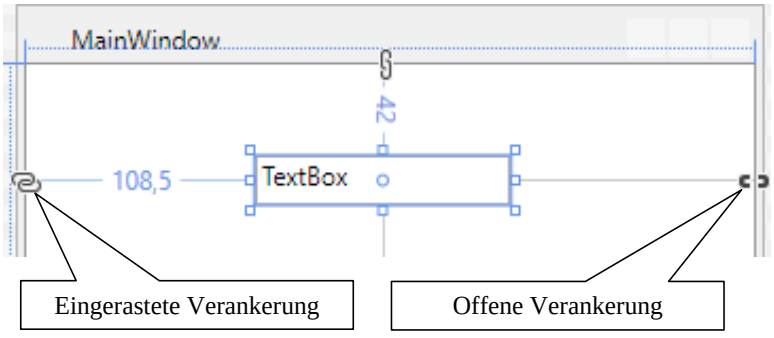
Über die Symbolleiste zwischen Entwurfs- und XAML-Ansicht

kann man eine ausgedünnte Version des Rasterliniengitters ein- bzw. ausblenden:



6.8.3.1.4 Verankerung und Abstände

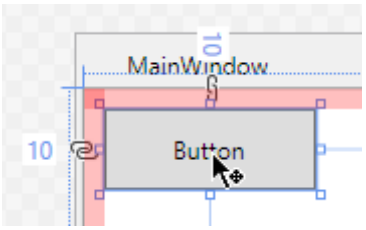
Per Voreinstellung sind die Steuerelemente am linken und am oberen Rand des umgebenden Containers angedockt, was bei einem markierten Steuerelement durch Verankerungssymbole angezeigt wird, z. B.:



Um eine Verankerung vorzunehmen oder aufzuheben, klickt man auf den zugehörigen Verankerungspunkt. Ist beim Öffnen einer Verankerung die gegenüberliegende Verankerung gerade offen, dann wird sie eingerastet.

Zu den Andockseiten werden die Randabstände numerisch (in DIP) angezeigt.

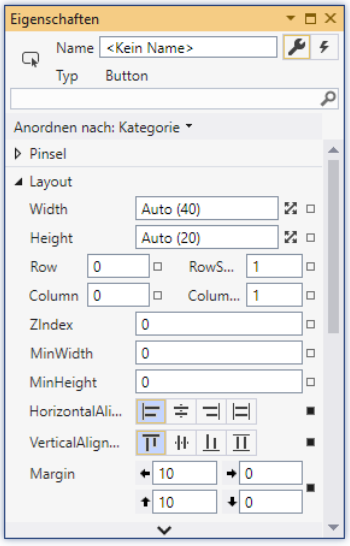
Durch rote Streifen schlägt der WPF-Designer frei zu lassende Zonen mit einer Breite von 10 DIP an den Container-Rändern vor:



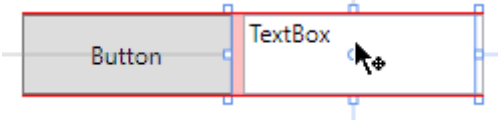
Um bestimmte Randabstände für ein Steuerelement zu realisieren, muss man es nicht unbedingt mit Mausgeschicklichkeit zu einer horizontalen bzw. vertikalen Rasterposition bewegen, sondern kann im XAML-Attribut **Margin** die gewünschten Werte für den linken, oberen, rechten und unteren Randabstand (in dieser Reihenfolge) eintragen, z. B.:

```
<Button Content="Button" Margin="10,10,0,0" VerticalAlignment="Top"
HorizontalAlignment="Left"/>
```

Eine weitere Möglichkeit zur numerischen Spezifikation bietet die Kategorie **Layout** im **Eigenschaften**-Fenster:



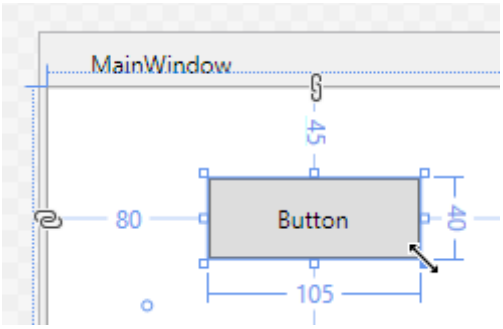
Die vom Designer vorgeschlagenen Mindestabstände zwischen Steuerelementen sind etwas knapp bemessen, z. B.:



Oft dient es der Übersichtlichkeit, etwas mehr Platz zwischen den Bedienelementen zu lassen.

6.8.3.1.5 Ausdehnungen und Transformationen

Für das markierte Steuerelement erlauben die aus Grafikprogrammen bekannten Anfasser eine Größenänderung, wobei die aktuellen Werte numerisch (in DIP) angezeigt werden, z. B.:



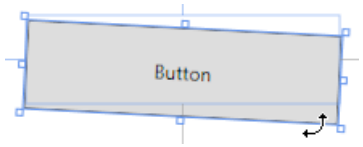
Um eine bestimmte Breite und Höhe für ein Steuerelement zu realisieren, muss man nicht unbedingt die Anfasser mit Mausgeschicklichkeit zu einer horizontalen bzw. vertikalen Rasterposition bewegen, sondern kann die XAML-Attribute **Width** und **Height** auf die gewünschten Werte setzen, z. B.:

```
<Button Content="Button" Margin="10,10,0,0" VerticalAlignment="Top"
HorizontalAlignment="Left" Width="195" Height="55"/>
```

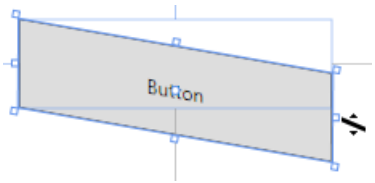
Eine weitere Möglichkeit zur numerischen Spezifikation bietet die Kategorie **Layout** im **Eigenschaften**-Fenster (siehe obiges Bildschirmfoto).

Bei Verzicht auf die XAML-Attribute **Width** und **Height** findet eine am Inhalt des Steuerelements orientierte automatische Berechnung statt.

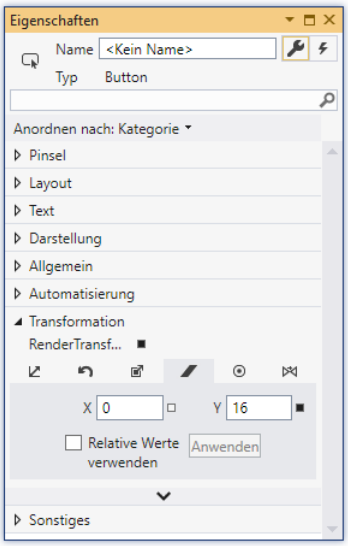
Es ist auch möglich, ein Steuerelement zu drehen



oder zu neigen:



Im **Eigenschaften**-Fenster finden sich diese Einstellungen in der Kategorie **Transformation**, z. B.:

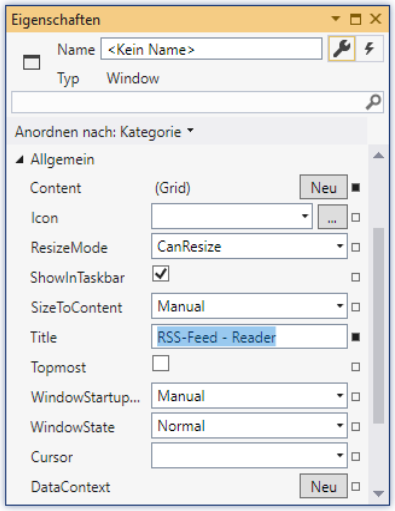


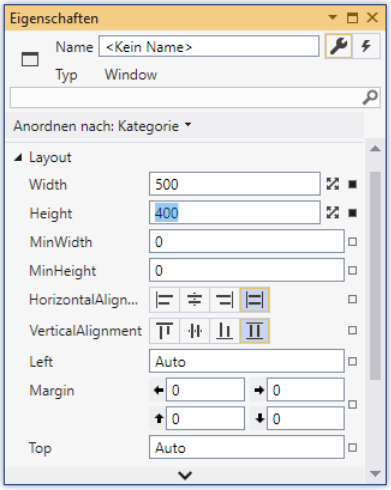
6.8.3.2 Arbeitsablauf

In diesem Abschnitt wird die Erstellung der Bedienoberfläche für den RSS-Feed - Reader Schritt für Schritt beschrieben.

6.8.3.2.1 Anwendungsfenster

Markieren Sie das Anwendungsfenster, um dann per **Eigenschaften**-Fenster eine Titelzeilenbeschriftung und die initiale Fenstergröße festzulegen:

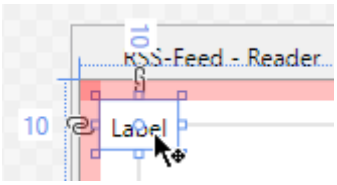




Achten Sie darauf, dass Sie wirklich das Anwendungsfenster erwischen (ein Objekt der Klasse **Window**) und nicht etwa den darin enthaltenen und aus didaktischen Gründen vorläufig ignorierten Container (ein Objekt aus der Klasse **Grid**). Um ganz sicher zu gehen, können Sie auch das **Window**-Wurzelelement im XAML-Fenster markieren.

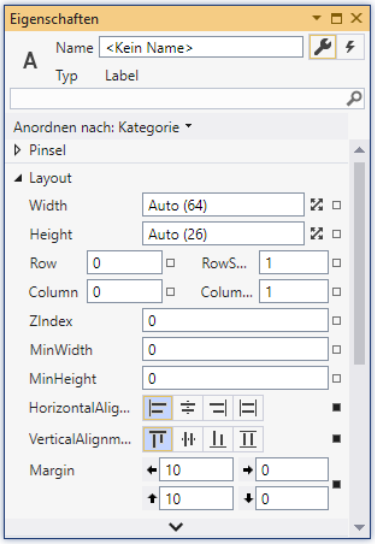
6.8.3.2.2 Label-Objekt

Setzen Sie ein **Label**-Objekt, das den Zweck des Texteingabefelds beschreibt, in die obere linke Ecke des Anwendungsfensters, und akzeptieren Sie die vom Designer vorgeschlagenen Randabstände von 10 DPI zum linken bzw. oberen Fensterrand:



Nach einem *einfachen* Mausklick auf die linke Seite des bereits markierte **Label**-Objekts kann die Beschriftung vor Ort geändert werden, z. B. auf „Feed-URL:“.

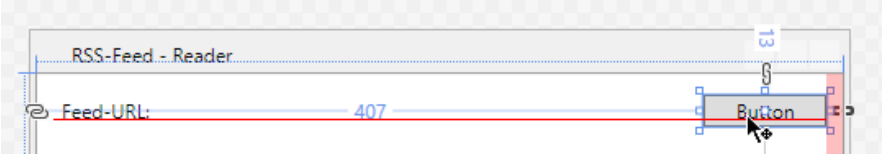
Über Breite und Höhe des **Label**-Objekts darf eine am Inhalt orientierte Automatik entscheiden, die im **Eigenschaften**-Fenster über den Symbolschalter aktiviert wird (Kategorie **Layout**):



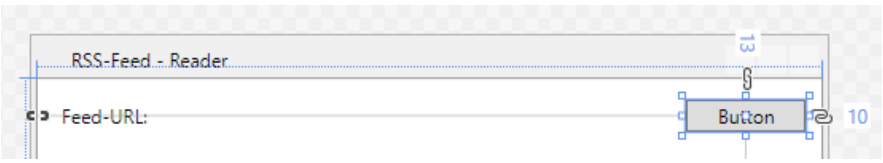
6.8.3.2.3 Button-Objekt

Ergänzen Sie ein **Button**-Objekt, mit dem das Laden einer Feed-Datei angefordert werden soll. Das Objekt sollte so in die rechte obere Fensterecke gesetzt werden, dass es ...

- in vertikaler Richtung die Textbasislinie des **Label**-Objekts übernimmt
- und zum rechten Fensterrand den Standardabstand von 10 DIP einhält.

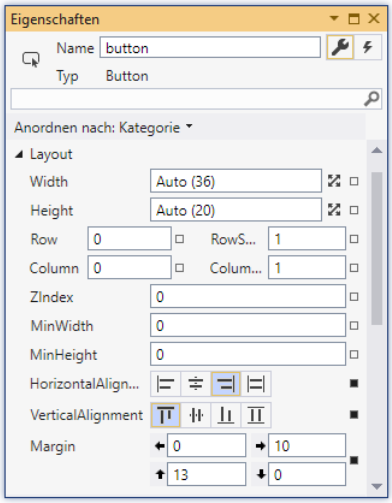


Das **Button**-Objekt sollte oben und rechts verankert sein, *nicht* jedoch an der linken oder unteren Fensterkante, damit es bei einer Vergrößerung des Fensters nicht mitwächst:



Nach einem *einfachen* Mausklick auf das bereits markierte **Button**-Objekt kann es vor Ort beschriftet werden, z. B. durch „Laden“.

Über Breite und Höhe des **Button**-Objekts darf eine am Inhalt orientierte Automatik entscheiden, die im **Eigenschaften**-Fenster über den Symbolschalter aktiviert wird (Kategorie **Layout**):



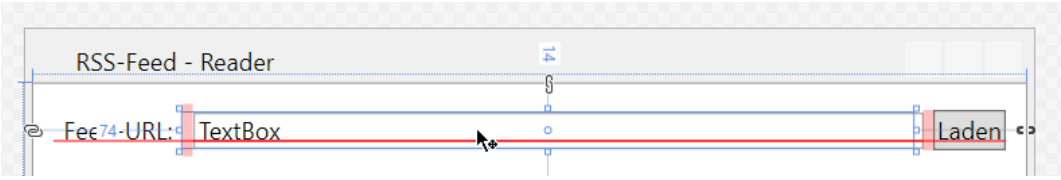
Markieren Sie die **IsDefault**-Eigenschaft der Schaltfläche, damit sie im laufenden Programm per **Enter**-Taste angesprochen werden kann (Kategorie **Allgemein**).

Als **Namen** für die Instanzvariable zum **Button**-Steuerelement, der im **Eigenschaften**-Fenster oder im XAML-Code (als Wert zum Attribut **x:Name**) eingetragen werden kann, wählen wir `button`.

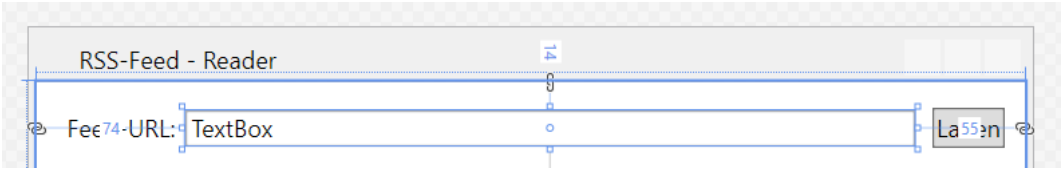
6.8.3.2.4 TextBox-Objekt

Ergänzen Sie ein **TextBox**-Objekt zur Aufnahme der vom Benutzer gewünschten Feed-Adresse. Das Objekt sollte so zwischen das **Label**- und das **Button**-Objekt gesetzt werden, dass es ...

- in vertikaler Richtung die gemeinsame Textbasislinie der Nachbarn übernimmt
- und horizontal zu beiden Nachbarn den vom Designer vorgeschlagenen Mindestabstand einhält.



Das **TextBox**-Objekt sollte links, oben und rechts verankert sein, damit es bei einer horizontalen (nicht aber bei einer vertikalen) Vergrößerung des Fensters mitwächst:



Sobald ein Steuerelement an zwei *gegenüberliegenden* Seiten verankert ist, wird für seine Ausdehnung in der zugehörigen Richtung (also für seine Breite bzw. Höhe) vom WPF-Designer die automatische Wertvergabe eingeschaltet, und im XAML-Code fehlt das zugehörige Attribut (**Width** bzw. **Height**).

Wenn Ihnen der Abstand zwischen dem **TextBox**- und dem **Button**-Objekt zu klein erscheint, können Sie per **Eigenschaften**-Fenster den rechten Randabstand des **TextBox**-Objekts fein dosiert vergrößern (Kategorie **Layout**).

Über die Höhe des **TextBox**-Objekts soll eine am Inhalt orientierte Automatik entscheiden, die im **Eigenschaften**-Fenster über den Symbolschalter aktiviert wird (Kategorie **Layout**).

Sorgen Sie über den Wert **NoWrap** für die Eigenschaft **TextWrapping** (Kategorie **Text**) dafür, dass trotz Platznot die Feed-Adresse nicht umgebrochen wird.

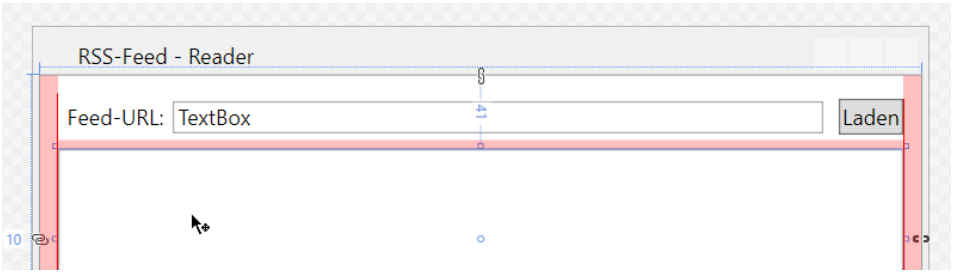
Den initialen Inhalt des **TextBox**-Objekts werden wir per Programm festlegen, sodass wir uns jetzt nicht darum kümmern müssen.

Als **Namen** für die Instanzvariable zum **TextBox**-Steuerelement, der im **Eigenschaften**-Fenster oder im XAML-Code (als Wert zum Attribut **x:Name**) eingetragen werden kann, wählen wir `textBox`.

6.8.3.2.5 ListBox-Objekt

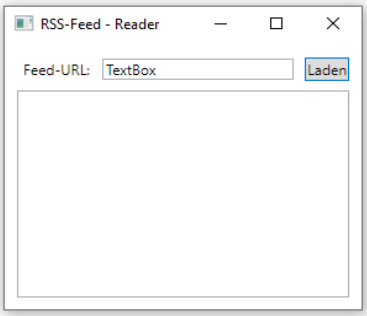
Ergänzen Sie ein **ListBox**-Objekt, das zur formatierten Anzeige der Feed-Items dienen soll. Das Objekt sollte so positioniert werden, dass es ...

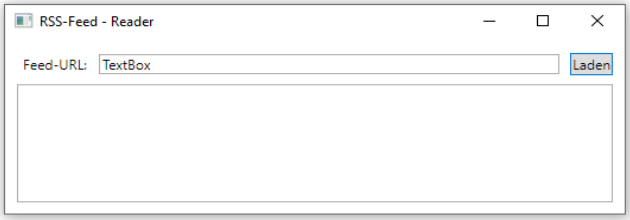
- zum linken, rechten und zum unteren Fensterrand den vom Designer vorgeschlagenen Abstand von 10 DIP einhält
- und nicht allzu dicht unter den drei anderen Bedienelementen sitzt:



Das **ListBox**-Objekt sollte an *allen* Fensterseiten verankert werden, damit es sich bei einer Veränderung der Fenstergröße in horizontaler und in vertikaler Richtung anpasst.

Weil das Programm von Anfang an startfähig ist, kann man sein Verhalten bei variabler Fenstergröße leicht überprüfen, z. B.:





Als **Namen** für die Instanzvariable zum **ListBox**-Steuerelement, der im **Eigenschaften**-Fenster oder im XAML-Code (als Wert zum Attribut **x:Name**) eingetragen werden kann, wählen wir `listBox`.

6.8.4 Fensterklasse `MainWindow`

Wie Sie bereits aus dem Abschnitt 5.12.6 wissen, erzeugt die Entwicklungsumgebung aufgrund der GUI-Gestaltung per WPF-Designer im Hintergrund Quellcode zu der Fensterklasse `MainWindow`. Weil bei der Klassendefinition sowohl der Entwickler als auch das Visual Studio beteiligt sind, wird der Quellcode auf zwei Dateien verteilt:

- **MainWindow.xaml.cs**
  Hier landen Ihre Beiträge (z. B. die Ereignisbehandlungsmethoden).
- **MainWindow.g.cs**
  Der Quellcode in dieser Datei wird bei jedem Erstellen des Projekts aufgrund der Datei **MainWindow.xaml** automatisch neu erzeugt, sodass eine Änderung durch den Entwickler sinnlos ist.

Dem C# - Compiler wird durch das Schlüsselwort **partial** in der Klassendefinition signalisiert, dass der Quellcode auf mehrere Dateien verteilt ist.

Der `MainWindow`-Quellcode in der Datei **MainWindow.xaml.cs** enthält einen Konstruktor, welcher die Methode **InitializeComponent()** aufruft:

```
using System;
  . . .
using System.Windows.Shapes;

namespace RssFeedReader {
   /// <summary>
   /// Interaktionslogik für MainWindow.xaml
   /// </summary>
   public partial class MainWindow : Window {
      public MainWindow() {
         InitializeComponent();
      }
   }
}
```

Diese wird in der Datei **MainWindow.g.cs** von der Entwicklungsumgebung implementiert.

Aufgrund unserer Tätigkeit im WPF-Designer enthält die Fensterklasse `MainWindow` mehrere Objekte anderer Klassen, die Steuerelemente der grafischen Bedienoberfläche repräsentieren. In der Datei **MainWindow.g.cs** finden sich die Deklarationen der zugehörigen Instanzvariablen:[1]

```
using System;
  . . .
using System.Windows.Shapes;

namespace RssFeedReader {
   /// <summary>
   /// MainWindow
   /// </summary>
   public partial class MainWindow : System.Windows.Window,
                                      System.Windows.Markup.IComponentConnector {
  . . .
      internal System.Windows.Controls.TextBox textBox;
  . . .
      internal System.Windows.Controls.Button button;
  . . .
      internal System.Windows.Controls.ListBox listBox;
```

[1] Wie man die Datei **MainWindow.g.cs** anzeigen lässt, wird im Abschnitt 5.12.6 erläutert.

```
        . . .
        public void InitializeComponent() {
          . . .
        }
    }
}
```

Damit beim Programmstart im Texteingabefeld die Adresse des Microsoft-Feeds zum Visual Studio erscheint, nehmen wir im `MainWindow`-Konstruktor für die **Text**-Eigenschaft des **TextBox**-Objekts eine entsprechende Initialisierung vor:

```
public MainWindow() {
   InitializeComponent();
   textBox.Text =
       "https://www.microsoft.com/de-de/techwiese/feeds/rss/aktuell.xml";
}
```

6.8.5 Click-Ereignisbehandlung zum Befehlsschalter (Teil 1)

Wir erstellen eine Ereignisbehandlungsmethode, die durch das Betätigen des Befehlsschalters (per Mausklick oder **Enter**-Taste) ausgelöst wird. Diese Methode soll folgende Leistungen erbringen:

- Unter Verwendung der **Text**-Eigenschaft des **TextBox**-Objekts wird die XML-Datei mit dem gewünschten RSS-Feed aus dem Internet geladen.
- Die Items im RSS-Feed werden an das **ListBox**-Objekt zur formatierten Anzeige übergeben.

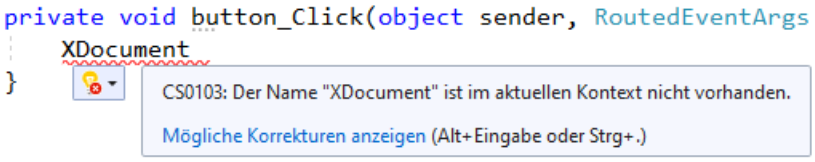
Setzen Sie im WPF-Designer einen Doppelklick auf den Befehlsschalter, sodass die Entwicklungsumgebung in der Datei **MainWindow.xaml.cs** die Instanzmethode `button_Click()` der Klasse `MainWindow` mit leerem Rumpf anlegt

```
private void button_Click(object sender, RoutedEventArgs e) {

}
```

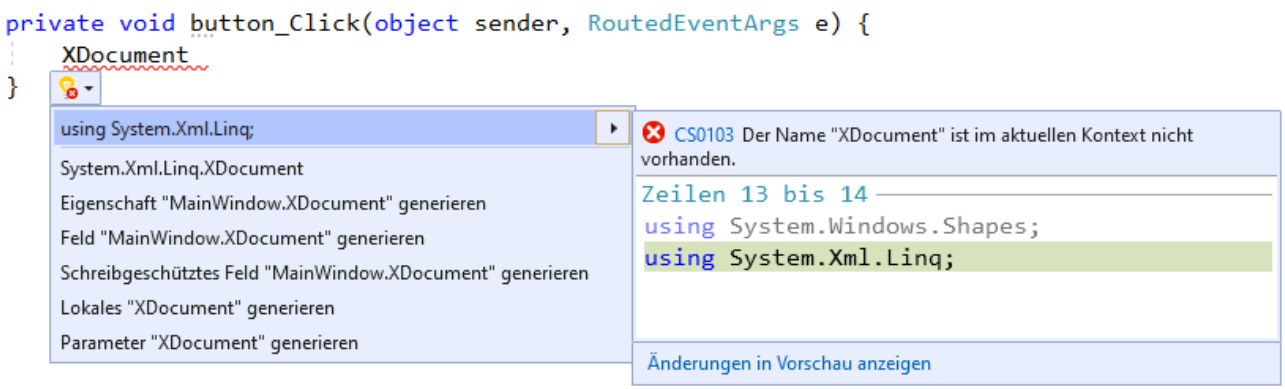
und die Quellcodedatei im Editor öffnet.

Wir befördern den Inhalt der vom Benutzer benannten RSS-Datei in ein Objekt der BCL-Klasse **XDocument**. Für das nicht triviale, mit einem Internetzugriff verbundene Laden der RSS-Datei verwenden wir die statische **XDocument**-Methode **Load()** und übergeben die **Text**-Eigenschaft (mit Datentyp **String**) des **TextBox**-Bedienelements als Aktualparameter.

Der optimistisch in der Methode `button_Click()` als Datentyp für eine lokale Variable eingetippte Klassenname **XDocument** wird von der Entwicklungsumgebung rot unterschlängelt:



Wenn der Mauszeiger über der Unterschlängelung verharrt, erscheinen neben einer Fehlerbeschreibung auch Unterstützungsangebote. Nach einem Mausklick auf den Pfeil neben der Glühbirne oder auf den Link **Mögliche Korrekturen anzeigen** wird u. a. vorgeschlagen, den Namensraum **System.Xml.Linq**, in dem sich die Klasse **XDocument** befindet, per **using**-Direktive zu importieren:



Wir übernehmen per Mausklick auf **using System.Xml.Linq;** den Vorschlag zum Namensraumimport.

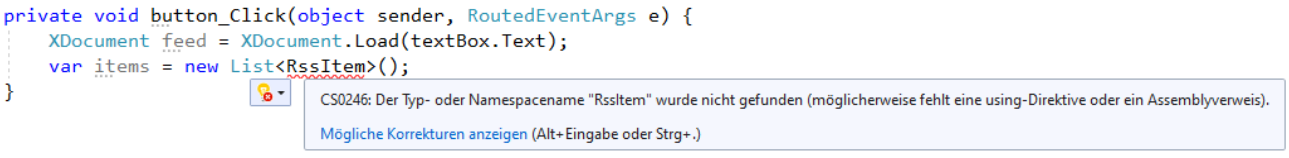
Nun erhält die **XDocument**-Referenzvariable den Namen `feed` und als Wert die Adresse des **XDocument**-Objekts, das von der eben beschriebenen statischen **XDocument**-Methode **Load()** basierend auf der Internetadresse in `textBox.Text` erstellt wird:

```
XDocument feed = XDocument.Load(textBox.Text);
```

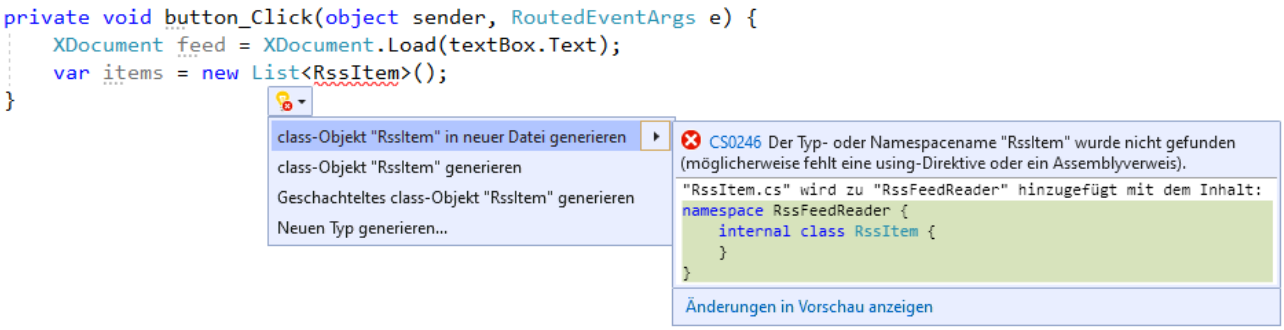
Um ein **ListBox**-Steuerelement zu füllen, kann man seiner Eigenschaft **ItemsSource** ein Objekt aus einer Klasse zuweisen, welche die Schnittstelle **IEnumerable** aus dem Namensraum **System.Collections** erfüllt. Eine Klasse erfüllt eine Schnittstelle, wenn sie alle von der Schnittstelle vorgeschriebenen Methoden besitzt. Im Fall der Schnittstelle **IEnumerable** wird von einer Klasse verlangt, dass sie Elemente mit einem identischen Typ verwalten und sukzessive ausliefern kann, sodass man z. B. in einer **foreach**-Schleife über die Elemente iterieren kann. Wir werden uns im Kapitel 9 mit Schnittstellen im Allgemeinen und im Abschnitt 9.7 mit der Schnittstelle **IEnumerable** im Speziellen beschäftigen.

Für unsere Zwecke eignet sich als **ListBox**-Datenquelle ein Objekt der Klasse **List**<`RssItem`>. Der (noch) ungewohnte Klassenname kommt zustande, weil wir mit der *generischen* Kollektionsklasse **List<T>** arbeiten, die einen größendynamischen Behälter für Elemente mit einem festen, beim Erzeugen des Containers festzulegenden Typ, darstellt. In unserem Fall treten RSS-Items als Elemente auf, und die modellierende Klasse mit dem Namen `RssItem` muss erst noch definiert werden. Mit generischen Typen und mit Kollektionen werden wir uns im Kapitel 8 bzw. 11 beschäftigen, sodass es motivationspsychologisch zu begrüßen ist, wenn Ihnen jetzt relevante Beispiele für das typgenerische Programmieren begegnen.

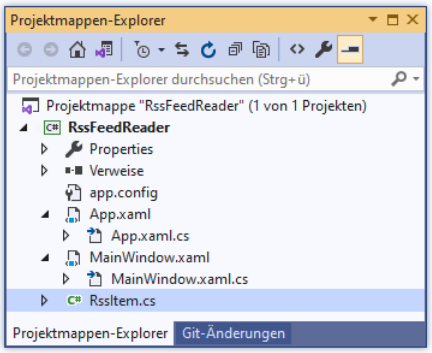
Um die Hilfsbereitschaft und Kompetenz der Entwicklungsumgebung zu testen, erzeugen wir mutig ein Objekt namens `items` aus der Klasse **List**<`RssItem`> unter Verwendung der noch fehlenden Klasse `RssItem`:



Zur erwarteten Fehlermeldung fordern wir die Korrekturvorschläge an



und wählen den ersten. Daraufhin erstellt das Visual Studio eine Klasse mit dem gewünschten Namen `RssItem` in einer eigenen Quellcodedatei, die im Projektmappen-Explorer aufgelistet wird:



In der automatisch erstellten Klassendefinition

```
namespace RssFeedReader {
  internal class RssItem {
  }
}
```

werden wir noch (automatisch implementierte) Eigenschaften für den Titel, die Kurzbeschreibung und den URL eines RSS-Items ergänzen, was gleich mit Hilfe der Entwicklungsumgebung geschehen soll.

Nun widmen wir uns der Aufgabe, die im **XDocument**-Objekt `feed` enthaltenen RSS-Items zu extrahieren und als Objekte der neuen Klasse `RssItem` in das Kollektionsobjekt `items` vom Typ **List**<`RssItem`> einzufüllen. Die **XDocment**-Instanzmethode **Descendants()** liefert ein Objekt, das die generische Schnittstelle **IEnumerble<XElement>** erfüllt und alle XML-Elemente mit dem per Aktualparameter angegebenen Namen aus der geladenen Feed-Datei enthält:

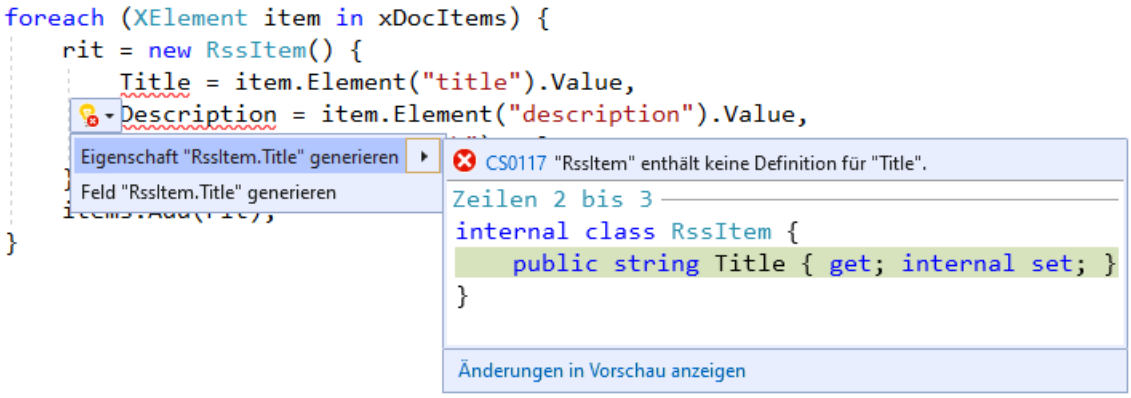
```
IEnumerable<XElement> xDocItems = feed.Descendants("item");
```

In einer **foreach**-Schleife erstellen wir aus jedem **XElement**-Objekt ein `RssItem`-Objekt und nehmen dieses per **Add()** - Methode in die Kollektion `items` vom Typ **List**<`RssItem`> auf:

```
private void button_Click(object sender, RoutedEventArgs e) {
    XDocument feed = XDocument.Load(textBox.Text);
    var items = new List<RssItem>();
    IEnumerable<XElement> xDocItems = feed.Descendants("item");
    RssItem rit;
    foreach (XElement item in xDocItems) {
        rit = new RssItem() {
            Title = item.Element("title").Value,
            Description = item.Element("description").Value,
            Url = item.Element("link").Value
        };
        items.Add(rit);
    }
}
```

Statt eines explizit definierten initialisierenden `RssItem`-Konstruktors wird hier die im Abschnitt 5.4.3.3 beschriebene Objektinitialisierung verwendet.

In der **foreach**-Schleife werden Subelemente eines `<item>` - Elements sowie korrespondierende Eigenschaften der Klasse `RssItem` verwendet, welche dort noch nicht definiert sind (`Title`, `Description` und `Url`). Unsere Entwicklungsumgebung erkennt das Problem und schlägt geeignete Maßnahmen vor, z. B.:



Wir übernehmen den ersten Vorschlag, und die Entwicklungsumgebung erweitert die Definition der Klasse `RssItem` um die automatisch implementierte Eigenschaft `Title` (siehe Abschnitt 5.5.2):

```
internal class RssItem {
  public string Title { get; internal set; }
}
```

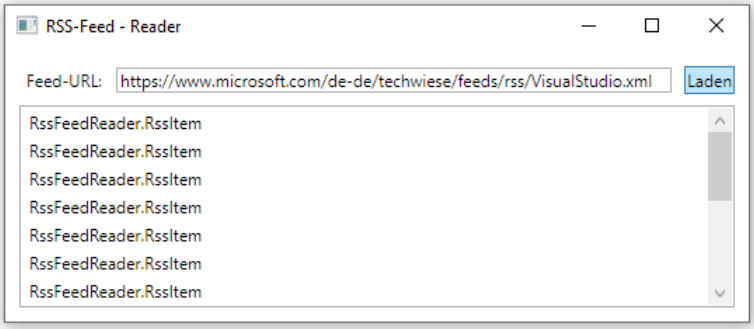
Analog erhalten wir ohne große Anstrengungen die komplette Definition der Klasse `RssItem`:

```
namespace RssFeedReader {
  internal class RssItem {
    public string Title { get; internal set; }
    public string Description { get; internal set; }
    public string Url { get; internal set; }
  }
}
```

Nun können wir am Ende der Ereignisbehandlungsmethode `button_Click()` der **ListBox**-Eigenschaft **ItemsSource** das Kollektionsobjekt `items` mit den `RssItem`-Elementen zuweisen:

```
private void button_Click(object sender, RoutedEventArgs e) {
   XDocument feed = XDocument.Load(textBox.Text);
   var items = new List<RssItem>();
   IEnumerable<XElement> xDocItems = feed.Descendants("item");
   RssItem rit;
   foreach (XElement item in xDocItems) {
      rit = new RssItem() {
         Title = item.Element("title").Value,
         Description = item.Element("description").Value,
         Url = item.Element("link").Value
      };
      items.Add(rit);
   }
   listBox.ItemsSource = items;
}
```

Ein Startversuch mit dem **Laden** des voreingestellten RSS-Feeds zeigt, dass wir auf einem guten Weg sind:



6.8.6 Formatierung der Listenelemente per DataTemplate-Objekt

Bislang zeigt das **ListBox**-Objekt zu jedem RSS-Item lediglich den Datentyp an (offenbar die Produktion der Methode **ToString()**, welche die Klasse `RssItem` von der Urahnklasse **Object** gerbt hat). Um zu einer informativen und optisch attraktiven Anzeige zu kommen, verwenden wir ein geeignet konfiguriertes Objekt der Klasse **DataTemplate**, das der **ListBox**-Eigenschaft **ItemTemplate** als Wert zugewiesen wird. Dies gelingt im Visual Studio am besten durch direktes Editieren der XAML-Datei zum Anwendungsfenster:

```
<ListBox x:Name="listBox" Margin="10,41,10,10">
  <ListBox.ItemTemplate>
      <DataTemplate>
        <StackPanel>
          <TextBlock Text="{Binding Path=Title}" Margin="1" TextAlignment="Center"
             TextWrapping="Wrap" FontSize="14" FontWeight="Bold"
             Foreground="DarkMagenta" />
          <TextBlock Text="{Binding Path=Description}" Margin="1,1,1,4"
             TextWrapping="Wrap" />
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Im XAML-Element **ListBox** wird die sogenannte *Eigenschaftselementsyntax* (vgl. Abschnitt 12.3.2.3) verwendet, um die **ListBox**-Eigenschaft **ItemTemplate** zu versorgen. Dieser Eigenschaft wird ein Objekt der Klasse **System.Windows.DataTemplate** zugewiesen, das in einem eigenen XAML-Element deklariert wird.

Das **DataTemplate**-Objekt verwendet einen Layoutcontainer vom Typ **StackPanel** (siehe Abschnitt 12.6.3) mit vertikaler Orientierung, um zwei **TextBlock**-Objekte übereinander zu präsentieren.

Ein **TextBlock**-Objekt erhält seine Daten über die **Datenbindungstechnologie**. Durch die folgende Attributsyntax mit einer sogenannten *Markup-Erweiterung* (vgl. Abschnitt 12.3.2.3.6)

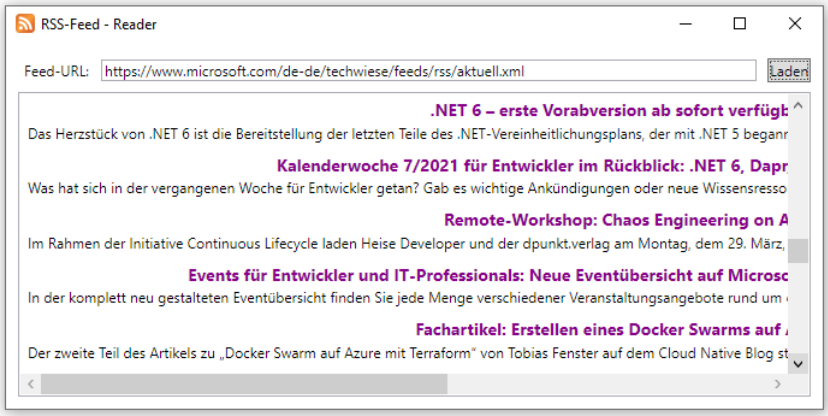
```
Text="{Binding Path=Title}"
```

wird ein Objekt der Klasse **Binding** beauftragt, aus dem aktuellen Element der zum **ListBox**-Objekt gehörigen Kollektion den Wert der Eigenschaft `Title` zu extrahieren. Das Kollektionsobjekt wird in der **Click**-Ereignismethode zum Befehlsschalter (siehe Abschnitt 6.8.5) erzeugt und der **ListBox**-Eigenschaft **ItemsSource** zugewiesen:

```
listBox.ItemsSource = items;
```

Die weiteren **TextBlock** - XAML-Attribute dienen der Formatierung. Wenn ihnen z. B. die Farbe **DarkMagenta** für den Titel missfällt, können Sie z. B. einen alternativen Farbnamen aus der Enumeration **ConsoleColor** (Namensraum **System**) verwenden.

Der bei **ListBox**-Steuerelementen per Voreinstellung vorhandene *horizontale* Rollbalken ist bei unserem nun schon deutlich verbesserten RSS-Feed - Reader für eine unpraktische Textpräsentation verantwortlich

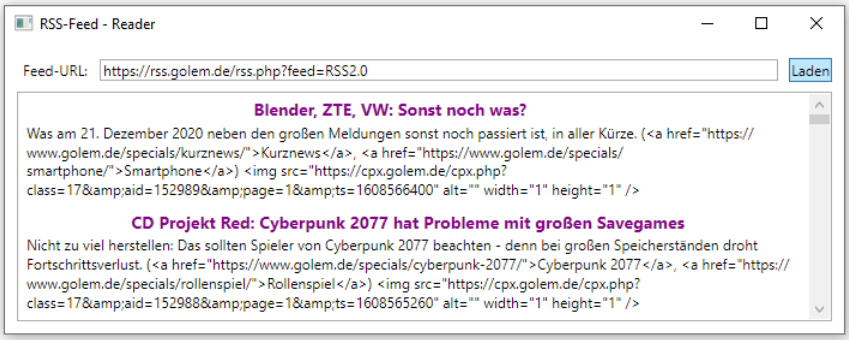


und wird daher über das folgende Attribut zum **ListBox**-Element in der XAML-Datei **MainWindow.xaml** abgeschaltet:

```
<ListBox . . . ScrollViewer.HorizontalScrollBarVisibility="Disabled">
   . . .
</ListBox>
```

HorizontalScrollBarVisibility ist eine sogenannte *Abhängigkeitseigenschaft* der Klasse **ScrollViewer**, und Sie werden noch erfahren, warum man mit ihrer Hilfe für ein Objekt der Klasse **ListBox** den horizontalen Rollbalken beeinflussen kann.

Außerdem ist noch ein Schönheitsfehler festzustellen: Die in den `<description>` - Elementen der RSS-Datei vorhandenen HTML-Elemente werden natürlich nicht interpretiert und sollten daher entfernt werden, z. B.:



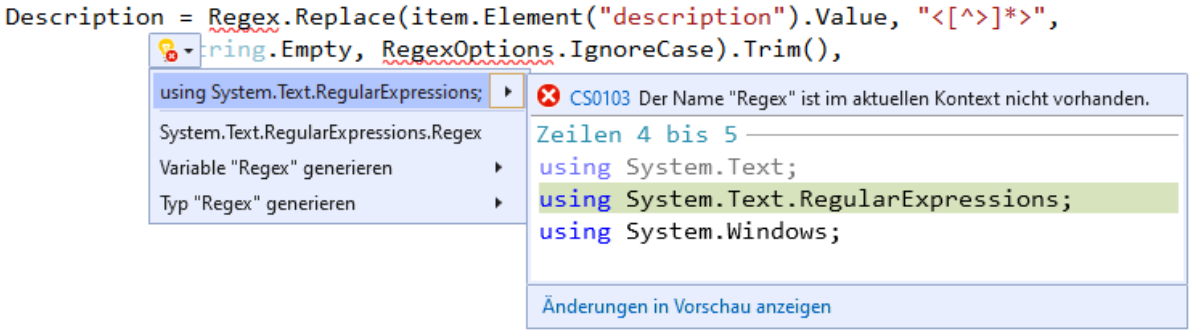
Eine bei **StackOverflow**, einem professionellen Forum zu Fragen der Software-Entwicklung, entdeckte Lösung von *Jossef Harush* erledigt diese Aufgabe mit Hilfe der statischen Methode **Replace()** aus der Klasse **Regex** im Namensraum **System.Text.RegularExpressions**.[1] Das Bereinigen der `<description>` - Elemente in den RSS-Items findet im Rahmen der Objektinitialisierung statt, die wir in der Methode `button_Click()` bei der Erstellung eines `RssItem`-Objekts verwenden (siehe Abschnitt 6.8.5):

```
rit = new RssItem() {
   Title = item.Element("title").Value,
   Description = Regex.Replace(item.Element("description").Value, "<[^>]*>",
                               String.Empty, RegexOptions.IgnoreCase).Trim(),
   Url = item.Element("link").Value
};
```

Die zu suchende und durch **String.Empty** zu ersetzende Zeichenfolge wird über einen sogenannten *regulären Ausdruck* definiert:[2]

- **<**
  Das erste Zeichen muss eine öffnende spitze Klammer sein.
- **[^>]***
  Dann dürfen beliebig viele Zeichen folgen (Quantor ***), die nicht mit der schließenden spitzen Klammer identisch sind (Negative Zeichenauswahl **[^>]**).
- **>**
  Das letzte Zeichen muss eine schließende spitze Klammer sein.

Damit die Klasse **Regex** und die Enumeration **RegexOptions** ohne Namensraumpräfix angesprochen werden können, importieren wir den Namensraum **System.Text.RegularExpressions**:

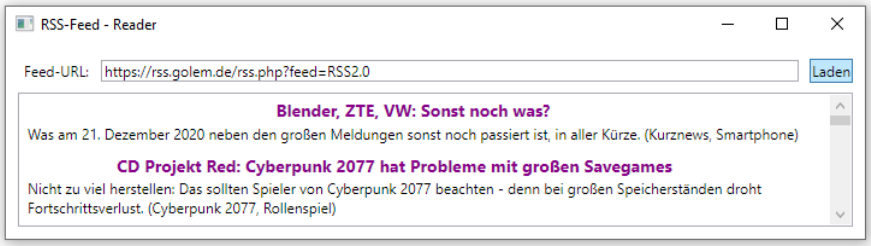


Nun werden die RSS-Items in akzeptabler Form präsentiert, z. B.:

---

1 http://stackoverflow.com/questions/23040422/delete-img-path-from-description

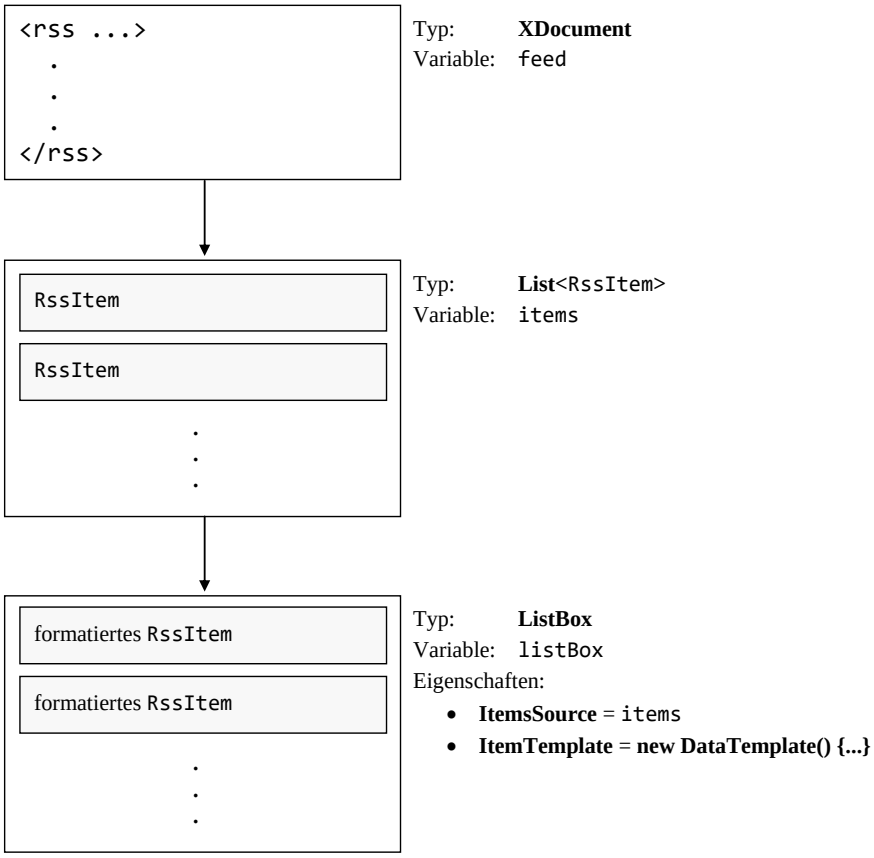
2 Siehe z. B. https://de.wikipedia.org/wiki/Regul%C3%A4rer_Ausdruck



In der folgenden Abbildung wird skizziert, wie die in den Abschnitten 6.8.5 und 6.8.6 beschriebenen Typen

- **XDocument**
- `RssItem`
- **List**`<RssItem>`
- **ListBox**
- **DataTemplate**

zusammenarbeiten, um aus einer RSS-Datei ein **ListBox**-Steuerelement mit formatierten RSS-Items zu erstellen:



6.8.7 Klick-Ereignisbehandlung zum Befehlsschalter (Teil 2)

Weil beim Feed-Abruf und beim Füllen der **ListBox** einiges schief gehen kann, verwenden wir eine **try-catch** - Anweisung im Vorgriff auf Kapitel 13 und zeigen ggf. im **catch**-Block eine Fehlermeldung an, welche die **Message**-Eigenschaft des bei einem Fehler übermittelten **Exception**-Objekts verwendet:

```
private void button_Click(object sender, RoutedEventArgs e) {
   Cursor oldCursor = this.Cursor;
   Cursor = Cursors.Wait;
   try {
      XDocument feed = XDocument.Load(textBox.Text);
      var items = new List<RssItem>();
      IEnumerable<XElement> xDocItems = feed.Descendants("item");
      RssItem rit;
      foreach (XElement item in xDocItems) {
         rit = new RssItem() {
            Title = item.Element("title").Value,
            Description = Regex.Replace(item.Element("description").Value, "<[^>]*>",
                                        String.Empty, RegexOptions.IgnoreCase).Trim(),
            Url = item.Element("link").Value
         };
         items.Add(rit);
      }
      listBox.ItemsSource = items;
   } catch (Exception ex) {
      listBox.ItemsSource = null;
      MessageBox.Show(this, ex.ToString(), ex.Message);
   } finally {
      Cursor = oldCursor;
   }
}
```

Das Laden eines Feeds kann etliche Sekunden dauern. Um den Benutzer darüber zu informieren, dass sein Auftrag in Bearbeitung ist, zeigen wir beim Aufruf der Ereignismethode den Wait-Cursor

an

```
Cursor oldCursor = this.Cursor;
Cursor = Cursors.Wait;
```

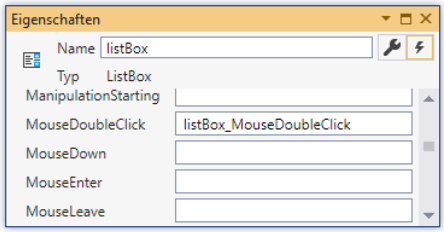
und reaktivieren vor dem Verlassen der Methode den vorherigen Cursor. Damit dieses Restaurieren unter allen Umständen, insbesondere auch nach einem Fehler im **try**-Block, ausgeführt wird, setzen wir die erforderliche Anweisung in einen **finally**-Block, der die **try-catch** - Anweisung erweitert:

```
finally {
   Cursor = oldCursor;
}
```

6.8.8 Doppelklick-Ereignisbehandlung zum ListBox-Steuerelement

Es wäre nett, wenn nach dem Doppelklick auf ein RSS-Item die zugehörige Vollinformation vom bevorzugten Browser angezeigt würde. Um dies zu erreichen, erstellen wir eine Behandlungsmethode zum **MouseDoubleClick**-Ereignis des **ListBox**-Steuerelements:

- Markieren Sie im WPF-Designer das **ListBox**-Steuerelement.
- Wechseln Sie im **Eigenschaften**-Fenster per Mausklick auf das Symbol  zur Anzeige der Ereignisse.
- Setzen Sie einen Doppelklick auf das Ereignis **MouseDoubleClick**:



- Daraufhin wird in der Quellcodedatei **MainWindow.xaml.cs** die Instanzmethode `listBox_MouseDoubleClick()` zu unserer Fensterklasse `MainWindow` angelegt.

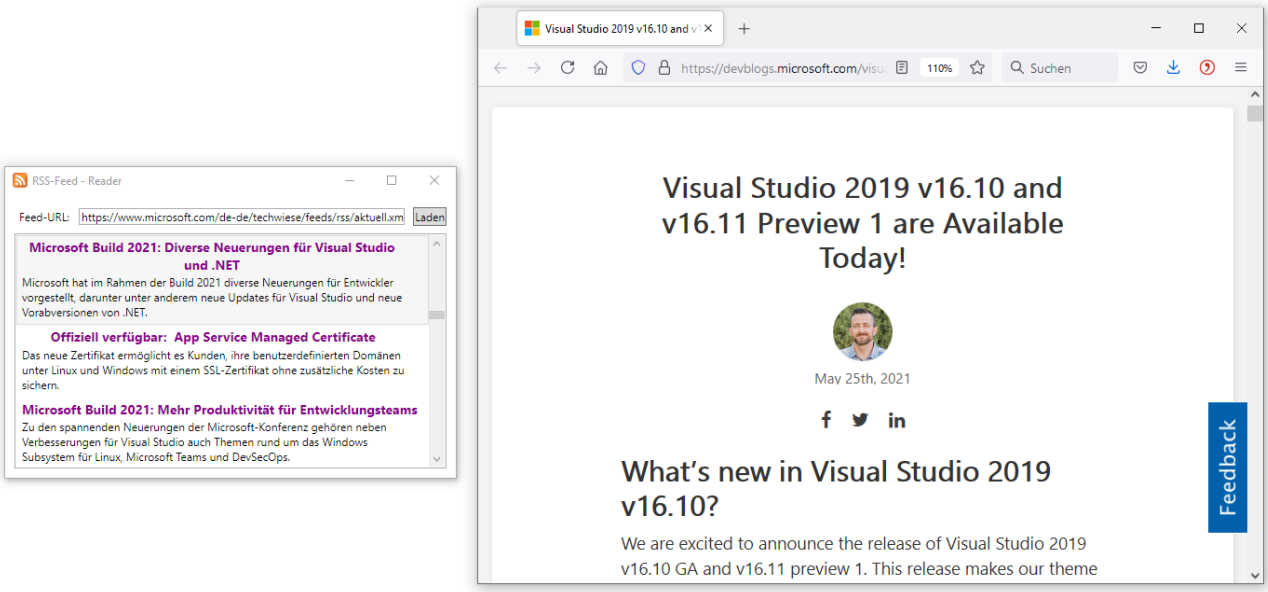
Im Rumpf dieser Methode verwenden wir die statische Methode **Start()** der Klasse **Process** im Namensraum **System.Diagnostics** dazu, um das Betriebssystem zu beauftragen, mit einem geeigneten Programm den Link in demjenigen `RssItem`-Objekt zu öffnen, das aktuell im **ListBox**-Objekt ausgewählt ist. Die **ListBox**-Eigenschaft **SelectedItem** besitzt den Typ **Object**, sodass eine explizite Typumwandlung erforderlich ist:

```
RssItem item = (RssItem)listBox.SelectedItem;
```

Weil beim Aufruf eines externen Programms einiges schief gehen kann, verwenden wir erneut im Vorgriff auf Kapitel 13 eine **try-catch** - Anweisung und zeigen ggf. im **catch**-Block eine Fehlermeldung an, welche die **Message**-Eigenschaft des **Exception**-Objekts verwendet:

```
private void listBox_MouseDoubleClick(object sender, MouseButtonEventArgs e) {
   RssItem item = (RssItem)listBox.SelectedItem;
   // Exist. SelectedItem? Liefert seine Url-Eigenschaft einen nicht-leeren String?
   if (item != null && !String.IsNullOrEmpty(item.Url))
      try {
         System.Diagnostics.Process.Start(item.Url);
      } catch (Exception ex) {
         MessageBox.Show(this, ex.ToString(), ex.Message);
      }
}
```

Nun ist unser Feed-Reader in einem brauchbaren Zustand. Nach einem Doppelklick auf ein Item öffnet der bevorzugte Browser den zugehörigen Link:



6.8.9 Symbol für das Programm und sein Fenster

Abschließend sollen das Programm und sein Fenster noch ein attraktives Symbol erhalten. Wir beziehen von der Wikipedia-Webseite

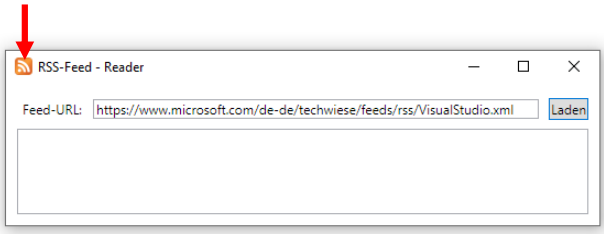
https://de.wikipedia.org/wiki/Datei:Feed-icon.svg

eine Bitmap-Datei mit einem RSS-Symbol im **PNG-Format** (***P**ortable **N**etwork **G**raphics*) mit einer 500 × 500 Pixelmatrix:

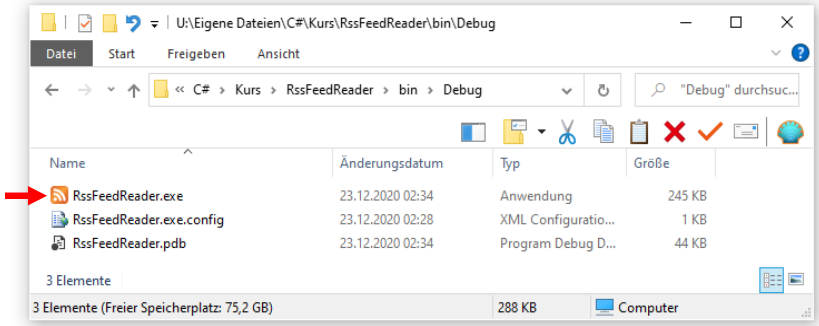
Daraus lässt sich z. B. mit Hilfe der folgenden Webseite

https://convertico.com/

eine Windows-Symboldatei (Namenserweiterung **.ico**) erstellen, die sich für das Fenstersymbol



und für das Anwendungssymbol



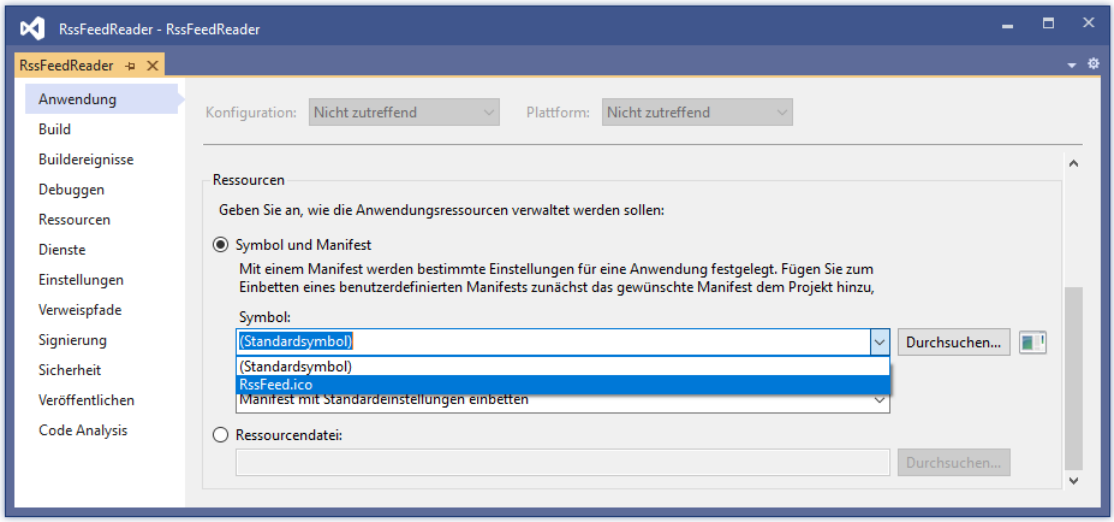
eignet.

Gehen Sie im Visual Studio folgendermaßen vor, um aus der **ico**-Datei das Anwendungssymbol und das Fenstersymbol zu beziehen:

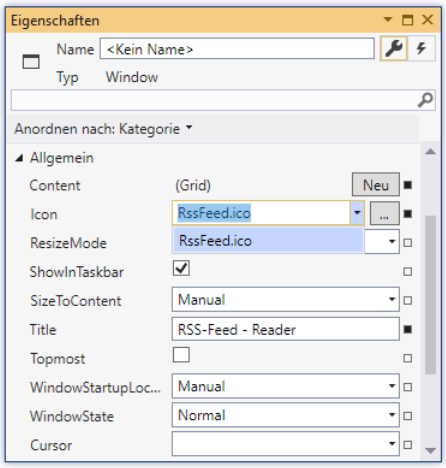
- Kopieren Sie die **ico**-Datei in den Projektordner.
- Nehmen Sie die **ico**-Datei in das Projekt auf (Item **Hinzufügen > Vorhandenes Element** aus dem Kontextmenü zum Projekteintrag im Projektmappen-Explorer).
- Öffnen Sie das Fenster mit den Projekteigenschaften über den Menübefehl

  Projekt > Eigenschaften

  und wählen Sie im Bereich **Anwendung** das **Symbol**:

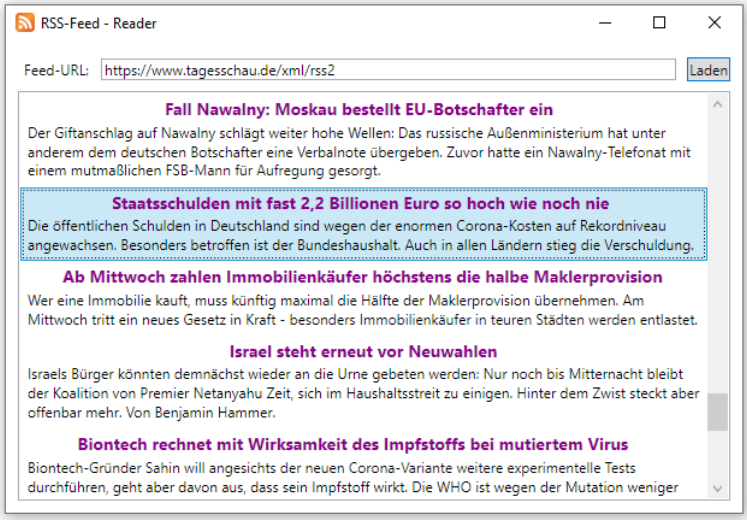


- Markieren Sie im WPF-Designer das Anwendungsfenster, und wählen Sie im **Eigenschaften**-Dialog per Drop-Down - Menü zur Eigenschaft **Icon** (in der Kategorie **Allgemein**) die Datei **RssFeed.ico**:



6.8.10 Selbstkritik und Ausblick

Unser Programm präsentiert RSS-Dateien recht ansehnlich, z. B.:



Allerdings lässt sich das Anwendungsfenster nicht verschieben, während eine RSS-Datei geladen wird. Der für die Darstellung des Fensters zuständige UI-Thread ist offenbar ausgelastet und kann nicht auf Anweisungen zur Änderung des Fensters reagieren. Mit der im Kapitel 17 behandelten Multithread-Programmierung können wir solche Probleme vermeiden (siehe speziell Abschnitt 17.2.3).

Den vollständigen Projektordner mit dem RSS-Feed -Reader auf dem aktuellen Entwicklungsstand finden Sie hier:

...\BspUeb\WPF\RssFeedReader\BlockedUI

6.9 Übungsaufgaben zum Kapitel 6

1) Im folgenden Programm wird den beiden **object**-Variablen `o1` und `o2` derselbe **int**-Wert zugewiesen. Wieso haben die beiden Variablen anschließend nicht denselben Inhalt?

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  object o1 = 1;`
`  object o2 = 1;`
`  Console.WriteLine(o1 == o2);`
`  }`
`}`
	`False`

2) Erstellen Sie ein Programm, das 6 Lottozahlen (von 1 bis 49) zieht und sortiert ausgibt. Vermutlich werden Sie für die Lottozahlen einen eindimensionalen **int**-Array verwenden. Dieser lässt sich mit der statischen Methode **Sort()** aus der Klasse **Array** im Namensraum **System** bequem sortieren.

3) Erstellen Sie ein Programm zur Primzahlensuche mit dem ***Sieb des Eratosthenes*** (ca. 275 - 195 v. Chr.). Dieser Algorithmus reduziert sukzessive eine Menge von Primzahlkandidaten, die initial alle natürlichen Zahlen bis zu einer Obergrenze K enthält, also $\{2, 3, ..., K\}$.

- Im ersten Schritt werden alle echten Vielfachen der Basiszahl 2 (also 4, 6, ...) aus der Kandidatenmenge gestrichen, während die Zahl 2 in der Liste verbleibt.
- Dann geschieht iterativ folgendes:
  - Als neue Basis b wird die kleinste Zahl gewählt, welche die beiden folgenden Bedingungen erfüllt:
    b ist größer als die vorherige Basiszahl.
    b ist im bisherigen Verlauf nicht gestrichen worden.
  - Die echten Vielfachen der neuen Basis (also $2 \cdot b$, $3 \cdot b$, ...) werden aus der Kandidatenmenge gestrichen, während die Zahl b in der Liste verbleibt.
- Das Streichverfahren kann enden, wenn für eine neue Basis b gilt:

$$b > \sqrt{K}$$

In der Kandidatenrestmenge befinden sich dann nur noch Primzahlen. Um dies einzusehen, nehmen wir an, es hätte eine Zahl $n \le K$ mit echtem Teiler das beschriebene Streichverfahren überstanden. Mit zwei positiven Zahlen u, v würde dann gelten:

$$n = u \cdot v \text{ und } u \le \sqrt{K} \text{ oder } v \le \sqrt{K} \text{ (wegen } n \le K)$$

Wir nehmen ohne Beschränkung der Allgemeinheit $u \leq \sqrt{K}$ an und unterscheiden zwei Fälle:

- *u* war zuvor als Basis dran.
  Dann wurde *n* bereits als Vielfaches von *u* gestrichen.
- *u* wurde zuvor als Vielfaches einer früheren Basis $\tilde{b}$ (< *b*) gestrichen ($u = k\tilde{b}$).
  Dann wurde auch *n* bereits als Vielfaches von $\tilde{b}$ gestrichen.

Damit erweist sich die Annahme als falsch, und es ist gezeigt, dass die Kandidatenrestmenge nur noch Primzahlen enthält.

Sollen z. B. alle Primzahlen kleiner oder gleich 18 bestimmt werden, so startet man mit der folgenden Kandidatenmenge:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Im ersten Schritt werden die echten Vielfachen der Basis 2 gestrichen:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

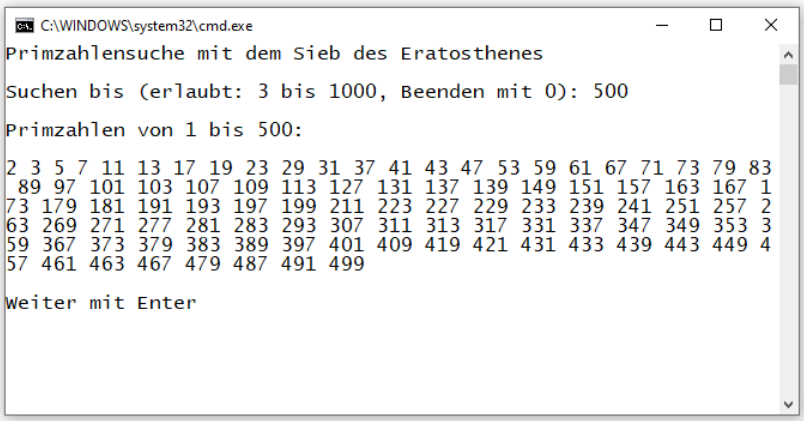
Als neue Basis wird die Zahl 3 gewählt (> 2, nicht gestrichen). Ihre echten Vielfachen werden gestrichen:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

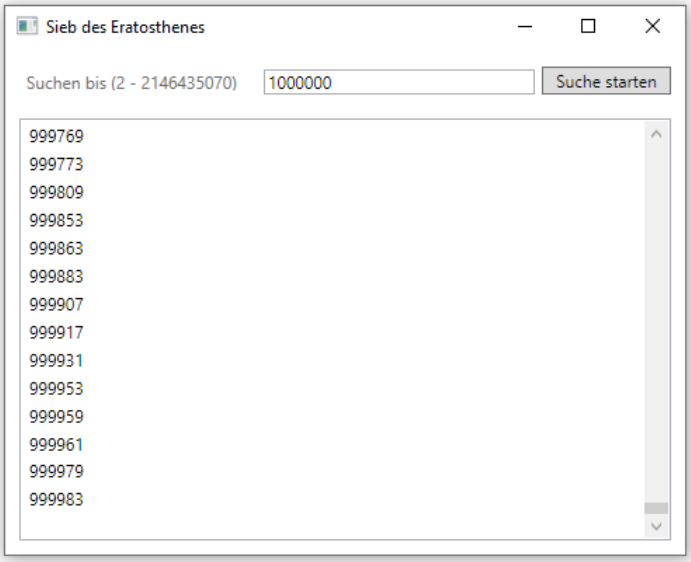
Als neue Basis wird die Zahl 5 gewählt (> 3, nicht gestrichen). Allerdings ist 5 größer als $\sqrt{18}$ (≈ 4,24) und der Algorithmus daher bereits beendet. Als Primzahlen kleiner oder gleich 18 erhalten wir also:

2, 3, 5, 7, 11, 13 und 17

Alternativ können Sie ein Programm für die Konsole



oder eine WPF-Anwendung erstellen:



Der Lösungsvorschlag für die WPF-Anwendung zeigt eine starke Ähnlichkeit mit dem im Abschnitt 6.8 entwickelten RSS-Feed - Reader. Es werden vier Steuerelemente verwendet:

- ein **Label**-Objekt
  Es beschreibt die unterstützten Suchaufträge.
- ein **TextBox**-Objekt
  Hier können die Benutzer das Maximum für die Suche eintragen.
- ein **Button**-Objekt
  Damit wird die Suche gestartet.
- ein **ListBox**-Objekt
  Hier werden die gefundenen Primzahlen ausgegeben.

Hinweise zum Befüllen des **ListBox**-Steuerelements mit den gefundenen Primzahlen:

- Erstellen Sie ein Objekt der konkretisierten generischen Kollektionsklasse **List<int>**:
  ```
  var items = new List<int>();
  ```
- Zur Aufnahme einer Primzahl in die Liste eignet sich die Methode **Add()**, z. B.:
  ```
  items.Add(i)
  ```
- Weisen Sie die fertige Liste der **ListBox**-Eigenschaft **ItemsSource** zu, z. B.:
  ```
  listBox.ItemsSource = items;
  ```

4) Erstellen Sie eine Klasse für zweidimensionale Matrizen mit Elementen vom Typ **float**. Implementieren Sie eine Methode zum Transponieren einer Matrix.

5) Erstellen Sie ein Programm zum Berechnen einer persönlichen Glückszahl (zwischen 1 und 100), indem Sie:

- den Vor- und Nachnamen als Befehlszeilenargumente einlesen,
- den ersten Buchstaben des Vornamens sowie den letzten Buchstaben des Nachnamens ermitteln (beide in Großschreibung),
- die Nummern der beiden Buchstaben im Unicode-Zeichensatz bestimmen,
- die beiden Buchstabennummern addieren und die Summe als Initialisierungswert für den Pseudozufallszahlengenerator aus der Klasse **Random** verwenden.

Beenden Sie Ihr Programm mit einer Fehlermeldung, wenn weniger als zwei Befehlszeilenargumente übergeben wurden.

Tipps:

- Um die durch Leerzeichen getrennten Befehlszeilenargumente im Programm als **String**-Array verfügbar zu haben, definiert man im Kopf der **Main()** - Methode einen Parameter vom Typ **String[]** (vgl. Abschnitt 4.7.2.3.2):

```
static void Main(string[] args) {...}
```

- Wie jede andere Methode kann auch **Main()** per **return**-Anweisung spontan beendet werden.

6) Erstellen Sie eine Klasse `StringUtil` mit einer statischen Methode `WrapLine()`, die einen **String** auf die Konsole schreibt und dabei einen korrekten Zeilenumbruch vornimmt. Anwender Ihrer Methode sollen optional ...

- die gewünschte Zeilenbreite vorgeben können
- und festlegen dürfen, ob ein Bindestrich zwischen zwei Wörtern (z. B. „EU-Parlament“) als optionaler Trennstrich am Zeilenende zu verwenden ist.

Weitere Anforderungen an die Methode:

- Aufeinanderfolgende Leerzeichen sollen wie ein einzelnes Leerzeichen wirken.
- Ist ein Wort breiter als die Ausgabezeile, ist ein Umbruch *innerhalb* des Wortes unvermeidlich.

Im folgenden Programm wird die Verwendung der Methode demonstriert:

```
using System;
class StringUtilTest {
  static void Main() {
    string s = "Dieser Satz - bitte beachten - passt nicht in eine Schmal-Zeile, " +
               "die nur wenige Spalten umfasst.";
    StringUtil.WrapLine(s, 30, true); Console.WriteLine();
    StringUtil.WrapLine(s, 50, true); Console.WriteLine();
    StringUtil.WrapLine(s, 40); Console.WriteLine();
    StringUtil.WrapLine(s); Console.WriteLine();
    Console.ReadLine();
  }
}
```

Der erste Methodenaufruf sollte folgende Ausgabe erzeugen:

```
Dieser Satz - bitte beachten -
passt nicht in eine Schmal-
Zeile, die nur wenige Spalten
umfasst.
```

Tipp: Eine wesentliche Hilfe kann die **String**-Methode **Split()** sein, die auf Basis einer einstellbaren Menge von Trennzeichen alle Teilzeichenfolgen der angesprochenen Instanz ermittelt und in einem **String**-Array ablegt. Im folgenden Programm wird die Arbeitsweise von **Split()** demonstriert:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`   String s = "Dies ist der Beispiel-Satz, der zerlegt werden soll.";`
`   String[] tokens = s.Split(new char[] {' ', '-'},`
`                              StringSplitOptions.RemoveEmptyEntries);`
`   foreach (String t in tokens)`
`    Console.WriteLine(t);`
`  }`
`}`
	`Dies`
	`ist`
	`der`
	`Beispiel`
	`Satz,`
	`der`
	`zerlegt`
	`werden`
	`soll.`

Die Trennzeichen sind *nicht* in den produzierten Teilzeichenfolgen enthalten, sodass z. B. ein als Trennzeichen definierter Bindestrich verloren geht. Das sollte nach Möglichkeit in Ihrem Programm verhindert werden.

Mit dem Enumerationswert

```
StringSplitOptions.RemoveEmptyEntries
```

für den zweiten **Split()** - Parameter werden leere Teilzeichenfolgen im resultierenden **String**-Array (z. B. resultierend aus mehreren aufeinander folgenden Leerzeichen) verhindert.

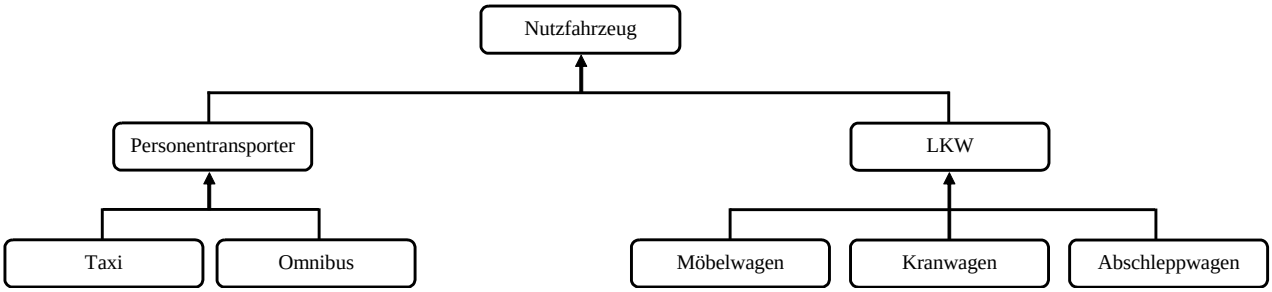
7 Vererbung und Polymorphie

Im Manuskript war schon mehrfach davon die Rede, dass die .NET - Datentypen in eine strenge Abstammungshierarchie eingeordnet sind. Nun betrachten wir die Vererbungsbeziehung zwischen Klassen und die damit verbundenen Vorteile für die Software-Entwicklung im Detail. Es muss aber auch von einigen Komplikationen und Tücken berichtet werden. Die Vererbung gilt als eine von drei Säulen der objektorientierten Programmierung. Sie wird nicht nur in der BCL intensiv und erfolgreich eingesetzt, sodass eine gründliche Behandlung angemessen ist.[1]

Von den drei Säulen der objektorientierten Programmierung wird eine weitere, nämlich die Polymorphie, wegen ihrer starken Bezüge zur Vererbung ebenfalls in diesem Kapitel beschrieben.

Modellierung realer Klassenhierarchien

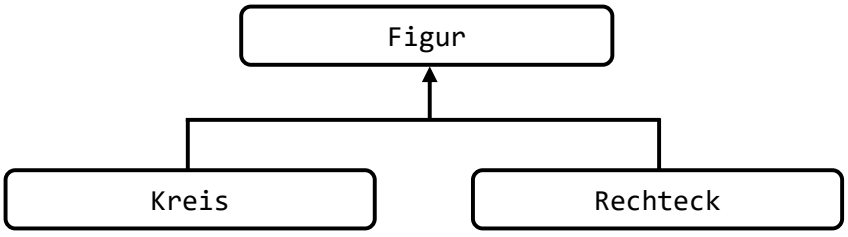
Beim Modellieren eines Gegenstandbereichs durch Klassen, die durch Merkmale (Instanz- und Klassenvariablen) sowie Handlungskompetenzen (Instanz- und Klassenmethoden) gekennzeichnet sind, müssen auch die Spezialisierungs- bzw. Generalisierungsbeziehungen zwischen real existierenden Klassen abgebildet werden. Eine Firma für Transportaufgaben aller Art mag ihre Nutzfahrzeuge folgendermaßen klassifizieren:



Einige Merkmale sind für alle Nutzfahrzeuge relevant (z. B. Anschaffungspreis, momentane Position), andere betreffen nur spezielle Klassen (z. B. maximale Anzahl der Fahrgäste, maximale Anhängelast). Ebenso sind einige Handlungsmöglichkeiten bei allen Nutzfahrzeugen vorhanden (z. B. eigene Position melden, ein Ziel ansteuern), während andere speziellen Fahrzeugen vorbehalten sind (z. B. Fahrgäste befördern, Lasten transportieren). Ein Programm zur Verwaltung der Fahrzeuge und ihrer Einsätze sollte diese reale Klassenhierarchie abbilden.

Übungsbeispiel

Bei unseren Beispielprogrammen bewegen wir uns in einem bescheideneren Rahmen und betrachten meist eine einfache Hierarchie mit Klassen für geometrische Figuren:



Vielleicht haben manche Leser als Gegenstück zum Rechteck (auf derselben Hierarchieebene) die *Ellipse* erwartet, die ebenfalls zwei ungleiche lange „Seiten“ besitzt. Weiterhin liegt es auf den ersten Blick nahe, den Kreis als Spezialisierung der Ellipse und das Quadrat als Spezialisierung des

---

[1] Im weiteren Verlauf des Kurses werden noch Alternativen zur Vererbung vorgestellt. Z. B. kann die Funktionalität einer Klasse modifiziert werden, indem einem Feld mit Delegatentyp eine kompatible Methode zugewiesen wird (siehe Kapitel 9). Wenn die Klasse ihr Delegatenobjekt aufruft, kommt die „injizierte“ Methode zum Einsatz. Diese Option zur Verhaltenskonfiguration ist oft flexibler als die Definition von abgeleiteten Klassen.

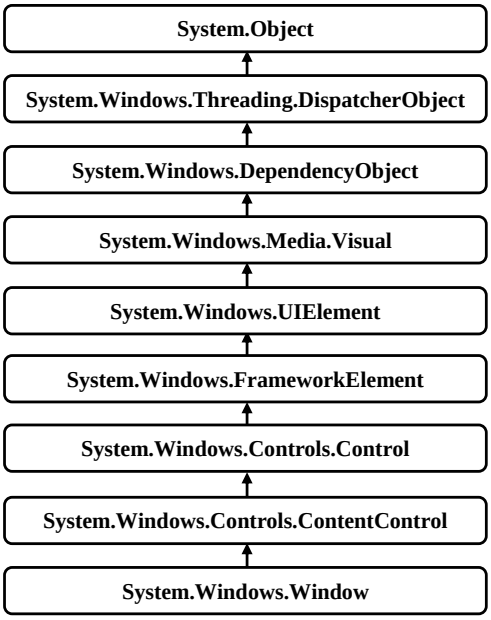
Rechtecks zu betrachten. Wir werden aber im Abschnitt 7.12 über das *Liskovsche Substitutionsprinzip* genau diese Ableitungen (von Kreis aus Ellipse bzw. von Quadrat aus Rechteck) kritisieren. Man spricht hier vom *Kreis-Ellipse* - oder *Quadrat-Rechteck - Problem*.[1] Es ist wohl akzeptabel, an Stelle der Ellipse den Kreis neben das Rechteck zu stellen, um das Erlernen der neuen Konzepte durch ein möglichst einfaches Beispiel ohne Verstoß gegen das Liskovsche Substitutionsprinzip zu erleichtern.

Die Vererbungstechnik der OOP

In objektorientierten Programmiersprachen ist es weder sinnvoll noch erforderlich, jede Klasse einer Hierarchie komplett neu zu definieren. Man geht stattdessen von der allgemeinsten Klasse aus und leitet durch Spezialisierung neue Klassen ab, nach Bedarf in beliebig vielen Stufen. Eine abgeleitete Klasse erbt alle Merkmale und Handlungskompetenzen ihrer **Basisklasse** (jedoch keine Konstruktoren) und kann nach Bedarf Anpassungen bzw. Erweiterungen zur Lösung spezieller Aufgaben vornehmen, z. B.:

- zusätzliche Felder deklarieren
- zusätzliche Methoden oder Eigenschaften definieren
- geerbte Methoden ersetzen, d.h. unter Beibehaltung der Signatur umgestalten

Die BCL ist das beste Beispiel für den erfolgreichen Einsatz der Vererbungstechnik. Viele von uns benötigte Klassen haben einen länglichen Stammbaum, z. B. die Klasse **Window** für das Hauptfenster einer WPF-Anwendung:



Wir haben bereits mehrere WPF-Anwendungen erstellt und dabei die Klasse **Window** als Basisklasse für die Ableitung einer projektspezifischen Klasse namens `MainWindow` verwendet (siehe z. B. Abschnitt 6.8.4).

Software-Recycling

Mit ihrem Vererbungsmechanismus bietet die objektorientierte Programmierung ideale Voraussetzungen dafür, vorhandene Software auf rationelle Weise zur Lösung neuer Aufgaben zu verwenden. Dabei können allmählich umfangreiche Software-Systeme entstehen, die gleichzeitig stabil und innovationsoffen sind (vgl. Abschnitt 5.1.1.3 zum Open-Closed - Prinzip). Die nicht selten anzutreffende Praxis, vorhandenen Code per *Copy & Paste* in neue Projekte bzw. Klassen zu übernehmen,

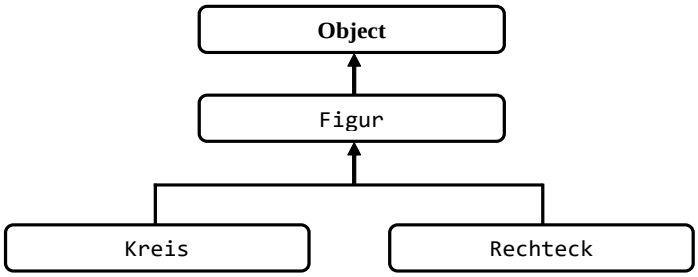
---

[1] Siehe z. B. https://en.wikipedia.org/wiki/Circle-ellipse_problem

hat gegenüber der Nutzung und Erweiterung einer sorgfältig geplanten Klassenhierarchie offensichtliche Nachteile. Natürlich kann C# nicht garantieren, dass jede Klassenhierarchie exzellent entworfen ist und langfristig von einer stetig wachsenden Entwicklergemeinde eingesetzt wird.

7.1 Das allgemeine Typsystem der .NET - Plattform

Alle Klassen und auch die sonstigen Typen (z. B. Strukturen, Enumerationen) der .NET - Plattform stammen von der Klasse **Object** im Namensraum **System** ab. Das gilt sowohl für die in der BCL enthaltenen als auch für die von Anwendungsentwicklern definierten Typen. Wird (wie bei unseren bisherigen Beispielen) in der Definition einer Klasse *keine* Basisklasse angegeben, dann stammt sie auf direktem Wege von der Urahnklasse **Object** ab. Die oben dargestellte Klassenhierarchie zum Figurenbeispiel muss also folgendermaßen vervollständigt werden:



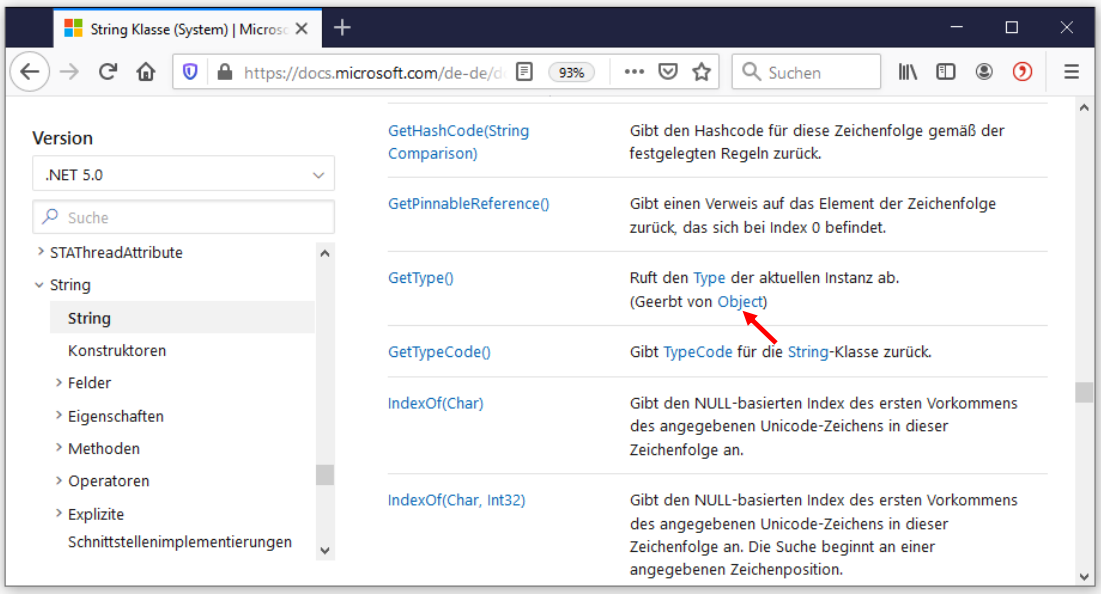
Auch die Strukturen (vgl. Abschnitt 6.1) sind in das allgemeine Typsystem (engl.: Common Type System, CTS) eingeordnet. Sie stammen implizit von der Klasse **System.ValueType** ab, die wiederum direkt von der Urahnklasse **Object** erbt (siehe Abschnitt 6.1.2). Aus einer Struktur kann aber weder eine andere Struktur noch eine Klasse abgeleitet werden. Analoges gilt für die Aufzählungstypen, die von der Klasse **System.Enum** (mit der Basisklasse **System.ValueType**) abstammen (siehe Abschnitt 6.4).

Jeder Typ erbt alle Merkmale und Handlungskompetenzen aus der eigenen Abstammungslinie von der Urahnklasse **Object** beginnend. Folglich kann z. B. jedes Objekt und jede Strukturinstanz die in der Urahnklasse definierte Methode **GetType()** ausführen, die ein auskunftsfreudiges **Type**-Objekt liefert. Im folgenden **WriteLine()** - Aufruf verraten drei **Type**-Objekte (über die implizit aufgerufene Methode **ToString()**) die Typbezeichnung samt Namensraum:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  var o = new Object();`
`  var s = "abc";`
`  var i = 13;`
`  Console.WriteLine(o.GetType() + "\n" +`
`  s.GetType() + "\n" +`
`  i.GetType());`
`  }`
`}`
	`System.Object`
	`System.String`
	`System.Int32`

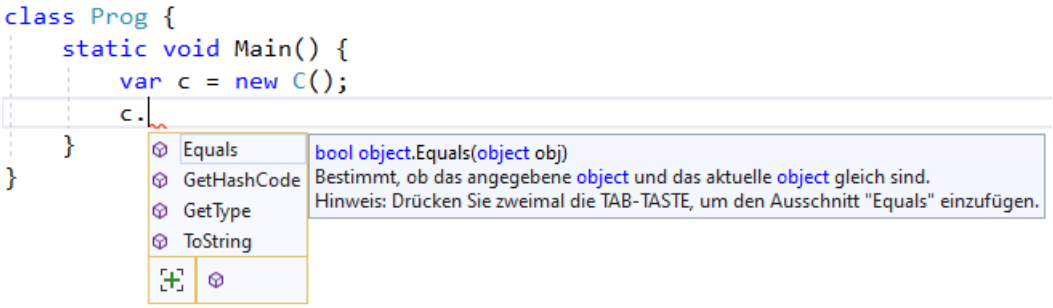
In der BCL-Dokumentation zu einem Datentyp sind die Erbstücke gekennzeichnet, z. B. bei der Klasse **String**:



Auch die Objekte der folgenden C# - Klasse

```
class C { }
```

beherrschen immerhin vier, von **Object** geerbte Methoden:



Die Methoden **Equals()**, **GetHashCode()** und **ToString()** sind allerdings in der Klasse **Object** nur rudimentär implementiert, sodass oft das Überschreiben durch eine eigene Implementation sinnvoll ist (siehe Abschnitt 7.7 zum Überschreiben von Methoden):

- Die **Object**-Methode **ToString()** liefert lediglich den Typnamen, kann also einzelne Instanzen *nicht* charakterisieren.
- Die Methode **Equals()** beurteilt die Identität von zwei Objekten nur über deren Speicheradressen, und die Methode **GetHashCode()**, die zu einem Objekt einen möglichst eindeutigen **int**-Wert liefern soll, orientiert sich nur an der Speicheradresse. Wenn Objekte eines Typs durch Kollektionsklassen wie **HashSet<T>** oder **Dictionary<K, V>** verwaltet werden sollen (siehe Kapitel 11), dann sind sinnvolle Überschreibungen der **Object**-Methoden **Equals()** und **GetHashCode()** erforderlich.

Nur für Klassen und (mit Einschränkungen) für Records ist es in C# möglich, ...

- in der eigenen Definition eine Basisklasse (bzw. einen Basis-Record) anzugeben,
- bei der Definition anderer Klassen bzw. Records als Basistyp benannt zu werden.

Bei Strukturen, Enumerationen und Tupeln ist das nicht möglich. Diese Typen ...

- haben eine implizite, fest vorgegebene Basisklasse
- und können nicht beerbt werden.

Die Strukturen können allerdings Schnittstellen implementieren, und eine implementierte Schnittstelle kann manchmal eine ähnliche Rolle spielen wie eine Basisklasse (z. B. bei der Polymorphie).

7.2 Definition einer abgeleiteten Klasse

Wir definieren im angekündigten Beispiel zunächst die Basisklasse `Figur`, die Instanzvariablen für die X- und die Y-Position der linken oberen Ecke einer zweidimensionalen Figur, zwei Konstruktoren sowie eine Methode `Wo()` zur Positionsmeldung besitzt:

```
using System;
public class Figur {
   double xpos = 100.0, ypos = 100.0;
   public Figur(double x, double y) {
      if (x >= 0.0 && y >= 0.0) {
         xpos = x;
         ypos = y;
      }
      Console.WriteLine("Figur-Konstruktor");
   }
   public Figur() { }

   public void Wo() {
      Console.WriteLine("\nOben Links:\t(" + xpos + ", " + ypos + ") ");
   }
}
```

Wir definieren die Klasse `Kreis` als Spezialisierung der Klasse `Figur`, indem wir hinter dem Klassennamen durch Doppelpunkt getrennt den Namen der Basisklasse angeben:

```
using System;
public class Kreis : Figur {
   double radius = 75.0;
   public Kreis(double x, double y, double rad) : base(x, y) {
      if (rad >= 0.0)
         radius = rad;
      Console.WriteLine("Kreis-Konstruktor");
   }
   public Kreis() { }

   public double Radius {
      get {return radius;}
      set {if (value >= 0.0) radius = value;}
   }
}
```

Die `Kreis`-Klasse erbt die beiden Positionsfelder sowie die Methode `Wo()` und ergänzt eine zusätzliche Instanzvariable für den Radius samt Eigenschaft für Zugriffe durch fremde Typen.

Es wird ein initialisierender `Kreis`-Konstruktor definiert, der über das Schlüsselwort **base** den initialisierenden Konstruktor der Basisklasse aufruft. Weil die `Figur`-Instanzvariablen (noch) als **private** deklariert sind, wäre dem `Kreis`-Konstruktor auch kein direkter Zugriff erlaubt.

In C# werden auch **private Member** vererbt, doch hat eine abgeleitete Top-Level - Klasse keinen direkten Zugriff auf diese Member.[1] Sie kann also z. B. ...

- weder direkt auf private Felder der Basisklasse zugreifen,
- noch private Methoden der Basisklasse aufrufen.

In der `Kreis`-Klasse wird (wie in der Basisklasse `Figur`) auch ein parameter*freier* Konstruktor definiert. Vielleicht hat jemand gehofft, die `Kreis`-Klasse würde den parameterfreien Konstruktor

[1] Ist eine abgeleitete Klasse in ihre Basisklasse *eingeschachtelt* (siehe Abschnitt 5.10), dann kann sie auf die privaten Member der Basisklasse zugreifen. Diese Zugriffsrechte basieren aber nicht auf der Vererbungsbeziehung, sondern sind bei jeder eingeschachtelten Klasse vorhanden.

ihrer Basisklasse (bei automatischer Anpassung des Namens) übernehmen. **Konstruktoren** werden jedoch grundsätzlich **nicht** vererbt. Ein Konstruktor hat die Aufgabe, ein Objekt für seinen Einsatz vorzubereiten, indem vor allem die Felder initialisiert werden. Weil ein Basisklassenkonstruktor zusätzliche Felder von abgeleiteten Klassen nicht kennt, ist ein Vererben von Konstruktoren an abgeleitete Klassen nicht sinnvoll. Ihre Konstruktoren muss eine abgeleitete Klasse also neu definieren, wobei es aber möglich ist, einen Basisklassenkonstruktor (z. B. zur Initialisierung von geerbten Instanzvariablen) einzuspannen (siehe Beispiel).

In C# ist **keine Mehrfachvererbung** möglich: Man kann also in einer Klassendefinition nur *eine* Basisklasse angeben. Im Sinne einer realitätsnahen Modellierung wäre eine Mehrfachvererbung gelegentlich durchaus wünschenswert. So könnte z. B. die Klasse `Receiver` von den Klassen `Tuner` und `Amplifier` erben. Man hat auf die in anderen Programmiersprachen (z. B. C++) erlaubte Mehrfachvererbung bewusst verzichtet, um von vornherein den kritischen Fall auszuschließen, dass eine abgeleitete Klasse gleichnamige *Instanzvariablen* von mehreren Klassen erbt, woraus Mehrdeutigkeiten und Fehler resultieren können (zum sogenannten *Deadly Diamond of Death* siehe Kreft & Langer 2014).

Einen gewissen Ersatz bieten die im Kapitel 9 behandelten Schnittstellen (Interfaces), weil ...

- bei Schnittstellen die Mehrfachvererbung erlaubt ist,
- und außerdem eine Klasse mehrere Schnittstellen implementieren darf.

Statische Member werden in C# genauso vererbt wie instanzbezogene.

7.3 base-Konstruktoren und Initialisierungs-Sequenzen

Zwar werden Konstruktoren nicht vererbt, doch ist bei der Entstehung einer Instanz eines abgeleiteten Typs aus *jeder* Basisklasse entlang der Ahnenreihe ein Konstruktor durch impliziten oder expliziten Aufruf beteiligt. Das folgende Programm erzeugt ein Objekt aus der Klasse `Figur` und ein Objekt aus der von `Figur` abgeleiteten Klasse `Kreis`, wobei die beteiligten Konstruktoren ihre Tätigkeit melden:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  var fig = new Figur(50.0, 50.0);`
`  Console.WriteLine();`
`  var krs = new Kreis(150.0, 200.0, 50.0);`
`  }`
`}`
	`Figur-Konstruktor`
	
	`Figur-Konstruktor`
	`Kreis-Konstruktor`

Vom ebenfalls beteiligten **Object**-Konstruktor ist nichts zu sehen, weil die BCL-Designer natürlich keine Kontrollausgabe eingebaut haben. Wir werden die Beiträge der einzelnen Konstruktoren bei der Erstellung eines neuen `Kreis`-Objekts gleich noch genauer analysieren.

Wie schon im Abschnitt 7.2 zu sehen war, erledigt man den *expliziten* Aufruf eines Basisklassenkonstruktors im Kopfbereich eines Unterklassenkonstruktors über das Schlüsselwort **base**, z. B.:

```
public Kreis(double x, double y, double rad) : base(x, y) {
     if (rad >= 0.0)
        radius = rad;
  Console.WriteLine("Kreis-Konstruktor");
}
```

Dadurch ist es möglich, geerbte Instanzvariablen zu initialisieren, die in der Basisklasse als **private** deklariert sind.

In einem Unterklassenkonstruktor *ohne* **base**-Klausel ruft der Compiler automatisch den *parameterfreien* Konstruktor der Basisklasse auf. Fehlt ein solcher, weil der Programmierer einen eigenen, parametrisierten Konstruktor erstellt und nicht durch einen expliziten parameterlosen Konstruktor ergänzt hat, dann protestiert der Compiler, z. B.:

```
Kreis.cs(12,12): error CS1501: Keine Überladung für die Methode Figur erfor-
dert 0-Argumente
```

Es gibt zwei offensichtliche Möglichkeiten, das Problem zu lösen:

- Im Unterklassen-Konstruktor über das Schlüsselwort **base** einen parametrisierten Basisklassen-Konstruktor aufrufen.
- In der Basisklasse einen parameterfreien Konstruktor definieren.

Der parameterfreie Basisklassenkonstruktor wird auch vom (implizit definierten) Standardkonstruktor einer abgeleiteten Klasse aufgerufen.

Es ist klar, dass ein Basisklassenkonstruktor mit passender Signatur nicht nur vorhanden, sondern auch verfügbar sein muss (z. B. dank **public**-Zugriffsmodifikator).

Beim Erzeugen eines Unterklassenobjekts laufen folgende Initialisierungs-Maßnahmen ab:

- Alle Instanzvariablen (auch die geerbten) werden (auf dem Heap) angelegt und mit den typspezifischen Nullwerten initialisiert.
- Der Unterklassenkonstruktor führt nacheinander folgende Aktionen aus:
  - Die in Felddeklarationen der Unterklasse enthaltenen Initialisierungen werden ausgeführt. Den zugehörigen IL-Code erzeugt der Compiler automatisch.
  - Es folgt der Aufruf eines Basisklassenkonstruktors.
  - Nach Beendigung des Basisklassenkonstruktors wird der Rumpf des Unterklassen-Konstruktors ausgeführt.
- Im aufgerufenen Basisklassenkonstruktor läuft dieselbe Sequenz ab (Instanzvariablen der Klasse gemäß Deklaration initialisieren, Aufruf eines Basisklassenkonstruktors, Anweisungsteil des Konstruktors).
- Das Verfahren wird bis zur obersten Hierarchieebene fortgesetzt, wobei natürlich der **Object**-Konstruktor keinen Basisklassenkonstruktor aufruft.

Betrachten wir zum Beispiel, was beim Erzeugen eines `Kreis`-Objekts mit dem Konstruktor-Aufruf

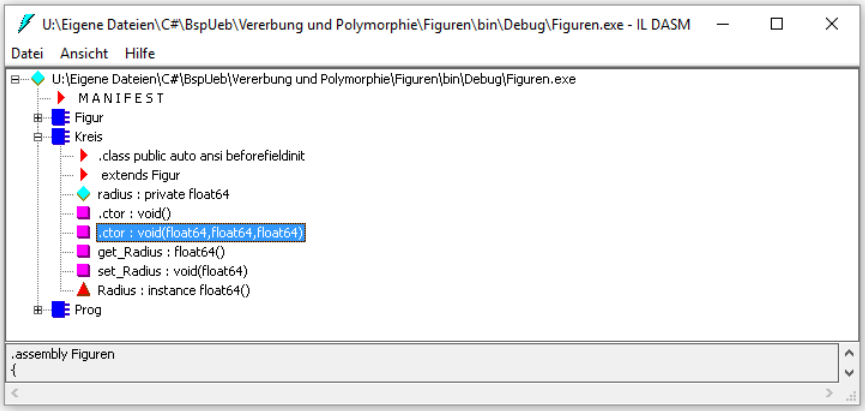
```
Kreis(150.0, 200.0, 50.0)
```

geschieht:

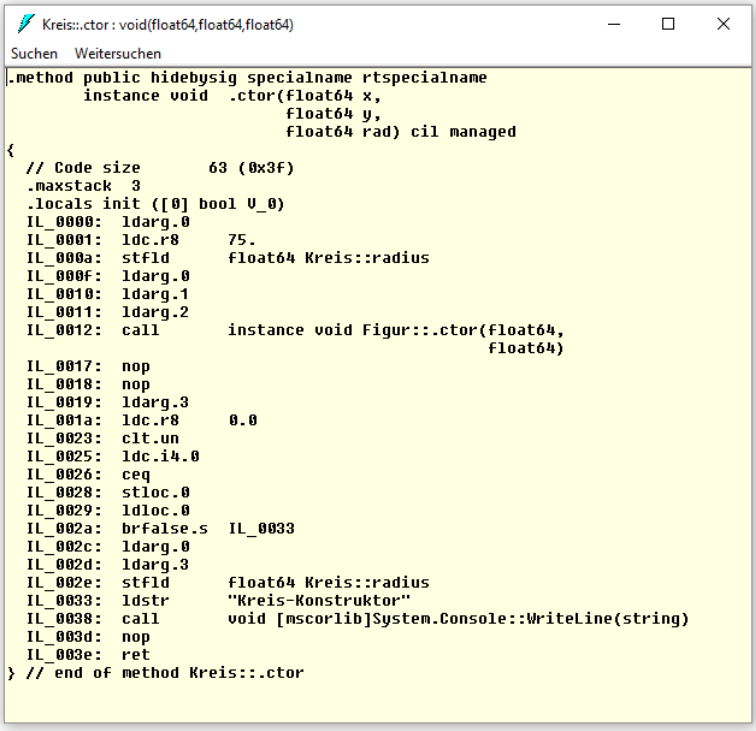
- Alle Instanzvariablen (auch die geerbten) werden angelegt und mit den typspezifischen Nullwerten initialisiert.
- Der `Kreis`-Konstruktor führt die Initialisierung `radius = 75.0` gemäß `Kreis`-Klassendefinition aus. Den dazu erforderlichen IL-Code hat der Compiler automatisch erstellt.
- Der explizit über das Schlüsselwort **base** aufgerufene `Figur`-Konstruktor mit Positionsparametern startet und führt die Initialisierungen `xpos = 100.0` sowie `ypos = 100.0` gemäß `Figur`-Klassendefinition aus. Den dazu erforderlichen IL-Code hat der Compiler automatisch erstellt.

- Der parameterlose **Object**-Konstruktor startet. Der bleibt allerdings ziemlich passiv:[1]
  ```
  public Object() { }
  ```
  Es gibt auch keine Instanzvariablen in der **Object**-Klassendefinition, die einen Initialisierungswert erhalten könnten.[2]
- Der Rumpf des parametrisierten `Figur`-Konstruktors wird ausgeführt, wobei `xpos` und `ypos` die Aktualparameterwerte 150 bzw. 200 erhalten.
- Der Rumpf des parametrisierten `Kreis`-Konstruktors wird ausgeführt, wobei `radius` den Aktualparameterwert 50 erhält.

Wie eine Inspektion der Klasse `Kreis` mit dem Hilfsprogramms **ILDasm** zeigt, haben die beiden Konstruktoren den Namen **.ctor**:



Wir öffnen per Doppelklick den IL-Code zum parametrisierten Konstruktor und sehen die oben beschriebene Sequenz (Instanzvariable `radius` gemäß Deklaration initialisieren, Aufruf des parametrisierten `Figur`-Konstruktors, Anweisungsteil des `Kreis`-Konstruktors):



[1] Die **Object**-Konstruktoren aus .NET 5.0 und aus dem .NET Framework 4.8 sind identisch.

[2] Auch diese Aussage gilt sowohl für .NET 5.0 wie für alle .NET Framework - Versionen.

Für die automatische Null-Initialisierung (vgl. Abschnitt 5.2.3) ist kein IL-Code erforderlich.

Neben Instanzkonstruktoren werden auch die folgenden Klassenmitglieder *nicht* vererbt:

- Statische Konstruktoren (siehe Abschnitt 5.6.4)
- Finalisierer (siehe Abschnitt 5.4.4)

7.4 Der Zugriffsmodifikator protected

Auf **private**-Member einer Basisklasse haben Methoden einer abgeleiteten Klasse (sowie die Methoden beliebiger anderer Klassen) *keinen* direkten Zugriff. Um abgeleiteten Klassen besondere Rechte einzuräumen, bietet C# den Zugriffsmodifikator **protected**, welcher den direkten Zugriff durch die eigene Klasse *und* durch alle (direkt oder indirekt) *abgeleitete* Klassen erlaubt, z. B.:

```
using System;
public class Figur {
   protected double xpos = 100.0, ypos = 100.0;
   public Figur(double x, double y) {
      if (x >= 0.0 && y >= 0.0) {
         xpos = x;
         ypos = y;
      }
      Console.WriteLine("Figur-Konstruktor");
   }
   public Figur() { }
   public void Wo() {
      Console.WriteLine("\nOben Links:\t(" + xpos + ", " + ypos + ") ");
   }
}
```

Weil die Basisklasse `Figur` ihre Instanzvariablen `xpos` und `ypos` nun als **protected** deklariert, können sie in der `Kreis`-Methode `OLE2Zen()`, welche die obere linke Ecke in das aktuelle Zentrum verschiebt, verändert werden:

```
using System;
public class Kreis : Figur {
   double radius = 75.0;
   public Kreis(double x, double y, double rad) : base(x, y) {
      if (rad >= 0.0)
         radius = rad;
      Console.WriteLine("Kreis-Konstruktor");
   }
   public Kreis() {}

   public double Radius {
      get {return radius;}
      set {if (value >= 0.0) radius = value;}
   }

   public void OLE2Zen() {
      xpos = xpos + radius;
      ypos = ypos + radius;
   }
}
```

Es ist zu beachten, dass hier *geerbte Instanzvariablen* von `Kreis`-Objekten verändert werden. Auf das `xpos`-Feld eines `Figur`-Objekts haben die Methoden der `Kreis`-Klasse jedoch *keinen* direkten Zugriff.

Für Methoden *fremder* Klassen sind **protected**-deklarierte Member ebenso gesperrt wie private, z. B.:

```
using System;
class Prog {
   static void Main() {
      Kreis krs = new Kreis(10.0, 10.0, 5.0);
      //krs.xpos = 77.7;    // In der Prog-Methode ist der Zugriff verboten.
      krs.OLE2Zen();        // In der Kreis-Methode ist der Zugriff erlaubt.
   }
}
```

Der Modifikator **protected** ist nicht nur bei Feldern erlaubt, sondern bei beliebigen Mitgliedern, z. B. bei (statischen) *Methoden*, z. B.:

```
static protected void ProSt() {
   Console.WriteLine("Protected und statisch!");
}
```

7.5 Erbstücke durch spezialisierte Varianten verdecken

7.5.1 Methoden und andere ausführbare Member verdecken

Eine geerbte Basisklassenmethode kann in einer abgeleiteten Klasse durch eine Methode mit gleicher Signatur verdeckt (ausgeblendet) werden. Zwei Methoden haben genau dann dieselbe Signatur, wenn die Namen und die Parameterlisten (hinsichtlich Datentyp und Transfermodus aller Formalparameter) übereinstimmen, während z. B. die Rückgabetypen keine Rolle spielen (vgl. Abschnitte 5.3.5 und 8.5).

Bisher steht in der `Kreis`-Klasse zur Ortsangabe die geerbte Methode `Wo()` zur Verfügung, welche die Position der linken oberen Ecke eines Objekts ausgibt. In der `Kreis`-Klasse kann aber eine bessere Ortsangabenmethode realisiert werden, weil hier auch die rechte untere Ecke definiert ist:[1]

```
using System;
public class Kreis : Figur {
   . . .
   public new void Wo() {
      base.Wo();
      Console.WriteLine("Unten Rechts:\t(" + (xpos + 2 * radius) +
                        ", " + (ypos + 2 * radius) + ")");
   }
}
```

Ist die Deklaration einer verdeckenden Methode tatsächlich intendiert, sollte mit dem Modifikator **new** im Definitionskopf die folgende Warnung des Compilers vermieden werden:[2]

```
Kreis.cs(14,15): warning CS0108: "Kreis.Wo()" blendet den vererbten Member
        "Figur.Wo()" aus. Verwenden Sie das new-Schlüsselwort, wenn das
        Ausblenden vorgesehen war.
```

Mit dieser Warnung will der Compiler ein ungeplantes Verdecken verhindern.

Im Anweisungsteil der neuen Methode kann man sich oft durch Rückgriff auf die verdeckte Basisklassenmethode die Arbeit erleichtern, wobei erneut das Schlüsselwort **base** zum Einsatz kommt, z. B.:

```
base.Wo();
```

---

[1] Falls Sie sich über die Berechnungsvorschrift für die Y-Koordinate der rechten unteren Kreis-Ecke wundern: Bei der Grafikausgabe von Computersystemen ist die Position (0, 0) meist in der oberen linken Ecke des Bildschirms bzw. des aktuellen Fensters angesiedelt. Die X-Koordinaten wachsen (wie aus der Mathematik gewohnt) von links nach rechts, während die Y-Koordinaten von oben nach unten wachsen. Wir wollen uns im Hinblick auf die in absehbarer Zukunft anstehende Programmierung graphischer Benutzeroberflächen schon jetzt daran gewöhnen.

[2] Dieser Modifikator darf nicht mit dem *Operator* **new** verwechselt werden.

Es liegt *keine* Verdeckung vor, wenn in der Unterklasse eine Methode mit gleichem Namen, aber abweichender Parameterliste definiert wird. In diesem Fall sind die beiden Signaturen verschieden, und es handelt sich um eine *Überladung* (vgl. Abschnitt 5.3.5).

Das folgende Programm schickt an eine `Figur` und an einen `Kreis` jeweils die Nachricht `Wo()`, und beide zeigen ihr artspezifisches Verhalten:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  var f = new Figur(10.0, 20.0);`
`  f.Wo();`
`  var k = new Kreis(50.0, 50.0, 10.0);`
`  k.Wo();`
`  }`
`}`
	`Oben Links:  (10, 20)`
	
	`Oben Links:  (50, 50)`
	`Unten Rechts:  (70, 70)`

Im nächsten Programm wird ein `Kreis`-Objekt zweimal aufgefordert, die Methode `Wo()` auszuführen. Im ersten Aufruf wird eine Referenzvariable vom Typ `Kreis` verwendet, im zweiten Aufruf eine Referenzvariable vom Typ `Figur`. Das Objekt führt beim ersten Aufruf die `Kreis`-Methode und im zweiten Aufruf die `Figur`-Methode aus:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  Kreis k = new Kreis(50.0, 50.0, 10.0);`
`  k.Wo();`
`  Figur f = k;`
`  f.Wo();`
`  }`
`}`
	`Oben Links:  (50, 50)`
	`Unten Rechts:  (70, 70)`
	
	`Oben Links:  (50, 50)`

Hier drohen Programmierfehler.

Wird eine öffentliche Basisklassenmethode durch eine *private* Unterklassenmethode verdeckt, dann wird die verdeckende Methode nur *klassenintern* verwendet, und in der Schnittstelle der abgeleiteten Klasse bleibt das signaturgleiche Erbstück mit **public**-Zugriff erhalten, z. B.:

Quellcode	Ausgabe

`using System;`
`class Basisklasse {`
`  public void NenneTyp() {`
`  Console.WriteLine("Basisklasse");`
`  }`
`}`
 	
`class Spezialklasse : Basisklasse {`
`  new void NenneTyp() {`
`  Console.WriteLine("Spezialklasse");`
`  }`
`  public void DeinTyp() {`
`  NenneTyp();`
`  }`
`}`
 	
`class Prog {`
`  static void Main() {`
`  var a = new Spezialklasse();`
`  a.NenneTyp();`
`  a.DeinTyp();`
`  }`
`}`
	`Basisklasse`
	`Spezialklasse`

Eine solche Konstruktion ist aber potentiell verwirrend. Im Beispiel fordert die **Main()** - Methode der Klasse `Prog` ein Objekt der `Spezialklasse` auf, die Methode `NenneTyp()` auszuführen. Dabei wird die Methode aus der `Basisklasse` ausgeführt, weil die Methode `NenneTyp()` in der Spezialklasse als **private** deklariert ist. Wenn hingegen in der öffentlichen `Spezialklassen`-Methode `DeinTyp()` die Methode `NenneTyp()` aufgerufen wird, dann kommt die Version der `Spezialklasse` zum Einsatz. Dabei wurde in beiden Fällen eine Referenzvariable vom Typ der `Spezialklasse` verwendet.

Neben objektbezogenen können auch *statische* Methoden verdeckt werden, wobei die verdeckte Basisklassenvariante durch Voranstellen des Klassennamens angesprochen werden kann, z. B.:

Quellcode	Ausgabe

`using System;`
`class Basisklasse {`
`  public static void NenneTyp() {`
`  Console.WriteLine("Basisklasse");`
`  }`
`}`
`class Spezialklasse : Basisklasse {`
`  public new static void NenneTyp() {`
`  Console.Write("Spezialklasse\n abgeleitet von: ");`
`  Basisklasse.NenneTyp();`
`  }`
`}`
`class Prog {`
`  static void Main() {`
`  Basisklasse.NenneTyp();`
`  Spezialklasse.NenneTyp();`
`  }`
`}`
	`Basisklasse`
	`Spezialklasse`
	`  abgeleitet von: Basisklasse`

Das Verdecken ist nicht nur bei Methoden erlaubt, sondern auch bei Eigenschaften, Indexern und Ereignissen.

Insgesamt ist das Verdecken von Methoden und anderen ausführbaren Membern eine relativ komplizierte Angelegenheit, die zu einem schlecht nachvollziehbaren Programmverhalten führen kann. Das im Abschnitt 7.7 erläuterte Überschreiben von Methoden ist insofern einfacher, als ein Objekt aus einer abgeleiteten Klasse unabhängig vom deklarierten Typ der zur Kommunikation verwendeten Referenzvariablen immer das in der eigenen Klasse definierte Verhalten zeigt. Allerdings ist das Überschreiben mit Risiken verbunden, wenn der Designer der abgeleiteten Klasse die Basisklasse nicht genau kennt (siehe Abschnitt 7.9).

7.5.2 Felder verdecken

Auch geerbte Instanz- und Klassenvariablen lassen sich verdecken (ausblenden), was aber im Sinne eines gut nachvollziehbaren Quelltextes nur in Ausnahmefällen geschehen sollte. Verwendet man z. B. in der abgeleiteten Klasse AK für eine Instanzvariable einen Namen, der bereits eine Variable der beerbten Basisklasse BK bezeichnet, dann wird die Basisklassenvariable verdeckt. Sie ist jedoch weiterhin vorhanden und kommt in folgenden Situationen zum Einsatz:

- Von BK geerbte Methoden greifen weiterhin auf die BK-Variable zu, während die zusätzlichen Methoden der AK-Klasse auf die AK-Variable zugreifen.
- In AK-Methoden steht die verdeckte Variante über das Schlüsselwort **base** zur Verfügung.

Im folgenden Beispielprogramm führt ein AK-Objekt eine BK- und eine AK-Methode aus, um die beschriebenen Zugriffsvarianten zu demonstrieren:

Quellcode	Ausgabe

```
using System;
class BK {
   protected string x = "Bast";
   public void BM() {
      Console.WriteLine("x in BK-Methode:\t"+x);
   }
}

class AK : BK {
   new int x = 333;
   public void AM() {
      Console.WriteLine("x in AK-Methode:\t"+x);
      Console.WriteLine("base-x in AK-Methode:\t"+
                        base.x);
   }
}

class Prog {
   static void Main() {
      AK ako = new AK();
      ako.BM();
      ako.AM();
   }
}
```

```
x in BK-Methode:        Bast
x in AK-Methode:        333
base-x in AK-Methode:   Bast
```

Ist die Deklaration einer verdeckenden Variablen tatsächlich intendiert, sollte der Modifikator **new** angeben werden, um die folgende Warnung des Compilers zu vermeiden:[1]

```
Verdecken.cs(10,6,10,7): warning CS0108: "AK.x" blendet den vererbten Member "BK.x"
aus. Verwenden Sie das new-Schlüsselwort, wenn das Ausblenden vorgesehen war.
```

[1] Dieser Modifikator darf nicht mit dem *Operator* **new** verwechselt werden.

Mit diesem Hinweis will der Compiler ein ungeplantes Verdecken verhindern. In der Regel sollte der Hinweis zum Anlass genommen werden, die Namenskollision durch eine Umbenennung zu beseitigen, um die Lesbarkeit des Quellcodes zu verbessern.

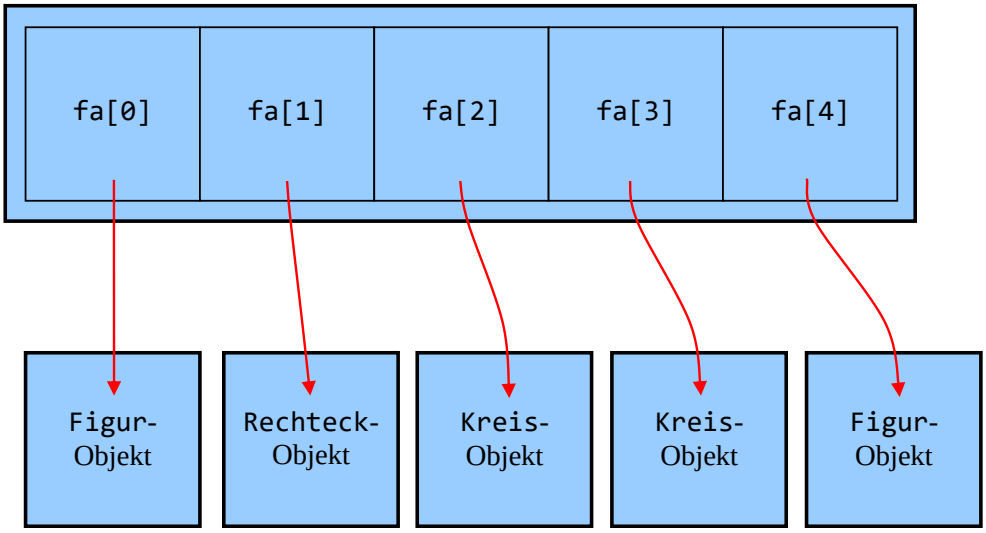
7.6 Verwaltung von Objekten über Basisklassenreferenzen

Eine Basisklassen-Referenzvariable darf die Adresse eines beliebigen Unterklassenobjekts aufnehmen. Schließlich besitzt Letzteres die komplette Ausstattung der Basisklasse und kann z. B. auf dort definierte Methodenaufrufe geeignet reagieren. Ein Objekt steht nicht nur zur eigenen Klasse in der „ist-ein“ - Beziehung, sondern erfüllt diese Relation auch in Bezug auf die direkte Basisklasse sowie in Bezug auf alle indirekten Basisklassen in der Ahnenreihe. Angewendet auf das Beispiel im Abschnitt 7.2 ergibt sich die sehr plausible Feststellung, dass jeder Kreis auch eine Figur ist.

Andererseits verfügt ein Basisklassenobjekt in der Regel *nicht* über die Ausstattung von abgeleiteten (erweiterten bzw. spezialisierten) Klassen. Daher ist es sinnlos und verboten, die Adresse eines Basisklassenobjektes in einer Unterklassen-Referenzvariablen abzulegen.

Über Referenzvariablen vom Typ einer *gemeinsamen* Basisklasse lassen sich also Objekte aus unterschiedlichen Klassen verwalten. Im Rahmen eines Grafikprogramms kommt vielleicht ein Array mit dem Elementtyp `Figur` zum Einsatz, dessen Elemente auf Objekte aus der Basisklasse oder aus einer abgeleiteten Klasse wie `Kreis` oder `Rechteck` zeigen:

Array `fa` mit Elementtyp `Figur`



Die abgebildete Konstellation wird im folgenden Programm realisiert:

```
using System;
class Prog {
   static void Main() {
      Figur[] fa = new Figur[5];
      fa[0] = new Figur(10.0, 10.0);
      fa[1] = new Rechteck(20.0, 20.0, 20.0, 20.0);
      fa[2] = new Kreis(30.0, 30.0, 30.0);
      fa[3] = new Kreis(40.0, 40.0, 30.0);
      fa[4] = new Figur(50.0, 50.0);
      foreach (Figur e in fa)
         e.Wo();
   }
}
```

Die Elemente im Array `fa` werden aufgefordert, die in der Klasse `Figur` definierte Methode `Wo()` auszuführen (vgl. Abschnitte 7.2 und 7.5.1). Bei Ansprache per *Basisklassenreferenz* führen die

Objekte (aus den Klassen `Figur`, `Kreis` oder `Rechteck`) auch dann die Basisklassenvariante der `Wo()` - Methode aus, wenn eine *verdeckende*, artspezifische `Wo()` - Methode vorhanden ist:

```
Oben Links:     (10, 10)

Oben Links:     (20, 20)

Oben Links:     (30, 30)

Oben Links:     (40, 40)

Oben Links:     (50, 50)
```

Im Abschnitt 7.7 über die *Polymorphie* werden Sie erfahren, dass man in der Basisklasse eine *virtuelle* Methode definieren und diese in den abgeleiteten Klassen *überschreiben* muss, damit auch bei Ansprache per Basisklassenreferenz ein artspezifisches Verhalten resultiert.

Über eine `Figur`-Referenzvariable, die auf ein `Kreis`-Objekt zeigt, sind *Erweiterungen* der `Kreis`-Klasse *nicht* unmittelbar zugänglich. Wenn (auf eigene Verantwortung des Programmierers) eine Basisklassenreferenz als Unterklassenreferenz behandelt werden soll, um eine unterklassenspezifische Methode, Eigenschaft oder Variable anzusprechen, dann muss eine explizite Typumwandlung vorgenommen werden, z. B.:

```
((Kreis)fa[1]).Radius
```

Geschieht dies zu Unrecht, tritt ein Ausnahmefehler vom Typ **InvalidCastException** auf, z. B.:

```
Unbehandelte Ausnahme: System.InvalidCastException: Das Objekt des Typs "Figur" kann
nicht in Typ "Kreis" umgewandelt werden.
```

Bisher haben wir die explizite Typumwandlung nur auf Werttypen (meist elementare Typen) angewendet, sie spielt aber auch bei Referenztypen eine wichtige Rolle. Welche expliziten Konvertierungen erlaubt sind, ist der C# - Sprachspezifikation (ECMA 2017, Abschnitt 11.3) zu entnehmen. Im konkreten Fall wird der deklarierte Typ (`Figur`) durch eine Spezialisierung bzw. Ableitung (`Kreis`) ersetzt. Der Compiler erlaubt die Konvertierung, übernimmt jedoch keine Verantwortung dafür.

Im Zweifelsfall sollte man per **is** - (Typtest-)Operator überprüfen, ob das referenzierte Objekt tatsächlich den vermuteten Laufzeittyp besitzt, z. B.:

```
foreach (Figur e in fa) {
   e.Wo();
   if (e is Kreis)
      Console.WriteLine("Radius:         " + ((Kreis)e).Radius);
}
```

Weil der **is**-Operator nicht den deklarierten (statischen) Typ prüft, sondern den Laufzeittyp (den dynamischen Typ), resultiert bei einer Referenzvariablen mit dem Wert **null** immer das Ergebnis false, z. B.:

Quellcode	Ausgabe

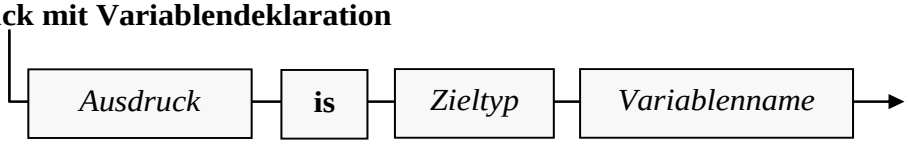
`using System;`
`class Prog {`
`  static void Main() {`
`  Kreis krs = null;`
`  Console.WriteLine(krs is Kreis);`
`  }`
`}`
	`False`

Seit C# 7.0 ist es möglich, in einem logischen Ausdruck mit **is**-Operator eine Variable vom verifizierten Typ zu deklarieren und mit dem Ausdruck zu initialisieren, sodass die explizite Typumwandlung überflüssig wird, z. B.:

```
foreach (Figur e in fa) {
   e.Wo();
   if (e is Kreis kr)
      Console.WriteLine("Radius:         " + kr.Radius);
}
```

Im folgenden Syntaxdiagramm wird die vom **is**-Operator abhängige Variablendeklaration und -initialisierung auf dem Stand von C# 7 beschrieben:

is-Ausdruck mit Variablendeklaration



Das Prüfergebnis des **is**-Operators ist positiv, wenn der *Laufzeittyp* des Ausdrucks ...

- exakt den Zieltyp hat,
- oder vom Zieltyp abgeleitet ist,
- oder das als Zieltyp angegebene Interface implementiert.

In C# 8 und 9 profitiert der **is**-Ausdruck von den erweiterten Mustervergleichs-Optionen, die im Abschnitt 15.1 behandelt werden. Im folgenden Beispiel wird für ein Objekt geprüft, ob es sich um eine `Kreis`-Instanz mit dem `Radius` 5 handelt:

```
Console.WriteLine(krs is Kreis { Radius: 5.0 });
```

Eine weitere Möglichkeit zur Vermeidung der expliziten Typumwandlung

```
((Kreis)e).Radius
```

bietet der **as**-Operator:

```
(e as Kreis).Radius
```

Die wichtigsten Regeln für den **as**-Operator:

- Der Zieltyp im rechten Operanden muss ein Referenztyp oder ein **null**-fähiger Werttyp sein (siehe Abschnitt 8.3). Als **null**-fähig bezeichnet man einen Werttyp dann, wenn neben den normalen Werten auch der Ausnahmewert **null** zur Verfügung steht, was z. B. bei den elementaren Datentypen *nicht* der Fall ist.
- Im linken Operanden verlangt der Compiler einen Ausdruck, dessen Wert potentiell in den Zieltyp konvertiert werden kann. Dies ist z. B. der Fall, wenn der Typ des Ausdrucks eine Basisklasse des Zieltyps ist (wie im obigen Beispiel). Weitere Details finden sich in der C# - Sprachspezifikation (ECMA 2017, Abschnitt 12.11.12).
- Stellt sich zur Laufzeit eine Konvertierung als unmöglich heraus, liefert der **as**-Operator im Unterschied zum gewohnten Typumwandlungsoperator *keinen* Ausnahmefehler vom Typ **InvalidCastException**, sondern den Ergebniswert **null**. Laut C# - Sprachspezifikation lässt sich das Verhalten des **as**-Operators im Beispiel

  ```
  e as Kreis
  ```

  mit Hilfe des Typumwandlungs- und des Konditionaloperators so beschreiben:

  ```
  (e is Kreis) ? (Kreis)e : null
  ```

  Im obigen Beispiel

  ```
  (e as Kreis).Radius
  ```

  kommt es zum Laufzeitfehler vom Typ **NullReferenceException**, wenn die Variable `e` nicht auf einen `Kreis` zeigt.

Lahres & Rayman (2009, Abschnitt 5.1.5) bewerten das explizite Konvertieren in einen spezielleren Typ als Notlösung, die durch ein gutes Programmdesign vermieden werden sollte.

Konsistent mit dem **is**-Operator, der nicht den deklarierten (statischen) Typ prüft, sondern den Laufzeittyp (den dynamischen Typ), liefert die schon in der Urahnklasse **Object** definierte Methode **GetType()** den Laufzeittyp ihres Parameters. Das folgende Programm

```
using System;
class Program {
    static void Main() {
        object[] oar = new object[3] { new object(), "123", 13 };
        foreach(var o in oar)
            Console.WriteLine($"{o, 15} hat den Laufzeittyp {o.GetType()}");
    }
}
```

produziert die Ausgabe:

```
System.Object hat den Laufzeittyp System.Object
          123 hat den Laufzeittyp System.String
           13 hat den Laufzeittyp System.Int32
```

7.7 Polymorphie (Überschreiben von Methoden)

Eine abgeleitete Klasse kann mit Hilfe gleich zu beschreibender Schlüsselwörter eine geerbte Instanzmethode *überschreiben*, statt sie zu *verdecken*. Wird ein Unterklassenobjekt über eine Variable vom Unterklassentyp referenziert, zeigt es bei überschreibenden *und* bei verdeckenden Methoden das Unterklassenverhalten. Bei Ansprache über eine Referenz vom *Basisklassentyp* gilt hingegen:

- Bei einer Verdeckung kommt die *Basisklassenvariante* zum Einsatz.
- Bei einer Überschreibung wird die *Unterklassenvariante* benutzt.

Während bei einer *Verdeckung* die auszuführende Methode schon beim Übersetzen festliegt, wird im Fall der *Überschreibung* erst zur Laufzeit die passende Methode gewählt. Man spricht hier von einer *dynamischen* oder *späten Bindung*. Grundsätzlich hat die späte Bindung einen erhöhten Zeitaufwand zur Folge, der jedoch kaum jemals praxisrelevant ist.[1]

Werden Objekte aus verschiedenen Klassen über Referenzvariablen eines gemeinsamen Basistyps verwaltet, dann sind nur Methoden nutzbar, die schon in der Basisklasse definiert sind. Bei überschreibenden Methoden reagieren die Objekte aber jeweils unterklassentypisch auf dieselbe Botschaft. Genau dieses Phänomen bezeichnet man als **Polymorphie**. Wer sich hier mit einem exotischen und nutzlosen Detail konfrontiert glaubt, sei an die Auffassung von Alan Kay erinnert, der wesentlich zur Entwicklung der objektorientierten Programmierung beigetragen hat. Er zählt die Polymorphie neben der Datenkapselung und der Vererbung zu den Grundelementen der objektorientierten Programmierung (siehe Abschnitt 5.1.1).

Zur Demonstration der Polymorphie definieren wir in der Basisklasse des Figurenbeispiels die Methode `Wo()` mit dem Modifikator **virtual** als überschreibbar:

[1] Siehe Einschätzungen in https://stackoverflow.com/questions/9937150/performance-impact-of-virtual-methods und Messungen (bei C++) in: https://stackoverflow.com/questions/449827/virtual-functions-and-performance-c

```
using System;
public class Figur {
   protected double xpos = 100.0, ypos = 100.0;
   public Figur(double x, double y) {
      if (x >= 0.0 && y >= 0.0) {
         xpos = x;
         ypos = y;
      }
   }
   public Figur() { }

   public virtual void Wo() {
      Console.WriteLine("\nOben Links:\t(" + xpos + ", " + ypos + ") ");
   }
}
```

In der abgeleiteten Klasse `Kreis` wird mit dem Schlüsselwort **override** das Überschreiben der geerbten `Wo()` - Methode angeordnet:

```
using System;
public class Kreis : Figur {
   double radius = 75.0;
   public Kreis(double x, double y, double rad) : base(x, y) {
      if (rad >= 0.0)
         radius = rad;
   }
   public Kreis() { }

   public double Radius {
      get {return radius;}
      set {if (value >= 0.0) radius = value;}
   }

   public override void Wo() {
      base.Wo();
      Console.WriteLine("Unten Rechts:\t(" + (xpos + 2.0 * radius) +
                ", " + (ypos + 2.0 * radius) + ")");
   }
}
```

Ein Array vom Typ `Figur` kann nach den Erläuterungen im Abschnitt 7.6 Referenzen auf Figuren und Kreise aufnehmen, z. B.:

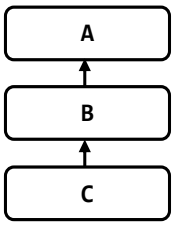
```
using System;
class Prog {
  static void Main() {
    Figur[] fa = new Figur[3];
    fa[0] = new Figur();
    fa[1] = new Kreis();
    for (int i = 0; i < 2; i++) {
      fa[i].Wo();
      if (fa[i] is Kreis)
        Console.WriteLine("Radius:         " + ((Kreis)fa[i]).Radius);
    }
    Console.Write("\nWollen Sie zum Abschluss noch eine Figur oder einen Kreis erleben?" +
     "\nWählen Sie durch Abschicken von \"f\" oder \"k\": ");
    if (Console.ReadLine().ToUpper()[0] == 'F')
      fa[2] = new Figur();
    else
      fa[2] = new Kreis();
    fa[2].Wo();
  }
}
```

Beim Ausführen der virtuellen und überschriebenen `Wo()` - Methode durch ein per Basisklassenreferenz angesprochenes Objekt stellt das Laufzeitsystem die tatsächliche Klassenzugehörigkeit (den *dynamischen* Typ) des angesprochenen Objekts fest und wählt die passende Methode:

```
Oben Links:     (100, 100)

Oben Links:     (100, 100)
Unten Rechts:   (250, 250)
Radius:         75

Wollen Sie zum Abschluss noch eine Figur oder einen Kreis erleben?
Wählen Sie durch Abschicken von "f" oder "k": k

Oben Links:     (100, 100)
Unten Rechts:   (250, 250)
```

Zum „Beweis“, dass tatsächlich eine späte Bindung stattfindet, darf im Beispielprogramm der Laufzeittyp des Array-Elements `fa[2]` vom Benutzer festgelegt werden.

Bei komplizierten Erbschaftsverhältnissen kann es passieren, dass beim Aufruf einer überschreibenden Methode durch eine Basisklassenreferenz *keine* Polymorphie stattfindet. Im folgenden Beispielprogramm (nach Mössenböck, 2019, S. 96) mit den Vererbungsbeziehungen



überschreibt die Klasse `C` die ursprünglich in der Klasse `A` virtuell definierte Methode `M()`. Die von `A` abgeleitete Klasse `B` verdeckt aber die Methode `M()` und deklariert die Ersetzung als virtuell:

```
public new virtual void M() { ... }
```

Wird unter dieser Konstellation ein `C`-Objekt per `A`-Referenz beauftragt, die Methode `M()` auszuführen, dann kommt die Variante der Klasse `A` zum Einsatz (keine Polymorphie). Wird dasselbe Objekt per `B`-Referenz beauftragt, die Methode `M()` auszuführen, dann kommt polymorph die überschreibende Variante der Klasse `C` zum Einsatz:

Quellcode	Ausgabe

`using System;`
`class A {`
`  public virtual void M() {`
`  Console.WriteLine("M in A");`
`  }`
`}`
 	
`class B : A {`
`  public new virtual void M() {`
`  Console.WriteLine("M in B");`
`  }`
`}`
 	
`class C : B {`
`  public override void M() {`
`  Console.WriteLine("M in C");`
`  }`
`}`
 	
`class Prog {`
`  static void Main() {`
`  C c = new C();`
`  A a = c;`
`  B b = c;`
`  a.M();`
`  b.M();`
`  }`
`}`
	`M in A`
	`M in C`

In Bezug auf den Zugriffsschutz von virtuellen bzw. überschreibenden Methoden gelten die folgenden Regeln:

- Bei virtuellen Methoden ist die Zugriffsstufe **private** verboten.
- Beim Überschreiben darf der Zugriffsschutz nicht geändert werden. Im aktuellen Beispiel ist die Methode `Wo()` in der Klasse `Figur` als **public** deklariert, sodass der folgende Versuch scheitert, die überschreibende Methode der Klasse `Kreis` als **protected** zu deklarieren:

```
protected override void Wo() {
```

> void Kreis.Wo()
>
> CS0507: "Kreis.Wo()": Die Zugriffsmodifizierer können beim Überschreiben des geerbten public-Members "Figur.Wo()" nicht geändert werden.

```
}
```

Das Überschreiben ist nicht nur bei Methoden erlaubt, sondern auch bei Eigenschaften, Indexern und Ereignissen.

Bei *statischen* Methoden sind die Modifikatoren **virtual** und **override** sinnlos und verboten, weil es hier nicht zu einer späten Bindung kommen kann. Das per **new**-Modifikator signalisierte Verdecken einer geerbten statischen Methode ist jedoch sinnvoll und erlaubt (siehe Abschnitt 7.5.1).

7.8 Versiegelte Methoden und Klassen

Gelegentlich möchte man das Überschreiben einer Methode *verhindern*, damit ein per Basisklassenreferenz angesprochenes Unterklassenobjekt das Originalverhalten zeigt. Dient etwa die Methode `Passwd()` einer Klasse `B` zum Abfragen eines Passworts, will der Programmierer eventuell verhindern, dass `Passwd()` in einer von `B` abstammenden Klasse `C` überschrieben wird. Damit führt ein per `B`-Referenz angesprochenes `C`-Objekt garantiert die `B`-Methode `Passwd()` aus.

Meist verhindert man in C# das Überschreiben einer Methode schlicht dadurch, dass man die Methode *nicht* als **virtual** deklariert. Wenn allerdings eine Methode den Modifikator **override** verwendet, um eine virtuelle Basisklassenvariante zu überschreiben, dann ist sie per Voreinstellung ihrerseits virtuell. Um eine solche Methode vor dem Überschreiben zu bewahren, muss sie auch den Modifikator **sealed** (dt.: *versiegelt*) erhalten. Der Modifikator **sealed** ist also nur erforderlich bei Methoden, die auch den Modifikator **override** besitzen, und er ist auch nur in dieser Situation erlaubt. Um den Modifikator **sealed** in einem Beispiel mit der Methode `Passwd()` vorführen zu können, muss also neben den oben vorgestellten Klassen `B` und `C` noch die dritte Klasse `A` ins Spiel kommen:

```
class A {
  public virtual void Passwd() { }
}

class B : A {
  public sealed override void Passwd() { }
}
```

Wer noch Zweifel an der notwendigen Kopplung der Schlüsselwörter **sealed** und **override** hat, wird hoffentlich durch die Betrachtung der folgenden Fälle überzeugt:

- In `B` wird eine Methode oder eine Methodenüberladung erstellt, die in `A` fehlt:
  Um das Überschreiben zu verhindern, lässt man einfach den Modifikator **virtual** weg.
- In `B` wird eine in `A` vorhandene und dort *nicht* als **virtual** definierte Methode verdeckt, wobei mit dem optionalen Modifikator **new** eine Compiler-Warnung verhindert werden sollte:
  Um das Überschreiben der `B`-Methode zu verhindern, lässt man einfach den Modifikator **virtual** weg.
- In `B` wird eine in `A` vorhandene und dort als **virtual** definierte Methode verdeckt, wobei mit dem optionalen Modifikator **new** eine Compiler-Warnung verhindert werden sollte:
  Die `B`-Methode ist nicht virtuell, und man kann das Überschreiben wiederum einfach dadurch verhindern, dass man den Modifikator **virtual** weglässt.
- In `B` wird eine in `A` vorhandene und dort als **virtual** definierte Methode überschrieben, was durch den Modifikator **override** deklariert werden muss:
  In diesem Fall ist die `B`-Methode virtuell, und der Modifikator **sealed** ist erforderlich, wenn das Überschreiben verhindert werden soll.

Die Aussagen zum Versiegeln von Methoden gelten analog für Eigenschaften, Indexer und Ereignisse.

Sicherheitsüberlegungen können auch zum Entschluss führen, eine komplette **Klasse** mit dem Schlüsselwort **sealed** zu fixieren, sodass sie zwar verwendet, aber nicht beerbt werden kann. Microsoft nennt als möglichen Grund die Existenz von sicherheitsrelevanten Geheimnissen, die eine Klasse geerbt hat. Die Erbstücke haben die Schutzstufe **protected**, und diese Schutzstufe kann generell nicht eingeschränkt werden.[1] Durch das Versiegeln wird verhindert, dass die Geheimnisse in einer weiteren Vererbungsgeneration bekannt werden.

Es gibt noch weitere Gründe für das Versiegeln von Klassen:

[1] https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/sealing

- Wenn für eine Klasse die Unveränderlichkeit ihrer Objekte erreicht werden soll (z. B. wegen der Vorteile beim Multithreading), dann muss auch die Vererbung unterbunden werden. Anderenfalls könnte eine abgeleitete Klasse die Unveränderlichkeit unterlaufen, denn ihre Objekte werden vom Compiler überall akzeptiert, wo die Basisklassenreferenz vorgeschrieben ist (z. B. bei einem Parameterdatentyp).
- Das Wissen um die Versiegelung einer Klasse könnte den Compiler zu Optimierungsmaßnahmen veranlassen.

Von den versiegelten BCL-Klassen ist **String** das bekannteste Beispiel:[1]

```
public sealed partial class String ...
```

In diesem Fall wird durch das Versiegeln die Unveränderlichkeit von **String**-Objekten sichergestellt, die z. B. für den internen **String**-Pool wichtig ist (siehe Abschnitt 6.3.1.3) und außerdem von vielen (Bibliotheks)klassen vorausgesetzt wird.

Von der Vererbung ausgeschlossen sind neben den versiegelten Klassen auch die statischen Klassen (vgl. Abschnitt 5.6.5).

7.9 Fragilität und Komplexität

In C# sind beim Ersetzen einer Basisklassenmethode durch eine signaturgleiche Unterklassenmethode (Verdecken oder Überschreiben) mehrere Modifikatoren (Zugriffsmodifikatoren, **virtual**, **override**, **new**, **sealed**) und Regeln im Spiel, die in der folgenden Tabelle zusammengestellt sind:

Ersetzungsart	Unterstützung der Polymorphie	Syntax
Verdecken	Nein	Mit dem Modifikator **new** im Kopf der Unterklassenmethode wird unabhängig von der Basismethodendefinition das Verdecken gewählt. Ohne den Ersetzungsmodifikator **new** kommt es ebenfalls zu einer Verdeckung und außerdem zu einer Warnung des Compilers. Eine verdeckende Methode ist nicht virtuell, kann also nicht überschrieben werden. Beim Methodenaufruf per Basisklassenreferenz kommt die Basisklassenvariante zum Einsatz. Auch statische Methoden können verdeckt werden.
Überschreiben	Ja	Im Kopf der Basisklassenmethode muss der Modifikator **virtual** stehen, und im Kopf der Unterklassenmethode muss der Modifikator **override** stehen. Beim Methodenaufruf per Basisklassenreferenz kommt die Variante der abgeleiteten Klasse zum Einsatz, wenn die Methode nicht in der Vererbungshistorie verdeckt und erneut virtualisiert worden ist. Eine überschreibende Methode darf den Zugriffsschutz der Basisklassenmethode nicht ändern. Eine virtuelle Basisklassenmethode kann mit dem Modifikator **sealed** das Überschreiben verhindern. Statische Methoden können *nicht* überschrieben werden.

Eine virtuelle Basisklassenmethode kann also verdeckt oder überschrieben werden:

---

[1] Aus dem Quellcode der BCL zu .NET 5.

- **Überschreibt** eine abgeleitete Klasse die Methode (Modifikator **override**), dann ist sie auch in der abgeleiteten Klasse virtuell (mit Bedeutung für die nächste Ableitungsgeneration). Den Modifikator **virtual** zusammen mit dem Modifikator **override** anzugeben, ist überflüssig und verboten.
- **Verdeckt** eine abgeleitete Klasse die Methode (Modifikator **new**), ist sie in der abgeleiteten Klasse nicht mehr virtuell, sofern nicht gleichzeitig auch der Modifikator **virtual** vergeben wird.

Bei einer *nicht*-virtuellen Basisklassenmethode ist nur das Verdecken möglich.

Software-Entwickler haben die Mittel und die Pflicht, eine riskante bzw. fehlerhafte Anwendung der Vererbung zu verhindern, z. B.:

- Durch das Überschreiben einer Methode ermöglicht man die Polymorphie, übernimmt aber auch mehr Verantwortung, denn:
  - Eine verdeckende Methode wird nur bei Verwendung einer Unterklassenreferenz aufgerufen (direkt oder indirekt durch andere Methoden in der Unterklasse).
  - Eine überschreibende Methode wird bei Verwendung einer Unterklassen- und bei Verwendung einer Basisklassenreferenz aufgerufen (direkt oder indirekt durch andere Methoden in der Unterklasse oder in einer Basisklasse).

  Vorsicht ist vor allem beim Überschreiben einer Methode aus einer *fremden* Basisklasse geboten. Wenn man den Quellcode der Basisklasse kennt, führt das Überschreiben aber kaum zu einem erhöhten Risiko für Programmierfehler.
- Im Konstruktor einer potentiellen Basisklasse dürfen keine überschreibbaren Methoden aufgerufen werden, weil der Konstruktor der Basisklasse *vor* dem Konstruktor der abgeleiteten Klasse ausgeführt wird (siehe Abschnitt 7.3). Folglich würde die überschreibende Methode der abgeleiteten Klasse *vor* dem Konstruktor der abgeleiteten Klasse aufgerufen. Ein Objekt der abgeleiteten Klasse befindet sich aber erst nach dem Konstruktoraufruf in einsatzfähigem Zustand (Bloch 2018, S. 95f).

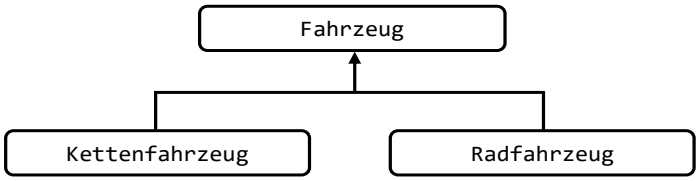
Das Verdecken von Methoden hat zur Folge, dass ein Objekt je nach dem deklarierten Typ der zur Ansprache verwendeten Referenzvariablen auf denselben Methodenaufruf unterschiedlich reagiert. Diese komplexe und potentiell verwirrende Konstellation sollte nach Möglichkeit durch einen eigenständigen Namen für die Unterklassenmethode vermieden werden (Mössenböck 2019, S. 97).

In C# wurde für das Ersetzen von Basisklassenmethoden die relativ komplexe Kombination von Überschreibung und Verdeckung entwickelt, um das Problem der sogenannten *fragilen Basisklassen* zu beheben (siehe z. B. Mössenböck 2019, S. 96f).[1] In der Programmiersprache Java gibt es beim Ersetzen von Instanzmethoden in abgeleiteten Klassen nur die Überschreibung. Das sorgt für angenehm einfache Verhältnisse, aber leider auch für Gefahren, weil eine Änderung der Basisklasse schädliche Effekte auf abgeleitete Klassen haben kann (siehe z. B. Baltes-Götz & Götz 2020, Abschnitt 7.10). Für Java wird die Vererbung mittlerweile von vielen ernst zu nehmenden Autoren als kritische Technik eingeschätzt, die besondere Sorgfalt erfordert (Bloch 2018, S. 87ff).

Nach den vielen Warnungen sind zwei Hinweise angemessen, damit die Anwendung der Vererbung nicht am Ende als entmutigend komplex erlebt wird:

[1] https://stackoverflow.com/questions/2921397/what-is-the-fragile-base-class-problem

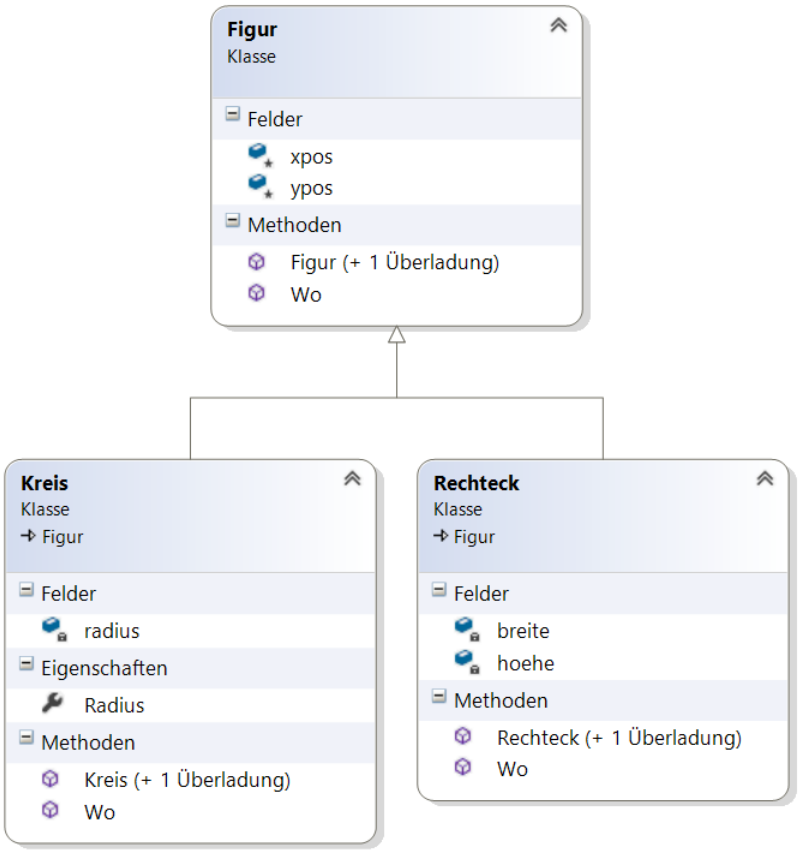
- Wenn man die Basisklasse und die abgeleitete Klasse selbst erstellt, also den Quellcode kennt, dann bewegt man sich auch bei Anwendung der Vererbung im normalen Risikobereich von Programmierfehlern.
- Bei einer konkreten Aufgabenstellung ist die Lage viel übersichtlicher als in den allgemeinen Erörterungen des aktuellen Kapitels. Wenn etwa Fahrzeuge mit Rad- und Kettenantrieb, die sich auf einem gemeinsamen Gelände bewegen, durch ein Programm zu modellieren und zu steuern sind, dann bietet sich die folgende Vererbungshierarchie an:



Bei der für alle Fahrzeugklassen benötigten Methode `ZielAnfahren()` ist eine polymorphe Lösung sinnvoll. Die Vererbungstechnik spart Entwicklungszeit, und es ist keine Debatte über fragile Basisklassen oder verdeckende Methoden erforderlich.

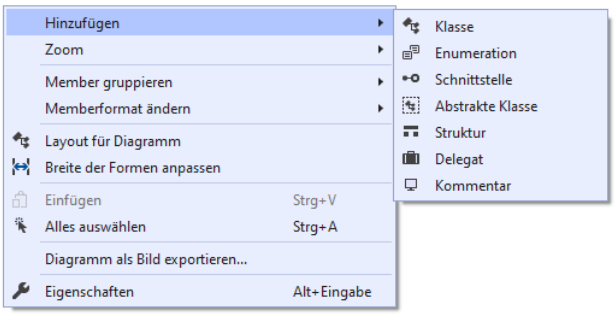
7.10 Klassendiagramme mit Vererbungsbeziehung

Unsere Entwicklungsumgebung Visual Studio Community 2019 kann mit wenigen Mausaktionen zur Erstellung des folgenden Klassendiagramms für das Figurenbeispiel veranlasst werden, wenn die im Abschnitt 3.3.2 beschriebene Erweiterung der Installation um den **Klassen-Designer** durchgeführt worden ist, wobei die Vererbungs- bzw. Spezialisierungsbeziehungen zwischen den Klassen nach den Regeln der UML-Notation (*Unified Modeling Language*) dargestellt werden:



Um das Klassendiagramm zu erstellen, markiert man im Projektmappen-Explorer die beteiligten Klassen und wählt aus dem Kontextmenü der Markierung das Item **Klassendiagramm anzeigen**.

Das Kontextmenü zum Fenster mit dem Klassendiagramm erlaubt einige Modifikationen und den Export des Diagramms in eine Datei mit wählbarem Format (z. B. EMF):



7.11 Abstrakte Methoden und Klassen

Um die eben beschriebene gemeinsame Verwaltung von Objekten aus diversen Unterklassen über Referenzvariablen vom Basisklassentyp realisieren und dabei Polymorphie nutzen zu können, dann müssen die beteiligten Methoden in der Basisklasse vorhanden sein. Wenn es in der Basisklasse zu einer Methode keine sinnvolle Implementierung gibt, erstellt man dort eine **abstrakte** Methode:

- Man beschränkt sich auf den Methodenkopf und setzt dort den Modifikator **abstract**.
- Den Methodenrumpf ersetzt man durch ein Semikolon.

Im Figurenbeispiel ergänzen wir eine Methode namens `Skaliere()`, mit der eine Figur zu artspezifischem Wachsen (oder Schrumpfen) um den per Parameter festgelegten Faktor aufgefordert werden kann. Ein Kreis wird auf diese Botschaft hin seinen Radius verändern, während ein Rechteck Breite und Höhe anzupassen hat. Weil die Methode in der Basisklasse `Figur` nicht sinnvoll realisierbar ist, wird sie dort abstrakt definiert:

```
public abstract class Figur {
   . . .
   public abstract void Skaliere(double faktor);
   . . .
}
```

Enthält eine Klasse mindestens *eine* abstrakte Methode, dann handelt es sich um eine **abstrakte Klasse**, und im Klassendefinitionskopf muss der Modifikator **abstract** angegeben werden.[1]

Abstrakte Methoden sind grundsätzlich virtuell (vgl. Abschnitt 7.7), wobei das Schlüsselwort **virtual** überflüssig und verboten ist.

Von einer abstrakten Klasse lassen sich keine Objekte erzeugen, aber man kann andere Klassen daraus ableiten. Implementiert eine abgeleitete Klasse die abstrakten Methoden, lassen sich Objekte von dieser Klasse herstellen; anderenfalls ist sie ebenfalls abstrakt.[2]

Wir leiten aus der nunmehr abstrakten Klasse `Figur` die konkreten Klassen `Kreis` und `Rechteck` ab, welche die abstrakte `Figur`-Methode `Skaliere()` implementieren:

---

[1] Während der (vom Compiler spendierte) Standardkonstruktor einer regulären Klasse die (nicht änderbare) Sichtbarkeit **public** besitzt, hat der Standardkonstruktor einer abstrakten Klasse die (nicht änderbare) Sichtbarkeit **protected**.

[2] Wenn im Beispiel `Figur`-Objekte erforderlich wären, dann könnte man in der Klasse `Figur` eine `Skaliere()` -Methode mit einem vorhandenen, aber leeren Rumpf definieren und auf die **abstract**-Modifikatoren verzichten. Das passive, vom Parameterwert unbeeindruckte Verhalten der `Figur`-Objekte nach der Botschaft `Skaliere()` wäre kein allzu großes Problem, aber auch kein ideales Design.

```
public class Kreis : Figur {
   double radius = 75.0;
   . . .
   public override void Skaliere(double faktor) {
      if (faktor >= 0.0)
         radius *= faktor;
   }
   . . .
}

public class Rechteck : Figur {
   double breite = 50.0, hoehe = 50.0;
   . . .
   public override void Skaliere(double faktor) {
      if (faktor >= 0.0) {
         breite *= faktor;
         hoehe *= faktor;
      }
   }
   . . .
}
```

Von den beiden Varianten der Ersetzung einer Basisklassenmethode durch eine signaturgleiche Unterklassenmethode (Verdecken und Überschreiben, vgl. Abschnitt 7.7) kommt bei einer abstrakten Basisklassenmethode nur das Überschreiben in Frage, wobei das zugehörige Schlüsselwort **override** anzugeben ist.

Neben den Methoden können auch Eigenschaften, Indexer und Ereignisse abstrakt definiert werden. Im Figurenbeispiel soll mit der Eigenschaft `Inhalt` die Möglichkeit geschaffen werden, den Flächeninhalt eines Objekts zu erfragen. Weil eine polymorphe Nutzung gewünscht ist, muss die Eigenschaft schon in der Basisklasse vorhanden sein. Dort ist aber keine sinnvolle Flächenberechnung möglich, sodass die Eigenschaft abstrakt definiert wird:

```
public abstract double Inhalt {
   get;
}
```

Im Definitionskopf ist der Modifikator **abstract** anzugeben, und bei der **get**-Methode ersetzt ein Semikolon die Implementation. Analog könnte auch eine **set**-Methode abstrakt definiert werden.

In den abgeleiteten Klassen `Kreis` und `Rechteck` wird die Eigenschaft `Inhalt` individuell realisiert:

```
public class Kreis : Figur {
   double radius = 75.0;
   . . .
   public override double Inhalt {
      get {return Math.PI * radius * radius;}
   }
   . . .
}

public class Rechteck : Figur {
   double breite = 50.0, hoehe = 50.0;
   . . .
   public override double Inhalt {
      get {return breite * hoehe;}
   }
   . . .
}
```

Obwohl sich aus einer abstrakten Klasse keine Objekte erzeugen lassen, kann sie doch als Datentyp verwendet werden. Referenzen dieses Typs sind ja auch unverzichtbar, wenn Objekte diverser Unterklassen gemeinsam verwaltet werden sollen, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`   Figur[] fa = new Figur[2];`
`   fa[0] = new Kreis(50.0, 50.0, 5.0);`
`   fa[1] = new Rechteck(10.0, 10.0, 5.0, 5.0);`
`   fa[0].Skaliere(2.0);`
`   fa[1].Skaliere(2.0);`
`   double ges = 0.0;`
`   for (int i = 0;  i < fa.Length; i++) {`
`    Console.WriteLine("Fläche Figur {0}: {1,10:f2}\n",`
`                      i, fa[i].Inhalt);`
`    ges += fa[i].Inhalt;`
`   }`
`   Console.WriteLine("Gesamtfläche:   {0,10:f2}",ges);`
`  }`
`}`
	`Fläche Figur 0:     314,16`
	
	`Fläche Figur 1:     100,00`
	
	`Gesamtfläche:       414,16`

Mit Hilfe der im Kapitel 9 vorzustellenden *Schnittstellen* werden wir noch mehr Flexibilität gewinnen und polymorphe Methodenaufrufe auch für Typen *ohne* gemeinsame Basisklasse realisieren.

7.12 Das Liskovsche Substitutionsprinzip

In diesem Abschnitt geht es um eine auf den ersten Blick theoretisch wirkende, aber durchaus praxisrelevante Klärung zur objektorientierten Vererbungsbeziehung. Das nach Barbara Liskov benannte Substitutionsprinzip verlangt von einer Klassenhierarchie (Liskov & Wing 1999, S. 1):

> Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

Wird beim Entwurf einer Klassenhierarchie das Liskovsche Substitutionsprinzip (LSP) beachtet, dann können Objekte einer abgeleiteten Klasse stets die Rolle von Basisklassenobjekten übernehmen, d.h. u. a.:

- Das „vertraglich“ zugesicherte Verhalten der Basisklassenmethoden wird auch von den verdeckenden oder überschreibenden Unterklassenvarianten eingehalten.
- Unterklassenobjekte werden bei Verwendung in der Rolle von Basisklassenobjekten nicht beschädigt.

Eine Verletzung der Ersetzbarkeitsregel kann auch bei einfachen Beispielen auftreten, wobei oft eine aus dem Anwendungsbereich stammende Plausibilität zum fehlerhaften Design verleitet. So ist z. B. ein Quadrat aus mathematischer Sicht ein spezielles Rechteck. Definiert man in einer Klasse für Rechtecke die Methoden `SkaliereX()` und `SkaliereY()` zur Änderung der Länge in X- bzw. Y-Richtung, so gehört zum „vertraglich“ zugesicherten Verhalten dieser Methoden:

- Bei einem Zuwachs in X-Richtung bleibt die Y-Ausdehnung unverändert.
- Verdoppelt man die Breite eines Objekts, verdoppelt sich auch der Flächeninhalt.

Die simple Tatsache, dass aus mathematischer Sicht jedes Quadrat ein Rechteck ist, rät offenbar dazu, eine Klasse für Quadrate aus der Klasse für Rechtecke abzuleiten. In der neuen Klasse ist allerdings die Konsistenzbedingung zu ergänzen, dass bei einem Quadrat stets alle Seiten gleich lang bleiben. Um das Auftreten irregulärer Objekte der Klasse `Quadrat` zu verhindern, wird man z. B. die Methode `SkaliereX()` so überschreiben, dass bei einer X-Modifikation automatisch auch die

Y-Ausdehnung angepasst wird. Damit ist aber der `SkaliereX()` - Vertrag verletzt, wenn ein Quadrat die Rechteckrolle übernimmt. Eine verdoppelte X-Länge führt etwa nicht zur doppelten, sondern zur vierfachen Fläche. Verzichtet man andererseits in der Klasse `Quadrat` auf das Überschreiben der Methode `SkaliereX()`, ist bei den Objekten dieser Klasse die Konsistenzbedingung identischer Seitenlängen massiv gefährdet. Offenbar haben Plausibilitätsüberlegungen zu einer schlecht entworfenen Klassenhierarchie geführt.

Eine exakte Verhaltensanalyse zeigt, dass ein Quadrat in funktionaler Hinsicht eben doch kein Rechteck ist. Es fehlt die für Rechtecke typische Option, die Ausdehnung in X- bzw. Y-Richtung separat zu verändern. Diese Option könnte in einem Algorithmus, der den Datentyp `Rechteck` voraussetzt, von Bedeutung sein. Es muss damit gerechnet werden, dass der Algorithmus irgendwann (bei einer Erweiterung der Software) auf Objekte mit einem von `Rechteck` abstammenden Datentyp trifft. Passiert dies mit der Klasse `Quadrat` könnte es zu Problemen kommen, weil nach einer Verdopplung der X-Ausdehnung der Flächeninhalt entgegen der Erwartung nicht auf das Doppelte, sondern auf das Vierfache wächst.

Um die Einhaltung des Substitutionsprinzips beurteilen zu können, bedarf es einer sorgfältigen Analyse. Wenn etwa Objekte der Klasse `Rechteck` *unveränderlich* wären, wenn also die Methoden `SkaliereX()` und `SkaliereY()` in der Klassendefinition von `Rechteck` fehlen würden, dann könnte die Klasse `Quadrat` sehr wohl als Spezialisierung von `Rechteck` definiert werden.

C# bietet gute Voraussetzungen für eine erfolgreiche objektorientierte Programmierung, kann aber z. B. eine Verletzung des Substitutionsprinzips nicht verhindern.

7.13 Erweiterungsmethoden

Soll eine Klasse um zusätzliche Handlungskompetenzen erweitert werden, erstellt man üblicherweise eine abgeleitete Klasse. Seit der C# - Version 3.0 steht eine alternative Erweiterungstechnik zur Verfügung für Situationen, in denen das Ableiten eines neuen Typs nicht möglich ist (z. B. bei Strukturen oder bei versiegelten Klassen).

Ihre vermutlich wichtigste Anwendung finden Erweiterungsmethoden bei den Standardabfragen der LINQ-Technik (*Language Integrated Query*), doch können Erweiterungsmethoden auch unabhängig von der LINQ-Technik erfolgreich eingesetzt werden.

7.13.1 Technische Realisation

Um die Instanzen eines vorhandenen Typs mit zusätzlichen Handlungskompetenzen auszustatten, definiert man eine statische Klasse und darin statische Methoden, die einen ersten, über das Schlüsselwort **this** besonders gekennzeichneten Parameter vom zu erweiternden Typ besitzen. Bei den Erweiterungsmethoden ist also durchaus eine neue Klasse erforderlich, doch können den Instanzen des zu erweiternden Typs die neuen Botschaften (Methoden) syntaktisch genauso zugestellt werden wie die typeigenen.

Der Compiler erstellt jedoch statische Methodenaufrufe, und der Zugriffsschutz wird nicht verletzt, weil eine Erweiterungsmethode keinen direkten Zugriff auf private Member (z. B. Methoden, Instanzvariablen) des erweiterten Typs hat.

Durch die im folgenden Beispiel definierte Erweiterungsmethode `Empty()` für die Klasse **String** wird festgestellt, ob ein **String**-Objekt vorhanden ist, aber keine Zeichen enthält:

Quellcodesegment	Ausgabe

`using System;`
 	
`public static class StringExt {`
`  public static bool Empty(this String arg) {`
`  return arg != null && arg.Length == 0;`
`  }`
`}`
 	
`class Prog {`
`  static void Main() {`
`  Console.WriteLine("".Empty());`
`  Console.WriteLine("m".Empty());`
`  }`
`}`
	`True`
	`False`

Tritt eine Erweiterungsmethode in Konkurrenz mit einer typeigenen, dann gewinnt letztere. Folglich besteht ein erhebliches Risiko beim Einsatz von Erweiterungsmethoden. Wenn z. B. in einer späteren Version der Klasse **String** die Instanzmethode **Empty()** mit identischer Signatur hinzukommt, wird diese bei einem Aufruf gegenüber der Erweiterungsmethode `Empty()` bevorzugt, und ein für die Benutzung der Erweiterungsmethode konzipiertes Programm zeigt vermutlich nicht mehr das intendierte Verhalten.

Im folgenden Beispiel wird für Instanzen der Struktur **Double** das Potenzieren durch die Erweiterungsmethode `H()` syntaktisch vereinfacht:

Quellcodesegment	Ausgabe

`using System;`
 	
`public static class Mathe {`
`  public static double H(this double arg, double expo) {`
`  return Math.Pow(arg, expo);`
`  }`
`}`
 	
`class Prog {`
`  static void Main() {`
`  double pi = 3.14;`
`  Console.WriteLine(pi.H(2));`
`  }`
`}`
	`9,8596`

7.13.2 Anwendungsempfehlungen

Um die Funktionalität einer Klasse zu erweitern, sollte nach einer Empfehlung von Microsoft nach Möglichkeit eine abgeleitete Klasse definiert werden.[1] Erweiterungsmethoden sollten nur dann in Erwägung gezogen werden, wenn die Definition eines abgeleiteten Typs ausgeschlossen ist (bei einer versiegelten Klasse oder bei einer Struktur). Dabei muss das Risiko in Kauf genommen werden, dass eine neue Version des erweiterten Typs eine Instanzmethode erhält, welche die Erweiterungsmethode aus dem Spiel nimmt.

[1] https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/classes-and-structs/extension-methods

7.14 Übungsaufgaben zum Kapitel 7

1) Warum kann der folgende Quellcode nicht übersetzt werden?

```
using System;
class Basisklasse {
   int ibas = 3;
   public Basisklasse(int i) { ibas = i; }
   public virtual void Hallo() {
      Console.WriteLine("Hallo-Methode der Basisklasse");
   }
}
class Abgeleitet : Basisklasse {
   public override void Hallo() {
      Console.WriteLine("Hallo-Methode der abgeleiteten Klasse");
   }
}

class Prog {
   static void Main() {
      Abgeleitet s = new Abgeleitet();
      s.Hallo();
   }
}
```

2) Im folgenden Beispiel wird die Klasse `Kreis` aus der Klasse `Figur` abgeleitet:

```
public class Figur {
   double xpos = 100.0, ypos = 100.0;
   public Figur(double x, double y) {
      if (x >= 0.0 && y >= 0.0) {
         xpos = x;
         ypos = y;
      }
   }
   public Figur() { }
}

public class Kreis : Figur {
    double radius = 75.0;
    public Kreis(double x, double y, double rad) {
        if (x >= 0.0 && y >= 0.0 && rad >= 0.0) {
                xpos = x;
                ypos = y;
                radius = rad;
        }
    }
    public Kreis() { }
}
```

Trotzdem erlaubt der Compiler im initialisierenden `Kreis`-Konstruktor den `Kreis`-Objekten keinen Zugriff auf ihre geerbten Instanzvariablen `xpos` und `ypos`. Wie ist das Problem zu erklären und zu lösen?

3) Erläutern Sie die folgenden Begriffe:

- Überladen von Methoden
- Verdecken von Methoden
- Überschreiben von Methoden

Welche von den drei genannten Programmiertechniken ist bei statischen Methoden *nicht* anwendbar?

4) Überschreiben Sie in der Klasse `Bruch` die von der Basisklasse **Object** geerbte Methode **ToString()**, die in der Klasse **Object** den folgenden Definitionskopf hat:

public virtual string ToString()

Die verbesserte **ToString()** - Rückgabe kann z. B. so aussehen:

Quellcodes-Segment	Ausgabe ohne **ToString()** - Überschreibung	Ausgabe mit **ToString()** - Überschreibung

`using System;`
`class Bruchrechnung {`
`  static void Main() {`
`  var b = new Bruch(7, 8, "");`
`  Console.WriteLine(b);`
`  }`
`}`
	`Bruch`	`7/8`

8 Typgenerisches Programmieren

In C# haben die Felder von Klassen und Strukturen sowie die Parameter von Methoden einen festen Datentyp, sodass der Compiler für Typsicherheit sorgen, d.h. die Zuweisung ungeeigneter Werte bzw. Objekte verhindern kann. Oft werden aber für unterschiedliche Datentypen völlig analog arbeitende Klassen, Strukturen oder Methoden benötigt, z. B. eine Klasse zur Verwaltung einer geordneten Liste mit Elementen eines bestimmten Typs. Statt die Definition für jeden in Frage kommenden Elementdatentyp zu wiederholen, kann man die Definition *typgenerisch* formulieren. Bei der *Verwendung* einer generischen Listenklasse ist der zu verarbeitende Elementtyp konkret festzulegen, und es entsteht eine sogenannte *geschlossene konstruierte Klasse*.[1] Im Ergebnis erhält man durch *eine* Definition zahlreiche konkrete Klassen, wobei die Typsicherheit nicht abgeschwächt wird.

Durch generisches Programmieren werden also folgende Ziele erreicht:

- Wiederverwendung von Code
  Code mit generischem Design kann für unterschiedliche Typkonkretisierungen verwendet werden.
- Typsicherheit
  Im Vergleich zur veralteten Verwendung eines allgemeinen Referenzdatentyps (wie z. B. **Object**) für die Elemente in einer Liste oder einer anderen Kollektion spart man sich lästige und fehleranfällige Typanpassungen.
- Performanz
  Bei der Verwendung eines allgemeinen Referenzdatentyps (wie z. B. **Object**) für Kollektionselemente werden aufwändige (Un)boxing - Operationen in großer Zahl fällig, wenn Elemente mit Werttyp zu verwalten sind. Im Vergleich zu dieser veralteten Technik bringt die generische Programmierung eine erhebliche Verbesserung der Performanz.

Ein besonders erfolgreiches Anwendungsfeld für Typgenerizität sind die Klassen zur Verwaltung von Listen, Mengen, (Schlüssel-Wert) - Tabellen und sonstigen Kollektionen. Für die Objekte dieser Klassen wird im Manuskript alternativ zur offiziellen Bezeichnung *Kollektion* aus sprachlichen Gründen gelegentlich auch die Bezeichnung *Container* verwendet. Dass an anderer Stelle auch von Containern (zur Verwaltung von GUI-Bedienelementen) die Rede ist, sollte keine Verwirrung stiften.

Typgenerische Definitionen eignen sich nicht nur für Klassen, Strukturen und Methoden, sondern auch für die später zu behandelnden Schnittstellen, Delegaten und Ereignisse.

8.1 Motive für generische Typen

Im Abschnitt 6.2.12 haben wir die Klasse **ArrayList** aus dem Namensraum **System.Collections** als Container für Objekte beliebigen Typs verwendet:[2]

```
ArrayList al = new();
al.Add("Text");
al.Add(3.14);
al.Add(13);
```

Im Unterschied zu einem Array (siehe Abschnitt 6.2) bietet die Klasse **ArrayList**:

---

[1] https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/generics/generic-classes

[2] Wer sich noch nicht an die seit C# 9.0 möglichen **new**-Ausdrücke ohne Konstruktor gewöhnt hat, findet im Abschnitt 5.4.3.2 eine Erläuterung zu zieltypisierten **new**-Ausdrücken.

- eine automatische Größenanpassung
- Typflexibilität (durch die Verwendung des maximal allgemeinen Elementtyps **Object**)

Während das obige Beispiel die Größen- *und* die Typflexibilität nutzt, ist oft ein Container mit automatischer Größenanpassung (ein dynamischer Array) für Objekte eines bestimmten, *festen* Typs gefragt (z. B. zur Verwaltung von **String**-Objekten). Bei dieser Einsatzart stören die folgenden Nachteile der Typbeliebigkeit:

- Fehlende Typsicherheit
  Wenn beliebige Objekte zugelassen sind, die intern über Referenzvariablen vom Typ **Object** verwaltet werden, dann kann der Compiler *nicht* sicherstellen, dass ausschließlich Objekte des gewünschten Typs in den Container eingefüllt werden. Viele Programmierfehler werden erst zur Laufzeit (womöglich vom Kunden) entdeckt.
- Notwendigkeit von expliziten Typumwandlungen
  Entnommene Objekte können erst nach einer expliziten Typumwandlung die Methoden ihres Typs ausführen. Die häufig benötigten Typanpassungen sind lästig und fehleranfällig.
- Leistungsschädliche (Un)boxing-Operationen
  Wenn Variablen mit Werttyp zu verwalten sind, resultieren leistungsschädliche (Un)boxing-Operationen, weil intern ein Referenzdatentyp (wie z. B. **Object**) verwendet wird.

Im folgenden Beispielprogramm sollen **String**-Objekte in einem **ArrayList**-Container verwaltet werden:

```
using System;
using System.Collections;
class Prog {
 static void Main() {
  ArrayList al = new();
  al.Add("Otto");
  al.Add("Rempremerding");
  al.Add('.');
  for(int i = 0; i < al.Count; i++)
  Console.WriteLine($"Länge von Element {i}: {((String)al[i]).Length}");
 }
}
```

Bevor ein **String**-Element des Containers nach seiner Länge gefragt werden kann, ist eine lästige Typanpassung fällig, weil der Compiler nur den deklarierten Typ **Object** kennt:

```
((String)al[i]).Length
```

Beim dritten **Add()** - Aufruf wird eine Instanz vom Typ **char** per Autoboxing in den Container befördert. Weil der Container eigentlich zur Aufbewahrung von **String**-Objekten gedacht war, liegt hier ein Programmierfehler vor, den der Compiler wegen der fehlenden Typsicherheit nicht bemerken kann. Beim Versuch, die **char**-Instanz als **String**-Objekt zu behandeln, scheitert das Programm an einem Ausnahmefehler vom Typ **InvalidCastException**:

```
Unbehandelte Ausnahme: System.InvalidCastException: Das Objekt des Typs "System.Char"
kann nicht in Typ "System.String" umgewandelt werden.
   bei Prog.Main() in Prog.cs:Zeile 10.
```

Es ist nicht schwer, eine spezialisierte Container-Klasse zur Verwaltung von **String**-Objekten zu definieren, um die beiden Probleme (syntaktische Umständlichkeit, mangelnde Typsicherheit) zu vermeiden. Vermutlich werden analog funktionierende Behälter aber auch für alternative Elementtypen benötigt, und entsprechend viele Klassen zu definieren, die sich nur durch den Inhaltstyp unterscheiden, wäre nicht rationell. Für eine solche Aufgabenstellung bietet C# seit der Version 2.0 die generischen Typen. Durch die Verwendung von Typformalparametern bei der Definition wird die gesamte Handlungskompetenz typunabhängig formuliert. Bei jedem Instanziieren (also beim Erstellen einer Container-Instanz) ist jedoch ein konkreter Elementtyp anzugeben. Weil der Compi-

ler den konkreten Typ kennt, kann er beim Befüllen des Containers für Typsicherheit sorgen, und bei der Verwendung von entnommenen Elementen sind keine lästigen Typumwandlungen erforderlich.

Mit der generischen Klasse **List<T>** im Namensraum **System.Collections.Generic** enthält die BCL eine perfekte **ArrayList**-Alternative zur Verwaltung einer dynamischen Liste von Elementen mit identischem Typ. Das obige Beispielprogramm ist schnell auf die neue Technik umgestellt:

```
using System;
using System.Collections.Generic;
class Prog {
   static void Main() {
      List<String> gl = new();
      gl.Add("Otto");
      gl.Add("Rempremerding");
      gl.Add(".");
      for (int i = 0; i < gl.Count; i++)
         Console.WriteLine($"Länge von Element {i}: {gl[i].Length}");
   }
}
```

Bei der Deklaration einer **List<T>** - Referenzvariablen ist an Stelle des Typparameters **T** ein konkreter Datentyp anzugeben, z. B.:

```
List<String> gl = new();
```

Jeder Versuch, ein Element mit abweichendem Typ in den **List<String>** - Container einzufügen, wird vom Compiler verhindert, z. B.:

```
gl.Add('.');
```

Die Elemente des auf **String**-Objekte spezialisierten Containers beherrschen ohne Typanpassung die Methoden ihrer Klasse, z. B.:

```
for (int i = 0; i < gl.Count; i++)
   Console.WriteLine($"Länge von Element {i}: {gl[i].Length}");
```

Bei einem dynamischen Container für Elemente mit einem festen *Werttyp* erspart die Klasse **List<T>** im Vergleich zu **ArrayList** zudem die zeitaufwändigen (Un)boxing-Operationen. Mit diesem Thema sollen Sie sich im Rahmen einer Übungsaufgabe beschäftigen.

Wenn ausnahmsweise für die Elemente in einer Liste *unterschiedliche* (nicht in Vererbungsrelation stehende) Datentypen erlaubt sein sollen, dann kann die Klasse **ArrayList** noch als akzeptable Lösung betrachtet werden. Allerdings erlaubt die generische Klasse **List<T>** auch für solche Fälle eine attraktivere Lösung, indem der Typformalparameter durch die Klasse **Object** konkretisiert wird. Für die Klasse **List<Object>** ist im Vergleich zu **ArrayList** mit einer besseren Kompatibilität zu rechnen, weil moderne Software-Entwicklung bevorzugt generische Datentypen verwendet (z. B. wegen der Typsicherheit).

8.2 Generische Klassen

Aus der Entwicklerperspektive besteht der wesentliche Vorteil einer generischen Klasse darin, dass mit *einer* Definition beliebig viele konkrete Klassen für spezielle Datentypen geschaffen werden. Dieses Konstruktionsprinzip ist speziell bei den Kollektionsklassen sehr verbreitet (siehe BCL-Namensraum **System.Collections.Generic**), aber keinesfalls auf Container mit ihrer weitgehend inhaltstyp*un*abhängigen Verwaltungslogik beschränkt.

Analog zu den generischen Klassen bietet C# auch generische Strukturen, Schnittstellen, Delegaten und Ereignisse. Wir befassen uns im Kapitel 8 hauptsächlich mit generischen Klassen und Strukturen, machen aber auch schon erste Erfahrungen mit generischen Schnittstellen, die wir im Kapitel 9 vertiefen werden.

8.2.1 Definition

Bei einer generischen Klassendefinition verwendet man **Typformalparameter**, die im Definitionskopf hinter dem Klassennamen zwischen spitzen Klammern und durch Kommata getrennt angegeben werden. Wir erstellen als Beispiel eine generische Klasse namens `SimpleStack<T>`, die einen LIFO-Stapel (*last-in-first-out*) verwaltet und einen Typformalparameter für den beliebig wählbaren Elementtyp verwendet. In der Praxis wird man für eine solche Standardaufgabe allerdings die Klasse **Stack<T>** aus dem BCL-Namensraum **System.Collections.Generic** verwenden.

```
public class SimpleStack<T> {
   int capacity = 5;
   T[] data;
   int size;

   public SimpleStack() {
      data = new T[capacity];
   }

   public SimpleStack(int max) {
      if (max > 0)
         capacity = max;
      data = new T[capacity];
   }

   public int Count {
      get { return size; }
   }

   public void Push(T element) {
      if (size < capacity)
         data[size++] = element;
      else
         throw new System.InvalidOperationException("Kapazität erschöpft");
   }

   public T Pop() {
      if (size > 0)
         return data[--size];
      else
         throw new System.InvalidOperationException("Stapel leer");
   }
}
```

Mit der Methode `Push()` legt man ein neues Element auf den Stapel, sofern das noch möglich ist. Wenn beim Aufruf die Kapazität des Stapels erschöpft ist, dann wirft die Methode `Push()` dem Aufrufer per **throw**-Anweisung ein Ausnahmeobjekt aus der Klasse **InvalidOperationException** zu, um ihn über das Problem zu informieren. Nachdem Sie im Vorgriff auf das noch ausstehende Kapitel 13 über die Ausnahmebehandlung schon mehrfach die Rolle des potentiellen *Empfängers* von Ausnahmeobjekten beobachten konnten, erleben Sie nun einen Ausblick auf die Rolle des *Senders* von Ausnahmeobjekten.

Mit der Methode `Pop()` wird das oberste Element vom Stapel abgehoben und dem Aufrufer übergeben. Ist der Stapel leer, wird der Aufrufer durch ein Ausnahmeobjekt aus der Klasse **InvalidOperationException** informiert.

Innerhalb der Klassendefinition wird der Typformalparameter wie ein Datentyp verwendet, z. B.:

- als Elementtyp für den internen Array mit den Daten des Stapels
- als Datentyp für den Formalparameter der Methode `Push()`
- als Datentyp für die Rückgabe der Methode `Pop()`

In einer Konstruktordefinition ist der Klassenname *ohne* Typformalparameter zu schreiben, z. B.:

```
public SimpleStack() {
   data = new T[capacity];
}
```

Vielleicht vermissen Sie beim `SimpleStack<T>` die Größendynamik der professionellen Kollektionsklassen (z. B. **Stack<T>**). Um das automatische Wachsen zu realisieren, könnte man den intern zur Datenspeicherung benutzten Array bei Bedarf durch ein größeres Exemplar (z. B. mit doppelter Länge) ersetzen und die bisherigen Elemente kopieren (siehe Beschreibung der **Array**-Methode **Resize()** im Abschnitt 6.2.2). Im Beispiel wird der Einfachheit halber auf die Größendynamik verzichtet.

Im folgenden Programm wird aus der generischen Klasse `SimpleStack<T>` eine konkrete Klasse (man sagt auch: *eine geschlossene konstruierte Klasse*) zur Verwaltung eines **double**-Stapels erzeugt:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  var ds = new SimpleStack<double>(10);`
`  ds.Push(3.141);`
`  ds.Push(2.718);`
`  while (ds.Count > 0)`
`  Console.WriteLine("Oben lag: " + ds.Pop());`
`  }`
`}`
	`Oben lag: 2,718`
	`Oben lag: 3,141`

Im Abschnitt 11.5 werden Sie mit der BCL-Klasse **Dictionary<K, V>** ein Beispiel für eine generische Klasse mit *zwei* Typformalparametern kennenlernen.

Für einen einzelnen Typformalparameter sollte in der Regel der Großbuchstabe **T** verwendet werden. Wenn mehrere Typformalparameter auftreten, oder wenn ein einzelner Buchstabe nicht ausreicht, um die Bedeutung eines Typformalparameters zu klären, dann rät Microsoft zu einem aussagekräftigen, mit dem Buchstaben **T** eingeleiteten Namen, z. B.:[1]

Dictionary<TKey, TValue>

Der Vollständigkeit halber sei noch erwähnt, dass innerhalb eines Namensraums verschiedene generische Typen denselben Namen verwenden dürfen, solange ihre Typformalparameterlisten verschieden lang sind. Auch ein namensgleicher nichtgenerischer Typ ist erlaubt.

Im Namen der Datei mit dem Quellcode einer generischen Klasse tauchen keine Typformalparameter auf. Z. B. steckt die Klasse **Dictionary<TKey, TValue>** aus der BCL zu .NET 5.0 in einer Datei mit dem Namen **Dictionary.cs**.

[1] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/generic-type-parameters#type-parameter-naming-guidelines

8.2.2 Restringierte Typformalparameter

Häufig muss eine generische Definition bei den Typen, die einen Typparameter konkretisieren dürfen, gewisse Handlungskompetenzen voraussetzen. Muss z. B. ein generischer Container seine Elemente sortieren, dann verlangt man in der Regel von einem konkreten Elementtyp, dass er die *Schnittstelle* **IComparable<T>** erfüllt, d.h. eine Methode namens **CompareTo()** mit dem folgenden Definitionskopf besitzt (beschrieben unter Verwendung des Typparameters **T**):

public int CompareTo(T *instanz***)**

Im Abschnitt 6.3.1.2.2 haben Sie erfahren, dass die Klasse **String** eine solche Methode besitzt, und wie **CompareTo()** das Prüfergebnis über den Rückgabewert signalisiert:

		CompareTo() - Ergebnis
Das befragte Objekt ist im Vergleich zum Aktualparameter …	kleiner	-1
	gleich	0
	größer	1

Damit sollte klar genug sein, was die Schnittstelle (das Interface) **IComparable<T>** von einem implementierenden Typ verlangt. Mit dem generellen Thema *Schnittstellen* werden wir uns im Kapitel 9 ausführlich beschäftigen.

Wir erstellen nun eine generische Listenverwaltungsklasse, die neue Elemente automatisch an der richtigen Sortierposition einfügt und daher ihren Typformalparameter auf den Schnittstellentyp **IComparable<T>** einschränkt:[1]

```
public class SimpleSortedList<T> where T : System.IComparable<T> {
   int capacity = 5;
   T[] data;
   int size;

   public SimpleSortedList() {
      data = new T[capacity];
   }

   public SimpleSortedList(int len) {
      if (len > 0)
         capacity = len;
      data = new T[capacity];
   }

   public void Add(T element) {
      if (size == data.Length)
         throw new System.InvalidOperationException("Kapazität erschöpft");
      int i;
      for (i = size - 1; i >= 0 && element.CompareTo(data[i]) < 0; i--)
         data[i + 1] = data[i];
      data[i + 1] = element;
      size++;
   }

   public int Count {
      get => size;
   }
```

[1] Die Definition der Klasse wurde durch Paul Frischknecht entscheidend verbessert.

```
    public T this[int index] {
       get {
          if (index < 0 || index >= size)
             throw new System.ArgumentOutOfRangeException();
          return data[index];
       }
    }
 }
```

In der Methode **Add()** und im Indexer (vgl. Abschnitt 5.8) erlauben wir uns erneut einen Vorgriff auf das Kapitel über Ausnahmefehler und informieren den Aufrufer über eine erschöpfte Kapazität bzw. über einen ungültigen Index, indem wir eine **InvalidOperationException** bzw. eine **ArgumentOutOfRangeException** werfen.

Bei der Formulierung von Einschränkungen für einen Typparameter wird das Schlüsselwort **where** verwendet, wobei u. a. folgende Regeln gelten:[1]

- Man kann eine Basisklasse vorschreiben.
  Seit C# 7.3 sind als Basisklassen auch **System.Enum** (siehe Abschnitt 6.4) sowie **System.Delegate** und **System.MulticastDelegate** (siehe Kapitel 10) erlaubt, sodass man auch einen Aufzählungs- bzw. einen Delegatentyp vorschreiben kann.
- Mit dem Schlüsselwort **class** wird vereinbart, dass nur Referenztypen erlaubt sind.
- Mit dem Schlüsselwort **struct** wird vereinbart, dass nur Werttypen erlaubt sind (siehe Beispiel im Abschnitt 8.3).
- Man kann auch *mehrere* Restriktionen durch Kommata getrennt angeben, die von einem konkreten Typ allesamt zu erfüllen sind. Die Schlüsselwörter **class** und **struct** müssen ggf. am Anfang einer Liste von Restriktionen stehen.
- Während nur *eine* Basisklasse vorgeschrieben werden darf, sind beliebig viele Schnittstellen (vgl. Kapitel 9) erlaubt, die ein konkreter Typ *alle* erfüllen muss.
- Mit dem Listeneintrag **new()** wird für die konkreten Typen ein parameterfreier Konstruktor vorgeschrieben. In einer *Liste* von Restriktionen muss der Eintrag **new()** ggf. am Ende stehen.
- Sind mehrere Typformalparameter mit Restriktionen vorhanden, ist für jeden Parameter eine eigene **where**-Klausel anzugeben, und die **where**-Klauseln sind durch Kommata zu trennen.

Weitere Details zu den Optionen und Motiven für Typrestriktionen finden sich z. B. bei Griffiths (2013, S. 136ff) und Richter (2006, S. 394ff).

Im folgenden Programm wird aus der generischen Klasse `SimpleSortedList<T>` eine konkrete Klasse (man sagt auch: *eine geschlossene konstruierte Klasse*) zur Verwaltung einer sortierten **int**-Liste erzeugt:

Quellcode	Ausgabe

`class Prog {`
`  static void Main() {`
`  var si = new SimpleSortedList<int>(5);`
`  si.Add(11); si.Add(2); si.Add(1); si.Add(4);`
`  for (int i = 0; i < si.Count; i++)`
`  System.Console.WriteLine(si[i]);`
`  }`
`}`
	`1`
	`2`
	`4`
	`11`

[1] https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/generics/constraints-on-type-parameters

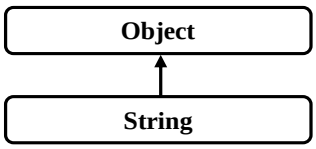
8.2.3 Generische Klassen und Vererbung

Bei der Definition einer generischen Klasse kann man als Basisklasse verwenden:

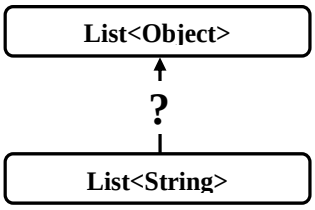
- eine nicht-generische Klasse, z. B.
```
class GenDerived<T> : BaseClass {
 . . .
}
```
- eine geschlossene konstruierte Klasse
```
class GenDerived<T> : GenBase<int, double> {
 . . .
}
```
- eine sogenannte *offene konstruierte Klasse*, die noch Typformalparameter aus der neu definierten generischen Klasse enthält, wobei Typrestriktionen der Basisklasse ggf. zu wiederholen sind, z. B.:
```
class GenBase<T1, T2> where T2 : IComparable<T2> {
 . . .
}
class GenDerived<T> : GenBase<int, T> where T : IComparable<T> {
 . . .
}
```
Wird im Beispiel aus der generischen Klasse `GenDerived<T>` die geschlossene konstruierte Klasse `GenDerived<int>` erstellt, dann ist `GenBase<int, int>` deren Basisklasse.

Offenbar ist die nachträgliche Integration der generischen Typen in das *Common Type System* (CTS) der .NET - Plattform gut gelungen.

Im Zusammenhang mit der Vererbung und der Generizität bleibt die wichtige Frage zu klären, ob sich bei zwei geschlossenen Konstruktionen aus einer generischen Klasse mit dem Typparameter **T** (z. B. **List<T>**) eine Spezialisierungsbeziehung zwischen den beiden Aktualparametertypen, z. B.:



auf die konstruierten Klassen überträgt, z. B.:



Obwohl eine positive Beantwortung der aufgeworfenen Frage auf den allerersten Blick plausibel erscheint, hätte sie doch üble Folgen. Man könnte nämlich z. B. ...

- die Adresse eines Objekts vom Typ **List<String>** einer Referenzvariablen vom Typ **List<Object>** zuweisen
- und mit Hilfe dieser Referenzvariablen über die dann verfügbare Methode

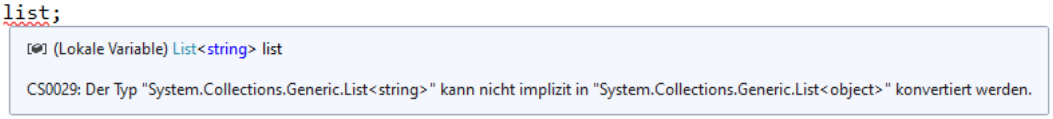
  public void Add(Object *element***)**

  eine Instanz beliebigen Typs in die Liste aufnehmen.

Damit wäre das essentielle Prinzip der Sortenreinheit verletzt, und bei nächster Gelegenheit würde der Versuch, dem eingeschmuggelten Objekt eine **String**-Kompetenz abzuverlangen, zu einem Laufzeitfehler führen.

Um das beschriebene Desaster zu vermeiden, wird in C# eine Spezialisierungsbeziehung zwischen zwei Aktualparametertypen **NICHT** auf die zugehörigen Konkretisierungen derselben generischen Klasse übertragen, und der folgende Versuch, das Bemühen des C# - Compilers um Typsicherheit auszutricksen, misslingt:

```
var list = new List<string> { "a", "b" };
List<object> lob = list;
```



Man sagt, dass in C# die Typformalparameter bei generischen Klassen **invariant** sind.

Wenn aus der Spezialisierungsbeziehung von Elementdatentypen eine entsprechende Spezialisierungsbeziehung für Container-Datentypen folgt, spricht man von einem **kovarianten** Verhalten, und dieser Fall ist in C# durchaus anzutreffen. Es ist sogar ein beim Einsatz von Arrays permanent drohender Programmierfehler, den der Compiler aus Kompatibilitätsgründen *nicht* verhindern kann. Die oben als tückisch entlarvte und bei generischen Klassen verbotene Kovarianz ist bei Arrays in C# (und auch in anderen Programmiersprachen wie Java) leider erlaubt. Der folgende Quellcode wird ohne Kritik übersetzt

```
object[] oarr = new string[] { "a", "b" };
oarr[0] = 13;
foreach (string s in oarr)
   Console.WriteLine(s.Length);
```

und verursacht zur Laufzeit einen Ausnahmefehler:

```
Unbehandelte Ausnahme: System.ArrayTypeMismatchException: Es wurde versucht,
auf ein Element zuzugreifen, dessen Typ mit dem Array nicht kompatibel ist.
```

Obwohl bisher wenig Positives über die Kovarianz gesagt wurde, kann sie unter bestimmten Voraussetzungen doch sinnvoll eingesetzt werden. Bei generischen Schnittstellen und Delegaten ist es möglich, einen Typformalparameter explizit als kovariant zu deklarieren (siehe Abschnitte 9.3.1 bzw. 10.1.7).

8.3 Nullable<T> als Beispiel für generische Strukturen

In diesem Abschnitt wird die generische Struktur **Nullable<T>** aus der BCL zu .NET 5.0 vorgestellt:[1]

```
public partial struct Nullable<T> where T : struct {
   private readonly bool hasValue;
   internal T value;

   public Nullable(T value) {
      this.value = value;
      this.hasValue = true;
   }

   public readonly bool HasValue {
      get => hasValue;
   }
```

[1] In Bezug auf das Thema des aktuellen Abschnitts bestehen keine wesentlichen Unterschiede zwischen den **Nullable<T>** - Strukturen in .NET 5.0 und .NET Framework 4.8. Der BCL-Quellcode ist hier zu finden:
.NET 5: https://github.com/dotnet/runtime, freundlicher präsentiert in: https://source.dot.net/
.NET Framework 4.8: https://referencesource.microsoft.com/

```
    public readonly T Value {
       get { ... }
    }
    . . .
}
```

Die Struktur ist als **partial** gekennzeichnet, weil ihre Implementation auf zwei Dateien verteilt ist, was für uns aktuell ohne Belang ist (vgl. Abschnitt 5.12.6).

In der **where**-Klausel wird dafür gesorgt, dass der Typparameter nur durch eine Struktur konkretisiert werden darf.

Eine Instanz einer Konkretisierung der generischen Struktur **Nullable<T>** kann einen Wert einer gewöhnlichen Struktur (z. B. einen Wert eines elementaren Datentyps) verpacken, aber auch den Ausnahmewert **null** annehmen. Eine Variable vom Typ einer **Nullable<T>** - Konkretisierung kann daher einen regulären Wert annehmen, oder einen undefinierten Zustand signalisieren. Verpackt man z. B. den Typ **bool**, dann kann die resultierende Variable neben den Werten **true** und **false** auch noch den dritten Wert **null** annehmen, der als *unbekannt* zu interpretieren ist.

Im folgenden Beispiel wird eine Variable vom Typ **Nullable<bool>** deklariert:

```
Nullable<bool> status;
```

Man kann einer **Nullable**-Instanz jeden Wert des Grundtyps und außerdem den Wert **null** zuweisen, z. B.:

```
status = null;
```

Die **null**-fähige Variante zu einem Strukturtyp lässt sich auch durch den Namen des Grundtyps und ein angehängtes Fragezeichen notieren, z. B.:

```
bool? status;
```

Eine **Nullable**-Instanz informiert in der booleschen get-only - Eigenschaft **HasValue** darüber, ob ein definierter Wert vorhanden ist, und hält diesen Wert ggf. in der Eigenschaft **Value** für den ausschließlich lesenden Zugriff bereit, z. B.:

```
var alter = new int?[2];
alter[0] = 30;
alter[1] = null;
foreach (int? ni in alter)
   if (ni.HasValue)
      Console.WriteLine(ni.Value);
   else
      Console.WriteLine("unbekannt");
```

Ist kein definierter Wert vorhanden, führt ein Leseversuch zu einem Ausnahmefehler vom Typ **InvalidOperationException**.

Während der Compiler den Grundtyp bei Bedarf implizit in den zugehörigen **Nullable**-Typ konvertiert, ist für den umgekehrten Übergang eine *explizite* Konvertierung unter der Verantwortung des Programmierers erforderlich, z. B.:

```
double? dn = 77.7;
double d = (double)dn;
```

Hat das **Nullable**-Argument des Typumwandlungsoperators den Wert **null**, kommt es zu einem Ausnahmefehler vom Typ **InvalidOperationException**.

Die beim Grundtyp unterstützten **Operatoren** sind auch bei der **null**-fähigen Verpackung erlaubt, z. B.:

```
double? d1 = 1.0, d2 = 2.0;
double? s = d1 + d2;
```

Hat ein beteiligter Operand den Wert **null**, so erhält auch der Ausdruck diesen Wert, z. B.:

```
double? d1 = 1.0, d2 = null;
double? s = d1 + d2;
Console.WriteLine(s.HasValue); // liefert false
```

Man kann einer gewöhnlichen (nicht **null**-fähigen) Strukturinstanz den Wert **null** nicht zuweisen. Ein Vergleich mit diesem Wert ist hingegen erlaubt, wobei das Ergebnis immer **false** ist, z. B. beim Vergleich:

```
0 == null
```

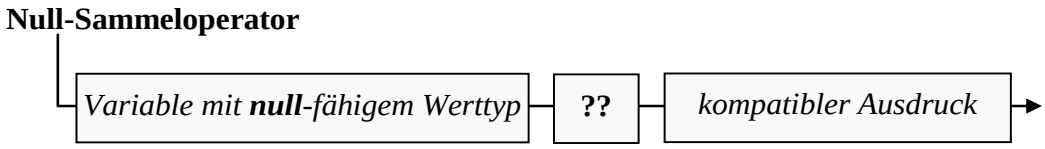
Ein (statisches) Feld mit **Nullable**-Typ wird mit **null** initialisiert, z. B.:

Quellcode	Ausgabe

`class Prog {`
`  int i;`
`  int? ni;`
`  static void Main() {`
`  var p = new Prog();`
`  System.Console.WriteLine(p.i+"\n"+p.ni.HasValue);`
`  }`
`}`
	`0`
	`False`

Von dieser Regel sind auch die Elemente eines Arrays betroffen, weil sie als Instanzvariablen aufgefasst werden können.

Mit dem sogenannten **Null-Sammeloperator**, der durch zwei Fragezeichen ausgedrückt wird, lässt sich die Zuweisung einer **null**-fähigen Strukturinstanz an eine Variable des Grundtyps samt Ersatzwert für die kritische **null**-Situation bequem formulieren:[1]



Ist der linke **??** - Operand von **null** verschieden, liefert er den Wert des Ausdrucks. Anderenfalls kommt der rechte Operand zum Zug, der vom Grundtyp und initialisiert sein muss. Beispiele zur Verwendung des **??** - Operators:

Quellcodesegment	Ausgabe

`int? ni = 4;`
`int i = ni ?? -1;`
`Console.WriteLine(i);`
`ni = null;`
`i = ni ?? -1;`
`Console.WriteLine(i);`
	`4`
	`-1`

Die Anweisung

```
int i = ni ?? -1;
```

ist äquivalent zu:

```
int i = (ni != null) ? (int)ni : -1;
```

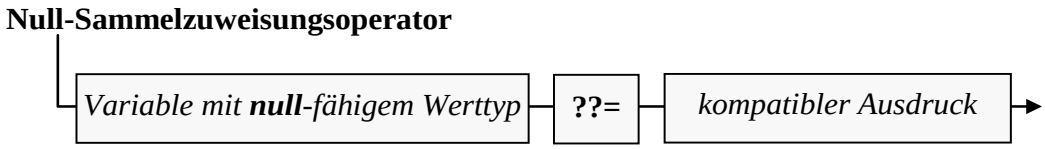
Der Null-Sammeloperator kann auch auf Referenztypen angewendet werden, z. B.:

---

[1] Die Bezeichnung *Null-Sammeloperator* wurde von der folgenden Webseite übernommen:
https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/builtin-types/nullable-value-types
Eine alternative Bezeichnung lautet: *Null-Koaleszenz - Operator*.

```
string s = null;
bool shortOrNull = (s ?? "").Length < 50;
```

Seit C# 8 wird mit dem sogenannten **Null-Sammelzuweisungsoperator** (engl.: *null-coalescing assignment operator*) das Zusammenwachsen (Bedeutung von *Koaleszenz*) der gewöhnlichen und der **null**-fähigen Werttypen zusätzlich gefördert:[1]



Der Variablen mit **null**-fähigem Werttyp wird genau dann der Wert des kompatiblen rechten Ausdrucks zugewiesen, wenn sie aktuell den Wert **null** besitzt.
Beispiele:

Quellcodesegment	Ausgabe

`int? ni = null;`
`ni ??= -1;`
`Console.WriteLine(ni);`
`ni ??= 0;`
`Console.WriteLine(ni);`
	`-1`
	`-1`

Die Anweisung

```
ni ??= -1;
```

ist äquivalent zu:

```
if (ni == null) ni = -1;
```

Der Null-Sammelzuweisungsoperator kann auch auf Referenztypen angewendet werden, z. B.:

```
string s = null;
s ??= "";
```

Das Bemühen von Compiler und CLR, eine **Nullable**-Instanz wie eine Instanz des zugehörigen Grundtyps zu behandeln, geht so weit, dass bei einer **GetType()** - Anfrage der Grundtyp genannt wird, z. B.:

Quellcodesegment	Ausgabe
`double? d = 3.0;` `Console.WriteLine(d.GetType());`	`System.Double`

Weitere Beispiele für generische Strukturen (mit spezieller syntaktischer Unterstützung durch den Compiler) haben wir übrigens im Zusammenhang mit den Tupeltypen kennengelernt (siehe Abschnitt 6.6.2).

[1] Die Bezeichnung *Null-Sammelzuweisungsoperator* wurde von der folgenden Webseite übernommen:
https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/operators/null-coalescing-operator
Eine alternative Bezeichnung lautet: *Null-Koaleszenz - Zuweisungsoperator*.

8.4 Generische Typen zur Laufzeit

Verwendet ein Programm einen generischen Typ mit einem Typformalparameter (z. B. **List<T>**), dann verhält sich die Laufzeitumgebung folgendermaßen:[1]

- Beim ersten Auftreten einer geschlossenen Konstruktion mit einem *Werttyp* als Aktualparameter (z. B. **List<int>**) wird im Speicher ein eigener Typ mit passender Ersetzung des Typparameters erstellt. Für jeden anderen Werttyp als Aktualparameter muss eine zusätzliche Konkretisierung mit passender Ersetzung des Typparameters erstellt werden.
- Beim ersten Auftreten einer geschlossenen Konstruktion mit einem *Referenztyp* als Aktualparameter (z. B. **List<String>**) wird ein Typ mit Ersetzung des Typparameters durch die Klasse **Object** erstellt. Diese Variante kann aber für *jede* geschlossene Konstruktion mit einem anderen *Referenz*-Typaktualparameter verwendet werden.

Über einen konkretisierten generischen Typ sind alle Laufzeitinformationen vorhanden, sodass z. B. eine Typprüfung per **is**-Operator möglich ist:

Quellcode	Ausgabe
`using System;` `public class SimpleStack<T> {` `  . . .` `}` `class Prog {` `  public static void Main() {` `  var sist= new SimpleStack<string>();` `  Console.WriteLine(sist.GetType());` `  Console.WriteLine(sist is SimpleStack<string>);` `  }` `}`	`SimpleStack`1[System.String]` `True`

8.5 Generische Methoden

Im Vergleich zu mehreren überladenen Methoden (vgl. Abschnitt 5.3.5), die analoge Operationen mit verschiedenen Datentypen ausführen, ist *eine* generische Methode oft die bessere Lösung. Im folgenden Beispiel liefert eine statische und generische Methode das Maximum von zwei Argumenten, wobei der gemeinsame Datentyp der Argumente die Schnittstelle **IComparable<T>** (vgl. Abschnitt 8.2.2) erfüllen, also eine Methode mit dem Definitionskopf

public int CompareTo(T *element***)**

besitzen muss:

[1] https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/generics/generics-in-the-run-time

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static T Max<T>(T x, T y) where T : IComparable<T> {`
`  return x.CompareTo(y) >= 0 ? x : y;`
`  }`
 	
`  public static void Main() {`
`  Console.WriteLine("int-max:\t" + Max(12, 13));`
`  Console.WriteLine("double-max:\t" + Max(2.16, 47.11));`
`  Console.WriteLine("String-max:\t" + Max("abc", "def"));`
`  }`
`}`
	`int-max:  13`
	`double-max:  47,11`
	`String-max:  def`

In der Definition einer generischen Methode befindet sich hinter dem Namen zwischen spitzen Klammern mindestens ein Typformalparameter. Mehrere Typparameter werden durch Kommata getrennt. Ein Typformalparameter ist als Datentyp erlaubt für:

- den Rückgabewert
- Parameter
- lokale Variablen

Wie bei generischen Typen kann man über das Schlüsselwort **where** Restriktionen für Typparameter formulieren.

Verwendet eine Methode einer *generischen* Klasse einen Typparameter der Klasse (als Parametertyp, Rückgabetyp oder Datentyp für eine lokale Variable), dann spricht man *nicht* von einer generischen Methode, weil keine *eigenen* Typparameter definiert werden, z. B. bei der Methode `Push()` der im Abschnitt 8.2.1 beschriebenen Klasse `SimpleStack<T>`:

```
public void Push(T element) {
  . . .
}
```

Die verwendeten Typparameter sind folglich *nicht* hinter dem Methodennamen zwischen spitzen Klammern anzugeben.

Beim Aufruf einer generischen Methode kann der Compiler fast immer aus den Datentypen der Aktualparameter die passende Konkretisierung ermitteln (Typinferenz). Daher konnte im obigen Beispielprogramm an Stelle der kompletten Syntax

```
Console.WriteLine("int-max:\t" + Max<int>(12, 13));
Console.WriteLine("double-max:\t" + Max<double>(2.16, 47.11));
Console.WriteLine("String-max:\t" + Max<string>("abc", "def"));
```

die folgende Kurzschreibweise verwendet werden:

```
Console.WriteLine("int-max:\t" + Max(12, 13));
Console.WriteLine("double-max:\t" + Max(2.16, 47.11));
Console.WriteLine("String-max:\t" + Max("abc", "def"));
```

Das Regelwerk zu der insgesamt recht komplexen Typinferenz ist in der C# - Sprachspezifikation im Abschnitt 12.6.3 beschrieben (ECMA 2017).

Der im Zusammenhang mit dem Überladen von Methoden relevante Begriff der *Methodensignatur* muss zur Berücksichtigung der Generizität angepasst werden (vgl. Abschnitt 5.3.5). Zwei Methoden besitzen genau dann *dieselbe* Signatur, wenn die folgenden Bedingungen erfüllt sind:

- Die Namen der Methoden sind identisch.
- Die Typformalparameterlisten sind gleich lang.
- Die Formalparameterlisten sind gleich lang.
- Positionsgleiche Formalparameter stimmen hinsichtlich Datentyp und Transfermodus (Wert- oder Verweisparameter) überein.

Irrelevant für die Signatur einer Methode sind (vgl. ECMA 2017, S. 46ff):

- Der Rückgabetyp
- Die Modifikatoren
- Die *Namen* der Formal- bzw. Typformalparameter
- Das beim letzten Formalparameter erlaubte **params**-Schlüsselwort.

8.6 default(T)

In einer Methode eines generischen Typs oder in einer generischen Methode ist es gelegentlich erforderlich, den typspezifischen Nullwert zu einem Typformalparameter zu verwenden. Aufgrund der zahlreichen möglichen Konkretisierungen eines Typformalparameters ist die Ermittlung des jeweiligen Standardwerts keine triviale Aufgabe, die man zum Glück dem **default**-Operator überlassen kann.[1] Der Ausdruck **default(T)** liefert ...

- den Wert **null**, wenn beim Konkretisieren des generischen Typs **T** ein Referenztyp oder ein **Nullable**-Strukturtyp (vgl. Abschnitt 8.3) angegeben wurde,
- den typspezifischen Nullwert, wenn für **T** ein elementarer Datentyp angegeben wurde,
- eine Strukturinstanz mit typspezifischen Nullwert-Initialisierungen für alle Felder, wenn für **T** ein Strukturtyp angegeben wurde:
  - Felder mit elementarem Datentyp erhalten den typspezifischen Nullwert.
  - Felder mit einem Referenztyp oder mit einem **Nullable**-Strukturtyp erhalten den Wert **null**.

Im folgenden Programm

```
using System;

struct MitNullInt {
   int i;
   int? inu;
   public MitNullInt(int i, int? inu) {
      this.i = i;
      this.inu = inu;
   }
   public override String ToString() {
      return "(" + i + ", " + (inu == null ? "null" : inu.ToString()) + ")";
   }
}

class DefaultDemo {
   static void WriteDef<T>() {
      Console.Write("default of " + typeof(T) + ": ");
      if (default(T) == null)
         Console.WriteLine("null");
      else
         Console.WriteLine(default(T));
   }
```

[1] Das Schlüsselwort **default** wird in gänzlich anderer Bedeutung auch in der **switch**-Anweisung verwendet (siehe Abschnitt 4.7.2.3).

```
    static void Main() {
      WriteDef<int>();
      WriteDef<System.Numerics.Complex>();
      WriteDef<String>();
      WriteDef<int?>();
      WriteDef<MitNullInt>();
    }
}
```

gibt eine generische Methode den **default**-Wert aus für:

- den numerischen Werttyp **int**,
- den Strukturtyp **Complex** aus dem BCL-Namensraum **System.Numerics** für komplexe Zahlen im Sinne der mathematischen Analysis, die aus einem Real- und einem Imaginärteil bestehen,[1]
- den Referenztyp **String**,
- den konkretisierten generischen Strukturtyp **Nullable<int>** (alias: **int?**),
- den selbst definierten Strukturtyp `MitNullInt`.

Es resultiert die Ausgabe:

```
default of System.Int32: 0
default of System.Numerics.Complex: (0, 0)
default of System.String: null
default of System.Nullable`1[System.Int32]: null
default of MitNullInt: (0, null)
```

Seit C# 7.1 kann im Ausdruck **default(T)** die Typangabe entfallen, wenn sie vom Compiler eindeutig zu ermitteln ist, z. B.

```
int idef = default;
```

8.7 Übungsaufgaben zum Kapitel 8

1) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. In C# können Typformalparameter auch durch Werttypen konkretisiert werden.
2. Wenn eine Methode einer generischen Klasse einen Typformalparameter aus dem Klassendefinitionskopf als Rückgabetyp verwendet, liegt eine generische Methode vor.
3. Weil **Object** eine Basisklasse von **String** ist, kann ein Objekt vom Typ **List<String>** durch eine Variable vom Typ **List<Object>** verwaltet werden.
4. Eine generische Klasse darf von einer nicht-generischen Klasse abstammen.

2) Als dynamisch wachsender Container für Elemente mit einem festen *Werttyp* (z. B. **int**) ist die Klasse **ArrayList** nicht gut geeignet, weil der Elementtyp **Object** zeitaufwändige (Un)boxing-Operationen erfordert. Dieser Aufwand entfällt bei einer passenden Konkretisierung der generischen Klasse **List<T>**, welche dieselbe Größendynamik bietet. Vergleichen Sie mit einem Testprogramm den Zeitaufwand beim Einfügen von 1 Million **int**-Werten in einen **ArrayList**- bzw. **List<int>** - Container.

3) Erstellen Sie eine verbesserte Version der generischen Methode `Max<T>()` aus dem Abschnitt 8.5, die einen generischen Serienparameter (siehe Abschnitt 5.3.1.3.3) akzeptiert und das maximale Element liefert.

---

[1] Mathematisch *nicht* vorbelastete Leser können sich unter einer komplexen Zahl ein Paar aus zwei reellen Zahlen vorstellen.

9 Interfaces

Zu vielen Klassen führt die BCL-Dokumentation zu .NET 5.0 hinter dem Namen und einem Doppelpunkt *mehrere* Typen auf, z. B. bei der Klasse **String**:

```
public sealed class String : ICloneable, IComparable, IComparable<string>,
   IConvertible, IEquatable<string>, System.Collections.Generic.IEnumerable<char>
```

Um sechs Basisklassen kann es sich nicht handeln, weil C# keine Mehrfachvererbung unterstützt. Außerdem ist der BCL-Dokumentation zu entnehmen, dass die Klasse **String** direkt von der Urahnklasse **Object** abstammt. Am Anfangsbuchstaben *I* sind in der BCL-Dokumentation zuverlässig die von einer Klasse oder Struktur implementierten *Schnittstellen* (engl.: *Interfaces*) zu erkennen. Hierbei handelt es sich um **Verpflichtungserklärungen** gegenüber dem Compiler. Ein Interface definiert eine Reihe von Handlungskompetenzen abstrakt (ohne Implementierung) über Definitionsköpfe von Methoden, Eigenschaften, Indexern und Ereignissen. Wenn sich ein Typ zu einem Interface bekennt, muss er die dort geforderten Handlungskompetenzen implementieren. Als Gegenleistung werden seine Instanzen vom Compiler überall dort akzeptiert (z. B. als Aktualparameter für einen Methodenaufruf), wo die jeweilige Schnittstelle als Datentyp vorgeschrieben ist.

Die Liste der von einem Typ implementierten Interfaces liefert wichtige Informationen über die Handlungskompetenzen seiner Instanzen. Über die Klasse **String** ist u. a. zu erfahren:

- **IComparable**, **IComparable<string>**
  Die Klasse implementiert das traditionelle Interface **IComparable** und die moderne Konkretisierung **IComparable<string>** der generischen Schnittstelle **IComparable<T>**, die beide zum Namensraum **System** gehören. Wie bei Klassen und Strukturen ermöglicht auch bei Schnittstellen die generische Definition mit Typformalparametern beliebig viele Konkretisierungen, sodass die Typsicherheit gewährleistet ist, und lästige Typumwandlungen entfallen.

  Weil **String** die Schnittstelle **IComparable** implementiert, muss eine Methode

  public int CompareTo(Object *obj***)**

  vorhanden sein.[1] Um den Vertrag **IComparable<string>** zu erfüllen, wird eine Methode mit dem folgenden Definitionskopf benötigt:

  public int CompareTo(string *str***)**

  Weil **String**-Objekte die Fähigkeit zum Vergleich mit Artgenossen besitzen, kann z. B. ein Array mit Elementen dieses Typs über die (statische) Methode **Array.Sort()** sortiert werden:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  String[] star = {"eins", "zwei", "drei"};`
`  Array.Sort(star);`
`  foreach (String s in star)`
`  Console.WriteLine(s);`
`  }`
`}`
	`drei`
	`eins`
	`zwei`

[1] Aus Kompatibilitätsgründen hält die Klasse **String** am Bekenntnis zur nicht-generischen Schnittstelle **IComparable** fest, sodass der Compiler kuriose Methodenaufrufe wie im folgenden Beispiel nicht verhindern kann:

```
Console.WriteLine("Alpha".CompareTo(3));
```

Die **String**-Methode **CompareTo(Object** *obj***)** reagiert darauf mit einer **ArgumentException**:

```
Unhandled exception. System.ArgumentException: Object must be of type String.
```

- **IClonable**
  Die Klasse **String** implementiert auch das Interface **ICloneable** (aus dem Namensraum **System**) und besitzt folglich eine Methode, welche eine Kopie des angesprochenen Objekts erzeugt:

  public Object Clone()

  Weil die Rückgabe den deklarierten Typ **Object** besitzt, ist bei der Zuweisung an eine **String**-Variable eine explizite Typumwandlung erforderlich, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  String s1 = "eins";`
`  String s2 = (String)s1.Clone();`
`  Console.WriteLine(s2);`
`  }`
`}`
	`eins`

  Zum Interface **ICloneable** gibt es keine generische Alternative. Ein Grund dafür mag darin bestehen, dass die **Clone()** - Methode und damit das **ICloneable**-Interface wegen einer Implementierungsunklarheit keinen guten Ruf besitzen. Es geht um den Unterschied zwischen der *tiefen* Kopie (engl.: *deep copy*), die auch alle direkten und indirekten Member-Objekte dupliziert, und der *flachen* Kopie (engl.: *shallow copy*), die Member-Objekte des Originals weiterverwendet. Bei fremden Klassen ist potentiell unklar, welche **Clone()** - Implementierung sie verwenden. Für eine eigene Klasse kann man eine tiefe Kopie nicht garantieren, sobald Member-Objekte aus Klassen vorhanden sind, die man nicht unter Kontrolle hat. Um **Clone()** zur Erstellung einer flachen Kopie zu implementieren, kann man auf die von der Klasse **Object** geerbte Methode **MemberwiseClone()** (mit Sichtbarkeit **protected**) zurückgreifen, z. B.:

```
public object Clone() {
  return MemberwiseClone();
}
```

Seit C# 8 sind in Schnittstellen neben den *abstrakt* definierten Verhaltenskompetenzen, die ein implementierender Typ selbst realisieren *muss*, auch *konkrete* Methoden mit einem vollständigen Rumpf (Anweisungsblock) erlaubt (siehe Abschnitt 9.1.1.2). Ein implementierender Typ kann eine konkrete Methode unverändert nutzen oder durch eine eigene Lösung überschreiben. Durch die in C# 8 eingeführten konkreten Methoden soll verhindert werden, dass bei der Erweiterung einer Schnittstelle um neue Methoden die bisher abgegebenen Verpflichtungserklärungen implementierender Typen ungültig werden. Primär dienen Schnittstellen aber weiterhin dem Zweck, Verhaltenskompetenzen von Typen abstrakt zu definieren.

Eine Schnittstelle ist ein **Datentyp**. Es lassen sich zwar keine *Instanzen* von diesem Typ erzeugen, aber *Referenzvariablen* sind erlaubt und als Abstraktionsmittel sehr nützlich. Sie dürfen auf Instanzen beliebiger Typen zeigen, welche die Schnittstelle implementieren. Somit können Instanzen unabhängig von den Vererbungsbeziehungen ihrer Typen gemeinsam verwaltet werden (z. B. in einem Array). Der Zweck einer Schnittstelle besteht darin, Handlungskompetenzen über abstrakte (und damit virtuelle) Methoden vorzuschreiben. Implementierende Typen realisieren diese Methoden konkret, und Methodenaufrufe erfolgen **polymorph** (mit später bzw. dynamischer Bindung, siehe Abschnitt 7.7).

Implementiert eine Klasse oder Struktur ein Interface, dann …

- muss sie die im Interface abstrakt enthaltenen Methoden, Eigenschaften, Indexer und Ereignisse konkret realisieren,
- werden Instanzen von diesem Typ vom Compiler überall dort akzeptiert, wo der Interface-Datentyp vorgeschrieben ist.

Im Programmieralltag kommen wir auf unterschiedliche Weise mit Schnittstellen in Kontakt, z. B.:

- **Implementierung von vorhandenen Schnittstellen in einer eigenen Typdefinition**
  Implementiert eine Klasse oder Struktur eine Schnittstelle, dann werden ihre Instanzen vom Compiler überall dort akzeptiert (z. B. als Aktualparameter bei einem Methodenaufruf), wo der jeweilige Schnittstellentyp gefordert ist.
  Beispiel: Wenn unsere Klasse `Bruch` das Interface **IComparable<`Bruch`>** implementiert, können wir die bequeme Methode **Array.Sort()** verwenden, um einen Array mit `Bruch`-Objekten zu sortieren.
- **Schnittstellen als Datentypen in eigenen Methoden- oder Typdefinitionen verwenden**
  In einer eigenen Methodendefinition ist es oft sinnvoll, Parameterdatentypen und/oder den Rückgabetyp über Schnittstellen festzulegen. In den Anweisungen der Methode werden Verhaltenskompetenzen der Parameterinstanzen genutzt, die durch Schnittstellenverpflichtungen garantiert sind. Bei einer Schnittstelle als Rückgabetyp weiß der Aufrufer, dass er ein Objekt mit bestimmten Verhaltenskompetenzen erhält; die genaue Klassenzugehörigkeit muss der Aufrufer nicht kennen. Damit wird die Typsicherheit ohne überflüssige Einengung erreicht. Ein Beispiel für eine Methode mit Interface-Parameter ist die folgende Überladung der Methode **Sort()** in der generischen BCL-Klasse **List<T>**:

  public void Sort(IComparer<T> *comparer***)**

  Ein die Schnittstelle **IComparer<T>** erfüllendes Objekt liefert eine frei definierbare Anordnung von zwei Elementen vom Typ **T** aufgrund der folgenden Methode:

  public int Compare (T *x***, T** *y***)**

  Auch bei Instanzvariablen oder statischen Variablen steigern Interface-Datentypen die Flexibilität. Sind bei der Definition eines generischen Typs für einen beschränkten Typformalparameter bestimmte Verhaltenskompetenzen zu fordern, gelingt das oft am besten per Schnittstellendatentyp (siehe Abschnitt 8.2.2).
- **Schnittstellen definieren**
  Natürlich kommen als Datentypen in Methoden- oder Typdefinitionen auch selbst definierte Schnittstellen in Frage.

9.1 Interfaces definieren

Wir behandeln zuerst das im Programmieralltag vergleichsweise seltene Definieren einer Schnittstelle, weil man dabei einen guten Eindruck von den Bestandteilen einer Schnittstelle und von ihrer Rolle bei der objektorientierten Programmierung gewinnt. Allerdings verzichten wir auf ein eigenes Beispiel und betrachten stattdessen die angenehm einfach aufgebaute und außerordentlich wichtige Schnittstelle **IComparable<T>** aus der BCL zum .NET - Framework 4.8:[1]

[1] Wenn der Name einer Schnittstelle mit einem kovarianten bzw. kontravarianten Typformalparameter (siehe Abschnitt 9.3) im Text auftaucht (z. B. **IComparable<T>**), dann wird das in der Definition erforderliche Schlüsselwort **out** bzw. **in** meist weggelassen.

```
public interface IComparable<in T> {
   // Interface does not need to be marked with the serializable attribute
   // Compares this object to another object, returning an integer that
   // indicates the relationship. An implementation of this method must return
   // a value less than zero if this is less than object, zero
   // if this is equal to object, or a value greater than zero
   // if this is greater than object.
   //
   int CompareTo(T other);
}
```

Wie der Kommentar im obigen .NET - Originalquellcode zeigt, sind bei einer Schnittstellen-Definition neben den syntaktischen Forderungen meist auch Verhaltenserwartungen im Spiel. Der Compiler kann aber z. B. aufgrund der obigen **IComparable<T>** - Definition sinnlose **CompareTo()** - Implementierungen *nicht* verhindern.

Einige Regeln für Schnittstellendefinitionen:

- Zugriffsmodifikatoren
  Bei einem Top-Level - Interface sind die Zugriffsmodifikatoren **public** und **internal** erlaubt. Wird **public** *nicht* angegeben, ist die Schnittstelle nur innerhalb ihres Assemblies verwendbar (Sichtbarkeit **internal**). Für ein eingeschachteltes Interface sind dieselben Zugriffsmodifikatoren verfügbar wie für andere Member.
- Modifikator **abstract**
  Schnittstellen sind grundsätzlich **abstract**. Der Modifikator **abstract** ist überflüssig und verboten.
- Schlüsselwort **interface**
  Das obligatorische Schlüsselwort dient zur Unterscheidung von Klassen- oder Strukturdefinitionen.
- Schnittstellenname
  Per Konvention beginnt der Interfacename mit einem großen *I*.

  Bei generischen Schnittstellen folgen auf den Namen die Typformalparameter zwischen spitzen Klammern und durch Kommata getrennt. Wie bei generischen Klassen und Strukturen können auch bei generischen Schnittstellen Restriktionen für die Typparameter formuliert werden (vgl. Abschnitt 8.2.2).

  Mit der seit .NET 4.0 erlaubten Kennzeichnung eines Typformalparameters als kovariant (Schlüsselwort **out**) bzw. kontravariant (Schlüsselwort **in**) beschäftigen wir uns im Abschnitt 9.3.
- Erlaubte Interface-Member
  Als Interface-Member sind *instanzbezogene* Methoden, Eigenschaften, Indexer und Ereignisse erlaubt. Verboten sind instanzbezogene Konstruktoren und Felder. Seit C# 8 sind hinzugekommen:
  - statische Methoden, Eigenschaften und Ereignisse
  - statische Felder
  - ein statischer Konstruktor
  - geschachtelte Typen

- Modifikatoren für Interface-Member
  Für alle Interface-Member ist die Sichtbarkeit **public** voreingestellt. Der Modifikator ist meist überflüssig, aber zulässig. Seit C# 8 sind für alle Interface-Member zusätzlich als Zugriffsmodifikatoren erlaubt: **private**, **protected**, **internal**. Für Methoden, Eigenschaften, Indexer und Ereignisse werden seit C# 8 außerdem u. a. die folgenden Modifikatoren unterstützt: **sealed**, **static**, **new**.[1]

9.1.1 Instanzmethoden

Die anschließenden Erläuterungen gelten analog auch für Eigenschaften, Indexer und Ereignisse.

9.1.1.1 Abstrakte Instanzmethoden

Durch abstrakte Instanzmethoden werden Verhaltenskompetenzen beschrieben, die implementierende Typen besitzen müssen. Auf den Methodendefinitionskopf folgt an Stelle des durch geschweifte Klammern begrenzten Rumpfes ein Semikolon, z. B.:

```
int CompareTo(T other);
```

Eine abstrakte Instanzmethode ist implizit **public**, **abstract** und **virtual**. Die Modifikatoren **abstract** und **public** sind überflüssig, aber erlaubt; der Modifikator **virtual** ist verboten. Bei der Implementierung einer abstrakten Schnittstellenmethode (siehe Abschnitt 9.4) ...

- muss (und darf) der Modifikator **override** *nicht* angegeben werden,
- darf die Sichtbarkeit **public** nicht reduziert werden, wobei der Modifikator **public** anzugeben ist.

Beim ersten Lesen ist es vertretbar, an dieser Stelle zum Abschnitt 9.4 zu springen. Die teilweise komplexen Details in den ausgelassenen Abschnitten sollten Sie sich erst dann zumuten, wenn es zum Verständnis von wichtigen Diskussionen oder Beispielen erforderlich ist.

9.1.1.2 Konkrete Instanzmethoden (mit Implementation)

Seit C# 8 ist es möglich, Instanzmethoden *mit* Implementation (also mit einem ausführbaren Methodenrumpf) in eine Schnittstelle aufzunehmen. Wir werden anschließend der Kürze halber meist von *konkreten Schnittstellenmethoden* sprechen. Implementierende Typen können eine konkrete Interface-Methode unverändert verwenden oder überschreiben.

Ein wesentliches Motiv für die Einführung der konkreten Schnittstellenmethoden bestand darin, die nachträgliche Erweiterung von Schnittstellen um neue Methoden ohne Nachteil für vorhandene implementierende Typen zu ermöglichen. Bis zur Version 7.x wurde in C# eine Möglichkeit vermisst, vorhandene Interfaces um neue Instanzmethoden zu erweitern, ohne die Kompatibilität mit implementierenden Typen zu verlieren. Seit C# 8 ist das Problem durch die Möglichkeit zur Erweiterung von Schnittstellen um konkrete Methoden gelöst. Vorhandene implementierende Typen erfüllen auch ein um konkrete Methoden erweitertes Interface, weil es ihnen keine zusätzlichen Pflichten aufbürdet.

Für Kenner der Programmiersprache Java ist offensichtlich, dass die mit Java 8 eingeführten **default**-Methoden das Vorbild für die Einführung der konkreten Schnittstellenmethoden in C# waren (siehe z. B. Baltes-Götz & Götz 2020).[2]

---

[1] Zur Bedeutung der Modifikatoren bietet Microsoft wenig mehr als das folgende Dokument:
https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/proposals/csharp-8.0/default-interface-methods

[2] Durch die Erweiterung um implementierte Schnittstellenmethoden greifen Java und C# auch das seit längerer Zeit in der Informatik diskutierte *Trait*-Konzept auf (Schärli et al. 2003). Dabei geht es um die Wiederverwendung von Methoden durch Typen, die in keiner Vererbungsbeziehung stehen müssen.

Wir betrachten ein einfaches Beispiel mit einer Schnittstelle namens `IWinterface`

```
interface IWinterface {
   void SagA();
}
```

und einer implementierenden Klasse namens `Cimple`:

```
class Cimple : IWinterface {
   public void SagA() {
      Console.WriteLine("A");
   }

   static void Main() {
      Cimple cimpl = new Cimple();
      cimpl.SagA();
   }
}
```

Wie es schon zu Beginn des Kapitels am Beispiel der Klasse **String** zu sehen war und gleich im Abschnitt 9.4 erläutert wird, muss eine Klasse oder Struktur das Bekenntnis zu einer Schnittstelle im Definitionskopf ankündigen:

- Auf den Namen der Klasse oder Struktur folgt ein Doppelpunkt.
- Dann folgt der Name der Schnittstelle.

Im Beispiel soll die Schnittstelle `IWinterface` um eine Instanzmethode `SagB()` erweitert werden, ohne alte Klassen (z. B. `Cimple`) ändern zu müssen. Dazu ergänzt man `SagB()` als konkrete Instanzmethode:

```
interface IWinterface {
   void SagA();

   void SagB() {
      SagA();
      Console.WriteLine("B");
   }
}
```

In einer konkreten Schnittstellenmethode dürfen andere Schnittstellenmethoden verwendet werden (abstrakte und konkrete, instanzbezogene und statische, öffentliche und private).

Implementiert eine bestehende Klasse eine neuerdings um eine konkrete Methode erweiterte Schnittstelle, dann bleibt die Klasse kompatibel zum erweiterten Interface, d. h. die Klasse muss weder erweitert, noch neu übersetzt werden.

Wenn ein Objekt der bestehenden Klasse unter Verwendung einer Referenzvariablen vom Typ der konkret erweiterten Schnittstelle „gezwungen“ wird, die neuartige und fremde Methode auszuführen, dann muss das aber nicht unbedingt harmlos enden. Bei der Erweiterung einer Basisklasse um eine zusätzliche Methode tritt allerdings für eine abgeleitete Klasse ein ähnliches Problem auf.

Eine neue bzw. aktualisierte Klasse, die das Interface implementiert, kann die konkrete Methode unverändert nutzen oder durch eine eigene Implementation überschreiben. Im folgenden Beispiel wird die *erste* Option verwendet. Es ist zu beachten, dass ein Objekt der Klasse `Cimple` über eine Referenz vom Typ der Schnittstelle `IWinterface` angesprochen werden muss, um die konkrete Schnittstellenmethode ausführen zu können:

Quellcode	Ausgabe

`using System;`
 	
`class Cimple : IWinterface {`
`  public void SagA() {`
`  Console.WriteLine("A");`
`  }`
 	
`  static void Main() {`
`  Cimple cimple = new Cimple();`
`  ((IWinterface)cimple).SagB();`
`  }`
`}`
	`A`
	`B`

Eine implementierende Klasse **erbt keine konkreten Schnittstellenmethoden**, sodass der folgende Aufruf scheitert:

```
cimple.SagB();
```

CS1061: "Cimple" enthält keine Definition für "SagB", und es konnte keine zugängliche SagB-Erweiterungsmethode gefunden werden, die ein erstes Argument vom Typ "Cimple" akzeptiert (möglicherweise fehlt eine using-Direktive oder ein Assemblyverweis).

Mögliche Korrekturen anzeigen (Alt+Eingabe oder Strg+.)

Wenn eine Klasse eine konkrete Schnittstellenmethode implementiert und dabei den **public**-Zugriff beibehält, dann wird die eigene Implementation auch beim Aufruf über eine Interface-Referenz verwendet (Polymorphie):

Quellcode	Ausgabe

`using System;`
 	
`class Cimple : IWinterface {`
`  public void SagA() {`
`  Console.WriteLine("A");`
`  }`
 	
`  public void SagB() {`
`  Console.WriteLine("BC");`
`  }`
 	
`  static void Main() {`
`  Cimple cimple = new Cimple();`
`  cimple.SagB();`
`  ((IWinterface)cimple).SagB();`
`  }`
`}`
	`BC`
	`BC`

Die Dominanz der eigenen Implementation beim Methodenaufruf per Schnittstellenreferenz gilt auch dann, wenn die eigene Implementation von der Basisklasse geerbt wurde.

Implementiert ein Typ *mehrere* Schnittstellen mit signaturgleichen konkreten Methoden und verwendet diese Methoden ohne eigene Realisierung, dann entscheidet der deklarierte Referenzvariablentyp darüber, welche Methode von einem Objekt ausgeführt wird, z. B.:

Quellcode	Ausgabe

`using System;`
`interface IWinterface {`
`  void SagWas() {Console.WriteLine("Winter");}`
`}`
`interface IOtherface {`
`  void SagWas() {Console.WriteLine("Other");}`
`}`
`class Cimple : IWinterface, IOtherface {`
`  static void Main() {`
`  Cimple cimple = new Cimple();`
`  IWinterface iw = cimple; iw.SagWas();`
`  IOtherface io = cimple; io.SagWas();`
`  }`
`}`
	`Winter`
	`Other`

Wenn ein implementierender Typ eine konkrete Schnittstellenmethode selbst realisiert, dann darf die Sichtbarkeit **public** durch Vergabe eines Zugriffsmodifikators reduziert werden, weil ja im Unterschied zu einer abstrakten Schnittstellenmethode beim Aufruf über eine Schnittstellenreferenz eine Implementation verfügbar ist. Allerdings kann (wie beim Verdecken von Methoden in abgeleiteten Klassen, vgl. Abschnitt 7.5.1) ein schlecht nachvollziehbares Programmverhalten resultieren. Lässt der implementierende Typ z. B. den Modifikator **public** weg, dann entsteht eine private Methode. Diese wird in eigenen Methoden über eine Referenz vom eigenen Typ angesprochen, während bei Verwendung der Schnittstellenreferenz die konkrete Schnittstellenmethode zum Einsatz kommt, z. B.:

Quellcode	Ausgabe

`using System;`
 	
`interface IWinterface {`
`  void SagWas() {Console.WriteLine("Winter");}`
`}`
 	
`class Cimple : IWinterface {`
`  void SagWas() {`
`  Console.WriteLine("Cimple");`
`  }`
 	
`  static void Main() {`
`  Cimple cimple = new Cimple();`
`  cimple.SagWas();`
`  ((IWinterface)cimple).SagWas();`
`  }`
`}`
	`Cimple`
	`Winter`

Weil ein Typ seit C# 8 die Möglichkeit hat, *mehrere* Schnittstellen mit konkreten Methoden zu implementieren, scheint die beim Design von C# bewusst ausgeschlossene Mehrfachvererbung durch eine Hintertür eingedrungen zu sein. Die mit der Mehrfachvererbung verbundenen Risiken bleiben aber ausgeschlossen, denn:

- Ein Typ *erbt* keine konkreten Schnittstellenmethoden. Diese können nur über eine Referenz vom Schnittstellentyp genutzt werden.
- In Schnittstellen sind Felder generell statisch. Folglich können *Instanzvariablen* nur von der Basisklasse übernommen werden, und der sogenannte *Deadly Diamond of Death* ist ausgeschlossen (siehe Kreft & Langer 2014).

Eine konkrete Instanzmethode einer Schnittstelle ist implizit **public** und **virtual**. Die Modifikatoren **public** und **virtual** sind in der Schnittstellendefinition überflüssig, aber erlaubt.

Der Zugriffsmodifikator **private** sorgt dafür, dass die konkrete Methode nur Schnittstellen-intern (von anderen konkreten Methoden) aufgerufen werden kann und natürlich nicht **virtual** ist.

Eine mit dem Modifikator **sealed** dekorierte konkrete Methode ist nicht **virtual**, kann also von einem implementierenden Typ *nicht* überschrieben werden. Bei einer als **private** deklarierten konkreten Methode ist der Modifikator **sealed** überflüssig und verboten.

Implementiert ein Typ eine konkrete, öffentliche und nicht versiegelte Schnittstellenmethode, dann muss (und darf) der Modifikator **override** *nicht* angegeben werden.

9.1.2 Statische Felder und ausführbare Member

Die folgende Schnittstelle erwartet von implementierenden Typen öffentliche Eigenschaften mit Lesezugriff für den Namen und den Geburtstag von Personen:

```
using System;

interface IPerson {
    string Name { get; }
    DateTime BirthDay { get; }
    const int allowedBirthDayDelay = 10;
    private static int maxBirthDayDelay = allowedBirthDayDelay;

    static int MaxBirthDayDelay {
        get { return maxBirthDayDelay; }
        set {
            if (value <= allowedBirthDayDelay)
                maxBirthDayDelay = value;
        }
    }

    void SendBirthDayGreetings() {
        DateTime birthDayThisYear =
                  new DateTime(DateTime.Today.Year, BirthDay.Month, BirthDay.Day);
        DateTime birthDayLastYear =
                  new DateTime(DateTime.Today.Year-1, BirthDay.Month, BirthDay.Day);
        double elapsedDays;
        if (DateTime.Today < birthDayThisYear)
            elapsedDays = (DateTime.Today - birthDayLastYear).TotalDays;
        else
            elapsedDays = (DateTime.Today - birthDayThisYear).TotalDays;
        if (elapsedDays >= 0 && elapsedDays <= maxBirthDayDelay)
            Console.WriteLine($"Liebe*r {Name}, herzlichen Glückwunsch zum Geburtstag!");
    }
}
```

Die Schnittstelle bietet eine konkrete Instanzmethode namens `SendBirthDayGreetings()`, die gratuliert, wenn der Geburtstag erst wenige Tage zurückliegt. Per Voreinstellung wird gratuliert, wenn der Geburtstag höchstens 10 Tage zurückliegt:

```
const int allowedBirthDayDelay = 10;
private static int maxBirthDayDelay = allowedBirthDayDelay;
```

In diesem Codesegment werden verwendet:

- Eine Konstante (ein Feld mit dem Modifikator **const**)
  Sie ist implizit statisch (vgl. Abschnitt 5.2.5).
- Ein statisches Feld

Die erlaubte Distanz zum Geburtstag kann von implementierenden Typen über eine konkrete statische Eigenschaft ermittelt und (in Grenzen) verändert werden:

```
  static int MaxBirthDayDelay {
    get { return maxBirthDayDelay; }
    set {
      if (value <= allowedBirthDayDelay)
         maxBirthDayDelay = value;
    }
  }
```

In der folgenden implementierenden Klasse `Person` wird die statische und konkrete Schnittstellen-Eigenschaft `MaxBirthDayDelay` genutzt, um die maximale Geburtstagsverzögerung festzulegen. Anschließend wird die konkrete Schnittstellen-Instanzmethode `SendBirthDayGreetings()` aufgerufen:

```
  class Person : IPerson {
    public string Name { get; private set;}
    public DateTime BirthDay { get; private set; }

    static void Main() {
      Person p = new Person() {Name = "Franz", BirthDay = new DateTime(1988, 12, 22)};
      IPerson.MaxBirthDayDelay = 8;
      IPerson iface = p;
      iface.SendBirthDayGreetings();
    }
  }
```

Die maximale Verzögerung könnte auch per Methodenparameter festgelegt werden, müsste dann aber bei jedem Gruß wiederholt werden.

Eine konkrete und statische Schnittstellenmethode mit der voreingestellten Sichtbarkeit **public** kann übrigens von einer Klasse auch ohne Implementierung der Schnittstelle genutzt werden, z. B.:

Quellcode	Ausgabe

`using System;`
 	
`interface IWinterface {`
`  static void SagWas() {`
`  Console.WriteLine("Winter");`
`  }`
`}`
 	
`class C {`
`  static void Main() {`
`  IWinterface.SagWas();`
`  }`
`}`
	`Winter`

Eine konkrete statische Methode mit dem Zugriffsmodifikator **protected** kann hingegen von einem Typ nur dann genutzt werden, wenn er die Schnittstelle implementiert.

9.2 Vererbung bzw. Erweiterung bei Schnittstellen

Ein Interface kann andere Interfaces **beerben** (bzw. erweitern), wobei dieselbe Syntax wie beim Ableiten von Klassen zu verwenden ist. In der BCL wird z. B. das generische Interface **IEnumerable<T>** durch das Interface **ICollection<T>** (beide im Namensraum **System.Collections.Generic**) erweitert:

```
public interface ICollection<T> : IEnumerable<T> {
   int Count { get; }
   bool IsReadOnly { get; }
   void Add(T item);
   void Clear();
   bool Contains(T item);
   void CopyTo(T[] array, int arrayIndex);
   bool Remove(T item);
}
```

Manchmal werden generische Schnittstellen als Erweiterung einer älteren, nicht-generischen Variante definiert, z. B. bei der Schnittstelle **IEnumerable<T>**:

```
public interface IEnumerable<out T> : IEnumerable {
   new IEnumerator<T> GetEnumerator();
}
```

Im konkreten Fall enthalten die abgeleitete Schnittstelle und die Basisschnittstelle

```
public interface IEnumerable {
   IEnumerator GetEnumerator();
}
```

eine Methode namens **GetEnumerator()** mit leerer Parameterliste. Weil die (unterschiedlichen) Rückgabewerte für die Methodensignatur unerheblich sind, liegen Methoden mit identischer Signatur vor, und die Methode der abgeleiteten Schnittstelle verdeckt die Variante der Basisschnittstelle. Um eine Warnung des Compilers zu vermeiden, wird in der abgeleiteten Schnittstelle der Modifikator **new** vor die Methode **GetEnumerator()** gesetzt (vgl. Abschnitt 7.5.1).

Bei der Implementation einer erweiternden Schnittstelle durch eine Klasse oder Struktur sind auch die Handlungskompetenzen der Basisschnittstellen zu realisieren. Im Fall der Schnittstelle **IEnumerable<T>** kommt es dabei zu einer Namenskollision, weil zwei parameterfreie Methoden namens **GetEnumerator()** implementiert werden müssen. Das Problem wird durch die sogenannte *explizite Schnittstellenimplementierung* gelöst (siehe Abschnitt 9.6 und Beispiel im Abschnitt 9.7.1).

Während bei *Klassen* die Mehrfachvererbung *nicht* unterstützt wird, ist sie bei *Schnittstellen* erlaubt.

Die Schnittstellenhierarchie ist unabhängig von der Klassenhierarchie, und ein Interface kann von beliebigen Klassen bzw. Strukturen implementiert werden.

Seit C# 8 sind die neu hinzugekommenen *konkreten* (implementierten) Methoden, Eigenschaften, Indexer und Ereignisse bei der Erweiterung von Schnittstellen zu berücksichtigen. In einer abgeleiteten (erweiternden) Schnittstelle kann eine geerbte *konkrete* Methode (oder ein sonstiges ausführbares Mitglied) durch eine eigene konkrete Variante überschrieben werden, wobei der Modifikator **override** weder erforderlich, noch zulässig ist. Wie bei der expliziten Schnittstellenimplementierung (siehe Abschnitt 9.6) ist der Name der Basisschnittstelle anzugeben, z. B.:

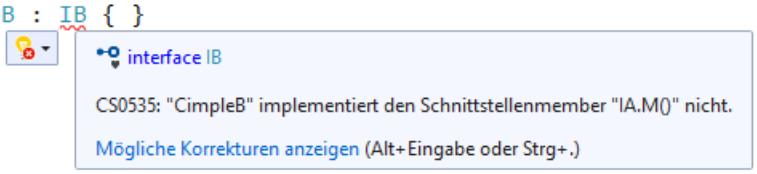
Quellcode	Ausgabe

`using System;`
 	
`interface IA {`
`  void M() { Console.WriteLine("IA.M"); }`
`}`
 	
`interface IB : IA {`
`  void IA.M() { Console.WriteLine("IB.M"); }`
`}`
 	
`class Cimple : IB {`
`  static void Main() {`
`  Cimple cimple = new Cimple();`
`  ((IA)cimple).M();`
`  ((IB)cimple).M();`
`  }`
`}`
	`IB.M`
	`IB.M`

Weil die Klasse `Cimple` die erweiternde (spezialisierende) Schnittstelle `IB` implementiert, wird beim `M()` - Aufruf über die Referenz vom `IA` die maximal spezialisierte (in `IB` definierte) Variante ausgeführt.

In einer abgeleiteten (erweiternden) Schnittstelle kann eine geerbte konkrete Methode durch eine abstrakte Definition ersetzt werden (reabstrahiert werden), z. B.:

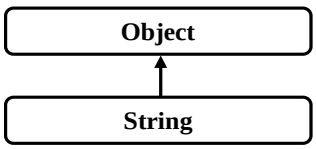
```
using System;

interface IA {
    void M() { Console.WriteLine("IA.M"); }
}

interface IB : IA {
    abstract void IA.M();
}

class CimpleA : IA { }

class CimpleB : IB { }
```



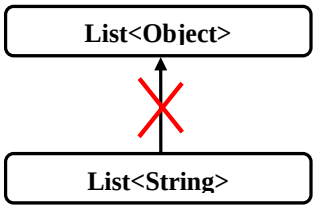
Die Klasse `CimpleB` muss die Instanzmethode `IA.M()` implementieren.

9.3 Kovariante und kontravariante Typparameter in generischen Schnittstellen

Im Abschnitt 8.2.3 wurde begründet, dass generische Kollektionstypen (z. B. **List<T>**) generell *kein kovariantes Verhalten* zeigen, dass sich also eine Spezialisierungsbeziehung zwischen zwei Aktualparametertypen wie im folgenden Beispiel



nicht auf die konkretisierten Kollektionsklassen überträgt:



Für das generell sinnvolle *invariante* Verhalten der generischen Kollektionsklassen ist aber in manchen Situationen eine Ausnahme erwünscht, was nun anhand von zwei anderen in der Spezialisierungsbeziehung stehenden Klassen erläutert werden soll. Wir betrachten eine Klasse namens `Figur`, die nur begrenzte Ähnlichkeit mit einer namensgleichen früheren Beispielklasse besitzt und durch Instanzeigenschaften u. a. die X- und die Y-Koordinate ihrer Objekte veröffentlicht:

```
public class Figur {
   protected String name = "unbenannt";
   protected double xpos, ypos;

   public Figur(string n, double x, double y) {
      name = n;
      if (x >= 0.0 && y >= 0.0) {
         xpos = x;
         ypos = y;
      }
   }
   public Figur() { }

   public String Name { get { return name; } }
   public double X { get { return xpos; } }
   public double Y { get { return ypos; } }
}
```

Von `Figur` stammt die Klasse `Kreis` ab:

```
public class Kreis : Figur {
   protected double radius;

   public Kreis (string n, double x, double y, double rad) : base(n, x, y) {
      if (rad >= 0.0) radius = rad;
   }
   public Kreis() { }

   public double Radius { get { return radius; } }
}
```

Es wird (in einer beliebigen Klasse) eine statische Methode namens `SmallestX()` definiert, die für eine Liste mit `Figur`-Elementen die insgesamt kleinste X-Koordinate ermittelt:

```
static double SmallestX(List<Figur> li) {
   double smallestX = Double.MaxValue;
   foreach (Figur f in li)
      if (f.X < smallestX)
         smallestX = f.X;
   return smallestX;
}
```

Während die Methode bei einem Aktualparameter mit dem Typ **List**`<Figur>` erwartungsgemäß arbeitet, scheitert die Übersetzung bei einem Aktualparameter mit dem Typ **List**`<Kreis>`, obwohl ein `Kreis`-Objekt alle Eigenschaften und Kompetenzen eines `Figur`-Objekts besitzt:

```
static void Main() {
    var listeF = new List<Figur>();
    listeF.Add(new Figur("A", 210.0, 50.0));
    listeF.Add(new Figur("B", 250.0, 100.0));
    var listeK = new List<Kreis>();
    listeK.Add(new Kreis("A", 250.0, 50.0, 100.0));
    listeK.Add(new Kreis("B", 250.0, 100.0, 120.0));
    Console.WriteLine(SmallestX(listeF));
    Console.WriteLine(SmallestX(listeK));
}
```

(Lokale Variable) List<Kreis> listeK

CS1503: Argument "1": Konvertierung von "System.Collections.Generic.List<Kreis>" in "System.Collections.Generic.List<Figur>" nicht möglich.

Mögliche Korrekturen anzeigen (Alt+Eingabe oder Strg+.)

Weil sich die generischen Kollektionsklassen invariant verhalten, ist die Klasse **List**`<Kreis>` *keine* Spezialisierung der Klasse **List**`<Figur>`. Daher verhindert der Compiler z. B., dass ein Objekt der Klasse **List**`<Kreis>` einer Referenzvariablen vom Typ **List**`<Figur>` zugewiesen wird. Über diese Referenz könnten nämlich `Figur`-Objekte in die `Kreis`-Liste eingeschmuggelt werden.

Bei der aktuell intendierten Verwendung eines **List**`<Kreis>` - Objekts als `SmallestX()` - Aktualparameter findet aber *keine Änderung* der Parameterliste statt. Es wäre wünschenswert, dass der Compiler die Zuweisung erlaubt, wenn man ihm versichert, dass eine als Aktualparameter fungierende Liste nur als *Quelle* von Objekten, nicht aber als *Senke* bzw. Ablage für Objekte verwendet wird. Genau dies ermöglichen die im Abschnitt 9.3.1 beschriebenen *kovarianten* Typformalparameter von Schnittstellen.

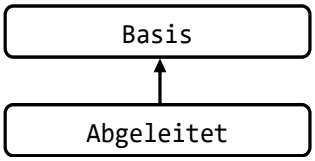
Im Abschnitt 9.3.2 werden *kontravariante* Typformalparameter vorgestellt, die eine weitere Flexibilisierung der Generizität in C# erlauben. Für beide Formen der Varianz (Ko- und Kontravarianz) gelten die folgenden Regeln:[1]

- Variante Typparameter werden seit .NET 4.0 unterstützt.
- Sie sind nur bei generischen Schnittstellen und generischen Delegaten (siehe Abschnitt 10.1.7) erlaubt, also insbesondere *nicht* bei generischen Klassen. Um das Problem im Beispiel zu beheben, muss also eine generische Schnittstelle ins Spiel kommen.
- Eine generische Schnittstelle oder ein generischer Delegat darf sowohl kovariante als auch kontravariante Typparameter besitzen.
- Die Ko- bzw. Kontravarianz klappt nur mit Referenztypen. Weil Werttypen nicht beerbt werden können, ist es allerdings ziemlich schwer, mit dieser Regel in Konflikt zu geraten.

9.3.1 Kovarianz

Das generische Interface **IEnumerable<out T>** im BCL-Namensraum **System.Collections.Generic** definiert die Voraussetzungen für eine iterierbare Kollektion (vgl. Abschnitt 9.7). Seit .NET 4.0 ist sein Typformalparameter **T** durch das Schlüsselwort **out** als kovariant definiert. Damit wird festgelegt, dass **T** *nicht* als Parametertyp einer Schnittstellenmethode, sondern ausschließlich als Rückgabetyp verwendet wird.[2]

Werden zu zwei Klassen mit der Spezialisierungsbeziehung



[1] https://docs.microsoft.com/en-us/dotnet/standard/generics/covariance-and-contravariance

[2] Selbst Ausgabeparameter vom Typ **T** sind verboten.

die **IEnumerable<out T>** - Konkretisierungen **IEnumerable<Basis>** und **IEnumerable<Abgeleitet>** gebildet, dann akzeptiert der Compiler an Stelle eines Objekts vom **IEnumerable<Basis>** auch ein Objekt vom Typ **IEnumerable<Abgeleitet>**.

Für die Schnittstelle **IEnumerable<out T>** ist die Deklaration des Typparameters als kovariant gerechtfertigt, weil die einzige Schnittstellenmethode **GetEnumerator()** ein Objekt liefert, das die Schnittstelle **IEnumerator<T>** erfüllt (vgl. Abschnitt 9.7):

IEnumerator<out T> GetEnumerator()

Auch in der Schnittstelle **IEnumerator<out T>** ist **T** als kovariant definiert. Der Typformalparameter hat dort seinen einzigen Auftritt als **get**-Rückgabe der Eigenschaft **Current**:

T Current { get; }

Im Ergebnis kann z. B. einer Variablen vom Typ **IEnumerable<Object>** ein Objekt vom Typ **List<String>** zugewiesen werden, weil dieses Objekt die Schnittstelle **IEnumerable<String>** erfüllt, und weil außerdem **String** von **Object** abstammt:

```
IEnumerable<Object> lob = new List<String>();
```

Über eine Referenz vom Schnittstellentyp **IEnumerable<Object>** wird von einem Objekt der Klasse **List<String>** eine **GetEnumerator()** - Rückgabe vom Typ **IEnumerator<Object>** erwartet. Das tatsächlich von `lob` erstellte Objekt vom Typ **IEnumerator<String>** wird dieser Erwartung gerecht, weil seine **Current**-Eigenschaft den Rückgabetyp **String** besitzt.

Generell kann bei einem kovarianten, also ausschließlich ausgaberelevanten Typ ein Objekt mit spezieller Konkretisierung ohne Risiko durch eine allgemeinere Referenz angesprochen werden. Es produziert bei Methodenaufrufen seine speziellen Instanzen, und alles ist in bester Ordnung.

Die oben vorgestellte Methode `SmallestX()` benötigt als Aktualparameter eine Kollektion mit den folgenden Eigenschaften:

- Das Interface **IEnumerable<out T>** wird erfüllt, damit die **foreach**-Schleife klappt.
- Als Elementtyp ist die Klasse `Figur` oder eine beliebige Ableitung erlaubt, damit die Eigenschaft `X` genutzt werden kann.

Daher sollte statt **List**`<Figur>` unbedingt **IEnumerable**`<Figur>` als Parameterdatentyp verwendet werden:

```
static double SmallestX(IEnumerable<Figur> li) {
   double smallestX = Double.MaxValue;
   foreach (Figur f in li)
      if (f.X < smallestX)
         smallestX = f.X;
   return smallestX;
}
```

Wegen der Kovarianz des **IEnumerable<out T>** - Typparameters **T** erfüllt der die Schnittstelle **IEnumerable**`<Kreis>` implementierende Typ **List**`<Kreis>` auch die Schnittstelle **IEnumerable**`<Figur>`. Daher kann ein Objekt der Klasse **List**`<Kreis>` als Aktualparameter an `SmallestX()` übergeben werden:

```
static void Main() {
   var listeF = new List<Figur>();
   listeF.Add(new Figur("A", 250.0, 50.0));
   listeF.Add(new Figur("B", 150.0, 50.0));
   listeF.Add(new Figur("C", 50.0, 50.0));

   var listeK = new List<Kreis>();
   listeK.Add(new Kreis("A", 210.0, 50.0, 100.0));
   listeK.Add(new Kreis("B", 250.0, 100.0, 120.0));
   listeK.Add(new Kreis("C", 130.0, 110.0, 80.0));
   Console.WriteLine(SmallestX(listeF));
   Console.WriteLine(SmallestX(listeK));
}
```

9.3.2 Kontravarianz

Über das generische Interface **IComparable<in T>** im BCL-Namensraum **System** signalisiert ein Datentyp, dass für seine Instanzen eine Ordnung definiert ist. Das Interface verlangt von implementierenden Typen, eine Methode mit dem folgenden Definitionskopf:

public int CompareTo(T *other***)**

Seit .NET 4.0 ist im Interface **IComparable<in T>** der Typformalparameter **T** über das Schlüsselwort **in** als *kontravariant* definiert. Damit wird dem Compiler signalisiert, dass **T** in den Schnittstellenmethoden *nicht* zur Spezifikation eines Rückgabetyps, sondern ausschließlich als Typ von Methoden*parametern* benutzt wird.

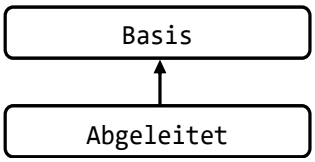
Durch die Kontravarianz des Typparameters von **IComparable<in T>** ist sichergestellt, dass Methoden dieser Schnittstelle ...

- lediglich **T**-Instanzen verarbeiten,
- aber keinesfalls **T**-Instanzen abliefern.

Wo ein die Schnittstelle **IComparable<in T>** implementierendes Objekt benötigt wird, kann auch ein die Schnittstelle **IComparable<in B>** implementierendes Objekt verwendet werden, wenn **B** eine Basisklasse von **T** ist, denn:

- Eine für Basisklassenobjekte **B** geeignete Methode kommt auch mit **T**-Objekten zurecht.
- Es passiert nie, dass statt eines **T**-Objekts ein (schwächer ausgestattetes) Basisklassenobjekt ausgeliefert wird.

Die Typsicherheit ist also nicht gefährdet. Infolgedessen wird bei zwei Klassen mit der Spezialisierungsbeziehung



für die beiden zugehörigen **IComparable<in T>** - Implementierungen die **Zuweisungskompatibilität umgekehrt**. Dies bedeutet, dass der Compiler an Stelle eines Objekts vom **IComparable<**`Abgeleitet`**>** auch ein Objekt vom Typ **IComparable<**`Basis`**>** akzeptiert.

Zur Demonstration implementieren wir in der Klasse `Figur` im Vorgriff auf den Abschnitt 9.4 die Schnittstelle **IComparable<**`Figur`**>**:

```
using System;
public class Figur : IComparable<Figur> {
   . . .
   public int CompareTo(Figur vergl) {
      if (xpos < vergl.xpos)
         return -1;
      else
         if (xpos > vergl.xpos)
            return 1;
      return 0;
   }
}
```

Weil die Klasse `Kreis` von der Klasse `Figur` abstammt, implementiert sie ebenfalls die Schnittstelle **IComparable**<`Figur`>. Wegen der Kontravarianz des Typformalparameters von **IComparable<in T>** ist **IComparable**<`Figur`> zuweisungskompatibel zu **IComparable**<`Kreis`>. Wo eigentlich der Typ **IComparable**<`Kreis`> verlangt ist, wird also auch **IComparable**<`Figur`> akzeptiert.

Daher kann zum Sortieren einer Liste mit Kreisen

```
var listeK = new List<Kreis>();
listeK.Add(new Kreis("A", 210.0, 50.0, 100.0));
listeK.Add(new Kreis("B", 250.0, 100.0, 120.0));
listeK.Add(new Kreis("C", 130.0, 110.0, 80.0));
```

die Methode **Sort()** der Klasse **List<T>** verwendet werden:

```
listeK.Sort();
```

Sie erwartet, dass der Elementtyp **T** die generische Schnittstelle **IComparable<T>** oder die nichtgenerische Schnittstelle **IComparable** erfüllt, wie ein Blick in die BCL-Dokumentation zeigt:

List<`Kreis`> implementiert zwar nicht **IComparable**<`Kreis`>, aber **IComparable**<`Figur`>, und das genügt wegen der Kontravarianz des Typparameters von **IComparable<in T>**.

9.4 Interfaces implementieren

Soll für eine Klasse oder Struktur angezeigt werden, dass ihre Instanzen die Kompetenzen bestimmter Schnittstellen besitzen, dann müssen diese Schnittstellen im Kopf der Typdefinition aufgelistet werden. Man setzt hinter den Typbezeichner einen Doppelpunkt, gibt bei Klassen ggf. zunächst eine Basisklasse an und listet dann die Schnittstellen auf, untereinander und von der Basisklasse jeweils durch ein Komma getrennt.

Ein Beispiel kennen Sie bereits, weil im Abschnitt 9.3.2 zur Erläuterung der Kontravarianz eine Variante der Klasse `Figur` vorgestellt wurde, die das Interface **IComparable**<`Figur`> implementiert, sodass `Figur`-Kollektionen bequem sortiert werden können:

```
using System;
public class Figur : IComparable<Figur> {
   protected String name = "unbenannt";
   protected double xpos, ypos;

   public Figur(string n, double x, double y) {
      name = n;
      if (x >= 0.0 && y >= 0.0) {
         xpos = x;
         ypos = y;
      }
   }
   public Figur() {}

   public String Name { get { return name; } }
   public double X { get { return xpos; } }
   public double Y { get { return ypos; } }

   public int CompareTo(Figur vergl) {
      if (xpos < vergl.xpos)
         return -1;
      else
         if (xpos > vergl.xpos)
            return 1;
      return 0;
   }
}
```

Alle abstrakt definierten Handlungskompetenzen einer Schnittstelle, die im Kopf einer Klassen- oder Strukturdefinition angemeldet wird, müssen implementiert werden. Bei der generischen Schnittstelle **IComparable<in T>** ist nur eine **public**-Methode namens **CompareTo()** mit einem Parameter vom Typ **T** und einem Rückgabewert vom Typ **int** erforderlich. In semantischer Hinsicht soll **CompareTo()** eine `Figur` beauftragen, sich mit dem per Aktualparameter bestimmten Artgenossen zu vergleichen. Bei der obigen Realisation werden Figuren nach der X-Koordinate ihrer linken oberen Ecke verglichen:[1]

- Liegt die angesprochene Figur links vom Vergleichspartner, dann wird die Zahl -1 zurück gemeldet.
- Haben beide Figuren in der linken oberen Ecke dieselbe X-Koordinate, lautet die Antwort 0.
- Ansonsten wird eine 1 gemeldet.

Weil im Beispiel der `Figur`-Vergleich auf einem **double**-Vergleich basiert, kann die `Figur`-Methode **CompareTo()** eleganter formuliert werden:[2]

```
public int CompareTo(Figur vergl) {
   return xpos.CompareTo(vergl.xpos);
}
```

Bei dieser für die Praxis empfohlenen Lösung ist allerdings nicht mehr erkennbar, wie sich eine **CompareTo()** - Methode verhalten muss. Weil das korrekte **CompareTo()** - Verhalten aktuell im Mittelpunkt steht, wird die umständliche Implementation der Klasse `Figur` beibehalten.

---

[1] In der `Figur`-Methode **CompareTo()** werden die im Abschnitt 4.5.4 diskutierten Genauigkeitsprobleme des binären Gleitkommatyps **double** der Einfachheit halber ignoriert.

[2] Die elegante Lösung stammt von Paul Frischknecht.

Weil die abstrakten Methoden einer Schnittstelle grundsätzlich **public** sind, muss diese Schutzstufe auch für die *implementierenden* Methoden gelten, wozu in deren Definition der Zugriffsmodifikator explizit anzugeben ist. Anderenfalls äußert sich der Compiler so:

```
Figur.cs(2,14): error CS0737: "Figur" implementiert den Schnittstellenmember
        "System.IComparable<Figur>.CompareTo(Figur)" nicht.
        "Figur.CompareTo(Figur)" ist nicht öffentlich und kann daher keinen
        Schnittstellenmember implementieren.
```

Für die implementierenden Methoden muss (und darf) das Schlüsselwort **override** (im Unterschied zum Überschreiben von abstrakten Basisklassenmethoden) *nicht* angegeben werden.

Das gilt auch, wenn ein Typ eine konkrete, öffentliche und nicht versiegelte Schnittstellenmethode überschreibt.

Eine Klasse kann auf das Implementieren einiger abstrakter Interface-Handlungskompetenzen verzichten und diese wie auch sich selbst als **abstract** deklarieren.

Weil sich `Figur`-Objekte mit einem Artgenossen vergleichen können, gelingt das Sortieren einer **List<Figur>** - Kollektion mit der Methode **Sort()** mühelos, z. B.:[1]

Quellcode	Ausgabe

`using System;`
`using System.Collections.Generic;`
`class Prog {`
`  static void Main() {`
`  var listeF = new List<Figur>();`
`  listeF.Add(new Figur("A", 250.0, 50.0));`
`  listeF.Add(new Figur("B", 150.0, 50.0));`
`  listeF.Add(new Figur("C", 50.0, 50.0));`
`  foreach (var v in lf)`
`  Console.Write(v.Name + " ");`
`  Console.WriteLine();`
`  lf.Sort();`
`  foreach (var v in lf)`
`  Console.Write(v.Name + " ");`
`  }`
`}`
	`A B C`
	`C B A`

In der Dokumentation zur Methode **Sort()** in der generischen Klasse **List<T>** wird verlangt, dass eine Konkretisierung des Elementtyps **T** die generische Schnittstelle **IComparable<T>** oder die nicht-generische Schnittstelle **IComparable** erfüllen muss (siehe Abschnitt 9.3.2). Der Typformalparameter der generischen Klasse **List<T>** ist aber *nicht* restringiert:

```
public class List<T> : IList<T>, IList, IReadOnlyList<T> { ... }
```

Der Compiler protestiert daher *nicht*, wenn die Methode **Sort()** auf eine Konkretisierung von **List<T>** angewendet wird, obwohl der konkretisierende Typ weder die generische Schnittstelle **IComparable<T>** noch die Schnittstelle **IComparable** implementiert. In diesem Fall wird der Quellcode fehlerfrei übersetzt, doch beim Aufruf von **Sort()** kommt es zu einem Laufzeitfehler vom Typ **InvalidOperationException**. Hier besteht wohl noch Potential zu einer Verbesserung des Compilers.

Ob ein Typ eine Schnittstelle erfüllt, kann über den Typtest-Parameter **is** festgestellt werden, z. B.:

```
if (listeK is IComparable<Kreis>)
    listeK.Sort();
```

[1] Auf die Bequemlichkeit eines Kollektionsinitialisierers (siehe Abschnitt 11.3.1) wird hier bewusst verzichtet, um die Anzahl der Vorgriffe im Manuskript nicht zu groß werden zu lassen.

Die von der Klasse `Figur` abgeleitete Klasse `Kreis` erbt von ihrer Basisklasse die Implementation der Schnittstelle **IComparable**<`Figur`>. Wegen der Kontravarianz des Typformalparameters **T** im generischen Interface **IComparable<in T>** wird ein die Schnittstelle **IComparable**<`Figur`> implementierender Typ akzeptiert, wo ein die Schnittstelle **IComparable**<`Kreis`> implementierender Typ verlangt ist (siehe Abschnitt 9.3.2).

Wie Sie wissen, ist bei C# - *Klassen* keine Mehrfachvererbung erlaubt. Diese Möglichkeit wurde wegen einiger Risiken bewusst *nicht* aus C++ übernommen. Allerdings erlauben Schnittstellen in manchen Fällen eine Ersatzlösung, denn:

- Eine Klasse darf beliebig viele Schnittstellen implementieren, sodass ihre Objekte entsprechend viele Datentypen erfüllen. So könnte man z. B. die Schnittstellen `ITuner` und `IAmplifier` sowie die Klasse `Receiver` derart definieren, dass sich ein `Receiver`-Objekt …
  - wie ein `ITuner`
  - und wie ein `IAmplifier`

  verhalten kann. Wie wir inzwischen wissen, erbt eine Klasse beim Implementieren von Schnittstellen aber keine Methoden, sondern sie gibt Verpflichtungserklärungen ab und muss die entsprechenden Leistungen erbringen. Ein Typ erbt auch die seit C# 8 unterstützten konkreten Schnittstellenmethoden nicht. Wenn eine Klasse diese Methoden unverändert übernimmt, können sie nur über eine Referenz vom Schnittstellentyp angesprochen werden.
- Bei Schnittstellen ist die Mehrfachvererbung erlaubt. Diese Möglichkeit wird aber sehr viel seltener benutzt als das Implementieren von mehreren Schnittstellen durch eine Klasse oder Struktur. Implementiert eine Klasse oder Struktur eine Schnittstelle mit mehreren Basisschnittstellen, dann muss sie alle abstrakten Handlungskompetenzen aus den beteiligten Schnittstellen realisieren.

Im Zusammenhang mit dem Thema *Vererbung* ist von Bedeutung, dass eine abgeleitete Klasse die Schnittstellen-Zulassungen von ihrer Basisklasse übernimmt, ohne die Verpflichtungserklärungen in ihrem eigenen Definitionskopf wiederholen zu müssen. Wird z. B. die Klasse `Kreis` (siehe Abschnitt 9.3) von der oben vorgestellten Klasse `Figur` abgeleitet, dann erfüllt auch die Klasse `Kreis` die Schnittstelle **IComparable**<`Figur`>, jedoch *nicht* die Schnittstelle **IComparable**<`Kreis`>. Dass ein die Schnittstelle **IComparable**<`Figur`> implementierender Typ akzeptiert wird, wenn eigentlich das Implementieren von **IComparable**<`Kreis`> verlangt ist, liegt an der seit .NET 4 gegebenen Kontravarianz des Typformalparameters **T** in der generischen Schnittstelle **IComparable<in T>** (siehe Abschnitt 9.3.2). Unter dieser Voraussetzung arbeitet die **List<T>** - Methode **Sort()** auch bei einer **List**<`Kreis`> - Kollektion, z. B.:

Quellcode	Ausgabe

```
using System;
using System.Collections.Generic;
class Prog {
   static void Main() {
      var lk = new List<Kreis>();
      lk.Add(new Kreis("A", 250.0, 50.0, 10.0));
      lk.Add(new Kreis("B", 150.0, 50.0, 20.0));
      lk.Add(new Kreis("C", 50.0, 50.0, 30.0));
      foreach (var v in lk)
         Console.Write(v.Name + " ");
      Console.WriteLine();
      lk.Sort();
      foreach (var v in lk)
         Console.Write(v.Name + " ");
  }
}
```

```
A B C
C B A
```

Im Vergleich zu einem nicht-generischen Interface hat ein analoges generisches Interface folgende Vorteile:

- Man gewinnt Typsicherheit,
- erspart sich lästige Typumwandlungen
- und vermeidet bei Strukturen zeitaufwändige Boxing-Operationen.

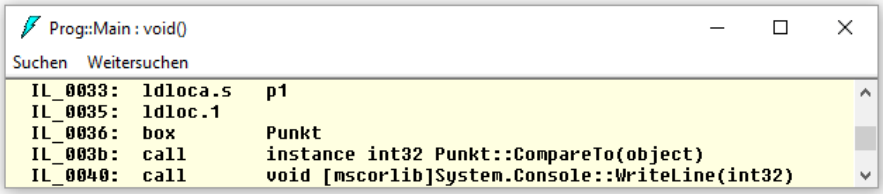
Dies soll bei einer einfachen Struktur für Punkte in der Zahlenebene und einer durch die X-Koordinate definierten Anordnung demonstriert werden. In der ersten Variante wird die traditionelle Schnittstelle **IComparable** implementiert, also eine **CompareTo()** - Methode mit **Object**-Parameter realisiert:

```
using System;
public struct Punkt : IComparable {
   double xpos, ypos;
   public Punkt(double x, double y) {
      xpos = x; ypos = y;
   }
   public int CompareTo(object v) {
      Punkt vergl = (Punkt) v;
      if (xpos < vergl.xpos)
         return -1;
      else
         if (xpos > vergl.xpos)
            return 1;
      return 0;
   }
}
```

In **CompareTo()** ist eine explizite Typumwandlung von **Object** zu `Punkt` erforderlich, die zu einer vom Compiler nicht zu verhindernden **InvalidCastException** führen kann, z. B. im folgenden Programm:

```
using System;
class Prog {
   static void Main() {
      Punkt p1 = new Punkt(1.0, 2.0),
            p2 = new Punkt(2.0, 3.0);
      Console.WriteLine(p1.CompareTo(p2)); // Boxing
      Console.WriteLine(p1.CompareTo(13)); // Laufzeitfehler
   }
}
```

Wie der IL-Code der **Main()** - Methode zeigt, erfordert außerdem jeder gelungene **CompareTo()** - Aufruf eine Boxing -Operation:



Implementiert die Struktur stattdessen das Interface **IComparable**<Punkt>,

```
using System;
public struct Punkt : IComparable<Punkt> {
   double xpos, ypos;
   public Punkt(double x, double y) {
      xpos = x; ypos = y;
   }
   public int CompareTo(Punkt vergl) {
      if (xpos < vergl.xpos)
         return -1;
      else
         if (xpos > vergl.xpos)
            return 1;
      return 0;
   }
}
```

dann kann die fehlerhafte Quellcodezeile nicht mehr übersetzt werden,

```
Console.WriteLine(p1.CompareTo(13));
```

und ein gelungener **CompareTo()** - Aufruf geht ohne Boxing über die Bühne:

9.5 Interfaces als Referenzdatentypen

Bei der Definition einer Schnittstelle entsteht ein neuer *Referenzdatentyp*, der zur Variablendeklaration und als Formalparameter einsetzbar ist. Eine Referenzvariable des neuen Typs kann auf Instanzen einer implementierenden Klasse oder Struktur zeigen, z. B.:

Quellcode	Ausgabe

`using System;`
`public interface IType {`
`  string SagWas();`
`}`
`class K1 : IType {`
`  public string SagWas() { return "K1"; }`
`}`
`class K2 : IType {`
`  public string SagWas() { return "K2"; }`
`}`
`struct S : IType {`
`  public string SagWas() { return "S"; }`
`}`
`class Prog {`
`  static void Main() {`
`  IType[] ida = {new K1(), new K2(), new S()};`
`  foreach (IType idin in ida)`
`  Console.WriteLine(idin.SagWas());`
`  }`
`}`
	`K1`
	`K2`
	`S`

Damit wird es möglich, Instanzen von beliebigen Klassen und Strukturen, die dasselbe Interface implementieren, in einem Array (oder einer anderen Kollektion) gemeinsam zu verwalten. Über eine Interface-Variable können die Methoden der Schnittstelle sowie die Methoden der Klasse **Object** aufgerufen werden.

Ist eine öffentliche Schnittstellenmethode virtuell, also ...

- entweder abstrakt
- oder konkret und nicht versiegelt,

dann findet beim Aufruf eine späte Bildung statt (Polymorphie).

Interface-Typen sind grundsätzlich *Referenztypen*, sodass eine Variable mit einem solchen Datentyp nur eine Objektadresse aufnehmen kann. Wird einer solchen Referenzvariablen eine Strukturinstanz zugewiesen, ist ein Boxing fällig. In zeitkritischen Programmen muss eine hohe Zahl von Boxing-Operationen vermieden werden.

Werden Instanzen eines Strukturtyps, der ein Interface implementiert, bei der Ansprache per Interface-Referenz temporär in ein Objekt verpackt, kann es zu tückischen Programmierfehlern kommen. Im folgenden, von einer Microsoft-Webseite stammenden Beispiel wird vergeblich versucht, eine Instanzvariable einer per Interface-Referenz angesprochenen Strukturinstanz zu inkrementieren:[1]

```
using System;

interface IPromotion {
    void Promote();
}

struct Employee : IPromotion {
    public string Name;
    public int JobGrade;
```

[1] https://docs.microsoft.com/en-us/archive/blogs/abhinaba/c-structs-and-interface

```
        public void Promote() {
            JobGrade++;
        }

        public Employee(string name, int jobGrade) {
            this.Name = name;
            this.JobGrade = jobGrade;
        }

        public override string ToString() {
            return string.Format("{0} ({1})", Name, JobGrade);
        }
    }

    class Prog {
        static void Main(string[] args)    {
            Employee employee = new Employee("Cool Guy", 65);
            IPromotion p = employee;
            Console.WriteLine(employee);
            p.Promote();
            Console.WriteLine(employee);
        }
    }
```

Durch die Methode `Promote()` wird lediglich das per Autoboxing erstellte temporäre Objekt geändert, aber nicht die Strukturinstanz. Daher produziert das Programm die Ausgabe:

```
Cool Guy (65)
Cool Guy (65)
```

Macht man aus der Struktur `Employee` eine Klasse, dann verhält sich das Programm wunschgemäß:

```
Cool Guy (65)
Cool Guy (66)
```

9.6 Explizite Schnittstellenimplementierung

Virtuelle Methoden von Schnittstellen werden in der Regel durch **public**-deklarierte Methoden des implementierenden Typs realisiert. Bei dieser sogenannten *impliziten* Implementation kann es zu Namenskollisionen kommen, wenn …

- ein Typ mehrere Schnittstellen implementiert,
- und zwei oder mehrere Schnittstellen eine signaturgleiche Methode besitzen, für die aber unterschiedliche Funktionsweisen vorgesehen sind.

Für diese relativ seltene Situation bietet C# die sogenannte *explizite* Schnittstellenimplementierung, wobei der implementierende Typ …

- die Methode mehrfach implementiert,
- bei jeder Implementation dem Methodennamen den Namen der zugehörigen Schnittstelle durch Punkt getrennt voranstellt,
- *keine* Zugriffsmodifikatoren angibt.

Im folgenden Beispiel implementiert die Klasse `M` die Methoden `I1.M()` und `I2.M()` durch explizite Angabe der jeweiligen Schnittstelle:

```
public interface I1 {
   int M();
}
public interface I2 {
   int M();
}
public class K : I1, I2 {
   int I1.M() {
      return 13;
   }

   int I2.M() {
      return 4711;
   }
}
```

Objekte der Klasse `M` können *beide* Methoden ausführen, sofern sie über den passenden Interface-Datentyp angesprochen werden, z. B.:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  K k = new K();`
`  I1 i1 = k;`
`  I2 i2 = k;`
`  Console.WriteLine(i1.M());`
`  Console.WriteLine(i2.M());`
`  }`
`}`
	`13`
	`4711`

Über eine Referenzvariable vom eigenen Datentyp angesprochen, ist jedoch *keine* Methode namens `M()` verfügbar, sodass die folgende Anweisung

```
Console.WriteLine(k.M());
```

den Compiler zu einer Fehlermeldung veranlasst:

```
"K" enthält keine Definition für "M"
```

Würde die Klasse `K` die Methode `M()` (ausschließlich) auf bisher gewohnte Weise implementieren, dann wäre *dasselbe* Verhalten über Referenzen der Typen `I1`, `I2` und `K` abrufbar.

Weitere Hinweise zur expliziten Implementation finden sich z. B. bei Richter (2012, Kap. 13).

9.7 Iteratoren

Ein Iterator erlaubt es, die Elemente einer Kollektion nacheinander abzurufen und wird meist implizit im Rahmen einer **foreach**-Schleife verwendet.

9.7.1 IEnumerable<T> - Implementation

Für das im Abschnitt 9.3.1 zur Illustration der Kovarianz verwendete und offenbar sehr wichtige (weil z. B. für die **foreach**-Schleife relevante) Interface **IEnumerable<T>**

```
public interface IEnumerable<out T> : IEnumerable {
   new IEnumerator<T> GetEnumerator();
}
public interface IEnumerable {
   IEnumerator GetEnumerator();
}
```

ist (wie auch für die nicht-generische Variante **IEnumerable**) noch zu erläutern, wie die Methode **GetEnumerator()** zu implementieren ist. Dabei kommt eine spezielle, als *Iterator* bezeichnete Methode ins Spiel. Was zunächst nach Lernaufwand klingt, stellt sich bald als Entlastung heraus, weil wir darum herumkommen, eine Klasse zu erstellen, die das Interface **IEnumerator<T>** inklusive der Erblasten **IDisposable** und **IEnumerator** implementiert:

```
public interface IEnumerator<out T> : IDisposable, IEnumerator {
   new T Current {
      get;
   }
}
public interface IEnumerator {
   bool MoveNext();
   Object Current {
      get;
   }
   void Reset();
}
public interface IDisposable {
   void Dispose();
}
```

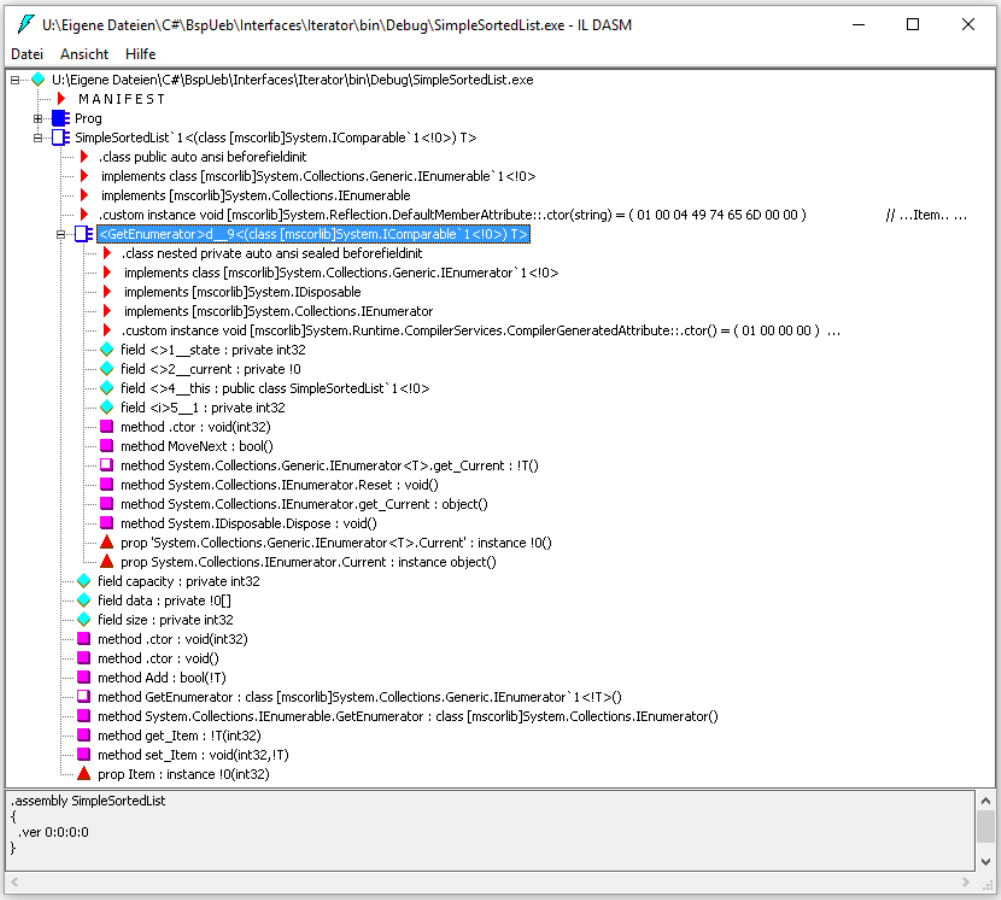
Als Anwendungsbeispiel erweitern wir die im Abschnitt 8.2.2 erstellte Kollektionsklasse `SimpleSortedList<T>`, die ihre Elemente in einem sortieren Zustand hält, um einen Iterator, damit über die Elemente per **foreach**-Schleife iteriert werden kann. Der Zweck ist im Wesentlichen bereits durch die folgende **GetEnumerator()** - Implementation erreicht:

```
public IEnumerator<T> GetEnumerator() {
   for (int i = 0; i < size; i++)
      yield return data[i];
}
```

Bei einem Objekt der Klasse `SimpleSortedList<T>` befinden sich die Elemente in einem privaten Array namens `data`. Der Iterator liefert in einer gewöhnlichen **for**-Schleife die Elemente Stück für Stück in einer Anweisung mit dem Namen **yield return** aus.

Im praktischen Einsatz wird nach jeder **yield return** - Ausführung die erreichte Position von der Laufzeitumgebung gespeichert, und beim nächsten Aufruf des Iterators wird die Ausführung an der gespeicherten Position fortgesetzt.[1] Wir wollen gar nicht erst darüber nachdenken, mit welcher Genialität und mit welchem Aufwand der Compiler aus unserem simplen Beitrag eine **IEnumerator<T>** - Implementation erstellt. Eine Inspektion mit dem Hilfsprogramm **ILDasm** zeigt, dass der Compiler unsere **GetEnumerator()** - Implementation in eine innere Klasse umgesetzt hat:

[1] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/iterators



Aufgrund der simplen **GetEnumerator()** - Implementation werden Objekte der Klasse `SimpleSortedList<T>` nun von der **foreach**-Schleife akzeptiert:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  var sst = new SimpleSortedList<string>(5);`
`  sst.Add("one"); sst.Add("two"); sst.Add("three");`
`  sst.Add("four"); sst.Add("five");`
`  foreach (string s in sst)`
`  Console.WriteLine(s);`
`  }`
`}`
	`five`
	`four`
	`one`
	`three`
	`two`

Eine **foreach**-Anweisung wird vom Compiler ungefähr so umgesetzt (siehe ECMA 2017, S. 232ff):

- Durch einen Aufruf der Methode **GetEnumerator()** wird ein Objekt erstellt, das die Schnittstelle **IEnumerator<T>** erfüllt (anschließend *Enumerator* genannt).
- Im **try**-Block einer **try-finally** - Anweisung läuft eine **while**-Schleife, die im Bedingungsteil einen Aufruf der **MoveNext()** - Methode des Enumerators enthält. Im Anweisungsteil der **while**-Schleife wird die **Current**-Eigenschaft des Enumerators verwendet, um das aktuelle Element abzurufen.
- Im **finally**-Zweig der **try-finally** - Anweisung wird die **Dispose()** - Methode des Enumerators aufgerufen. So ist sichergestellt, dass **Dispose()** auch beim Auftreten eines Fehlers aufgerufen wird.

Wegen des Aufwands und des Fehlerrisikos sollte ein Enumerator in der Regel nur indirekt via **foreach** eingesetzt werden.

Wenn die Klasse `SimpleSortedList<T>` von sich behaupten möchte, die Schnittstelle **IEnumerable<T>** zu implementieren,

```
public class SimpleSortedList<T> : IEnumerable<T>  where T : IComparable<T> { ... }
```

muss sie auch die nichtgenerische Überladung von **GetEnumerator()** implementieren, was aber praktisch keinen Zusatzaufwand bedeutet:

```
IEnumerator IEnumerable.GetEnumerator() {
   return GetEnumerator();
}
```

Weil die gerade vorgestellte **GetEnumerator()** - Überladung die zur Schnittstelle **IEnumerable** gehörige Methode implementiert, ist der Schnittstellenname voranzustellen (siehe Abschnitt 9.6 zur expliziten Schnittstellenimplementierung).

9.7.2 Benannte Iteratoren

Als Alternative oder Ergänzung zu der eben beschriebenen Implementation der Schnittstelle **IEnumerable<out T>** lassen sich in C# benannte Iteratoren mit Methoden- oder Eigenschaftssyntax realisieren.

In der Klasse `SimpleSortedList<T>` soll als Ergänzung zum Standarditerator auch ein Iterator mit Parametern zur Bereichsspezifikation angeboten werden. Dazu eignet sich ein benannter Iterator mit Methodensyntax. Statt **GetEnumerator()** ist ein alternativer Methodenname zu verwenden, und als Rückgabetyp ist **IEnumerable<T>** vorgeschrieben, z. B.:

```
public IEnumerable<T> RangeIt(int first, int last) {
   if (first < 0 || last >= size)
      yield break;
   for (int i = first; i <= last; i++)
      yield return data[i];
}
```

In der **for**-Schleife werden nur noch die Kollektionselemente innerhalb des gewünschten Bereichs per **yield return** ausgeliefert. Wird die Methode mit ungeeigneten Bereichsgrenzen aufgerufen, liefert sie mit Hilfe der Anweisung **yield break** ein leeres **IEnumerable<T>** - Objekt.

Das folgende Programm demonstriert die Bereichsiteration ohne und mit Spezifikationsfehler. In der **foreach**-Schleife ist an das Kollektionsobjekt ein Methodenaufruf mit dem Namen des Bereichsiterators zu richten:[1]

[1] Auf die Bequemlichkeit eines Kollektionsinitialisierers (siehe Abschnitt 11.3.1) wird hier bewusst verzichtet, um die Anzahl der Vorgriffe im Manuskript nicht zu groß werden zu lassen.

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  var si = new SimpleSortedList<int>(5);`
`  si.Add(11); si.Add(2); si.Add(1); si.Add(4); si.Add(7);`
 	
`  Console.WriteLine("Standard-Iterator:");`
`  foreach (int i in si)`
`  Console.WriteLine(i);`
 	
`  Console.WriteLine("\nBereichs-Iterator:");`
`  foreach (int i in si.RangeIt(1, 4))`
`  Console.WriteLine(i);`
 	
`  Console.WriteLine("\nBereichüberschreitung:");`
`  foreach (int i in si.RangeIt(1, 14))`
`  Console.WriteLine(i);`
`  }`
`}`
	`Standard-Iterator:`
	`1`
	`2`
	`4`
	`7`
	`11`
	
	`Bereichs-Iterator:`
	`2`
	`4`
	`7`
	`11`
	
	`Bereichüberschreitung:`

Natürlich lassen sich mit benannten Iteratoren auch andere Aufgaben als die Bereichsauswahl realisieren.

Ein Typ darf benannte Iteratoren *unabhängig* von der Implementation der Schnittstelle **IEnumerable<out T>** anbieten.

Das gilt auch für die benannten Iteratoren mit Eigenschaftssyntax. Zur Demonstration erhält die Klasse `SimpleSortedList<T>` einen Abwärtsiterator namens `DownIt`:[1]

```
public IEnumerable<T> DownIt {
  get {
    for (int i = size - 1; i >= 0; i--)
      yield return data[i];
  }
}
```

In der **foreach**-Schleife ist an das Kollektionsobjekt der Eigenschaftsname anzuhängen:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  var si = new SimpleSortedList<int>(5);`
`  si.Add(11); si.Add(2); si.Add(1); si.Add(4); si.Add(7);`
 	
`  Console.WriteLine("\nAbwärts-Iterator:");`
`  foreach (int i in si.DownIt)`
`  Console.WriteLine(i);`
`  }`
`}`
	`Abwärts-Iterator:`
	`11`
	`7`
	`4`
	`2`
	`1`

Weitere Details zu Iteratoren finden sich z. B. in Griffith (2013, S. 175ff) und Mössenböck (2019, S. 169ff).

[1] Idee übernommen aus https://docs.microsoft.com/de-de/previous-versions/visualstudio/visual-studio-2010/ee5kxzk0(v=vs.100)

9.8 Übungsaufgaben zum Kapitel 9

1) Erstellen Sie eine Variante unserer Klasse `Bruch` (vgl. z. B. Abschnitt 5.1.3), welche die Schnittstelle **ICloneable** und die generische Schnittstellen **IComparable<T>** implementiert. Die vorhandene `Bruch`-Methode `Klone()`

```
public Bruch Klone() {
  return new Bruch(zaehler, nenner, etikett);
}
```

wird dabei *nicht* überflüssig, weil sie im Gegensatz zur **ICloneable**-Variante eine `Bruch`-Referenz abliefert und damit Typumwandlungen erspart.

2) Wie unterscheiden sich Interfaces von abstrakten Klassen?

3) In welchem Sinn kann die .NET-Schnittstellentechnik partiell die Mehrfachvererbung von C++ ersetzen?

4) Erweitern Sie in der Klasse `SimpleSortedList` den benannten Iterator `RangeIt()` so, dass über einen optionalen Richtungsparameter auch ein Bereich von Elementen in umgekehrter Reihenfolge angefordert werden kann.

10 Delegaten und Ereignisse

In diesem Kapitel ergänzen wir unser Wissen über das Common Type System (CTS) der .NET-Plattform. Wir lernen einen neuen Datentyp kennen und erfahren endlich, um welche speziellen Member es sich bei den schon oft erwähnten Ereignissen handelt:

- **Delegatenklassen**
  Ein Objekt einer Delegatenklasse enthält eine ein- oder mehrelementige Liste mit Methoden, die einen bestimmten Rückgabetyp und eine bestimmte Parameterliste besitzen. Über ein Delegatenobjekt können die dort registrierten Methoden aufgerufen werden.
- **Ereignisse**
  Möchte eine Klasse anderen Klassen die Möglichkeit geben, zu besonderen Gelegenheiten eine Botschaft, d.h. einen bestimmten Methodenaufruf, zu erhalten, dann bietet sie ein Ereignis an.

Ereignisse basieren wesentlich auf Delegatenobjekten, sind aber nicht die einzige Anwendung der Delegatenklassen.

10.1 Delegaten

Ein Objekt aus einer Delegatenklasse kann auf eine Methode mit einem bestimmten Rückgabetyp und mit einer bestimmten Parameterliste zeigen, und diese Methode kann über das Delegatenobjekt aufgerufen werden. Ein Delegatenobjekt kann z. B. als Aktualparameter verwendet werden beim Aufruf einer Methode, die einen Formalparameter mit Delegatentyp besitzt. Dann wird der „Rahmenmethode“ beim Aufruf eine „Hilfsmethode“ übergeben, die innerhalb der Rahmenmethode als Funktionsergänzung verwendet wird. Man kann hier von einer **Funktionsinjektion** sprechen. Weiterhin kann der Rahmenmethode per Delegatenobjekt die Möglichkeit zur **Kommunikation per Rückruf** gegeben werden.

Funktionsinjektion und Rückrufkommunikation sind auch beim Klassendesign von Nutzen und hier über Felder bzw. Mitgliedsobjekte mit Delegatentyp zu realisieren. Ein wichtiges Beispiel sind die Steuerelementklassen für WPF-Anwendungen (z. B. **Application**, **Button**), die sogenannte *Ereignisse* anbieten (z. B. **Exit**, **Click**), bei denen Instanzvariablen mit Delegatentyp eine wesentliche Rolle spielen (siehe Abschnitt 10.2). Ereignisinteressenten befördern eine Methode in das Delegatenobjekt zum Ereignis. Wenn das Ereignis eintritt, d.h. wenn das Delegatenobjekt zum Ereignis aufgerufen wird, dann wird die im Delegatenobjekt registrierte Methode ausgeführt.

Delegatentypen wirken zunächst etwas abstrakt, sind aber speziell im Kontext der ereignisorientierten Programmierung unverzichtbar. Es handelt sich um Klassen, deren Objekte auf Methoden mit einer bestimmten Parameterliste und mit einem bestimmten Rückgabetyp zeigen. Dazu enthalten die Delegatenobjekte eine (ein- oder mehrelementige) Aufrufliste mit kompatiblen Methoden. Diese Liste kann Instanz- und Klassenmethoden enthalten.

Bei einem Ereignis lassen sich auch mehrere Methoden registrieren, weil das zum Ereignis gehörende Delegatenobjekt auch eine mehrelementige Aufrufliste verwalten kann.

Beim Aufruf eines Delegatenobjekts werden alle Methoden in seiner Aufrufliste nacheinander ausgeführt, wobei die zuletzt ausgeführte Methode beim Rückgabewert und bei eventuell vorhandenen **out**- oder **ref**-Parametern dominiert. Somit bietet C# mit den Delegaten eine objektorientierte Variante der Funktions-Pointer aus anderen Programmiersprachen (z. B. C++, Pascal, Modula-2).

Vielleicht helfen die folgenden Begriffserläuterungen dabei, Missverständnisse zu vermeiden:

Bei der Definition eines **Delegatentyps** wird nach dem Schlüsselwort **delegate** angegeben:

- der Rückgabetyp von kompatiblen Methoden, die in die Aufrufliste eines Objekts mit diesem Delegatentyp aufgenommen werden können
- der Name des Delegatentyps
- die Parameterliste von kompatiblen Methoden

Beispiel:

```
delegate void DemoGate(int w);
```

Eine **die Delegatensignatur erfüllende Methode** muss einen passenden Rückgabetyp und eine passende Parameterliste haben, z. B.:

```
static void SagA(int w){
    for (int i = 1; i <= w; i++)
          Console.Write('A');
}
```

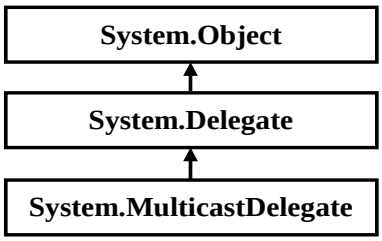
Ein **Delegatenobjekt** ...

- kann per **new**-Operator erstellt werden, indem an den Konstruktor eine kompatible Methode übergeben wird, z. B.:
  ```
  new DemoGate(SagA);
  ```
- enthält eine Aufrufliste mit kompatiblen Methoden

Eine **Delegatenvariable** besitzt ...

- einen Delegatentyp, z.B.
  ```
  DemoGate deleVar = new DemoGate(SagA);
  ```
- als Wert ...
  die Adresse eines Delegatenobjekts (z. B. vom Typ `DemoGate`)
  oder **null**

Alle Delegatentypen sind versiegelte Klassen und stammen implizit von der Klasse **MulticastDelegate** im Namensraum **System** ab:



Delegaten spielen eine unverzichtbare Rolle bei der Ereignisverarbeitung in GUI-Programmen, sind aber auch darüber hinaus von Interesse. Z. B. kann die Funktionalität einer Klasse modifiziert werden, indem einem Feld mit Delegatentyp eine kompatible Methode zugewiesen wird. Wenn die Klasse ihr Delegatenobjekt aufruft, kommt die „injizierte“ Methode zum Einsatz. Diese Option zur Verhaltenskonfiguration ist oft flexibler als die Definition von abgeleiteten Klassen. Allerdings wird die Vererbung, die ja eine von drei Kernmerkmalen der objektorientierten Programmierung ist

(vgl. Abschnitt 5.1.1), von den Delegaten nicht überflüssig gemacht. Die Definition einer abgeleiteten Klasse ist z. B. empfehlenswert, um eine Basisklasse um *zusätzliche* Felder und Methoden zu erweitern.[1]

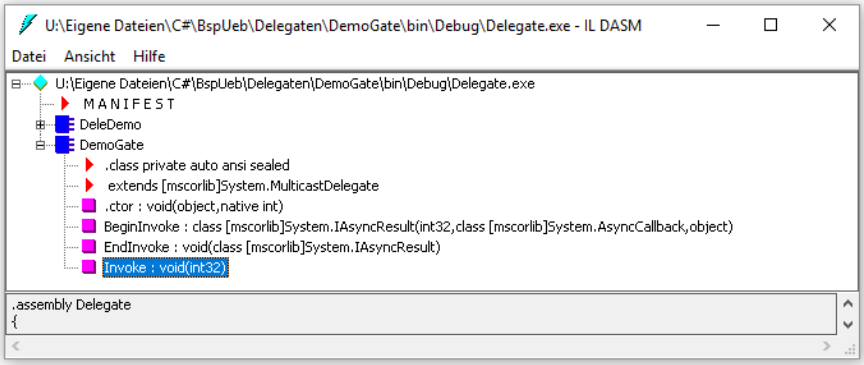
10.1.1 Delegatentypen definieren

Mit der folgenden Anweisung wird unter Verwendung des Schlüsselworts **delegate** der Delegatentyp `DemoGate` definiert:

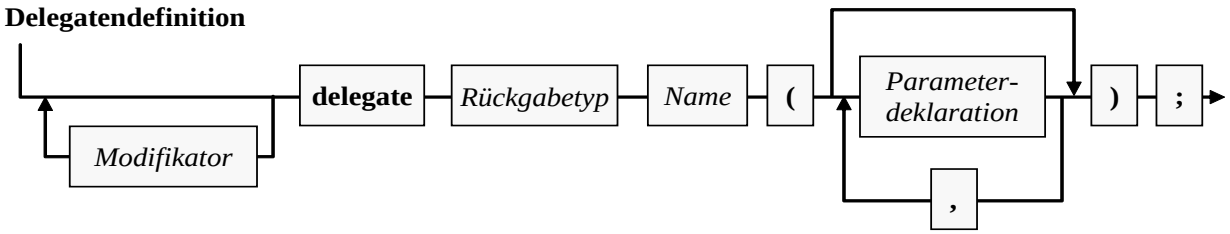
```
delegate void DemoGate(int w);
```

Über Objekte dieses Typs können Instanz- oder Klassenmethoden mit dem Rückgabetyp **void** und einem einzigen Parameter vom Typ **int** aufgerufen werden.

Eine Inspektion mit dem Hilfsprogramm **ILDasm** zeigt, dass aus der obigen Definition die Klasse `DemoGate` entstanden ist, die u. a. einen Konstruktor und die Instanzmethode **Invoke()** besitzt:



Es folgt ein leicht vereinfachtes Syntaxdiagramm zur Definition eines Delegatentyps:



Bei einem Top-Level - Delegaten sind wie bei anderen Top-Level - Typen die Zugriffsmodifikatoren **public** und **internal** erlaubt. Wird **public** *nicht* angegeben, ist der Delegat nur innerhalb seines Assemblies verwendbar (Sichtbarkeit **internal**). Für einen eingeschachtelten Delegaten sind dieselben Zugriffsmodifikatoren verfügbar wie für andere Member (siehe Abschnitt 5.11).

In der Literatur wird oft lax behauptet, ein Delegatenobjekt könne auf Methoden mit einer bestimmten *Signatur* zeigen. Wir wissen aus den Abschnitten 5.3.5 und 8.5, dass zwei Methoden genau dann dieselbe Signatur haben, wenn die Namen identisch sind, ggf. (bei generischen Methoden) die Typformalparameterlisten gleich lang sind, und außerdem die Parameterlisten (hinsichtlich Datentyp und Transfermodus aller Formalparameter) übereinstimmen, während die Rückgabetypen *keine* Rolle spielen. Diese Begriffsdefinition wird in der C# - Sprachspezifikation festgelegt (ECMA 2017, Abschnitt 8.6). Für einen Delegatentyp ist aber der Rückgabetyp essentiell, während der Name einer implementierenden Methode keine Rolle spielt. Außerdem muss die Parameterdeklaration einer Methode in einem strengen Sinn mit der Parameterdeklaration des Delegatentyps übereinstimmen:

---

[1] https://docs.microsoft.com/de-de/dotnet/csharp/tutorials/inheritance

- Die Anzahl der Parameter muss übereinstimmen.
- Die Datentypen müssen übereinstimmen.
- Die Transfermodi müssen übereinstimmen (Wert vs. Verweisparameter).
- Bei den Verweisparametern (siehe Abschnitt 5.3.1.3.2) müssen die Transferrichtungen (**in**, **out**, **ref**) übereinstimmen.

Wir werden im weiteren Verlauf die Anforderungen einer Delegatendefinition an kompatible Methoden als *Delegatensignatur* bezeichnen.

Die im Abschnitt 10.1.7 beschriebene Ko- und Kontravarianz für sogenannte *Methodengruppen* erlaubt eine Zuweisungsliberalität. Die zum Erstellen eines Delegatenobjekts verwendete Methode darf im Vergleich zur Delegatensignatur ...

- einen spezielleren Rückgabetyp
- und allgemeinere Parametertypen haben.

C# erlaubt auch eine *generische* Delegatendefinition. Ein wichtiges Beispiel ist der Delegatentyp **Predicate<in T>** aus dem BCL-Namensraum **System** mit einem kontravarianten Typformalparameter **T** (siehe Abschnitt 10.1.7):

public delegate bool Predicate<in T>(T *x*)

Ein Objekt der Konkretisierung **Predicate<String>** kann z. B. auf der folgenden Methode basieren:[1]

```
static bool ShortText(string t) {
   return (t ?? "").Length < 50;
}
```

10.1.2 Delegatenobjekte erzeugen und aufrufen

In diesem Abschnitt wird an einem einfachen Beispiel demonstriert, …

- wie der Aufruf einer Methode,
- die eine bestimmte Delegatensignatur besitzt,
- über ein Objekt eines Delegatentyps mit dieser Delegatensignatur erfolgen kann.

Die folgende Klasse `DeleDemo`

```
class DeleDemo {
   static void SagA(int w) {
      for (int i = 1; i <= w; i++)
         Console.Write('A');
      Console.WriteLine();
   }
   static void Main() {
      var deleVar = new DemoGate(SagA);
      deleVar(3);
   }
}
```

besitzt zwei statische Methoden:

- Die Methode `SagA()` schreibt eine per Parameter wählbare Anzahl von A's auf die Konsole. Sie erfüllt den Delegatentyp `DemoGate` (definiert im Abschnitt 10.1).
- In der **Main()** - Methode wird die lokale Referenzvariable `deleVar` vom Delegatentyp `DemoGate` deklariert und initialisiert. Über den implizit definierten `DemoGate`-Konstruktor wird ein Objekt des Delegatentyps erzeugt, das auf die Methode `SagA()` zeigt.

---

[1] Über den vermutlich noch ungewohnten Null-Sammeloperator **??** informiert der Abschnitt 8.3.

In der Aufrufliste des über die Referenzvariable `deleVar` ansprechbaren `DemoGate`-Objekts befindet sich ausschließlich die statische Methode `SagA()`. Ein Delegatenobjekt lässt sich wie eine Methode aufrufen. Im Beispiel führt der Aufruf

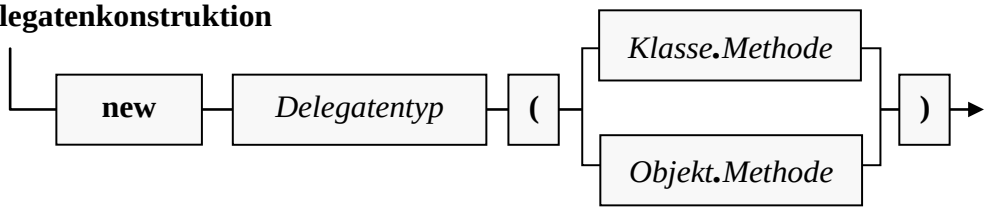
```
deleVar(3);
```

zur Ausgabe:

```
AAA
```

Natürlich unterstützen Delegatenobjekte nicht nur statische Methoden. Dem Standardkonstruktor einer Delegatenklasse übergibt man als einzigen Parameter den Namen einer statischen Methode (nötigenfalls mit vorangestelltem Klassennamen) *oder* den Namen einer Instanzmethode (nötigenfalls mit vorangestellter Objektreferenz):

Explizite Delegatenkonstruktion

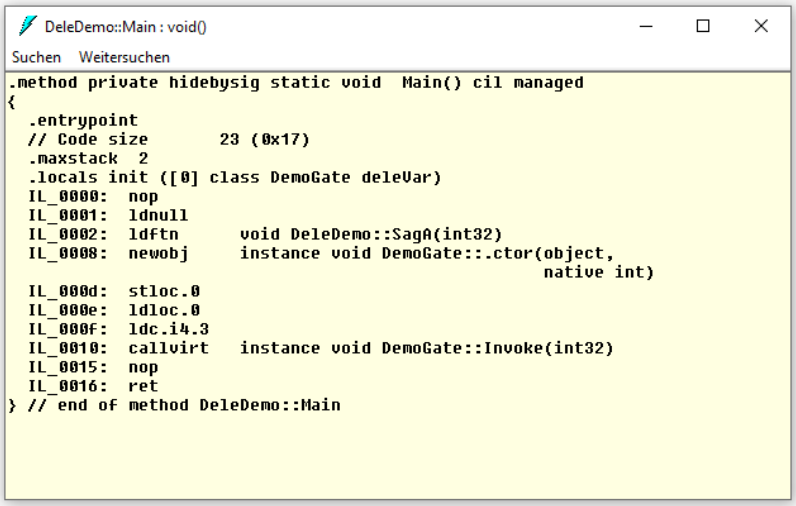


Dabei wird an den Methodennamen *keine* Parameterliste angehängt.

Eine Assembly-Inspektion mit dem Hilfsprogramm **ILDasm** zeigt, dass die Anweisungen

```
deleVar = new DemoGate(SagA);
deleVar(3);
```

in der `DeleDemo`-Methode **Main()**



folgendes bewirken

- Es wird ein `DemoGate`-Objekt erstellt (**newobj**).
- Dessen Adresse landet in der Variablen `deleVar` (**stloc.0**).
- Das Objekt wird beauftragt, die `DemoGate`-Methode **Invoke()** auszuführen (**callvirt**).

An Stelle der expliziten Delegatenobjektkreation

```
DemoGate deleVar = new DemoGate(SagA);
```

erlaubt der Compiler auch die Abkürzung:[1]

[1] Diese seit C# 2.0 verfügbare Vereinfachung wird als *Methodengruppen-Konvertierung* bezeichnet (engl.: *method group conversion*), siehe

```
DemoGate deleVar = SagA;
```

Eine Methode in die Aufrufliste eines Delegatenobjekts aufzunehmen und schlussendlich auf indirekte Weise aufzurufen, kann natürlich nicht der Sinn der Delegatentechnik sein. Viel praxisrelevanter ist die Möglichkeit, über einen Parameter oder ein Feld mit Delegatentyp eine Funktionalität in eine Methode oder in eine Klasse zu „injizieren“. Ein typisches Beispiel ist die folgende Überladung der Methode **Sort()** aus der generischen Klasse **List<T>** im Namensraum **System.Collections.Generic**:

public void Sort(Comparison<T> *comparison***)**

Sie erwartet als Parameter ein Delegatenobjekt, das den folgendermaßen definierten generischen Delegatentyp **Comparison<T>** erfüllt:

public delegate int Comparison<in T>(T *x***, T** *y***)**

Beim **Sort()** - Aufruf übergibt man eine Methode, welche für zwei **T**-Instanzen die Anordnung definiert und steuert so die Sortierung. Die Rahmenmethode ruft die injizierte Methode auf, um für zwei **T**-Instanzen die Anordnung zu ermitteln.

Durch die folgende Methode mit der Delegatensignatur von **Comparison<int>** werden **int**-Werte aufsteigend sortiert mit der Besonderheit, dass eine gerade Zahl jeder ungeraden Zahl unterlegen ist:[1]

```
static int EvenLower(int first, int second) {
   if (first % 2 == second % 2)
      return first.CompareTo(second);
   if (first % 2 == 0)
      return -1;
   return 1;
}
```

Wird die folgende Liste mit **int**-Werten[2]

```
var intList = new List<int>();
intList.Add(5); intList.Add(4); intList.Add(3);
intList.Add(2); intList.Add(1); intList.Add(6);
```

unter Verwendung von `EvenLower()` sortiert,

```
intList.Sort(EvenLower);
```

dann führt die Kontrollausgabe

```
foreach (int i in intList)
   Console.WriteLine(i);
```

zum Ergebnis:

```
2 4 6 1 3 5
```

Gerade wurde per Delegatenobjekt eine Verhaltenskonfiguration bzw. Funktionserweiterung in eine Rahmenmethode übertragen. Oft steht bei der Delegatenübergabe ein kommunikativer Zweck im Vordergrund: Durch einen Aufruf eines Delegatenobjekts kann die Rahmenmethode z. B. über die Beendigung einer Aufgabe informieren.

---

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/conversions#method-group-conversions

[1] Die elegante Formulierung der Methode stammt von Paul Frischknecht. Den mathematischen Hintergrund seiner Idee beschreibt er so: Sind die beiden Zahlen Elemente der gleichen Nebenklasse mod(2) (Nebenklasse im Sinne der Gruppentheorie), dann greift die übliche Vergleichsmethode. Andernfalls gelten die geraden Zahlen als "kleiner".

[2] Auf die Bequemlichkeit eines Kollektionsinitialisierers (siehe Abschnitt 11.3.1) wird hier bewusst verzichtet, um die Anzahl der Vorgriffe im Manuskript nicht zu groß werden zu lassen.

10.1.3 Delegatenobjekte kombinieren

Nach dem Motto „Wer *A* sagt, muss auch *B* sagen“ erweitern wir die Klasse `DeleDemo` aus dem Abschnitt 10.1.2 um die statische Methode `SagB()`:

```
static void SagB(int w) {
  for (int i = 1; i <= w; i++)
    Console.Write('B');
  Console.WriteLine();
}
```

In der Methode **Main()** ergänzen wir die Anweisung:

```
deleVar += new DemoGate(SagB);
```

Es entsteht zunächst ein weiteres `DemoGate`-Objekt mit der statischen Methode `SagB()` in seiner einelementigen Aufrufliste. Dieses Objekt wird anschließend per „+=“ - Operator mit dem von `deleVar` referenzierten Objekt kombiniert, wobei ein weiteres `DemoGate`-Delegatenobjekt mit einer *zweielementigen* Aufrufliste entsteht, dessen Adresse schließlich in der Referenzvariablen `deleVar` landet. Die beiden Delegatenobjekte mit einelementiger Aufrufliste werden zu obsoletem Müll (vgl. ECMA 2017, S. 385). Delegatenobjekte sind ebenso unveränderlich wie z. B. die Objekte der Klasse **String** (vgl. Abschnitt 6.3.1.1).

Beim Aufruf des neuen Delegatenobjektes werden nacheinander *zwei* Methoden ausgeführt:

```
AAA
BBB
```

Über den „-=“ - Operator kann man ein Delegatenobjekt mit *verkürzter* Aufrufliste erzeugen, z. B.:

```
deleVar -= SagB;
```

Beim kompletten Entleeren der Aufrufliste entsteht aber kein leeres Delegatenobjekt, sondern die Delegatenvariable erhält den Wert **null**. Über eine Delegatenvariable lassen sich also genau dann registrierte Methoden starten, wenn ihr Wert von **null** verschieden ist.

Neben den Aktualisierungsoperatoren += und -= kann man auch den Additions- und den Subtraktionsoperator verwenden, z. B.:

```
deleVar = deleVar + new DemoGate(SagB);
```

Besitzt ein Delegatenobjekt eine *mehrelementige* Aufrufliste, spricht man von einem *Multicastdelegaten*. Bei einem Aufruf des Delegatenobjekts werden alle Methoden in seiner Aufrufliste nacheinander ausgeführt, wobei die zuletzt aufgerufene Methode beim Rückgabewert und bei eventuell vorhandenen **out**- oder **ref**-Parametern dominiert.

10.1.4 Delegaten versus Schnittstellen

Die Injektion von Funktionalität in eine Klasse oder eine Methode kann oft alternativ über einen Delegatentyp oder durch eine Schnittstelle erreicht werden. In der generischen Klasse **List<T>** im Namensraum **System.Collections.Generic** finden sich z. B. zwei Überladungen der Methode **Sort()**, die per Parameterobjekt ein frei konfigurierbares Sortierkriterium unterstützen. Durch das Parameterobjekt wird letztlich eine Methode beigesteuert, die für zwei Instanzen vom Typ **T** per **int**-Rückgabe die Anordnung festlegt. Die beiden Überladungen sehen sehr ähnlich aus:

- **public void Sort(Comparison<T>** *comparison***)**
  Diese aus einem Beispiel im Abschnitt 10.1.2 bekannte Überladung erwartet als Parameter ein Objekt vom generischen Delegatentyp **Comparison<T>**:

  public delegate int Comparison<in T>(T *x***, T** *y***)**

  Der Typformalparameter **T** ist über das Schlüsselwort **in** als kontravariant definiert (siehe Abschnitt 10.1.7). Ein Objekt vom Typ **Comparison<T>** zeigt auf eine Methode, welche zwei Argumente vom Typ **T** besitzt und eine Rückgabe vom Typ **int** liefert. Wird **T** durch **String** konkretisiert, taugt z. B. die folgende Methode:

  ```
  public int StringComp(String x, String y) { ... }
  ```

- **public void Sort(IComparer<T>** *comparer***)**
  Diese Überladung erwartet als Parameter ein Objekt aus einer Klasse, welche die Schnittstelle **IComparer<T>** implementiert:

  public interface IComparer<in T>

  Auch der Typformalparameter **T** der generischen Schnittstelle **IComparer<T>** ist über das Schlüsselwort **in** als kontravariant definiert (siehe Abschnitt 9.3.2). Wird **T** durch **String** konkretisiert, muss eine Methode mit dem folgenden Definitionskopf vorhanden sein:

  public int Compare(String *x***, String** *y***)**

Argumente für ein Design mit Delegatentyp:

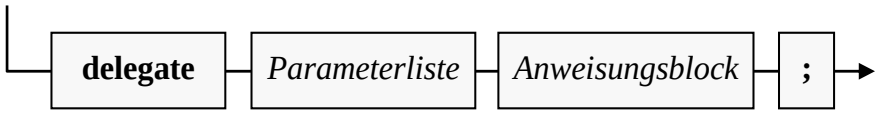
- Eine Klasse kann *mehrere* Methoden enthalten, die *denselben* Delegatentyp erfüllen, weil der Methodenname frei wählbar ist. Demgegenüber kann eine Klasse eine Schnittstellenmethode nur einmal implementieren, weil der Name fest vorgegeben ist.
- Ein Delegatenobjekt lässt sich auch auf Basis einer statischen Methode erstellen, während eine Schnittstelle grundsätzlich Instanzmethoden verlangt.
- Über eine anonyme Methode (siehe Abschnitt 10.1.5) oder einen Lambda-Ausdruck (siehe Abschnitt 10.1.6) lässt sich ein Delegatenobjekt syntaktisch elegant realisieren, während zum Implementieren einer Schnittstelle eine explizite Typdefinition erforderlich ist.

Bei einem Design mit Schnittstelle kann man allerdings den implementierenden Klassen beliebig viele Methoden diktieren, sodass deren Objekte über mehrere garantierte Kompetenzen verfügen. Über einen Delegatentyp lässt sich demgegenüber nur *eine* Methode vorschreiben.

10.1.5 Delegatenobjekte durch anonyme Methoden erstellen

Statt beim Erzeugen eines Delegatenobjekts den Namen einer vorhandenen, andernorts definierten Methode zu übergeben, kann man seit C# 2.0 auch einen Anweisungsblock erstellen, dem das Schlüsselwort **delegate** samt Delegatentyp-konformer Parameterliste vorangestellt wird:

Delegatenobjekt aus einer anonymen Methode



Dies wird in der folgenden Variante des Beispiels aus dem Abschnitt 10.1.2 demonstriert:

Quellcode	Ausgabe

`using System;`
 	
`delegate void DemoGate(int w);`
 	
`class AnoMeth {`
`  static void Main() {`
`  DemoGate deleVar =`
`  delegate (int w) {`
`  for (int i = 1; i <= w; i++)`
`  Console.Write('A');`
`  Console.WriteLine();`
`  };`
`  deleVar(3);`
`  }`
`}`
	`AAA`

Man kann die gewünschte Funktionalität vor Ort realisieren und muss keine Kreativität in einen Methodennamen investieren.

Bei entsprechender Definition des zu erfüllenden Delegatentyps können (und müssen) anonyme Methoden selbstverständlich auch einen Rückgabewert liefern, z. B.:

Quellcode	Ausgabe

`using System;`
 	
`delegate int DemoGate(int w);`
 	
`class AnoMeth {`
`  static void Main() {`
`  DemoGate deleVar = delegate (int w) {return w;};`
`  Console.WriteLine(deleVar(3));`
`  }`
`}`
	`3`

Ein Nachteil anonymer Methoden besteht darin, dass sie nicht an anderen Stellen benutzt werden können. Das Visual Studio empfiehlt daher, statt einer anonymen Methode eine *lokale* Methode zu verwenden (siehe Abschnitt 5.3.1.5).

Eine anonyme Methode darf auf lokale Variablen der umgebenden Methode sowie auf (statische) Felder der Umgebung zugreifen. Im folgenden Beispiel wird die lokale Variable `c` der Methode **Main()** verwendet:

Quellcode	Ausgabe

`using System;`
 	
`delegate void DemoGate(int w);`
 	
`class AnoMeth {`
`  static void Main() {`
`  char c = 'A';`
`  DemoGate deleVar =`
`  delegate (int w) {`
`  for (int i = 1; i <= w; i++)`
`  Console.Write(c);`
`  Console.WriteLine();`
`  c++;`
`  };`
`  Console.WriteLine("c vor dem Delegatenaufr.:  " + c);`
`  deleVar(3);`
`  Console.WriteLine("c nach dem Delegatenaufr.: " + c);`
`  }`
`}`
	`c vor dem Delegatenaufr.:  A`
	`AAA`
	`c nach dem Delegatenaufr.: B`

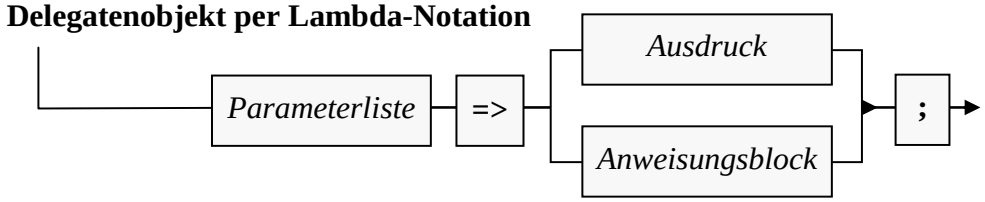
Wie das Beispiel demonstriert, kann in einer anonymen Methode eine Variable aus der Umgebung auch *verändert* werden.

Eine Methode kann ein von ihr erzeugtes Delegatenobjekt z. B. in einer Instanzvariablen ablegen, und oft ist die erzeugende Methode schon beendet, wenn das Delegatenobjekt aufgerufen wird. Damit der Delegatenaufruf auch dann noch klappt, wenn die generierende Methode beendet ist, und ihre lokalen Variablen längst vom Stack verschwunden sind, erzeugt der Compiler zur Aufbewahrung der eingefangenen Variablen ein Objekt auf dem Heap (siehe Griffiths 2013, S. 316ff).

10.1.6 Delegatenobjekte per Lambda-Notation erstellen

Die mit C# 3.0 zur Unterstützung der LINQ-Technik eingeführte *Lambda-Notation* macht es möglich, bei der Realisation von Delegatenobjekten im Vergleich zur Verwendung von anonymen Methoden (siehe Abschnitt 10.1.5) etwas Schreibaufwand zu sparen. Mittlerweile lassen sich per Lambda-Notation auch benannte Methoden sowie Eigenschaften und Indexer erstellen (siehe Abschnitt 5.7.3, über die sogenannten *expression bodied members*).

Zur Definition eines Delegatenobjekts per Lambda-Notation kann ein einzelner Ausdruck (Ausdrucks-Lambda) oder ein Anweisungsblock (Anweisungs-Lambda) verwendet werden:



Für den generischen Delegatentyp **Predicate<in T>** aus dem BCL-Namensraum **System** mit einem kontravarianten Typformalparameter **T** (siehe Abschnitt 10.1.7) und einer booleschen Rückgabe

public delegate bool Predicate<in T>(T *x*)

sind anschließend äquivalente Realisationen über eine anonyme Methode

```
Predicate<int> even = delegate (int i) { return i % 2 == 0; };
```

und über einen Lambda-Ausdruck zu sehen:

```
Predicate<int> even = (int i) => i % 2 == 0;
```

Bei der Parameterliste eines Lambda-Ausdrucks besteht einige Flexibilität:

- Man kann auf die Angabe der Parameter*typen* verzichten, weil sich diese aus dem zu erfüllenden Delegatentyp zwingend ergeben, z. B.:
  ```
  Predicate<int> even = (i) => i % 2 == 0;
  ```
- Bei einem einzelnen, implizit typisierten Parameter kann man die runden Klammern weglassen, z. B.:
  ```
  Predicate<int> even = i => i % 2 == 0;
  ```

Wird ein Delegatentyp mit einem von **void** verschiedenen Rückgabetyp durch ein Anweisungs-Lambda realisiert, dann muss der Rückgabewert per **return**-Anweisung geliefert werden, z. B.:

```
Predicate<String> evenLen = s => {
  int sl = s.Length;
  return sl % 2 == 0;
};
```

Im Beispiel aus dem Abschnitt 10.1.5 ergibt sich im Vergleich zur Verwendung einer anonymen Methode kein großer Einspareffekt:

Quellcode	Ausgabe

`using System;`
`delegate void DemoGate(int w);`
`class LambdaDelegate {`
`  static char c = 'A';`
`  static void Main() {`
`  DemoGate deleVar = w => {`
`  for (int i = 1; i <= w; i++)`
`  Console.Write(c);`
`  Console.WriteLine();`
`  };`
`  deleVar(3);`
`  }`
`}`
	`AAA`

Man spart das Schlüsselwort **delegate** und kann die Fähigkeiten des Compilers zur Typinferenz nutzen.

Außerdem wird im Beispiel demonstriert, dass ein Ausdrucks- oder Anweisungs-Lambda (wie eine anonyme Methode) auf Variablen in der Umgebung (hier: auf ein statisches Feld) zugreifen kann.

10.1.7 Generische Delegaten, Ko- und Kontravarianz

Neben generischen Klassen, Strukturen, Schnittstellen und Methoden (siehe Kapitel 8) unterstützt C# auch generische Delegaten. Der Type **Comparison<T>** aus dem BCL-Namensraum **System**

```
public delegate int Comparison<in T>(T x, T y);
```

wurde bereits im Abschnitt 10.1.4 vorgestellt. Er kommt in einer **Sort()** - Überladung der generischen Kollektionsklasse **List<T>** (vgl. Abschnitt 8.1) als Parameterdatentyp zum Einsatz. Über ein **Comparison<String>** - Objekt, das auf eine geeignet konstruierte **String**-Vergleichsmethode zeigt, wird im folgenden Beispiel dafür gesorgt, dass in einer sortierten Namensliste „Karl“ stets der Größte ist:[1]

[1] Auf die Bequemlichkeit eines Kollektionsinitialisierers (siehe Abschnitt 11.3.1) wird hier bewusst verzichtet, um die Anzahl der Vorgriffe im Manuskript nicht zu groß werden zu lassen.

Quellcode	Ausgabe

```
using System;
using System.Collections.Generic;

class Prog {
   static int ComKarlison(string a, string b) {
      switch (a.CompareTo(b)) {
         case -1: return a.Equals("Karl") ? 1 : -1;
         case 1: return b.Equals("Karl") ? -1 : 1;
         default: return 0;
      }
   }

   static void Main() {
      var li = new List<string>();
      li.Add("Fritz"); li.Add("Karl");
      li.Add("Anita"); li.Add("Theo");
      li.Sort(ComKarlison);
      foreach (var s in li)
         Console.WriteLine(s);
   }
}
```

```
Anita
Fritz
Theo
Karl
```

Weil im generischen Delegatentyp **Comparison<in T>** aus der BCL der Typformalparameter über das Schlüsselwort **in** als kontravariant definiert ist, kann unter den folgenden, schon mehrfach in Beispielen verwendeten Voraussetzungen

```
public class Figur { . . . }
public class Kreis : Figur { . . .}
```

einer Delegatenvariablen vom Typ **Comparison**<Kreis> ein Delegatenobjekt vom Typ **Comparison**<Figur> zugewiesen werden. Im folgenden Beispielprogramm wird der BCL-Delegat **Comparison**<**T**> durch die Eigenkreation Vergleich<T> ersetzt, damit die Notwendigkeit der **in**-Deklaration des Typformalparameters getestet werden kann:

```
using System;

public delegate int Vergleich<in T>(T x, T y);

class Prog {
    static int FigurenVergleich(Figur f1, Figur f2) {
        if (f1.X < f2.X)
            return -1;
        if (f1.X > f2.X)
            return 1;
        return 0;
    }

    static Kreis KleinsterKreis(Kreis k1, Kreis k2, Vergleich<Kreis> vk) {
        return vk(k1, k2) <= 0 ? k1 : k2;
    }

    static void Main() {
        var k1 = new Kreis(2, 2, 1);
        var k2 = new Kreis(1, 4, 1);
        Vergleich<Figur> vf = FigurenVergleich;
        Console.WriteLine(KleinsterKreis(k1, k2, vf).X);
    }
}
```

Obwohl die Methode `KleinsterKreis()` als dritten Parameter ein Delegatenobjekt vom Typ `Vergleich<Kreis>` deklariert, wird beim Aufruf der Methode auch ein Delegatenobjekt vom Typ `Vergleich<Figur>` akzeptiert.

Ist der Typformalparameter von `Vergleich<T>` *nicht* als kontravariant definiert,

```
public delegate int Vergleich<T>(T x, T y);
```

dann verbietet der Compiler hingegen die Verwendung eines `Vergleich<Figur>` - Delegaten an Stelle eines `Vergleich<Kreis>` - Delegaten:

```
static void Main() {
    var k1 = new Kreis(2, 2, 1);
    var k2 = new Kreis(1, 4, 1);
    Vergleich<Figur> vf = FigurenVergleich;
    Console.WriteLine(KleinsterKreis(k1, k2, vf).X);
}
```

(Lokale Variable) Vergleich<Figur> vf

CS1503: Argument "3": Konvertierung von "Vergleich<Figur>" in "Vergleich<Kreis>" nicht möglich.

Mögliche Korrekturen anzeigen (Alt+Eingabe oder Strg+.)

Weil im Delegatentyp **Func<out T>** aus dem BCL-Namensraum **System** der Typformalparameter über das Schlüsselwort **out** als kovariant definiert ist,

```
public delegate TResult Func<out TResult>();
```

kann in der `Figur-Kreis` - Konstellation auch ein Objekt vom Delegatentyp **Func**`<Kreis>` verwendet werden (z. B. als Aktualparameter), wenn laut Deklaration ein Objekt vom Delegatentyp **Func**`<Figur>` erwartet wird. Im folgenden Beispielprogramm wird der BCL-Delegat **Func<T>** durch die Eigenkreation `Funk<T>` ersetzt, damit die Notwendigkeit der **out**-Deklaration des Typformalparameters getestet werden kann. Die Methode `Where()` erwartet als Parameter einen `Figur`-Lieferanten, gibt sich aber auch mit einem `Kreis`-Lieferanten zufrieden:

```
using System;

public delegate T Funk<out T>();

class Prog {
    static void Where(Funk<Figur> figur) {
        Console.WriteLine($"({figur().X}, {figur().Y})");
    }

    static void Main() {
        Funk<Kreis> fk = delegate () {
            Random zzg = new Random();
            return new Kreis(zzg.Next(5), zzg.Next(5), zzg.Next(3));
        };
        Where(fk);
    }
}
```

Ist der Typformalparameter von `Funk<T>` *nicht* als kovariant definiert,

```
public delegate T Funk<T>();
```

dann verbietet der Compiler hingegen die Verwendung eines `Funk<Kreis>` - Delegaten an Stelle eines `Funk<Figur>` - Delegaten:

```
static void Main() {
    Funk<Kreis> fk = delegate () {
        Random zzg = new Random();
        return new Kreis(zzg.Next(5), zzg.Next(5), zzg.Next(3));
    };
    Where(fk);
}
```

(Lokale Variable) Funk<Kreis> fk

CS1503: Argument "1": Konvertierung von "Funk<Kreis>" in "Funk<Figur>" nicht möglich.

Mögliche Korrekturen anzeigen (Alt+Eingabe oder Strg+.)

Eine als Ko- bzw. Kontravarianz zu beschreibende Zuweisungsliberalität besteht auch für *nichtgenerische* Delegatentypen und die mit .NET 2.0 eingeführten **Methodengruppen**. Einer Variablen vom Typ

```
public delegate Object NonGenericDelegate(String arg);
```

darf z. B. die folgende Methode mit einem allgemeineren Argumenttyp und einem spezielleren Rückgabetyp

```
static String AORS(Object arg) { return null; }
```

zugewiesen werden:[1]

```
NonGenericDelegate nGenDel = AORS;
```

10.2 Ereignisse

Möchte eine Klasse anderen Klassen die Möglichkeit geben, zu besonderen Gelegenheiten eine Botschaft, d.h. einen bestimmten Methodenaufruf, zu erhalten, dann bietet sie ein *Ereignis* (engl.: *event*) an.[2] Bei einem Ereignis ist eine private Delegatenvariable (vgl. Abschnitt 10.1) und ein Paar von öffentlichen Zugriffmethoden zum (De-)Registrieren von Methoden im Spiel. Interessenten können für das Ereignis eine Behandlungsmethode registrieren lassen, die zu bestimmten Gelegenheiten aufgerufen werden soll, z. B.:

- Ein Befehlsschalter aus der Klasse **Button** in der Bedienoberfläche eines WPF-Programms informiert über sein **Click**-Ereignis registrierte Interessenten darüber, dass er vom Benutzer ausgelöst worden ist.
- Ein Objekt vom Typ **ObservableCollection<T>** informiert über sein **Changed**-Ereignis über eine Änderung seiner Daten. Eine registrierte Behandlungsmethode kann z. B. mit einer Aktualisierung der Bedienoberfläche des Programms reagieren.

Wir lernen mit dem Ereignis ein neues Klassenmitglied kennen, das wie die anderen Klassenmitglieder Instanz- oder Klassen-bezogen sein kann.

Ereignisse stellen also ein wichtiges Kommunikationsmittel dar. Sie ermöglichen es einer Klasse oder einem Objekt, andere Akteure darüber zu informieren, dass etwas Bestimmtes passiert ist.

---

1 https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/concepts/covariance-contravariance/

2 Grundsätzlich können auch *Strukturen* Ereignisse anbieten, was aber in der Praxis so gut wie nie geschieht und außerdem durch Komplikationen mit der Thread-Sicherheit belastet ist, siehe http://csharpindepth.com/Articles/Chapter2/Events.aspx

10.2.1 Technische Realisation von Ereignissen

Obwohl Ereignisse in der Regel als **public** deklariert werden (siehe Abschnitt 10.2.3), resultiert in der veröffentlichenden Klasse stets eine *private* Delegatenvariable. Hinzu kommen jedoch explizit oder implizit definierte *öffentliche* Zugriffsmethoden zum Erweitern oder Verkürzen der Aufrufliste durch fremde Klassen. Diese sind über die Aktualisierungsoperatoren mit dem Ereignisnamen als linkem Argument zu verwenden:

- `+=` nimmt eine Behandlungsmethode in die Aufrufliste des zum Ereignis gehörenden Delegatenobjekts auf
- `-=` entfernt eine Behandlungsmethode aus der Aufrufliste des zum Ereignis gehörenden Delegatenobjekts

Im folgenden Beispiel wird eine Behandlungsmethode bei dem vom Objekt `surpriseButton` angebotenen Ereignis `Seven` registriert:

```
surpriseButton.Seven += surpriseButton_Seven;
```

Das öffentlich zugängliche Ereignis, das eigentlich ein Paar von Zugriffsmethoden ist, und die private Delegatenvariable stehen zueinander in derselben Beziehung wie eine Eigenschaft und die zugrunde liegende Instanzvariable (vgl. Abschnitt 5.5). Dementsprechend ähnelt die Syntax zur Definition eines Ereignisses stark einer Eigenschaftsdefinition, wobei die Schlüsselwörter **get** und **set** zu ersetzen sind durch **add** und **remove**. Wir betrachten als Beispiel das von der WPF-Klasse **Application** im Namensraum **System.Windows** definierte Ereignis **Exit**, das anderen Klassen die Möglichkeit bietet, sich über das bevorstehende (und unabwendbare) Programmende informieren zu lassen:[1]

```
/// <summary>
/// The Exit event is fired when an application is shutting down.
/// This event is raised by the OnExit method.
/// </summary>
public event ExitEventHandler Exit {
   add {
      VerifyAccess();
      Events.AddHandler(EVENT_EXIT, value);
   }
   remove {
      VerifyAccess();
      Events.RemoveHandler(EVENT_EXIT, value);
   }
}
```

[1] Der Quellcode zur Klasse **Application** in der BCL zu .NET 5.0 kann über die Webseite https://source.dot.net/ inspiziert werden. **Events** ist eine private Eigenschaft der Klasse **Application**, die im Lesezugriff ein Objekt der Klasse **EventHandlerList** liefert. Dieses Kollektionsobjekt verwaltet eine Liste von (**Object**, **Delegate**) - Paaren und beherrscht dazu die Methoden **AddHandler()** und **RemoveHandler()**, welche die Aufrufliste zu dem im ersten Parameter angegebenen Delegatenobjekt erweitern bzw. reduzieren. Die für das Feuern des **Exit**-Ereignisses zuständige **Application**-Methode **OnExit()** holt sich das zum Ereignis gehörende Delegatenobjekt aus der Kollektion und ruft dann im Sinne von Abschnitt 10.1.2 das Delegatenobjekt auf:

```
protected virtual void OnExit(ExitEventArgs e) {
  VerifyAccess();
  ExitEventHandler handler = (ExitEventHandler)Events[EVENT_EXIT];
  if (handler != null) {
     handler(this, e);
  }
}
```

Analog zu den automatisch implementierten Eigenschaften (siehe Abschnitt 5.5.2) kann man auch bei der Ereignisdefinition etliche Routinearbeit dem Compiler überlassen, wie das Ereignis **Activated** aus der Klasse **Application** zeigt:

```
public event EventHandler Activated;
```

Diese Ereignisdefinition sieht aus wie die Deklaration einer Delegatenvariablen, wobei zusätzlich das Schlüsselwort **event** anzugeben ist. Allerdings ist es mit der Deklaration allein nicht getan, weil ein sinnvolles Ereignis auch irgendwann ausgelöst werden muss. Dies geschieht meist in einer Methode, deren Namen mit *On* startet und dann den Ereignisnamen übernimmt, z. B.:

```
protected virtual void OnActivated(EventArgs e) {
   VerifyAccess();
   if (Activated != null) {
      Activated(this, e);
   }
}
```

Weil die zu einem Ereignis gehörende Delegatenvariable *privat* ist, können auch abgeleitete Klassen das Ereignis *nicht* über die Delegatenvariable auslösen. Über die gerade für das **Application**-Ereignis **Activated** vorgestellte virtuelle (also überschreibbare) Methode **OnActivated()** mit dem Zugriffsschutz **protected** können abgeleitete Klassen jedoch die volle Kontrolle über die Ereignisentstehung übernehmen (siehe Abschnitt 10.2.3).

Weil Ereignisse (analog zu Eigenschaften) als Paare von Methoden aufgefasst werden können (siehe obiges Beispiel **Exit**), sind in ihrer Definition die folgenden Modifikatoren erlaubt:

- Zugriffsmodifikatoren
  Es sind dieselben Zugriffsmodifikatoren erlaubt wie bei anderen Klassenmitgliedern (siehe Abschnitt 5.11).
- **static**
- **abstract**
  Das Ereignis wird ohne Zugriffsmethoden erstellt, sodass es von abgeleiteten (nicht-abstrakten) Klassen überschrieben werden muss.
- **virtual**
  Das Ereignis kann in einer abgeleiteten Klasse überschrieben werden.
- **new**, **override**
  Eine abgeleitete Klasse ersetzt oder überschreibt ein Ereignis. Für ein überschreibendes Ereignis darf die Verfügbarkeit im Vergleich zum überschriebenen Ereignis nicht geändert werden.
- **sealed**
  Für ein Ereignis kann per **sealed**-Modifikator verhindert werden, dass es in abgeleiteten Klassen überschrieben wird. Warum das Schlüsselwort **sealed** nur in Kombination mit dem Schlüsselwort **override** erforderlich und erlaubt ist, wurde im Abschnitt 7.8 im Zusammenhang mit Methoden erläutert.

10.2.2 Behandlungsmethoden registrieren

Um auf ein Ereignis einer .NET - Klasse reagieren zu können, …

- implementiert man eine Methode, die mit dem Delegatentyp des Ereignisses kompatibel ist (siehe Abschnitt 10.1.7 zur Zuweisungskompatibilität bei Delegaten),
- erzeugt man ein Delegatenobjekt, das auf diese Methode zeigt,
- weist man dem Ereignis dieses Delegatenobjekt per „+=“ - Operator zu. Dabei wird die Aufrufliste des zum Ereignis gehörenden Delegatenobjekts erweitert (siehe Abschnitt 10.1.3).

Wie Sie seit dem Abschnitt 10.1.2 wissen, kann zum Erstellen eines Delegatenobjekts unter Verwendung einer vorhandenen Methode die explizite Schreibweise mit **new**-Operator wie im folgenden Beispiel

```
surpriseButton.Seven += new SevenEventHandler(this.surpriseButton_Seven);
```

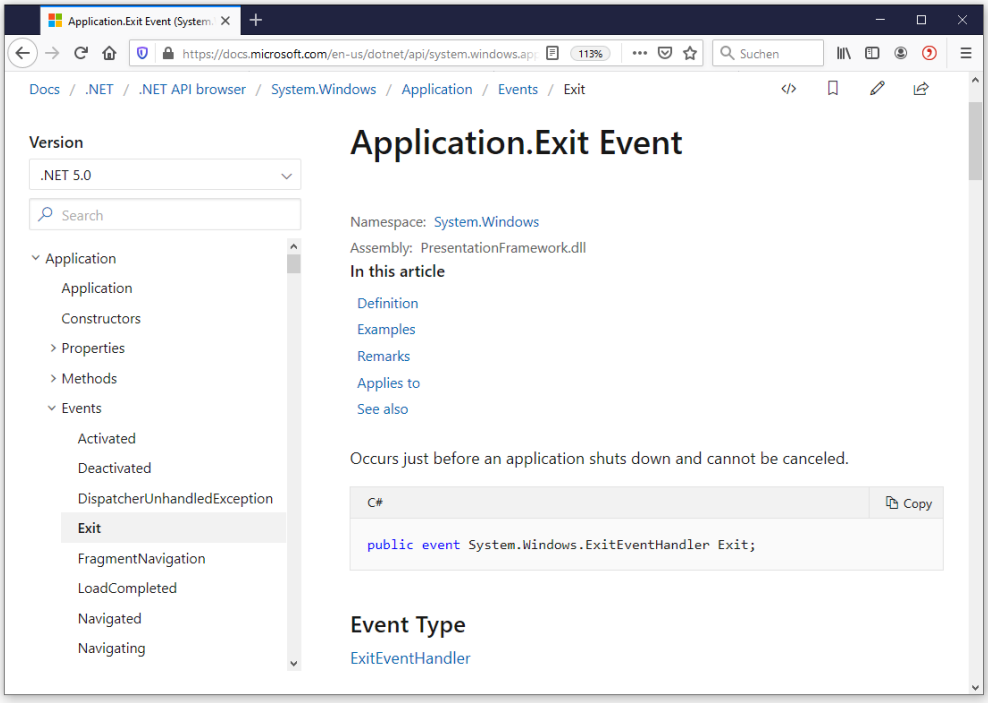
oder die Kurzform

```
surpriseButton.Seven += surpriseButton_Seven;
```

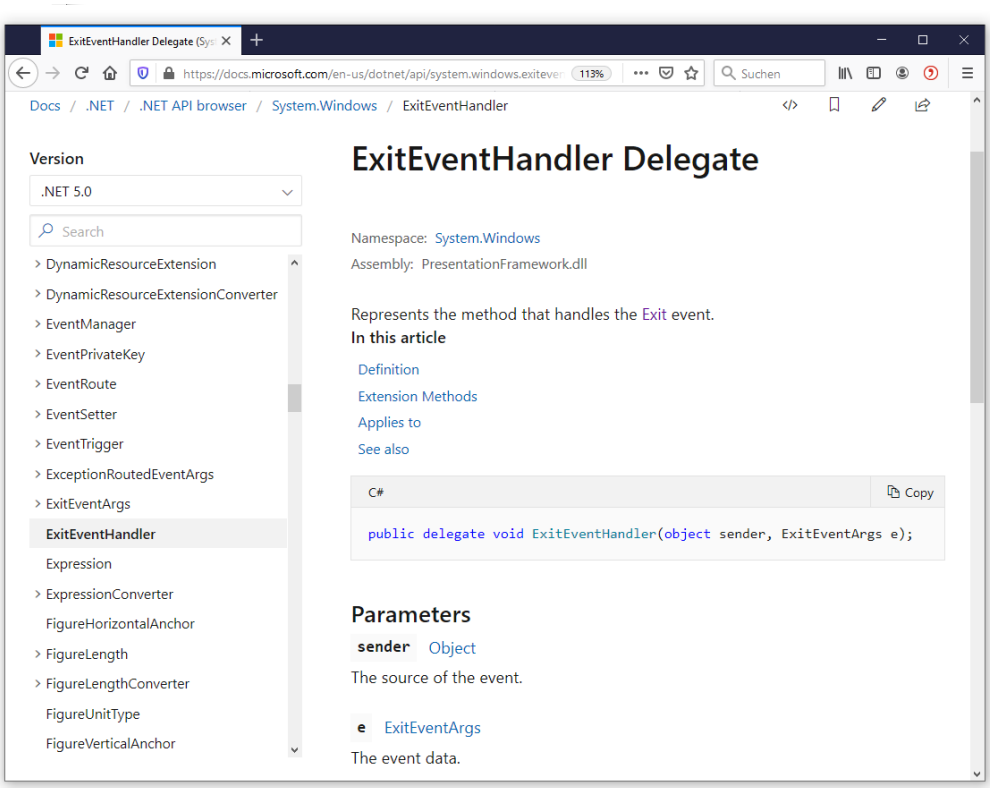
verwendet werden, sodass vom zweiten Punkt aus der obigen Auflistung nichts zu sehen ist.

In diesem Abschnitt behandeln wir exemplarisch das Ereignis **Exit**, das anlässlich der unmittelbar bevorstehenden (und nicht mehr zu stoppenden) Beendigung einer WPF-Anwendung auftritt. Es wird von einem die Anwendung repräsentierenden Objekt ausgelöst, das zur Klasse **Application** im Namensraum **System.Windows** oder zu einer abgeleiteten Klasse gehört (siehe Abschnitt 12.2.3).

Zu welchem Delegatentyp eine Ereignisbehandlungsmethode kompatibel sein muss, erfährt man in der BCL-Dokumentation. Beim Ereignis **Exit** der Klasse **Application** handelt es sich um den Typ **ExitEventHandler**:



In der Dokumentation zum Delegatentyp **ExitEventHandler** ist zu erfahren, welchen Rückgabetyp und welche Parameterliste eine kompatible Methode benötigt, z. B.:



Die **ExitEventHandler**-Delegatensignatur verlangt von kompatiblen Behandlungsmethoden zwei Parameter:

- **Object**-Parameter **sender**
  Die aufgerufenen Behandlungsmethoden erhalten über diesen Parameter eine Referenz zur Ereignisquelle. Man kann eine Behandlungsmethode bei *mehreren* Ereignisquellen anmelden, z. B. bei mehreren Befehlsschaltern. Weil der Methode beim Aufruf die Ereignisquelle im Parameter **sender** mitgeteilt wird, kann sie situationsgerecht reagieren.
- **ExitEventArgs**-Parameter **e**
  Behandlungsmethoden erhalten im Allgemeinen über den zweiten Parameter nähere Informationen zum Ereignis. Dabei wird ein Objekt aus der Klasse **System.EventArgs** oder aus einer abgeleiteten Klasse verwendet.

Im frei wählbaren Namen einer Behandlungsmethode nennt man in der Regel die Ereignisquelle (z. B. die Klasse **Application**) und das Ereignis (im Beispiel: **Exit**). Das Visual Studio setzt bei automatisch erstellten Behandlungsmethoden zwischen die beiden Namensbestandteile einen Unterstrich.

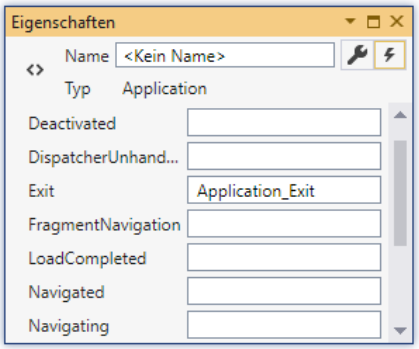
Wie Sie bereits aus eigener Erfahrung wissen (vgl. z. B. Abschnitt 5.12.8), ist bei der praktischen Arbeit mit dem Visual Studio das Erstellen und Registrieren einer Ereignisbehandlungsmethode eine einfache Angelegenheit. Um ein rudimentäres Beispielprogramm samt **Exit**-Ereignisbehandlung semiautomatisch zu erstellen, kann man so vorgehen:

- Man erstellt eine neue **WPF-App (.NET Framework)**. Alternativ kann eine **WPF-App (.NET)** verwendet werden (WPF-Anwendung für .NET Core oder .NET 5).
- Im WPF-Designer versorgt man die Hauptfenster-Eigenschaften **Titel**, **Width** und **Height** mit geeigneten Werten.
- Man öffnet die Datei **App.xaml** per Doppelklick auf ihren Eintrag im Projektmappen-Explorer. Sie dient zur Konfiguration der Klasse `App`, die das Visual Studio als **Application**-Ableitung automatisch erstellt hat. U. a. kann man über die Datei **App.xaml** Behandlungsmethoden zu den Ereignissen der Klasse `App` registrieren lassen.

- Man setzt die Einfügemarke in das Element **Application**,

```
<Application x:Class="ApplicationExit.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:ApplicationExit"
             StartupUri="MainWindow.xaml" Exit="Application_Exit">
    <Application.Resources>

    </Application.Resources>
</Application>
```

  wechselt im **Eigenschaften**-Fenster per Mausklick auf das Symbol ⚡ zur Liste mit den Ereignissen der Klasse `App` und setzt einen Doppelklick auf die Textbox zum Ereignis **Exit**:



- Daraufhin wird in der Datei **App.xaml.cs** mit der vom Entwickler zu verantwortenden partiellen `App`-Klassendefinition eine Ereignisbehandlungsmethode angelegt. Wir komplettieren sie durch einen Aufruf der statischen Methode **Show()** der Klasse **MessageBox**, z. B.:

```
private void Application_Exit(object sender, ExitEventArgs e) {
   MessageBox.Show("Vielen Dank für den Einsatz dieser Software!",
                   "Application Exit");
}
```

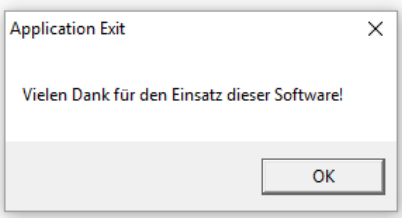
Außerdem wird in der Datei **App.g.cs** mit der vom Visual Studio gepflegten partiellen `App`-Klassendefinition (vgl. Abschnitt 12.3.4 zur Erläuterung und Lokalisation der Datei) die Ereignisregistrierung vorgenommen. In der Methode **Main()** wird ein Objekt der Klasse `App` (im Namensraum `ApplicationExit`) mit dem Namen `app` erzeugt und mit der Ausführung der Methode **InitializeComponent()** beauftragt:

```
public static void Main() {
    ApplicationExit.App app = new ApplicationExit.App();
    app.InitializeComponent();
    app.Run();
}
```

In **InitializeComponent()** wird die Instanzmethode `Application_Exit` in die Aufrufliste zum Ereignis `Exit` aufgenommen:

```
this.Exit += new System.Windows.ExitEventHandler(this.Application_Exit);
```

Im Beispiel kommt die Methode `Application_Exit` z. B. dann zum Einsatz, wenn der Benutzer auf das Schließkreuz in der Titelzeile des Anwendungsfensters klickt:



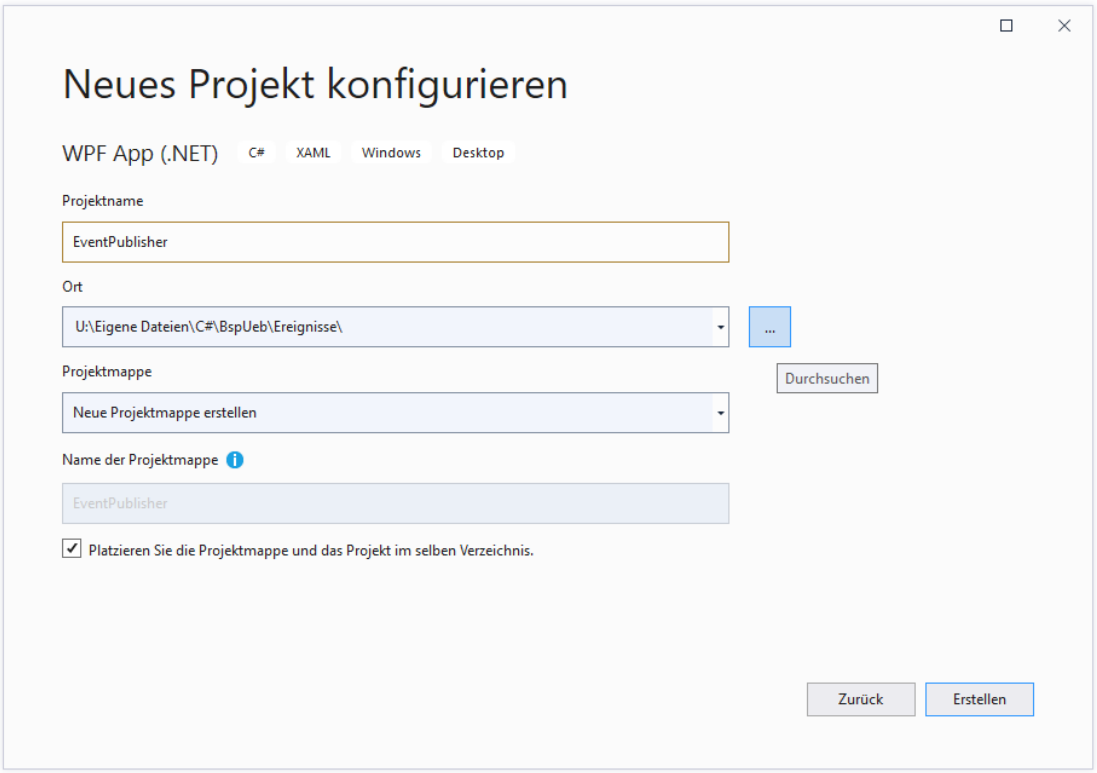
Auf eine unbedingt zu vermeidende Speicherverschwendung im Zusammenhang mit der Registrierung von Ereignisempfängern in GUI-Programmen wird bei Griffiths (2013, S. 330ff) hingewiesen:

- Ein Programm enthält mehrere Fenster.
- Ein Fenster enthält ein Member-Objekt mit einer Behandlungsmethode, die bei einem Ereignis registriert ist.
- Der Programmierer versäumt es, beim Schließen des Fensters die Ereignisregistrierung aufzuheben.
- Solange die Ereignisquelle vorhanden ist, bleiben die zum geschlossenen Fenster gehörenden Objekte erreichbar, können also nicht vom Garbage Collector abgeräumt werden.

10.2.3 Ereignisse anbieten

Zwar kommt das Registrieren eigener Behandlungsmethoden bei Ereignissen von BCL-Klassen häufiger vor als das Veröffentlichen von Ereignissen in einer eigenen Klassendefinition, doch müssen wir auch die Rolle eines Ereignisemittenden beherrschen. Weil Ereignisse meist im Kontext mit grafischen Bedienoberflächen von Interesse sind, soll auch das Beispiel aus diesem Bereich gewählt werden. Statt ein lebensfernes Konsolenbeispiel zu konstruieren, nehmen wir lieber weitere Vorgriffe auf die WPF-Technik in Kauf.

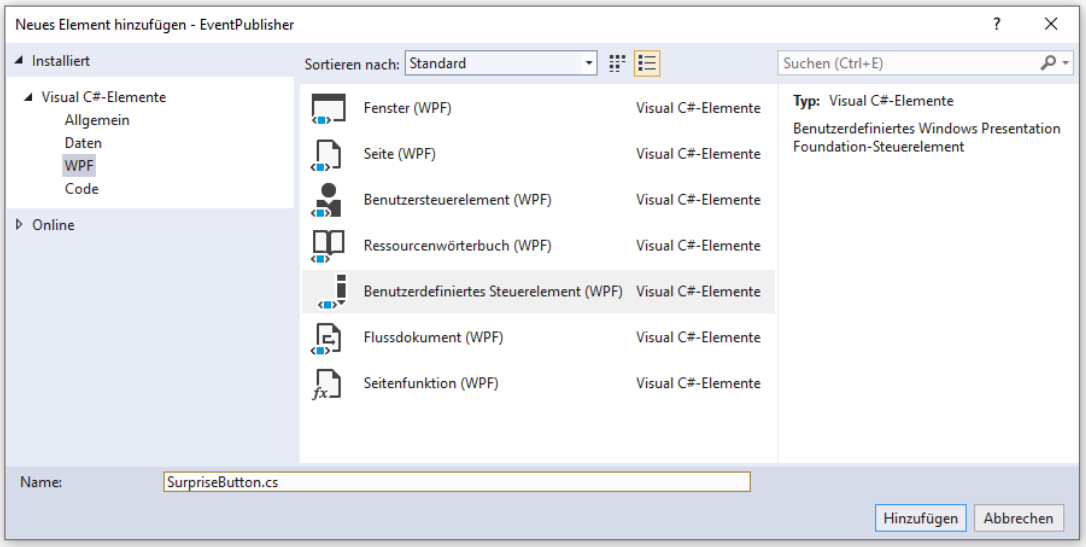
Wir erstellen im Visual Studio ein neues Projekt mit dem Namen `EventPublisher` unter Verwendung der für .NET Core und .NET 5 geeigneten Vorlage **WPF-App (.NET)**:



Nach dem **Erstellen** des Projekts ergänzen wir eine aus der BCL-Klasse **Button** abgeleitete benutzerdefinierte Steuerelementklasse, deren Objekte bei jedem Mausklick auf den Befehlsschalter eine Zufallszahl von 0 bis 9999 ziehen und das noch zu definierende Ereignis `Seven` auslösen, wenn die Zufallszahl restfrei durch 7 teilbar ist.

Die nicht ganz triviale Aufgabe, ein benutzerdefiniertes Steuerelement zu erstellen, ist im Visual Studio schnell erledigt. Öffnen Sie im **Projektmappen-Explorer** per Maus-Rechtsklick das Kontextmenü zum *Projekt* (nicht zur *Projektmappe*), und fügen Sie ein **neues Element** hinzu. Wählen Sie im Dialog für **neue Elemente** aus der Abteilung **Visual C# - Elemente > WPF** die Option

Benutzerdefiniertes Steuerelement (WPF), und tragen Sie **SurpriseButton.cs** als **Namen** ein:



Nach dem **Hinzufügen** wird die Datei im Quellcode-Editor der Entwicklungsumgebung geöffnet.

Tragen Sie **Button** (statt **Control**) als Basisklasse für `SurpriseButton` ein.

Wenn der vom Assistenten erstellte statische Konstruktor

```
static SurpriseButton() {
    DefaultStyleKeyProperty.OverrideMetadata(typeof(SurpriseButton),
            new FrameworkPropertyMetadata(typeof(SurpriseButton)));
}
```

im Quellcode verbleibt, dann ist im Anwendungsfenster von unserer **Button**-Ableitung nichts zu sehen. Grundsätzlich ist die Anweisung im statischen Konstruktor sinnvoll und notwendig, wenn sich ein benutzerdefiniertes Steuerelement stilistisch von der Basisklasse unterscheidet, wobei offenbar weitere Bedingungen für einen guten Auftritt zu erfüllen sind. Wir übernehmen für unsere **Button**-Ableitung das Erscheinungsbild der Basisklasse und streichen den statischen Konstruktor, sodass der folgende Zwischenstand für die Klasse `SurpriseButton` resultiert:

```
public class SurpriseButton : Button {
}
```

Ergänzen Sie (z. B. in der Quellcodedatei **SurpriseButton.cs**) eine von **EventArgs** abstammende Klasse, die zur Übertragung von Informationen an Interessenten für das Ereignis `Seven` dient:

```
public class SevenEventArgs : EventArgs {
  public int Nr;
}
```

Die von einem Empfänger für das Ereignis `Seven` zu implementierende Methode muss die folgende Delegatensignatur besitzen:

```
public delegate void SevenEventHandler(object sender, SevenEventArgs e);
```

Auch dieser Delegatentyp kann z. B. in der Quellcodedatei **SurpriseButton.cs** definiert werden.

Üblicherweise endet der Name eines für Ereignisempfänger zuständigen Delegatentyps mit *EventHandler*. Außerdem besitzt ein solcher Delegatentyp zwei Parameter:

- **Object** *sender*
  Im ersten Parameter identifiziert sich die Ereignisquelle.
- **EventArgs** *e*
  Der zweite Parameter besitzt einen von **EventArgs** abstammenden Typ und liefert spezifische Informationen über das Ereignis.

Unser Schalter mit Überraschungseffekt bietet ein Ereignis namens `Seven` mit dem Zugriffsschutz **public** an. Weil wir die Standardimplementation der Methoden zum Erweitern und Reduzieren der Aufrufliste (siehe Abschnitt 10.2.1) nicht ändern wollen, sieht die Ereignisdefinition in der Klasse `SurpriseButton` fast so aus wie eine Felddeklaration, wobei aber das Schlüsselwort **event** anzugeben ist:

```
public class SurpriseButton : Button {
    public event SevenEventHandler Seven;
}
```

In der Klasse `SurpriseButton` überschreiben wir die von **Button** geerbte Methode **OnClick()**, die bei jedem Mausklick auf den Befehlsschalter aufgerufen wird:

```
protected override void OnClick() {
   base.OnClick();
   if (Seven != null) {
      int losnummer = zzg.Next(10_000);
      if (losnummer % 7 == 0) {
         SevenEventArgs sea = new SevenEventArgs();
         sea.Nr = losnummer;
         Seven(this, sea);
      }
   }
}
```

Für das in **OnClick()** als Zufallszahlengenerator verwendete Instanzobjekt aus der Klasse **Random** wird noch eine Deklaration mit Initialisierung nachgetragen:

```
Random zzg = new Random();
```

Unser benutzerdefinierter Schalter reagiert auf einen Mausklick zunächst mit dem Basisklassenverhalten.[1] Ist ein `Seven`-Ereignisempfänger registriert, wird außerdem ...

- eine Zufallszahl vom Typ **int** gezogen,
- und bei Teilbarkeit durch 7 das Ereignis `Seven` ausgelöst.

Jede registrierte Ereignisbehandlungsmethode wird über den Absender informiert und mit einem Objekt vom Typ `SevenEventArgs` versorgt, das in der Instanzvariablen `Nr` die gezogene Losnummer bereithält.

Man darf ein Ereignis nur dann auslösen, wenn tatsächlich ein Delegatenobjekt vorhanden ist.[2] Das Delegatenobjekt zu einem Ereignis entsteht beim Registrieren der ersten Behandlungsmethode und verschwindet beim Entleeren seiner Aufrufliste.

Lassen Sie das Projekt mit der Tastenkombination **Strg+Umschalt+B** oder mit dem Menübefehl:

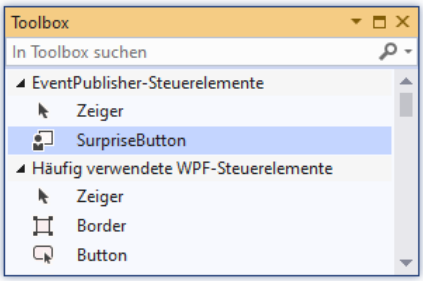
Erstellen > Projektmappe erstellen

erstellen, damit unser benutzerdefiniertes Steuerelement `SurpriseButton` in die **Toolbox** des WPF-Designers aufgenommen wird:

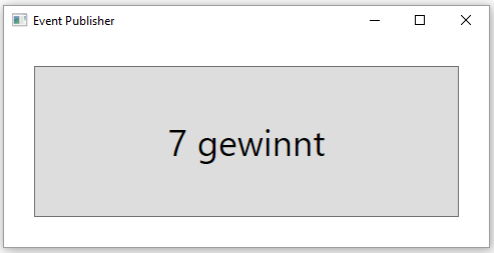
---

[1] Warum die Methode **OnClick()** bei einem Mausklick auf den Befehlsschalter aufgerufen wird, erfahren Sie später.

[2] Bei einer Multi Thread - Anwendung (siehe Abschnitt 17) sollte man sogar durch eine geeignete Synchronisation verhindern, dass ein Delegatenobjekt zwischen Existenzprüfung und Aufruf durch einen anderen Thread verworfen wird.



Setzen Sie einen Überraschungsschalter auf das Anwendungsfester, und gestalten Sie nach Bedarf die Bedienoberfläche, z. B.:



Die Hauptfensterklasse `MainWindow` besitzt nun eine Schaltfläche aus der Klasse `Suprise-Button`, wie eine Inspektion der Datei **MainWindow.xaml** zeigt:[1]

```
<Window x:Class="EventPublisher.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:EventPublisher"
        Title="Event Publisher" Height="250" Width="500">
    <Grid>
        <local:SurpriseButton x:Name="surpriseButton" Content="7 gewinnt"
        Margin="30" FontSize="36"/>
    </Grid>
</Window>
```

Der Schalter aus der Klasse `SurpriseButton` hat neben Werten für die optischen Eigenschaften (**Content**, **Margin**, **FontSize**) einen Instanzvariablennamen erhalten, damit Ereignisregistrierungen vorgenommen werden können.

Um über ein eingetretenes `Seven`-Ereignis informiert zu werden, muss die Klasse `MainWindow` ...

- eine Methode mit der Delegatensignatur `SevenEventHandler` implementieren
- und diese Methode beim Ereignis `Seven` des `SupriseButton`-Steuerelements registrieren, was z. B. im `MainWindow`-Konstruktor geschehen kann.

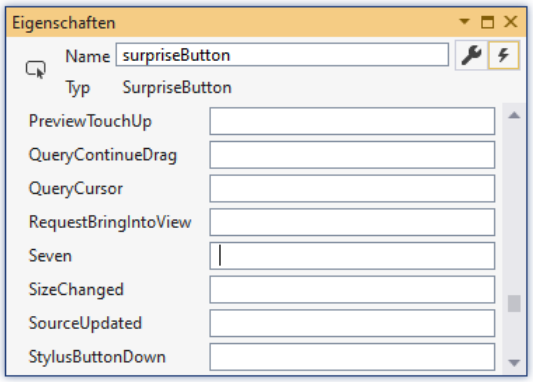
Hier ist der vom Entwickler zu verantwortende Quellcode der Klasse `MainWindow` zu sehen:

[1] Einige irrelevante Zeilen wurden gestrichen (vgl. Abschnitt 12.3.2.1).

```
public partial class MainWindow : Window {
   public MainWindow() {
      InitializeComponent();
      surpriseButton.Seven += surpriseButton_Seven;
   }

   void surpriseButton_Seven(object sender, SevenEventArgs e) {
      MessageBox.Show("Sie haben gewonnen. Losnummer: " + e.Nr, "Event Consumer");
   }
}
```

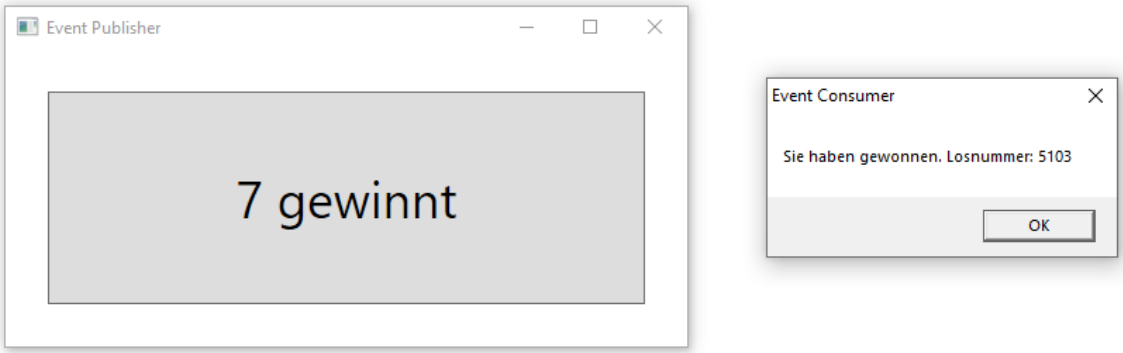
Beim Namen der Ereignisbehandlungsmethode passen wir uns dem Stil der Entwicklungsumgebung an und lassen auf den Namen der Ereignisquelle einen Unterstrich sowie den Ereignisnamen folgen.

Man kann die Ereignisregistrierung auch über das **Eigenschaften**-Fenster

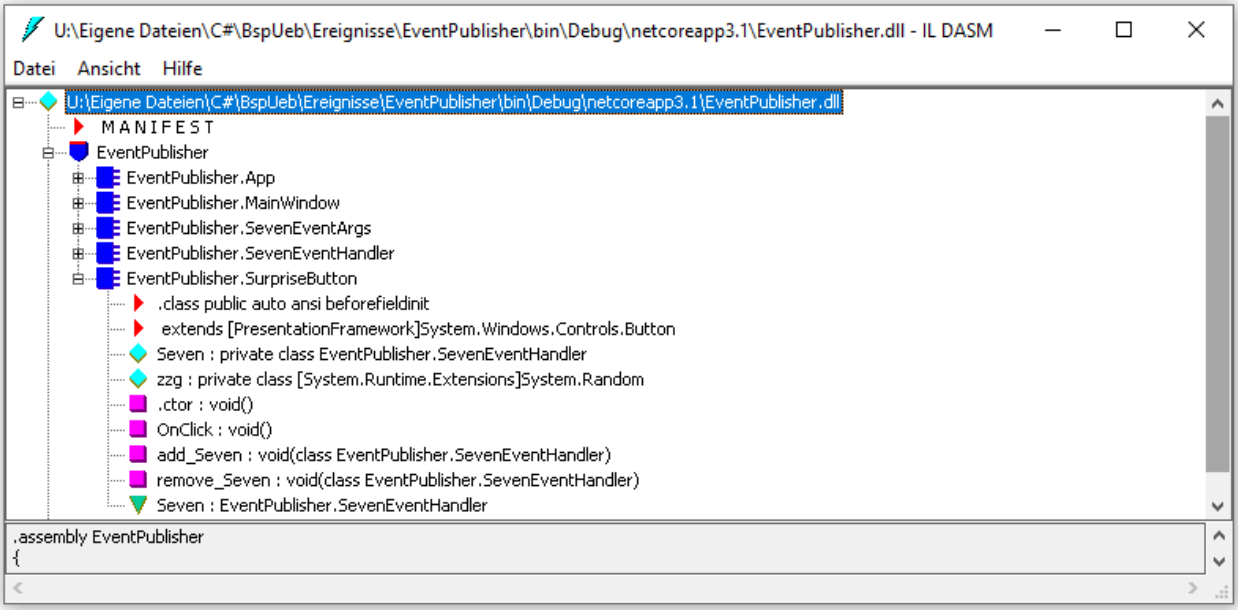


oder im XAML-Code vornehmen.

Bei ihrer Reaktion auf die Benachrichtigung nutzt die Methode `surpriseButton_Seven()` eine Information aus dem erhaltenen `SevenEventArgs`-Parameterobjekt mit der Ereignisbeschreibung:

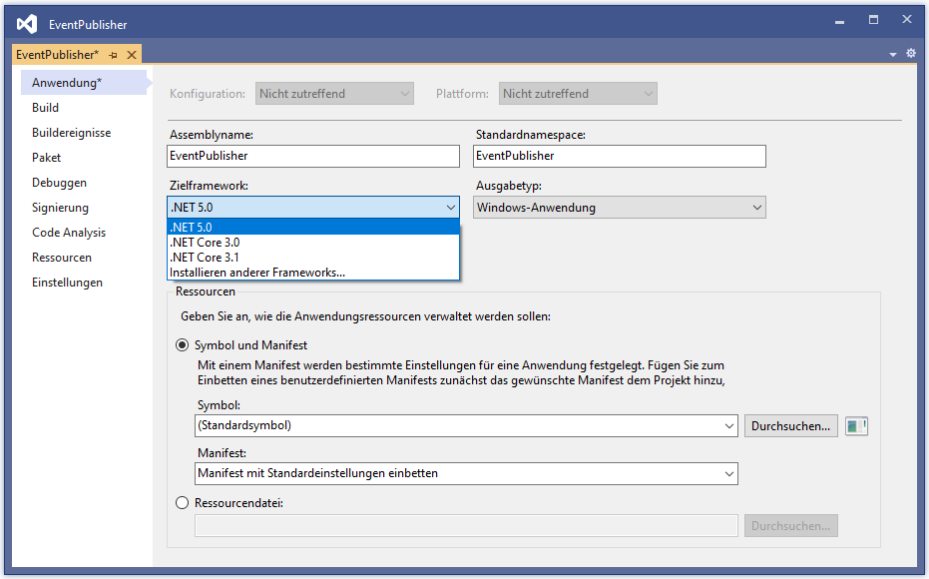


Eine Assembly-Inspektion mit dem Windows-SDK - Hilfsprogramm **ILDasm** bestätigt die zu Beginn von Abschnitt 10.2.1 formulierte Bemerkung über die Innenarchitektur von Ereignissen. In der Klasse `SurpriseButton` befinden sich aufgrund der `Seven`-Ereignisdefinition neben einem privaten Feld vom Typ `SevenEventHandler` zwei öffentliche Methoden zur Veränderung der Ereignisaufrufliste (`add_Seven()` und `remove_Seven()`):



Wegen der Wahl der Projektvorlage **WPF-App (.NET)** zeigen sich im **ILDasm**-Fenster zwei Besonderheiten im Vergleich zu den bisher verwendeten WPF-Anwendungen für das .NET Framework:

- Das Zielframework des Projekts ist **.NET Core 3.1**. Diese Einstellung der Vorlage kann nach
  Projekt > Eigenschaften > Anwendung
  geändert werden, z. B.:



- Der IL-Code befindet sich in der Datei **EventPublisher.dll**. Die Datei **EventPublisher.exe** wird zum Programmstart benötigt.

Wie zu Beginn von Abschnitt 10.2.1 beschrieben, hat auch bei einem Ereignis mit Schutzstufe **public** die beim Auslösen zu verwendende Delegatenvariable stets die Schutzstufe **private**. Damit können auch abgeleitete Klassen das Ereignis *nicht* über die Delegatenvariable auslösen, was in vielen Situationen durchaus erwünscht ist. Die folgenden Maßnahmen ermöglichen einer abgeleiteten Klasse die Kontrolle über das Ereignis:

- Das Ereignis wird durch eine Methode mit der Schutzstufe **protected** ausgelöst. Damit haben abgeleitete Klassen die Möglichkeit, das Ereignis auszulösen
- Die auslösende Methode wird als virtuell (überschreibbar) definiert. Damit kann sich eine abgeleitete Klasse durch Überschreiben der Auslösungsmethode in die Ereignisverarbeitung einschalten und z. B. das Auslösen verhindern.

Um diese Techniken zu demonstrieren, nehmen wir entsprechende Änderungen an der Klasse `SurpriseButton` vor:

```
public class SurpriseButton : Button {
   internal event SevenEventHandler Seven;
   Random zzg = new Random();

   protected override void OnClick() {
      base.OnClick();
      int losnummer = zzg.Next(10000);
      if (losnummer % 7 == 0 && Seven != null) {
         SevenEventArgs sea = new SevenEventArgs();
         sea.Nr = losnummer;
         OnSeven(sea);
      }
   }
   protected virtual void OnSeven(SevenEventArgs sea) {
      Seven(this, sea);
   }
}
```

Im Namen einer Ereignis-auslösenden Methode gibt man in der Regel nach dem Präfix **On** das Ereignis an.

Nun kann eine abgeleitete Klasse bei der Ereignisentstehung mitreden und z. B. bei einer Losnummer < 5000 den Gewinn verweigern:

```
class SBDerivation : SurpriseButton {
   protected override void OnSeven(SevenEventArgs sea) {
      if (sea.Nr >= 5000)
         base.OnSeven(sea);
   }
}
```

Das komplette Projekt ist im folgenden Ordner zu finden

…\BspUeb\Ereignisse\Event\EventPublisher

Als Ereignisanbieter kommen keinesfalls nur die mit dem Benutzer interagierenden Steuerelemente in Frage. Auch die mit der Datenverwaltung (mit der Geschäftslogik) beschäftigten Klassen nutzen Ereignisse, um z. B. über Änderungen im Datenbestand zu informieren.

10.3 Übungsaufgaben zum Kapitel 10

1) Welche von den folgenden Aussagen sind richtig?

1. Ein Datenobjekt ist unveränderlich, kann also nicht verändert, sondern nur ersetzt werden.
2. Der Rückgabetyp spielt bei der Delegatensignatur keine Rolle.
3. Bei der Delegatendefinition über eine anonyme Methode oder per Lambda-Notation können lokale Variablen sowie (statische) Felder der Umgebung verwendet werden.
4. C# unterstützt bei den Typformalparametern von generischen Delegaten die Ko- und Kontravarianz.

2) Der BCL-Delegatentyp **Predicate<in T>** im Namensraum **System** liefert für ein Argument vom Typ **T** eine boolesche Rückgabe, die darüber informiert, ob das Argument einem Kriterium genügt:

public delegate bool Predicate<in T>(T *instance***)**

Ein Parameter vom Typ **Predicate<in T>** dient in der Methode **FindAll()** der generischen Klasse **List<T>** dazu, aus der angesprochenen Liste eine Teilmenge für eine zu erstellende neue Liste zu gewinnen:

public List<T> FindAll(Predicate<T> *match***)**

Erstellen Sie per Lambda-Syntax ein Objekt vom Typ **Predicate<String>**, um aus einer Liste mit Zeichenfolgen die Elemente mit maximal 4 Zeichen in eine neue Liste zu befördern.

3) Von einem Delegatenobjekt, das an eine Methode zu übergeben ist, werden oft mehrere Varianten in Abhängigkeit von einem Parameter benötigt. Um bei Aufgabe 2 die Teillisten der Namen mit einer Maximallänge von i = 1, .., 9 Zeichen auszugeben, sollten die benötigten **Predicate<String>** -Objekte von einer Methode mit **int**-Parameter erzeugt werden, z. B. mit dem folgenden Definitionskopf:

```
public Predicate<String> ListLE(int k)
```

Man kann hier von einer *Metamethode* sprechen, weil als Rückgabe letztlich Methoden geliefert werden. Implementieren und testen Sie eine solche Metamethode.

4) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. Ein Ereignis ist ein Datentyp.
2. Weil die Delegatenvariable zu einem Ereignis grundsätzlich **privat** ist, kann man auch einer abgeleiteten Klasse keine Möglichkeit verschaffen, Kontrolle über die Auslösung eines Ereignisses zu gewinnen.
3. Die Registrierung eines Ereignisempfängers wieder aufzuheben, lohnt sich nicht.
4. Ein Ereignis kann versiegelt werden.

11 Kollektionen

Ein besonders erfolgreiches Anwendungsfeld für die im Kapitel 8 beschriebene Typgenerizität sind die Typen zur Verwaltung von Listen, Mengen, (Schlüssel-Wert) - Tabellen, Stapeln, Warteschlangen und anderen Kollektionen im Namensraum **System.Collections.Generic**. Diese Klassen werden von praktisch jedem C# - Programmierer in diversen Anwendungen zur Datenverwaltung benötigt. Auch das im Abschnitt 6.8 vorgestellte Beispielprogramm zur Präsentation von RSS-Items verwendet zur Verwaltung seiner Daten ein generisches Kollektionsobjekt:

```
var items = new List<RssItem>();
```

Wer nach der Lektüre von Kapitel 8 noch Zweifel am Nutzen der generischen Typen hatte, lernt nun zahlreiche generische Klassen mit hohem praktischem Nutzwert kennen, was im Hinblick auf die Generizität zu einem Erfahrungs- und Motivationsgewinn führen sollte. Zugleich wird belegt, dass auch scheinbar abstrakte C# - Sprachmerkmale (wie die Generizität) die (professionelle) Praxis enorm erleichtern.

Verwendet eine Anwendung mehrere nebenläufige *Ausführungsfäden* (Multithreading, siehe Kapitel 17), dann wird eventuell eine Thread-sichere Kollektion aus dem mit .NET 4.0 eingeführten Namensraum **System.Collections.Concurrent** benötigt. Bekannte Klassen aus diesem Namensraum sind **BlockingCollection<T>**, **ConcurrentDictionary<K, V>**, **ConcurrentQueue<T>** und **ConcurrentStack<T>**.[1]

Die nicht-generischen Kollektionsklassen im alten Namensraum **System.Collections** haben die im Abschnitt 8.1 am Beispiel der Klasse **ArrayList** beschriebenen Probleme und sollten *nicht* mehr verwendet werden. Neben **ArrayList** waren aus dem Namensraum **System.Collections** auch die Klassen **Hashtable**, **Queue** und **Stack** populär, bevor die überlegenen generischen Lösungen entstanden sind.

Für die Objekte der im aktuellen Kapitel vorzustellenden Klassen wird im Manuskript alternativ zur offiziellen Bezeichnung *Kollektionen* aus sprachlichen Gründen gelegentlich auch die Bezeichnung *Container* verwendet.

In diesem Kapitel beschäftigen wir uns nicht damit, eigene generische Kollektionstypen zu definieren (siehe einfache Beispiele im Abschnitt 8.2), sondern wir konzentrieren und darauf, die in der BCL zahlreich vorhandenen Typen zu nutzen. Diese Typen besitzen ausgefeilte Handlungskompetenzen (Methoden) für typische Aufgabenstellungen (z. B. Vereinigung von zwei Mengen ohne Entstehung von Dubletten), damit Programmierer möglichst selten „das Rad neu erfinden müssen“.

Es ist allgemeiner Konsens, dass in einer objektorientierten Software die Verwaltung und Transformation der Daten nach den Regeln der Geschäftslogik getrennt werden sollte von der Präsentation der Daten und der Benutzerinteraktion. Zur Bewältigung der ersten Teilaufgabe tragen die im aktuellen Kapitel vorzustellenden Kollektionstypen entscheidend bei.

11.1 Arrays versus Kollektionen

Die im Abschnitt 6.2 vorgestellten Arrays taugen als Container für Elemente mit einem identischen, frei wählbaren Typ. Sie bieten Speichereffizienz und einen schnellen Indexzugriff auf die Elemente. Man kann sich fragen, wozu eigentlich noch weitere Kollektionsklassen benötigt werden.

Beginnen wir bei den sehr häufig auftretenden listenartigen Datenstrukturen. Dabei zeigen Arrays folgende Schwächen:

---

[1] https://docs.microsoft.com/de-de/dotnet/standard/collections/thread-safe/

- Die Größe eines Arrays muss beim Erstellen festgelegt werden und kann nicht mehr geändert werden. Man kann zwar einen neuen Array anlegen und die Elemente des alten Arrays kopieren, doch entsteht dabei lästige Routinearbeit.
- Das Einfügen und Entfernen von *inneren* Elementen ist mit einem hohen Aufwand verbunden.

Kollektionen zur Verwaltung von Listen bieten hingegen Größendynamik sowie (je nach Architektur) performantes Einfügen und Löschen von inneren Elementen. Zwar verwenden viele Listenverwaltungs-Kollektionen im Hintergrund einen Array zum Speichern ihrer Elemente und müssen diesen Array bei einer Kapazitätsüberschreitung ersetzen, doch geschieht dies immerhin automatisch.

Sind für Elementsammlungen häufige Existenzprüfungen erforderlich, bietet ein Array wenig Unterstützung. Sind seine Elemente nicht sortiert, muss für jedes Element geprüft werden, ob es mit dem gesuchten übereinstimmt. Kollektionen zur Verwaltung von Mengen (siehe Abschnitt 11.4) bieten hingegen schnelle Detektionsmöglichkeiten und verhindern identische Elemente (Dubletten).

Oft sind Mengen von (Schlüssel-Wert) - Paaren zu verwalten, z. B. eine Tabelle mit den bei einem Web-Dienst aktuell angemeldeten Benutzern, wobei eine eindeutige Kennung als Schlüssel fungiert und auf ein Objekt mit den Eigenschaften des Benutzers zeigt. Eventuell stammen die Eigenschaften aus einer Datenbankzeile, die nach der Anmeldung des Benutzers von einem Datenbankserver bezogen und dann zum schnellen Zugriff im Hauptspeicher aufbewahrt wird. Es melden sich ständig Benutzer an oder ab, und beim Versuch, eine solche Datenstruktur mit einem Array zu verwalten, treten die eben schon beschriebenen Probleme auf (feste Anzahl von Elementen, umständliches Einfügen und Löschen, aufwändige Suche nach Schlüsseln).

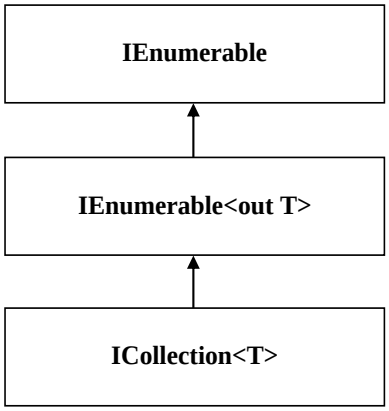
Als weiterer Nachteil von Arrays ist ihr kovariantes Verhalten hinsichtlich des Elementtyps zu nennen (siehe Abschnitt 8.2.3).

Im Zusammenhang mit der Verwaltung von veränderlichen Strukturinstanzen wird aber auch von einem Vorteil der Arrays gegenüber generischen Kollektionen zu berichten sein (siehe Abschnitt 11.3.1).

Im zu modellierenden Aufgabenbereich treten oft Datenstrukturen vom Typ Liste, Menge, (Schlüssel-Wert) - Tabelle, Stapel oder Warteschlange auf, und im BCL-Namensraum **System.Collections.Generic** finden sich oft passende Typen, sodass im Vergleich zur Verwendung von Arrays eine bessere Modellierung und ein besser lesbarer Quellcode resultieren. Sicherlich sind Sie mittlerweile in der Lage, eine generische Kollektionsklasse zu definieren, die ihre Elemente in einem Array lagert und diesen bei Bedarf automatisch durch ein größeres Exemplar ersetzt. Allerdings sind in der BCL bereits exzellente Lösungen für derartige Standardaufgaben vorhanden, sodass wir uns auf andere Herausforderungen konzentrieren können.

11.2 Das Interface ICollection<T> mit Basiskompetenzen von Kollektionen

Viele von den anschließend vorgestellten Kollektionsklassen erfüllen die Schnittstelle **ICollection<T>** sowie die Schnittstelle **IEnumerable<T>**, von der **ICollection<T>** abstammt (alle Typen aus dem Namensraum **System.Collections.Generics**). Die generische Schnittstelle **IEnumerable<T>** erweitert die nicht-generische Schnittstelle **System.Collections.IEnumerable**:



Die Schnittstelle **IEnumerable<T>** verpflichtet eine implementierende Klasse, als Rückgabe der Methode **GetEnumerator()** ein Objekt vom Typ **IEnumerator<T>** zu liefern, das ein Iterieren durch die Kollektionselemente erlaubt und in der Regel im Rahmen einer **foreach**-Schleife verwendet wird (siehe Abschnitt 9.7).

Eine das Interface **ICollection<T>** implementierende Klasse beherrscht die folgenden Methoden:

- **public void Add(T** *element***)**
  Das Parameterelement wird in die Kollektion aufgenommen. Bei einer Liste wird es am Ende angehängt.
- **public bool Contains(T** *element***)**
  Diese Methode informiert darüber, ob das Parameterelement in der Kollektion vorhanden ist.
- **public bool Remove(T** *element***)**
  Das erste Vorkommen des Elements wird aus der Kollektion entfernt, falls es dort vorhanden ist. Über den Rückgabewert erfährt man, ob die Kollektion durch den Aufruf verändert worden ist.
- **public void CopyTo(T[]** *array***, int** *index***)**
  Alle Kollektionselemente werden in einen kompatiblen Array ab der angegebenen Indexposition kopiert. Vorhandene Array-Elemente werden dabei überschrieben.
- **public void Clear()**
  Alle Elemente werden entfernt.

Neben Methoden stehen auch Eigenschaften im Pflichtenheft einer Klasse, die das Interface **ICollection<T>** implementiert:

- **public int Count { get; }**
  Es wird die Anzahl der Elemente geliefert.
- **public bool IsReadOnly { get; }**
  Man erfährt, ob die Kollektion schreibgeschützt ist.

Wenn eine Kollektionsklasse ...

- die Schnittstelle **IEnumerable** implementiert,
- und eine Instanz- oder Erweiterungsmethode namens **Add()** zur Aufnahme eines Elements besitzt,[1]

dann kann die Kollektion bei der Erstellung auf syntaktisch einfache Weise per **Kollektionsinitialisierer** bevölkert werden, z. B.:[2]

---

[1] Zu Erweiterungsmethoden siehe Abschnitt 7.13.

[2] https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/classes-and-structs/object-and-collection-initializers

```
var nl = new List<String> {"Otto", "Marita", "Theo"};
```

11.3 Verwaltung einer Liste

Eine Liste enthält eine Sequenz von Elementen mit einer definierten Reihenfolge. Alle Elemente sind vom selben Typ (Klasse, Struktur oder Schnittstelle). Erfüllt eine Listenverwaltungsklasse die Schnittstelle **IList<T>**, dann kann man auf die Elemente wahlfrei über einen nullbasierten Index zugreifen. Es ist also wie bei einem Array möglich, das Element an einer bestimmten Position abzurufen oder festzulegen.

Im Unterschied zu einem Array wird eine Liste bei Bedarf automatisch vergrößert. Wir haben also einen größendynamischen Container zur Verfügung, der dank Typgenerizität Elemente von einem wählbaren Typ sortenrein (mit Compiler-Typsicherheit) verwaltet.

Für Listen finden sich sehr viele Einsatzmöglichkeiten bei der Software-Entwicklung. Man verwendet sie z. B. für ...

- die aktuell von einem Steuerelement der Bedienoberfläche angebotenen Optionen,
- die Bestellungen eines Kunden, der aus einer Datenbank geladen wurde,
- die aus einem Text sukzessiv extrahierten Wörter.

Die Elemente einer Liste müssen (im Unterschied zu den Elementen einer Menge, vgl. Abschnitt 11.4) *nicht* verschieden sein.

Neben den anschließend vorgestellten Typen zur Listenverwaltung enthält die BCL noch Lösungen für wichtige Spezialfälle, die im Manuskript nicht behandelt werden, z. B.:

- Stapel
  Die Klasse **Stack<T>** verwaltet eine Liste nach dem LIFO-Prinzip (*Last In First Out*). Im Abschnitt 8.2.1 haben wir eine simple Variante selbst erstellt.
- Warteschlange
  Die Klasse **Queue<T>** verwaltet eine Liste nach dem FIFO-Prinzip (*First In First Out*).

11.3.1 Die Klasse List<T> mit Array-Unterbau

Die generische Klasse **List<T>** zur bequemen und typsicheren Verwaltung einer Sequenz von Elementen eines festen Typs ist schon aus dem Abschnitt 8.1 bekannt. Ein **List<T>** - Objekt speichert seine Elemente in einem internen Array **T[]**, sodass ...

- Lesezugriffe sehr performant sind,
- Veränderungen hingegen zeitaufwändig sind, wenn ...
  - neue Instanzen im Inneren der Liste eingefügt oder entfernt werden müssen
  - oder aufgrund einer Kapazitätsüberschreitung ein neuer Array angelegt werden muss.

Eine automatische Kapazitätserweiterung orientiert sich an der bisherigen Größe, damit diese kostspielige Maßnahme möglichst selten erforderlich wird. In einer von insgesamt drei Konstruktorüberladungen kann man die initiale Kapazität festlegen, z. B.:

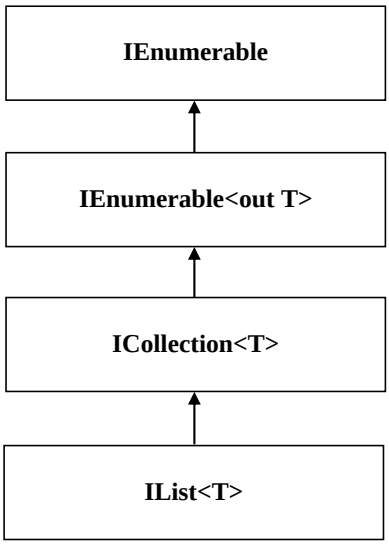
```
var nl = new List<String>(500);
```

Über die Eigenschaft **Capacity** erfährt man die aktuelle Kapazität einer Liste, welche meist die per **Count**-Eigenschaft abfragbare aktuelle Länge übertrifft, z. B.:

Quellcode	Ausgabe

`using System;`
`using System.Collections.Generic;`
`class Prog {`
`  static void Main() {`
`  var nl = new List<String>(5) {"Otto", "Marita", "Theo"};`
`  Console.WriteLine($"Länge:  {nl.Count}\nKapazität: {nl.Capacity}");`
`  }`
`}`
	`Länge:  3`
	`Kapazität: 5`

Die Klasse **List<T>** implementiert die Schnittstelle **IList<T>** und damit indirekt auch die Schnittstelle **ICollection<T>**, von der **IList<T>** abstammt, sowie die Schnittstellen im Stammbaum von **ICollection<T>**:



Über die **ICollection<T>** - Methoden (siehe Abschnitt 11.2) hinaus besitzt ein die Schnittstelle **IList<T>** implementierender Typ die folgenden Methoden:

- **public int IndexOf(T** *element***)**
  Falls das Element in der Liste vorhanden ist, wird der Index des ersten Auftretens geliefert, anderenfalls der Wert -1.
- **public void Insert(int** *pos***, T** *element***)**
  Das Parameterelement wird an der gewünschten Indexposition eingefügt.
- **public void RemoveAt(int** *pos***)**
  Das Element an der angegebenen Indexposition wird entfernt.

Außerdem schreibt die Schnittstelle **IList<T>** einen Indexer vor, der es ermöglicht, das Element an der Position *index* abzurufen oder festzulegen:

public T this[int *index***] { get; set; }**

Die Klasse **List<T>** enthält über die **IList<T>** - Verpflichtungen hinaus die Instanzmethode **AsReadOnly()**, um einen schreibgeschützten Zugriff zu ermöglichen:

public ReadOnlyCollection<T> AsReadOnly()

Der Aufrufer erhält eine *Sicht* auf die angesprochene Liste, ...

- die keine Veränderung zulässt,
- aber jede Änderung der zugrunde liegenden Liste sofort berücksichtigt.

Außerdem beherrscht die Klasse **List<T>** zum Suchen und Sortieren dieselben Methoden wie ein Array (vgl. Abschnitt 6.2.6), wobei die statischen Methoden der Klasse **Array** durch Instanzmethoden der Klasse **List<T>** ersetzt werden, z. B.:

Quellcode	Ausgabe

`using System;`
`using System.Collections.Generic;`
 	
`class Prog {`
`  static void Main() {`
`  var iar = new List<int> { 2, 13, 12, 15, 7 };`
`  Console.WriteLine("Index: " +`
`  iar.FindIndex(i => i % 3 == 0));`
`  }`
`}`
	`Index: 2`

Die Methode **FindIndex()** sucht ein Element mit einer bestimmten Eigenschaft. Im Beispiel wird der Index des ersten durch 3 teilbaren Elements ermittelt. Als Parameter von **FindIndex()** wird zur Beurteilung der Elemente ein per Ausdrucks-Lambda (siehe Abschnitt 10.1.6) realisiertes Delegatenobjekt vom Typ **Predicate<T>** verwendet (Rückgabetyp **bool**). Wird die Methode **FindIndex()** nicht fündig, liefert sie die Rückgabe -1.

Weil die Klasse **List<T>** den von der Schnittstelle **IList<T>** vorgeschriebenen Indexer besitzt, ist wie bei einem Array ein Elementzugriff über den **[]** - Operator möglich (zu Indexern siehe Abschnitt 5.8).

Im Zusammenhang mit dem Indexer für **List<T>** - Objekte soll noch einmal demonstriert werden, warum viele Autoren dringend von veränderlichen *Strukturen* abraten (vgl. Abschnitt 6.1.1). Sind die Listenelemente vom Typ einer Klasse, treten beim Indexzugriff keine Überraschungen auf. Im folgenden Beispielprogramm wird ein per Indexer angesprochenes Objekt der Klasse `Punkt` gebeten, sich zu `Bewegen()`. Anschließend wird per **WriteLine()** - Aufruf nachgewiesen, dass sich das Objekt an der neuen Position befindet:

Quellcode	Ausgabe

`using System;`
`using System.Collections.Generic;`
`public class Punkt {`
`  double x, y;`
`  public Punkt(double xpar, double ypar) {`
`  x = xpar;`
`  y = ypar;`
`  }`
`  public void Bewegen(double hor, double vert) {`
`  x = x + hor;`
`  y = y + vert;`
`  }`
`  public override string ToString() {`
`  return "(" + x + "; " + y + ")";`
`  }`
`}`
`class Prog {`
`  static void Main() {`
`  var pl = new List<Punkt> {new Punkt(1, 2), new Punkt(3, 4)};`
`  pl[0].Bewegen(5, 5);`
`  Console.WriteLine(pl[0]);`
`  }`
`}`
	`(6; 7)`

Wird im Beispiel jedoch eine *Struktur* namens `Punkt` anstatt einer Klasse definiert, dann führt die per Indexer angesprochene Instanz die Methode `Bewegen()` zwar aus, doch produziert der an-

schließende **WriteLine()** - Methodenaufruf die (vermutlich von vielen Programmierern *nicht* erwartete) Ausgabe

```
(1; 2)
```

Hinter dem Indexer steckt u. a. eine verkappte **get**-Methode, und die liefert eine *Kopie* der Strukturinstanz mit dem angefragten Indexwert.

Das beschriebene Problem tritt übrigens *nicht* auf, wenn statt eines **List**<Punkt> - Objekts der Array Punkt[] verwendet wird, weil dann z. B. der Ausdruck pl[0] das Element mit dem Indexwert 0 *identifiziert*, anstatt eine Kopie dieses Elements zu liefern (Griffith 2013, S. 167).

Quellcode	Ausgabe

`using System;`
`using System.Collections.Generic;`
`public struct Punkt {`
`  . . .`
`}`
`class Prog {`
`  static void Main() {`
`  Punkt[] pa = new Punkt[2] { new Punkt(1, 2), new Punkt(3, 4) };`
`  pa[0].Bewegen(5, 5);`
`  Console.WriteLine(pa[0]);`
`  }`
`}`
	`(6; 7)`

11.3.2 Die Klasse LinkedList<T> für verkettete Elemente

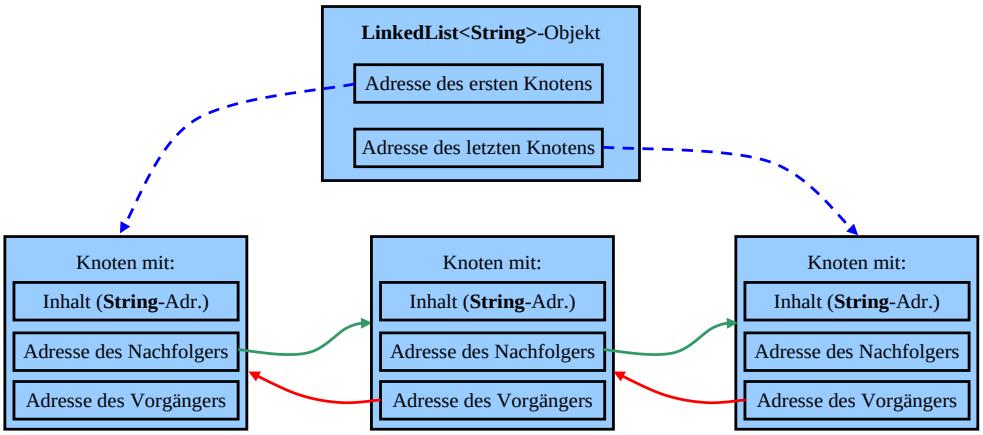
Die Klasse **List<T>** arbeitet intern mit einem Array zum Speichern der Elemente und bietet daher einen schnellen wahlfreien Zugriff. Auch das Anhängen neuer Elemente am Ende der Liste verläuft flott, wenn nicht gerade die Kapazität des Arrays erschöpft ist. Dann wird es erforderlich, einen größeren Array anzulegen und alle Elemente dorthin zu kopieren. Beim Einfügen bzw. Löschen von *inneren* Elementen müssen die neuen bzw. früheren rechten Nachbarn zeitaufwändig nach rechts bzw. links verschoben werden.

Die Klasse **LinkedList<T>** arbeitet intern mit einer **doppelt verketteten Liste** bestehend aus selbständigen Objekten der Klasse **LinkedListNode<T>**, die jeweils ein Element aufbewahren.

Ein Objekt der Knotenklasse **LinkedListNode<T>** ...

- macht den eigentlichen Inhalt über eine get/set - Eigenschaft namens **Value** vom Typ **T** zugänglich
- und kennt über die get-only - Eigenschaften **Previous** bzw. **Next** vom Typ **LinkedListNode<T>** den vorherigen bzw. nächsten Knoten.

Anschließend ist ein **LinkedList<String>** - Objekt zu sehen, das eine Liste mit drei Knoten bzw. Elementen verwaltet:



Im Abschnitt 5.8 haben wir übrigens eine (allerdings nur einfach) verkettete Liste selbst gebaut, um die Definition eines Indexers zu demonstrieren.

Vorteile einer verketteten Liste im Vergleich zu einer Array-basierten Liste:

- Die Länge der Liste ist zu keinem Zeitpunkt festgelegt, sodass keine aufwändigen Maßnahmen zur Kapazitätsanpassung erforderlich sind.
- Beim Einfügen und Löschen von inneren Elementen müssen keine anderen Elemente verschoben werden. Stattdessen wird ein Knoten zur Aufbewahrung des Elements erzeugt bzw. gelöscht, und die Adressketten werden neu verknüpft.

Nachteile einer verketteten Liste im Vergleich zu einer Array-basierten Liste:

- Die Klasse **LinkedList<T>** besitzt keinen Indexer für den wahlfreien Zugriff auf Listenelemente. Um das Listenelement an einer bestimmten Position aufzusuchen, müsste die Liste ausgehend vom ersten oder letzten Element durchlaufen werden, was zu einer schlechten Performanz führen würde.
- Eine verkettete Liste benötigt mehr Speicher, weil sich jedes Element in einem selbständigen Knotenobjekt auf dem Heap befindet.

Insgesamt sind verkettete Listen geeignet für Algorithmen, die …

- häufig innere Elemente einfügen oder entfernen,
- Elemente nur sequentiell aufsuchen.

Die Klasse **LinkedList<T>** implementiert zwar die Schnittstelle **ICollection<T>**, aber *nicht* die Schnittstelle **IList<T>** (siehe Stammbaum der Kollektions-Schnittstellen im Abschnitt 11.3.1), sodass insbesondere kein Indexer zur Verfügung steht. Vorhanden sind neben den vom Interface **ICollection<T>** vorgeschriebenen Methoden und Eigenschaften (siehe Abschnitt 11.2) u. a. die folgenden Handlungskompetenzen:

- **First, Last**
  Diese Eigenschaften zeigen auf den ersten bzw. auf den letzten Knoten, bei einer leeren Liste auf **null**.
- **public LinkedListNode<T> AddFirst(T** *element***)**
  Diese Methode fügt ein neues Objekt der Klasse **LinkedListNode<T>** zur Aufbewahrung des Parameterelements am Anfang ein und liefert eine Referenz auf das neu erstellte Objekt zurück.
- **public LinkedListNode<T> AddLast(T** *element***)**
  Es wird ein neues Objekt der Klasse **LinkedListNode<T>** zur Aufbewahrung des Parameterelements am Ende angehängt und eine Referenz auf dieses Objekt zurückgeliefert.

- **public LinkedListNode<T> AddBefore(LinkedListNode<T>** *node***, T** *element***)**
  Vor dem im ersten Parameter angegebenen Knoten wird ein neues Objekt der Klasse **LinkedListNode<T>** zur Aufbewahrung des zweiten Parameters angelegt und eine Referenz auf den eingefügten Knoten zurückgeliefert.
- **public LinkedListNode<T> AddAfter(LinkedListNode<T>** *node***, T** *element***)**
  Nach dem im ersten Parameter angegebenen Knoten wird ein neues Objekt der Klasse **LinkedListNode<T>** zur Aufbewahrung des zweiten Parameters angelegt und eine Referenz auf den eingefügten Knoten zurückgeliefert.
- **public void RemoveFirst()**
  Der erste Knoten wird entfernt.
- **public void RemoveLast()**
  Der letzte Knoten wird entfernt.
- **public void Remove(LinkedListNode<T>** *element***)**
  Der per Aktualparameter angegebene Knoten wird entfernt.

Die Methode

public void Add(T *element***)**

aus der implementierten Schnittstelle **ICollection<T>** wird von **LinkedList<T>** durch die explizite Schnittstellenimplementierung (siehe Abschnitt 9.6) realisiert:

```
void ICollection<T>.Add(T value) {
  AddLast(value);
}
```

Folglich kann die Methode nur über eine **ICollection<T>** - Referenz aufgerufen werden, z. B.:

```
var lili = new LinkedList<string>();
lili.AddLast("a");
ICollection<string> iclili = lili;
iclili.Add("b");
```

11.4 Verwaltung einer Menge

Zur Verwaltung einer *Menge* von Elementen, die im Unterschied zu einer Liste *keine Dubletten* aufweisen darf, enthält die BCL u. a. die generischen Klassen **HashSet<T>** und **SortedSet<T>**. Diese Klassen sind nützlich, wenn Mengen im Sinne der Mathematik zu modellieren sind und entsprechende Operationen benötigt werden (z. B. Durchschnitt, Vereinigung oder Differenz von zwei Mengen). Im Vergleich zu anderen Kollektionsklassen können sie Mengenzugehörigkeitsprüfungen sehr schnell ausführen, was sie unverzichtbar macht, wenn derartige Prüfungen in großer Zahl auftreten.

Während sich die Klasse **HashSet<T>** anbietet, wenn für die Instanzen des Elementtyps keine Anordnung besteht bzw. interessiert, eignet sich **SortedSet<T>** dann, wenn die Instanzen des Elementtyps eine relevante Ordnung besitzen (z. B. Zeichenfolgen).

Anwendungsbeispiele:

- In einem Programm zur Urlaubsplanung in einer Firma könnten die Urlaubstage eines Mitarbeiters jeweils in einer Menge verwaltet werden, um z. B. über eine Durchschnittsbildung schwach besetzte Tage festzustellen.
- In einem Bewerbungsverfahren mit einer Liste von unverzichtbaren Kompetenzen (z. B. C#, HTML, CSS, SQL, etc.) könnten die Kompetenzen jedes Bewerbers durch eine Menge verwaltet werden, um z. B. über eine Vereinigungsbildung festzustellen, ob bei einer bestimmten Kombination von Neueinstellungen alle Kompetenzen vertreten sind.

11.4.1 Hashtabellen und die Klasse HashSet<T>

Die beiden im Manuskript behandelten generischen Mengenverwaltungsklassen (**HashSet<T>** und **SortedSet<T>**) haben die folgenden Eigenschaften gemeinsam:

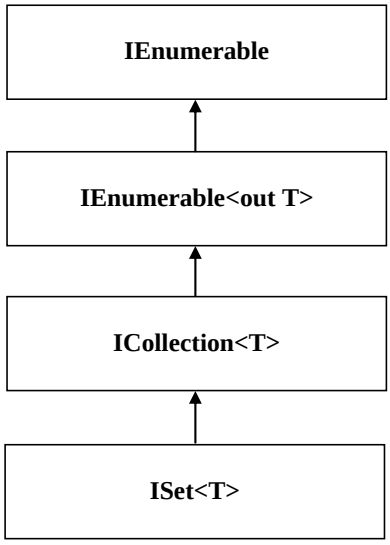
- garantierte Eindeutigkeit der Elemente (Verhinderung von Dubletten)
- schnelle Existenzprüfung

Als Besonderheiten der Klasse **HashSet<T>** sind zu nennen:

- keine relevante Anordnung der Elemente
- sehr schnelle Existenzprüfung

11.4.1.1 Handlungskompetenzen der Klasse HashSet<T>

Welche Handlungskompetenzen man bei der Klasse **HashSet<T>** erwarten kann, zeigt der Stammbaum der implementierten Schnittstellen:



Die wichtigsten **HashSet<T>** - Instanzmethoden sind:

- **public bool Add(T** *element***)**
  Das Parameterelement wird in die Menge aufgenommen, falls es dort noch nicht vorhanden ist. Über den Rückgabewert erfährt man, ob die Menge durch den Aufruf verändert worden ist.
- **public bool Contains(T** *element***)**
  Diese Methode informiert darüber, ob das fragliche Element in der Menge vorhanden ist.
- **public void IntersectWith(IEnumerable<T>** *other***)**
  Das angesprochene **HashSet<T>** - Objekt streicht alle Elemente, die sich *nicht* in der Parameterkollektion (ein Objekt einer Klasse, welche die Schnittstelle **IEnumerable<T>** erfüllt) befinden. Man erhält also die Schnittmenge:

  M (angesprochene Menge) $\cap$ **P** (Parametermenge)

- **public void UnionWith(IEnumerable<T>** *other***)**
  Das angesprochene **HashSet<T>** - Objekt nimmt aus der Parameterkollektion (ein Objekt einer Klasse, welche die Schnittstelle **IEnumerable<T>** erfüllt) alle Elemente auf, die nicht zu Dubletten führen. Man erhält also die Vereinigungsmenge:

  M (angesprochene Menge) ∪ **P** (Parametermenge)

- **public void ExceptWith(IEnumerable<T>** *other***)**
  Das angesprochene **HashSet<T>** - Objekt streicht alle Elemente, die sich in der Parameterkollektion (ein Objekt einer Klasse, welche die Schnittstelle **IEnumerable<T>** erfüllt) befinden. Man erhält also die Differenzmenge:

  M (angesprochene Menge) - **P** (Parametermenge)

- **public bool IsSubsetOf(IEnumerable<T>** *other***)**
  Wenn das angesprochene **HashSet<T>** - Objekt eine Teilmenge der Parameterkollektion (ein Objekt einer Klasse, welche die Schnittstelle **IEnumerable<T>** erfüllt) ist, wird **true** zurückgemeldet, anderenfalls **false**.

- **public bool IsProperSubsetOf(IEnumerable<T>** *other***)**
  Wenn das angesprochene **HashSet<T>** - Objekt eine echte Teilmenge der Parameterkollektion (ein Objekt einer Klasse, welche die Schnittstelle **IEnumerable<T>** erfüllt) ist (also weniger Elemente enthält), wird **true** zurückgemeldet, anderenfalls **false**. Die angesprochene Menge **M** ist eine *echte* Teilmenge der Parametermenge **P**, wenn **M** eine Teilmenge von **P** ist, und **P** mindestens ein Element enthält, das nicht zu **M** gehört.

- **public bool IsSupersetOf(IEnumerable<T>** *other***)**
  Wenn das angesprochene **HashSet<T>** - Objekt eine Obermenge der Parameterkollektion (ein Objekt einer Klasse, welche die Schnittstelle **IEnumerable<T>** erfüllt) ist, dann wird **true** zurückgemeldet, anderenfalls **false**.

- **public bool IsProperSupersetOf(IEnumerable<T>** *other***)**
  Wenn das angesprochene **HashSet<T>** - Objekt eine echte Obermenge der Parameterkollektion (ein Objekt einer Klasse, welche die Schnittstelle **IEnumerable<T>** erfüllt) ist (also eine Obermenge ist und noch zusätzliche Elemente enthält), dann wird **true** zurückgemeldet, anderenfalls **false**.

Das folgende Programm demonstriert einige von den aufgelisteten Methoden:

Quellcode	Ausgabe

```
using System;
using System.Collections.Generic;
class HashSetDemo {
  static void Main() {
    var m1 = new HashSet<char> { 'a', 'b', 'c' };
    Console.WriteLine("Menge 1:");
    foreach (char c in m1) Console.Write(c + " ");

    m1.Add('b');
    Console.WriteLine("\n\nMenge 1 nach 2. b-Aufn.:");
    foreach (char c in m1) Console.Write(c + " ");

    var m2 = new HashSet<char>() { 'c', 'd', 'e' };
    Console.WriteLine("\n\nMenge 2:");
    foreach (char c in m2) Console.Write(c + " ");

    Console.WriteLine("\n\nb in der Menge 2? "+m2.Contains('b'));

    var schnitt = new HashSet<char>(m1);
    schnitt.IntersectWith(m2);
    Console.WriteLine("\nSchnittmenge:");
    foreach (char c in schnitt) Console.Write(c + " ");

    var vereinigung = new HashSet<char>(m1);
    vereinigung.UnionWith(m2);
    Console.WriteLine("\n\nVereinigungsmenge:");
    foreach (char c in vereinigung) Console.Write(c + " ");

    var differenz = new HashSet<char>(m1);
    differenz.ExceptWith(m2);
    Console.WriteLine("\n\nDifferenzmenge:");
    foreach (char c in differenz) Console.Write(c + " ");

    m1.Remove('a');
    Console.WriteLine("\n\nMenge 1 ohne a:");
    foreach (char c in m1) Console.Write(c + " ");

    m1.Clear();
    Console.WriteLine("\n\nMenge 1 nach Clear:");
    foreach (char c in m1) Console.Write(c + " ");
  }
}
```

```
Menge 1:
a b c

Menge 1 nach 2. b-Aufn.:
a b c

Menge 2:
c d e

b in der Menge 2? False

Schnittmenge:
c

Vereinigungsmenge:
a b c d e

Differenzmenge:
a b

Menge 1 ohne a:
b c

Menge 1 nach Clear:
```

Weil **HashSet<T>** die Schnittstelle **IEnumerable<T>** erfüllt, kann man in einer **foreach**-Schleife über die Elemente einer **HashSet<T>** - Kollektion iterieren (siehe Abschnitt 9.7), wobei aber die Reihenfolge der Elemente quasi zufällig ist.

Damit die Instanzen eines Typs durch die Kollektionsklasse **HashSet<T>** sinnvoll verwaltet werden können, müssen in der Regel die von der Urahnklasse **Object** geerbten Methoden **Equals()** und **GetHashCode()** überschrieben werden:

- **Equals()** definiert die Gleichheit von zwei Instanzen und ist damit das Kriterium zur Identifikation von Dubletten
- Mit Hilfe von **GetHashCode()** werden die schnellen Existenzprüfungen realisiert (siehe Abschnitt 11.4.1.2).

Über die folgende **HashSet<T>** - Konstruktorüberladung kann man dafür sorgen, dass für Identitätsvergleiche nicht die **Equals()** - Methode des Elementtyps zuständig ist, sondern ein Objekt aus einer Klasse, welche die Schnittstelle **IEqualityComparer<T>** im Namensraum **System.Collections.Generic** implementiert:

public HashSet(IEqualityComparer<T> *comparer***)**

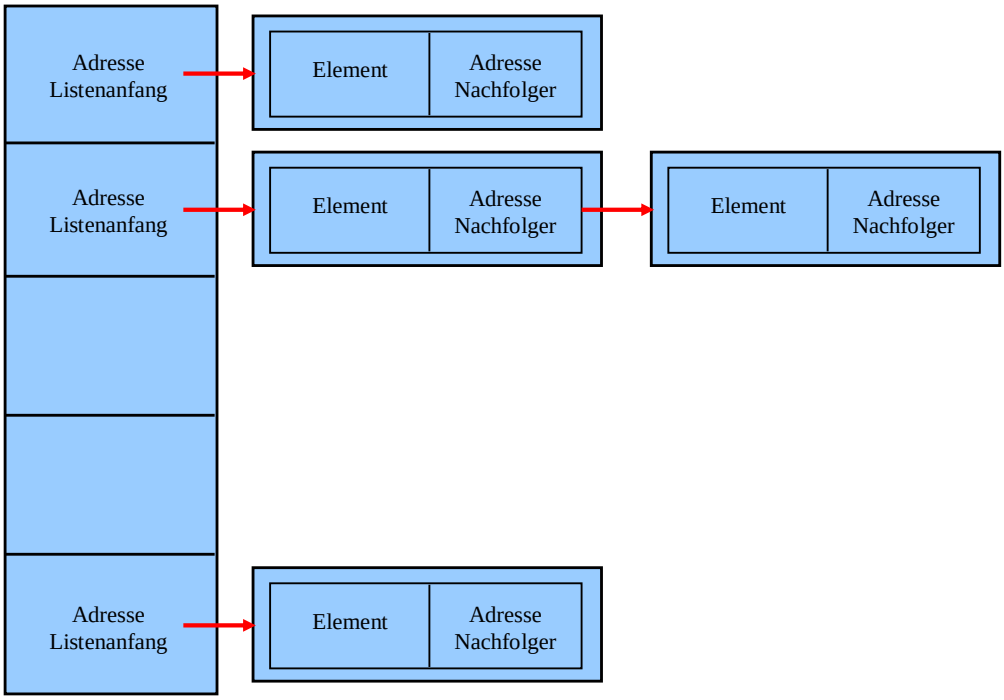
Die Klasse **StringComparer** erfüllt das Interface **IEqualityComparer<String>**. Ihre statische Eigenschaft **OrdinalIgnoreCase** zeigt auf ein Objekt der Klasse **StringComparer**, das für **String**-Objekte einen Identitätsvergleich unabhängig von der Groß-/Kleinschreibung realisiert, z. B.:

Quellcode	Ausgabe

`using System;`
`using System.Collections.Generic;`
`class Program {`
`  static void Main() {`
`  var hs = new HashSet<string>(StringComparer.OrdinalIgnoreCase) {`
`  "Eins", "eins", "zwei" };`
`  foreach (string s in hs) Console.WriteLine(s);`
`  }`
`}`
	`Eins`
	`zwei`

11.4.1.2 Hashtabellen

Zur Realisation von schnellen Existenzprüfungen kommt bei der Klasse **HashSet<T>** eine sogenannte *Hashtabelle* zum Einsatz. Die zentrale Designidee besteht darin, zur Speicherung der Elemente einen Array zu verwenden, der sogenannte *buckets* (dt.: *Eimer*) als Elemente enthält, wobei es sich um einfach verkettete Listen handelt:



Bei der Aufnahme eines neuen Elements und auch bei der Existenzprüfung wird der Bucket-Index über die typspezifische Implementierung der bereits in der Urahnklasse **Object** vorhandenen und für einen Elementtyp geeignet zu überschreibenden Instanzmethode **GetHashCode()** ermittelt:

public int GetHashCode()

Beim Einfügen eines neuen Elements ist die Bucket-Liste zum berechneten Index idealerweise noch leer. Anderenfalls spricht man von einer *Hash-Kollision*, und es entsteht ein Zusatzaufwand.

Wegen der folgenden Anforderungen an eine zum Befüllen einer Hashtabelle einzusetzende **GetHashCode()** - Methode (bzw. an die in dieser Methode realisierte Hash-Funktion) ist in der

Regel im **T**-Konkretisierungstyp das **Object**-Erbstück **GetHashCode()** durch eine angepasste Implementierung zu überschreiben:[1]

- Sind zwei Instanzen identisch im Sinne der **Equals()** - Methode, dann müssen sie bei einem **GetHashCode()** - Aufruf denselben Wert liefern.
- Während eines Programmlaufs müssen alle Methodenaufrufe für ein Objekt denselben Wert liefern, solange bei diesem Objekt keine Veränderungen mit Relevanz für die **Equals()** - Methode auftreten.
- Die **GetHashCode()** - Werte sollten möglichst zufällig und damit gleichmäßig über den möglichen Indexwertebereich verteilt sein. Ansonsten resultieren zu wenige und zu lange Buckets, und die Existenzprüfung wird ineffizient.
- Die **GetHashCode()** - Methode muss extrem schnell sein und darf keine Ausnahme werfen.

Aus dem Hashcode einer Instanz und der Hashtabellen-Kapazität wird der Array-Index per Modulo-Operation ermittelt:

Array-Index = Hashcode % Kapazität

Um für eine Instanz mit der Methode **Contains()** festzustellen, ob sie bereits in der Hashtabelle bzw. in der Menge enthalten ist, muss die Instanz nicht über **Equals()** - Aufrufe mit allen Insassen verglichen werden. Stattdessen wird ihr Hashcode berechnet und der Array-Index ermittelt. Befindet sich hier noch keine Listenadresse, ist die Existenzfrage geklärt (**Contains()** - Rückmeldung **false**). Anderenfalls ist nur für die Instanzen der im Array-Element adressierten Liste eine **Equals()** - Untersuchung erforderlich.

Damit es selten zu Hash-Kollisionen kommt, sollte die Array-Größe ungefähr das 1,5 - fache der Anzahl aufzunehmender Elemente betragen (Horstmann & Cornell, 2002, S. 137). Die folgende **HashSet<T>** - Konstruktorüberladung (verfügbar ab .NET 5.0 und .NET - Framework 4.7.2) erlaubt eine Einstellung der initialen Größe:

public HashSet(int *capacity***)**

Es wird oft empfohlen, für die Kapazität einer Hashtabelle eine Primzahl zu wählen, sodass bei erwarteten 100 Elementen die Kapazität nicht auf 150, sondern auf 151 festzulegen ist. Nach Horstmann & Cornell (2002, S. 137) ist der Nutzen einer Primzahl als Kapazität nicht zweifelsfrei nachgewiesen. Weil kein Schaden auftreten kann, sollte man der Empfehlung trotzdem folgen.

Im Vergleich zur Aufbewahrung in einer **List<T>** - Kollektion nimmt man bei einer **HashSet<T>** - Kollektion einen erhöhten Speicherbedarf in Kauf. Es resultiert aber eine enorme Zeitersparnis, wenn in einem Algorithmus viele Existenzprüfungen erforderlich sind. In einem Testprogramm mit den Aufgaben

- eine Kollektion mit 20.000 **String**-Objekten füllen
- für 20.000 neue **String**-Objekte prüfen, ob sie bereits in der Kollektion vorhanden sind

---

[1] Weitere Anforderungen an eine **GetHashCode()** - Implementierung erläutert der C# - Insider Eric Lippert in seinem Blog:

https://ericlippert.com/2011/02/28/guidelines-and-rules-for-gethashcode/

Dort wird auch aufgeklärt, warum man sich beim CTS-Design (*Common Type System*) dafür entschieden hat, dass bereits die Urahnklasse **Object** die Methode **GetHashCode()** beherrschen soll:

> I think if we were redesigning the type system from scratch today, hashing might be done differently, perhaps with an IHashable interface. But when the CLR type system was designed there were no generic types and therefore a general-purpose hash table needed to be able to store any object.

zeigen die Klassen **List<String>**, **LinkedList<String>** und **HashSet<String>** sowie die im Abschnitt 11.4.2 behandelte Klasse **SortedSet<String>** die folgenden Leistungen:[1]

```
Kollektionsklasse:                    List`1
 Zeit zum Füllen:                      4,0 Millisek.
 Zeit für die Existenzprüfungen:       4530,1 Millisek.

Kollektionsklasse:                    LinkedList`1
 Zeit zum Füllen:                      5,0 Millisek.
 Zeit für die Existenzprüfungen:       4473,1 Millisek.

Kollektionsklasse:                    HashSet`1
 Zeit zum Füllen:                      6,0 Millisek.
 Zeit für die Existenzprüfungen:       4,0 Millisek.

Kollektionsklasse:                    SortedSet`1
 Zeit zum Füllen:                      42,0 Millisek.
 Zeit für die Existenzprüfungen:       43,1 Millisek.
```

Wie erwartet, ist die Klasse **HashSet<String>** bei den Existenzprüfungen den Klassen **List<String>** und **LinkedList<String>** drastisch überlegen.

Bei der Klasse **String** dürfen wir ohne Kontrolle eine angemessene Überschreibung der Methode **GetHashCode()** voraussetzen.

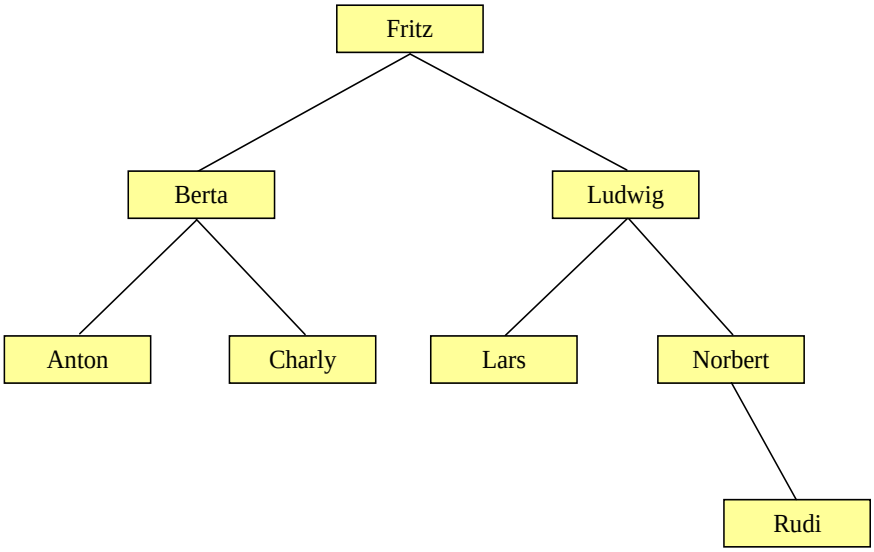
Die gleich vorzustellende Klasse **SortedSet<T>** ist wie **HashSet<T>** zur Verwaltung einer Menge konzipiert und implementiert daher ebenfalls das Interface **ISet<T>**. Sie hält aber ihre Elemente in einem sortierten Zustand und benötigt für diese Extraleistung Zeit. Bei den Existenzprüfungen ist aber auch **SortedSet<T>** den Klassen **List<String>** und **LinkedList<String>** deutlich überlegen.

11.4.2 Balancierte Binärbäume und die Klasse SortedSet<T>

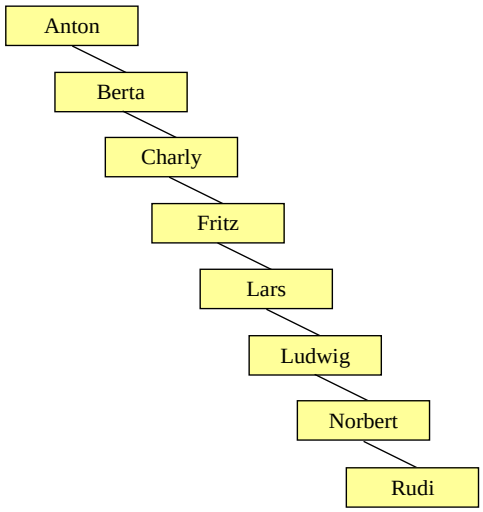
Existiert über den Elementen einer Menge eine vollständige Ordnung (z. B. bei Zeichenfolgen aufgrund der Sortierordnung), dann kann man über einen sogenannten *Binärbaum* die Elemente im sortierten Zustand halten, ohne den Aufwand bei den zentralen Mengenverwaltungsmethoden (z. B. **Add()**, **Contains()** und **Remove()**) zu groß werden zu lassen.

In einem Binärbaum hat jeder Knoten maximal zwei direkte Nachfolger, wobei der linke Nachfolger einen kleineren und der rechte Nachfolger einen höheren Rang hat, was die folgende Abbildung für Zeichenketten illustriert:

[1] Die Zeiten stammen von einem PC unter Windows 10 (64 Bit) mit Intel-CPU Core i3 (3,2 GHz). Ein Visual Studio - Projekt mit dem Testprogramm ist hier zu finden:
...\BspUeb\Kollektionen\ListSetContains



Bei einem **balancierten** Binärbaum kommen Forderungen zur maximalen Entfernung zwischen der Wurzel und einem Element hinzu, d.h. Forderungen zur Anzahl der Ebenen, um den Aufwand beim Suchen und Einfügen von Elementen zu begrenzen. Der bisher betrachtete Namensbaum ist balanciert (4 Ebenen), während in der folgenden Abbildung eine extrem unbalancierte Anordnung derselben Elemente zu sehen ist (8 Ebenen):



Zur Beurteilung des Aufwands bei der Suche nach einem Element (oder bei der Neuaufnahme eines Elements) gehen wir von einem balancierten und vollständig gefüllten Binärbaum aus. Hier haben alle Knoten, die keine Endknoten sind, genau zwei Nachfolger. In der ersten Variante des Namensbaums lag diese Situation vor der Aufnahme von Rudi vor. Der maximale Aufwand bei einer Existenzprüfung oder Neuaufnahme ist identisch mit der Zahl *m* der Ebenen, weil pro Ebene eine Identitätsprüfung vorgenommen werden muss. Wir schätzen nun ab, wie viele Ebenen ein balancierter Binärbaum zur Aufnahme von *k* Elementen benötigt.

Aus der Anzahl *m* der Ebenen kann nach der folgenden Formel die Anzahl *k* der enthaltenen Elemente berechnet werden:

$$k = 2^m - 1$$

Bei *m* = 3 Ebenen resultieren z. B. 7 Elemente (siehe Beispiel). Man erhält *k* als Partialsumme der geometrischen Reihe:[1]

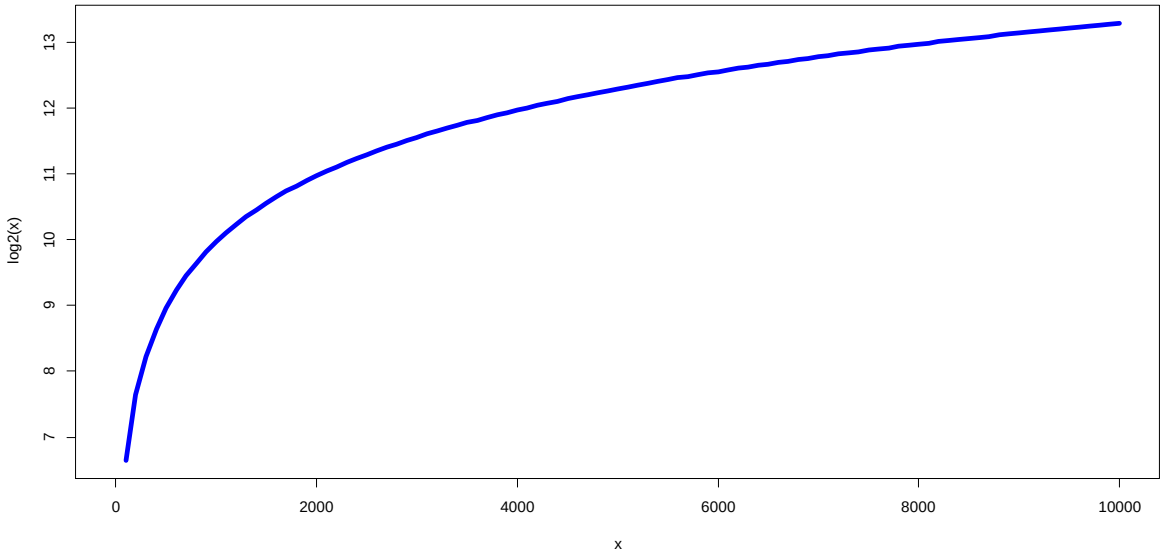
[1] Der mit elementaren Mitteln zu führende Beweis ist z. B. hier zu finden:

$$k = \sum_{i=0}^{m-1} 2^i = \frac{1-2^m}{1-2} = 2^m - 1$$

Für hinreichend großes *k* kann man die Beziehung zwischen *k* und *m* vereinfachen und dann durch Anwendung der Logarithmus-Funktion nach *m* auflösen, um die zur Verwaltung von *k* Elementen erforderliche Anzahl von Ebenen zu ermitteln:

$$\begin{aligned} k &= 2^m \\ \Leftrightarrow \quad \log_2(k) &= m \end{aligned}$$

Für hinreichend großes *k* sind also $\log_2(k)$ Ebenen erforderlich, und im ungünstigsten Fall werden ebenso viele Algorithmusschritte benötigt, um ein Element zu suchen oder die Position für ein neues Element zu bestimmen. Man sagt unter Verwendung einer Notation mit dem griechischen Großbuchstaben *O* (Omikron), der Algorithmus sei von der Ordnung $O(\log_2 k)$. Weil das monotone Wachstum der Logarithmus-Funktion relativ flach verläuft, steigt der Aufwand nur langsam mit der Anzahl *k* der Elemente an:[1]



Bei einer Hashtabelle wächst der Aufwand einer Existenzprüfung *nicht* mit der Anzahl der Elemente, und es resultiert die günstigere Ordnung $O(1)$. Bei einer Liste hingegen ist der Aufwand einer Existenzprüfung direkt proportional zur Anzahl der Elemente, und es resultiert die ungünstige Ordnung $O(k)$.

In der BCL nutzt die Klasse **SortedSet<T>** das Prinzip des balancierten Binärbaums, wobei durch die sogenannte **Rot-Schwarz** -Architektur sichergestellt wird, dass der Baum immer balanciert ist (Albahari & Johannsen 2020, S. 343). Damit verursacht die Anordnung der Elemente keine allzu großen Kosten für die Mengenoperationen.

Die Klassen **HashSet<T>** und **SortedSet<T>** implementieren im Wesentlichen dieselben Schnittstellen (siehe Abschnitt 11.4.1.1) und bieten damit ähnliche Funktionsumfänge. Als Gegenleistung für den höheren Aufwand bei Existenzprüfungen ($O(\log_2 k)$ statt $O(1)$) gewinnt man bei der Klasse **SortedSet<T>** die folgenden Zusatzfunktionen:

---

https://de.wikibooks.org/wiki/Mathe_f%C3%BCr_Nicht-Freaks:_Geometrische_Reihe

[1] Der Funktionsplot wurde mit R 3.6 erstellt über den Funktionsaufruf:

```
curve(log2(x), 0, 10000, col="blue", lwd=5)
```

- Die Elemente einer **SortedSet<T>** - Kollektion erscheinen beim Iterieren in der Sortierordnung, und der benannte Iterator **Reverse()** erlaubt das Iterieren in umgekehrter Richtung (siehe Abschnitt 9.7.2 zu benannten Iteratoren).
- **SortedSet<T>** besitzt die Eigenschaften **Min** und **Max**, die das kleinste bzw. größte Element in der Menge liefern:
  - **public T Min { get; }**
  - **public T Max { get; }**

Einen Indexer, der bei **HashSet<T>** aus naheliegenden Gründen fehlt, bietet auch **SortedSet<T>** nicht an, sodass kein Elementzugriff per **[]** - Operator möglich ist.

11.5 Verwaltung von (Schlüssel-Wert) - Paaren

Zur Verwaltung einer Menge von (Schlüssel-Wert) - Paaren ist in der BCL die generische Klasse **Dictionary<K, V>** vorhanden. Die Schlüssel (mit einer Konkretisierung des Typformalparameters **K** als Datentyp) werden wie eine Menge verwaltet, sodass Eindeutigkeit garantiert ist (ohne Dubletten), aber keine relevante Anordnung besteht. Über einen Schlüssel ist der zugehörige Wert ansprechbar (mit einer Konkretisierung des Typformalparameters **V** als Datentyp), wobei der Zugriff dank Hashtabellen-Technik sehr schnell erfolgt. Bei der Konkretisierung des Typformalparameters **K** müssen in der Regel die von der Urahnklasse **Object** geerbten Methoden **Equals()** und **GetHashCode()** überschrieben werden, damit Dubletten verhindert werden und Zugriffe effizient stattfinden (vgl. Abschnitt 11.4.1.2).

Neben **List<T>** ist **Dictionary<K, V>** wohl die am häufigsten verwendete BCL-Kollektionsklasse (Albahari & Johannsen 2020, S. 347). Man könnte z. B. den lokalen Zwischenspeicher für eine Personalverwaltungsdatenbank realisieren mit ...

- einer eindeutigen Personalnummer (Typ **int** als **K**-Konkretisierung)
- und einer geeigneten Klasse `Personal` (mit Instanzvariablen für den Namen, die Telefonnummer etc.) als **V**- Konkretisierung.

Ein Objekt der Klasse **Dictionary<K, V>** beherrscht u. a. die folgenden Instanzmethoden:

- **public void Add(K** *key*, **V** *value***)**
  Falls der Schlüssel noch nicht existiert, wird ein neues (Schlüssel-Wert) - Paar aufgenommen. Ansonsten wirft die Methode eine Ausnahme vom Typ **ArgumentException**.
- **public bool ContainsKey(K** *key***)**
  Diese Methode informiert darüber, ob ein Element mit dem fraglichen Schlüssel vorhanden ist, und wird sehr schnell ausgeführt.
- **public bool ContainsValue(V** *value***)**
  Diese Methode informiert darüber, ob ein Element mit dem fraglichen Wert vorhanden ist, und nimmt viel Zeit in Anspruch.
- **public bool TryGetValue(K** *key***, out V** *value***)**
  Wenn ein Element mit dem angegebenen Schlüssel vorhanden ist, wird sein Wert in den **out**-Parameter *value* geschrieben und die Rückgabe **true** geliefert. Ist der Schlüssel *nicht* vorhanden, wird die Rückgabe **false** geliefert und der typspezifische Nullwert in den **out**-Parameter geschrieben. Die Rückgabe **false** zu ignorieren und z. B. beim **V**-Typ **int** eine erhaltene 0 als Information über ein Kollektionselement zu interpretieren, ist ein gravierender Programmierfehler.

- **public bool Remove(K *key*)**
  Das Element mit dem angegebenen Schlüssel wird aus der Kollektion entfernt, falls der Schlüssel vorhanden ist. Über den Rückgabewert erfährt man, ob die Kollektion durch den Aufruf geändert worden ist.
- **public void Clear()**
  Es werden alle Elemente entfernt.

Weil die Konkretisierungen der generischen Klasse **Dictionary<K, V>** über einen Indexer verfügen, kann man über einen in eckige Klammern eingeschlossenen Schlüssel den zugehörigen Wert ermitteln und festlegen. Es ist sogar möglich, per Indexer ein (Schlüssel-Wert) - Paar einzufügen, z. B.:

```
var fred = new Dictionary<char, int>();
fred['c'] = 1;
Console.WriteLine(fred['c']);
```

Für die Klasse **Dictionary<K, V>** sind zwei Kollektionsinitialisierer-Varianten verfügbar:

- Bei dieser Variante findet ein impliziter Aufruf des Indexers durch den Compiler statt:
```
var fred = new Dictionary<char, int> {
    ['c'] = 1, ['b'] = 3
};
```
- Bei dieser Variante findet ein impliziter Aufruf der Methode **Add(K, V)** durch den Compiler statt:
```
var fred = new Dictionary<char, int> {
    {'c', 1}, {'b', 3}
};
```

Durchläuft man eine **Dictionary<K, V>** - Kollektion per **foreach** - Schleife, ist als Elementtyp die passend konkretisierte generische Struktur **KeyValuePair<K, V>** im Einsatz, z. B.:

```
foreach (KeyValuePair<char, int> kvp in fred)
   Console.WriteLine($"{kvp.Key} : {kvp.Value}");
```

Dank der Fähigkeit des Compilers zur Typinferenz kann man sich aber die explizite Angabe sparen, z. B.:

```
foreach (var kvp in fred)
   Console.WriteLine($"{kvp.Key} : {kvp.Value}");
```

Über die Eigenschaften **Key** bzw. **Value** erhält man den Schlüssel bzw. den Wert einer Instanz.

Außerdem existieren noch die benannten Iteratoren **Keys** und **Values** mit Eigenschaftssyntax (vgl. Abschnitt 9.7.2), die ein Iterieren über die Schlüssel bzw. über die Werte in der Kollektion erlauben:

```
foreach (char k in fred.Keys)
   Console.WriteLine(k);

foreach (int i in fred.Values)
   Console.WriteLine(i);
```

Das folgende Programm verwendet ein **Dictionary<char, int>** - Objekt dazu, um für einen **String** die Häufigkeiten der enthaltenen Zeichen zu ermitteln:

Quellcode	Ausgabe

```
using System;
using System.Collections.Generic;

class DictionaryDemo {
   static void CountLetters(String text) {
      var fred = new Dictionary<char, int>();
      foreach (char c in text)
         if (fred.ContainsKey(c)) {
            fred[c]++;
         }
         else
            fred.Add(c, 1);
      foreach (KeyValuePair<char, int> kvp in fred)
         Console.WriteLine($"{kvp.Key} : {kvp.Value}");
   }

   static void Main() {
      CountLetters("Otto spielt Lotto.");
   }
}
```

```
O : 1
t : 5
o : 3
  : 2
s : 1
p : 1
i : 1
e : 1
l : 1
L : 1
. : 1
```

Um eine nach den Schlüsseln *sortierte* Tabelle von (Schlüssel-Wert) - Paaren zu erhalten, ersetzt man die Klasse **Dictionary<K, V>** durch die Alternative **SortedDictionary<K, V>**. Wie von der Klasse **SortedSet<T>** wird zur performanten Realisation einer sortierten Menge auch von der Klasse **SortedDictionary<K, V>** das Prinzip des balancierten Binärbaums mit Rot-Schwarz - Architektur verwendet (Albahari & Johannsen 2020, S. 349).

11.6 Übungsaufgaben zum Kapitel 11

1) Welche von den folgenden Aussagen sind richtig bzw. falsch?

1. Ein Objekt aus der Kollektionsklasse **List<T>** oder **LinkedList<T>** hält seine Elemente in sortiertem Zustand.
2. Die Klasse **HashSet<T>** bietet die beste Leistung bei Existenzprüfungen.
3. Die Klasse **List<T>** beherrscht zum Suchen und Sortieren dieselben Methoden wie ein Array.
4. Die zur Verwaltung von (Schlüssel-Wert) - Elementen dienende Klasse **Dictionary<K, V>** glänzt darin, dass zu einem Schlüssel sehr schnell der zugehörige Wert ermittelt werden kann.
5. Die Kollektionsklassen **SortedList<T>** und **SortedDictionary<K, V>** halten ihre Elemente in sortiertem Zustand.

2) Erstellen Sie eine Variante der im Abschnitt 5.8 vorgestellten Personenverwaltung, indem Sie die Klasse `PersonDB` (Eigenbau einer verketteten Liste mit Indexer) durch die generische Kollektionsklasse **List<T>** ersetzten.

12 Einstieg in die GUI-Programmierung mit WPF-Technik

Mit den Eigenschaften und Vorteilen einer grafischen Benutzeroberfläche (engl.: ***G**raphical **U**ser **I**nterface*) sind Sie sicher sehr gut vertraut. Eine GUI-Anwendung präsentiert dem Anwender standardisierte Bedienelemente zur Datenpräsentation und Benutzerinteraktion, z. B.:

- Texteingabefelder
- Befehlsschalter
- Kontrollkästchen und Optionsfelder
- Schieberegler und Auswahllisten
- Baumansichten für Dokumentenstrukturen etc.
- Tabellen zur Datenpräsentation
- Menüs
- Fortschrittsbalken
- Komponenten zur Präsentation von bildlichen und audio-visuellen Medien

Die von einer GUI-Bibliothek zur Verfügung gestellten Bedienelemente bezeichnet man als *Steuerelemente*, *controls* oder *widgets* (Wortkombination aus *window* und *gadgets*).

Von standardisierten Bedienelementen profitieren Entwickler *und* Anwender:

- Entwickler können dank fertiger und dabei auch noch flexibel konfigurierbarer Komponenten die Bedienoberfläche einer Anwendung zügig aufbauen. Für eine weitere RAD-Beschleunigung (***R**apid **A**pplication **D**evelopment*) sorgen grafische GUI-Designer (z. B. WPF-Designer im Visual Studio, den wir schon mehrfach benutzt haben).
- Weil die Steuerelemente intuitiv (z. B. per Maus oder Finger) und in verschiedenen Programmen weitgehend konsistent zu bedienen sind, erleichtern sie dem Anwender den Umgang mit Software.

Die in der umfangreichen WPF-Bibliothek (***W**indows **P**resentation **F**oundation*) enthaltenen Standardkomponenten erlauben durch ihre Vielfalt, ihre Konfigurierbarkeit und durch die Möglichkeiten zur flexiblen (z. B. hierarchisch strukturierten) Anordnung die Erstellung von individuellen und ergonomischen Bedienoberflächen für sehr viele Anwendungen. Bei manchen Programmen genügen die Standardkomponenten aber *nicht* für die spezielle Präsentation und/oder Bearbeitung von zwei- oder dreidimensionalen Daten (z. B. bei einem Editor für statistische Diagramme), und es wird eine individuelle Grafikprogrammierung erforderlich. Wir beschränken uns in diesem Kapitel auf einen Einstieg in die Erstellung von Bedienoberflächen mit Hilfe von Standardkomponenten aus der WPF-Bibliothek.

12.1 Einordnung

12.1.1 GUI-Technologien der .NET-Plattform

Die *Windows Presentation Foundation* ist ein 2006 als .NET - Bestandteil eingeführtes GUI-Framework, das sich gegenüber seinem Vorgänger *WinForms* durchgesetzt hat und aktuell (Frühjahr 2021) bei der Entwicklung von Desktop-Anwendungen für Windows immer noch eine gute Wahl ist.[1]

Das neueste Angebot von Microsoft zur Desktop-Programmierung ist die *Universelle Windows-Plattform* (UWP), die auf der sogenannten *Windows-Runtime* (WinRT) basiert, aber trotzdem mit

[1] In einer früheren Version dieses Manuskripts (Baltes-Götz 2010) ist eine Behandlung der WinForms - GUI-Technik zu finden, siehe https://www.uni-trier.de/index.php?id=30454

C# und anderen .NET - Sprachen genutzt werden kann.[1] UWP-Apps hatten bei ihrer Einführung wichtige Alleinstellungsmerkmale:

- Vertrieb über den Windows-Store
- Unterstützung verschiedener Gerätegattungen mit Windows 10 als Betriebssystem (PC, Smartphone, Xbox, HoloLens etc.)[2]

Nachdem die in der Liste unterstützter UWP-Geräte lange Zeit in den Vordergrund gestellten Smartphones unter Windows vom Markt verschwunden sind, zeigt Microsoft nur noch wenig Begeisterung für die UWP, und für neue Anwendungen ist die UWP keine empfehlenswerte Alternative zur WPF.[3]

Außerdem verwenden derzeit noch mehr ca. 20% der Desktop-Computer unter Windows die Version 7 oder 8.1.[4] Wenn eine Software für Desktop-Computer *alle* auf dem Markt befindlichen Windows-Versionen unterstützen soll, führt derzeit kein Weg an der WPF vorbei.

WPF (und WinForms) werden auch in .NET 5 und in .NET Core ab Version 3.0 unterstützt, wenn auch nur unter Windows. Außerdem ist es möglich, in einer WPF-Anwendung die Windows 10 - APIs zu nutzen.[5] Wer keine Software für die Spielekonsole Xbox oder die Augmented-Reality - Brille HoloLens plant, sondern nur Desktop-Computer unter Windows versorgen möchte, fährt mit WPF-Anwendungen also nicht schlecht. Auch der Vertrieb von WPF-Apps über den Windows Store ist längst kein Problem mehr.[6]

Wer doch von der WPF auf die UWP umsteigt, kann wesentliche Teile des erworbenen GUI - Know-Hows weiterverwenden. Z. B. lässt sich auch in einer UWP-App die Bedienoberfläche mit dem XML-Dialekt XAML (*e**X**tensible **A**pplication **M**arkup **L**anguage*, siehe Abschnitt 12.3) deklarieren, wenn auch mit leicht geänderter Syntax (Schacherl 2014, Abschnitt 1.3.7).

Für die nahe Zukunft (Ende 2021, zusammen mit .NET 6) verspricht Microsoft, durch die Integration von **Xamarin.Forms** (einer Bedienoberfläche für verschiedene Mobilplattformen) in .NET unter dem Namen **MAUI** (*Multi-platform App UI*) eine gemeinsame grafische Bedienoberfläche für Windows, macOS, iOS und Android zu schaffen.[7] Weil das MAUI wie die WPF zur Oberflächendeklaration auf die XAML setzt, werden die im aktuellen Kapitel erworbenen Kenntnisse auch für MAUI - Anwendungen nützlich sein.

Lokal installierte WPF-Anwendungen für Desktop-Rechner unter Windows stehen durch das Aufkommen von Internet-basierten Anwendungen unter zunehmendem Konkurrenzdruck, haben aber immer noch viele Vorteile, z. B.:

---

1 https://www.it-visions.de/glossar/alle/6241/Windows_Runtime.aspx

2 https://docs.microsoft.com/de-de/windows/uwp/get-started/universal-application-platform-guide

3 https://docs.microsoft.com/en-us/answers/questions/6911/is-there-a-future-for-uwp.html

4 https://gs.statcounter.com/windows-version-market-share/desktop/worldwide/

5 https://blogs.windows.com/buildingapps/2017/01/25/calling-windows-10-apis-desktop-application/

6 https://docs.microsoft.com/en-us/archive/msdn-magazine/2013/february/windows-store-apps-a-guide-for-wpf-and-silverlight-developers-part-1

7 https://github.com/dotnet/maui,
https://www.heise.de/developer/meldung/Build-2020-Aus-Xamarin-Forms-wird-MAUI-4724947.html

- Uneingeschränkte Nutzung der PC-Hardware
- Überlegenheit beim wichtigsten Ergonomiekriterium: **Reaktionszeit**
- Benutzerfreundliche, vielfältige Bedienelemente
- Unabhängigkeit vom Internet

Wir werden uns im Kurs auf WPF-Desktop-Anwendungen beschränken, und können dieses Thema, dem umfangreiche Monographien gewidmet sind (z. B. MacDonald 2012, Nathan 2010, Wegener 2013), nicht annähernd erschöpfend behandeln.

12.1.2 Vergleich zwischen GUI- und Konsolenanwendungen

Im Vergleich zu Konsolenprogrammen geht es bei GUI-Anwendungen nicht nur intuitiver, sondern vor allem auch ereignisreicher[1] und mit mehr Mitspracherechten für den Anwender zu. Ein Konsolenprogramm entscheidet selbst darüber, welche Anweisung als nächstes ausgeführt wird, und wann der Benutzer eine Eingabe machen darf. Um seine Aufgaben zu erledigen, verwendet ein Konsolenprogramm diverse Dienste des Laufzeitsystems, z. B. bei der Aus- oder Eingabe von Zeichen.

Für den Ablauf einer Applikation mit grafischer Benutzeroberfläche ist ein **ereignisorientiertes und benutzergesteuertes Paradigma** wesentlich, wobei das Laufzeitsystem als Vermittler oder (seltener) als Quelle von Ereignissen in erheblichem Maße den Ablauf mitbestimmt, indem es Methoden der GUI-Applikation aufruft, z. B. zum Zeichnen von Fensterinhalten. Ausgelöst werden die Ereignisse in der Regel vom Benutzer, der mit der Hilfe von Eingabegeräten wie Maus, Tastatur, Touchscreen etc. praktisch permanent in der Lage ist, unterschiedliche Wünsche zu artikulieren. Ein GUI-Programm präsentiert mehr oder weniger viele Bedienelemente, die dem Anwender das Auslösen von Ereignissen ermöglichen. Das Programm wartet die meiste Zeit darauf, auf ein vom **Benutzer** ausgelöstes **Ereignis** mit einer vorbereiteten Ereignisbehandlungsmethode zu reagieren.

Im Vergleich zu einem Konsolenprogramm ist bei einem GUI-Programm die dominante Richtung im Kontrollfluss zwischen Anwendung und Laufzeitsystem invertiert. Die Ereignisbehandlungsmethoden einer GUI-Anwendung sind Beispiele für sogenannte *Call Back - Routinen*. Man spricht auch vom *Hollywood-Prinzip*, weil in dieser Gegend oft nach der Devise kommuniziert wird: „*Don't call us. We call you*".

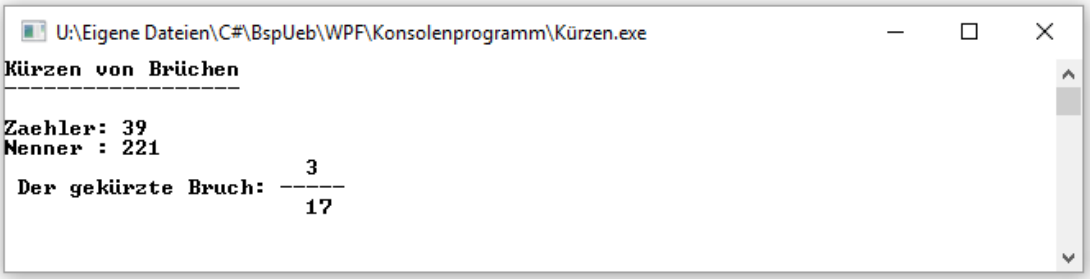
Während sich ein Konsolenprogramm gegenüber dem Anwender autoritär und gegenüber dem Laufzeitsystem fordernd verhält, präsentiert ein GUI-Programm dem Anwender Service-Angebote und befolgt die Anweisungen des Laufzeitsystems:

- Eine Konsolenanwendung diktiert den Ablauf und erlaubt dem Benutzer gelegentlich eine Eingabe. Um seinen Job erledigen zu können, verlangt das Programm Dienstleistungen vom Laufzeitsystem, z. B.: „Bitte den nächsten Tastendruck übermitteln." Das Laufzeitsystem erledigt solche Anforderungen und gibt die Kontrolle dann wieder an die Konsolenanwendung zurück. Eine Konsolenanwendung benimmt sich so, als wenn sie das einzige Anwendungsprogramm wäre und das Laufzeitsystem als Dienstleister zur Verfügung hätte.
- Eine GUI-Anwendung besteht hingegen aus einer Sammlung von Ereignisbehandlungsmethoden, wobei die zugehörigen Ereignisse vom Benutzer ausgelöst werden, indem er eines der zahlreichen Bedienelemente benutzt. Die Ereignisse werden zunächst vom Laufzeitsystem registriert, das daraufhin Methoden des GUI-Programms aufruft.

---

[1] Momentan wird bewusst ein starker Kontrast zwischen den bisher überwiegend benutzten Konsolenanwendungen und den nun vorzustellenden GUI-Anwendungen hinsichtlich der ereignisorientierten Programmierung herausgearbeitet. Allerdings kann grundsätzlich auch eine .NET - Konsolenanwendung mit Ereignissen umgehen. Wir werden z. B. im Abschnitt 16.6.3 ein Konsolenprogramm erstellen, das auf Ereignisse im Dateisystem (z. B. auf das Erstellen, Umbenennen oder Löschen von Dateien) reagiert.

Betrachten wir zur Illustration eine Konsolen- und eine GUI-Anwendung zum Kürzen von Brüchen. Bei der Konsolenanwendung

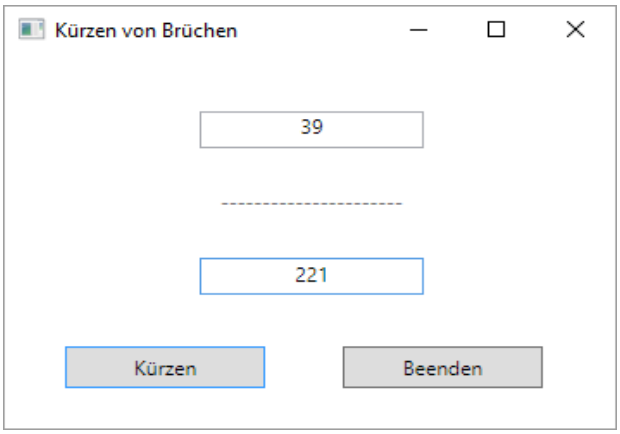


wird der gesamte Ablauf vom Programm diktiert:

- Es fragt nach dem Zähler.
- Es fragt nach dem Nenner.
- Es schreibt das Ergebnis auf die Konsole.

Wenn der Benutzer z. B. nach der Eingabe des Nenners zum ersten Bruch den Zähler dieses Bruchs noch einmal ändern möchte, dann muss er das Programm beenden und neu starten.

Im Unterschied zu diesem **programmgesteuerten Ablauf** wird bei der (im Abschnitt 5.12 erstellten) GUI-Variante



das Geschehen vom Benutzer diktiert, der die vier Bedienelemente (zwei Texteingabefelder und zwei Schaltflächen) in beliebiger Reihenfolge verwenden kann, wobei das Programm mit seinen Ereignisbehandlungsmethoden reagiert (**benutzergesteuerter Ablauf**).

12.1.3 WPF-Optionen und -Merkmale

Grundsätzlich ist das Erstellen einer GUI-Anwendung für Windows mit erheblichem Aufwand verbunden. Allerdings bietet die WPF-Technik außerordentlich leistungsfähige Klassen und Unterstützungsleistungen zur GUI-Programmierung, deren Verwendung durch Hilfsmittel der Entwicklungsumgebungen (z. B. Fensterdesigner) zusätzlich erleichtert wird.

Als WPF - Vorteile sind u. a. zu nennen:

- Die Oberflächengestaltung und die Codierung der Programmlogik werden getrennt durch die Möglichkeit, die Benutzeroberfläche in einem XML-Dialekt namens *XAML* (*eXtensible Application Markup Language*, siehe Abschnitt 12.3) zu deklarieren. Dadurch können Software-Entwickler und Grafikdesigner bei großen Projekten gut zusammenarbeiten.
- Gute Verbindungen zwischen Datenbeständen und GUI-Komponenten
- Attraktive 2D- und 3D-Grafik auf vektorieller Basis (ohne Qualitätsverlust beim Skalieren)

- Einheitliche Behandlung von Steuer- und Grafikelementen (z. B. Mausklickereignisse bei Grafikelementen)
- Hardware-beschleunigte Grafikausgabe
- Multimediale Vielfalt (Audio/Video) und Animationen

Beginnend im Jahr 2007 hat Microsoft ca. 10 Jahre lang versucht, **Browser-Anwendungen mit WPF-Technik** zu etablieren. Diese werden ...

- bei jedem Einsatz aus dem Internet bezogen
- und im Fenster eines WWW-Browsers ausgeführt, der ein Plugin mit der Microsoft-Technik **Silverlight** bereithalten muss.

So wie andere proprietäre (herstellergebundene) klientenseitige Web-Techniken (z. B. Flash) ist Silverlight jedoch durch HTML 5, das multimediale Anwendungen ohne Browser-Plugin ermöglicht, weitgehend verdrängt worden. Man sollte heutzutage nicht mehr damit rechnen, dass der Internet-Browser potentieller Benutzer Silverlight unterstützt. In einem kontrollierbaren Umfeld (z. B. unternehmensintern) ist Silverlight (bis zum Support-Ende im Oktober 2021) eventuell noch eine relevante Option, um die .NET - Software-Entwicklung mit der Internet-basierten Softwareverteilung zu kombinieren. Allerdings ist davon abzuraten, noch ein neues Projekt mit Silverlight zu beginnen. Im Kurs werden daher Web-Anwendungen mit der WPF-Technik *nicht* behandelt.

Wie die meisten modernen GUI-Frameworks ist auch die WPF aus Konsistenz- und Performanzgründen nach dem **Single-Thread-Prinzip** konzipiert. Daher muss der Zugriff auf die GUI-Komponenten dem UI-Thread vorbehalten bleiben. Andererseits muss sich bei allen im UI-Thread ausgeführten Methoden der Zeitaufwand in Grenzen halten (maximal 100 Millisekunden), weil sonst die Bedienoberfläche zäh reagiert. Solange eine Methode läuft (z. B. gestartet als Reaktion auf ein Ereignis), kann die Anwendung nicht auf andere Ereignisse (z. B. Mausklicks auf Steuerelemente) reagieren. Auch ist keine Aktualisierung der Anzeige möglich, zu der z. B. das Laufzeitsystem auffordert, weil ein bisher verdeckter Fensterbereich sichtbar geworden ist. Zeitaufwändige Arbeiten (z. B. Netzwerk-, Datei oder Datenbankzugriffe, aufwändige Berechnungen) gehören also in einen Arbeits-Thread. In der Regel müssen aber irgendwann Ergebnisse der Hintergrundtätigkeit an der Oberfläche sichtbar werden, wobei wegen der eingangs genannten Regel aus dem Arbeits-Thread keine direkten Zugriffe auf Steuerelemente möglich sind.

Selbstverständlich gibt es für die gerade skizzierte, höchst alltägliche Aufgabenstellung eine Routinelösung, die im Abschnitt 17.2.3 vorgestellt wird. Im aktuellen Kapitel gehen wir dem Thema *Multithreading* noch aus dem Weg, weil genügend andere Herausforderungen zu bewältigen sind.

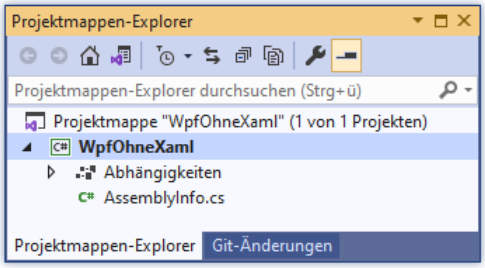
12.2 Elementare Bausteine einer WPF-Anwendung

12.2.1 Eine minimalistische WPF-Anwendung ohne XAML

Um die elementaren Klassen und Abläufe bei einer WPF-Anwendung zu studieren, vermeiden wir zunächst den Entstehungsprozess mit XAML-Beteiligung und die damit verbundenen, komplexen Hintergrundaktivitäten und betrachten stattdessen ein sehr einfaches, komplett durch C# - Anweisungen realisiertes Beispielprogramm. Wir legen im Visual Studio ein neues Projekt mit dem Namen `WpfOhneXaml` an, das die bei einer WPF-Anwendung erforderlichen Abhängigkeiten enthält, aber keinerlei Quellcode- oder XAML-Dateien. Dazu verwenden wir die (für .NET Core 3.x bzw. .NET 5 konzipierte) Vorlage **WPF-App (.NET)** und entfernen mit dem Projektmappen-Explorer die automatisch generierten XAML-Bestandteile:

- **App.xaml** (samt der zugehörigen Quellcodedatei **App.xaml.cs**) und
- **MainWindow.xaml** (samt der zugehörigen Quellcodedatei **MainWindow.xaml.cs**)

Anschließend sollte der **Projektmappen-Explorer** das folgende Bild zeigen:



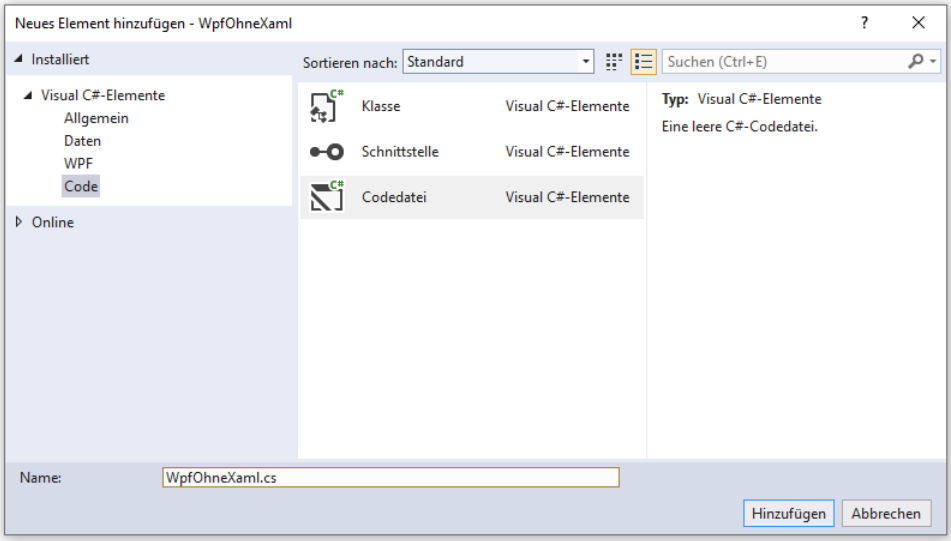
Wie das mit

Projekt > Eigenschaften

geöffnete Fenster zeigt, entsteht ein Anwendungs-Assembly mit ...

- dem **Zielframework** .NET Core 3.1
  Eine Änderung auf .NET 5 ist möglich, aber ohne Relevanz für den weiteren Verlauf des Abschnitts.
- dem Ausgabetyp Windows, sodass beim Starten kein Konsolenfenster erscheint (vgl. Abschnitt 3.3.6.2).

Wir legen über den Kontextmenübefehl **Hinzufügen > Neues Element** zum Projekt eine neue **Codedatei** namens **WpfOhneXaml.cs** an



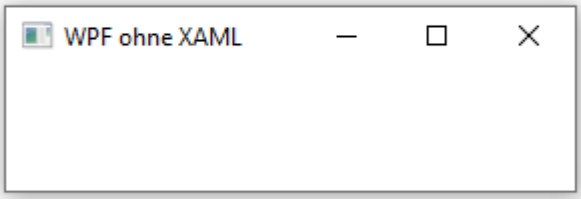
und definieren dort die Klasse `WpfOhneXaml` mit der Basisklasse **Window** aus dem Namensraum **System.Windows**:

```
using System.Windows;
class WpfOhneXaml : Window {
   WpfOhneXaml() {
      Title = "WPF ohne XAML";
      Width = 300;
      Height = 100;
   }

   [System.STAThread]
   static void Main() {
      Application app = new Application();
      WpfOhneXaml hf = new WpfOhneXaml();
      app.Run(hf);
   }
}
```

Auch ein GUI-Programm besteht aus Klassen, wobei eine Startklasse mit einer statischen **Main()** - Methode vorhanden sein muss (vgl. Abschnitt 1.5). Beim Programmstart wird die Startklasse vom Laufzeitsystem aufgefordert, ihre **Main()** - Methode auszuführen. Ein Hauptzweck dieser Methode besteht darin, Objekte zu erzeugen und somit Leben auf die objektorientierte Bühne zu bringen. Gleich wird erläutert, wie die zentralen Klassen des WPF-Frameworks dabei mitwirken.

Bei einer WPF-Anwendung muss die Methode **Main()** mit dem Attribut **System.STAThread** dekoriert werden.[1] Attribute sind Objekte, die mit einer speziellen Syntax (siehe Beispiel) an Klassen, Methoden etc. geheftet werden, um Informationen für den Compiler und/oder die CLR bereitzustellen. Nähere Informationen folgen im Kapitel 14. Im konkreten Fall handelt es sich um eine Deklaration gegenüber dem Windows-Betriebssystem zum Modus der Interprozesskommunikation per COM (***C**omponent **O**bject **M**odel*).[2] Eine WPF-Anwendung benötigt die COM-Interoperabilität z. B. beim Zugriff auf die Windows-Zwischenablage.

Ein allzu großer Funktionsumfang ist vom Beispielprogramm natürlich nicht zu erwarten:



Immerhin kann man das Anwendungsfenster (dank Windows und .NET) verschieben, seine Größe ändern, die Titelzeilen-Standardschaltflächen zum Minimieren, Maximieren und Beenden benutzen usw.

Im aktuellen Kapitel 12 werden Sie folgende Klassen als wichtige Bestandteile einer WPF-Anwendung kennenlernen:

- **Window** (Namensraum **System.Windows**)
  Von dieser Klasse stammen alle Anwendungs- oder Dialogfenster ab.
- **Application** (Namensraum **System.Windows**)
  Diese Klasse stellt für eine WPF-Anwendung wichtige Dienste bereit. In der Regel definiert man eine eigene **Application**-Ableitung. Ein Objekt dieser Klasse repräsentiert die Anwendung und wird daher im Manuskript als *Anwendungsobjekt* bezeichnet.
- Klassen für Steuerelemente (z. B. **Label**, **Button**, **TextBox**) und Layoutcontainer zur Verwaltung von Steuerelementen (z. B. **Grid**, **StackPanel**)
- **RoutedEvent**
  Für WPF-Anwendungen wurde mit den sogenannten *Routingereignissen* eine Ergänzung bzw. Erweiterung der CLR-Ereignistechnik eingeführt. Ein Routingereignis basiert auf einem Objekt der Klasse **RoutedEvent**.

Im weiteren Verlauf von Abschnitt 12.2 werden wir uns mit den Klassen **Window** und **Application** beschäftigen.

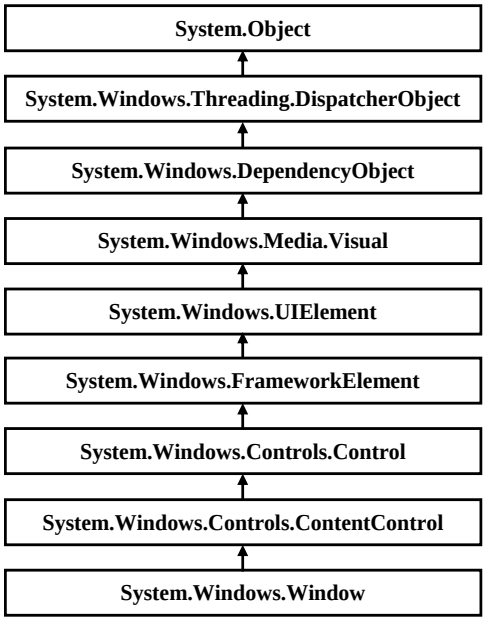
---

[1] Eigentlich heißt die Klasse **STAThreadAttribute**, doch man darf *Attribute* weglassen, was in der Regel auch geschieht.

[2] https://docs.microsoft.com/en-us/windows/desktop/com/single-threaded-apartments

12.2.2 (Haupt)fenster und die Klasse Window

Alle Fenster der im Manuskript auftauchenden WPF-Anwendungen werden über Objekte einer von **System.Windows.Window** abstammenden Klasse verwaltet. **Window** erbt seine Funktionalität wiederum zum großen Teil von allgemeineren Klassen:



In diesem Stammbaum tauchen viele fundamentale WPF-Klassen auf, sodass es nicht nur im Hinblick auf die Klasse **Window** sinnvoll ist, die Basisklassen mit Ihren Zuständigkeiten kurz zu erläutern:[1]

- **System.Windows.Threading.DispatcherObject**
  Um eine konsistente und verzögerungsfrei reagierende Bedienoberfläche zu garantieren, verwendet das WPF-Framework die Single-Thread - Architektur: Auf ein UI-Element darf nur derjenige Thread zugreifen, der das Element erzeugt hat. Um dies sicherzustellen, stammen die meisten WPF-Klassen (z. B. zur Verwaltung von Steuerelementen) von der Klasse **DispatcherObject** ab und sind daher dem **Dispatcher** des erzeugenden Threads (einem Objekt der Klasse **System.Windows.Threading.Dispatcher**) fest zugeordnet. Der **Dispatcher** ist in einem Thread für die Verwaltung der Warteschlange mit zu erledigenden Aufgaben (Methodenaufrufen) zuständig. Eine korrekt konzipierte WPF-Methode überprüft vor einem Zugriff auf ein **DispatcherObject**-Objekt, ob sie in demjenigen Thread ausgeführt wird, der das **DispatcherObject**-Objekt erstellt hat. Dazu steht in der Klasse **DispatcherObject** die Methode **VerifyAccess()** zur Verfügung (siehe Beispiele im Abschnitt 10.2.1). Ist die Thread-Bedingung verletzt, wirft **VerifyAccess()** eine **InvalidOperationException**. Dieser Ausnahmefehler ist in WPF- Programmen nicht selten zu sehen. Aufrufe von **VerifyAccess()** kommen dann in Frage, wenn man eine eigene **DispatcherObject**-Ableitung definiert (z. B. für ein eigenes Steuerelement). In den Methoden dieser Klasse müssen ihre Objekte vor illegitimen Zugriffen geschützt werden.

[1] Weitere Details bietet die Webseite:
https://docs.microsoft.com/de-de/dotnet/framework/wpf/advanced/wpf-architecture

- **System.Windows.DependencyObject**
  Objekte dieser Klasse können *Abhängigkeitseigenschaften* (engl.: *dependency properties*) anbieten. Diese spielen im WPF-Framework eine zentrale Rolle und haben u. a. die folgenden Vorteile im Vergleich zu gewöhnlichen CLR-Eigenschaften:
  - Die Ausprägung einer Abhängigkeitseigenschaft kann durch verschiedene Quellen beeinflusst werden (z. B. durch Stile, elterliche Steuerelemente).
  - Ausprägungen können an andere Abhängigkeitseigenschaften gemeldet werden.

  Die meisten Eigenschaften von WPF-Steuerelementen sind als Abhängigkeitseigenschaften realisiert, können aber meist durch eine Hüllenkonstruktion auch wie gewöhnliche CLR-Eigenschaften verwendet werden. Durch die Abhängigkeitseigenschaften wird im Vergleich zu gewöhnlichen CLR-Eigenschaften nicht nur eine gesteigerte Funktionalität ermöglicht, sondern auch in erheblichem Umfang Speicherplatz eingespart (Nathan 2010, S. 83).
- **System.Windows.Media.Visual**
  Objekte dieser Klasse besitzen elementare Grafikkompetenzen (z. B. Ausgabe auf dem Bildschirm, Trefferdiagnose bei Mausklicks).
- **System.Windows.UIElement**
  Diese Klasse steuert u. a. Kompetenzen für die Layout-, Ereignis- und Fokusbehandlung bei.
- **System.Windows.FrameworkElement**
  Diese Klasse ist bei vielen WPF-Techniken beteiligt, z. B. Layout, Datenbindung, Stile und Animationen.
- **System.Windows.Controls.Control**
  Diese Klasse, von der viele konkrete Steuerelementklassen (z. B. **Button**, **TextBox**) abstammen, steuert die Fähigkeit zur Interaktion mit dem Benutzer bei. Außerdem unterstützt sie **ControlTemplate**-Objekte mit Einstellungspaketen zur Oberflächengestaltung.
- **System.Windows.Controls.ContentControl**
  Die von **System.Windows.Controls.ContentControl** abstammenden Klassen beherrschen das WPF-Inhaltsmodell, können also untergeordnete Elemente aufnehmen.

Das obige Beispielprogramm besteht aus der von **Window** abgeleiteten Klasse `WpfOhneXaml`

```
class WpfOhneXaml : Window
```

und erzeugt in seiner **Main()** - Methode *ein* Objekt aus dieser Klasse (also ein entsprechendes Fenster):

```
WpfOhneXaml hf = new WpfOhneXaml();
```

Als Aktualparameter im Methodenaufruf

```
app.Run(hf);
```

(gerichtet an das **Application**-Objekt `app`, das wir als *Anwendungsobjekt* bezeichnen)

```
Application app = new Application();
```

wird das `WpfOhneXaml`-Objekt `hf` zum Vertreter des **Anwendungs- oder Hauptfensters** des Programms. Unter den Fenstern eines Programms zeichnet sich das Hauptfenster durch folgende Besonderheiten aus:

- Über den an das Anwendungsobjekt gerichteten (und nicht wiederholbaren) **Run()** - Aufruf wird für die Anzeige des Hauptfensters gesorgt. Sein Auftritt muss also *nicht* über die **Window**-Methode **Show()** veranlasst werden.
- In der Regel werden erhebliche Teile der Programm-Funktionalität im Hauptfenster angeboten.
- Beim Schließen des Hauptfensters wird das Programm beendet, wenn die **Application**-Eigenschaft **ShutdownMode** den Wert **OnMainWindowClose** besitzt (vgl. Abschnitt 12.2.3).

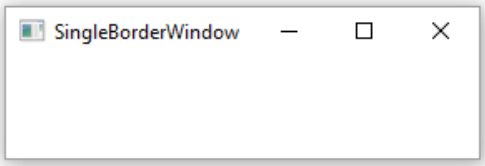
Unsere Beispielprogramme kommen meist mit einem einzigen Fenster aus. Selbstverständlich kann eine WPF-Anwendung auch *mehrere* Fenster verwenden.

So wie im Konstruktor des Beispielprogramms mit der Anweisung

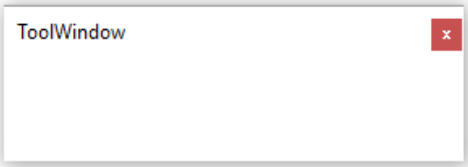
```
Title = "WPF ohne XAML";
```

die Titelzeilenbeschriftung des Hauptfensters über die **Window**-Eigenschaft **Title** festgelegt wird, sind zahlreiche weitere Eigenschaften dieser Klasse veränderbar, z. B.:

- Die Startgröße eines Fensters kann über die **FrameworkElement**-Eigenschaften **Height** und **Width** (vom Typ **double**) geändert werden, z. B.:
  ```
  Width = 300; Height = 100;
  ```
- Über die **Window**-Eigenschaft **WindowStyle** mit der gleichnamigen Enumeration (im Namensraum **System.Windows**) als Datentyp beeinflusst man die Ausstattung der Titelzeile mit Standardschaltflächen und die Rahmengestaltung, z. B.



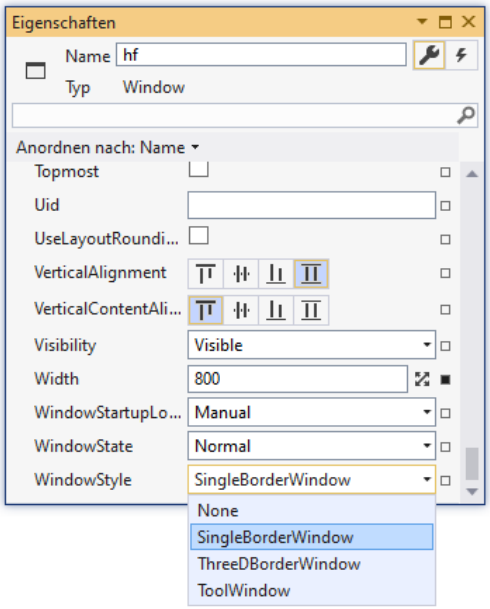
SingleBorderWindow



ToolWindow

None

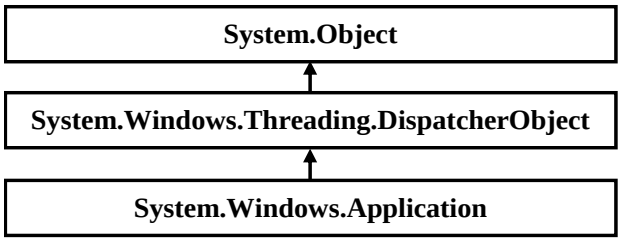
Beim Einsatz des WPF-Designers, mit dem wir schon etliche Erfahrungen gesammelt haben, ist die Eigenschaftsmodifikation zur Entwurfszeit mit dem **Eigenschaften**-Fenster bequem zu erledigen, z. B.:



Wir verzichten momentan auf diesen Komfort, um die Grundstruktur einer WPF-Anwendung an einem rudimentären Beispiel studieren zu können.

12.2.3 Windows-Nachrichten und die Klasse Application

Eine WPF-Anwendung wird durch ein Objekt aus der Klasse **Application** oder aus einer Spezialisierung von **Application** repräsentiert. Im Vergleich zur Klasse **Window** hat die Klasse **Application** einen kleinen Stammbaum:



Das **Application**-Objekt erbringt u. a. folgende Leistungen:

- Es präsentiert wichtige Ereignisse im Lebenslauf einer Programminstanz, auf die Sie eventuell in einer Behandlungsmethode reagieren möchten:
  - **Startup**
    Das Ereignis wird in der Startphase ausgelöst. Eine Behandlungsmethode eignet sich u. a. dazu, um Befehlszeilenargumente über die **Environment**-Methode **GetCommandLineArgs()** abzurufen und dann auszuwerten, z. B.:

```
using System;
using System.Windows;

namespace ApplicationStartup {
  public partial class App : Application {
    private void Application_Startup(object sender, StartupEventArgs e) {
      String[] args = Environment.GetCommandLineArgs();
      for (int i = 1; i< args.Length; i++)
        MessageBox.Show("Argument "+i+": "+ args[i], "Application Startup");
    }
  }
}
```

Die Ereignisbehandlungsmethode muss registriert werden, was z. B. in der XAML-Datei zum Anwendungsobjekt geschehen kann:

```
<Application x:Class="ApplicationExit.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:ApplicationExit"
             StartupUri="MainWindow.xaml" Startup="Application_Startup">
    <Application.Resources>
    </Application.Resources>
</Application>
```

- **Activated**, **Deactivated**
  Das Programm (d.h. eines seiner Fenster) ist in den Vordergrund geholt bzw. von dort verdrängt worden.
- **DispatcherUnhandledException**
  Wenn im primären UI-Thread eine unbehandelte Ausnahme auftritt, besteht die Standardreaktion des WPF-Frameworks darin, nach einer Infodialogbox das Programm zu beenden. Eine **DispatcherUnhandledException**-Behandlungsmethode kann z. B. einen Log-Eintrag schreiben und/oder die Beendigung des Programms verhindern (siehe Abschnitt 13.2.3).[1]
- **SessionEnding**
  Der Benutzer ist im Begriff, seine Windows-Sitzung zu beenden (Abmelden des Benutzers oder Herunterfahren des Rechners).
- **Exit**
  Das Programm sieht seinem Ende entgegen. Wie man auf dieses Ereignis reagieren kann, wurde im Abschnitt 10.2.2 demonstriert.

- Die **Application**-Eigenschaft **MainWindow** zeigt auf das Hauptfenster des Programms, und die Eigenschaft **Windows** zeigt auf eine Liste mit allen Top-Level - Fenstern.
- Die statische **Application**-Eigenschaft **Current** zeigt auf das Anwendungsobjekt.
- Durch die **Run()** - Methode des **Application**-Objekts
  public int Run()
  wird die Verarbeitung der für eine Anwendung relevanten Ereignisse in Gang gesetzt, was im weiteren Verlauf des aktuellen Abschnitts näher erläutert wird.

Durch die von Windows registrierten Benutzeraktivitäten (z. B. Mausklicks, Tastenschläge) und sonstige Ursachen entstehen **Ereignisse** (im Sinne des Betriebssystems), die zu **Nachrichten** an betroffene Anwendungen führen. Wird z. B. ein Fenster vom Benutzer aus der Taskleiste zurückgeholt, dann fordert Windows die Anwendung mit der **WM_PAINT**-Nachricht auf, den Klientenbereich des Fensters neu zu zeichnen. Um diese laufend eintreffenden und in eine Warteschlange eingereihten Nachrichten kümmert sich bei einer WPF-Anwendung eine über die **Application**-Instanzmethode **Run()** gestartete Routine des Laufzeitsystems in einer **while**-Schleife. Hat der Programmierer zu einer Nachricht eine Behandlungsmethode erstellt und zugeordnet, so wird diese aufgerufen. Man kann ein GUI-Programm als Ansammlung von Behandlungsmethoden auffassen, die beim Eintreffen einer passenden Nachricht aufgerufen werden. Solange eine Behandlungsmethode läuft, kann (im selben UI-Thread) keine weitere gestartet werden.

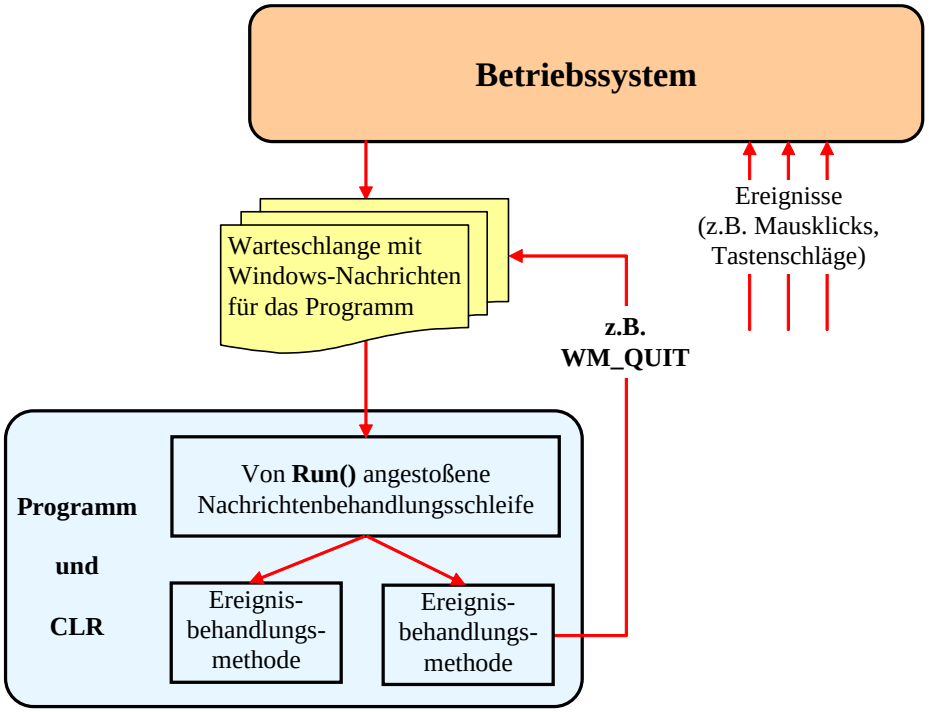
Zu manchen Nachrichten werden von Windows oder von der CLR (Common Language Runtime) ohne Zutun des Programmierers Behandlungsroutinen bereitgestellt. So kann unser Beispielpro-

---

[1] In einer WPF-Anwendung muss man sich auch um unbehandelte Ausnahmen in einem Hintergrund-Thread kümmern (siehe Abschnitt 17.4.6). WPF-Anwendungen, die *mehrere* Top-Level - Fenster besitzen und dementsprechend mehrere UI-Threads (jeweils mit einem eigenen Dispatcher) zur Bedienung der Fenster einsetzen, werden im Kurs nicht behandelt (siehe https://docs.microsoft.com/de-de/dotnet/framework/wpf/advanced/threading-model).

gramm z. B. auf die Standardschaltflächen in der Titelzeile (zum Maximieren, Minimieren oder Schließen) reagieren, ohne dass wir dazu eine Zeile Quellcode schreiben müssten.

Die folgende Abbildung zeigt einige Details zum Nachrichtenverkehr zwischen dem Betriebssystem, der per **Run()** initiierten Nachrichtenbehandlungsschleife und den Ereignisbehandlungsmethoden eines Programms mit *einem* UI-Thread:



Wird die Nachricht **WM_QUIT** aus der Warteschlange gefischt, dann wird ggf. eine registrierte Methode zu Ihrer Behandlung noch ausgeführt. Sobald dies erledigt ist, endet die Nachrichtenbehandlungsschleife, und der **Run()** - Aufruf kehrt zurück.

Verantwortlich für die Nachricht **WM_QUIT** ist die **Application**-Methode **Shutdown()**, die vom Laufzeitsystem oder vom Programm aufgerufen werden kann. Ein Aufruf per Programm muss im selben Thread stattfinden, der das Anwendungsobjekt erstellt hat.

Das Laufzeitsystem ruft die Methode **Shutdown()** in folgenden Fällen auf (MacDonald 2012, S. 198):

- Das letzte Fenster der Anwendung wird vom Benutzer geschlossen, und die **Application**-Eigenschaft **ShutdownMode** hat den voreingestellten Wert **OnLastWindowClose**.
- Das Hauptfenster der Anwendung wird vom Benutzer geschlossen, und die **Application**-Eigenschaft **ShutdownMode** hat den Wert **OnMainWindowClose**.
- Die Windows-Sitzung endet, weil sich der Benutzer abmeldet, oder das Betriebssystem heruntergefahren wird, und das daraufhin ausgelöste Ereignis **SessionEnding** bleibt entweder unbehandelt oder wird ohne Widerspruch gegen das Sitzungsende behandelt.

Hat die **Application**-Eigenschaft **ShutdownMode** den Wert **OnExplicitShutdown**, dann führt das Schließen des letzten Fensters oder des Hauptfensters *nicht* zur Beendigung des Programms. Um in dieser Lage bei laufender Windows-Sitzung für die Nachricht **WM_QUIT** zu sorgen, muss die Methode **Shutdown()** per Programm aufgerufen werden. Dabei kann man einen Return Code festlegen, der zur weiteren Verarbeitung an das Betriebssystem übergeben wird.

Eine WPF-Anwendung über die statische Methode **Environment.Exit()** zu beenden, ist in der Regel *nicht* sinnvoll, weil der komplette Prozess mit allen darin vorhandenen Threads abrupt beendet wird.

Fassen wir leicht vereinfachend zusammen, was durch den Aufruf der **Application**-Methode **Run()** im Beispielprogramm von Abschnitt 12.2.1

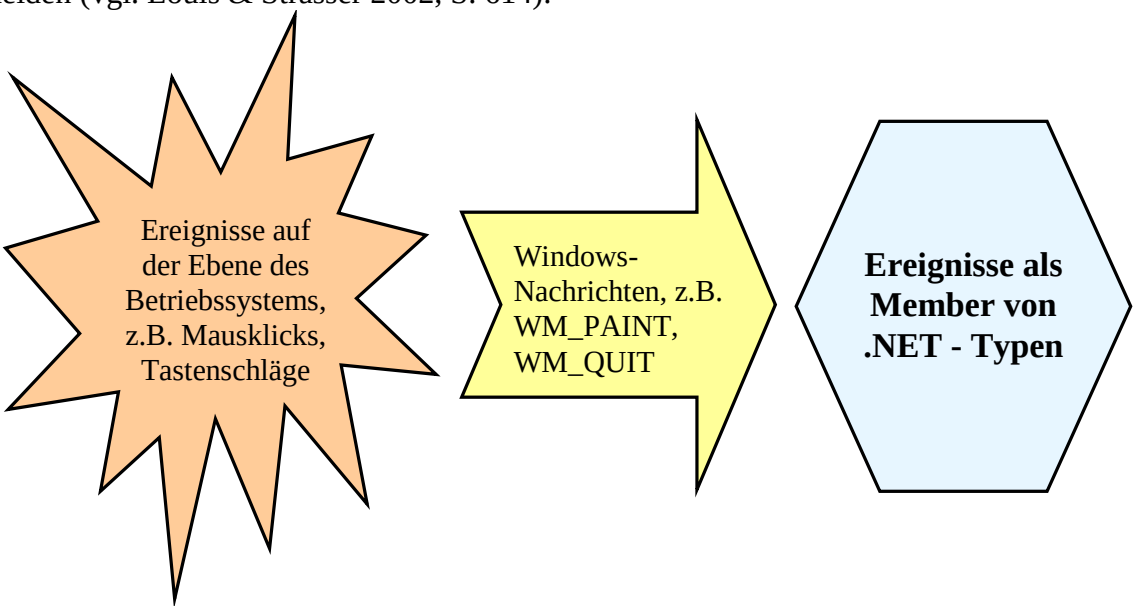
```
static void Main() {
   Application app = new Application();
   WpfOhneXaml hf = new WpfOhneXaml();
   app.Run(hf);
}
```

veranlasst wird:

- Das **Hauptfenster** erscheint.
- Das Programm wird um eine **Nachrichtenschleife** erweitert, welche regelmäßig die von Windows für das Programm verwaltete Nachrichtenwarteschlange inspiziert und auf Ereignisse ggf. mit dem Aufruf einer registrierten Methode reagiert.[1]

Nachdem die Nachricht **WM_QUIT** eingetroffen ist und ggf. (über das **Exit**-Ereignis des Anwendungsobjekts) behandelt wurde, enden die Nachrichtenschleife und die **Run()** - Methode. In der Regel stellt eine GUI-Anwendung nach der Rückkehr des **Run()** - Aufrufs ihre Tätigkeit ein (siehe **Main()** - Methode des Beispielprogramms). Ein erneuter Aufruf der **Run()** - Methode ist jedenfalls *nicht* möglich.

Weil das Windows-API (*Application Programming Interface*) durch .NET gekapselt wird, muss sich ein C# - Programmierer nicht direkt um Windows-Nachrichten kümmern, sondern kann die von vielen Klassen und Strukturen präsentierten **Ereignisse** im .NET - Sinn (siehe Abschnitt 10.2) durch eigene Methoden behandeln.[2] Z. B. stellt die Klasse **Application** zur Windows-Nachricht **WM_QUIT** das Ereignis **Exit** zur Verfügung, bei dem .NET - Programmierer eine eigene Methode per Delegatenobjekt registrieren können, wenn sie auf das Ereignis (bzw. auf die zugrunde liegende Windows-Nachricht) regieren möchten (siehe Beispiel im Abschnitt 10.2.2). Eine registrierte Methode wird automatisch aufgerufen, wenn das zugehörige Ereignis eintritt. Insgesamt kann man unterscheiden (vgl. Louis & Strasser 2002, S. 614):



[1] Genau genommen ist für jeden Thread (vgl. Abschnitt 17), der ein Fenster auf dem Bildschirm präsentiert, eine Nachrichtenwarteschlange und dementsprechend eine Nachrichtenschleife erforderlich. Ein WPF-Programm kann also *mehrere* UI-Threads haben, was aber in den Kursbeispielen nicht vorkommen wird.

[2] Grundsätzlich können auch Strukturen Ereignisse anbieten, doch geschieht dies sehr selten.

12.3 Die eXtensible Application Markup Language (XAML)

Die XML-basierte *eXtensible Application Markup Language* (XAML) wurde von Microsoft zur Deklaration der Bedienoberfläche einer WPF-Anwendung entwickelt, wird aber auch für andere GUI-Frameworks aus Microsofts Entwicklungslaboren genutzt (vgl. Abschnitt 12.1.1):

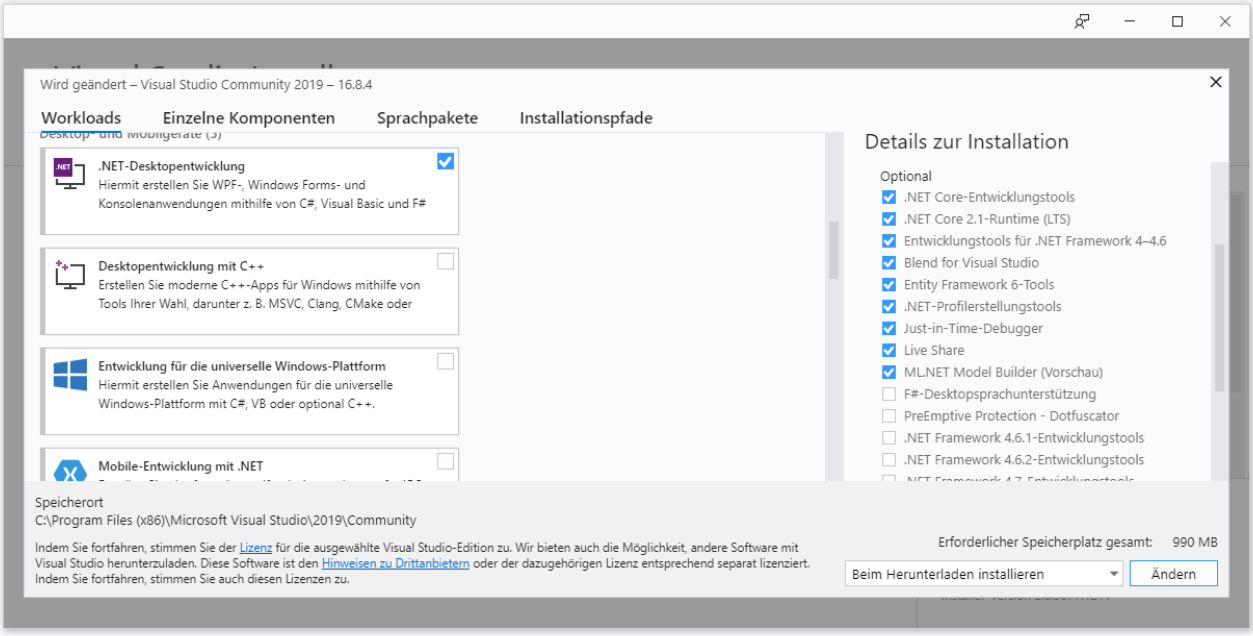
- UWP (***U**niversal **W**indows **P**latform*)
- MAUI (***M**ulti-platform **A**pp **UI**)*

Vor Einführung der WPF-Technik waren die Anwendungslogik *und* die Bedienoberfläche einer GUI-Anwendung (mit dem WinForms-Framework) in derselben Programmiersprache zu realisieren. Eine GUI-Kreation mit C# - Anweisungen ist zwar auch im WPF-Framework möglich, doch existiert hier mit der deklarativen Sprache XAML eine Alternative, die folgenden Zielen dienen soll:

- Vereinfachung der GUI-Gestaltung
- Trennung von Anwendungslogik und GUI-Design

Damit bestehen gute Voraussetzungen für die erfolgreiche Zusammenarbeit von Software-Entwicklern und Grafikdesignern.

Man kann den XAML-Code direkt editieren, und/oder die Unterstützung des WPF-Designers im Visual Studio in Anspruch nehmen. Für Projekte mit hohen grafischen und/oder multimedialen Ansprüchen steht außerdem noch das auf die GUI-Gestaltung spezialisierte Microsoft-Produkt **Blend for Visual Studio** zur Verfügung, das optional zusammen mit dem Visual Studio als Bestandteil der Workload **.NET-Desktopentwicklung** installiert werden kann (siehe Abschnitt 3.3.2 zur Anpassung einer Visual Studio - Installation):



Im Kurs verwenden wir oft den WPF-Designer als wertvolle Unterstützung beim Erstellen des XAML-Codes zu einem Fenster, während Blend nicht zum Einsatz kommt.

Dank WPF-Designer kann man in den meisten Fällen darauf verzichten, den XAML-Code direkt zu editieren und kommt somit auch ohne die im weiteren Verlauf von Abschnitt 12.3 noch folgenden, nicht immer leicht verdaulichen Erläuterungen zur XAML aus. In manchen Fällen ist aber das direkte Verfassen von XAML-Code effektiver oder sogar alternativlos.

12.3.1 Elementare Regeln zum Aufbau einer XML-Datei

Die *eXtensible Markup Language* (XML) hat sich als universelles Mittel zur Deklaration von hierarchisch strukturieren Daten etabliert und wird auch in .NET - Projekten intensiv verwendet, wobei besonders zu erwähnen sind:

- WPF-Fenster- und -Anwendungsdeklaration im XAML-Format
- Konfigurationsdateien im XML-Format

Daher werden in diesem Abschnitt elementare Regeln zum Aufbau einer XML-Datei erläutert.

Eine XML-Datei enthält Text und ist auch für die Lektüre durch Menschen relativ gut geeignet. In der ersten Zeile steht in der Regel die (nicht obligatorische) **XML-Deklaration** mit einer Versions- und einer Codierungsangabe:

```
<?xml version="1.0" encoding="utf-8" ?>
```

Bei den im aktuellen Kapitel vornehmlich relevanten XAML-Dateien wird die XML-Deklaration allerdings meist weggelassen.

Jede XML-Datei besteht aus hierarchisch verschachtelten Elementen und enthält nur *ein Wurzelelement*, z. B. bei einer XAML-Datei zur Deklaration eines WPF-Fensters ein **Window** - Element:

```
<Window . . . >
  <Grid>
    <Button . . . >
      . . .
    </Button>
  </Grid>
</Window>
```

Wie das Beispiel **Window** zeigt, besteht ein XML-Element *mit* Inhalt (z. B. mit untergeordneten Elementen) aus:

- Startkennung (oft bezeichnet als *Start-Tag*)
- Inhalt
- Endkennung (oft bezeichnet als *End-Tag*)

Start- und Endkennung werden durch Paare spitzer Klammern begrenzt und enthalten den Namen des Elements. In der Endkennung folgt auf die einleitende spitze Klammer ein Schrägstrich.

Als Bestandteile seiner Startkennung kann ein Element beliebig viele **Attribute** enthalten, die aus Name-Wert - Paaren bestehen und durch mindestens ein Leerzeichen getrennt sind. Das folgende **Button** - Element zur Deklaration eines Befehlsschalters besitzt z. B. die Attribute **Name** (benennt die Instanzvariable zum Steuerelement), **HorizontalAlignment** und **VerticalAlignment** (horizontale bzw. vertikale Ausrichtung des Steuerelements im umgebenden Steuerelement-Container):

```
<Button Name="button" HorizontalAlignment="Center" VerticalAlignment="Center">
  . . .
</Button>
```

Alle Attributwerte sind durch (doppelte oder einfache) Anführungszeichen zu begrenzen.

Ein Element *ohne* untergeordnete Elemente (oder sonstige Inhalte) kann sich auf die Startkennung beschränken, die dann mit einem Schrägstrich zu enden hat, z. B. beim folgenden **Image** - Element aus einer XAML-Datei:

```
<Image Source="count.png" />
```

Wie beim C# - Quellcode ist auch beim XML-Code die **Groß-/Kleinschreibung relevant** (mit seltenen Ausnahmen, auf die bei Gelegenheit hingewiesen wird).

Das folgende Beispiel zeigt, wie **Kommentare** in einer XML-Datei untergebracht werden:

```
<!-- Kommentar -->
```

12.3.2 XAML-Beschreibung

Das Erstellen und Initialisieren der Objekte der Bedienoberfläche kann auch bei einer WPF-Anwendung per C# - Quellcode erfolgen. Es ist jedoch zu empfehlen, für diesen Zweck die eXtensible Application Markup Language (XAML) zu verwenden. Die XAML-Deklarationen werden in Textdateien mit der UTF-8 - Codierung und der Namenserweiterung .**xaml** untergebracht.

12.3.2.1 Wurzelelement und XML-Namensräume

Eine XAML-Datei einhält *ein* Wurzelelement, wobei für uns in Frage kommen:

- **Application**
  Dieses Wurzelelement wird in der XAML-Datei mit der Anwendungsdeklaration verwendet. Es nennt auf jeden Fall als Wert des Attributs **x:Class** die Anwendungsklasse (Standardname: `App`) samt .NET-Namensraum und als Wert des Attributs **StartupUri** die XAML-Datei zum Hauptfenster. Außerdem können Behandlungsmethoden zu Ereignissen der Anwendungsklasse sowie Ressourcen deklariert werden. Als Name der zugehörigen XAML-Datei wird meist **App.xaml** verwendet. Welchen XAML-Code das Visual Studio für eine neue WPF-Anwendung erzeugt, betrachten wir am Beispiel unseres RSS-Feed-Reader - Projekts (mit dem Namensraum `RssFeedReader`, vgl. Abschnitt 6.8):[1]

```
<Application x:Class="RssFeedReader.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:RssFeedReader"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

- **Window**
  Dieses Wurzelelement wird in der XAML-Datei zur Deklaration eines Fensters (bzw. einer Fensterklasse) verwendet. Beim Hauptfenster einer Anwendung verwendet das Visual Studio für die realisierende Klasse per Voreinstellung den Namen `MainWindow`, und der XAML-Code landet in der Datei **MainWindow.xaml**. Welchen XAML-Code das Visual Studio für ein neues WPF-Fenster erzeugt, betrachten wir am Beispiel unseres RSS-Feed-Reader - Projekts:

```
<Window x:Class="RssFeedReader.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:RssFeedReader"
        Title="MainWindow" Height="450" Width="800">
    <Grid>

    </Grid>
</Window>
```

  Mit Ausnahme der Eigenschaftselemente (siehe Abschnitt 12.3.2.3.2) repräsentiert jedes Element einer XAML-Datei eine Instanz eines .NET - Typs, und der Elementname ist identisch mit dem Namen dieses Typs (z. B. **Button**) oder mit dem Namen seiner Basisklasse (z. B. **Window**).

---

[1] Der Übersichtlichkeit halber wurden überflüssige XAML-Namensraumdeklarationen entfernt.

Die Startkennung zu einem XAML - Wurzelelement enthält mehrere **xmlns**-Attribute zur Deklaration von **XML-Namensräumen**, wobei die folgende Syntax verwendet wird:

xmlns[**:***prefix*]="*URI*"

In einem Namensraum werden erlaubte Elemente und Attribute festgelegt, und zur Vermeidung von Namenskollisionen kann zu einem Namensraum ein Präfix angegeben werden.

Es besteht eine starke Beziehung zwischen den vertrauten Namensräumen für .NET - Typen und den XML-Namensräumen in einer XAML-Datei. Es wäre allerdings sehr umständlich, für die auf verschiedene .NET - Namensräume verteilten WPF-Klassen jeweils das korrekte Präfix angeben zu müssen. Daher wurden *alle* .NET - Namensräume mit WPF-Typen in einem einzigen XAML-Namensraum zusammengefasst. Dies ist möglich, weil keine Namenskollisionen auftreten. Um die Verwendung des sehr wichtigen XAML-Namensraums mit den WPF-Klassen zu erleichtern, hat er *kein* Präfix:

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

Die Bezeichnung für einen XML-Namensraum verwendet das URI-Format (*Uniform Resource Identifier*), um Eindeutigkeit sicherzustellen. Wie man sich leicht vergewissern kann, existiert aber im Internet kein Ort mit diesem Namen.

Neben dem Namensraum für die WPF-Klassen gibt es einen weiteren extrem wichtigen XML-Namensraum. Er enthält die XAML-Sprachdefinition und verwendet das Präfix **x**:[1]

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Außerdem vereinbart das Visual Studio einen XML-Namensraum mit dem Präfix **local** für den projektspezifischen .NET - Namensraum:

```
xmlns:local="clr-namespace:RssFeedReader"
```

Die Vereinbarung von XML-Namensräumen erfolgt meist im Start-Tag des Wurzelelements, ist aber grundsätzlich in *jedem* Start-Tag erlaubt.

Die folgenden **x**-Attribute werden oft benötigt (nicht nur im Wurzelelement):

- **x:Class**
  Im **x:Class** - Attribut wird die zu einem Element gehörende Code-Behind - Klasse (siehe Abschnitt 12.3.3) samt .NET - Namensraum angegeben, z. B.:
  ```
  x:Class="RssFeedReader.MainWindow"
  ```
- **x:Key**
  Zu einer WPF-Anwendung existiert ein Kollektionsobjekt aus der Klasse **ResourceDictionary** mit anwendungsglobal verwendbaren Ressourcen (z. B. Farbdefinitionen). In der XAML-Datei zur Anwendungsklasse wird die Ressourcen-Kollektion vereinbart und der **Application**-Eigenschaft **Resources** zugewiesen. Wird ein Objekt (z. B. aus der zur Definition einer Oberflächengestaltung geeigneten Klasse **SolidColorBrush**) in die Ressourcen-Kollektion aufgenommen, wird zu seiner Ansprache ein Schlüssel vereinbart, z. B.:
  ```
  <Application x:Class= ... >
      <Application.Resources>
          <SolidColorBrush x:Key="BgColor" Color="WhiteSmoke"/>
      </Application.Resources>
  </Application>
  ```

[1] Weitere Information über die XML-Namensräume mit den WPF-Klassen bzw. mit der XAML-Sprachdefinition finden sich hier:
https://docs.microsoft.com/de-de/dotnet/framework/wpf/advanced/xaml-namespaces-and-namespace-mapping-for-wpf-xaml

Wie ein Ressourcen-Objekt später verwendet (z. B. einer Eigenschaft zugewiesen) werden kann, ist im Abschnitt 12.3.2.3.6 über die Markup-Erweiterungen zu erfahren.

- **x:Name**
  Über dieses Attribut legt man für die Instanz, die aus der XAML-Laufzeitverarbeitung eines XAML-Elements entsteht, einen Namen fest, z. B.:
  ```
  <TextBox x:Name="textBox" ... />
  ```
  Es resultiert eine Instanzvariable mit dem gewählten Namen, die in der Code-Behind - Datei (siehe Abschnitt 12.3.3) einen Instanzzugriff erlaubt. Man muss nicht unbedingt für jede aus einem XAML-Element resultierende Instanz einen Namen vereinbaren. Es ist z. B. kein Name erforderlich, um bei den Ereignissen eines Steuerelements Behandlungsmethoden registrieren zu können.

Zum Attribut **x:Name** existiert die funktionsverwandte Alternative **Name** (ohne Namensraum-Präfix), die bei .NET - Klassen mit einer **Name**-Instanzeigenschaft in Frage kommt. Dazu gehört auch die Klasse **FrameworkElement**, von der alle WPF-Steuerelemente abstammen. Wenn eine Klasse mit **Name**-Instanzeigenschaft außerdem mit dem Attribut (im Sinn von Kapitel 14) **RuntimeNameProperty** dekoriert ist (wie z. B. die Klasse **FrameworkElement**), dann kann statt des XAML-Attributs **x:Name** auch das Attribut **Name** verwendet werden. In diesem Fall übernimmt der XAML-Parser die angegebene Zeichenfolge für die Referenzvariable *und* für die Instanzeigenschaft des Objekts (MacDonald 2012, 28f), z. B.:

```
<StackPanel Name="stackPanel" Orientation="Horizontal">
```

Beim Erstellen und Initialisieren von Objekten per XAML-Code muss man sich nicht auf WPF-Klassen beschränken. Es werden beliebige .NET - Klassen unterstützt, sofern diese über einen parameterlosen und öffentlichen Konstruktor verfügen. Um beliebige CLR-Typen im XAML-Code ansprechen zu können, bezieht man ihren .NET - Namensraum ein (siehe obiges Beispiel mit dem Namensraum des Projekts).

12.3.2.2 Instanzelemente

Mit Ausnahme der Eigenschaftselemente (siehe Abschnitt 12.3.2.3.2) repräsentiert jedes XAML-Element eine Instanz (z. B. ein Objekt) mit einem beliebigen, in einem zugänglichen Assembly realisierten .NET - Typ. In der Startkennung eines Instanzelements ist nach der öffnenden spitzen Klammer der Typname anzugeben.

Sind keine untergeordneten Elemente vorhanden, ist die Startkennung mit einem Schrägstrich und einer schließenden spitzen Klammer zu komplettieren, z. B.

```
<TextBlock Text="Click to Feed" Margin="5" VerticalAlignment="Center" />
```

Sind untergeordnete Elemente (z. B. Steuerelemente in einem Layoutcontainer) oder sonstige Inhalte (z. B. ein Text) vorhanden, dann beendet man die Startkennung mit einer schließenden spitzen Klammer, lässt die untergeordneten Elemente bzw. den sonstigen Inhalt folgen und setzt eine Endkennung, die zwischen einem Paar spitzer Klammern einen Schrägstrich und den Element- bzw. Typnamen enthält, z. B.:

```
<StackPanel Name="stackPanel" Orientation="Horizontal">
    <Image Width="20" Source="vs.png" VerticalAlignment="Center"/>
    <TextBlock Text="Click" Margin="5" VerticalAlignment="Center" />
</StackPanel>
```

In diesem **StackPanel**-Element wird übrigens von den beiden im Abschnitt 12.3.2.1 beschriebenen **Name**-Attributen die Variante aus dem XAML-Namensraum für WPF-Klassen verwendet (ohne Namensraumpräfix **x**).

Ein Instanzelement stellt einen Auftrag an die XAML-Verarbeitung dar, eine Instanz vom angegebenen Typ zu erstellen. Das tatsächliche Erstellen findet zur Laufzeit über einen Aufruf des parameterlosen Konstruktors zum jeweiligen Typ statt (siehe Abschnitt 12.3.4).

Das Instanziieren einer *nicht* zum XAML-Namensraum für WPF-Klassen, sondern zum projektspezifischen XAML-Namensraum gehörigen Klasse ist uns übrigens im Abschnitt 10.2.3 im Zusammenhang mit einem benutzerdefinierten Steuerelement begegnet:

```
<Grid>
   <local:SurpriseButton x:Name="surpriseButton" Content="7 gewinnt"
    Margin="30" FontSize="36"/>
</Grid>
```

12.3.2.3 Eigenschaftsausprägungen zuweisen

12.3.2.3.1 Attributsyntax

Viele Instanzeigenschaften können in der Instanzelement-Startkennung über die **XML-Attributsyntax** einen Wert erhalten, wobei auf den Namen der Eigenschaft das „=“ - Zeichen und durch Anführungszeichen begrenzt eine Zeichenfolge als Wert folgt, z. B.[1]

```
<TextBlock Text="Click to Feed" Margin="5" VerticalAlignment="Center" />
```

Bei der Wandlung einer Zeichenfolge in den jeweiligen Eigenschaftstyp sind **Typkonverter** im Spiel, die eventuell eine komplexe Aufgabe zu erfüllen haben. Im letzten Beispiel muss aus der Zeichenfolge zur **TextBlock**-Eigenschaft **Margin**, die für einen Abstand zur Umgebung sorgt, eine Instanz vom Strukturtyp **System.Windows.Thickness** entstehen. Welche Klasse für die Wandlung zuständig ist, wird durch ein Attribut (im Sinn von Kapitel 14) zum Typ der Eigenschaft festgelegt, im Beispiel:

```
[TypeConverter(typeof(ThicknessConverter))]
[Localizability(LocalizationCategory.None, Readability = Readability.Unreadable)]
public struct Thickness : IEquatable<Thickness> { ... }
```

12.3.2.3.2 Eigenschaftselementsyntax

Die Attributsyntax eignet sich nicht, wenn einer Eigenschaft eine komplexe Member-Instanz zugewiesen werden soll. In diesem Fall ordnet man dem Instanzelement ein **Eigenschaftselement** unter, dessen Name nach dem Schema

Typname.Eigenschaftsname

zu bilden ist. Als Inhalt des Eigenschaftselements tritt ein Instanzelement mit dem Typ der Eigenschaft auf. Im folgenden Beispiel wird der **Content**-Eigenschaft eines **Button**-Objekts (Datentyp: **Object**) ein **StackPanel**-Layoutobjekt (siehe Abschnitt 12.6.3) zur Verwaltung der **Button**-Oberfläche zugewiesen:

```
<Button Name="button" Background="WhiteSmoke">
   <Button.Content>
      <StackPanel Orientation="Horizontal">
         . . .
      </StackPanel>
   </Button.Content>
</Button>
```

Gleich wird sich allerdings herausstellen, dass bei der **Button**-Eigenschaft **Content** eine Vereinfachung der XAML-Syntax möglich ist. Weil diese Eigenschaft bei der Klasse **Button** sehr oft ange-

---

[1] Das Visual Studio verwendet doppelte Anführungszeichen, doch sind auch einfache erlaubt.

sprochen werden muss, ist sie als *XAML-Inhaltseigenschaft* (siehe Abschnitt 12.3.2.3.5) für **Button**-Elemente festgelegt worden, sodass man auf die Einrahmung

```
<Button.Content> ... </Button.Content>
```

verzichten kann.

Die Attributsyntax ist letztlich eine bequeme Kurzform der Eigenschaftselementsyntax. Z. B. ist

```
<Button Name="button" Content="Click to Feed" Background="WhiteSmoke"/>
```

die Kurzform von:

```
<Button Name="button" Content="Click to Feed">
  <Button.Background>
      <SolidColorBrush Color="WhiteSmoke"/>
  </Button.Background>
</Button>
```

12.3.2.3.3 Textinhaltsyntax

Bei einigen Eigenschaftselementen kann Text ohne Begrenzung durch Anführungszeichen als Inhalt angegeben werden, z. B.:

```
<Button>
    <Button.Content>
        Knopf
    </Button.Content>
</Button>
```

Für den Datentyp der betroffenen Eigenschaft muss eine von den folgenden Bedingungen erfüllt sein:

- Einer Eigenschaft von diesem Typ kann eine Zeichenfolge zugewiesen werden, was z. B. beim Typ **Object** (dem Typ der **Button**-Eigenschaft **Content**) der Fall ist.
- Für den Typ ist eine Klasse mit entsprechenden Kompetenzen als **Typkonverter** definiert, z. B.:

  ```
  <Button.Background>
      LightBlue
  </Button.Background>
  ```

- Der Typ ist im XAML-Sprachumfang bekannt, was bei vielen elementaren Datentypen (z. B. **Int32**, **Double**) der Fall ist, z. B.:

  ```
  <Button.Height>
      40
  </Button.Height>
  ```

Die Textinhaltsyntax wird meist bei der *Inhaltseigenschaft* (siehe Abschnitt 12.3.2.3.5) verwendet, sodass sich das erste Beispiel des aktuellen Abschnitts vereinfachen lässt:

```
<Button>
    Knopf
</Button>
```

Noch bequemer ist natürlich in diesem Fall die Attributsyntax:

```
<Button Content="Knopf"/>
```

12.3.2.3.4 Kollektionssyntax

Besitzt eine Eigenschaft einen Kollektionstyp, der das Interface **IList<T>** implementiert, dann ist die **XAML-Kollektionssyntax** erlaubt, d. h.:

- Man verzichtet auf ein Eigenschaftselement zum Kollektionsobjekt.
- Man listet Instanzelemente auf, die schlussendlich über die **IList<T>** - Methode **Add()** in die Kollektion aufgenommen werden.

Im folgenden Beispiel wird ein Farbverlauf durch ein Objekt der Klasse **LinearGradientBrush** mit der Eigenschaft **GradientStops** per Eigenschaftselementsyntax definiert. Statt ein Instanzelement vom Typ **GradientStopCollection** samt Kindelementen anzugeben, werden ausschließlich die Kindelemente (vom Typ **GradientStop**) aufgelistet:

```
<LinearGradientBrush>
  <LinearGradientBrush.GradientStops>
    <GradientStop Offset="0.0" Color="Red" />
    <GradientStop Offset="1.0" Color="Blue" />
  </LinearGradientBrush.GradientStops>
</LinearGradientBrush>
```

12.3.2.3.5 Inhaltseigenschaft

Jede WPF-Klasse kann von ihren Instanzeigenschaften genau eine als **Inhaltseigenschaft** (engl.: *content property*) deklarieren. Dazu wird der Klasse das Attribut **ContentProperty** (im Namensraum **System.Windows.Markup**) angeheftet, wobei hier der Begriff *Attribut* im Sinn von Kapitel 14 zu verstehen ist, z. B. bei der Klasse **ContentControl**:

```
[ContentProperty("Content")]
[Localizability(LocalizationCategory.None, Readability = Readability.Unreadable)]
public class ContentControl : Control, IAddChild {
  . . .
}
```

Ist ein Kindelement als Wert der Inhaltseigenschaft anzugeben, kann auf das Kennungspaar gemäß XAML-Eigenschaftselementsyntax verzichtet werden. Folglich kann das im Abschnitt 12.3.2.3 präsentierte Beispiel mit einer Deklaration für die **Button**-Eigenschaft **Content**

```
<Button Name="button" Background="WhiteSmoke">
  <Button.Content>
    <StackPanel Orientation="Horizontal">
      . . .
    </StackPanel>
  </Button.Content>
</Button>
```

einfacher geschrieben werden:

```
<Button Name="button" Background="WhiteSmoke">
  <StackPanel Orientation="Horizontal">
      . . .
  </StackPanel>
</Button>
```

Bei der Layoutcontainer-Klasse **StackPanel** (siehe Abschnitt 12.6.3) ist die (von **Panel** geerbte) Eigenschaft **Children** als Inhaltseigenschaft deklariert:

```
[Localizability(LocalizationCategory.Ignore)]
[ContentProperty("Children")]
public abstract class Panel : FrameworkElement, IAddChild {
  . . .
}
```

Folglich kann das Kennungspaar

```
<StackPanel.Children> ... </StackPanel.Children>
```

weggelassen werden.

Die Eigenschaft **Children** ist vom Typ **UIElementCollection**, der das Interface **IList<T>** implementiert. Folglich darf man die XAML-Kollektionssyntax verwenden (siehe Abschnitt 12.3.2.3.4). Im folgenden Beispiel

```
<StackPanel Orientation="Horizontal">
    <Image Source="count.png"  />
    <AccessText>Count</AccessText>
</StackPanel>
```

werden zwei XAML-Vereinfachungsmöglichkeiten genutzt:

- Inhaltseigenschaft
  Das Eigenschaftselement **StackPanel.Children** ist weggelassen (Inhaltseigenschaft).
- Kollektionssyntax
  Auf das **UIElementCollection**-Instanzelement wird ebenfalls verzichtet. Es werden direkt die Kindelemente zu dieser Kollektion angegeben.

Bei der Inhaltseigenschaft wird oft die im Abschnitt 12.3.2.3.3 beschriebene Textinhaltsyntax verwendet, z. B.:

```
<Button Name="button">
   Count
</Button>
```

12.3.2.3.6 Markup-Erweiterungen

Mit den bisher vorgestellten Techniken zur Festlegung von Eigenschaftsausprägungen ist es *nicht* möglich, ...

- eine Referenz zu einem bereits existenten Objekt zuzuweisen,
- eine Verbindung zu einer Quelle von dynamisch zu aktualisierenden Werten herzustellen.

Dies gelingt mit einer sogenannten **Markup-Erweiterung**, die meist per Attributsyntax unter Verwendung einer geschweift eingeklammerten Zeichenfolge realisiert wird.[1] Im folgenden Beispiel wird einer Eigenschaft eine Referenz auf ein anderenorts definiertes Objekt zugewiesen:

```
Background="{StaticResource BgColor}"
```

Eine Markup-Erweiterung hat einen Typ, der bei Verwendung der Attributsyntax unmittelbar nach der öffnenden Klammer anzugeben ist. Die Erweiterungstypen stammen von der Klasse **MarkupExtension** im .NET - Namensraum **System.Windows.Markup** ab. Viele für WPF-Anwendungen wichtige Markup-Erweiterungstypen wurden in den XAML-Standardnamensraum aufgenommen, sodass bei der Typbezeichnung kein Namensraumpräfix benötigt wird. Besonders wichtig sind:

[1] Es ist auch möglich, allerdings weniger gebräuchlich, Markup-Erweiterungen in Eigenschaftselementen zu verwenden (siehe MacDonald 2012, S. 34).

- **StaticResource**
  In WPF-Anwendungen werden häufig sogenannte statische Ressourcen per Markup-Erweiterung zugewiesen. So wird es möglich, einmalig definierte Einstellungsobjekte (z. B. zur Hintergrundgestaltung) in mehreren Steuerelementen oder Fenstern eines Programms zu verwenden. In der folgenden XAML-Datei zur Anwendungsklasse eines Programms wird in die Kollektion zur **Application**-Eigenschaft **Resources** (Datentyp **ResourceDictionary**) ein **SolidColorBrush**-Objekt aufgenommen, das anschließend anwendungsglobal über den vereinbarten Schlüssel `BgColor` genutzt werden kann:

```
<Application x:Class= ... >
    <Application.Resources>
        <SolidColorBrush x:Key="BgColor" Color="WhiteSmoke"/>
    </Application.Resources>
</Application>
```

  In der folgenden XAML-Datei zum Hauptfenster derselben Anwendung wird das **SolidColorBrush**-Objekt per Markup-Erweiterung der **Window**-Eigenschaft **Background** als statische Ressource zugewiesen, wobei nach dem Markup-Erweiterungstyp der Schlüssel anzugeben ist:

```
<Window x:Class="MarkupExtDemo.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Routing" Height="350" Width="525"
    Background="{StaticResource BgColor}">
  . . .
</Window>
```

- **Binding**
  Im Abschnitt 6.8.6 haben wir schon Bekanntschaft mit der WPF-Datenbindungstechnologie gemacht. Für den zur formatierten Anzeige eines **ListBox**-Elements verwendeten **TextBlock**

```
<TextBlock Text="{Binding Path=Title}" Margin="1" TextAlignment="Center"
   TextWrapping="Wrap" FontSize="14" FontWeight="Bold"
   Foreground="DarkMagenta" />
```

  wird durch die folgende Attributsyntax mit Markup-Erweiterung

```
Text="{Binding Path=Title}"
```

  ein Objekt der Klasse **Binding** beauftragt, aus dem aktuellen Element der zum **ListBox**-Objekt gehörigen Kollektion den Wert der Eigenschaft `Title` zu extrahieren. Eine Begrenzung des Eigenschaftsnamens durch Anführungszeichen ist *nicht* erforderlich bzw. erlaubt, weil es ansonsten zu einer schwer interpretierbaren Verschachtelung von Anführungszeichen käme.

Manche Markup-Erweiterungsklassen sind *nicht* im XAML-Standardnamensraum für WPF-Klassen vorhanden, sondern im generellen XAML-Namensraum definiert, sodass dem Typnamen das Namensraumpräfix **x** vorangestellt wird. Ein wichtiges Beispiel ist die Markup-Erweiterung **StaticExtension**, die eine Referenz auf ein statisches Feld oder auf eine statische Eigenschaft liefert. Im folgenden Beispiel wird sie dazu verwendet, der Steuerelementeigenschaft **Background** das durch die statische Eigenschaft **ControlDarkBrush** der Klasse **SystemColors** referenzierte Objekt zuzuweisen. Dabei darf der Namensbestandteil *Extension* weggelassen werden:

```
Background="{x:Static SystemColors.ControlDarkBrush}"
```

Durch geschweifte Klammern begrenzt wird eine Zeichenfolge mit dem folgenden Aufbau erwartet:

Prefix:*Type Name*.*FieldOrPropertyName*

Das Präfix ist nur bei Typen anzugeben, die *nicht* in den XAML-Standardnamensraum eingeblendet wurden.

12.3.2.4 Ereignisbehandlungsmethoden registrieren

Wie die folgende Startkennung zum **Application**-Wurzelelement aus der XAML-Datei zu einer Anwendungsklasse zeigt, kann man per Attributsyntax nicht nur einer Eigenschaft, sondern auch einem Ereignis einen Wert zuweisen, wobei der Name der Behandlungsmethode anzugeben ist:

```
<Application x:Class="ApplicationExitVS.App"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   StartupUri="MainWindow.xaml" Exit="Application_Exit">
      . . .
</Application>
```

Implementiert wird eine solche Behandlungsmethode in der Code-Behind - Datei zum XAML-Code (siehe Abschnitt 12.3.3).

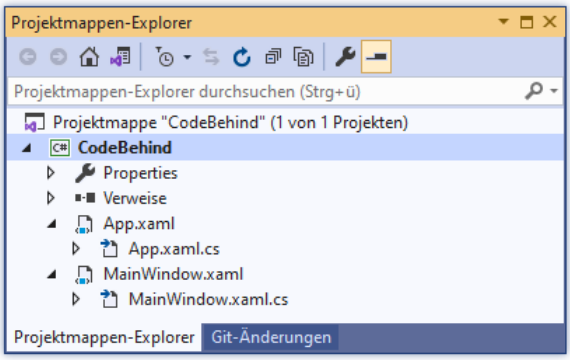
Man kann darüber diskutieren, ob hier eventuell das Prinzip der Trennung von Anwendungslogik und GUI-Gestaltung verletzt ist.

12.3.3 Code-Behind - Dateien

Ein vom Visual Studio über die Vorlagen **WPF-App (.NET Framework)** oder **WPF-App (.NET)** neu erstelltes Projekt enthält die beiden XAML-Dateien

- **App.xaml** zur Deklaration der Anwendungsklasse
- **MainWindow.xaml** zur Deklaration der Hauptfensterklasse

Wie der **Projektmappen-Explorer** nach dem Aufklappen der Zweige zu den beiden XAML-Dateien zeigt, gehört jeweils eine C# - Quellcodedatei dazu, deren Name aus dem XAML-Dateinamen durch Anhängen der Erweiterung **.cs** entsteht:



Diese beiden Dateien sind für die Anwendungslogik zuständig, werden im Wesentlichen vom Entwickler erstellt und als *Code-Behind - Dateien* bezeichnet.

Sie enthalten nach etlichen **using**-Direktiven zum Import von .NET - Namensräumen jeweils eine partielle Klassendefinition:

- **App.xaml.cs**
  In der Datei **App.xaml.cs** findet sich eine partielle Definition der Anwendungsklasse `App`, die von der BCL-Klasse **Application** abstammt (vgl. Abschnitt 12.2.3):

```
using System;
  . . .
namespace CodeBehind {
 /// <summary>
 /// Interaktionslogik für "App.xaml"
 /// </summary>
 public partial class App : Application {
 }
}
```

  Mit dem Schlüsselwort **partial** wird dem Compiler signalisiert, dass es zu der im aktuellen Quellcode vorhandenen Klassendefinition noch ein Ergänzungsstück in einer anderen Quellcodedatei gibt, wobei die beiden partiellen Definitionen gleichberechtigt und voneinander abhängig ein Ganzes ergeben.

  Wenn man im WPF-Designer der Entwicklungsumgebung eine Ereignisbehandlung zum Anwendungsobjekt anfordert, dann wird die partielle Klassendefinition in der Datei **App.xaml.cs** um eine Behandlungsmethode erweitert, z. B.

```
public partial class App : Application {
   private void Application_Exit(object sender, ExitEventArgs e) {
   }
}
```

  Diesen Methodenrumpf zum **Exit**-Ereignis der Klasse **Application** kann man z. B. so erstellen:

  - Datei **App.xaml** per Doppelklick auf den Eintrag im Projektmappen-Explorer öffnen
  - Im **Eigenschaften**-Fenster per Mausklick auf das Symbol ⚡ die Liste mit den Ereignissen anfordern
  - Doppelklick auf die Textbox zum Ereignis **Exit**

  Dabei erhält in der zugehörigen XAML-Datei das Wurzelelement **Application** einen Wert für das Attribut **Exit**, z. B.:

```
<Application x:Class="CodeBehind.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:CodeBehind"
             StartupUri="MainWindow.xaml" Exit="Application_Exit">
  . . .
</Application>
```

- **MainWindow.xaml.cs**
  In der Datei **MainWindow.xaml.cs** findet sich eine partielle Definition der Fensterklasse `MainWindow`, die von der BCL-Klasse **Window** abstammt (vgl. Abschnitt 12.2.2):

```
using System;
  . . .
namespace CodeBehind {
 /// <summary>
 /// Interaktionslogik für MainWindow.xaml
 /// </summary>
 public partial class MainWindow : Window {
    public MainWindow() {
       InitializeComponent();
    }
 }
}
```

Wenn man im WPF-Designer der Entwicklungsumgebung eine Ereignisbehandlung zu einem Steuerelement auf der Fensteroberfläche anfordert, dann wird die partielle Klassendefinition in der Datei **MainWindow.xaml.cs** um eine Behandlungsmethode erweitert, z. B.

```
public partial class MainWindow : Window {
 public MainWindow() {
    InitializeComponent();
 }
    private void button_Click(object sender, RoutedEventArgs e) {
    }
}
```

Diesen Methodenrumpf zum **Click**-Ereignis der Klasse **Button** kann man z. B. so erstellen:

- Datei **MainWindow.xaml** per Doppelklick auf den Eintrag im Projektmappen-Explorer öffnen
- Im WPF-Designer einen Doppelklick auf das **Button**-Objekt setzen

Dabei erhält in der zugehörigen XAML-Datei **MainWindow.xaml** das zum **Button**-Objekt gehörige Element einen Wert für das Attribut **Click**, z. B.:

```
<Button ... Click="button_Click" />
```

Bei einem von **Click** verschiedenen Ereignistyp ist das Eigenschaftenfenster zu verwenden:

- Das Steuerelement im WPF-Designer oder in der XAML-Ansicht markieren
- Im **Eigenschaften**-Fenster per Mausklick auf das Symbol ⚡ die Liste mit den Ereignissen anfordern
- Doppelklick auf die Textbox zum Ereignis

Im nächsten Abschnitt ist zu erfahren, wo die Ergänzungsstücke zu den partiellen Klassendefinitionen in **App.xaml.cs** und **MainWindow.xaml.cs** stecken.

12.3.4 XAML-Verarbeitung beim Erstellen und Starten einer WPF-Anwendung

In diesem Abschnitt beschäftigen wir uns mit der XAML-Verarbeitung beim Erstellen und Starten einer WPF-Anwendung und stoßen dabei auf interessante Details. So wird z. B. die Startmethode **Main()** lokalisiert, die auch bei einer WPF-Anwendung ihre übliche Rolle in der Startphase spielt. Allerdings ist dieses Hintergrundwissen für die Praxis der Anwendungsentwicklung mit dem Visual Studio *nicht* erforderlich, sodass Sie den Abschnitt im Vertrauen auf die WPF-Magie ignorieren können.

Enthält eine WPF-Anwendung XAML-Dateien (= Normalfall), dann lässt unsere Entwicklungsumgebung bei jeder Erstellung der Anwendung zuerst zu jeder Fensterklasse die zugehörige XAML-Datei (z. B. **MainWindow.xaml**) von der **Microsoft-Build-Engine** in eine binäre (validierte und effizient zu verarbeitende) Variante mit der Namenserweiterung **.baml** übersetzen (z. B. **MainWindow.baml**). Diese Datei landet bei einem Projekt mit der Zielplattform **Any CPU** und der Build-Konfiguration **Debug** im Projektunterordner[1]

...\obj\Debug

Die BAML-Dateien zu den Fensterklassen werden als Ressourcen in das vom Compiler erzeugte Assembly eingebunden und beim Programmstart vom XAML-Parser ausgewertet (siehe unten). Dabei entstehen GUI-Objekte mit den im XAML-Code festgelegten Eigenschaften.

Außerdem erzeugt der XAML-Compiler im eben angegebenen Projektunterordner aus jeder XAML-Datei (zu einem Fenster oder zur Anwendung gehörend) eine C# - Quellcodedatei mit einer

---

[1] Die exakten Zielordner bei Verwendung von .NET Core 3.1 oder .NET 5.0 als **Zielframework**:

- **...\obj\Debug\netcoreapp3.1**
- **...\obj\Debug\net5.0-windows**

partiellen Klassendefinition, bei einer WPF-Anwendung mit der Anwendungsklasse `App` und einem einzigen Fenster (Hauptfensterklasse `MainWindow`) also die Dateien:

- **MainWindow.g.cs**[1]
  In der partiellen Klassendefinition zum Hauptfenster
  ```
  public partial class MainWindow :
    System.Windows.Window, System.Windows.Markup.IComponentConnector {
     . . .
  }
  ```
  finden sich die Referenzvariablen zu den benannten Steuerelementen des Fensters, z. B.:
  ```
  internal System.Windows.Controls.Button button;
  ```
  Die Variablennamen stammen aus den **Name** - oder **x:Name** - Attributen der zugehörigen XAML-Instanzelemente (vgl. Abschnitt 12.3.2.1).
  Außerdem findet sich hier die im Fensterklassenkonstruktor (siehe Code-Behind - Datei **MainWindow.xaml.cs** im Abschnitt 12.3.3) aufgerufene Methode **InitializeComponent()**. Diese sorgt durch einen Aufruf der statischen **Application**-Methode **LoadComponent()** für das Laden der BAML-Ressource und damit für das Instanziieren und Initialisieren der zugehörigen Objekte:
  ```
  public void InitializeComponent() {
    if (_contentLoaded) {
     return;
    }
    _contentLoaded = true;
    System.Uri resourceLocater =
     new System.Uri("/CodeBehind;component/mainwindow.xaml", System.UriKind.Relative);
      . . .
    System.Windows.Application.LoadComponent(this, resourceLocater);
      . . .
  }
  ```
  Schließlich enthält die partielle `MainWindow`-Klassendefinition noch die Methode **Connect**(), die von der Schnittstelle **System.Windows.Markup.IComponentConnector** gefordert wird. Hier werden die Ereignisbehandlungsmethoden zu den Steuerelementen registriert, die in der Code-Behind - Datei **MainWindow.xaml.cs** implementiert werden (siehe Abschnitt 12.3.3):
  ```
    void System.Windows.Markup.IComponentConnector.Connect(int connectionId,
                                                           object target) {
        switch (connectionId)
        {
        case 1:
        this.button = ((System.Windows.Controls.Button)(target));

        #line 10 "..\..\MainWindow.xaml"
        this.button.Click += new System.Windows.RoutedEventHandler(this.button_Click);

        #line default
        #line hidden
        return;
        }
        this._contentLoaded = true;
    }
  ```
  Die Klasse `MainWindow` implementiert die Methode **Connect()** *explizit*, sodass im Methodenkopf der Schnittstellenname angegeben werden muss und kein Zugriffsmodifikator gesetzt werden darf (siehe Abschnitt 9.6).

---

[1] Warum es neben **MainWindow.g.cs** im selben Ordner noch eine inhaltsidentische Datei mit dem Namen **MainWindow.g.i.cs** gibt, müssen wir nicht erforschen.

- **App.g.cs**[1]
  Hier findet sich die partielle Definition der von **System.Windows.Application** abgeleiteten Anwendungsklasse mit dem (vom Visual Studio gewählten) Namen App:

```
public partial class App : System.Windows.Application {
    . . .
  public void InitializeComponent() {

    #line 5 "..\..\App.xaml"
    this.Exit += new System.Windows.ExitEventHandler(this.Application_Exit);

    #line default
    #line hidden

    #line 5 "..\..\App.xaml"
    this.StartupUri = new System.Uri("MainWindow.xaml", System.UriKind.Relative);

    #line default
    #line hidden
  }

  [System.STAThreadAttribute()]
   . . .
  public static void Main() {
    CodeBehind.App app = new CodeBehind.App();
    app.InitializeComponent();
    app.Run();
  }
}
```

  Um das Hauptfenster festzulegen, wird in der App-Methode **InitializeComponent()** (nicht zu verwechseln mit der gleichnamigen Methode in der Klasse MainWindow) der **Application**-Eigenschaft **StartupUri** ein **Uri**-Objekt zugewiesen, das auf der XAML-Deklaration der Hauptfensterklasse basiert.

  Schließlich findet sich in der partiellen, automatisch erstellten App-Klassendefinition die Startmethode **Main()**. Im Unterschied zu der manuell erstellten **Main()** - Methode unserer minimalen WPF-Anwendung im Abschnitt 12.2.1 wird das Hauptfenster nicht per **Run()** - Parameter festgelegt. Es ist dem Anwendungsobjekt bereits über seine **StartupUri**-Eigenschaft bekannt.

  Behandlungsmethoden für App-Ereignisse werden in der eben beschriebenen Methode **InitializeComponent()** registriert und in der Code-Behind - Datei **App.xaml.cs** zur Anwendungsklasse implementiert (siehe Abschnitt 12.3.3).

Die mit **#line** startenden Zeilen in **MainWindow.g.cs** bzw. **App.g.cs** enthalten C# - Präprozessordirektiven:[2]

- **#line** *number* "*file*"
  Für die Ausgabe einer Fehlerlokalisierung durch den Compiler werden die Zeilennummer und der Dateiname geändert.
- **#line default**
  Die Fehlerlokalisierung wird auf das Standardverfahren zurückgesetzt.
- **#line hidden**
  Alle Zeilen bis zur nächsten **#line**-Direktive werden vor dem Debugger verborgen, also z. B. bei der schrittweisen Ausführung nicht angezeigt. Auf die Zeilennummerierung hat diese Direktive keinen Einfluss.

---

[1] Warum es neben **App.g.cs** im selben Ordner noch eine inhaltsidentische Datei mit dem Namen **App.g.i.cs** gibt, müssen wir nicht erforschen.

[2] https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/preprocessor-directives/index

Gehen Sie folgendermaßen vor, wenn Sie die generierten Quellcode-Dateien im Visual Studio öffnen wollen:

- Im **Projektmappen-Explorer** über den Symbolschalter **alle Dateien anzeigen** lassen
- Pfad **obj\Debug** mit den generierten Dateien öffnen
- Gewünschte Datei per Doppelklick im Quellcode-Editor öffnen

12.4 Routingereignisse

Wie zu Beginn von Kapitel 12 erwähnt, geht es bei der GUI-Programmierung sehr ereignisreich zu. Folgerichtigerweise wurde für die Windows Presentation Foundation mit den sogenannten *Routingereignissen* eine Ergänzung bzw. Erweiterung der CLR-Ereignistechnik eingeführt. Microsoft nennt zwei Besonderheiten der Routingereignisse im Vergleich zu den im Abschnitt 10.2 behandelten generellen Ereignissen:[1]

- Funktionsweise
  Eine Behandlungsmethode (z. B. zum **Click**-Ereignis) kann nicht nur bei der Ereignisquelle registriert werden, sondern auch bei den übergeordneten UI-Elementen bis hinauf zum Fensterobjekt. Tritt ein Ereignis auf, werden nicht nur die bei der Ereignisquelle registrierten Behandlungsmethoden aufgerufen, sondern nacheinander auch die bei übergeordneten Elementen registrierten Behandlungsmethoden. Dabei kann die Weiterleitung von der Quelle bis zum Fenster erfolgen oder auch umgekehrt. Weil UI-Elementverschachtelungen noch nicht explizit betrachtet wurden, soll ein typisches Beispiel skizziert werden:
  - Ein **TextBlock**-Objekt, das zur Quelle eines **MouseDown**-Ereignisses werden kann,
  - befindet sich in einem **StackPanel**-Layoutcontainer,
  - der zu einem **Button**-Objekt gehört,
  - das von einem **Grid**-Layoutcontainer verwaltet wird,
  - der in einem **Window**-Objekt als Wurzel-Container arbeitet.
- Implementierung
  Ein Routingereignis basiert auf einem Objekt der Klasse **RoutedEvent** und wird vom WPF-Ereignissystem verarbeitet.

Hinsichtlich der automatischen Weiterleitung entlang einer Hierarchie von potentiellen Interessenten zeigt das WPF-Ereignissystem eine Verwandtschaft zur Kommunikation von Ausnahmefehlern durch das Laufzeitsystem (siehe Kapitel 13).

Um jede Verwechslung der im Abschnitt 10.2 beschriebenen Ereignisse mit den nun vorzustellenden Routingereignissen zu vermeiden, werden erstere ab jetzt als *CLR-Ereignisse* bezeichnet.

Weiterhin ist zu betonen, dass in einer WPF-Anwendung keinesfalls alle Ereignisse vom Routing-Typ sind, dass also im WPF-Framework die CLR-Ereignisse nicht ersetzt, sondern ergänzt werden.

Wenn man eine Behandlungsmethode für ein Routingereignis ausschließlich bei der *Ereignisquelle* registriert, was wir bisher getan haben, dann ist der Unterschied zwischen einem normalen Ereignis und einem Routingereignis irrelevant. Wir haben z. B. für den im Abschnitt 6.8 entwickelten RSS-Feed - Reader mit Assistentenhilfe (in der Code-Behind - Datei zur Hauptfensterklasse `MainWindow`) eine Behandlungsmethode zum **Click**-Ereignis des **Button**-Steuerelements erstellt:

---

[1] https://docs.microsoft.com/de-de/dotnet/framework/wpf/advanced/routed-events-overview

```
private void button_Click(object sender, RoutedEventArgs e) {
   . . .
}
```

Diese wird in der vom Visual Studio generierten Quellcodedatei **MainWindow.g.cs** (vgl. Abschnitt 12.3.4) wie bei einem gewöhnlichen Ereignis (vgl. Abschnitt 10.2.2) „an der Quelle“ registriert:

```
this.button.Click += new System.Windows.RoutedEventHandler(this.button_Click);
```

Um diese Verwendung von Routingereignissen mit der vertrauten C# - Syntax für Ereignisse zu ermöglichen, bieten die meisten Routingereignisse einen Wrapper (dt.: eine *Hülle*).

12.4.1 Routingereignisse definieren

Der folgende Quellcode zeigt, wie in der BCL-Klasse **ButtonBase** (Namensraum **System.Windows.Controls.Primitives**), von der z. B. die Klasse **Button** abstammt, das **Click**-Routingereignis (ein Objekt der Klasse **RoutedEvent**) durch die statische **EventManager**-Methode **RegisterRoutedEvent()** erzeugt und registriert wird:[1]

```
public static readonly RoutedEvent ClickEvent = EventManager.RegisterRoutedEvent(
   "Click", RoutingStrategy.Bubble, typeof(RoutedEventHandler), typeof(ButtonBase));
```

Die Objektadresse landet im schreibgeschützten, öffentlichen und statischen Feld **ClickEvent**. Beim Registrieren sind anzugeben:

- Der Ereignisname
- Die Routingstrategie (siehe Abschnitt 12.4.2)
- Die von Behandlungsmethoden zu erfüllende Delegatenklasse
- Der Datentyp des Eigentümers
  Hier gibt man die Klasse an, die das Routingereignis registriert.

Der Name eines statischen Felds vom Typ **RoutedEvent** sollte mit *Event* enden.

In der Regel wird ein CLR-Ereignis als Wrapper (dt.: *Hülle*) um das Routingereignis konstruiert, damit die normale Syntax zum Registrieren von Ereignisbehandlungsmethoden anwendbar ist. Im Beispiel wird das CLR-Ereignis **Click** mit der im Abschnitt 10.2.1 beschriebenen Syntax definiert, wobei die explizit realisierten **add**- und **remove**-Methoden für die Verbindung zum **RoutedEvent**-Objekt sorgen:

```
public event RoutedEventHandler Click {
   add {
      AddHandler(ClickEvent, value);
   }
   remove {
      RemoveHandler(ClickEvent, value);
   }
}
```

Aufgerufen wird das Routingereignis in der **ButtonBase**-Methode **OnClick()**, wobei aber *nicht* wie bei einem CLR-Ereignis das Delegatenobjekt aufgerufen wird (vgl. Abschnitt 10.2.3), sondern die **UIElement**-Methode **RaiseEvent()**:

```
protected virtual void OnClick() {
   RoutedEventArgs newEvent = new RoutedEventArgs(ButtonBase.ClickEvent, this);
   RaiseEvent(newEvent);
   MS.Internal.Commands.CommandHelpers.ExecuteCommandSource(this);
}
```

---

[1] https://source.dot.net/#PresentationFramework/System/Windows/Controls/Primitives/ButtonBase.cs

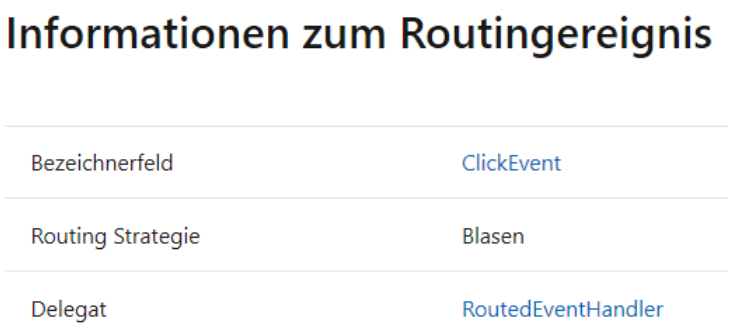
12.4.2 Routingstrategien

Nach der verwendeten Weiterleitungsstrategie sind folgende Kategorien von Routingereignissen zu unterscheiden:

- **Tunnelereignisse**
  Hier kommt als Routingstrategie das **Tunneling** zum Einsatz. Das von einem GUI-Element (z. B. **Image**-Objekt als Inhaltsbestandteil einer **Button**-Oberfläche) ausgelöste Ereignis wird zuerst dem obersten Element in der GUI-Elementhierarchie (also einem **Window**-Objekt) zur Behandlung angeboten, d. h. die dort registrierten Behandlungsmethoden werden ggf. aufgerufen. Anschließend durchläuft das Ereignis die hierarchisch *absteigende* Route bis zur Ereignisquelle. Weil die übergeordneten Elemente das Ereignis *vor* dem unmittelbar betroffenen Element sehen, starten die Namen der Tunnelereignisse mit dem Präfix **Preview**, z. B. **PreviewMouseDown**. Dementsprechend werden Tunnelereignisse gelegentlich auch als *Vorschauereignisse* bezeichnet.
  Beispiel: **PreviewMouseLeftButtonDown**-Ereignis der Klasse **UIElement**
- **Blasenereignisse**
  Hier kommt als Routingstrategie das **Bubbling** zum Einsatz. Zuerst werden die bei der Ereignisquelle registrierten Behandlungsmethoden aufgerufen. Dann werden die bei übergeordneten Elementen in der GUI-Elementhierarchie (bis hinauf zum **Window**-Objekt) registrierten Behandlungsmethoden nacheinander aufgerufen.
  Beispiel: **Click**-Ereignis der Klasse **ButtonBase**
- **Direktereignisse**
  Nur die bei der Quelle registrierten Behandlungsmethoden werden aufgerufen, was der Verarbeitung von normalen CLR-Ereignissen entspricht. Allerdings sind einige Techniken des WPF-Ereignissystems realisiert, z. B. statische Behandlungsmethoden, die für alle Objekte ihrer Klasse zuständig sind und noch vor den Instanz-Behandlungsmethoden aufgerufen werden (siehe Abschnitt 12.4.5).
  Beispiele: **MouseEnter**-Ereignis der Klasse **UIElement**

Ein Tunnel- oder Blasenereignis ist *nicht* erledigt, nachdem *eine* Behandlungsmethode aufgerufen worden ist, sondern es wird entlang der Hierarchie weiter angeboten, bis es seine Endstation erreicht hat oder als behandelt deklariert wird. Dazu ist in einer Behandlungsmethode die Eigenschaft **Handled** des übergebenen Beschreibungsobjekts auf den Wert **true** zu setzen.

Für Routingereignisse wird in der Dokumentation die Routingstrategie angegeben, z. B. beim **Click**-Ereignis der Klasse **ButtonBase**:



Bei einem Routingereignis muss die Behandlungsmethode den folgenden Delegatentyp erfüllen:

public delegate void RoutedEventHandler(object *sender***, RoutedEventArgs** *e***)**

Infolgedessen erhält sie beim Aufruf die folgenden Parameter:

- **object** *sender*
  Dieses Objekt hat die Behandlungsmethode aufgerufen. Bei einem Tunnel- oder Blasenereignis wird durch den ersten Parameter dasjenige Objekt referenziert, bei dem das Ereignis auf seiner Route gerade angekommen ist. In diesem Fall zeigt der erste Parameter also i. A. *nicht* auf die Ereignisquelle.
- **RoutedEventArgs** *e*
  Das Ereignisbeschreibungsobjekt besitzt u. a. die beiden folgenden Eigenschaften:
  - **Source**
    Man erfährt, welches Objekt das Routingereignis ausgelöst hat.
  - **Handled**
    Über diese boolesche Eigenschaft kann der Ereignisbehandlungsstatus eingesehen oder gesetzt werden.

Speziell die mit Eingaben (z. B. per Maus oder Tastatur) beschäftigten Routingereignisse treten meist als *Paar* aus einem Tunnel- und einem Blasenereignis auf (z. B. **PreviewMouseDown** und **MouseDown**), wobei zur Abfolge gilt:

- Zunächst wird das Tunnelereignis ausgelöst. Auf dem Weg von der Wurzel der Elementhierarchie bis zur Ereignisquelle werden registrierte Behandlungsmethoden sukzessive aufgerufen.
- Wenn das Tunnelereignis seine Endstation erreicht, wird das zugehörige Blasenereignis ausgelöst.
- Ein Tunnelereignis als behandelt zu markieren, wirkt sich analog auf das zugehörige Blasenereignis aus, weil beide Routingereignisse *dasselbe* Ereignisbeschreibungsobjekt (aus der Klasse **RoutedEventArgs** oder aus einer abgeleiteten Klasse) verwenden. Tunnelereignisse dienen meist dazu, eine Ereignisbehandlung zu blockieren oder vorzubereiten.
- Auch ein als behandelt markiertes Ereignis setzt seine Wanderung fort und wird Behandlungsmethoden angeboten, die auf besondere Weise über die **UIElement**-Methode **AddHandler()** registriert worden sind (mit dem Wert **true** für den booleschen Parameter **handledEventsToo**).

Eine Ausnahme vom Paarungsprinzip ist das Blasenereignis **Click** der Klasse **ButtonBase**, zu dem kein Tunnelereignis existiert.

12.4.3 Praktische Bedeutung und Einsatzempfehlungen

Nachdem etliche technische Details zu den Routingereignissen geklärt worden sind, geht es nun um die praktische Relevanz der verschiedenen Techniken und um Empfehlungen zur Verwendung.

Generell ist das Routing vor allem bei Eingabeereignissen wichtig, wobei elterliche Steuerelemente von Eingabeereignissen erfahren, die bei einem ihrer Kinder aufgetreten sind. Wenn die Ereignisquelle relevant ist, kann sie über die Eigenschaft **Source** des übergebenen Beschreibungsobjekts ermittelt werden.

Blasenereignisse bewähren sich z. B. in den folgenden Einsatzszenerien:

- **Gemeinsame Ereignisbehandlung für mehrere Steuerelemente**
  Befinden sich z. B. in einem Container mehrere **Button**-Objekte, und sollen deren **Click**-Ereignisse durch eine gemeinsame Methode behandelt werden, dann genügt eine einmalige Registrierung dieser Methode beim Container. Stünde hinter **Click** *kein* „aufperlendes“ Routingereignis, dann müsste die Registrierung für jeden Schalter wiederholt werden.

- **Ereignisse für zusammengesetzte Steuerelemente**
  Ein **Button**-Objekt kann Inhalte aufnehmen, z. B. ein **Image**- und ein **TextBlock**-Objekt:

```
<Button Name="button" ... Click="button_Click">
  <StackPanel>
    <Image  Source="vs.png"/>
    <TextBlock Text="Click"/>
  </StackPanel>
</Button>
```

  Die ein **Click**-Ereignis konstituierenden elementaren Ereignisse wie **MouseLeftButtonDown** und **MouseLeftButtonUp** treten jeweils bei einem Inhaltsbestandteil auf, z. B. ...
  - beim **Image**-Objekt das Routingereignis **MouseLeftButtonDown**
  - und wegen einer leichten Mausbewegung beim **TextBlock**-Objekt das Routingereignis **MouseLeftButtonUp**.

  Weil alle Ereignisse an das **Button**-Objekt weitergeleitet werden, kann dort ein Klick auf den Schalter erkannt und als neues, „höherwertigeres“ Routingereignis gefeuert werden.

Tunnelereignisse werden von Anwendungsprogrammen viel seltener behandelt als Blasenereignisse. Manchmal ist es aber für ein elterliches UI-Element relevant, *vor* seinen Kindern von einem Ereignis zu erfahren. So wird es möglich, ...

- **ein Ereignis zu blockieren**
  Dazu wird in der Behandlungsmethode zu einem Tunnelereignis die **Handled**-Eigenschaft des übergebenen Beschreibungsobjekts auf den Wert **true** gesetzt. Wenn z. B. ein **TextBox**-Steuerelement nur Ziffern erhalten soll, kann man über das Ereignis **PreviewTextInput** eine Eingabevalidierung vornehmen und das zugehörige **TextInput**-Ereignis blockieren, wenn ein irreguläres Zeichen eingegeben wurde (zu weiteren Details der Eingabevalidierung siehe MacDonald 2012, S. 124).
- **eine Ereignisbehandlung vorzubereiten**

Der Abschnitt 12.4 muss sich auf generelle Informationen über das WPF-Ereignissystem beschränken. Über spezielle Ereignisgruppen (z. B. Maus- und Tastaturereignisse) informiert z. B. MacDonald (2012, Kapitel 4).

12.4.4 Eine Beobachtungsstudie

Um das WPF-Ereignissystem bei der Arbeit beobachten zu können, erstellen wir eine Anwendung mit der folgenden Hierarchie von UI-Elementen:

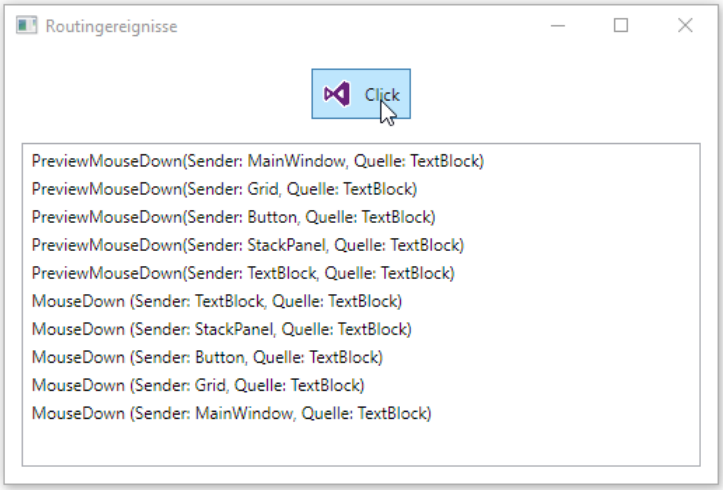
Ein Wurzelelement aus der von **Window** abstammenden Klasse `MainWindow`,
darin ein Layoutcontainer aus der Klasse **Grid** (siehe Abschnitt 12.6.1),
in der **Grid**-Zelle (0, 0) u. a. ein Befehlsschalter aus der Klasse **Button**,
darin ein Layoutcontainer aus der Klasse **StackPanel** (siehe Abschnitt 12.6.2),
darin ein Objekt der Klasse **Image**
und ein Objekt der Klasse **TextBlock**

In der **Grid**-Zelle (0, 0) befindet sich außerdem noch ein **ListBox**-Objekt, das zum Protokollieren von Ereignissen dient.

Das Programm beobachtet die folgenden Routingereignisse bei allen betroffenen GUI-Elementen:

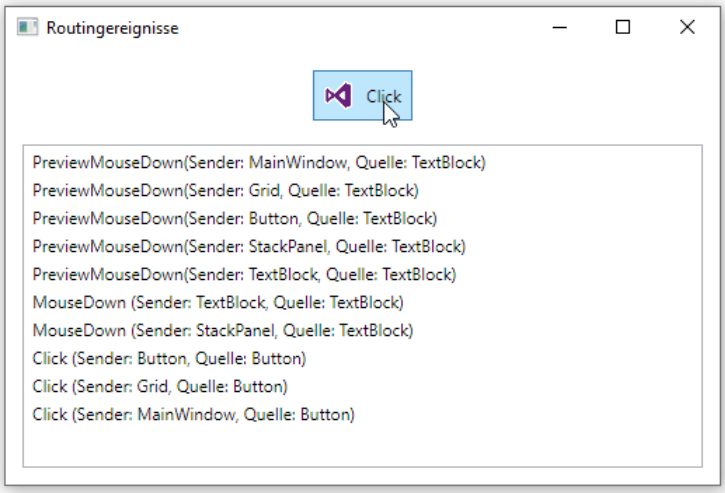
- Tunnelereignis **PreviewMouseDown** aus der Klasse **UIElement**
- Blasenereignis **MouseDown** aus der Klasse **UIElement**
- Blasenereignis **Click** aus der Klasse **ButtonBase** (Basisklasse von **Button**)

Beim folgenden Einsatz



hat das **TextBlock**-Objekt wegen eines *rechten* Mausklicks (Maustaste drücken und loslassen) das Tunnelereignis **PreviewMouseDown** gefeuert. Das Ereignis ist gemäß Abschnitt 12.4.2 mit dem Wurzelelement beginnend von allen GUI-Elementen auf dem Weg von der Wurzel bis zur Quelle behandelt worden. Danach wurde das zugehörige Blasenereignis aus der Klasse **MouseDown** ausgelöst, das den umgekehrten Weg von der Quelle bis zur Wurzel genommen hat.

Nach einem *linken* Mausklick (Maustaste drücken und loslassen) auf das **TextBlock**-Objekt zeigt sich eine andere Ereignishistorie:



Zu Beginn zeigt sich kein Unterschied zum Rechtsklick: Das vom **TextBlock**-Objekt gefeuerte Tunnelereignis **PreviewMouseDown** wird an der Wurzel beginnend auf allen Hierarchieebenen behandelt. Danach startet erwartungsgemäß das aufperlende Gegenstück **MouseDown** an der Quelle, endet allerdings überraschend beim **StackPanel**-Objekt. Stattdessen feuert das **Button**-Objekt das Blasenereignis **Click**, das seinen Weg nach oben bis zur Wurzel der GUI-Elementhierarchie nimmt.

Das Programm verwendet für die drei beobachteten Routingereignisse jeweils eine Behandlungsmethode, die bei allen UI-Elementen in der oben beschriebenen Verschachtelung registriert wird. Neben dem Ereignistyp protokollieren die Behandlungsmethoden auch ...

- den Aufrufer der Methode
  Sein Datentyp wird über das erste Parameterobjekt festgestellt.
- die Ereignisquelle
  Ihr Datentyp wird über die **Source**-Eigenschaft des zweiten Parameterobjekts festgestellt.

Die Behandlungsmethoden werden in der Code-Behind - Datei mit der partiellen Definition der Fensterklasse untergebracht (vgl. Abschnitt 12.3.3):

```
using System.Windows;
using System.Windows.Input;

namespace Routingereignisse {
    public partial class MainWindow : Window {
        public MainWindow() {
            InitializeComponent();
        }
        // Tunnelereignis PreviewMouseDown
        private void HandlePreviewMouseDown(object sender, MouseButtonEventArgs e) {
            listBox.Items.Add($"PreviewMouseDown(Sender: {sender.GetType().Name}, " +
                              $"Quelle: {e.Source.GetType().Name})");
        }
        // Blasenereignis MouseDown
        private void HandleMouseDown(object sender, MouseButtonEventArgs e) {
            listBox.Items.Add($"MouseDown (Sender: {sender.GetType().Name}, " +
                              $"Quelle: {e.Source.GetType().Name})");
        }
        // Blasenereignis Click
        private void HandleClick(object sender, RoutedEventArgs e) {
            listBox.Items.Add($"Click (Sender: {sender.GetType().Name}, " +
                              $"Quelle: {e.Source.GetType().Name})");
        }
    }
}
```

In den Namen der Ereignisbehandlungsmethoden gibt der erste Bestandteil nicht wie sonst üblich den Absender des Ereignisses an, weil die Methoden für verschiedene Absender zuständig sind.

Um die Protokolleinträge vom **ListBox**-Steuerelement anzeigen zu lassen, werden sie dem Kollektionsobjekt aus der der Klasse **ItemCollection** übergeben, das über die **ListBox**-Eigenschaft **Items** ansprechbar ist und u. a. die Methode **Add()** beherrscht (vgl. Abschnitt 12.7.4.5.1).

Das Registrieren der Ereignisbehandlungsmethoden ist per XAML-Code erheblich einfacher zu bewerkstelligen als mit C# - Code:

```
<Window x:Class="Routing.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Routing" Height="350" Width="525"
        PreviewMouseDown="HandlePreviewMouseDown" MouseDown="HandleMouseDown"
        ButtonBase.Click="HandleClick">
    <Grid PreviewMouseDown="HandlePreviewMouseDown" MouseDown="HandleMouseDown"
          ButtonBase.Click="HandleClick">
        <Button Name="button" HorizontalAlignment="Center" Margin="198,23,192,252"
                PreviewMouseDown="HandlePreviewMouseDown" MouseDown="HandleMouseDown"
                Click="HandleClick">
            <StackPanel Name="stackPanel" Orientation="Horizontal"
                        PreviewMouseDown="HandlePreviewMouseDown"
                        MouseDown="HandleMouseDown">
                <Image  Name="image" Width="20" Margin="5"
                        Source="rss.gif" VerticalAlignment="Center"
                        PreviewMouseDown="HandlePreviewMouseDown"
                        MouseDown="HandleMouseDown"/>
                <TextBlock Name="textBlock" HorizontalAlignment="Right" Margin="5"
                           Text="Click" VerticalAlignment="Center"
                           PreviewMouseDown="HandlePreviewMouseDown"
                           MouseDown="HandleMouseDown"/>
            </StackPanel>
        </Button>
        <ListBox Name="listBox" Margin="12,76,12,12" />
    </Grid>
</Window>
```

Wie im Abschnitt 12.3.2.4 beschrieben, kann die XAML-Attributsyntax auch für Ereignisse verwendet werden, wobei der jeweilige Methodenname als Zeichenfolgenliteral anzugeben ist.

Besondere Aufmerksamkeit verdient die **Click**-Ereignis - Registrierung für das **Window**- bzw. für das **Grid**-Element, z. B.:

```
<Window x:Class="Routing.MainWindow"
   . . .
        ButtonBase.Click="HandleClick">
   . . .
</Window>
```

Das **Click**-Ereignis kann beim Wurzelelement registriert werden, obwohl es in der Klasse **Window** nicht definiert ist. Dies ist möglich, weil **Click** ein Blasenereignis ist und folglich als sogenanntes **angefügtes Ereignis** (engl.: *attached event*) auf übergeordneten Ebenen der UI-Elementhierarchie behandelt werden kann. Bei der Registrierung ist verständlicherweise vor dem Ereignisnamen der Name der Klasse anzugeben, zu der das Ereignis gehört. Im Beispiel hat die Klasse **Button** das Routingereignis von ihrer Basisklasse **ButtonBase** geerbt.

Die Registrierung

```
ButtonBase.Click="HandleClick"
```

wurde direkt in den XAML-Code eingetragen. In der Ereignisliste des **Eigenschaften**-Fensters zum **Window**-Objekt fehlen angefügte Ereignisse.

Die angefügten Ereignisse ermöglichen die im Abschnitt 12.4.3 beschriebene rationelle Behandlung von Blasenereignissen: Befinden sich mehrere Steuerelemente desselben Typs in einem Container lässt sich über ein angefügtes Ereignis auf Container-Ebene eine gemeinsame Ereignisbehandlung durch eine einzige Methode realisieren. In dieser Behandlungsmethode kann die Ereignisquelle über die **Source**-Eigenschaft des übergebenen **RoutedEventArgs**-Objekts identifiziert werden.

Das komplette Projekt ist im folgenden Ordner zu finden

…\BspUeb\WPF\Routingereignisse\Beobachtungsstudie

12.4.5 Ereignisbehandlung durch statische Methoden

Das WPF-Ereignissystem ermöglicht es einer von **System.Windows.DependencyObject** abstammenden Klasse, für ein Routingereignis eine *statische* Behandlungsmethode zu registrieren, die damit für alle Objekte der Klasse zuständig ist. Ist zusätzlich bei einem Objekt der Klasse eine Behandlungsmethode für dasselbe Routingereignis registriert, wird die statische Behandlungsmethode *vor* der objektbezogenen ausgeführt.

Zur Demonstration definieren wir eine **Button**-Ableitung namens `MaiButton,` die eine statische **ClickEvent**-Behandlungsmethode definiert und im statischen Konstruktor über die statische Methode **RegisterClassHandler()** der Klasse **EventManager** beim WPF-Ereignissystem registriert:

```
using System;
using System.Windows;
using System.Windows.Controls;
```

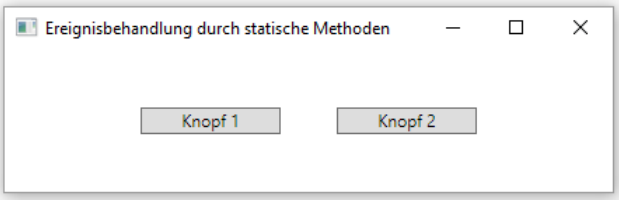
```
class MaiButton : Button {
   protected static void StaticClickEventHandler(object sender, RoutedEventArgs e) {
      MessageBox.Show("Statischer Click-Handler der Klasse MaiButton");
   }
   static MaiButton() {
      EventManager.RegisterClassHandler(typeof(MaiButton), ClickEvent,
         new RoutedEventHandler(StaticClickEventHandler), true);
   }
}
```

Der zweite **RegisterClassHandler()** - Parameter ist vom Typ **RoutedEvent**. Unter dem Namen **ClickEvent** hat die Klasse **ButtonBase** beim WPF-Ereignissystem dasjenige Routingereignis registriert, das dank Wrapper (vgl. Abschnitt 12.4.1) als CLR-Ereignis namens **Click** angesprochen werden kann.

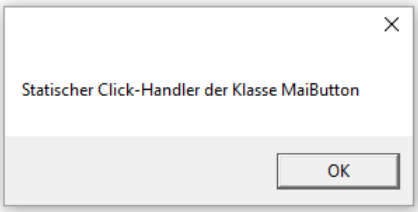
Die folgende WPF-Anwendung (handgestrickt ohne XAML)

```
class StaticEventHandling : Window {
   StaticEventHandling() {
      Height = 140; Width = 450;
      Title = "Ereignisbehandlung durch statische Methoden";
      StackPanel lm = new StackPanel();
      lm.Orientation = Orientation.Horizontal;
      lm.HorizontalAlignment = HorizontalAlignment.Center;
      lm.VerticalAlignment = VerticalAlignment.Center;
      Content = lm;
      MaiButton k1 = new MaiButton(); k1.Content = "Knopf 1"; k1.Width = 100;
      k1.Margin = new Thickness(20); lm.Children.Add(k1);
      MaiButton k2 = new MaiButton(); k2.Content = "Knopf 2"; k2.Width = 100;
      k2.Margin = new Thickness(20); lm.Children.Add(k2);

      k1.Click += new RoutedEventHandler(knopf1Click);
   }

   private void knopf1Click(object sender, RoutedEventArgs e) {
      MessageBox.Show("Click-Handler von Knopf 1");
   }

   [STAThread]
   static void Main() {
      new Application().Run(new StaticEventHandling());
   }
}
```

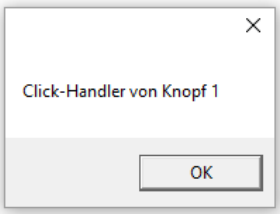
präsentiert zwei Befehlsschalter



aus der Klasse `MaiButton`, wobei der linke über eine eigene **Click**-Behandlungsmethode verfügt. Beim Klick auf einen beliebigen Schalter findet zunächst die klassenbezogene Ereignisbehandlung statt:



Wurde der linke Schalter getroffen, findet danach auch noch die objektbezogene Ereignisbehandlung statt:



12.5 Abhängigkeitseigenschaften

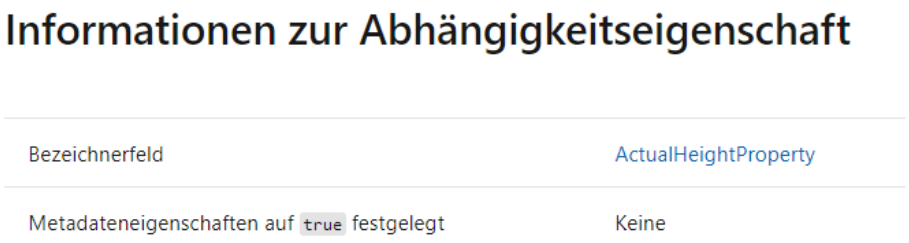
Die WPF-Designer haben nicht nur die CLR-Ereignisse durch die leistungsfähigeren Routingereignisse ergänzt, sondern auch zu den CLR-Eigenschaften, die meist aus einer Instanzvariablen und einem Methodenpaar für den lesenden bzw. schreibenden Zugriff bestehen (vgl. Abschnitt 5.5), mit den sogenannten *Abhängigkeitseigenschaften* (engl.: *dependency properties*) eine funktional erheblich erweiterte Variante geschaffen.

Wie es der Name vermuten lässt, kann die Ausprägung einer Abhängigkeitseigenschaft auf flexible Weise durch verschiedene Quellen beeinflusst werden (z. B. durch Stile, Benutzereinstellungen, Animationen, elterliche Steuerelemente). Außerdem können Ausprägungen validiert und Änderungen an andere Abhängigkeitseigenschaften gemeldet werden (Datenbindung).

Um die neue Funktionalität zu realisieren, wurde keine .NET - Programmiersprache angepasst (von XAML mal abgesehen), sondern das WPF-Framework entsprechend ausgestattet. Alle Klassen, die Abhängigkeitseigenschaften verwenden sollen, müssen von der Klasse **DependencyObject** im Namensraum **System.Windows** abstammen.

Die meisten Abhängigkeitseigenschaften können durch eine Hüllenkonstruktion auch wie gewöhnliche CLR-Eigenschaften verwendet werden. Obwohl wir in Beispielen schon etliche WPF-Abhängigkeitseigenschaften verwendet haben, ist uns bisher keine Besonderheit gegenüber gewöhnlichen CLR-Eigenschaften aufgefallen.

Ob es sich bei einer Eigenschaft einer WPF-Steuerelementklasse um eine Abhängigkeitseigenschaft handelt, verrät die Anwesenheit der **Dependency Property Information** in der BCL-Dokumentation, z. B.:



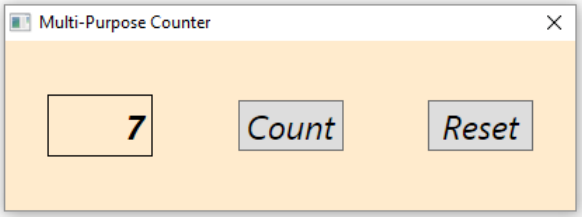
Inspiziert man z. B. die über 130 Eigenschaften der Klasse **Button**, dann finden sich nur wenige, die *keine* Abhängigkeitseigenschaften sind.

Die allgemeine Technik und Funktionsvielfalt der Abhängigkeitseigenschaften darzustellen (siehe z. B. MacDonald 2012, Natan 2010), würde vermutlich die Motivation der Leser auf eine harte Probe stellen. Ein komplettes Verständnis benötigen ohnehin nur solche Entwickler, die eigene WPF-Steuerelemente entwerfen und dabei Abhängigkeitseigenschaften anbieten wollen. Der aktuelle Abschnitt beschränkt sich auf zwei schon beim WPF-Einstieg unvermeidliche bzw. nützliche Funktionen des WPF-Eigenschaftssystems:

- Bei vielen WPF-Abhängigkeitseigenschaften wird die einem GUI-Element zugewiesene Ausprägung (z. B. die Schriftgröße) auf eingeschachtelte GUI-Elemente übertragen („vererbt“), falls keine andere Eigenschaftsquelle dominiert
- Mit den sogenannten *angefügten Eigenschaften* werden spezielle Abhängigkeitseigenschaften vorgestellt, die zur Verwaltung der in Layoutcontainern enthaltenen Steuerelemente unverzichtbar und auch für andere Zwecke nützlich sind.

12.5.1 Eigenschaftsübertragung auf eingeschachtelte Elemente

Bei vielen WPF-Abhängigkeitseigenschaften findet eine sogenannte *Eigenschaftswertvererbung* statt, d.h. die an ein Element in der GUI-Hierarchie eines Fensters vergebene Ausprägung wird auf eingeschachtelte Elemente übertragen. Allerdings hängt die Eigenschaftsausprägung bei einem konkreten eingeschachtelten Element eventuell noch von anderen Quellen ab. Im folgenden Beispiel (vgl. Abschnitt 12.7.4 über Befehlsschalter)

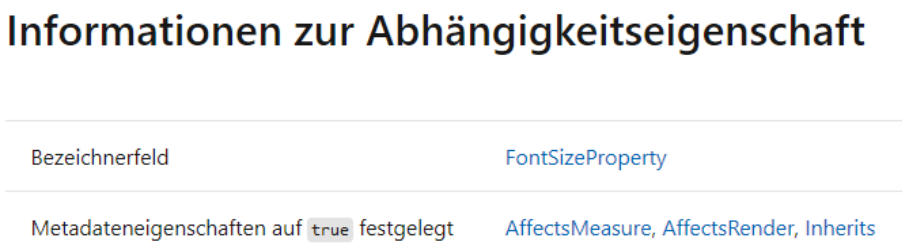


werden für das XAML-Wurzelelement **Window** eine Hintergrundfarbe, eine Schriftgröße und ein Schriftstil festgelegt:

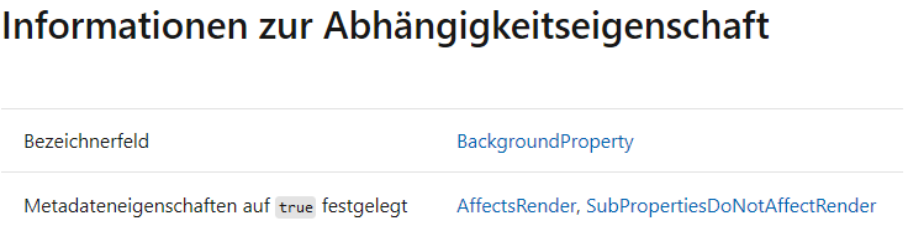
```
<Window x:Class="Button.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Multi-Purpose Counter" Height="149" Width="412" ResizeMode="NoResize"
        Background="BlanchedAlmond" FontSize="24" FontStyle="Italic">
  <Grid Button.Click="Button_Click">
    <Grid.ColumnDefinitions>
      <ColumnDefinition /> <ColumnDefinition /> <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Label Name="label" Content="0" Width="75" HorizontalAlignment="Center"
          VerticalAlignment="Center" BorderThickness="1" BorderBrush="Black"
          HorizontalContentAlignment="Right" FontWeight="Bold"  />
    <Button Name="count" Content="_Count" Width="75" Grid.Column="1"
          HorizontalAlignment="Center" VerticalAlignment="Center" IsDefault="True" />
    <Button Name="reset" Content="_Reset" Width="75" Grid.Column="2"
          HorizontalAlignment="Center" VerticalAlignment="Center">
    </Button>
  </Grid>
</Window>
```

Die Schriftattribute werden „vererbt“, die Hintergrundfarbe hingegen nicht. Weil das **Label**-Objekt als Hintergrundfarbe die transparente Voreinstellung besitzt, sieht es so aus, als würde es die Hintergrundfarbe des Fensters übernehmen.

Ob die Werte einer Eigenschaft auf eingeschachtelte Elemente übertragen werden, hängt davon ab, ob in ihrem Metadaten-Objekt die (Meta-)Eigenschaft **Inherits** auf **true** gesetzt ist. Bei der Eigenschaft **FontSize** ist das der Fall, wie eine Inspektion der BCL-Dokumentation zeigt:



Dasselbe gilt für die Eigenschaft **FontStyle**. Von der Eigenschaft **Background** wird die Übertragung von Eigenschaftswerten hingegen *nicht* unterstützt:



Es ist für die Übertragung der Eigenschaftswerte vom Fenster auf die **Button**-Steuerelemente *kein* Problem, dass der zwischengeschaltete **Grid**-Container die Eigenschaften **FontSize** und **FontStyle** nicht kennt.

Man spricht bei der Übertragung von Eigenschaftsausprägungen auf eingeschachtelte Elemente von *Eigenschaftswertvererbung* (engl. *property value inheritance*), doch hat diese Spezialität des WPF-Eigenschaftssystems *nichts* mit der im Kapitel 7 beschriebenen Ableitung von Klassen zu tun.[1]

12.5.2 Angefügte Eigenschaften

Bei den Visual Studio - Projektvorlagen **WPF-App (.NET)** und **WPF-App (.NET Framework)** kommt per Voreinstellung ein **Grid**-Layoutcontainer zum Einsatz, der im allgemeinen mehrere Zeilen und/oder Spalten zur Platzierung der enthaltenen Steuerelemente verwaltet. Im Abschnitt 12.6.1.1 ist zu erfahren, wie man über **ColumnDefinition**- und **RowDefinition**-Elemente die Struktur eines **Grid**-Containers festlegt. Fügt man ein Steuerelement ohne Ortsangabe in einen **Grid**-Container ein, dann landet es in der Zelle (0, 0), also oben links. Um für ein Steuerelement eine alternative Zelle festzulegen, sind im zugehörigen XAML-Element die Attribute **Grid.Row** bzw. **Grid.Column** mit der null-basierten Nummer der Zeile bzw. Spalte zu versorgen. Hier wird ein **Button**-Objekt in die Zelle (1, 1) einer **Grid**-Matrix mit drei Spalten und zwei Zeilen gesteckt:

```
<Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition /> <ColumnDefinition /> <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition /> <RowDefinition />
    </Grid.RowDefinitions>
    <Button Name="butt" Content="Knopf"
            HorizontalAlignment="Center" VerticalAlignment="Center"
            Grid.Row="1" Grid.Column="1" />
. . .
</Grid>
```

Während sich in der Steuerelementklasse **System.Windows.Controls.Button** zu XAML-Attributen wie **Content**, **HorizontalAlignment** etc. jeweils eine korrespondierende Eigenschaft findet, sucht

[1] https://docs.microsoft.com/de-de/dotnet/framework/wpf/advanced/property-value-inheritance

man in der Klassendefinition die Eigenschaften **Grid.Row** und **Grid.Column** oder auch **Row** und **Column** vergeblich.

Grid.Row und **Grid.Column** sind sogenannte *angefügte Eigenschaften* (engl.: *attached properties*), die von der Klasse **Grid** zur Verfügung gestellt werden, damit die im **Grid**-Layoutcontainer verwalteten GUI-Elemente ihre Positionswünsche formulieren können.

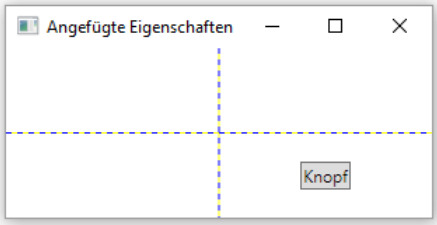
Bei einer angefügten Eigenschaft handelt es sich um eine Abhängigkeitseigenschaft mit den folgenden Besonderheiten:

- Bei der Eigenschaftsvergabe ist kein Objekt der definierenden Klasse (im Beispiel: **Grid**) betroffen, sondern ein Objekt einer anderen Klasse (im Beispiel: **Button**).
- Eine angefügte Eigenschaft kann im C# - Quellcode *nicht* wie eine normale CLR-Eigenschaft verwendet werden. Stattdessen sind statische **Set**- und **Get**-Methoden vorhanden, was gleich für das Beispiel demonstriert wird.

Im Beispiel korrespondieren zur XAML-Syntax die statischen **Grid**-Methoden **SetRow()** und **SetColumn()**, die als Parameter jeweils ein Objekt der indirekt von **DependencyObject** abstammenden Klasse **UIElement** und eine Positionsangabe (Zeilen- oder Spaltennummer) erwarten. Im folgenden Fensterklassenkonstruktor wird *ohne* XAML-Hilfe ein (2 × 2) - **Grid** erstellt und ein **Button**-Objekt in die Zelle (1, 1) gesteckt, um die Verwendung der statischen **Grid**-Methoden **SetRow()** und **SetColumn()** zu demonstrieren:

```
using System;
using System.Windows;
using System.Windows.Controls;

class AttachedProperties : Window {
   AttachedProperties() {
      Title = "Angefügte Eigenschaften";
      Grid grid = new Grid();
      grid.ColumnDefinitions.Add(new ColumnDefinition());
      grid.ColumnDefinitions.Add(new ColumnDefinition());
      grid.RowDefinitions.Add(new RowDefinition());
      grid.RowDefinitions.Add(new RowDefinition());
      grid.ShowGridLines = true;
      Content = grid;

      Button butt = new Button();
      butt.Content = "Knopf";
      butt.HorizontalAlignment = HorizontalAlignment.Center;
      butt.VerticalAlignment = VerticalAlignment.Center;
      Grid.SetRow(butt, 1);
      Grid.SetColumn(butt, 1);

      grid.Children.Add(butt);
   }

   [STAThread]
   static void Main() {
      Application app = new Application();
      AttachedProperties hf = new AttachedProperties();
      app.Run(hf);
   }
}
```

Das **Button**-Objekt erscheint in der gewünschten Zelle des **Grid**-Layoutcontainers:



Damit die korrekte Positionierung gut erkennbar ist, wurde über die Eigenschaft **ShowGridLines** die Zellenstruktur des **Grid**-Layoutcontainers sichtbar gemacht:

```
grid.ShowGridLines = true;
```

Als Gegenstücke zu den schreibenden Methoden **Grid.SetRow()** und **Grid.SetClumn()** sind die lesenden Methoden **Grid.GetRow()** und **Grid.GetClumn()** vorhanden.

Die zu einer angefügten Eigenschaft gehörige statische Schreib- bzw. Lesemethode (z. B. **Grid.SetRow()** bzw. **Grid.GetRow()**) ruft ihrerseits beim betroffenen Objekt die von der Klasse **DependencyObject** geerbte Instanzmethode **SetValue()** bzw. **GetValue()** auf, wobei die angefügte Eigenschaft als erster Parameter übergeben wird, z. B.:[1]

```
public static void SetRow(UIElement element, int value) {
   if (element == null) {
      throw new ArgumentNullException("element");
   }
   element.SetValue(RowProperty, value);
}
```

Weil die Methoden **SetValue()** und **GetValue()** öffentlich sind, kann man sie auch direkt aufrufen und somit die statischen Methoden der angefügten Eigenschaft übergehen. Im Beispiel kann man die Anweisungen

```
Grid.SetRow(butt, 1);
Grid.SetColumn(butt, 1);
```

ersetzen durch:

```
butt.SetValue(Grid.RowProperty, 1);
butt.SetValue(Grid.ColumnProperty, 1);
```

Durch die angefügten Eigenschaften werden zwei Vorteile realisiert, welche die Erweiterbarkeit der Windows Presentation Foundation fördern (MacDonald 2012, S. 35):

- Die Steuerelementklassen werden nicht mit Eigenschaften belastet, die lediglich in bestimmten Kontexten relevant sind.
- Es können jederzeit weitere Layoutcontainer entwickelt werden, die zusätzliche Eigenschaften zur Verwaltung ihrer Elemente benötigen.

12.6 Layoutcontainer

Die in einer WPF-Anwendung definierten Fenster besitzen mehr oder weniger viele Steuerelemente. Zur Verwaltung der Steuerelemente (z. B. Neuberechnung von Positionen und Größen bei einer Änderung der Fenstergröße) dienen Layoutcontainer. Bei einer neuen WPF-Anwendung wird vom Visual Studio für das Hauptfenster ein Layoutcontainer aus der Klasse **Grid** (Namensraum

[1] Der Quellcode zur Klasse **Grid** in der BCL zu .NET 5.0 kann über die folgende Webseite inspiziert werden: https://source.dot.net/

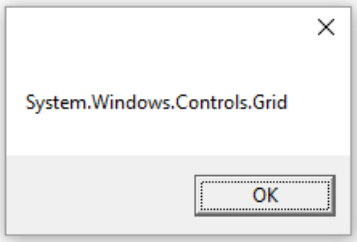
System.Windows.Controls) vorgeschlagen. Dies zeigt ein Blick auf den XAML-Code des Hauptfensters:[1]

```
<Window x:Class="LayoutContainer.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

Wird der Vorschlag übernommen, entsteht beim Programmstart ein **Grid**-Objekt, das der **Content**-Eigenschaft des Hauptfensters zugewiesen wird. Um dieses Detail zu demonstrieren, wird in der folgenden Fensterklassendefinition eine Behandlungsmethode zum **Window**-Ereignis **Loaded** erstellt und registriert:

```
using System.Windows;
namespace LayoutContainer {
    public partial class MainWindow : Window {
        public MainWindow() {
            InitializeComponent();
            Loaded += Window_Loaded;
        }
        private void Window_Loaded(object sender, RoutedEventArgs e) {
            MessageBox.Show(Content.ToString());
        }
    }
}
```

Beim Programmstart erfährt man:



Wenn wir im WPF-Designer Steuerelemente per Drag & Drop aus der **Toolbox** auf das Hauptfenster befördern, landen diese im **Grid**-Element, z. B.:

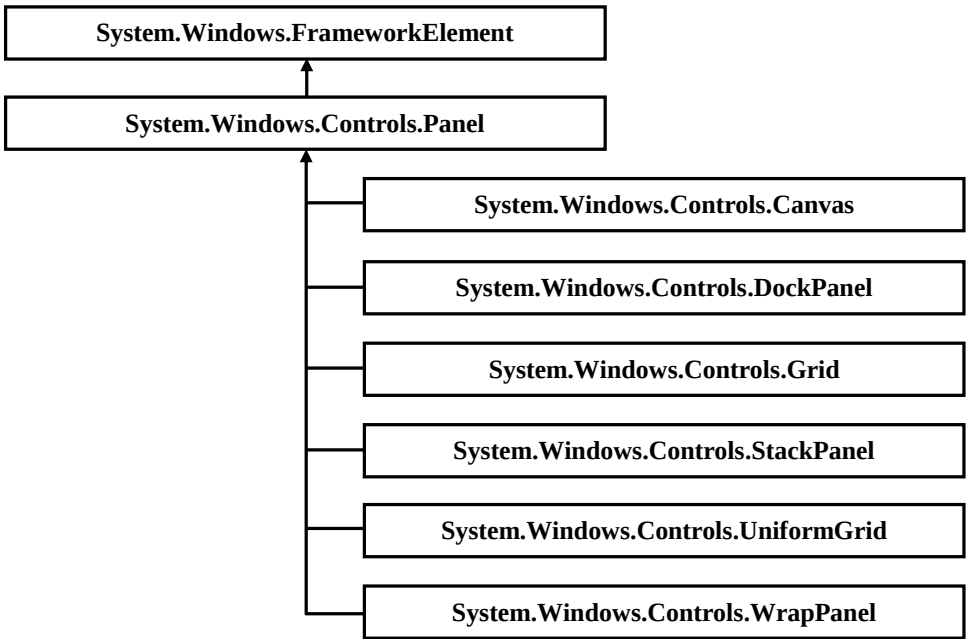
```
<Window x:Class="LayoutContainer.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Label x:Name="label" Content="Label" HorizontalAlignment="Left"
         Margin="30,30,0,0" VerticalAlignment="Top"/>
    </Grid>
</Window>
```

Von der Gitterstruktur, die der Klassenname **Grid** erwarten lässt, ist noch nichts zu sehen. Per Voreinstellung ist nur eine Zeile und eine Spalte vorhanden, und eingefügte Steuerelemente landen in der **Grid**-Zelle (0, 0). Wie man die **Grid**-Flexibilität nutzt, erfahren Sie ansatzweise im Abschnitt 12.6.1.

---

[1] Der Übersichtlichkeit halber wurden überflüssige XAML-Namensraumdeklarationen entfernt.

Alle WPF-Layoutcontainer stammen von der Klasse **System.Windows.Controls.Panel** ab. Hier sind die meistbenutzten Klassen zu sehen:



Von den zahlreichen Membern, die alle Layoutcontainer von ihrer Basisklasse **Panel** erben, ist besonders die Eigenschaft **Children** zu erwähnen, die auf ein Objekt der Klasse **UIElementCollection** zeigt, das die im Container enthaltenen **UIElement**-Objekte verwaltet. Es ist die Inhaltseigenschaft der Klasse **Grid** (vgl. Abschnitt 12.3.2.3.5):

```
[ContentProperty("Children")]
public abstract class Panel : FrameworkElement, IAddChild {
 . . .
   public UIElementCollection Children {
      get {
         return InternalChildren;
      }
   }
. . .
}
```

Für die Top-Level - Fenster einer WPF-Anwendung verwendet man in der Regel ein **Grid**- oder ein **DockPanel**-Objekt als Layoutcontainer. Ein flexibles Fensterdesign erfordert oft geschachtelte Layoutcontainer, und dabei finden auch die übrigen Klassen Verwendung.

12.6.1 Grid

In diesem Abschnitt werden einige Details zum voreingestellten Layoutcontainer einer WPF-Anwendung beschrieben.

12.6.1.1 Zeilen und Spalten definieren

Um die Zeilen bzw. Spalten eines **Grid**-Objekts per XAML zu definieren, ergänzt man im **Grid**-Element untergeordnete Eigenschaftselemente vom Typ **Grid.RowDefinitions** bzw. **Grid.ColumnDefinitions** (vgl. Abschnitt 12.3.2.3 zur XAML-Syntax für Eigenschaftselemente).

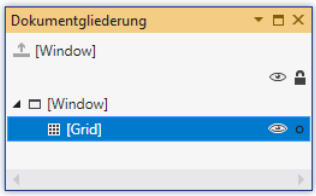
Im Element **Grid.RowDefinitions** werden einzelne Zeilen durch **RowDefinition**-Elemente vereinbart, z. B.:

```
<Grid>
   <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition/>
   </Grid.RowDefinitions>
</Grid>
```

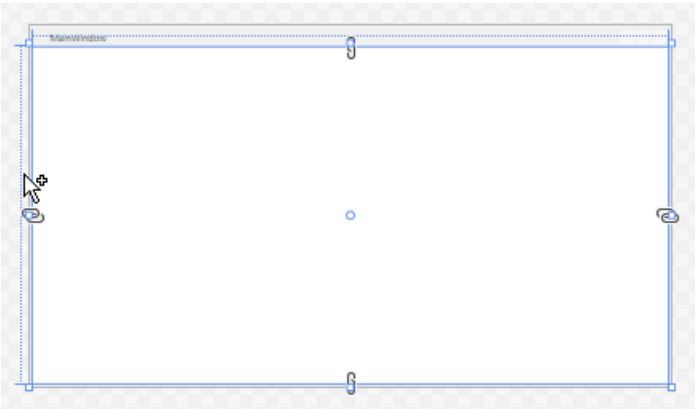
Die **Grid**-Eigenschaft **RowDefinitions** ist vom Typ **System.Windows.Controls.RowDefinitionCollection**, und die XAML-Kollektionssyntax (vgl. Abschnitt 12.3.2.3.4) erlaubt es in diesem Fall, auf ein Instanzelement zum Kollektionsobjekt zu verzichten.

Die XAML-Elemente zur Definition der Zeilenstruktur kann man auch über den WPF-Designer unserer Entwicklungsumgebung erstellen. Markieren Sie zunächst das **Grid**-Element mit einer von den folgenden Techniken:

- Mausklick auf das Innere des Fensters
- Mausklick auf das **Grid**-Element im XAML-Fenster
- **Dokumentengliederung** öffnen (z. B. mit **Ansicht > Weitere Fenster > Dokumentengliederung**) und auf das **Grid**-Objekt klicken

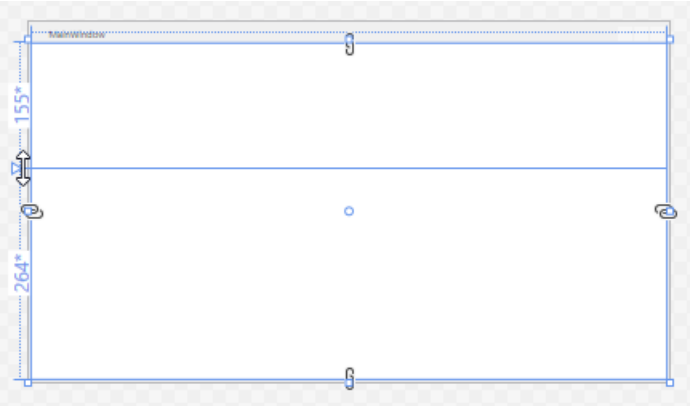


Setzen Sie dann per Mausklick auf die am linken Rand des **Grid**-Elements befindliche Zeilendefinitionszone eine Trennlinie:



Dabei wird im XAML-Code nötigenfalls ein Eigenschaftselement von Typ **Grid.RowDefinitions** erstellt und die Liste der **RowDefinition**-Elemente erweitert.

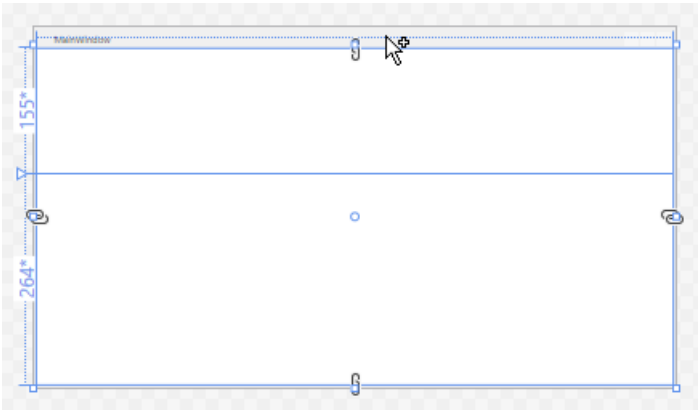
Die Trennlinie lässt sich anschließend per Maus verschieben, wobei die **Height**-Werte der beiden Zeilen festgelegt werden:



Das XAML-Ergebnis (inkl. **Height**-Definition):

```
<Grid>
   <Grid.RowDefinitions>
      <RowDefinition Height="155*"/>
      <RowDefinition Height="264*"/>
   </Grid.RowDefinitions>
</Grid>
```

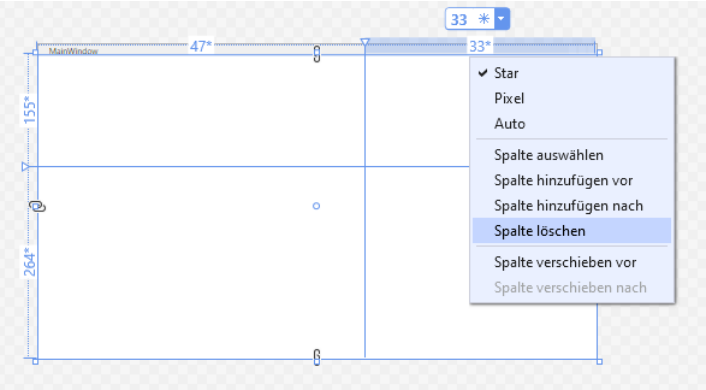
Analog lässt sich durch direktes Editieren des XAML-Codes oder durch Mausklicks auf die am oberen Rand des **Grid**-Elements befindliche Spaltendefinitionszone



die Spaltenstruktur vereinbaren, z. B. mit dem folgenden XAML-Ergebnis:

```
<Grid>
   <Grid.ColumnDefinitions>
      <ColumnDefinition Width="47*"/>
      <ColumnDefinition Width="33*"/>
   </Grid.ColumnDefinitions>
   <Grid.RowDefinitions>
      <RowDefinition Height="155*"/>
      <RowDefinition Height="264*"/>
   </Grid.RowDefinitions>
</Grid>
```

Wenn man den Mauszeiger links neben eine Zeile oder über eine Spalte positioniert, dann erscheint ein Werkzeug zur Modifikation von Größe und Position, z. B.:



Über das Symbol neben der numerischen Größenangabe wählt man das Verfahren zur Größenbestimmung:

- Angabe einer festen Pixelzahl, z. B.: 330
- Angabe eines Anteils an der horizontalen Ausdehnung des **Grid**-Containers, z. B.: 33 *
- Automatische Größenberechnung aufgrund des Inhalts, z. B.: auto

Die numerische, als Pixelzahl oder Anteil zu interpretierende Größenangabe lässt sich nach einem Mausklick ändern.

Per Drop-Down - Menü kann man z. B. das Verfahren zur Größenbestimmung festlegen oder eine Spalte bzw. Zeile löschen.

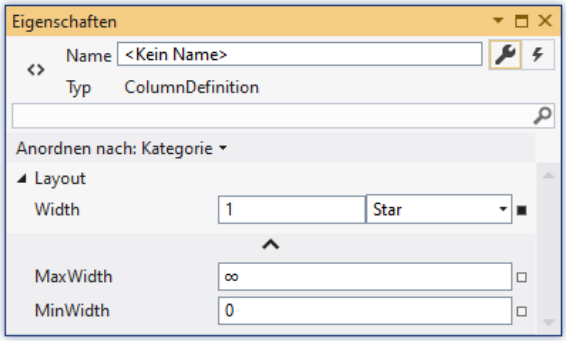
Anschließend wird beschrieben, wie man den Platz per XAML-Code auf Spalten bzw. Zeilen verteilt.

12.6.1.2 Platzaufteilung

Die Breite einer Spalte bzw. die Höhe einer Zeile lässt sich per **Width**- bzw. **Height**-Attribut festlegen, wobei folgende Alternativen zur Verfügung stehen:

- Bei einer fehlenden Größenangabe
  ```
  <ColumnDefinition/>
  ```
  oder bei der Anforderung
  ```
  <ColumnDefinition Width="*" />
  ```
  beansprucht die Spalte bzw. Zeile den gesamten noch nicht vergebenen Platz. Verhalten sich alle Konkurrenten so, wird der verfügbare Platz gleichmäßig aufgeteilt.
  Über Faktoren für die Sternangabe lässt sich ein mehrfacher Platzbedarf anmelden. So wird z. B. für die beiden folgenden Spalten der verfügbare Platz im Verhältnis 1:2 aufgeteilt:
  ```
  <ColumnDefinition Width="*" />
  <ColumnDefinition Width="2*" />
  ```
  Ein isolierter Stern (siehe erste Spaltendefinition) hat implizit den Faktor 1.
  Eine Spalte mit der Breite 0 ist unsichtbar:
  ```
  <ColumnDefinition x:Name="Column1" Width="0"/>
  ```
  Erhält die Spalte einen Referenzvariablennamen, kann ihre Breite per Programm mit Hilfe der Struktur **GridLength** und der Enumeration **GridUnitType** neu festgelegt werden, z. B.:
  ```
  Column1.Width = new GridLength(1, GridUnitType.Star);
  ```
- Besitzen alle Spalten bzw. Zeilen einen festen (nicht als Anteil zu verstehenden) Wert in Pixeln, werden die angeforderten Pixel ausgegeben, solange der Vorrat reicht, sodass eventuell eine hintere Spalte bzw. Zeile komplett verschwindet.
- Vergibt man den Wert **Auto**, dann orientiert sich die Breite einer Spalte bzw. die Höhe einer Zeile am maximalen Platzbedarf der enthaltenen Elemente.

Statt den **Width**- bzw. **Height**-Wert im XAML-Fenster einzutragen, kann man bei passender Markierung auch das **Eigenschaften**-Fenster benutzen, z. B.



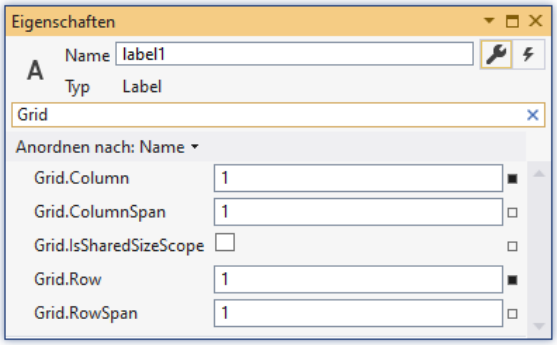
Zur Verwaltung des Platzbedarfs einer Spalte bzw. Zeile können zusätzlich die Eigenschaften **MinWidth**, **MaxWidth, MinHeight** und **MaxHeight** verwendet werden (vgl. Abschnitt 12.7.3.2).

12.6.1.3 Platzanweisung

Besitzt ein per **Grid**-Container verwaltetes Steuerelement keine Ortsangabe, landet es in der **Grid**-Zelle (0, 0), also oben links. Um für ein Steuerelement eine alternative Zelle festzulegen, sind im zugehörigen XAML-Element die Attribute **Grid.Row** bzw. **Grid.Column** mit der null-basierten Nummer der Zeile bzw. Spalte zu versorgen. Hier wird ein **Label**-Objekt in die Zelle (1, 1) gesetzt:

```
<Grid>
  . . .
<Label Name ="label1" Content="Label"
       HorizontalAlignment="Center" VerticalAlignment="Center"
       Grid.Column="1" Grid.Row="1" Width="200" />
</Grid>
```

Natürlich kann man auch das Eigenschaftenfenster zur Platzanweisung benutzen:



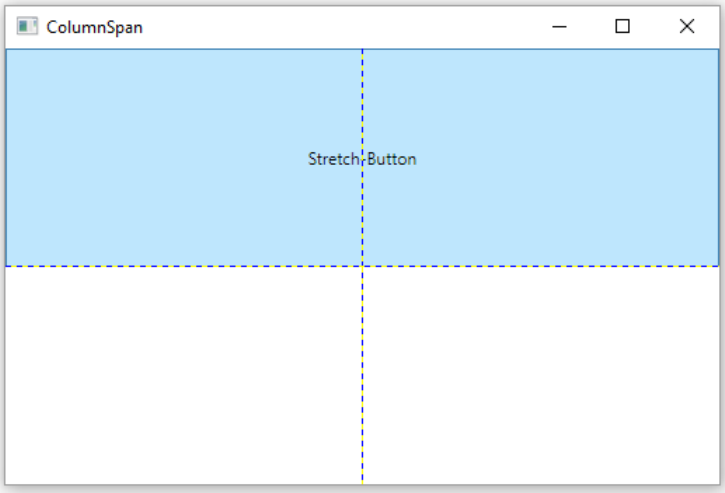
Bei **Grid.Row** bzw. **Grid.Column** handelt es sich um angefügte Eigenschaften (siehe Abschnitt 12.5.2).

12.6.1.4 Mehrzellige Elemente

Es ist nicht festgeschrieben, dass ein Steuerelement genau *eine* Zelle im **Grid**-Container belegen darf. Mit Hilfe der angefügten Eigenschaften **Grid.RowSpan** sowie **Grid.ColumnSpan** kann ein Steuerelement mehrere Zeilen und/oder Spalten beanspruchen. Mit dem folgenden XAML-Code wird ein (2 × 2) - **Grid** - Container definiert, wobei sich ein **Button**-Objekt über die *beiden* Zellen in der oberen Zeile erstreckt:

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ColumnSpan" Height="350" Width="525">
    <Grid ShowGridLines="True">
        <Grid.ColumnDefinitions>
            <ColumnDefinition /> <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition /> <RowDefinition />
        </Grid.RowDefinitions>
        <Button Content="Stretch-Button" Grid.ColumnSpan="2" />
    </Grid>
</Window>
```

Über den Wert **true** für die **Grid**-Eigenschaft **ShowGridLines** wird dafür gesorgt, dass die (2 × 2) -Struktur des Containers optisch präsent ist:[1]



Dass sich mehrere Steuerelemente eine **Grid**-Zelle teilen können, haben Sie schon wiederholt beobachtet (z. B. im Abschnitt 12.4.4). Bei einer vom Visual Studio neu angelegten WPF-Anwendung hat das Hauptfenster zunächst einen einzelligen **Grid**-Container, und alle aus der Toolbox per Drag & Drop übernommenen Steuerelemente landen in der einzigen Zelle mit den Koordinaten (0, 0). Sofern die Größen und Verankerungen der Elemente geschickt gewählt werden, resultiert ein sinnvoll nutzbares Fenster. Ein komplexes Layout in einem Fenster mit variabler Größe ergonomisch zu gestalten, ist aber mit (verschachtelten) Layoutcontainern einfacher.

12.6.2 DockPanel

Die Layoutcontainer-Klasse **DockPanel** bietet den verwalteten Steuerelementen über die angefügte Eigenschaft **Dock** mit den Werten **Left**, **Top**, **Right**, **Bottom** die Möglichkeit, sich an einer Seite festzusetzen und diese komplett zu belegen. Wollen mehrere Elemente eine Seite besetzen, werden sie dort in der Beitrittsreihenfolge gestapelt. Die Steuerelemente erhalten nach Möglichkeit ihre gewünschte Größe, und das zuletzt eingefügte Element erhält den kompletten noch freien Platz.[2]

Bei der folgenden Fensterklassendeklaration

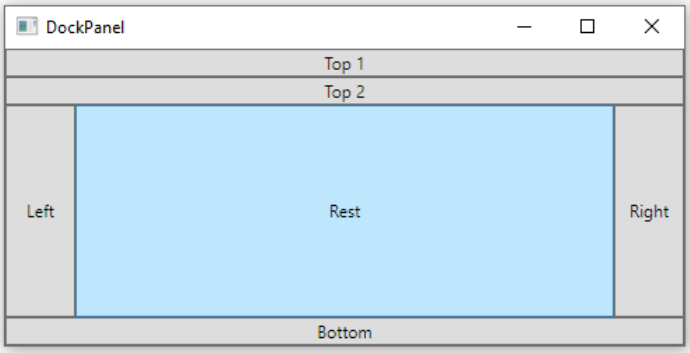
---

[1] Bei den Werten **true** und **false** für Eigenschaften vom Typ **bool** ist in XAML die Groß-/Kleinschreibung ausnahmsweise irrelevant.

[2] Wer sich an das **BorderLayout** im traditionsreichen GUI-Framework **Swing** der Programmiersprache Java erinnert fühlt, liegt genau richtig.

```
<Window x:Class="DockPanel.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ColumnSpan" Height="250" Width="500">
    <DockPanel>
        <Button DockPanel.Dock="Top">Top 1</Button>
        <Button DockPanel.Dock="Top">Top 2</Button>
        <Button DockPanel.Dock="Bottom">Bottom</Button>
        <Button DockPanel.Dock="Left" Width="50">Left</Button>
        <Button DockPanel.Dock="Right" Width="50">Right</Button>
        <Button>Rest</Button>
    </DockPanel>
</Window>
```

sind alle Seiten durch **Button**-Objekte mit **Dock**-Angabe besetzt, und ein fünftes **Button**-Objekt *ohne* **Dock**-Angabe belegt den frei gebliebenen Raum im Zentrum:



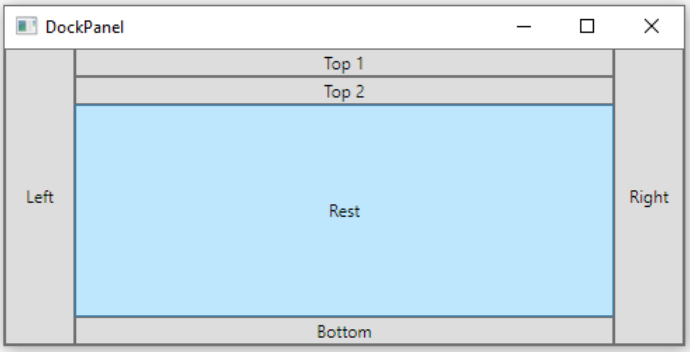
Die **DockPanel**-Struktur bietet ein für viele Programme geeignetes UI-Gerüst, z. B. für einen Editor:

- Oben werden das Menü und eine Symbolleiste untergebracht.
- Links befindet sich eine Navigationszone.
- In der Mitte wird das Dokument angezeigt und editiert.
- Rechts befindet sich eine Werkzeugsammlung.
- Unten befindet sich eine Statuszeile.

Ob die vier Ecken von den horizontalen oder von den vertikalen Steuerelementen eingenommen werden, hängt von der Aufnahmereihenfolge ab. Aus der Layoutdefinition

```
<DockPanel>
   <Button DockPanel.Dock="Left" Width="50">Left</Button>
   <Button DockPanel.Dock="Right" Width="50">Right</Button>
   <Button DockPanel.Dock="Top">Top 1</Button>
   <Button DockPanel.Dock="Top">Top 2</Button>
   <Button DockPanel.Dock="Bottom">Bottom</Button>
   <Button>Rest</Button>
</DockPanel>
```

folgt diese Anordnung:



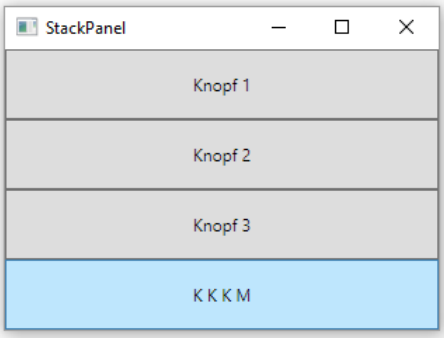
12.6.3 StackPanel

Mit einem Layoutcontainer der Klasse **StackPanel** realisiert man einen simplen vertikalen oder horizontalen Stapel von Steuerelementen. Die Elemente erhalten die gewünschten Ausdehnungen, solange der Vorrat an Pixeln reicht. Ist beim vertikalen Stapel die Gesamthöhe bzw. beim horizontalen Stapel die Gesamtbreite unzureichend, können die zuletzt eingefügten Elemente nicht mehr wunschgemäß versorgt werden.

Durch die folgende XAML-Deklaration

```
<Window x:Class="StackPanel.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        SizeToContent="Height" Width="325" Title="StackPanel">
    <StackPanel>
        <Button Height="50">Knopf 1</Button>
        <Button Height="50">Knopf 2</Button>
        <Button Height="50">Knopf 3</Button>
        <Button Height="50">K K K M</Button>
    </StackPanel>
</Window>
```

werden vier **Button**-Objekte übereinander gestapelt:



Weil die **Width**-Eigenschaften der Schalter keinen Wert erhalten, gilt die Voreinstellung **Stretch** (vgl. Abschnitt 12.7.3.2).

Im **Window**-Element sorgt die Eigenschaftszuweisung

```
SizeToContent="Height"
```

dafür, dass die Fensterhöhe an den vom **StackPanel** benötigten Platz angepasst wird.

Um einen *horizontalen* Stapel zu erhalten, setzt man die **StackPanel**-Eigenschaft **Orientation** auf den Wert **Horizontal**:

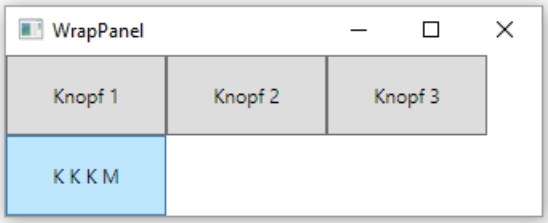
```
<StackPanel Orientation="Horizontal">
  . . .
</StackPanel>
```

12.6.4 WrapPanel

Beim **WrapPanel** kommt im Vergleich zum **StackPanel** ein automatischer Spalten- bzw. Zeilenumbruch hinzu. Durch die folgende XAML-Deklaration

```
<Window x:Class="WrapPanel.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="WrapPanel" SizeToContent="Height" Width="350">
    <WrapPanel Orientation="Horizontal">
        <Button Width="100" Height="50">Knopf 1</Button>
        <Button Width="100" Height="50">Knopf 2</Button>
        <Button Width="100" Height="50">Knopf 3</Button>
        <Button Width="100" Height="50">K K K M</Button>
    </WrapPanel>
</Window>
```

werden vier **Button**-Objekte nebeneinander gestapelt, bis der Platz erschöpft und daher ein Zeilenumbruch erforderlich ist:



12.6.5 UniformGrid

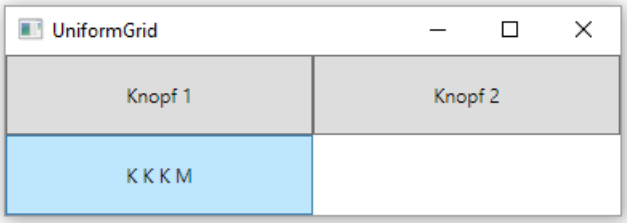
Im Vergleich zum **Grid**-Layoutcontainer bestehen beim **UniformGrid**-Container folgende Unterschiede:

- Die Spalten- und Zeilenzahl sind gleich.
- Diese gemeinsame Zahl wird automatisch ermittelt.
  Beim Einfügen von Elementen findet im Unterschied zum **Grid**-Container keine (implizite) Platzanweisung statt. Ist die Zahl der Elemente z. B. größer als 4 und kleiner als 10, erhält man einen (3 × 3) - Layoutcontainer.

Aus der folgenden XAML-Deklaration

```
<Window x:Class="UniformGrid.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="UniformGrid" SizeToContent="Height" Width="400">
    <UniformGrid>
        <Button Height="50">Knopf 1</Button>
        <Button Height="50">Knopf 2</Button>
        <Button Height="50">K K K M</Button>
    </UniformGrid>
</Window>
```

resultiert ein Container mit (2 × 2) gleich großen Zellen. Die drei Elemente werden nacheinander auf die Zellen verteilt, wobei der Spaltenindex schneller läuft:



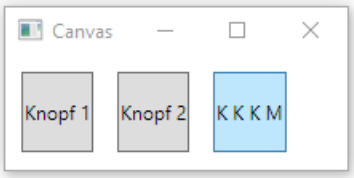
12.6.6 Canvas

Der **Canvas**-Container stellt eine Zeichenfläche ohne jede Layout-Logik bei Größenänderungen zur Verfügung und sollte nur in speziellen Situationen eingesetzt werden. Über die angefügten Eigenschaften **Canvas.Left** und **Canvas.Top** legt man für die verwalteten Elemente den Abstand zum linken bzw. zum oberen Rand fest. Wer unbedingt will, kann mit diesem Container nach schlechter alter Sitte Steuerelemente mit fester Größe und Position in liebevoller Kleinarbeit montieren.

Aus der folgenden XAML-Deklaration

```
<Window x:Class="Canvas.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Canvas" Height="110" Width="230">
    <Canvas>
        <Button Height="50" Canvas.Left="10" Canvas.Top="10">Knopf 1</Button>
        <Button Height="50" Canvas.Left="70" Canvas.Top="10">Knopf 2</Button>
        <Button Height="50" Canvas.Left="130" Canvas.Top="10">K K K M</Button>
    </Canvas>
</Window>
```

resultiert das Fenster:



12.6.7 Geschachtelte Layoutcontainer

Sollen z. B. in die Zelle (0, 0) eines **Grid**-Containers drei **Button**-Objekte (Befehlsschalter) übereinander positioniert werden, fügt man in diese Zelle zunächst einen Layoutcontainer der Klasse **StackPanel** ein, z. B.:

```
<Window x:Class="Geschachtelte_Layoutcontainer.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Geschachtelte Layoutcontainer" SizeToContent="Height" Width="400">
  <Grid ShowGridLines="True">
    <Grid.ColumnDefinitions>
        <ColumnDefinition /><ColumnDefinition />
    </Grid.ColumnDefinitions>
    <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center" Margin="10">
    </StackPanel>
  </Grid>
</Window>
```

Weil im **StackPanel**-Element die angefügten Eigenschaften **Grid.Row** und **Grid.Column** fehlen, wird jeweils der Wert 0 angenommen.

Das **StackPanel**-Objekt ist per XAML instruiert,

- sich in seiner Zelle horizontal und vertikal zur Mitte hin zu orientieren,
- seine Breite und Höhe automatisch an die enthaltenen Steuerelemente anzupassen,
- zur Umgebung einen allseitigen Abstand von 10 einzuhalten (Einheit: geräteunabhängige Pixel, siehe Abschnitt 12.7.3.2).

Es werden drei **Button**-Objekte in den **StackPanel**-Container eingefügt:

```
<StackPanel HorizontalAlignment="Center" VerticalAlignment="Center"
            Height="Auto" Width="Auto" Margin="10">
    <Button Content="Button 1" Height="25" Width="75" Margin="5" />
    <Button Content="Button 2" Height="25" Width="75" Margin="5" />
    <Button Content="Button 3" Height="25" Width="75" Margin="5" />
</StackPanel>
```

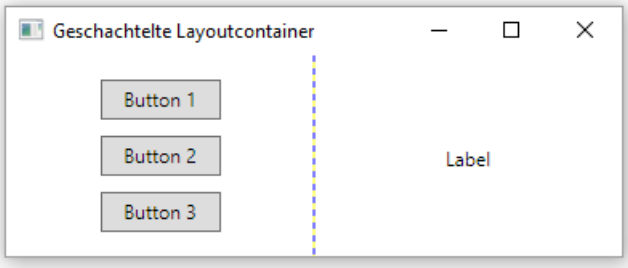
Die **Button**-Objekte sind ...

- 75 Pixel breit und 25 Pixel hoch,
- auf allen Seiten von einem freien Streifen mit der Breite 5 umgeben.

Außerdem wird ein **Label**-Objekt in die Zelle (0, 1) des **Grid**-Containers eingefügt:

```
<Label Content="Label" Grid.Column="1" Height="23"
       HorizontalAlignment="Center" VerticalAlignment="Center" />
```

Das Ergebnis:



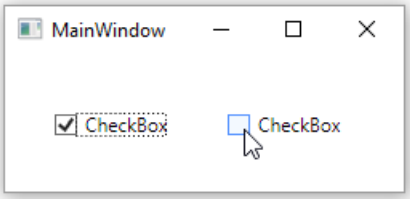
12.7 Basiswissen über Steuerelemente

Die WPF-Bibliothek bietet zahlreiche Klassen zur Realisation von Steuerelementen (Schaltflächen, Textfeldern, Kontrollkästchen, Listen etc.) an, sodass sich für sehr viele Programmierprojekte auf bequeme Weise ergonomische und attraktive Bedienoberflächen erstellen lassen.

12.7.1 Steuerelemente im Vergleich zu anderen Objekten

Als Besonderheiten der Steuerelementklassen (z. B. im Vergleich zu Klassen wie **String** oder **List<T>**) sind zu nennen:

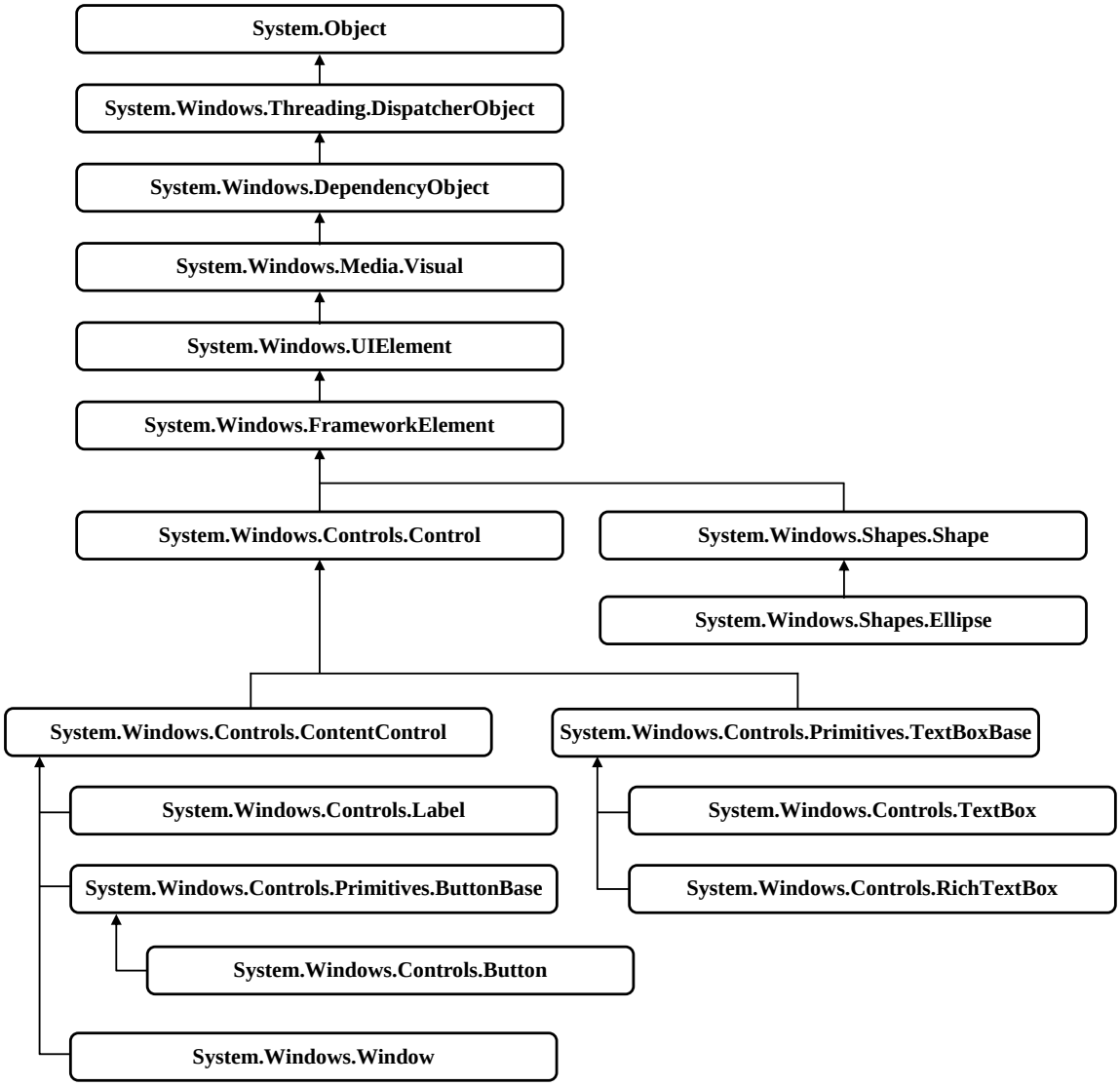
- Ihre Objekte können **auf dem Bildschirm auftreten** und dabei selbständig **mit dem Benutzer interagieren**. Wenn wir z. B. ein Kontrollkästchen (ein Objekt der Klasse **CheckBox**) in ein Fenster einbauen, dann erscheint bzw. verschwindet bei einem Mausklick die Markierung,



  ohne dass wir uns um diese Anpassung der Optik kümmern müssen.
- Die Steuerelemente kommunizieren über **Ereignisse** (im Sinn von Abschnitt 10.2 bzw. Abschnitt 12.4) mit anderen Klassen. Will man z. B. über die gesetzte Markierung eines Kontrollkästchen informiert werden, registriert man eine Behandlungsmethode, die den Delegatentyp **System.Windows.RoutedEventHandler** erfüllt, bei seinem Ereignis **Checked**.
- Ihre Eigenschaften (z. B. **Text**, **Height**, **HorizontalAlignment**, **Focusable**) können zur Entwurfszeit über **Werkzeuge der Entwicklungsumgebung** konfiguriert werden (siehe z. B. Abschnitt 5.12.5).

12.7.2 Abstammungsverhältnisse

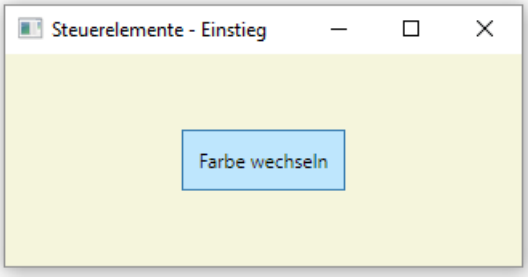
In der Windows Presentation Foundation stammen die *Steuerelemente* (**Button**, **TextBox** etc.) von der Basisklasse **System.Windows.Controls.Control** ab, während die zur optischen Gestaltung eingesetzten und nicht interaktionsfähigen *Grafikelemente* (**Image**, **Ellipse** etc.) von der Basisklasse **System.Windows.Shapes.Shape** abstammen:



Die von **System.Windows.Controls.ContentControl** abstammenden Klassen beherrschen das WPF-Inhaltsmodell, können also untergeordnete Elemente aufnehmen.

12.7.3 Verwendung

Wir beschäftigen uns zunächst anhand eines sehr einfachen Beispielprogramms



damit, wie man ...

- ein Steuerelement als Member-Objekt in eine Fensterklasse aufnimmt,
- seine Position auf dem Fenster festlegt,
- seine Eigenschaften zur Entwurfszeit bestimmt
- und Behandlungsmethoden bei seinen Ereignissen registriert.

12.7.3.1 Instanziieren

Wird mit dem WPF-Designer im Visual Studio ein Steuerelement aus der **Toolbox** auf ein Fenster gezogen, landet es in einem Layoutcontainer. Bei einer neuen WPF-Anwendung ist für das Hauptfenster per Voreinstellung ein **Grid**-Container zuständig, der sich vorläufig auf die Zelle (0, 0) beschränkt (also nur eine Zeile bzw. Spalte besitzt). Im XAML-Code zur Fensterklasse erscheint im **Grid**-Element ein Eintrag für das neue Steuerelement, z. B.:

```
<Grid>
    <Button Content="Button" HorizontalAlignment="Left"
     Margin="345,169,0,0" VerticalAlignment="Top"/>
</Grid>
```

Dem XAML-Code ist zu entnehmen, dass ...

- ein Objekt der Klasse **Button** entsteht,
- der Schalter aktuell die Beschriftung `Button` besitzt,
- der Schalter am linken Rand der **Grid**-Zelle (0, 0), also letztlich am linken Rand des Fensters, verankert ist mit dem Abstand 345,
- der Schalter am oberen Rand des Fensters verankert ist mit dem Abstand 169.

Wir haben im Abschnitt 12.3.4 erfahren, dass ...

- bei der Programmerstellung aus der XAML-Datei eine Binär-Variante mit der Namenserweiterung **.baml** entsteht,
- beim Programmstart die **baml**-Datei analysiert und im Beispiel das dort beschriebene **Button**-Objekt erzeugt wird.

12.7.3.2 Elementare Eigenschaften

Durch die folgenden Eigenschaften eines Steuerelements werden seine Ausdehnung sowie seine Position im umgebenden Container bzw. in der umgebenden Containerzelle geregelt. Sie sind schon in der Klasse **System.Windows.FrameworkElement** definiert, also auch bei Layoutcontainern und Grafikelementen anwendbar:

- **Width**, **Height**
  Mit den Eigenschaften **Width** und **Height** vom Typ **double** wählt man für die Breite bzw. Höhe eine Anzahl von geräteunabhängigen Pixeln (1/96 Zoll pro Einheit). Durch den voreingestellten Wert **NaN** werden die Breite bzw. Höhe passend zum Inhalt und zum angeforderten Innenrand (siehe unten) festgelegt.
- **MinWidth**, **MaxWidth, MinHeight, MaxHeight**
  Mit den Eigenschaften **MinWidth**, **MaxWidth, MinHeight** und **MaxHeight** vom Typ **double** wählt man für die Breite bzw. Höhe einen minimalen bzw. maximalen Wert. Im folgenden Beispiel resultiert ein Fenster mit der fixierten Höhe 200 und der initialen Breite 400, die vom Benutzer zwischen 300 und 500 verändert werden kann:

  ```
  <Window . . .
   Width="400" MinWidth="300" MaxWidth="500"
   Height="200" MinHeight="200" MaxHeight="200" >
   . . .
  </Window>
  ```

- **Margin**
  Mit dieser Eigenschaft (Typ.: **System.Windows.Thickness**) legt man fest, wie viel Platz um das Element herum (bis zum Rand der umgebenden Containerzelle oder bis zum Nachbarn) frei bleiben soll. Man gibt für einen Abstand per **double**-Wert eine Anzahl von geräteunabhängigen Pixeln an (1/96 Zoll pro Einheit), wobei unterschiedlich detaillierte Angaben möglich sind:
  - Mit *einer* Zahl legt man für alle Seiten denselben Rand fest.
  - Von *zwei* Zahlen legt die erste Zahl die beiden horizontalen und die zweite Zahl die beiden vertikalen Ränder fest.
  - *Vier* Zahlen beziehen sich (in dieser Reihenfolge) auf den linken, oberen, rechten und den unteren Rand.
- **HorizontalAlignment**
  Diese Eigenschaft bestimmt die horizontale Ausrichtung. Es sind vier Werte (aus der Enumeration **HorizontalAlignment** im Namensraum **System.Windows**) möglich:
  - **Left**
    Das Element ist am linken Rand verankert.
  - **Right**
    Das Element ist am rechten Rand verankert.
  - **Stretch**
    Das Element ist am linken *und* am rechten Rand verankert, sodass sich seine horizontale Ausdehnung zusammen mit der umgebenden Containerzelle ändert. Besitzt die **Width**-Eigenschaft einen expliziten Wert, wird der **HorizontalAlignment**-Wert **Stretch** ignoriert.
  - **Center**
    Das Element wird horizontal zentriert.
- **VerticalAlignment**
  Diese Eigenschaft bestimmt die vertikale Ausrichtung. Es sind vier Werte (aus der Enumeration **VerticalAlignment** im Namensraum **System.Windows**) möglich:
  - **Top**
    Das Element ist am oberen Rand verankert.
  - **Bottom**
    Das Element ist am unteren Rand verankert.
  - **Stretch**
    Das Element ist am oberen *und* am unteren Rand verankert, sodass sich seine vertikale Ausdehnung zusammen mit der umgebenden Containerzelle ändert. Besitzt die **Height**-Eigenschaft einen expliziten Wert, wird der **VerticalAlignment**-Wert **Stretch** ignoriert.
  - **Center**
    Das Element wird vertikal zentriert.

Welche Werte für die Eigenschaften **HorizontalAlignment** und **VerticalAlignment** realisierbar sind, hängt vom Typ des Layoutcontainers ab, z. B. ...

- haben die Elemente in einem **Grid**- oder **UniformGrid**-Container per Voreinstellung die horizontale Ausrichtung **Stretch** und können alternative horizontale Ausrichtungen erhalten,
- sind die Elemente in einem horizontal orientierten **StackPanel**- oder **WrapPanel**-Container grundsätzlich von links nach rechts angeordnet, d.h. das **HorizontalAlignment** - Attribut der Elemente ist wirkungslos.

Bei der vertikalen Ausrichtung haben die Elemente der genannten Layoutcontainer die freie Wahl.

Es folgen einige Eigenschaften, die den *Inhalt* von Steuerelemente im engeren Sinn, also von Objekten der Klasse **System.Windows.Controls.Control**, betreffen:

- **Content**
  Bei vielen Steuerelementen wird über die in **System.Windows.Controls.ContentControl** definierte Eigenschaft **Content** vom Typ **Object** der Inhalt festgelegt, z. B. bei einem **Button**-Objekt mit Beschriftung:
  ```
  <Button Content="Farbe wechseln" ... />
  ```
- **Padding**
  Mit dieser in **System.Windows.Controls.Control** definierten Eigenschaft vom Typ **System.Windows.Thickness** legt man eine freizuhaltende Randzone innerhalb der Steuerelementfläche (einen Innenrand) fest. Die Angaben sind analog zur Eigenschaft **Margin** zu machen (siehe oben).
- **HorizontalContentAlignment**, **VerticalContentAlignment**
  Mit diesen Eigenschaften bestimmt man die Ausrichtung des Inhalts innerhalb der rechteckigen Steuerelementfläche. Es sind dieselben Werte möglich wie bei **HorizontalAlignment** bzw. **VerticalAlignment** (siehe Abschnitt 12.7.3.2).

Über die WPF-Eigenschaftsübertragung an eingeschachtelte Elemente wurde schon im Abschnitt 12.5.1 berichtet.

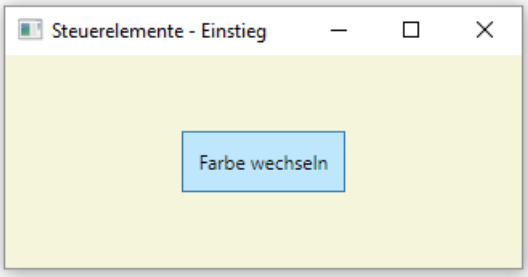
12.7.3.3 Ereignisbehandlung

Zur Behandlung der von einem Steuerelement angebotenen Ereignisse kommen zwei Verfahren in Frage:

- Wie schon mehrfach in Beispielen vorgeführt, kann man zu einem Ereignis eine Behandlungsmethode mit passender Delegatensignatur erstellen und beim Ereignis registrieren (siehe Abschnitt 12.7.3.3.1).
- Wird zu einer von **UIElement** abstammenden Klasse eine Ableitung definiert, kann man dort die ein Ereignis auslösende Methode überschreiben (siehe Abschnitt 12.7.3.3.2).

12.7.3.3.1 Behandlungsmethoden registrieren

Nach Bedarf werden in der Code-Behind - Datei zur Fensterklasse Behandlungsmethoden definiert und bei Ereignissen des Fensters oder der Steuerelemente registriert. Im aktuellen Beispiel



soll das **Click**-Ereignis des **Button**-Objekts behandelt werden. Dazu fordern wir im WPF-Designer bei markiertem **Button**-Objekt im **Eigenschaften**-Fenster per Mausklick auf das Symbol ⚡ die Liste mit den Ereignissen an und setzen einen Doppelklick auf den Eintrag **Click**. Weil das **Click**-Ereignis bei einem Befehlsschalter das Standardereignis ist, führt ein Doppelklick auf das **Button**-Objekt im WPF-Designer zum selben Ziel: Die Datei **MainWindow.xaml.cs** erscheint im Editor mit einer vorbereiteten Ereignisbehandlungsmethode,

```
private void button_Click(object sender, RoutedEventArgs e) {
}
```

die wir noch vervollständigen müssen, z. B.:

```
private void button_Click(object sender, RoutedEventArgs e) {
   Background = (Background == Brushes.Beige) ? Brushes.LightGray : Brushes.Beige;
}
```

Im Beispiel soll bei jedem Mausklick auf den Schalter die Hintergrundgestaltung des Fensters zwischen **Brushes.Beige** (Pinsel mit Volltonfarbe Beige) und **Brushes.LightGray** (Pinsel mit Volltonfarbe **LightGray**) wechseln.

Die Registrierung einer Ereignisbehandlungsmethode kann im XAML-Code des Fensters vorgenommen werden, was meist von der Entwicklungsumgebung erledigt wird, z. B.:

```
<Button Content="Farbe wechseln"
        HorizontalAlignment="Center" VerticalAlignment="Center"
        Padding="10" Click="button_Click" />
```

Wie Sie aus dem Abschnitt 12.3.4 wissen, ist die automatisch erstellte partielle Fensterklassendefinition in der Datei **MainWindow.g.cs** für das Registrieren der **Click**-Behandlungsmethode per C# - Ereignissyntax zuständig. Wir finden die erwartete Anweisung in der Methode **Connect**:

```
void System.Windows.Markup.IComponentConnector.Connect(int connectionId, object target) {
    switch (connectionId)
    {
    case 1:
    this.button = ((System.Windows.Controls.Button)(target));

    #line 7 "..\..\MainWindow.xaml"
    this.button.Click += new System.Windows.RoutedEventHandler(this.button_Click);

    #line default
    #line hidden
    return;
    }
    this._contentLoaded = true;
}
```

Wie der XAML-Code zum Beispiel zeigt, war es nicht erforderlich, für das **Button**-Objekt einen Namen festzulegen.

12.7.3.3.2 On-Methoden überschreiben

Zur bisher beschriebenen Technik der Ereignisbehandlung gibt es eine Alternative, die nun vorgestellt werden soll. Um UI-Ereignisse (z. B. **MouseEnter**, **MouseDown**) auszulösen, rufen von **UIElement** abstammende WPF-Klassen (z. B. **Window**, **Button**) unter Mitwirkung des Laufzeitsystems jeweils eine Methode auf, deren Name aus dem Präfix **On** und dem Ereignisnamen besteht, z. B.

- **OnMouseEnter()**
- **OnMouseDown()**

Weil die **On**-Methoden als **protected** sowie **virtual** definiert sind, kann man sie in abgeleiteten Klassen überschreiben, um die gewünschte Reaktion auf ein Ereignis direkt in der **On**-Methode vorzunehmen. Das Überschreiben der zugehörigen **On**-Methode ist bei den Ereignissen von Steuerelementen (z. B. **Button**) nicht unbedingt die bevorzugte Technik, weil dazu eine eigene Klassendefinition erforderlich ist. Bei den Fensterereignissen ist die Technik hingegen leicht nutzbar, weil wir die Klasse **Window** regelmäßig beerben.

In der Fensterklasse mit dem folgenden XAML-Code wird zunächst auf gewohnte Weise die Methode `Window_MouseDown()` beim **MouseDown**-Ereignis des Fensters registriert:

```
<Window x:Class="OnMouseDownWindow.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="OnMouseDown Window" Height="200" Width="350" MouseDown="Window_MouseDown">
    <Grid>
        <Label x:Name="label"/>
    </Grid>
</Window>
```

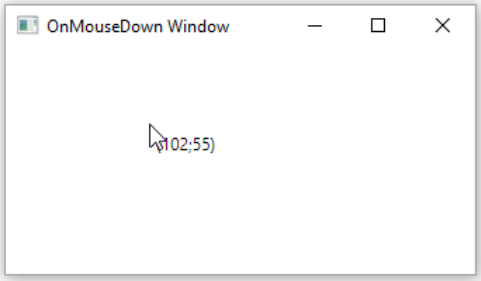
Die in der Code-Behind - Datei

```
using System;
 . . .
using System.Windows.Shapes;

namespace OnMouseDownWindow {
   public partial class MainWindow : Window {
      public MainWindow() {
         InitializeComponent();
      }

      private void Window_MouseDown(object sender, MouseButtonEventArgs e) {
         label.Margin = new Thickness(e.GetPosition(this).X, e.GetPosition(this).Y,0,0);
         label.Content = "(" + e.GetPosition(this).ToString() + ")";
      }
   }
}
```

implementierte Methode `Window_MouseDown()` ...

- legt die Position eines Labels über dessen **Margin**-Eigenschaft (siehe Abschnitt 12.7.3.2) neu auf die Klickstelle fest, die über die Methode **GetPosition()** des **MouseButtonEvent-Args**-Parameterobjekts bekannt wird,
- und verarbeitet die Koordinaten der Klickstelle zum neuen Wert der **Content**-Eigenschaft des Labels.

Somit kann das Programm die Koordinaten einer Klickstelle am Ort des Geschehens anzeigen, z. B.:



Als alternative Technik zur **MouseDown**-Behandlung kann in der abgeleiteten Klasse die **Window**-Methode **OnMouseDown()** überschrieben werden, wobei in der ersten Anweisung in der Regel die überschriebene Basisklassenmethode aufgerufen werden sollte:

```
protected override void OnMouseDown(MouseButtonEventArgs e) {
    base.OnMouseDown(e);
    label.Margin = new Thickness(e.GetPosition(this).X, e.GetPosition(this).Y, 0, 0);
    label.Content = "(" + e.GetPosition(this).ToString() + ")";
}
```

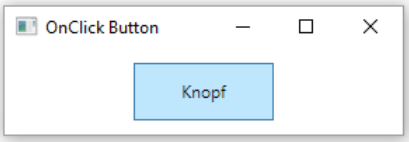
Den Aufruf der Basisklassenmethode zu unterlassen, hat bei vielen Ereignissen zur Folge, dass registrierte Behandlungsmethoden abgehängt werden, weil das Ereignis in der On-Methode ausgelöst wird. Dies trifft z. B. für die Methode **OnClick()** der Klasse **ButtonBase** zu:[1]

```
protected virtual void OnClick() {
   RoutedEventArgs newEvent = new RoutedEventArgs(ButtonBase.ClickEvent, this);
   RaiseEvent(newEvent);
   MS.Internal.Commands.CommandHelpers.ExecuteCommandSource(this);
}
```

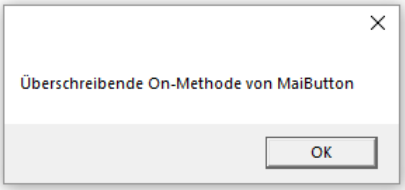
Bei einer Überschreibung ohne Basisklassenmethodenaufruf, werden beim Ereignis registrierte Methoden abgehängt, z. B.:

```
using System;
using System.Windows;
using System.Windows.Controls;

class MaiButton : Button {
   protected override void OnClick() {
      // base.OnClick(); // Erforderlicher Aufruf der überschriebenen Methode
      MessageBox.Show("Überschreibende On-Methode von MaiButton");
   }
}

class OnClickButton : Window {
   OnClickButton() {
      Height = 100; Width = 300;
      Title = "OnClick Button";
      MaiButton k1 = new MaiButton(); k1.Content = "Knopf"; k1.Width = 100;
      AddChild(k1);
      k1.Click += new RoutedEventHandler(knopf_Click);
   }

   private void knopf_Click(object sender, RoutedEventArgs e) {
      MessageBox.Show("Click-Handler von Knopf");
   }

   [STAThread]
   static void Main() {
      new Application().Run(new OnClickButton());
   }
}
```

Aufgrund des Programmierfehlers wird bei einem Mausklick auf den Schalter



nur noch die **OnClick()** - Überschreibung ausgeführt:

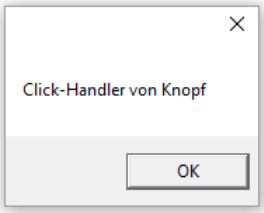


[1] Der Quellcode zur Klasse **ButtonBase** in der BCL zu .NET 5.0 kann über die folgende Webseite inspiziert werden: https://source.dot.net/

Mit der korrekten Überschreibung

```
protected override void OnClick() {
   base.OnClick();
   MessageBox.Show("Überschreibende On-Methode von MaiButton");
}
```

bleiben andere Ereignisbehandlungsmethoden im Spiel, z. B.:



Bei der **Window**-Methode **OnMouseDown()** hat die Unterlassung des Basismethodenaufrufs keine erkennbaren Konsequenzen. Halten Sie sich trotzdem grundsätzlich daran, zu Beginn einer überschreibenden Methode die Basisklassenvariante aufzurufen.

Wird eine Ereignisbehandlung durch das Überschreiben der zuständigen On-Methode erledigt, ist keine Registrierung (z. B. im XAML-Code) erforderlich.

12.7.4 Standardkomponenten

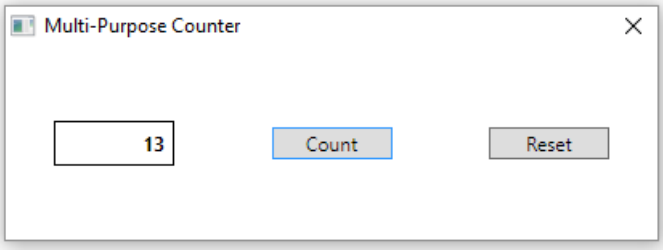
Dieser Abschnitt behandelt Steuerelemente, die sehr oft benötigt werden, und die teilweise auch schon in bisherigen Kursbeispielen aufgetreten sind: Befehlsschalter, Kontrollkästchen, Optionsfelder, Texteingabefelder, Listen und Kombinationsfelder.

12.7.4.1 Befehlsschalter

Befehlsschalter werden in der Windows Presentation Foundation durch die Klasse **Button** realisiert.

12.7.4.1.1 Beispiel

Im folgenden *Multi-Purpose Counter*, der z. B. zur Verkehrszählung taugt, kommen zwei **Button**-Objekte zum Einsatz:



Das GUI-Design und die Ereignisregistrierung werden durch den folgenden XAML-Code vorgenommen:

```
<Window x:Class="Button.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Multi-Purpose Counter" Height="149" Width="412" ResizeMode="NoResize">
    <Grid Button.Click="Button_Click">
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <Label Name="label" Content="0" Width="75"
           HorizontalAlignment="Center" VerticalAlignment="Center" BorderThickness="1"
           BorderBrush="Black" HorizontalContentAlignment="Right" FontWeight="Bold" />
        <Button Name="count" Content="Count" Width="75" Grid.Column="1"
           HorizontalAlignment="Center" VerticalAlignment="Center" IsDefault="True" />
        <Button Name="reset" Content="Reset" Width="75" Grid.Column="2"
           HorizontalAlignment="Center" VerticalAlignment="Center" />
    </Grid>
</Window>
```

Weil eine Änderung der Fenstergröße *nicht* erwünscht ist, erhält im Wurzelelement das **XAML**-Attribut **ResizeMode** (also letztlich die gleichnamige **Window**-Eigenschaft) den Wert **NoResize**. Dadurch verschwindet erwartungsgemäß auch der Titelzeilenschalter zum Maximieren des Fensters. Allerdings geht auch der Titelzeilenschalter zum Minimieren des Fensters verloren, sodass man das Programm nicht mehr in die Taskleiste beordern kann. Um die Fenstergröße zu fixieren, ohne den Minimieren-Schalter zu verlieren, kann man die **Window**-Eigenschaften bzw.-Attribute **MinWidth**, **MaxWidth, MinHeight** und **MaxHeight** verwenden (siehe Abschnitt 12.7.3.2).

Die **Height**-Eigenschaften der Steuerelemente werden nicht spezifiziert, und WPF orientiert sich infolgedessen an der Schriftgröße (mit ca. 5 Pixeln Innenrand).

Um für das **Label**-Steuerelement die Schriftauszeichnung **fett** zu wählen, erhält das **XAML**-Attribut **FontWeight** den Wert **Bold**:

```
FontWeight="Bold"
```

Außerdem erhält das Label einen Rahmen in schwarzer Farbe

```
BorderThickness="1" BorderBrush="Black"
```

und die (bei einer Zahlenanzeige übliche) rechtsbündige Textausrichtung:

```
HorizontalContentAlignment="Right"
```

Die für das **Click**-Ereignis *beider* **Button**-Steuerelemente zuständige Ereignisbehandlungsmethode `Button_Click()` wertet die **Source**-Eigenschaft des zweiten Parameters (vom Typ **RoutedEventArgs**) aus und orientiert ihr Verhalten an der Ereignisquelle:

```
public partial class MainWindow : Window {
  long anzahl;
   . . .
  private void Button_Click(object sender, RoutedEventArgs e) {
     if (e.Source == count)
        anzahl++;
     else
        anzahl = 0;
     label.Content = anzahl.ToString();
  }
}
```

Weil es um ein Routingereignis geht, kann die Behandlungsmethode dem gemeinsamen **Grid**-Container zugeordnet werden, statt sie bei *beiden* **Button**-Objekten registrieren zu müssen:

```
<Grid Button.Click="Button_Click">
  . . .
</Grid>
```

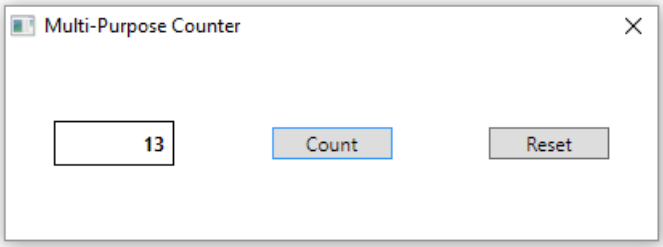
Wir verwenden hier ein sogenanntes *angefügtes Ereignis* (siehe Abschnitt 12.4.4). Damit wird der **Grid**-Layoutmanager zum Ereignisinhaber, und der beim Aufruf an `Button_Click()` übergebene **sender**-Parameter zeigt auf dieses Objekt. In `Button_Click()` wird daher die Ereignisquelle unter Verwendung des **RoutedEventArgs**-Objekts ermittelt.

12.7.4.1.2 Standard- und Escape-Schalter

Damit im aktuellen Beispielprogramm per **Enter**-Taste das **Click**-Ereignis des linken Schalters, der den Zählerstand inkrementiert, ausgelöst werden kann, wird die **IsDefault**-Eigenschaft dieses Schalters auf den Wert **true** gesetzt:

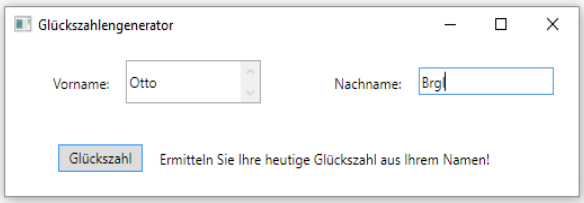
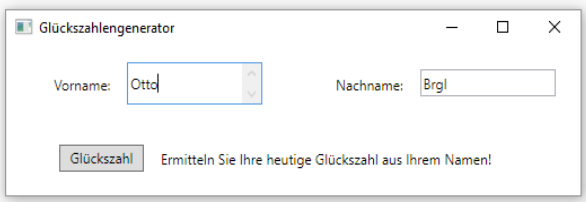
```
<Button Name="count" Content="Count" Width="75" Grid.Column="1"
   HorizontalAlignment="Center" VerticalAlignment="Center" IsDefault="True" />
```

Der Schalter wird damit zum **Standardschalter** des Fensters, und sein Privileg kommt im laufenden Programm durch eine blaue Umrandung zum Ausdruck:



Ein Multi-Counter - Benutzer kann nach dem Start sofort die **Enter**-Taste zum Zählen verwenden, ohne zuvor den Eingabefokus (per Maus oder Tabulatortaste) auf den Count-Schalter setzen zu müssen (siehe Abschnitt 12.7.4.4).

Die Standardschaltfläche verliert das Privileg, per **Enter**-Taste auslösbar zu sein, wenn ein anderes, an der **Enter**-Taste interessiertes Steuerelement (z. B. ein anderer Schalter oder ein mehrzeiliges Texteingabefeld) den Tastatur-Eingabefokus besitzt (siehe Abschnitt 12.7.4.4). Im folgenden Beispielprogramm (vgl. Abschnitt 12.7.4.3) ist der Schalter genau dann per **Enter**-Taste auslösbar, wenn das mehrzeilige (linke) Textfeld zur Eingabe der Vornamen den Eingabefokus *nicht* besitzt:

Schalter ist per **Enter**-Taste ansprechbar.	Schalter ist *nicht* per **Enter**-Taste ansprechbar.
	

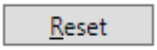
Analog zur Standardschaltflächen-Ernennung über die Eigenschaft **IsDefault** kann ein **Button**-Objekt über die Eigenschaft **IsCancel** als **Escape-Schalter** des Fensters festgelegt werden. Ihr **Click**-Ereignis kann dann per **Esc**-Taste ausgelöst werden.

12.7.4.1.3 Zugriffstaste

Soll das **Click**-Ereignis eines Befehlsschalters auch über einen **Alt-Tastenbefehl** (also über eine sogenannte **Zugriffstaste**) auslösbar sein, dann kommt ein **AccessText**-Element zum Einsatz, z. B.:

```
<Button Name="reset" Width="75" Grid.Column="2" HorizontalAlignment="Center"
        VerticalAlignment="Center" Click="ButtonOnClick">
    <AccessText>_Reset</AccessText>
</Button>
```

Der XAML-Compiler geht von einem Inhaltselement für den Schalter aus, und das Attribut **Content** im **Button**-Element muss entfallen, weil sonst die Eigenschaft **Content** mehrfach definiert wäre. Man wählt die auslösende Alt-Zugriffstaste, indem man dem zugehörigen Buchstaben im **AccessText**-Inhalt einen Unterstrich voranstellt. Spätestens nach einmaligem Drücken der Alt-Taste ist der Buchstabe im laufenden Programm durch Unterstreichung hervorgehoben, z. B.:

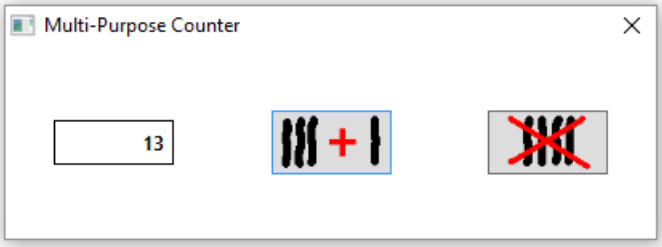


Weil bei **Button**-Objekten häufig eine Zugriffstaste benötigt wird, kann der XAML-Compiler das **AccessText**-Element automatisch einfügen, solange man einen Text als **Button**-Inhalt verwendet. Dazu setzt man im Wert des **Content**-Attributs einen Unterstich vor den Buchstaben, der in Kombination mit der Alt-Taste das **Click**-Ereignis des Schalters auslösen soll, z. B.:

```
<Button Name="reset" Content="_Reset" Width="75" Grid.Column="2"
    HorizontalAlignment="Center" VerticalAlignment="Center" Click="ButtonOnClick"/>
```

12.7.4.1.4 Bitmaps auf Schaltflächen

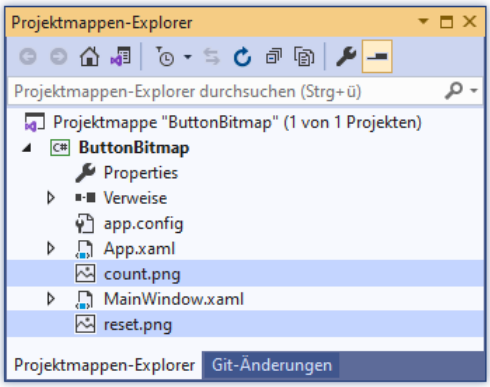
Um dem oben vorgestellten Mehrzweck-Zählprogramm ein individuelles Design zu geben, kann man die Beschriftungen der Schaltflächen durch Grafiken ersetzen (oder ergänzen), z. B.:



Für das Beispiel sind zunächst Bitmap-Dateien (Format **bmp** oder **png**) mit einer passenden Pixelmatrix zu erstellen (hier gewählt: 50 Zeilen und 100 Spalten), was z. B. mit der Windows-Zugabe *Paint* geschehen kann.

Gehen Sie im Visual Studio folgendermaßen vor, um die Bitmap-Dateien auf die Schaltflächen zu setzen:

- Kopieren Sie die Bitmap-Dateien in den Projektordner.
- Nehmen Sie die Bitmap-Dateien in das Projekt auf (Item **Hinzufügen > Vorhandenes Element** aus dem Kontextmenü zum Projekt). Anschließend werden die Dateien im **Projektmappen-Explorer** angezeigt, z. B.:

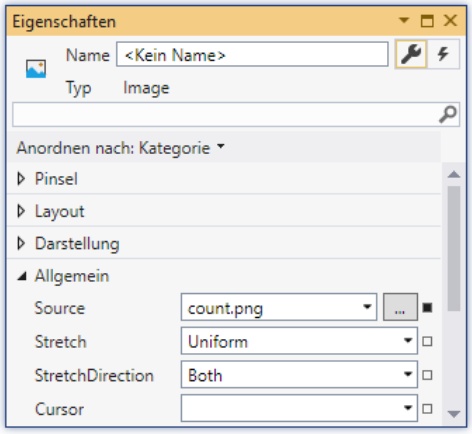


- Ersetzen Sie im XAML-Code bei den **Button**-Elementen das **Content**-Attribut mit der Beschriftung durch ein Inhaltselement vom Typ **Image**, z. B.:

```
<Button Name="count" Width="75" Height="40" Grid.Column="1" . . . IsDefault="True">
    <Image Source="count.png" />
</Button>
```

  Weil die Höhe des Schalters nicht der Bitmap-Datei überlassen werden sollte, ist ein **Height**-Attribut sinnvoll.

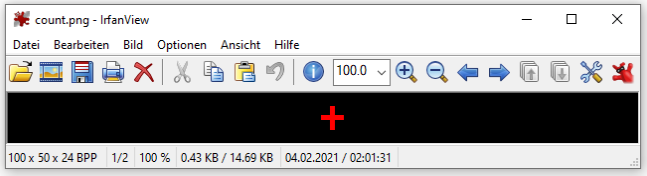
- Eine Bitmap-Datei als Quelle für die **Image**-Eigenschaft **Source** festzulegen, gelingt besonders bequem über das **Eigenschaften**-Fenster der Entwicklungsumgebung, z. B.:

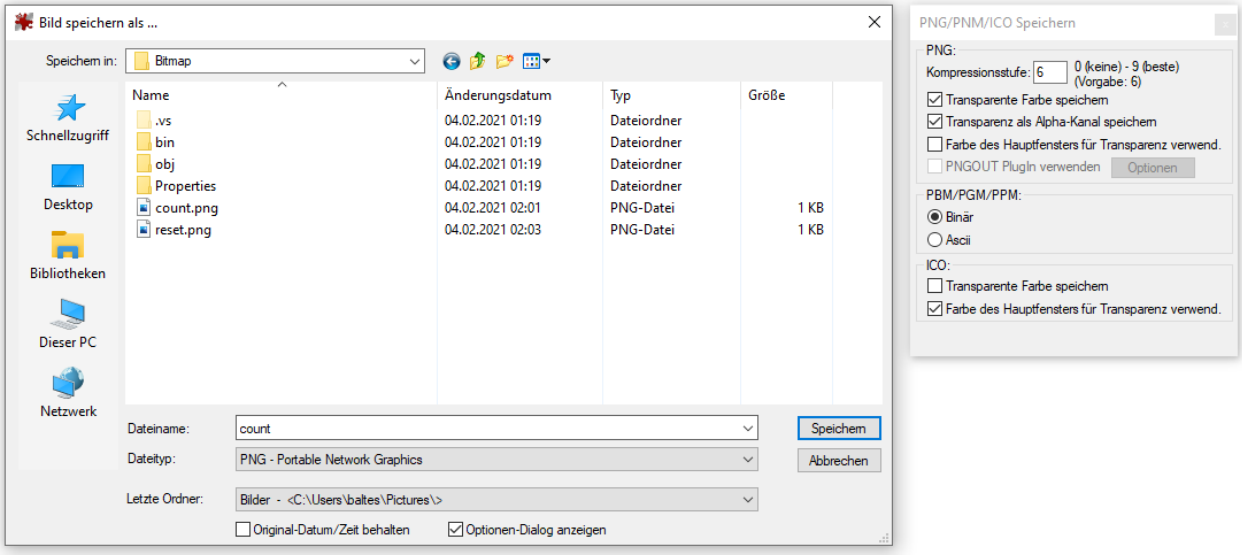


Macht man die Hintergrundfarbe der verwendeten Bitmaps transparent, dann harmonieren die bemalten Schalter unabhängig vom eingestellten Windows-Design optisch mit den restlichen Fensterbestandteilen. Dieses Ziel ist am einfachsten durch entsprechend präparierte Dateien zu realisieren. Eine PNG-Datei (*Portable Network Graphics*) mit transparenter Hintergrundfarbe lässt sich z. B. über die Freeware **IrfanView**[1] durch eine Option im **Speichern**-Dialog erstellen:

---

[1] Homepage: http://www.irfanview.de/
Nachdem IrfanView 4.56 eine BMP-Datei erfolgreich in eine PNG-Datei mit transparenter Hintergrundfarbe gewandelt hat, konnte das Programm die Ergebnisdatei anschließend nicht mehr erfolgreich öffnen:

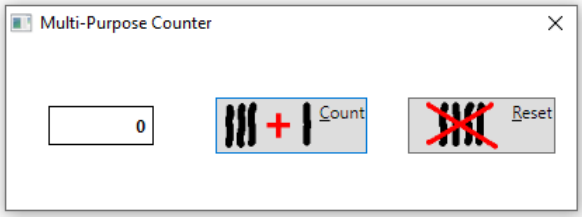




Soll ein **Button**-Steuerelement eine **Image**-Oberfläche *und* eine Zugriffstaste erhalten, setzt man unter Verwendung eines horizontal orientierten **StackPanel**-Layoutcontainers als Inhaltselement zum Bild noch ein **AccessText**-Element (siehe Abschnitt 12.7.4.1.3) auf die Schaltfläche, z. B.:

```
<Button Name="reset" Width="75" Height="40" Grid.Column="2"
        HorizontalAlignment="Center" VerticalAlignment="Center">
    <StackPanel Orientation="Horizontal">
        <Image Source="reset.png" />
        <AccessText>_Reset</AccessText>
    </StackPanel>
</Button>
```

Im Beispiel wird durch die horizontalen Ausdehnungen von **Button**-Objekt und Bitmap der unerwünschte optische Auftritt des **AccessText**-Elements verhindert:

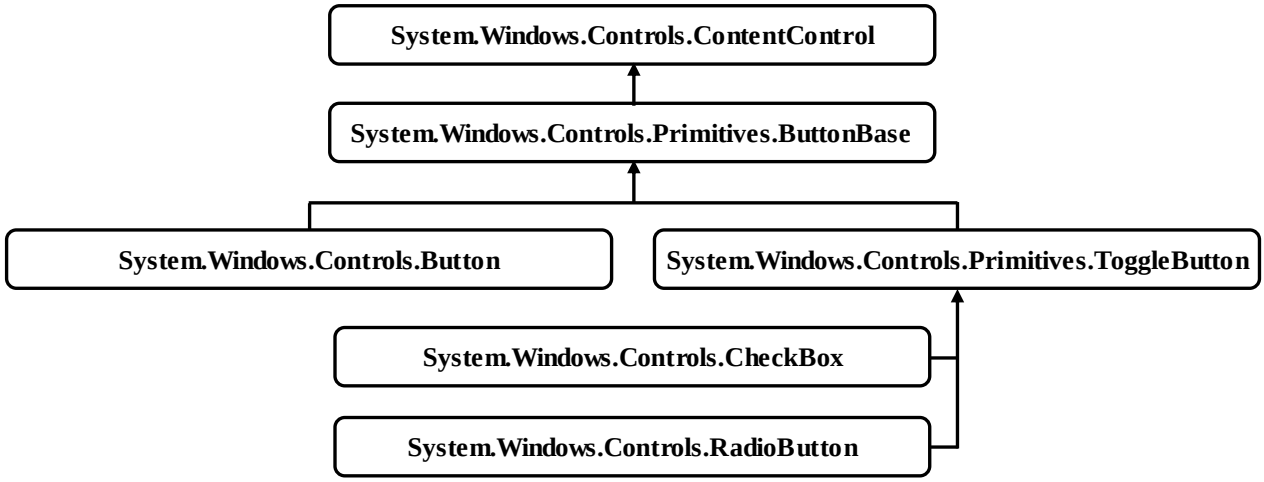


12.7.4.2 Kontrollkästchen und Optionsfelder

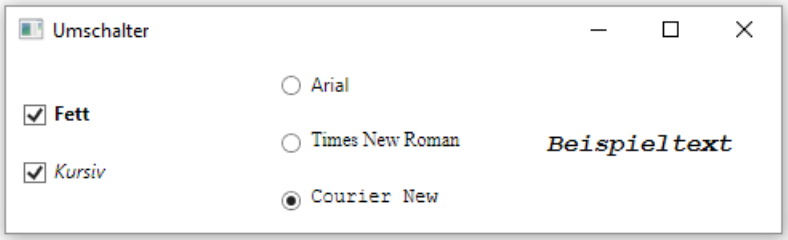
In diesem Abschnitt werden zwei Umschalter vorgestellt:

- Für **Kontrollkästchen** steht die Klasse **CheckBox** zur Verfügung.
- Für ein **Optionsfeld** verwendet man Objekte der Klasse **RadioButton**.

Beide Klassen haben **ToggleButton** als gemeinsame Basisklasse und stammen zusammen mit der schon im Abschnitt 12.7.4.1 vorgestellten Klasse **Button** von der abstrakten Basisklasse **ButtonBase** ab:



Im folgenden Programm kann für den Text eines **Label**-Steuerelements über zwei Kontrollkästchen die Schriftauszeichnung und über ein Optionsfeld die Schriftart gewählt werden:



Beim Fensterdesign per XAML-Code kommt ein dreispaltiger **Grid**-Layoutcontainer zum Einsatz. In den beiden ersten Spalten arbeitet jeweils ein vertikal orientierter **StackPanel**-Layoutcontainer mit den **CheckBox**- bzw. **RadioButton**-Objekten als Insassen. In der dritten Spalte befindet sich das **Label**-Objekt mit dem Beispieltext:

```
<Window x:Class="Umschalter.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Umschalter" Height="150" Width="500">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition /> <ColumnDefinition /> <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <StackPanel  VerticalAlignment="Center" ButtonBase.Click="CheckBox_Click">
            <CheckBox Name="chkBold" Content="Fett" Margin="10"
                      FontWeight="Bold" />
            <CheckBox Name="chkItalic" Content="Kursiv" Margin="10"
                      FontStyle="Italic"/>
        </StackPanel>
        <StackPanel  Grid.Column="1" VerticalAlignment="Center"
                     ButtonBase.Click="RadioButton_Click">
            <RadioButton Name="rbArial" Content="Arial" Margin="10"
                         FontFamily="Arial" IsChecked="True"/>
            <RadioButton Name="rbTimesNewRoman" Content="Times New Roman" Margin="10"
                         FontFamily="Times New Roman"/>
            <RadioButton Name="rbCourierNew" Content="Courier New" Margin="10"
                         FontFamily="Courier New"/>
        </StackPanel>
        <Label  Name="lblTextbeispiel" Grid.Column="2" Content="Beispieltext"
                Margin="10" VerticalAlignment="Center"
                FontFamily="Arial" FontSize="16" />
    </Grid>
</Window>
```

Damit initial die Option *Arial* markiert ist, wird beim zugehörigen **RadioButton**-Objekt die Eigenschaft **IsChecked** auf den Wert **true** gesetzt. Das **Label**-Objekt verwendet beim Start eine Schrift aus der Familie Arial ohne Auszeichnung in der Größe 16.

Alle unmittelbar zu einem Container gehörenden **RadioButton**-Objekte werden als *Gruppe* behandelt, wobei nur *ein* Mitglied eingerastet sein kann (Wert **true** bei der Eigenschaft **IsChecked**). Alternativ kann die Gruppenzugehörigkeit explizit durch das XAML-Attribut **GroupName** festgelegt werden. Alle **RadioButton**-Elemente mit demselben Wert bilden eine Gruppe, z. B.:

```
<RadioButton Name="rbOne" GroupName="g1" Content="One" ... />
```

Für die beiden Kontrollkästchen (`chkBold` und `chkItalic` genannt[1]) ist dieselbe **Click**-Behandlungsmethode `CheckBox_Click()` zuständig, die für das separate Ein- bzw. Ausschalten der Schriftattribute **fett** und *kursiv* sorgt:

```
private void CheckBox_Click(object sender, RoutedEventArgs e) {
   if (chkBold.IsChecked == true)
      lblTextbeispiel.FontWeight = FontWeights.Heavy;
   else
      lblTextbeispiel.FontWeight = FontWeights.Normal;
   if (chkItalic.IsChecked == true)
      lblTextbeispiel.FontStyle = FontStyles.Italic;
   else
      lblTextbeispiel.FontStyle = FontStyles.Normal;
}
```

Wer vermutet, die Eigenschaft **IsChecked** sei vom Typ **bool**, empfindet den logischen Ausdruck

```
chkBold.IsChecked == true
```

als umständlich formuliert. Tatsächlich hat **IsChecked** aber den Typ **Nullable<bool>** bzw. **bool?** (vgl. Abschnitt 8.3), und erst der (zulässige!) Vergleich mit dem Literal **true** sorgt für den benötigten Typ **bool** in der Bedingung für die **if**-Anweisung.

Weil es um ein Routingereignis geht, kann die Behandlungsmethode dem gemeinsamen **StackPanel**-Container zugeordnet werden, statt sie bei *beiden* **CheckBox**-Objekten registrieren zu müssen:

```
<StackPanel  VerticalAlignment="Center" ButtonBase.Click="CheckBox_Click">
    <CheckBox Content="fett" . . . /> <CheckBox Content="kursiv" . . . />
</StackPanel>
```

Im Beispielprogramm wird auch für alle **RadioButton**-Objekte eine gemeinsame **Click**-Behandlungsmethode verwendet:

```
private void RadioButton_Click(object sender, RoutedEventArgs e) {
    if (e.Source is RadioButton)
        lblTextbeispiel.FontFamily = ((RadioButton)e.Source).FontFamily;
}
```

---

[1] Im Beispielprogramm wird die sogenannte *Ungarische Notation* zur Bezeichnung der Instanzvariablennamen verwendet, wobei ein Präfix den Datentyp andeutet (z. B. `lblTextbeispiel`). Die in früheren Zeiten der Windows-Programmierung sehr verbreitete Konvention gilt mittlerweile als veraltet. Microsoft empfiehlt auf der Webseite zu Namenskonventionen

https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/general-naming-conventions

explizit: DO NOT use Hungarian notation. Wir verwenden sie im aktuellen Beispiel trotzdem, weil der zentrale Kritikpunkt nicht allzu zwingend erscheint: Bei einem Wechsel des Datentyps müsse der Name geändert werden. Der Wechsel des Datentyps ist bei einem Steuerelement ziemlich unwahrscheinlich. Außerdem wird im Beispiel der Datentyp nicht im strengen Sinn zum Namensbestandteil.

Die drei hier benötigten Objekte vom Typ **System.Windows.Media.FontFamily** können von den **RadioButton**-Objekten übernommen werden.[1]

Weil es um ein Routingereignis geht, kann die Behandlungsmethode dem gemeinsamen **StackPanel**-Container zugeordnet werden, statt sie bei den drei **RadioButton**-Objekten registrieren zu müssen:

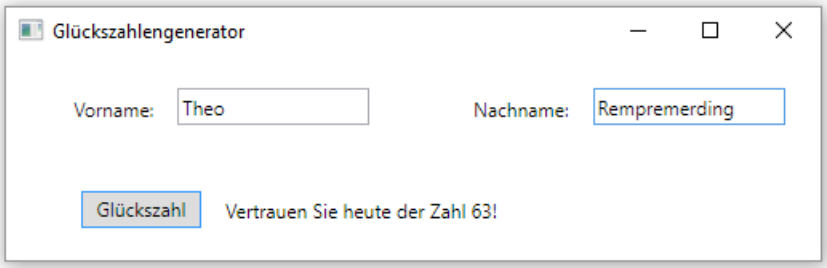
```
<StackPanel Grid.Column="1" VerticalAlignment="Center"
            ButtonBase.Click="RadioButton_Click">
    <RadioButton Name="rbArial" . . ./> <RadioButton Name="rbTimesNewRoman" . . . />
    <RadioButton Name="rbCourierNew" . . . />
</StackPanel>
```

Das komplette Projekt finden Sie im Ordner

…\BspUeb\WPF\Steuerelemente\Umschalter

12.7.4.3 Texteingabefelder

Kurze Texteingaben der Benutzer erfasst man mit Steuerelementen aus der Klasse **TextBox**. Auf dem folgenden Formular eines Programms zur Berechnung der persönlichen Glückszahl in Abhängigkeit vom Vor- und Nachnamen werden zwei **TextBox**-Objekte verwendet:



Das GUI-Design und die Ereignismethodenregistrierung werden durch den folgenden XAML-Code vorgenommen:

```
<Window x:Class="TextBox.MainWindow"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="Glückszahlengenerator" Height="166" Width="525">
  <Grid TextBox.TextChanged="TextBox_TextChanged">
    <Grid.RowDefinitions>
        <RowDefinition /> <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition /> <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <StackPanel Orientation="Horizontal"
                HorizontalAlignment="Center" VerticalAlignment="Center" Margin="10">
      <Label Content ="Vorname:" Height="28" Margin="10"/>
      <TextBox Name="vorname" Height="23" Width="120" VerticalContentAlignment="Center" />
    </StackPanel>
```

[1] Die Idee, die **FontFamily**-Objekte von den **RadioButton**-Objekten zu übernehmen, stammt von Jens Weber (Universität Trier).

```
    <StackPanel Orientation="Horizontal" Grid.Column="1"
                HorizontalAlignment="Center" VerticalAlignment="Center" Margin="10">
      <Label Content ="Nachname:" Height="28" Margin="10"/>
      <TextBox Name="nachname" Height="23" Width="120" VerticalContentAlignment="Center" />
    </StackPanel>
    <StackPanel Orientation="Horizontal" Grid.ColumnSpan="2" Grid.Row="1"
                HorizontalAlignment="Center" VerticalAlignment="Center" Margin="10" >
      <Button Name="button" Content="Glückszahl" Height="23" Width="75"
              Click="button_Click" IsDefault="True" Focusable="False" />
      <Label Name="info" Margin="10" Width="320" Height="28" />
    </StackPanel>
  </Grid>
</Window>
```

Über die **TextBox**-Eigenschaft **Text** kann man auf den Inhalt eines Texteingabefeldes zugreifen, z. B. in der folgenden Behandlungsmethode zum **Click**-Ereignis des Befehlsschalters:

```
private void button_Click(object sender, RoutedEventArgs e) {
  String vn = vorname.Text.ToUpper();
  String nn = nachname.Text.ToUpper();
  int seed = 0; // Startwert des Pseudozufallszahlengenerators
  if (vn.Length > 0 && nn.Length > 0) {
    foreach (char c in vn)
      seed += (int)c;
    foreach (char c in nn)
      seed += (int)c;
    Random zzg = new Random(DateTime.Today.Day + seed);
    info.Content = "Vertrauen Sie heute der Zahl " +
      (zzg.Next(100) + 1).ToString() + "!";
    valToRemove = true;
  }
}
```

Über das Ereignis **TextChanged** kann man auf jede Veränderung der **Text**-Eigenschaft eines **TextBox**-Objekts reagieren. Im Beispielprogramm wird dafür gesorgt, dass eine Glückszahlanzeige verschwindet, sobald sich einer der zugehörigen Texte ändert:

```
private void TextBox_TextChanged(object sender, TextChangedEventArgs e) {
  if (valToRemove) {
    info.Content = strInst;
    valToRemove = false;
  }
}
```

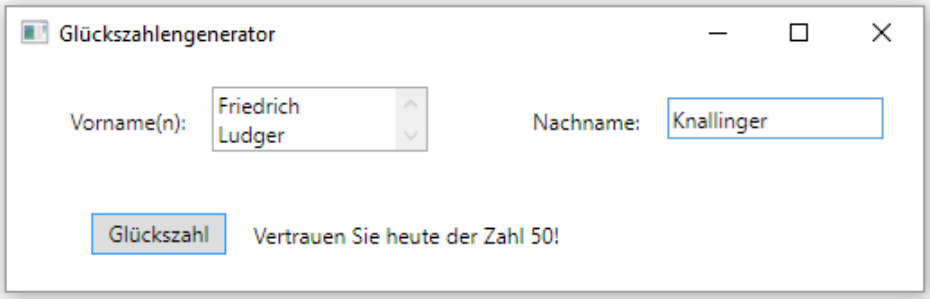
Weil es um ein Routingereignis geht, kann die Behandlungsmethode dem gemeinsamen **Grid**-Container zugeordnet werden, statt sie bei *beiden* **TextBox**-Objekten registrieren zu müssen:

```
<Grid TextBox.TextChanged="TextBox_TextChanged">
  . . .
</Grid>
```

TextBox-Steuerelemente besitzen etliche Kompetenzen, die den Benutzer erfreuen und dabei den Programmierer nicht belasten, z. B.:

- Textmarkierung per Maus oder Tastatur
- Kommunikation mit der Zwischenablage über die Tastenkombinationen **Strg+C**, **Strg+X** und **Strg+V**
- Mehrstufige Rücknahme der letzten Änderungen über **Strg+Z**
- Kontextmenü mit **Bearbeiten**-Items

Um ein *mehrzeiliges* Texteingabefeld zu erhalten,



aktiviert man über den Wert **Wrap** für das **TextWrapping**-Attribut den automatischen Zeilenumbruch, ermöglicht über das Attribut **AcceptsReturn** den manuellen Zeilenumbruch per **Enter**-Taste, aktiviert über das Attribut **VerticalScrollBarVisibility** einen vertikalen Rollbalken und sorgt über das **Height**-Attribut für ausreichend Platz:

```
<TextBox Name="vorname" Height="36" Width="120" VerticalContentAlignment="Center"
  TextWrapping="Wrap" AcceptsReturn="True" VerticalScrollBarVisibility="Visible" />
```

Die Ernennung des **Button**-Objekts zum Standardschalter (über den Wert **true** der Eigenschaft **IsDefault**, siehe Abschnitt 12.7.4.1.1) verliert ihre Wirkung, wenn das **Enter**-berechtigte mehrzeilige Texteingabefeld den Tastatur-Eingabefokus hat.

Trotz **TextWrapping** eignet sich die Klasse **TextBox** nur für kurze Texteingaben. Mit Hilfe des Steuerelements **RichTextBox** kann man einen kompletten Texteditor erstellen.

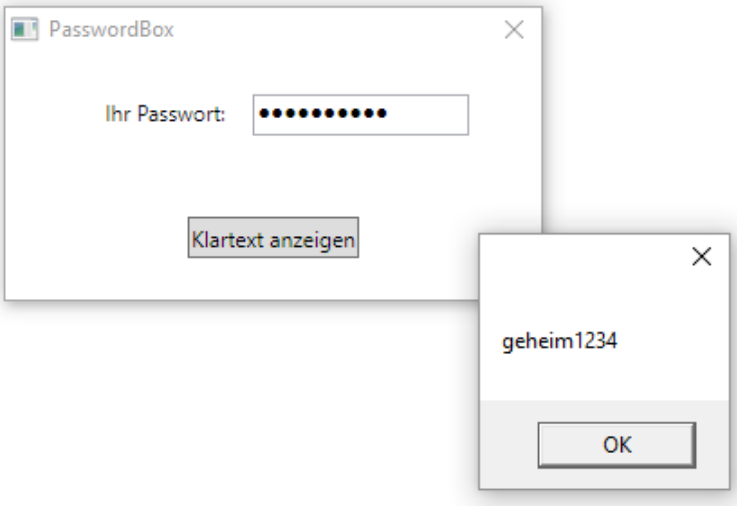
Das vollständige Projekt befindet sich im Ordner

…\BspUeb\WPF\Steuerelemente\Texterfassung\TextBox

Zur Erfassung von **Passwörtern** bietet die WPF-Bibliothek die Klasse **PasswordBox** mit einer gegenüber der verwandten Klasse **TextBox** leicht modifizierten bzw. reduzierten Funktionalität, z. B.:

- Statt der eingegebenen Zeichen des Passworts werden gefüllte Kreise angezeigt.
- Die Eigenschaft **Text** wird durch die Eigenschaft **Password** ersetzt.
- Es wird nicht in die Zwischenablage geschrieben.

Trotz des geänderten Eigenschaftsnamens ist ein erfasstes Passwort im Klartext vorhanden:

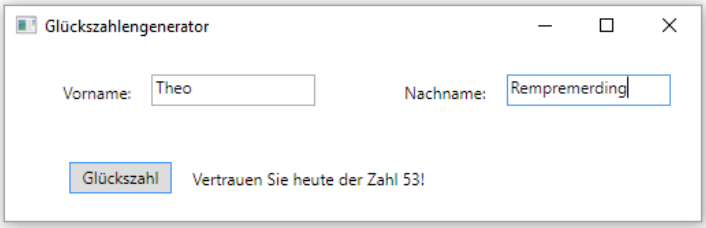


Der XAML-Code zum **PasswordBox**-Objekt:

```
<PasswordBox Name="pw" Height="23" Width="120" />
```

12.7.4.4 Tastatur-Eingabefokus

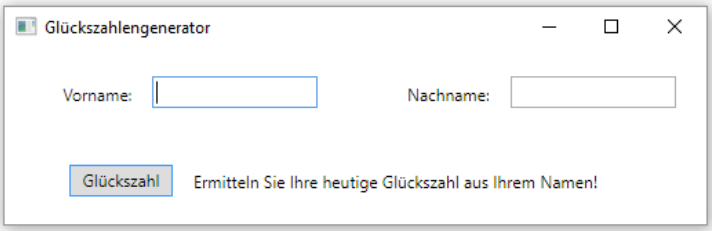
Im Beispielprogramm zum **TextBox**-Steuerelement (siehe Abschnitt 12.7.4.3) erhält der Schalter zur Anforderung einer persönlichen Glückszahl die Rolle des Standardschalters. Folglich kann der Schalter auch dann per **Enter**-Taste ausgelöst werden, wenn eines der Textfelder den Eingabefokus besitzt (u. a. erkennbar an der Texteinfügemarke):



In diesem Beispielprogramm bietet es sich sogar an, den Schalter über den Wert **false** seiner **Focusable**-Eigenschaft aus der Liste der fokussierbaren Steuerelemente zu entfernen, damit die Tabulatortaste nur noch zwischen den beiden Textfeldern wechselt:

```
<Button Content="Glückszahl" . . . IsDefault="True" Focusable="False" />
```

Im Glückszahlengenerator wäre es nett, wenn das Texteingabefeld für den Vornamen schon beim Programmstart den Eingabefokus hätte:



Eine Lösungsmöglichkeit besteht darin, das zu privilegierende Steuerelement über die Methode **Focus()** der Klasse **UIElement** aufzufordern, den Fokus zu übernehmen, z. B.:

```
vorname.Focus();
```

Damit diese Anweisung beim Laden des Fensters ausgeführt wird, stecken wir sie in eine Behandlungsmethode zum **Loaded**-Ereignis der Fensterklasse:

- Markieren Sie im WPF-Designer das Fenster (das **Window**-Objekt).
- Fordern Sie im **Eigenschaften**-Fenster per Mausklick auf das Symbol ⚡ die Liste mit den Ereignissen an.
- Setzen Sie einen Doppelklick auf die Textbox zum Ereignis **Loaded**.
- Daraufhin wird in der Quellcodedatei **MainWindow.xaml.cs** die private Instanzmethode `Window_Loaded()` zu unserer Fensterklasse `MainWindow` angelegt:

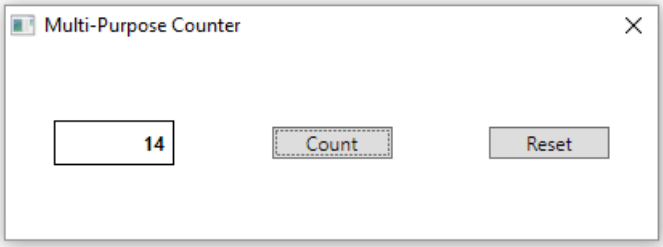
  ```
  private void Window_Loaded(object sender, RoutedEventArgs e) {
  }
  ```

  Diese Methode wird außerdem im XAML-Code zur **Loaded**-Ereignisbehandlung registriert:

  ```
  <Window x:Class="TextBox.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Glückszahlengenerator" Height="166" Width="525" Loaded="Window_Loaded">
  ```

- Ergänzen Sie im Rumpf der Methode `Window_Loaded()` den oben beschriebenen **Focus()** - Aufruf.

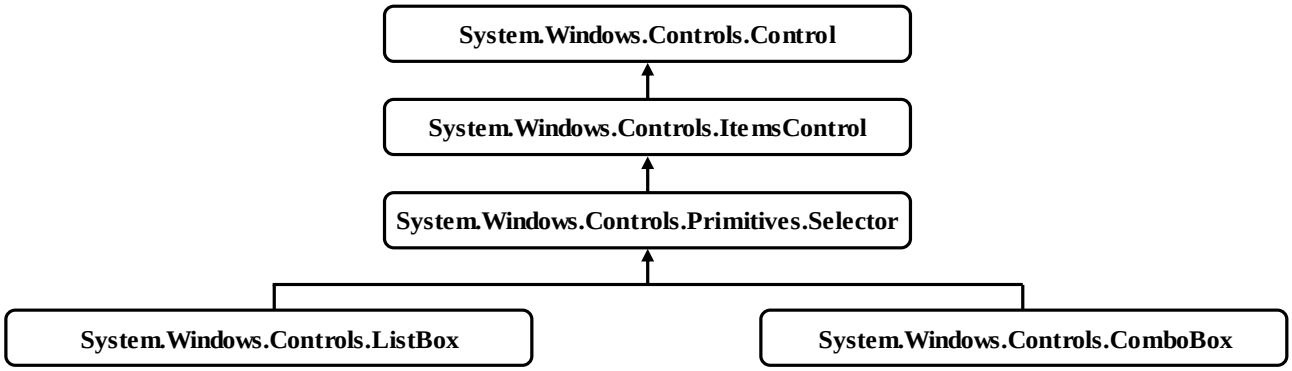
Nachdem ein Schalter per Mausklick oder Tabulatortaste den **Tastatur-Eingabefokus** erhalten hat, kann der Schalter auch per Leertaste angesprochen werden. Dieser Status ist optisch an einem gestrichelten Innenrand zu erkennen:[1]



Wenn der *Standardschalter* den Tastatur-Eingabefokus besitzt, dann entfällt die blaue Standardschalter-Umrandung.

12.7.4.5 Listen- und Kombinationsfelder

In diesem Abschnitt werden die Steuerelementklassen **ListBox** und **ComboBox** vorgestellt, die einen gemeinsamen Stammbaum besitzen:



Ein **ListBox**-Steuerelement präsentiert eine Liste von Elementen, von denen der Benutzer (z. B. durch Mausklicks) eines oder mehrere wählen kann.

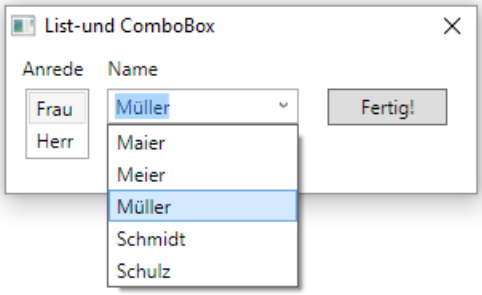
Das **ComboBox**-Steuerelement erfragt einen Text durch eine Kombination aus einem einzeiligen Texteingabefeld und einer Drop-Down-Liste, und der Benutzer hat *zwei* Möglichkeiten zur Beantwortung, sofern die Eigenschaft **IsEditable** den Wert **true** besitzt (siehe unten):

- Text eingeben
- Drop-Down-Liste öffnen und ein Item wählen

Das folgende Programm erlaubt per **ListBox** die Wahl einer Anrede und erleichtert per **ComboBox** die Eingabe des Namens, indem es eine Liste mit häufig auftretenden Namen bereithält:

---

[1] Nach einer Fokusübertragung per Tabulatortaste erscheint der gestrichelte Innenrand sofort. Bei einer Fokusübertragung per Maus wird gleichzeitig das **Click**-Ereignis ausgelöst, und der gestrichelte Innenrand erscheint erst, nachdem das Programm in den Hintergrund verdrängt und dann wieder in den Vordergrund geholt wurde.



12.7.4.5.1 ListBox mit Items befüllen

Ein **ListBox**-Steuerelement kann per XAML-Code über die Kollektionssyntax (siehe Abschnitt 12.3.2.3.5) mit Elementen versorgt werden, z. B.:

```
<ListBox Name="listBox" Margin="10,0,0,0" Width="40" SelectedIndex="0">
    <TextBlock>Frau</TextBlock>
    <TextBlock>Herr</TextBlock>
</ListBox>
```

Diese landen in einem Objekt der Klasse **ItemCollection**, das über die von **ItemsControl** geerbte Eigenschaft **Items** ansprechbar ist und Elemente vom Typ **ListBoxItem** verwaltet. Man darf Elemente von einem beliebigen „vorzeigbaren" Typ in eine **ListBox** einfüllen, z. B. **TextBlock**-Elemente. Diese werden automatisch in **ListBoxItem**-Objekte verpackt. Mit expliziten **ListBoxItem**-Elementen sieht das Beispiel so aus:

```
<ListBox Name="listBox" Margin="10,0,0,0" Width="40" SelectedIndex="0">
    <ListBoxItem>
        <TextBlock>Frau</TextBlock>
    </ListBoxItem>
    <ListBoxItem>
        <TextBlock>Herr</TextBlock>
    </ListBoxItem>
</ListBox>
```

Im Beispiel (mit einfachen Textelementen für die Anrede) ist die explizite **ListBoxItem**-Verpackung überflüssig. Weil die Klasse **ListBoxItem** von **ContentControl** abstammt, bringt ihre explizite Verwendung jedoch den Vorteil, dass ein Item aus mehreren Elementen (z. B. aus einem **Image** und einem **TextBlock**) zusammengesetzt werden kann.

Auch die folgende Variante ist möglich, wobei dann aber die bei **TextBlock**-Elementen vorhandene (und in einem späteren Beispiel genutzte) Möglichkeit zum Formatieren der Texte fehlt:

```
<ListBox Name="listBox" Margin="10,0,0,0" Width="40" SelectedIndex="0">
    <ListBoxItem>Frau</ListBoxItem>
    <ListBoxItem>Herr</ListBoxItem>
</ListBox>
```

Über die Kollektionsmethoden **Add()**, **Remove()** usw. lässt sich die **ItemCollection** zur Laufzeit per Programm verändern, z. B.:

```
listBox.Items.Add(new TextBlock {Text = "Hrl."});
```

12.7.4.5.2 ListBox mit Bindung an eine vorhandene Datenkollektion

Oft befinden sich die in einem **ListBox**-Steuerelement anzuzeigenden Elemente bereits in einer Datenkollektion, und es soll eine Datenbindung (siehe Beispiel im Abschnitt 6.8.6) zwischen dieser Datenkollektion als Quelle und dem **ListBox**-Objekt als Ziel vorgenommen werden. Dazu wird die Datenkollektion der **ListBox**-Eigenschaft **ItemsSource** zugewiesen. Besonders geeignet sind Da-

tenkollektionen aus der Klasse **ObservableCollection<T>** wegen ihrer Fähigkeit, Änderungen der Listenzusammenstellung per Ereignis an ein verbundenes **ListBox**-Steuerelement zu melden, das daraufhin seine Anzeige aktualisiert.

Sobald der **ItemsSource**-Eigenschaft eine Datenkollektion zugewiesen wurde, ist es nicht mehr möglich, die „eingebaute“, per **Items**-Eigenschaft ansprechbare **ItemCollection** des **ListBox**-Steuerelements zu verändern. Ein Versuch führt zu einer **InvalidOperationException**.

Mit einer *Datenkollektion* ist in diesem Abschnitt eine unabhängig vom **ListBox**-Steuerelement vorhandene Kollektion mit Elementen von einem beliebigen Typ gemeint, die vom „internen“ **ItemCollection**-Objekt des **ListBox**-Steuerelements, das Elemente vom Typ **ListBoxItem** verwaltet, unterschieden werden soll.

Wir haben übrigens im `RssFeedReeder`-Projekt (vgl. vor allem Abschnitt 6.8.6) ein realistisches Beispiel für die Bevölkerung eines **ListBox**-Steuerelements durch die Elemente einer Datenkollektion kennengelernt. Dort haben wir Objekte der Klasse `RssItem`

```
internal class RssItem {
   public string Title { get; internal set; }
   public string Description { get; internal set; }
   public string Url { get; internal set; }
}
```

in eine Kollektion vom Typ **List<RssItem>** eingefüllt und diese Kollektion an die **ListBox**-Eigenschaft **ItemsSource** übergeben:[1]

```
listBox.ItemsSource = items;
```

Um zu einer informativen und optisch attraktiven Anzeige der **ListBox**-Items zu kommen, verwendet man ein geeignet konfiguriertes Objekt der Klasse **DataTemplate**, das der **ListBox**-Eigenschaft **ItemTemplate** zugewiesen wird. Im `RssFeedReeder`-Projekt haben wir den folgenden XAML-Code mit einem Eigenschaftselement vom Typ **DataTemplate** verwendet:

```
<ListBox x:Name="listBox" Margin="10,50,10,10"
     ScrollViewer.HorizontalScrollBarVisibility="Disabled"
     MouseDoubleClick="listBox_MouseDoubleClick">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel>
        <TextBlock Text="{Binding Path=Title}" Margin="1" TextAlignment="Center"
                   TextWrapping="Wrap" FontSize="14" FontWeight="Bold"
                   Foreground="DarkMagenta" />
        <TextBlock Text="{Binding Path=Description}" Margin="1,1,1,4"
                   TextWrapping="Wrap" />
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Ein **DataTemplate**-Objekt hat die (von **FrameworkTemplate**) geerbte Inhaltseigenschaft **VisualTree** (siehe Abschnitt 12.3.2.3.5 zu Inhaltseigenschaften). Zur Versorgung der Eigenschaft **VisualTree** ist ein Wurzel-Container anzugeben.[2] Im Beispiel wird ein **StackPanel**-

---

[1] Die permanente Synchronisation einer variablen und beobachtbaren Liste mit dem **ListBox**-Steuerelement haben wir im `RssFeedReeder`-Projekt nicht benötigt. Auf die folgende Weise könnte man aus der **List<RssItem>** - Kollektion eine *beobachtbare* Kollektion erstellen:

```
ObservableCollection<RssItem> obsItems = new ObservableCollection<RssItem>(items);
```

Der Lösungsvorschlag stammt der Webseite:

https://stackoverflow.com/questions/18095932/how-to-cast-a-listt-to-an-observablecollectiont-in-wpf/18095988

[2] https://docs.microsoft.com/de-de/dotnet/api/system.windows.frameworktemplate.visualtree

Layoutcontainer verwendet, der zwei vertikal gestapelte **TextBlock**-Elemente enthält. Deren **Text**-Eigenschaft wird jeweils per Markup-Erweiterung (vgl. Abschnitt 12.3.2.3.6) vom Typ **Binding** mit einer Eigenschaft (`Title` bzw. `Description`) der Elemente in der mit **ItemsSource** ansprechbaren Datenkollektion verbunden.

12.7.4.5.3 ComboBox mit Items befüllen

Zur Erläuterung des **ComboBox**-Steuerelements kehren wir zum Beispielprogramm mit Wahlmöglichkeiten für Anrede und Namen zurück (siehe Einstieg im Abschnitt 12.7.4.5). Im XAML-Code zeigen sich kaum Unterschiede zwischen dem **ComboBox**- und dem **ListBox**-Element:

```
<ComboBox Name="comboBox" Margin="10,0,0,0" Width="120" IsEditable="True">
    <TextBlock>Maier</TextBlock>
     . . .
    <TextBlock>Schulz</TextBlock>
</ComboBox>
```

Es bestehen noch mehr Gemeinsamkeiten, z. B.:

- bei der (von **ItemsControl** geerbten) Eigenschaft **Items**, die auch bei der Klasse **ComboBox** auf ein Objekt der Klasse **ItemCollection** zeigt,
- beim Verfahren zur Datenbindung.

In seiner **ItemCollection** verwaltet ein **ComboBox**-Objekt Elemente vom Typ **ComboBoxItem**, die analog zum Typ **ListBoxItem** (siehe Abschnitt 12.7.4.5.1) entweder implizit oder explizit erstellt werden (MacDonald 2012, S. 187).

Ob ein **ComboBox**-Objekt ein Texteingabefeld anbietet, hängt von der Eigenschaft **IsEditable** ab, die im Beispiel auf den Wert **true** gesetzt wird.

12.7.4.5.4 Gewähltes Item ermitteln

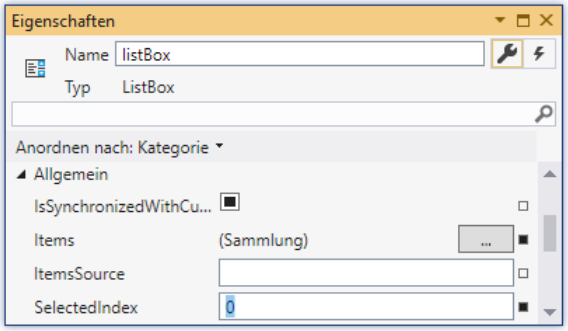
Wie sich in der **Click**-Behandlungsmethode zur Schaltfläche des Beispielprogramms mit Wahlmöglichkeiten für Anrede und Namen zeigt, ist beim **ListBox**-Objekt die aktuelle Wahl über die Eigenschaft **SelectedItem** ansprechbar:

```
private void button_Click(object sender, RoutedEventArgs e) {
   MessageBox.Show("Guten Tag, " + ((TextBlock)listBox.SelectedItem).Text + " " +
      comboBox.Text);
}
```

Weil diese Eigenschaft den Datentyp **Object** besitzt, ist die Textextraktion erst nach einer Typumwandlung möglich.

Von einem **ComboBox**-Objekt erfährt man die aktuelle Wahl bzw. Eintragung des Benutzers über die Eigenschaft **Text**. Zwar ist auch hier die Eigenschaft **SelectedItem** mit dem Indexwert zur aktuellen Auswahl vorhanden, doch ist bei diesem Steuerelement die Eigenschaft **Text** mit dem Datentyp **String** bequemer.

Bei einem **ListBox**-Objekt kann man über die Eigenschaft **SelectedIndex** dafür sorgen, dass der Anwenderbequemlichkeit halber initial ein Listenelement markiert ist, z. B.:



Im aktuellen Beispielprogramm sind die durchaus zahlreich vorhandenen **ListBox**- bzw. **ComboBox**-Ereignisse (z. B. **SelectionChanged**) nicht von Interesse.

Das vollständige Projekt befindet sich im Ordner

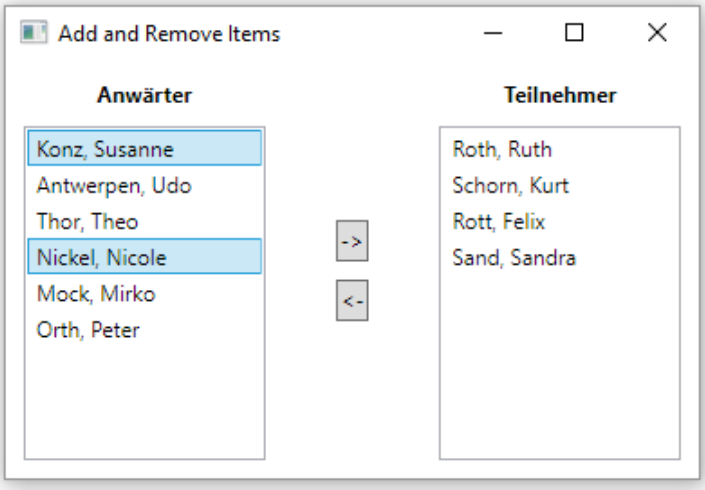
…\BspUeb\WPF\Steuerelemente\Listen\List- und ComboBox

12.7.4.5.5 Dynamische Listen und Mehrfachwahl

In einem weiteren Beispielprogramm sollen für **ListBox**-Objekte die folgenden Techniken demonstriert werden:

- Dynamische Aufnahme und Entfernung von Listenelementen
- Mehrfachauswahl von Listenelementen

Wir befüllen ein erstes **ListBox**-Objekt mit den Namen von Anwärtern und halten ein zweites **ListBox**-Objekt für die ausgewählten Teilnehmer bereit. In den beiden **ListBox**-Steuerelementen wird jeweils das interne **ItemCollection**-Objekt zur Verwaltung der Namen verwendet. Auf dem Anwendungsfenster befinden sich zwischen den beiden **ListBox**-Objekten zwei Transportschalter für das Verschieben von Namen zwischen den Listen:



Im XAML-Code wird durch verschachtelte Layoutcontainer für ein sinnvolles Verhalten bei variabler Fenstergröße gesorgt:

```
<Window x:Class="ListBox.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Add and Remove Items" Height="270" Width="400">
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="2*" />
      <ColumnDefinition Width="*" />
      <ColumnDefinition Width="2*" />
    </Grid.ColumnDefinitions>
    <DockPanel>
      <Label Content="Anwärter" DockPanel.Dock="Top" Margin="5"
             HorizontalAlignment="Center" FontWeight="Bold" />
      <ListBox Name="listeAnwaerter" Margin="10,0,10,10" SelectionMode="Extended">
        <TextBlock>Rott, Felix</TextBlock>
        <TextBlock>Konz, Susanne</TextBlock>
           . . .
        <TextBlock>Sand, Sandra</TextBlock>
        <TextBlock>Mock, Mirko</TextBlock>
      </ListBox>
    </DockPanel>
    <StackPanel Grid.Column="1" VerticalAlignment="Center"
          Button.Click="Button_Click">
      <Button Name="cmdRein" Content="->" Height="23"
          HorizontalAlignment="Center" Margin="5" />
      <Button Name="cmdRaus" Content="&lt;-" Height="23"
          HorizontalAlignment="Center" Margin="5" />
    </StackPanel>
    <DockPanel Grid.Column="2" >
      <Label Content="Teilnehmer" DockPanel.Dock="Top" Margin="5"
             HorizontalAlignment="Center" FontWeight="Bold" />
      <ListBox Name="listeTeilnehmer" Margin="10,0,10,10" SelectionMode="Extended" />
    </DockPanel>
  </Grid>
</Window>
```

Dem dreispaltigen Top-Level - Container vom Typ **Grid** sind untergeordnet:

- Links und rechts jeweils ein **DockPanel**-Container, der ein **Label**- und ein **ListBox**-Objekt verwaltet, wobei das zuletzt eingefügte **ListBox**-Objekt den gesamten unverbrauchten Raum einnimmt.
- In der Mitte ein vertikal orientierter **StackPanel**-Container für die beiden Schaltflächen.

Im Beispielprogramm können aus jeder Liste einzelne Kandidaten flexibel ausgewählt und in die jeweils andere Liste transportiert werden. Für die Markierungsflexibilität wird über die **ListBox**-Eigenschaft **SelectionMode** mit dem Wert **Extended** gesorgt:

- Bei gedrückter **Strg**-Taste können per Mausklick *mehrere* Elemente nacheinander markiert werden.
- Bei gedrückter **Umschalt**-Taste lässt sich eine Serie von hintereinander positionierten Elementen markieren, indem zunächst auf das erste und danach auf das letzte Element ein Mausklick gesetzt wird.
- Durch gemeinsames Drücken von **Strg**- und **Umschalt**-Taste lassen sich mehrere Serien und Einzelelemente durch Mausklicks in die Markierung einbeziehen.

Erhält die Eigenschaft **SelectionMode** den Wert **Multiple**, haben die **Strg**- und die **Umschalt**-Taste keine Funktion, jedoch können durch einfache Mausklicks mehrere Elemente nacheinander markiert werden.

Beim Rücktransportschalter muss die als Beschriftung gewünschte öffnende spitze Klammer wegen der Kollision mit der XML-Syntax etwas umständlich notiert werden:[1]

```
Content="&lt;-"
```

In der **Click**-Ereignismethode `Button_Click()` zu den beiden Schaltflächen (`cmdRein`, `cmdRaus`) werden die markierten Elemente über die **ListBox**-Eigenschaft **SelectedItems** angesprochen, die auf ein Objekt einer Klasse zeigt, die das Interface **System.Collections.IEnumerable** erfüllt. Offenbar zeigt **SelectedItems** Ähnlichkeiten mit der bereits bekannten Eigenschaft **Items**, die auf eine Liste mit *allen* Elementen zeigt.

```
private void Button_Click(object sender, RoutedEventArgs e) {
    if (e.Source == cmdRein) {
        Object[] tempAnwaerter = new Object[listeAnwaerter.SelectedItems.Count];
        listeAnwaerter.SelectedItems.CopyTo(tempAnwaerter, 0);
        foreach (object anw in tempAnwaerter) {
            listeAnwaerter.Items.Remove(anw);
            listeTeilnehmer.Items.Add(anw);
        }
    } else {
        Object[] tempTeilnehmer = new Object[listeTeilnehmer.SelectedItems.Count];
        listeTeilnehmer.SelectedItems.CopyTo(tempTeilnehmer, 0);
        foreach (object anw in tempTeilnehmer) {
            listeTeilnehmer.Items.Remove(anw);
            listeAnwaerter.Items.Add(anw);
        }
    }
    SortDescription sd = new SortDescription("Text", ListSortDirection.Ascending);
    listeAnwaerter.Items.SortDescriptions.Add(sd);
    listeTeilnehmer.Items.SortDescriptions.Add(sd);
}
```

Die zu verschiebenden Namen werden in einem temporären **Object**-Array gesammelt, über den anschließend eine **foreach**-Schleife iteriert, wobei die Listen verändert werden. Wenn man z. B. die Liste `listeAnwaerter.SelectedItems` als Kollektion einer **foreach**-Schleife verwendet, dann kommt es zu einer **InvalidOperationException**, weil sich die Kollektion einer **foreach**-Schleife während der Iteration nicht ändern darf.

Durch die letzten drei Zeilen der Methode `Button_Click()` werden die beiden Listen in sortiertem Zustand gehalten. Die **ItemCollection** einer **ListBox** besitzt eine Eigenschaft namens **SortDescriptions**, die auf ein Objekt der Klasse **SortDescriptionCollection** zeigt. Diese Kollektion enthält Instanzen der Struktur **SortDescription**, die jeweils ein Kriterium zum Sortieren der **ListBox**-Elemente beschreiben über:

- Name der Elementeigenschaft, nach der sortiert werden soll
- Sortierrichtung (auf- oder absteigend)

Im Beispiel sollen **TextBlock**-Elemente nach ihrer **Text**-Eigenschaft aufsteigend sortiert werden. Daher wird die folgende **SortDescription**-Instanz

```
SortDescription sd = new SortDescription("Text", ListSortDirection.Ascending);
```

erstellt und von beiden **ListBox**-Steuerelementen zum Sortieren verwendet, z. B.:

```
listeAnwaerter.Items.SortDescriptions.Add(sd);
```

---

[1] Warum im Wert eines XML-Attributs (abgegrenzt durch doppelte Anführungszeichen) das Zeichen „>“ erlaubt, das Zeichen „<“ hingegen verboten ist, können XML-Experten sicher erklären.

Weil es um das Routingereignis **Click** geht, kann die Behandlungsmethode dem gemeinsamen **StackPanel**-Container zugeordnet werden, statt sie bei beiden **Button**-Objekten registrieren zu müssen:

```
<StackPanel Grid.Column="1" VerticalAlignment="Center"
            Button.Click="Button_Click">
    <Button Content="->" . . . />
    <Button Content="&lt;-" . . . />
</StackPanel>
```

In der Ereignismethode lässt sich daher die Ereignisquelle *nicht* über den Parameter **sender** feststellen, der stets auf das **StackPanel**-Objekt zeigt. Stattdessen ist die Eigenschaft **Source** des Ereignisbeschreibungsobjekts aus der Klasse **RoutedEventArgs** zu verwenden, das über den zweiten Parameter geliefert wird.

Das vollständige Projekt befindet sich im Ordner

…\BspUeb\WPF\Steuerelemente\Listen\AddRemoveItems

12.7.4.6 ToolTip

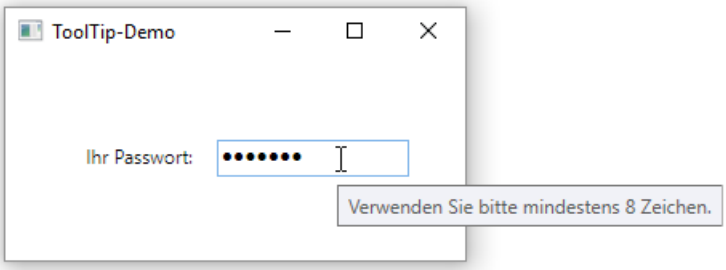
Über ein **ToolTip**-Objekt realisiert man eine vorübergehend sichtbare Information zu einem Steuerelement:

- Die Anzeige erscheint in der Nähe des Steuerelements, wenn sich die Maus über diesem Element befindet.
- Das **ToolTip**-Element verschwindet, wenn der Mauszeiger den Bereich des zu erläuternden Steuerelements verlässt, oder die Anzeigedauer abgelaufen ist (Voreinstellung: 5 Sekunden).

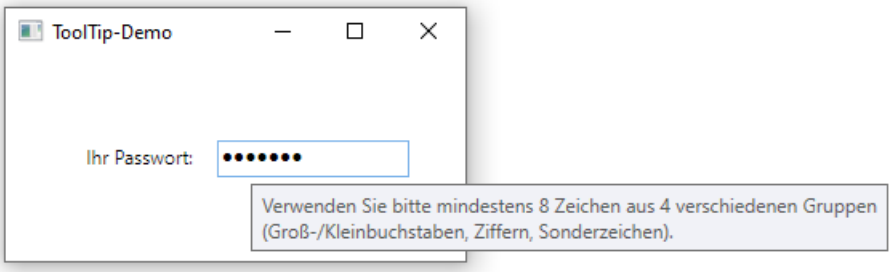
Statt im XAML-Code ein Eigenschaftselement vom Typ **ToolTip** zu deklarieren, kann man sich meist darauf beschränken, für das zu erläuternde Element ein **ToolTip**-Attribut zu formulieren, z. B.:

```
<PasswordBox ToolTip="Verwenden Sie bitte mindestens 8 Zeichen"
             Height="23" Name="pw" Width="120"/>
```

Hier erhalten die Benutzer einen Tipp zur Passwortlänge:



Wenn sich ein **ToolTip**-Text über mehrere Zeilen erstrecken soll,



dann eignet sich die folgende Lösung mit dem Eigenschaftselement **PasswordBox.ToolTip** und einem **TextBlock** als Wert für die Eigenschaft **ToolTip**:

```
<PasswordBox Height="23" Name="pw" Width="120">
    <PasswordBox.ToolTip>
        <TextBlock>
        Verwenden Sie bitte mindestens 8 Zeichen aus 4 verschiedenen Gruppen
        <LineBreak/>
        (Groß-/Kleinbuchstaben, Ziffern, Sonderzeichen).
        </TextBlock>
    </PasswordBox.ToolTip>
</PasswordBox>
```

Um die voreingestellte Anzeigedauer von 5000 Millisekunden zu verändern, benutzt man die angefügte Eigenschaft (vgl. Abschnitt 12.5.2) **ShowDuration** der Klasse **ToolTipService**, z. B.:

```
<PasswordBox ToolTipService.ShowDuration="3000" Height="23" Name="pw" Width="120">
    <PasswordBox.ToolTip>
        . . .
    </PasswordBox.ToolTip>
</PasswordBox>
```

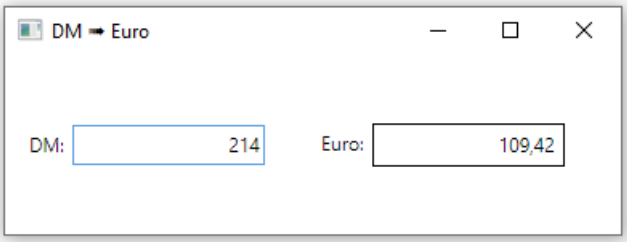
Über weitere angefügte Eigenschaften der Klasse **ToolTipService** lassen sich weitere **ToolTip**-Verhaltensweisen ändern (z. B. Startverzögerung, Ausstattung mit einem Schlagschatten).

12.7.5 Datenbindung zwischen zwei Steuerelementen

In den vielen WPF-Anwendungen spielt die zuverlässige und unaufwändige Kopplung zwischen

- komplexen Datenbeständen (in CLR-Objekten, relationalen Datenbanken oder hierarchisch organisierten XML-Dateien) einerseits
- und GUI-Elementen zur Anzeige und Modifikation der Daten andererseits

eine wichtige Rolle. Erfreulicherweise bietet die WPF gerade beim Thema Datenbindung eine exzellente Unterstützung für die Entwickler. Wir beschränken uns aus Zeitgründen auf die Datenbindung zwischen zwei WPF-Steuerelementen und betrachten das folgende, von Paul Frischknecht erstellte Programm, das den in ein **TextBox**-Steuerelement eingetragenen DM-Betrag spontan in einen Euro-Betrag konvertiert und in einem **Label**-Steuerelement anzeigt:



Vom XAML-Code

```
<Window x:Class="DmToEuroBinding.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="DM &#x27a0; Euro" Height="150" Width="400" Loaded="Window_Loaded" >
    <Grid Margin="10">
        <Grid.ColumnDefinitions>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <StackPanel Orientation="Horizontal" VerticalAlignment="Center">
            <Label Content="DM:" />
            <TextBox x:Name="tbEingabe" VerticalContentAlignment="Center" MinWidth="120"
             TextAlignment="Right" />
        </StackPanel>
```

```
        <StackPanel Orientation="Horizontal" Grid.Column="1" VerticalAlignment="Center">
            <Label Content="Euro:"/>
            <Label BorderBrush="Black" BorderThickness="1" MinWidth="120"
                   HorizontalContentAlignment="Right"
                   Content="{Binding ElementName=tbEingabe, Path=Text,
                            Converter={StaticResource DmToEuroConverter}}" />
        </StackPanel>
    </Grid>
</Window>
```

interessiert vor allem das **Label**-Steuerelement auf der rechten Seite des horizontalen **StackPanel**-Containers in der zweiten Spalte des **Grid**-Containers. Seine **Content**-Eigenschaft wird über ein **Binding**-Objekt mit einer Zeichenfolge versorgt:

- Die **Binding**-Eigenschaft **ElementName** benennt als Quelle das **TextBox**-Element mit dem Namen `eingabe`.
- Die **Binding**-Eigenschaft **Path** benennt die zu verwendende Eigenschaft des Quellobjekts.

Die **Text**-Eigenschaft des Quellelements wird aber nicht direkt verwendet, sondern konvertiert durch die in der Code-Behind - Datei **MainWindow.xaml.cs** definierte Klasse `DmToEuroConverter`, die das Interface **IValueConverter** implementiert, das u. a. die Methode **Convert()** vorschreibt:

```
using System;
using System.Windows;
using System.Windows.Data;

namespace DmToEuroBinding {
    public partial class MainWindow : Window {
        public MainWindow() {
            InitializeComponent();
        }

        private void Window_Loaded(object sender, RoutedEventArgs e) {
            tbEingabe.Focus();
        }
    }

    public class DmToEuroConverter : IValueConverter {
        const double factor = 1.95583;
        public object Convert(object value, Type targetType, object parameter,
                              System.Globalization.CultureInfo culture) {
            string s = "";
            try {
                s = string.IsNullOrEmpty((string)value) ? "" :
                    (System.Convert.ToDouble(value) / factor).ToString("0.00");
            } catch {
            }
            return s;
        }
        public object ConvertBack(object value, Type targetType, object parameter,
                                  System.Globalization.CultureInfo culture) {
            throw new NotImplementedException();
        }
    }
}
```

Der Konditionaloperator in der **return**-Anweisung der `DmToEuroConverter`-Methode **Convert()** liefert ...

- eine leere Zeichenfolge, wenn die **Text**-Eigenschaft der Quelle leer oder nicht als **double**-Wert interpretierbar ist,
- bei einer sinnvollen Eingabe eine Zeichenfolge mit dem Euro-Betrag.

In der Datei **App.xaml** wird ein Objekt der Klasse `DmToEuroConverter` als Ressource vereinbart:

```
<Application x:Class="DmToEuroBinding.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:DmToEuroBinding"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
        <local:DmToEuroConverter x:Key="DmToEuroConverter"/>
    </Application.Resources>
</Application>
```

Dieses Objekt wird im XAML-Code zum Anwendungsfenster der **Converter**-Eigenschaft des **Binding**-Objekts zugewiesen:

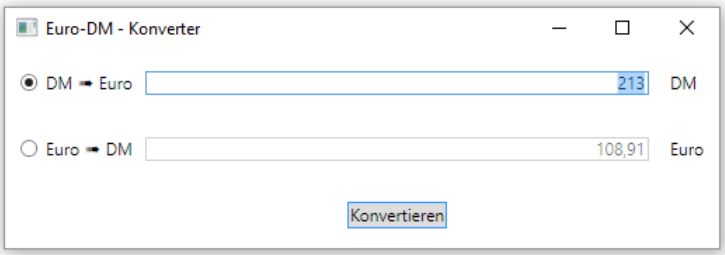
```
<Label BorderBrush="Black" BorderThickness="1" MinWidth="120"
        HorizontalContentAlignment="Right"
        Content="{Binding ElementName=eingabe, Path=Text,
                  Converter={StaticResource DmToEuroConverter}}" />
```

Das komplette Projekt finden Sie im Ordner

…\BspUeb\WPF\DmToEuroBinding

12.8 Übungsaufgaben zum Kapitel 12

1) Erstellen Sie eine verbesserte Variante des Euro-DM - Konverters, den wir im Abschnitt 3.3.4 entwickelt haben. Die neue Version sollte ungefähr die folgende Bedienoberfläche besitzen:



Hinweise:

- Das untere **TextBox**-Steuerelement soll nur zur Ausgabe dienen. Daher sollten über die Eigenschaft **IsEnabled** Benutzereingaben verhindert werden.
- Bei dem horizontalen Pfeil in den Beschriftungen der Optionsschalter handelt es sich um das Unicode-Zeichen mit der Nummer 0x27A0. Über eine spezielle Escape-Sequenz lassen sich beliebige Unicode-Zeichen in die XAML-Syntax integrieren, z. B.:

  ```
  Content="DM &#x27a0; Euro"
  ```

13 Ausnahmebehandlung

Durch Programmierfehler (z. B. versuchter Feldzugriff mit ungültigem Indexwert, Ganzzahldivision durch 0) oder durch besondere Umstände (z. B. Speichermangel, unterbrochene Netzwerkverbindung) kann die reguläre Ausführung einer Methode scheitern. C# bietet ein modernes Verfahren zur Meldung und Behandlung von Störungen der Programmausführung: An der Unfallstelle wird ein Ausnahmeobjekt aus der Klasse **Exception** (im Namensraum **System**) oder aus einer problemspezifischen Unterklasse erzeugt und der unmittelbar verantwortlichen Methode „zugeworfen“. Diese Methode wird somit über das Problem informiert und mit Daten für die Behandlung versorgt.

Die Initiative beim Auslösen einer Ausnahme kann ausgehen …

- von der **CLR**
  Entdeckt die CLR einen Fehler, der nicht zu schwerwiegend ist und vom Anwendungsprogramm prinzipiell behoben werden kann, dann wirft sie ein Ausnahmeobjekt, z. B. ein Objekt aus der Klasse **ArithmeticException** bei einer versuchten Ganzzahldivision durch 0.
- vom **Anwendungsprogramm**, wozu auch die verwendeten Bibliothekstypen gehören
  Zum Werfen einer Ausnahme dient die **throw**-Anweisung (siehe Abschnitt 13.5).

Die unmittelbar von einer Ausnahme betroffene Methode steht meist am Ende einer Sequenz verschachtelter Aufrufe, und entlang der Aufrufersequenz haben die beteiligten Methoden jeweils folgende Reaktionsmöglichkeiten:

- das Ausnahmeobjekt abfangen und das Problem behandeln
  Im tatsächlichen Programmablauf fliegen natürlich keine Objekte durch die Gegend, die mit irgendwelchen Gerätschaften eingefangen werden. Stattdessen überprüft die Laufzeitumgebung, ob die betroffene Methode geeigneten Code zur Behandlung des Ausnahmeobjekts (einen sogenannten *Exception-Handler*) enthält. Gegebenenfalls wird dieser Exception-Handler ausgeführt und erhält quasi als Aktualparameter das Ausnahmeobjekt mit Informationen über das Problem. Nach der Ausnahmebehandlung kann die Methode ...
  - entweder ihre Tätigkeit mit einem angepassten Handlungsplan fortsetzen
  - oder ihrerseits ein Ausnahmeobjekt werfen (entweder das ursprüngliche oder ein informativeres) und somit die Kontrolle an ihren Aufrufer zurückgeben.
- das Ausnahmeobjekt ignorieren
  In diesem Fall besitzt eine Methode keinen zum Ausnahmeobjekt passenden Exception-Handler. Die Methode wird beendet, und das Ausnahmeobjekt wird dem Vorgänger in der Aufrufersequenz überlassen.

Wir werden uns anhand verschiedener Versionen eines Beispielprogramms damit beschäftigen,

- was bei unbehandelten Ausnahmen geschieht,
- wie man Ausnahmen abfängt,
- wie man selbst Ausnahmen wirft,
- wie man eine eigene Ausnahmeklasse definiert.

Man kann von keinem Programm erwarten, dass es unter allen widrigen Umständen normal funktioniert. Doch müssen Schäden (z. B. Datenverluste) nach Möglichkeit verhindert werden, und der Benutzer sollte eine nützliche Information zum aufgetretenen Problem erhalten. Bei vielen Methodenaufrufen ist es realistisch und erforderlich, auf Störungen des normalen Ablaufs vorbereitet zu sein. Dies folgt schon aus **Murphy’s Law** (zitiert nach Wikipedia):[1]

> Anything that can go wrong will go wrong.

[1] https://de.wikipedia.org/wiki/Murphys_Gesetz

In C# wird allerdings keine Methode gezwungen, sich auf bestimmte (oder gar alle) zu befürchtende Ausnahmen mit einem passenden Exception-Handler vorzubereiten.

13.1 Unbehandelte Ausnahmen

Findet die CLR zu einer geworfenen Ausnahme entlang der Aufrufersequenz bis hinauf zur Methode **Main()** keinen passenden Exception-Handler, dann wird das Programm mit einer Fehlermeldung beendet.[1] Das folgende Konsolenprogramm soll die Fakultät zu einer Zahl berechnen, die beim Start als Befehlszeilenargument übergeben wurde. Dabei beschränkt sich die **Main()** - Methode auf die eigentliche Fakultätsberechnung und überlässt die Konvertierung und Validierung der übergebenen Zeichenfolge der Methode `Kon2Int()`. Diese wiederum stützt sich bei der Konvertierung auf die statische Methode **Parse()** der BCL-Struktur **Int32**:

```
using System;

class Fakul {
   static int Kon2Int(string instr) {
      int arg = Int32.Parse(instr);
      if (arg >= 0  &&  arg <= 170)
         return arg;
      else
         return -1;
   }

   static void Main(string[] args) {
      if (args.Length == 0) {
         Console.WriteLine("Kein Argument angegeben");
         Console.Read();
         Environment.Exit(1);
      }
      int argument = Kon2Int(args[0]);
      if (argument != -1) {
         double fakul = 1.0;
         for (int i = 1; i <= argument; i++)
               fakul = fakul * i;
         Console.WriteLine("Fakultät von {0}: {1}", args[0], fakul);
      }
      else
         Console.WriteLine("Keine ganze Zahl im Intervall [0, 170]: " + args[0]);
      Console.Read();
   }
}
```

Ein Programm sollte sich generell bemühen, Ausnahmefehler nach Möglichkeit durch Kontrollmaßnahmen (z. B. Parametervalidierung) zu verhindern. Im Beispiel überprüft die Methode **Main()** daher, ob tatsächlich ein Befehlszeilenargument in `args[0]` vorhanden ist, bevor diese **String**-Referenz beim Aufruf der Methode `Kon2Int()` als Parameter verwendet wird. Damit wird verhindert, dass es zu einer **IndexOutOfRangeException** kommt, wenn der Benutzer das Programm *ohne* Befehlszeilenargument startet.

In diesem Fall beendet **Main()** das Programm durch einen Aufruf der Methode **Environment.Exit()**, der als Aktualparameter ein **Exitcode** übergeben wird. Nach einem beendeten Programmeinsatz in einem per **cmd.exe** gestarteten Konsolenfenster befindet sich der **return**-Wert in der Pseudo-Umgebungsvariablen **errorlevel**, z. B.:

---

[1] Diese Aussage stimmt *nicht* bei der Multithreading-Programmierung mit der sogenannten *Task Parallel Library*. Im Abschnitt 17.4.6 folgen Empfehlungen für den Umgang mit unbehandelten Ausnahmen, die innerhalb einer sogenannten *Aufgabe* auftreten.

```
>fakul
Kein Argument angegeben

>echo %errorlevel%
1
```

Nach einem störungsfrei verlaufenen Programmeinsatz enthält **errorlevel** den Exitcode 0.

Eine WPF-Anwendung sollte übrigens *nicht* durch den „aggressiven" Methodenaufruf **Environment.Exit()** beendet werden. Um eine WPF-Anwendung per Programm zu beenden, eignet sich die **Application**-Methode **Shutdown()**.

Die Reaktion des Beispielprogramms auf einen Start ohne Befehlszeilenargument kann als akzeptabel gelten:

- An Stelle einer für viele Benutzer wenig hilfreichen Ausnahmemeldung durch das Laufzeitsystem
  ```
  Unbehandelte Ausnahme: System.IndexOutOfRangeException: Der Index war
  außerhalb des Arraybereichs.
     bei Fakul.Main(String[] args) in U:\ ... \Fakul.cs:Zeile 18.
  ```
  erscheint eine verständliche Erläuterung des Problems:
  ```
  Kein Argument angegeben
  ```
- Falls das Programm von einem anderen Programm (z. B. von einem Kommandostapel) gestartet worden ist, steht dem Aufrufer ein Exitcode zur Verfügung.

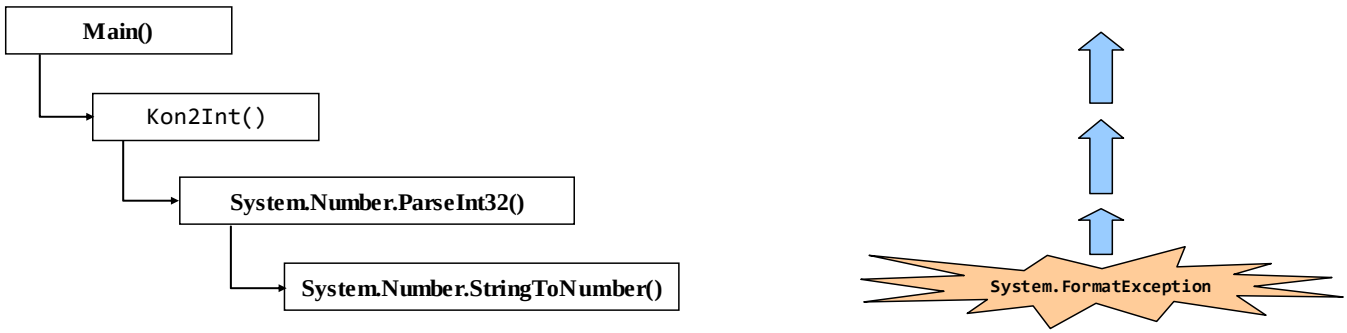
Die Methode `Kon2Int()` überprüft, ob die aus dem übergebenen **String**-Parameter ermittelte **int**-Zahl außerhalb des zulässigen Wertebereichs für eine Fakultätsberechnung (mit **double**-Ergebniswert) liegt, und meldet ggf. den Wert -1 als Fehlerindikator zurück. **Main()** kennt die spezielle Bedeutung dieser Rückgabe, sodass die unsinnige Fakultätsberechnung für ein negatives Argument und der wenig hilfreiche Ergebniswert Unendlich für ein Argument > 170 vermieden werden. Diese traditionelle Fehlerbehandlung per **Rückgabewert** (engl.: *return code*) ist *nicht* grundsätzlich als überholt und ineffizient zu bezeichnen, aber in vielen Situationen doch der gleich vorzustellenden Kommunikation über Ausnahmeobjekte unterlegen (siehe Abschnitt 13.3 zum Vergleich von Fehlerrückmeldung und Ausnahmebehandlung).

Trotz seiner präventiven Bemühungen ist das Programm leicht aus dem Tritt zu bringen, indem man es mit einer nicht konvertierbaren Zeichenfolge füttert (z. B. „vier"). Die zunächst betroffene, BCL-intern aufgerufene Methode **Number.StringToNumber()** wirft daraufhin eine **FormatException**. Diese wird vom Laufzeitsystem entlang der Aufrufsequenz an alle beteiligten Methoden bis hinauf zu **Main()** gemeldet:[1]

---

[1] Offenbar hat der Compiler den Aufruf der statischen **Int32**-Methode **Parse()** ersetzt durch einen Aufruf der statischen **Number**-Methode **ParseInt32()**. Wie dem Quellcode von **Int32** in der BCL von .NET 5.0 zu entnehmen ist, beschränkt sich **Int32.Parse()** beim Aufruf mit einem von **null** verschiedenen Parameter darauf, die Methode **Number.ParseInt32()** aufzurufen:

```
public static int Parse(string s) {
  if (s == null) ThrowHelper.ThrowArgumentNullException(ExceptionArgument.s);
  return Number.ParseInt32(s, NumberStyles.Integer, NumberFormatInfo.CurrentInfo);
}
```



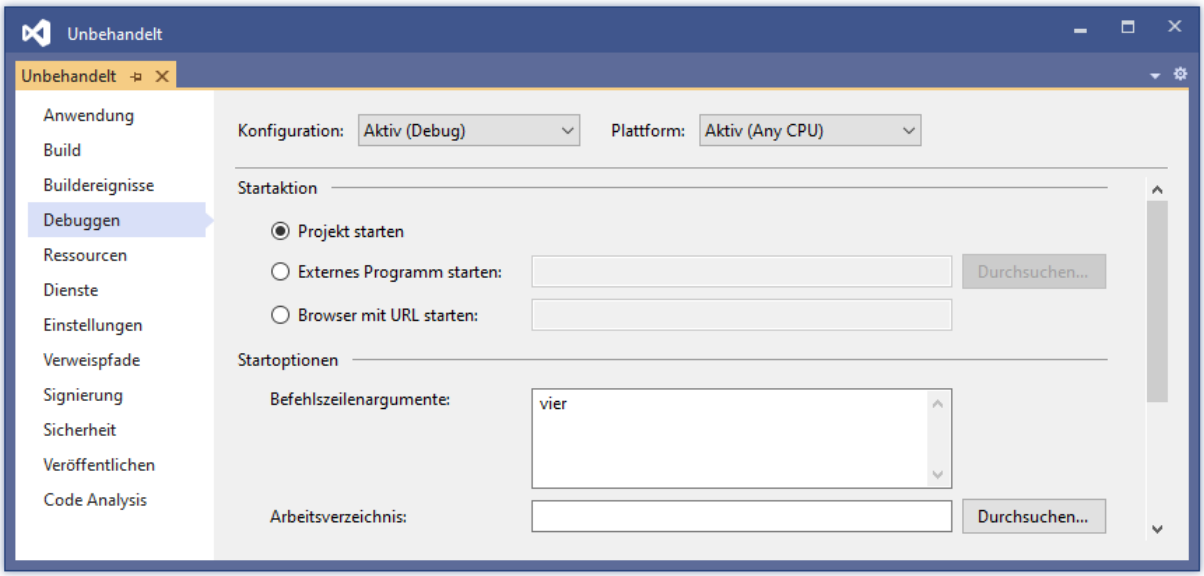
Weil kein Aufrufer eine geeignete Behandlungsroutine bereithält, endet das Programm mit einer Fehlermeldung durch das Laufzeitsystem:

```
Unbehandelte Ausnahme: System.FormatException: Die Eingabezeichenfolge hat das
falsche Format.
   bei System.Number.StringToNumber(String str, ..., Boolean parseDecimal)
   bei System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
   bei Fakul.Kon2Int(String s) in U:\Eigene Dateien\ ... \Fakul.cs:Zeile 5.
   bei Fakul.Main(String[] args) in U:\Eigene Dateien\ ... \Fakul.cs:Zeile 18.
```

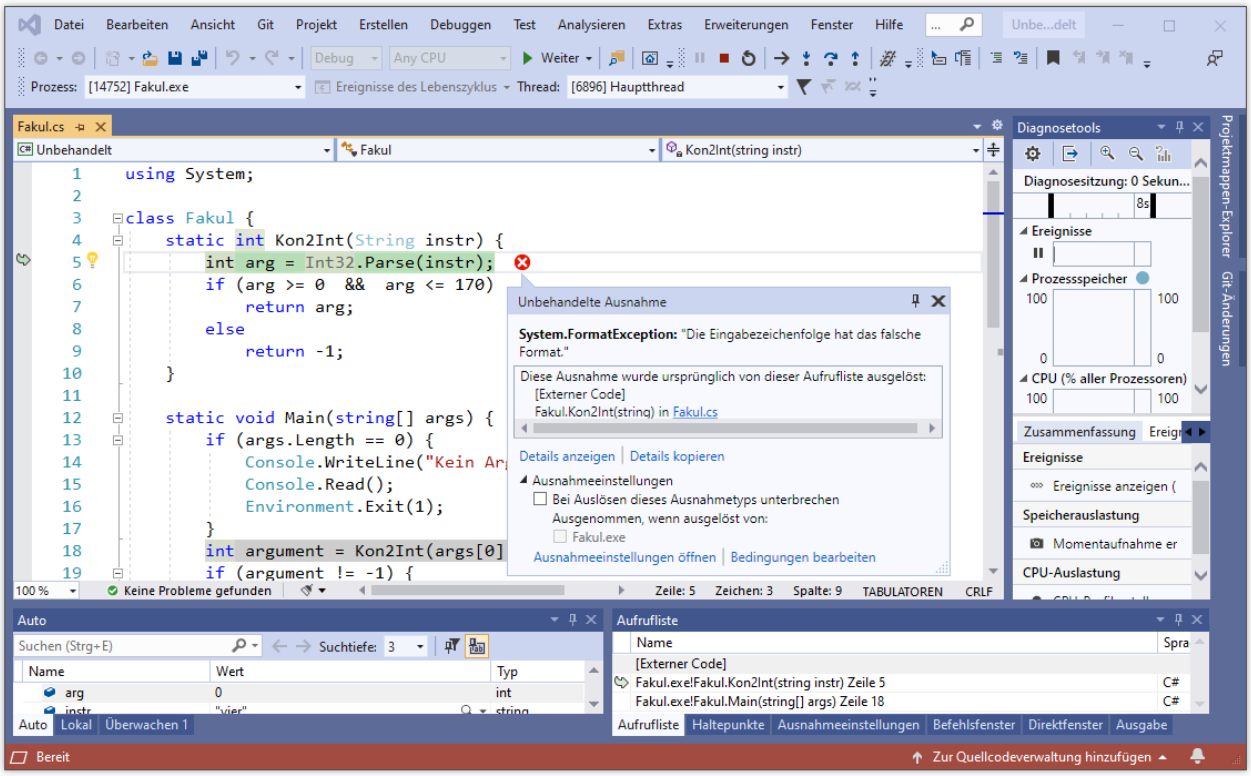
Um die Reaktion des Programms auf ein (fehlerhaftes) Befehlszeilenargument zu beobachten, muss man es übrigens nicht außerhalb der Entwicklungsumgebung in einem Konsolenfenster starten. Nach dem Menübefehl

Projekt > Eigenschaften

kann man auf der Registerkarte **Debuggen** die zu simulierenden **Befehlszeilenargumente** eintragen, z. B.:



Wurde die mit einer unbehandelten Ausnahme endende Anwendung aus dem Visual Studio (z. B. mit **F5**) im **Debug-Modus** gestartet, dann informiert die Entwicklungsumgebung über den Typ der Ausnahme und die Unfallstelle:



Um am Projekt weiterarbeiten zu können, muss der Debug-Modus beendet werden (z. B. mit dem Symbolschalter ■ oder mit der Tastenkombination **Umschalt+F5**).

13.2 Ausnahmen abfangen

Die Startversion des Programms zur Fakultätsberechnung beherrscht weder das Behandeln noch das Werfen von Ausnahmen. Wir machen uns nun daran, diese kommunikativen Kompetenzen nachzurüsten.

13.2.1 Die try - Anweisung

In C# wird die Behandlung von Ausnahmen über die **try** - Anweisung unterstützt:

```
try {
   Überwachter Block mit Anweisungen für den regulären Ablauf
}
catch (Ausnahmeklasse1 Parametername) {
   Anweisungen für die Behandlung von Ausnahmen aus der ersten Ausnahmeklasse
   oder aus einer daraus abgeleiteten Klasse
}
// Optional können weitere Ausnahmen abgefangen werden:
catch (Ausnahmeklasse2 Parametername) {
   Anweisungen für die Behandlung von Ausnahmen der zweiten Ausnahmeklasse
   oder aus einer daraus abgeleiteten Klasse
}
  . . .
// Optionaler finally-Block mit Abschluss- bzw. Bereinigungsarbeiten.
// Bei vorhandenem finally-Block, ist kein catch-Block erforderlich.
finally {
   Anweisungen, die unabhängig vom Auftreten einer Ausnahme ausgeführt werden sollen
}
```

Die Anweisungen für den ungestörten Ablauf setzt man in den **try**-Block. Nachdem eine Anweisung des **try**-Blocks eine Ausnahme verursacht oder aktiv geworfen hat, werden die weiteren Anweisungen des **try**-Blocks *nicht* mehr ausgeführt. In einem **try**-Block sollten Anweisungen stehen, die allesamt erfolgreich ausgeführt werden müssen, damit ein sinnvolles Ergebnis entsteht.

Treten bei der Ausführung dieses überwachten Blocks *keine* Fehler auf, wird das Programm hinter der **try**-Anweisung fortgesetzt, wobei ggf. vorher noch der **finally**-Block ausgeführt wird.

C# erlaubt folgende Varianten der **try** - Anweisung:

- **try-catch** - Anweisung
- **try-finally** - Anweisung
- **try-catch-finally** - Anweisung

Ein **try**-, **catch**- oder **finally**-Block benötigt auch dann ein einrahmendes Paar geschweifter Klammern, wenn nur *eine* Anweisung enthalten ist.

Weil es der obigen Syntaxbeschreibung im Quellcodedesign trotz Unterstützung durch Kommentare an Präzision fehlt, sollen Sie in einer Übungsaufgabe ein Syntaxdiagramm erstellen (siehe Abschnitt 13.7).

13.2.1.1 Ausnahmebehandlung per catch-Block

Ein **catch**-Block wird auch als *Exception-Handler* bezeichnet und besitzt im Kopfbereich eine einelementige Parameterliste. Anders als bei einer Methode kann sich die Parameterliste eines **catch**-Blocks auf die Typangabe beschränken oder ganz fehlen (siehe unten).

Tritt im **try**-Block eine Ausnahme auf, wird seine Ausführung abgebrochen. Anschließend sucht das Laufzeitsystem nach einem **catch**-Block, dessen Formalparameter den Typ der zu behandelnden Ausnahme oder einen Basistyp besitzt und führt dann den zugehörigen Anweisungsblock aus. Weil die Liste der **catch**-Blöcke von oben nach unten durchsucht wird, müssen Ausnahme*basis*klassen stets *unter* abgeleiteten Klassen stehen. Freundlicherweise stellt der Compiler die Einhaltung dieser Regel sicher. Von einer **try** - Anweisung wird maximal *ein* **catch**-Block ausgeführt. Weitere Details zum Programmablauf bei der Ausnahmebehandlung folgen im Abschnitt 13.2.2.

Nun zu den angekündigten Möglichkeiten, den Kopf eines **catch**-Blocks zu vereinfachen:

- Man kann auf die Angabe eines Formalparameter*namens* verzichten, hat dann aber im **catch**-Block kein Ausnahmeobjekt (mit Unfallbericht, siehe unten) zur Verfügung:

  catch (*Ausnahmeklasse***) {**
  Anweisungen für die Behandlung der Ausnahmeklasse
  }

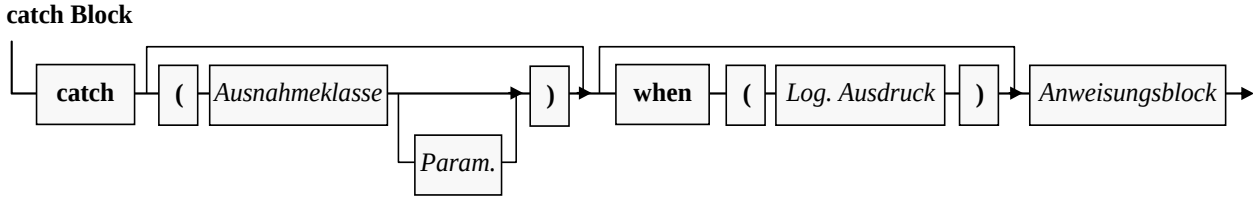
- Fehlt bei einem **catch**-Block die Parameterliste komplett, ist die (maximal breite) Ausnahme-Basisklasse **Exception** eingestellt, was offensichtlich nur beim *letzten* **catch**-Block sinnvoll ist:

  catch {
  Anweisungen für die Behandlung der Ausnahmeklasse ***Exception***
  }

  Die Behandlung der Ausnahmeklasse **Exception** kommt z. B. dann in Frage, wenn im **catch**-Block lediglich ein Eintrag in eine Protokolldatei geschrieben und dann die abgefangene Ausnahme erneut geworfen werden soll, was durch die folgende **throw**-Anweisung geschehen kann (siehe Abschnitt 13.5):

  ```
  throw;
  ```

Seit C# 6.0 kann der Zuständigkeitsbereich eines **catch**-Blocks durch eine **when**-Klausel eingeschränkt werden:



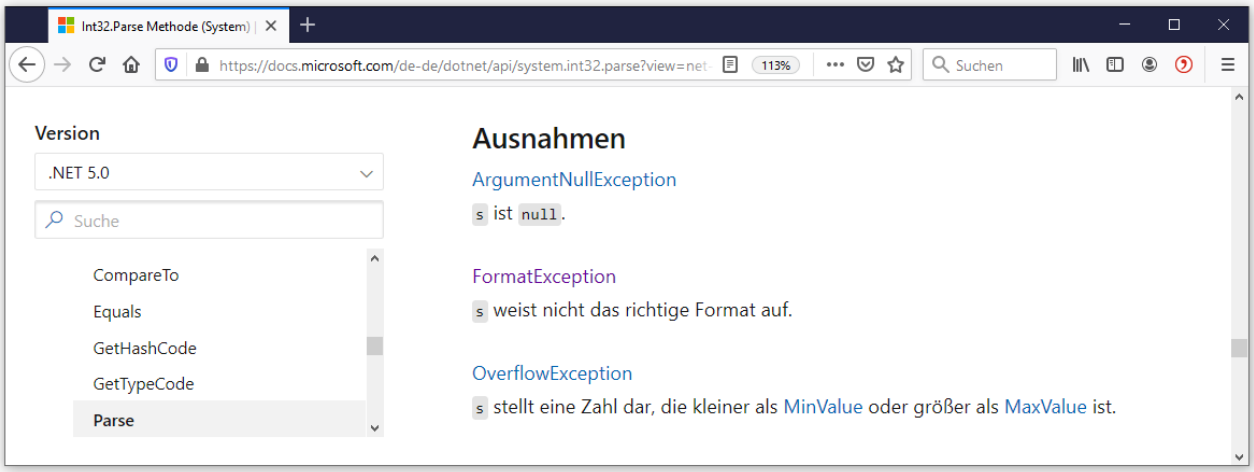
Der **catch**-Block wird nur dann ausgeführt, wenn ...

- eine Ausnahme zu seiner Klasse oder zu einer abgeleiteten Klasse gehört,
- *und* der logische Ausdruck den Wert **true** besitzt.

Hinsichtlich der Sequenz von **catch**-Blöcken gilt bei Verwendung von **when**-Klauseln (kurz: Filterbedingungen):

- Es sind mehrere **catch**-Blöcke mit derselben Ausnahmeklasse erlaubt, sofern sich die Filterbedingungen unterscheiden.
- Von mehreren **catch**-Blöcken mit derselben Ausnahmeklasse wird nur der erste zutreffende ausgeführt.
- Befindet sich unter mehreren **catch**-Blöcken mit derselben Ausnahmeklasse ein Block *ohne* Filterbedingung, dann muss dieser **catch**-Block zuletzt aufgeführt werden, weil er den größten Zuständigkeitsbereich besitzt.

Welche Ausnahmen von den Methoden eines BCL-Typs zu erwarten sind, erfährt man in der Dokumentation, z. B. bei der Methode **Int32.Parse(String)**:



Neben der bereits besprochenen **System.FormatException** (Zeichenfolge nicht konvertierbar) sind bei **Int32.Parse(String)** auch Ausnahmen aus den Klassen **System.ArgumentNullException** (keine Zeichenfolge vorhanden) und **System.OverflowException** (Konvertierungsergebnis kann nicht in einer **int**-Variablen abgelegt werden) möglich.

In der folgenden Variante der Methode `Kon2Int()` werden die von **Int32.Parse()** zu erwartenden Ausnahmen abgefangen. Eine **ArgumentNullException** kann im Beispielprogramm ausgeschlossen werden, weil die Methode **Main()** eine entsprechende Kontrolle vornimmt. Nach einer **OverflowException** liefert `Kon2Int()` den Wert -1 zurück (wie bei einem **int**-Wert außerhalb von [0, 170]):

```
static int Kon2Int(ref string instr) {
   int arg;
    try {
        arg = Int32.Parse(instr);
    } catch (OverflowException) {
        return -1;
    } catch (FormatException) when (Double.TryParse(instr, out double d)) {
        arg = (int)d;
        if (arg == d)
            instr = arg.ToString();
        else
            return -2;
    } catch (FormatException) {
        return -3;
    }
    if (arg >= 0 && arg <= 170)
     return arg;
   else
     return -1;
}
```

Im ersten **FormatException** - Handler wird die Filterung durch eine **when**-Klausel demonstriert. Wenn die eingegebene Zeichenfolge zwar nicht als ganze Zahl, aber als Gleitkommazahl interpretierbar ist, überprüft `Kon2Int()`, ob sich diese Zahl verlustfrei in einen **int**-Wert wandeln lässt. Gelingt dies, ...

- dann wird der **int**-Wert bei der weiteren Verarbeitung verwendet
- und seine **String**-Repräsentation an den **ref**-Parameter mit der Kandidatenzeichenfolge übergeben, sodass der Aufrufer die korrigierte Zeichenfolge erhält.

Anderenfalls meldet `Kon2Int()` den Wert -2 zurück (z. B. beim Aufruf mit dem Aktualparameter „5,1“).

Zur Beurteilung der Interpretierbarkeit als **double**-Wert dient die statische **Double**-Methode **TryParse()**, die per **bool**-Rückgabewert über die Konvertierbarkeit berichtet.[1] Weil im Beispielprogramm nach einer misslungenen Ganzzahlinterpretation mit erheblicher Wahrscheinlichkeit auch die Interpretation als Gleitkommazahl scheitert, wird die per Ausnahmeobjekt kommunizierende Methode **Double.Parse()** vermieden. Der nach einer erfolgreichen Konvertierung von **TryParse()** als **out**-Parameter gelieferte Wert landet in einer inline deklarierten Variablen (siehe Abschnitt 5.3.1.3.2.2).

Der zweite **FormatException**-Handler kommt zum Einsatz, wenn keine numerische Interpretation der Eingabe möglich ist. In diesem Fall landet der Wert -3 beim Aufrufer.

Die **catch**-Blöcke verwenden das von **Int32.Parse()** geworfene Ausnahmeobjekt *nicht* und verzichten daher auf einen Parameternamen.

Man kann sich fragen, ob `Kon2Int()` nicht auf den potentielle Ausnahmewerfer **Int32.Parse()** und damit auch auf die Ausnahmebehandlung per **try-catch** - Anweisung verzichten und stattdessen die mit Rückgabewert arbeitende Methode **Int32.TryParse()** verwenden sollte. Nach den Performanzüberlegungen im Abschnitt 13.3 wäre dies eine akzeptable Vorgehensweise. Trotzdem erfüllt das Beispiel in seiner jetzigen Form wohl die Aufgabe, die in vielen Situationen außerordentlich wichtige **try-catch** - Ausnahmebehandlung zu demonstrieren.

Je nach Algorithmus kommen als Aufgaben für einen **catch**-Block in Frage (selbstverständlich auch in Kombination):

- Reparatur
  Manchmal ist es möglich, den aufgetretenen Fehler zu beheben oder zu kompensieren (z. B. zu umgehen). In der Methode `Kon2Int()` unternimmt der erste **FormatException** - Handler einen Reparaturversuch.
- Rückabwicklung
  Bereits realisierte und aufgrund der Ausnahme nunmehr unerwünschte Effekte des unterbrochenen **try**-Blocks sollten nach Möglichkeit wieder rückgängig gemacht werden (z.B. durch ein Rollback nach einer gescheiterten Datenbank-Transaktion).
- Ersetzung der Ausnahme durch eine informativere Alternative
  Viele **catch**-Blöcke betätigen sich als Informationsvermittler und werfen selbst eine Ausnahme, um dem Aufrufer einen leichter verständlichen Unfallbericht zu liefern (siehe Abschnitt 13.5).
- Fehlermeldung und Fehlerprotokollierung
  Wenn eine gescheiterte Operation abgebrochen werden muss, sollte der Benutzer eine gut verständliche Fehlermeldung erhalten. Ein Eintrag in eine Logdatei kann den Software-Entwickler oder einen Administrator dabei unterstützen, die Ursache des Fehlers zu finden (siehe Kapitel 16 zur Dateiausgabe). Nach einer Fehlermeldung oder -protokollierung ist es in der Regel sinnvoll, die abgefangene Ausnahme erneut zu werfen, was durch die folgende **throw**-Anweisung geschehen kann (siehe Abschnitt 13.5):

  ```
  throw;
  ```

In der **Main()** - Methode des Beispielprogramms ist die differenziertere Fehlerrückmeldung zu berücksichtigen:

---

[1] **Double.TryParse()** interpretiert z. B. die Zeichenfolge „5...0“ als 50, weil die Punkte als Gruppentrennzeichen aufgefasst werden. Durch eine Überladung mit **NumberStyles**-Parameter lässt sich die Interpretation beeinflussen.

```
static void Main(string[] args) {
   if (args.Length == 0) {
      Console.WriteLine("Kein Argument angegeben");
      Console.Read();
      Environment.Exit(1);
   }
   int argument = Kon2Int(ref args[0]);
   switch (argument) {
      case int arg when arg >= 0:
         double fakul = 1.0;
         for (int i = 1; i <= argument; i++)
            fakul = fakul * i;
         Console.WriteLine("Fakultät von {0}: {1}", args[0], fakul);
         break;
      case -1:
         Console.WriteLine("Keine ganze Zahl im Intervall [0, 170]: " + args[0]);
         break;
      case -2:
         Console.WriteLine("Die Eingabe ist keine ganze Zahl: " + args[0]);
         break;
      case -3:
         Console.WriteLine("Die Eingabe ist nicht numerisch interpretierbar: " + args[0]);
         break;
   }
}
```

In der ersten **case**-Klausel wird die im Abschnitt 4.7.2.3.3 beschriebene Deklaration mit Typmuster verwendet.

Ein Programmstart mit dem Befehlszeilenargument „vier“ führt z. B. zur Fehlermeldung:

```
Die Eingabe ist nicht numerisch interpretierbar: vier
```

Die Methode `Kon2Int()` beherrscht nun das Abfangen von Ausnahmen zur Analyse und Behandlung von Problemen, verwendet aber zur Fehlersignalisierung an ihren Aufrufer die traditionelle Technik per Rückgabewert. Diese Mischung ist nicht grundsätzlich fehlerhaft oder ineffektiv, wird sich aber im weiteren Verlauf des Kapitels 13 noch ändern, nachdem wir uns mit dem Werfen von Ausnahmen beschäftigt haben. Wie man an der Koexistenz der Methoden **Parse()** und **TryParse()** in der BCL-Struktur **Int32** sieht, haben beide Verfahren zur Fehlersignalisierung ihre Berechtigung.

13.2.1.2 finally

Der **finally**-Block einer **try**-Anweisung wird unter fast allen Umständen ausgeführt:

- Nach der ungestörten Ausführung des **try**-Blocks
- Nach einer Ausnahmebehandlung in einem **catch**-Block (auch beim Verlassen des **catch**-Blocks durch eine neue Ausnahme)
- Nach dem Auftreten einer unbehandelten Ausnahme im **try**-Block
- Beim Verlassen der **try**-Anweisung durch eine **goto**-Anweisung (siehe Abschnitt 4.7.3.5) im **try**-Block oder in einem **catch**-Block
- Beim Beenden der Methode durch eine **return**-Anweisung im **try**-Block oder in einem **catch**-Block

Nur zwei Ursachen können die Ausführung eines **finally**-Blocks verhindern (Albahari & Johannsen 2020, S. 174):

- Der **try**-Block oder ein **catch**-Block hängt in einer Endlosschleife.
- Im **try**-Block oder in einem **catch**-Block wird die statische Methode **Exit()** der Klasse **Environment** aufgerufen und somit der komplette Prozess terminiert.

Damit ist ein **finally**-Block der ideale Ort für Anweisungen, die wenn irgend möglich ausgeführt werden sollen, z. B. zur Freigabe von Ressourcen wie Datei-, Datenbank- und Netzwerkverbindungen. Wir verwenden (dem Kapitel 16 über Dateibearbeitung vorgreifend) zur **finally**-Demonstration eine statische Methode, die aus einer Textdatei pro Zeile eine **double**-Zahl zu lesen versucht, um den Mittelwert aus den vorhandenen Zahlen zu berechnen:

```
using System;
using System.IO;

class FinallyDemo {
   static void Mean(String dateiname) {
      FileStream fs = null;
      try {
         fs = new FileStream(dateiname, FileMode.Open);
         String s;
         int n = 0;
         double summe = 0.0;
         StreamReader sr = new StreamReader(fs);
         while ((s = sr.ReadLine()) != null) {
            summe += Convert.ToDouble(s);
            n++;
         }
         Console.WriteLine($"Deskriptive Statistiken zur Datei {dateiname}\n");
         Console.WriteLine($"Anzahl:\t{n}");
         Console.WriteLine($"Summe:\t{summe}");
         Console.WriteLine($"Mittel:\t{summe / n}");
      } catch (Exception ex) {
         Console.WriteLine($"Fehler in Mean() beim Lesen der Datei {dateiname}\n{ex}\n");
         throw;
      } finally {
         if (fs != null)
            fs.Dispose();
      }
   }

   static void Main() {
      try {
         Mean(AppDomain.CurrentDomain.BaseDirectory + "daten.txt");
      } catch {
         Console.WriteLine("Fehler bei der Mittelwertsberechnung");
      }
   }
}
```

Das Ergebnis eines erfolgreichen Aufrufs:

```
Deskriptive Statistiken zur Datei E:\Daten\ ... \daten.txt

Anzahl: 18
Summe:  90
Mittel: 5
```

Vom **FileStream** - Konstruktor, der eine vorhandene Datei öffnen soll, sind diverse Ausnahmen zu erwarten, u. a.:

- **System.IO.FileNotFoundException**
  Die Datei existiert nicht.
- **System.IO.IOException**
  Diese Ausnahme tritt z. B. auf, wenn ein anderer Prozess die Datei durch sein exklusives Zugriffsrecht blockiert.

Im Beispiel kann *nach* dem erfolgreichen Öffnen der Datei ein Ausnahmefehler auftreten, wenn die Methode **Convert.ToDouble()** auf eine nicht konvertierbare Zeichenfolge trifft.

Eine geöffnete Datei sollte möglichst früh per **Dispose()** - Aufruf geschlossen werden, um andere Programme möglichst wenig zu behindern.[1] Das muss auch für den Ausnahmefall sichergestellt werden, indem das Schließen in einem **finally**-Block erfolgt. Stünde der **Dispose()** - Aufruf z. B. am Ende des **try**-Blocks, bliebe nach einem Ausnahmefehler in **Convert.ToDouble()** die Datei geöffnet bis zum Programmende. Für andere Programme wäre die Datei nicht verwendbar, z. B.:

Der **catch**-Block des Beispielprogramms schreibt eine Fehlermeldung auf die Konsole, z. B.:

```
Fehler in Mean() beim Lesen der Datei E:\Daten\ ... \daten.txt
System.FormatException: Die Eingabezeichenfolge hat das falsche Format.
   bei System.Number.ParseDouble(String value, NumberStyles options, NumberFormatInfo numfmt)
   bei System.Convert.ToDouble(String value)
   bei FinallyDemo.Mean(String dateiname) in E:\Daten\ ... \FinallyDemo.cs:Zeile 14.
```

Außerdem wirft er per **throw**-Anweisung (siehe Abschnitt 13.5) dieselbe Ausnahme erneut, sodass die Methode `Mean()` beendet und der Aufrufer über das Scheitern informiert wird.

Wie im Kapitel 16 über Dateibearbeitung noch im Detail erläutert wird, lässt sich durch eine **using**-Anweisung, die strikt von der **using**-Direktive zu unterscheiden ist, das garantierte Schließen einer vom Programm geöffneten Datei einfacher erreichen, wobei der Compiler einen **finally**-Block mit **Dispose()** - Aufruf automatisch erstellt:

```
static void Mean(String dateiname) {
    using (FileStream fs = new FileStream(dateiname, FileMode.Open)) {
        String s;
        int n = 0;
        double summe = 0.0;
        StreamReader sr = new StreamReader(fs);
        while ((s = sr.ReadLine()) != null) {
            summe += Convert.ToDouble(s);
            n++;
        }
        Console.WriteLine($"Deskriptive Statistiken zur Datei {dateiname}\n");
        Console.WriteLine($"Anzahl:\t{n}");
        Console.WriteLine($"Summe:\t{summe}");
        Console.WriteLine($"Mittel:\t{summe / n}");
    }
}
```

[1] Statt **Dispose()** kann auch die funktionsgleiche Methode **Close()** aufgerufen werden (Albahari & Johannsen 2020, S. 642).

Ausnahmen werden allerdings von der **using**-Anweisung *nicht* behandelt, sondern an den Aufrufer weitergeleitet. Außerdem ist es nicht immer erwünscht, dass ein Ressourcen nutzendes Objekt am Ende des Blocks, in dem es erstellt wurde, verworfen wird.[1] Daher ist die explizite **try-catch-finally** - Anweisung gelegentlich gegenüber der **using**-Anweisung zu bevorzugen.

Eine im **finally**-Block aufgetretene Ausnahme wird an den Vorgänger in der Aufrufverschachtelung übermittelt, wobei es sich auch um eine umgebende **try**-Anweisung handeln kann. Dabei wird eine im **try**-Block oder in einem **catch**-Block geworfene Ausnahme ersetzt, was die Fehleranalyse erschweren kann, z. B.:

Quellcode	Ausgabe

`using System;`
 	
`class Prog {`
`  static void M(int i) {`
`  try {`
`  if (i < 0)`
`  throw new ArgumentException();`
`  else`
`  Console.WriteLine(i);`
`  } finally {`
`  throw new Exception();`
`  }`
`  }`
 	
`  static void Main() {`
`  try {`
`  M(-1);`
`  } catch (Exception e) {`
`  Console.WriteLine(e.GetType());`
`  }`
`  }`
`}`
	`System.Exception`

13.2.2 Programmablauf bei der Ausnahmebehandlung

Findet das Laufzeitsystem für eine Ausnahme in der verantwortlichen Methode keinen zuständigen **catch**-Block, dann sucht es entlang der Aufrufersequenz weiter. Dies macht es leicht, die Behandlung einer Ausnahme der bestgerüsteten Methode zu überlassen.

Initiator der Ausnahme kann sein:

- die betroffene Methode selbst,
- eine aufgerufene Methode, welche die Ausnahme geworfen, aber nicht behandelt hat,
- die CLR (z. B. bei einer Ganzzahldivision durch 0).

13.2.2.1 Beispiel

Das folgende Beispiel demonstriert einige Programmabläufe aufgrund von Ausnahmen, die auf verschiedenen Stufen einer Aufrufhierarchie geworfen bzw. behandelt werden. Um das Beispiel einfach zu halten, wird auf Praxisnähe verzichtet. Das Programm nimmt via Kommandozeile ein Argument entgegen, interpretiert es numerisch und ermittelt den Rest aus der Division der Zahl 10 durch das Argument:

---

[1] https://docs.microsoft.com/de-de/dotnet/standard/garbage-collection/using-objects

```
using System;
class Sequenzen {
   static int Calc(String instr) {
      int erg = 0;
      try {
         Console.WriteLine("try-Block von Calc()");
         erg = 10 % Convert.ToInt32(instr);
      } catch (FormatException) {
         Console.WriteLine("FormatException-Handler in Calc()");
      } finally {
         Console.WriteLine("finally-Block von Calc()");
      }
      Console.WriteLine("Nach try-Anweisung in Calc()");
      return erg;
   }

   static void Main(string[] args) {
      try {
         Console.WriteLine("try-Block von Main()");
         Console.WriteLine("10 % " + args[0] + " = " + Calc(args[0]));
      } catch (ArithmeticException) {
         Console.WriteLine("ArithmeticException-Handler in Main()");
      } finally {
         Console.WriteLine("finally-Block von Main()");
      }
      Console.WriteLine("Nach try-Anweisung in Main()");
   }
}
```

Die Methode **Main()** lässt die eigentliche Arbeit von der Methode `Calc()` erledigen und bettet den `Calc()` - Aufruf in eine **try-catch-finally** - Anweisung mit **catch**-Block für die **ArithmeticException** ein, die z. B. bei einer Ganzzahldivision durch 0 von der CLR geworfen wird. `Calc()` benutzt die Klassenmethode **Convert.ToInt32()** sowie den Modulo-Operator im **try**-Block einer **try-catch-finally** - Anweisung, wobei nur die potentiell von **Convert.ToInt32()** zu erwartende **FormatException** in einem **catch**-Block abgefangen wird.

Wir betrachten einige Konstellationen mit ihren Konsequenzen für den Programmablauf:

a) Normaler Ablauf
b) Exception in `Calc()`, die dort auch behandelt wird
c) Exception in `Calc()`, die in **Main()** behandelt wird
d) Exception in **Main()**, die nirgends behandelt wird

a) Normaler Ablauf

Beim Programmablauf *ohne* Ausnahmen (hier mit Befehlszeilenargument „8“) kommt es zu folgenden Ausgaben:

```
try-Block von Main()
try-Block von Calc()
finally-Block von Calc()
Nach try-Anweisung in Calc()
10 % 8 = 2
finally-Block von Main()
Nach try-Anweisung in Main()
```

b) Exception in `Calc()`, die dort auch behandelt wird

Wird `Calc()` bei der Ausführung der Anweisung

```
erg = 10 % Convert.ToInt32(instr);
```

mit einer **FormatException** konfrontiert, die z. B. wegen des Befehlszeilenarguments „acht“ von **Convert.ToInt32()** geworfen, aber nicht behandelt wurde, dann kommt der zugehörige **catch**-Block in `Calc()` zum Einsatz. Dann folgen:

- **finally**-Block in `Calc()`
- restliche Anweisungen in `Calc()`

Im **try**-Block von `Calc()` hinter dem Unfallort stehende Anweisungen werden *nicht* ausgeführt. So wird verhindert, dass ein Algorithmus mit fehlerhaften Zwischenergebnissen weiterläuft. Bei der traditionellen Fehlerbehandlung (siehe Abschnitt 13.3) kann das passieren und zu schwer aufklärbaren Fehlern führen.

An **Main()** wird keine Ausnahme gemeldet, also werden in dieser Methode nacheinander ausgeführt:

- **try**-Block
- **finally**-Block
- restliche Anweisungen

Insgesamt erhält man die folgenden Ausgaben:

```
try-Block von Main()
try-Block von Calc()
FormatException-Handler in Calc()
finally-Block von Calc()
Nach try-Anweisung in Calc()
10 % acht = 0
finally-Block von Main()
Nach try-Anweisung in Main()
```

Zu der unsinnigen Ausgabe

```
10 % acht = 0
```

kommt es, weil die **FormatException** in `Calc()` nicht *sinnvoll* behandelt wird. Wenn ein **catch**-Block lediglich eine Fehlermeldung ausgibt und/oder einen Logdateieintrag schreibt, sollte er in der Regel die gefangene Ausnahme unbedingt erneut werfen oder stattdessen eine informativere Ausnahme werfen. Das aktuelle Beispiel soll nur dazu dienen, Programmabläufe bei der Ausnahmebehandlung zu demonstrieren.

c) Exception in `Calc()`, die in Main() behandelt wird

Wird eine **ArithmeticException** an `Calc()` gemeldet (z. B. wegen Befehlszeilenargument „0“), findet sich in der Methode kein passender Exception-Handler. Weil die Ausnahme im **try**-Block einer **try-catch-finally** - Anweisung auftrat, wird noch der zugehörige **finally**-Block ausgeführt, bevor die Methode verlassen wird, um entlang der Aufrufersequenz nach einem geeigneten Handler zu suchen.

Im `calc()` - Aufrufer **Main()** findet sich ein **ArithmeticException**-Handler, der nun zum Einsatz kommt. Dann geht es weiter mit dem zugehörigen **finally**-Block. Schließlich wird das Programm hinter der **try**-Anweisung der Methode **Main()** fortgesetzt. Insgesamt erhält man die folgenden Ausgaben:

```
try-Block von Main()
try-Block von Calc()
finally-Block von Calc()
ArithmeticException-Handler in Main()
finally-Block von Main()
Nach try-Anweisung in Main()
```

d) Exception in Main(), die nirgends behandelt wird

Übergibt der Benutzer kein Befehlszeilenargument, tritt in **Main()** beim Zugriff auf `args[0]` eine **IndexOutOfRangeException** auf (von der CLR geworfen). Weil sich kein zuständiger Handler findet, wird das Programm von der CLR beendet. Zuvor wird der **finally**-Block von **Main()** noch ausgeführt, die Anweisungen hinter der **try**-Anweisung aber nicht mehr:

```
try-Block von Main()

Unbehandelte Ausnahme: System.IndexOutOfRangeException: Der Index war außerhalb
des Arraybereichs.
   bei Sequenzen.Main(String[] args) in
      U:\Eigene Dateien\Sequenzen\Sequenzen.cs:Zeile 23.
finally-Block von Main()
```

13.2.2.2 Komplexe Fälle

Es ist oft erforderlich, **try**-Anweisungen zu schachteln, wobei innerhalb eines **try**-, **catch**- oder **finally**-Blocks wiederum eine komplette **try**-Anweisung stehen darf. Daraus ergeben sich weitere Ablaufvarianten für eine flexible Ausnahmebehandlung.

Wenn eine per Delegatenobjekt aufgerufene Methode wegen einer unbehandelten Ausnahme vorzeitig endet, dann werden ggf. in der Delegatenaufrufliste nachfolgende Methoden *nicht* aufgerufen.

13.2.3 Unbehandelte Ausnahmen in einer WPF-Anwendung abfangen

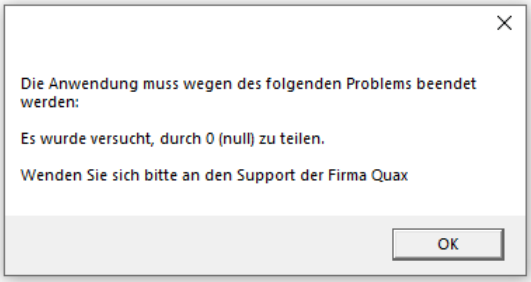
Durch eine Behandlungsmethode zum Ereignis **DispatcherUnhandledException** der Klasse **Application** kann eine WPF-Anwendung auf eine ansonsten unbehandelte Ausnahme reagieren. Wird die folgende Methode

```
private void App_DispatcherUnhandledException(object sender,
                System.Windows.Threading.DispatcherUnhandledExceptionEventArgs e) {
  MessageBox.Show("Die Anwendung muss wegen des folgenden Problems beendet werden:\n\n" +
                  e.Exception.Message +
                  "\n\nWenden Sie sich bitte an den Support der Firma Quax"); ;
  e.Handled = true;
  Application.Current.Shutdown();
}
```

beim **Application**-Ereignis **DispatcherUnhandledException** registriert, z. B. in der Datei **App.xaml**,

```
<Application x:Class="HandleUnhandledException.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:HandleUnhandledException"
             StartupUri="MainWindow.xaml"
             DispatcherUnhandledException="App_DispatcherUnhandledException">
    <Application.Resources>
    </Application.Resources>
</Application>
```

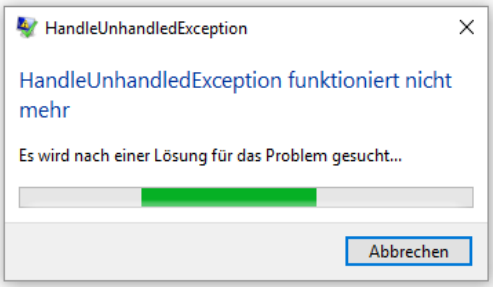
dann erscheint nach einer unbehandelten Ausnahme ein Info-Dialog,

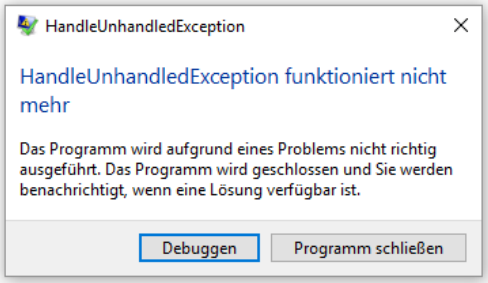


Über die Eigenschaft **Exception** der Klasse **DispatcherUnhandledExceptionEventArgs** ist die ursächliche Ausnahme ansprechbar, sodass die Benutzer informiert werden können. Im Beispiel wird nur die **Message**-Eigenschaft der ursächlichen Ausnahme angezeigt,

```
e.Exception.Message
```

weil der **StackTrace** sehr lang und damit eher für einen Protokolldateieintrag geeignet ist. Um die Windows-Standarddialoge für havarierte Anwendungen





zu verhindern, wird die zugrundeliegende Ausnahme als behandelt erklärt:

```
e.Handled = true;
```

Dann wird die Anwendung mit der **Application**-Methode **Shutdown()** beendet:

```
Application.Current.Shutdown();
```

Wenn es zu verantworten ist und vom Anwender gewünscht wird,

kann man die Ausnahme als erledigt deklarieren und die Anwendung fortsetzen:

```
private void App_DispatcherUnhandledException(object sender,
                System.Windows.Threading.DispatcherUnhandledExceptionEventArgs e) {
    e.Handled = true;
    if (MessageBox.Show("Es ist ein Problem aufgetreten:\n\n" +
          e.Exception.Message +
          "\n\nSoll die Anwendung trotzdem fortgesetzt werden?",
          "HandleUnhandledException", MessageBoxButton.YesNo) == MessageBoxResult.No)
        Application.Current.Shutdown();
}
```

13.3 Ausnahmeobjekte im Vergleich zur traditionellen Fehlerbehandlung

Die konventionelle Fehlerbehandlung verwendet meist den **Rückgabewert** einer Methode zur Berichterstattung über Probleme bei der Ausführung. Ein Rückgabewert kann …

- ausschließlich zur Fehlermeldung dienen
  Meist wird dann ein ganzzahliger **Returncode** mit dem Datentyp **int** verwendet, wobei die 0 einen erfolgreichen Ablauf signalisiert, während andere Zahlen jeweils für einen bestimmten Fehlertyp stehen. Soll nur zwischen Erfolg und Misserfolg unterschieden werden, bietet sich der Rückgabewert **bool** an.
- neben den Ergebnissen einer ungestörten Ausführung über spezielle Werte auch Problemfälle signalisieren (siehe Methode `Kon2Int()` des Beispielprogramms im Abschnitt 13.1 bzw. 13.2)

Wenn der Rückgabewert mit der Fehlersignalisierung komplett ausgelastet ist, kann zum Ergebnistransport ein **out**- oder **ref**-Parameter verwendet werden (siehe Abschnitt 5.3.1.3.2 zu den Verweisparametern). So verhält sich z. B. die statische Methode **TryParse()** der BCL-Struktur **Int32** (siehe unten).

Wenn eine Methode per Rückgabewert eine Nutzinformation (z. B. ein Berechnungsergebnis) übermitteln soll, und bei einer ungestörten Methodenausführung *jeder* Wert des Rückgabetyps auftreten kann, dann sind keine Werte als Fehlerindikatoren verfügbar. In dieser Situation arbeiten Programme mit klassischer Fehlersignalisierung gelegentlich mit einem (z. B. per get-only - Eigenschaft realisierten) Fehlerstatus als Kommunikationsmittel, wobei die Beachtung ebenso wenig garantiert ist wie bei einem Returncode.

Sollen z. B. drei Methoden, deren Rückgabewerte ausschließlich zur Fehlermeldung dienen, nacheinander aufgerufen werden, dann wird die vom Algorithmus diktierte simple Sequenz:

```
static void Main() {
   M1();
   M2();
   M3();
}
```

nach der Ergänzung der Fehlerbehandlungen zu einer länglichen und recht unübersichtlichen Konstruktion:

```
static void Main() {
    int returncode;
    returncode = M1();
    // Behandlung von potentiellen M1() - Fehlern
    if (returncode == 1) {
        // ...
        Environment.Exit(11);
    }
    ...
    returncode = M2();
    // Behandlung von potentiellen M2() - Fehlern
    if (returncode == 1) {
        // ...
        Environment.Exit(21);
    }
    ...
    returncode = M3();
    // Behandlung von potentiellen M3() - Fehlern
    if (returncode == 1) {
        // ...
        Environment.Exit(31);
    }
    ...
}
```

Mit Hilfe der Ausnahmetechnik bleibt hingegen im Kernalgorithmus die Übersichtlichkeit erhalten. Wir nehmen nun an, dass die drei Methoden `M1()`, `M2()` und `M3()` durch Ausnahmeobjekte über Fehler informieren:

```
static void Main() {
   try {
      M1();
      M2();
      M3();
   } catch (ExA a) {
      // Behandlung von Ausnahmen aus der Klasse ExA
   } catch (ExB b) {
      // Behandlung von Ausnahmen aus der Klasse ExB
   } catch (ExC c) {
      // Behandlung von Ausnahmen aus der Klasse ExC
   }
}
```

Es ist zu beachten, dass z. B. nach der Behandlung einer durch die Methode `M1()` verursachten Ausnahme die weiteren Anweisungen des überwachten **try**-Blocks nicht mehr ausgeführt werden.

Das traditionelle Verfahren der Fehlerrückmeldung hat neben dem unübersichtlichen Quellcode noch weitere Nachteile:

- Ungesicherte Beachtung von Rückgabewerten
  Gut gesetzte Rückgabewerte nützen nichts, wenn sich die Aufrufer nicht darum kümmern.
- Umständliche Weiterleitung von Fehlern
  Wenn ein Fehler nicht an Ort und Stelle behandelt werden soll, muss die Fehlerinformation aufwändig entlang der Aufrufersequenz nach oben gemeldet werden.
- Beschränkung auf Methoden
  Die Rückgabe eines Fehlerindikators ist keine Option bei Eigenschaften, Indexern, Ereignissen oder überladenen Operatoren.

Gegenüber der konventionellen Fehlerbehandlung hat die Kommunikation über Ausnahmeobjekte u. a. folgende Vorteile:

- Bessere Lesbarkeit des Quellcodes
  Mit Hilfe einer **try**-Anweisung erreicht man eine bessere Trennung zwischen den Anweisungen für den normalen Programmablauf und den diversen Ausnahmebehandlungen, sodass der Quellcode übersichtlich bleibt.
- Garantierte Beachtung von Ausnahmen
  Im Unterschied zu Rückgabewerten können Ausnahmen nicht ignoriert werden. Reagiert ein Programm nicht darauf, wird es vom Laufzeitsystem beendet.
- Automatische Weitermeldung bis zur bestgerüsteten Methode
  Oft ist der unmittelbare Verursacher nicht gut gerüstet zur Behandlung einer Ausnahme, z. B. nach dem vergeblichen Öffnen einer Datei. Dann sollte eine „höhere" Methode über das weitere Vorgehen entscheiden und z. B. beim Benutzer eine alternative Datei erfragen.
- Bessere Fehlerinformationen für den Aufrufer
  Über ein **Exception**-Objekt kann der Aufrufer beliebig genau über einen aufgetretenen Fehler informiert werden, was bei einem traditionellen Rückgabewert nicht der Fall ist.

Wie die Realisation des ersten **catch**-Blocks zur **FormatException** im Abschnitt 13.2.1 gezeigt hat (Aufruf von **TryParse()** statt **Parse()**), ist die Fehlermeldung per Ausnahmeobjekt dem traditionellen Rückgabewert nicht grundsätzlich überlegen. Bei der Entscheidung für eine Technik zur Fehlerkommunikation ist u. a. die Wahrscheinlichkeit für das Auftreten des Fehlers relevant:

- Wenn ein **Fehler mit erheblicher Wahrscheinlichkeit** auftritt, sollte eine routinemäßige, aktive Kontrolle stattfinden. Daher sollte eine Methode, die einen solchen Fehler zu melden hat, davon ausgehen, dass der Aufrufer mit dem Problem rechnet und per Rückgabewert kommunizieren. Über ein mit erheblicher Wahrscheinlichkeit auftretendes Problem per Ausnahmeobjekt zu informieren, wäre eine unangemessen aufwändige Kommunikationstechnik. Die Ausnahmebehandlung sollte wegen der hohen Kosten nicht zum Bestandteil der Programmablaufsteuerung werden.
- Bei **Fehlern mit geringer Wahrscheinlichkeit** haben jedoch häufige, meist überflüssige Kontrollen eine Leistungseinbuße zur Folge. Hier sollte man es besser auf eine Ausnahme ankommen lassen. Die hohen Kosten einer Ausnahme entstehen nur dann, wenn sie tatsächlich geworfen wird.

Eine gute Wahl des Verfahrens zur Fehlerkommunikation ist besonders dann relevant, wenn eine Bibliotheksmethode für einen größeren Nutzerkreis entstehen soll. Microsoft spricht in seinen *Design Guidelines* für diesen Fall eine unmissverständliche Empfehlung aus:[1]

> **X DO NOT** return error codes.
> Exceptions are the primary means of reporting errors in frameworks.

Wer eine Methode (z. B. aus einer BCL-Klasse) nutzt, muss das dort realisierte Verfahren zur Fehlerkommunikation kennen und sich darauf einstellen. Manchmal besteht die Wahl zwischen zwei Methoden, die sich nur beim Verfahren zur Fehlerkommunikation unterscheiden. So bietet die Struktur **Int32** *zwei* statische Methoden zur Wandlung einer Zeichenfolge in einen **int**-Wert an:

[1] https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/exception-throwing

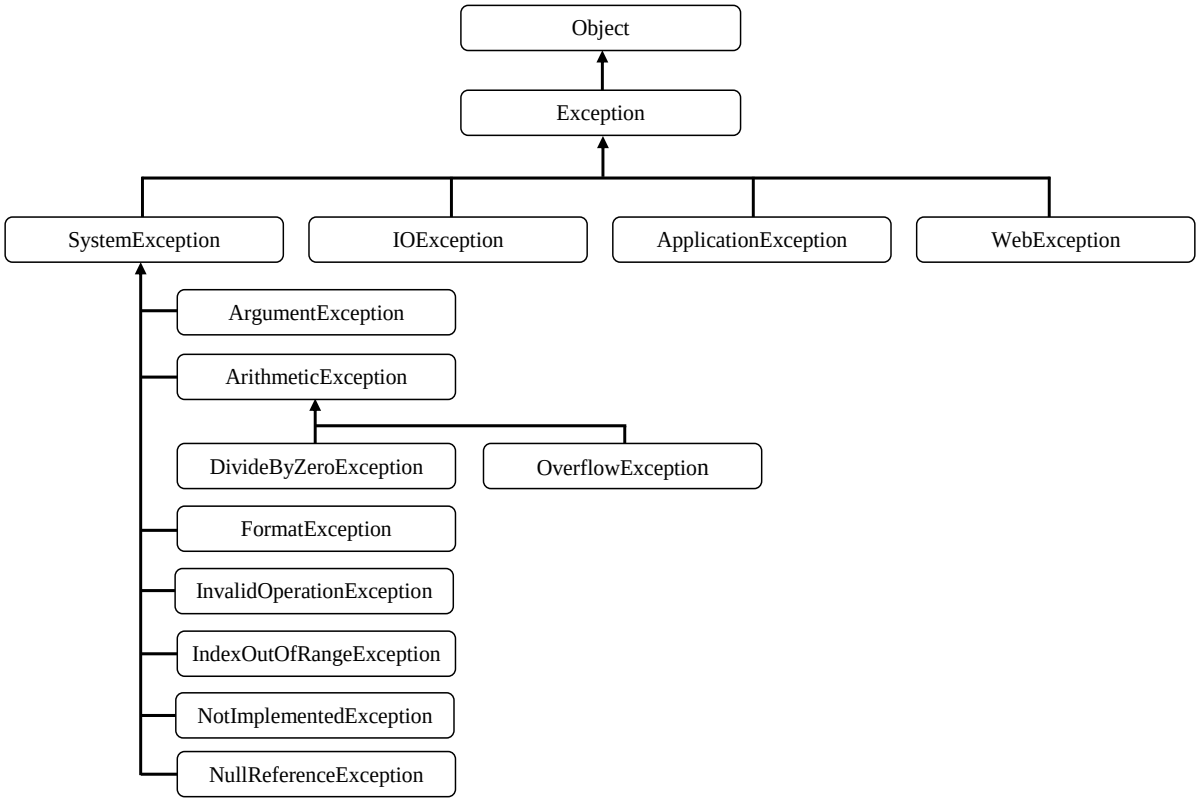
- **public static int Parse(String** *s***)**
  Diese Methode transportiert das Ergebnis einer erfolgreichen Wandlung per Rückgabewert und reagiert auf ein ungeeignetes Argument mit einem Ausnahmeobjekt:

Bedingung	**Ausnahmeklasse**
s ist gleich **null**	**ArgumentNullException**
s ist nicht konvertierbar	**FormatException**
Das Konvertierungsergebnis liegt nicht im **int**-Wertebereich	**OverflowException**

- **public static bool TryParse(String** *s***, out int** *result***)**
  Diese Methode benutzt den Rückgabewert als Fehlerindikator und transportiert das Ergebnis per **out**-Parameter. Scheitert die Konvertierung, erhält der **out**-Parameter den Wert 0. Die Rückgabe **false** zu ignorieren und den Wert 0 wie das Ergebnis einer erfolgreichen Wandlung zu verarbeiten, ist ein gravierender Programmierfehler.

13.4 Ausnahmeklassen in der BCL

Das .NET - Framework kennt zahlreiche vordefinierte Ausnahmeklassen, die mit ihren Vererbungsbeziehungen eine Klassenhierarchie bilden, aus der die folgende Abbildung einen kleinen Ausschnitt zeigt:



In einem **catch**-Block können auch *mehrere* Ausnahmeklassen durch Wahl einer entsprechend breiten Basisklasse behandelt werden.

Schon in der Klasse **Exception** sind u. a. die folgenden Eigenschaften mit Detailinformationen zu einer Ausnahme definiert:

- **Message**
  Diese **String**-Eigenschaft enthält eine Fehlermeldung mit Angaben zur Ursache der Ausnahme. Der folgende Aufruf der statischen Methode **ToInt32()** in der BCL-Klasse **Convert**

  ```
  Convert.ToInt32("Drei")
  ```

  sorgt z. B. für eine **FormatException** mit der **Message**-Eigenschaft:

  ```
  Die Eingabezeichenfolge hat das falsche Format.
  ```

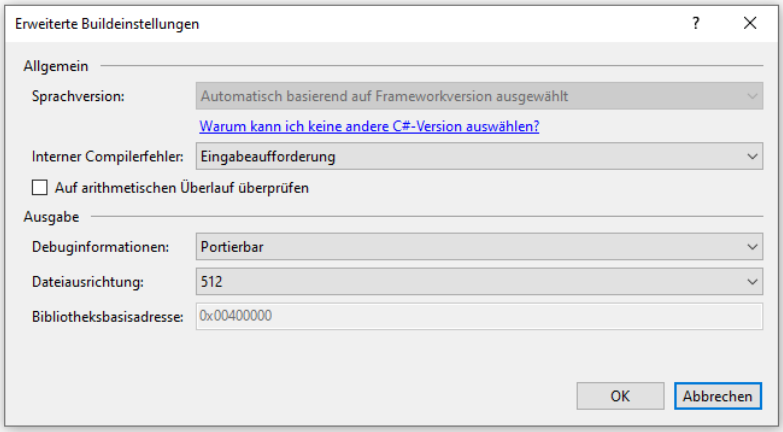
- **StackTrace**
  Diese **String**-Eigenschaft beschreibt den Aufrufstapel mit den beim Auftreten der Ausnahme aktiven Methoden. Im Abschnitt 13.1 war schon die **StackTrace**-Eigenschaft der **FormatException** zu sehen, die im Beispielprogramm zur Fakultätsberechnung aus einem irregulären Befehlszeilenargument resultiert:

  ```
  Unbehandelte Ausnahme: System.FormatException: Die Eingabezeichenfolge hat das
  falsche Format.
     bei System.Number.StringToNumber(String str, ..., Boolean parseDecimal)
     bei System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
     bei Fakul.Kon2Int(String s) in U:\Eigene Dateien\ ... \Fakul.cs:Zeile 5.
     bei Fakul.Main(String[] args) in U:\Eigene Dateien\ ... \Fakul.cs:Zeile 18.
  ```

  Dateinamen und Zeilennummern enthält die Aufrufreihenfolge übrigens nur dann, wenn für die betroffene Projekt-Erstellungskonfiguration der Entwicklungsumgebung (**Debug** oder **Release**) die **Debuginformationen** nicht abgeschaltet wurden. Im Visual Studio 2019 nimmt man die Einstellung nach

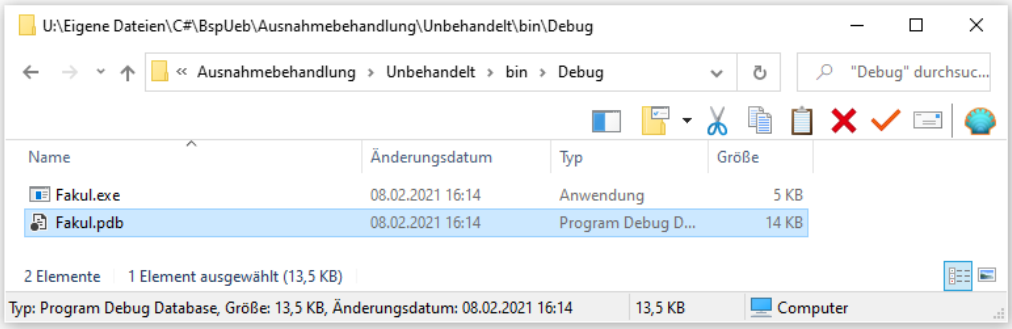
  Projekt > Einstellungen > Build > Erweitert

  vor, wobei für ein Programm mit dem **Zielframework** .NET 5.0 voreingestellt ist:



  Die Vorbereitung für die Ausgabe einer genauen Fehlerlokalisation ist an der Anwesenheit einer Datei mit der Namenserweiterung **.pdb** im Ordner mit dem Assembly zu erkennen, z. B.:[1]

[1] PDB steht für *Program Data Base*.



Durch Optimierungsmaßnahmen des Compilers (z. B. Inlining) kann die Aufrufersequenz kürzer als erwartet ausfallen.

- **InnerException**
  Viele **catch**-Blöcke betätigen sich als Informationsvermittler und werfen selbst eine Ausnahme, um dem Aufrufer einen leichter verständlichen Unfallbericht zu liefern (siehe Abschnitt 13.6). Um dem Aufrufer auch die ursprüngliche Ausnahme zur Verfügung zu stellen, kann man ihre Adresse in der Eigenschaft **InnerException** mitliefern (siehe Abschnitt 13.6).
- **Data**
  Über die **Data**-Eigenschaft mit dem Interface-Typ **IDictionary** kann man Zusatzinformationen zur Ausnahme in einer beliebig langen Schlüssel-Wert - Liste mit Elementen vom Strukturtyp **DictionaryEntry** unterbringen. Man fügt die Zusatzinformationen mit der **IDictionary**-Methode **Add()** ein:

  public void Add (object *key***, object** *value***)**

  Im folgenden Beispiel werden drei Einträge in die **Data**-Liste eines neuen Ausnahmeobjekts aufgenommen:

```
if (arg < 0 || arg > 170) {
   BadFaculArgException bfa =
     new BadFaculArgException("Wert ausselhalb [0, 170]");
   bfa.Data.Add("Input", instr);
   bfa.Data.Add("Type", 4);
   bfa.Data.Add("Value", arg);
   throw bfa;
}
```

Die **ToString()** - Methode eines **Exception**-Objekts liefert:

- den Namen der Ausnahmeklasse
- die **Message**-Zeichenfolge (die beim Erzeugen der Ausnahme formulierte Fehlermeldung)
- die **StackTrace**-Zeichenfolge (die Aufrufreihenfolge)

Beispiel:

```
System.FormatException: Die Eingabezeichenfolge hat das falsche Format.
   bei System.Number.StringToNumber(String str, . . .)
   bei System.Number.ParseInt32(String s, . . .)
   bei System.Convert.ToInt32(String value)
   bei Sequenzen.Calc(String instr) in U:\Eigene Dateien\ ... \Sequenzen.cs:Zeile 8.
```

Objekte aus den folgenden BCL-Ausnahmeklassen (alle im Namensraum **System**) werden in C# - Programmen oft behandelt oder geworfen:

- **ArgumentException**
  Fehlendes oder fehlerhaftes Argument
- **FormatException**
  Ein Argument (z. B. eine zu konvertierende Zeichenfolge oder eine Formatierungszeichenfolge) hat einen ungültigen Aufbau.
- **InvalidOperationException**
  Eine Methode kann unabhängig von den Aktualparameterwerten nicht ausgeführt werden (z. B. Änderung einer Kollektion während einer Iteration, Zugriff auf ein UI-Element durch einen Thread, der das Element nicht erstellt hat.)
- **IndexOutOfRangeException**
  Versuchter Zugriff auf ein nicht vorhandenes Array-Element
- **NotImplementedException**
  In einer Methode ist die angeforderte Operation (noch) nicht implementiert.
- **NullReferenceException**
  Diese Ausnahme wird in der Regel von der CLR geworfen beim Versuch, ein nicht existentes Objekt anzusprechen.

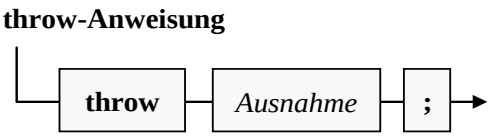
13.5 Ausnahmen werfen (throw)

Unsere eigenen Methoden und Konstruktoren müssen sich nicht auf das *Abfangen* von Ausnahmen beschränken, die vom Laufzeitsystem oder von Bibliotheksmethoden stammen, sondern können sich auch als Werfer betätigen, um bei misslungenen Aufrufen den Absender mit Hilfe der flexiblen **Exception**-Technologie zu informieren.

Insbesondere sollten Methoden und Konstruktoren übergebene Parameterwerte routinemäßig prüfen und ggf. die Ausführung durch das Werfen einer Ausnahme abbrechen. In folgender Variante der Methode `Kon2Int()` aus dem Standardbeispiel von Kapitel 13 wird ein Ausnahmeobjekt aus der BCL-Klasse **ArgumentOutOfRangeException** geworfen, wenn die erfolgreiche Interpretation des Parameters `instr` ein unzulässiges Fakultätsargument ergibt:

```
static int Kon2Int(string instr) {
   int arg;
   arg = Int32.Parse(instr);
   if (arg < 0  ||  arg > 170)
      throw new ArgumentOutOfRangeException(nameof(instr), arg,
          "Argument ausserhalb [0, 170]");
   else
      return arg;
}
```

Zum Auslösen einer Ausnahme dient die **throw**-Anweisung. Sie enthält nach dem Schlüsselwort **throw** eine Referenz auf ein Ausnahmeobjekt (aus der Klasse **System.Exception** oder aus einer abgeleiteten Klasse):



Wie im obigen Beispiel benutzt man oft den **new**-Operator mit nachfolgendem Konstruktor, um vor Ort das Ausnahmeobjekt zu erzeugen.

Im Beispiel wird ein **ArgumentOutOfRangeException**-Konstruktor mit *drei* Parametern verwendet, wobei der Name und der Wert des irregulären Arguments sowie eine Fehlermeldung anzugeben sind. Den Namen des **Parse()** - Parameters mit der zu wandelnden Zeichenfolge liefert der **nameof**-Operator. Aus dem Aufruf

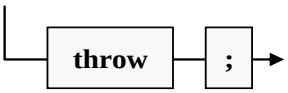
```
try {
   argument = Kon2Int(args[0]);
} catch (Exception e) {
   Console.WriteLine(e.Message);
}
```

mit dem Argument 188 resultiert die Meldung:

```
Argument ausserhalb [0, 170]
Parametername: instr
Der tatsächliche Wert war 188.
```

In einem **catch**-Block darf das Schlüsselwort **throw** auch *ohne* Ausnahmeobjekt-Referenz stehen:

throw-Anweisung ohne Ausnahme-Objekt



In diesem Fall wird die gerade behandelte Ausnahme erneut geworfen. Dieses Verhalten kommt z. B. in Frage, wenn ...

- lediglich ein Protokolleintrag geschrieben werden soll, ohne in die Problembehandlung einzugreifen,
- sich nach einem Lösungsversuch herausstellt, dass der **catch**-Block das Problem nicht aus der Welt schaffen kann, sodass die Ausnahme an den Vorgänger in der Aufrufverschachtelung übergeben werden muss, wobei es sich auch um eine umgebende **try**-Anweisung handeln kann.

Die im **catch**-Block erlaubte reduzierte Form der **throw**-Anweisung hat den Vorteil, dass die **StackTrace**-Information im Ausnahme-Objekt erhalten bleibt. Z. B. wird die folgende Aufrufersequenz

```
   bei System.Number.StringToNumber(String str, NumberStyles options, NumberBuffer& number,
NumberFormatInfo info, Boolean parseDecimal)
   bei System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
   bei Fakul.Kon2Int(String instr) in U:\Eigene Dateien\C#\ ... \catch\Fakul.cs:Zeile 9.
   bei Fakul.Main(String[] args) in U:\Eigene Dateien\C#\ ... \catch\Fakul.cs:Zeile 26.
```

durch die explizite Verwendung der übergebenen **Exception**-Referenz in der **throw**-Anweisung

```
catch (FormatException fo) {
  throw fo;
}
```

reduziert zu:

```
   bei Fakul.Kon2Int(String instr) in U:\Eigene Dateien\C#\ ... \catch\Fakul.cs:Zeile 9.
   bei Fakul.Main(String[] args) in U:\Eigene Dateien\C#\ ... \catch \Fakul.cs:Zeile 26.
```

Statt die ursprüngliche Ausnahme in einem **catch**-Block erneut zu werfen, kommt auch die Verwendung einer anderen Ausnahmeklasse in Frage, die aufgrund der bisherigen Analyse besser geeignet erscheint:

- Meist wird eine *spezifischere* Ausnahmeklasse verwendet, die eine genauere Fehlerbeschreibung ermöglicht. Damit die ursprüngliche Ausnahme als Anlage beigefügt werden kann, bieten die BCL-Ausnahmeklassen einen Konstruktor mit einem Parameter vom Typ **Exception**. Ein Handler kann ggf. über die Eigenschaft **InnerException** auf die Anlage zugreifen.
- Umgekehrt können Sicherheitsüberlegungen zur Wahl einer *allgemeineren* Ausnahmeklasse führen, um wenig technische Details gegenüber potentiellen Angreifern offenzulegen (Albahari & Johannsen 2020, S. 177).

In der aktuellen `Kon2Int()` - Variante wird auf die Behandlung der von **Int32.Parse()** potentiell zu erwartenden Ausnahmen (**FormatException**, **ArgumentNullException**, **OverflowException**) verzichtet. Folglich hat ein `Kon2Int()` - Nutzer insgesamt mit den folgenden Ausnahmeklassen zu rechnen:

- **FormatException**
- **ArgumentNullException**
- **OverflowException**
- **ArgumentOutOfRangeException**

In der `Kon2Int()` - Dokumentation muss darüber informiert werden.

Während die aktuelle `Kon2Int()` - Version durch das Werfen einer Ausnahme aus einer von vier verschiedenen Klassen über Fehler informiert, verzichtet die im Abschnitt 13.2.1.1 vorgestellte Version auf das Werfen von Ausnahmen und kommuniziert Fehler traditionell über einen Returncode mit unterschiedlichen Werten für die Fehlerkategorien. Selbstverständlich ist auch eine Mischung aus der modernen und der traditionellen Fehlersignalisierung möglich. Im Abschnitt 13.3 finden sich Hinweise zur Wahl des Verfahrens.

Seit der Version 7.0 unterstützt C# neben der bisher beschriebenen **throw**-Anweisung auch den **throw**-Ausdruck. Das ermöglicht das Werfen von Ausnahmen an Stellen, die syntaktisch einen Ausdruck verlangen, z. B. im zweiten oder dritten Operanden eines Konditionaloperators:

Quellcode	Ausgabe

`using System;`
 	
`class Prog {`
`  static double Log2(double arg) {`
`  return arg > 0 ?`
`  Math.Log(arg) / Math.Log(2) :`
`  throw new ArgumentException("arg must be positive");`
`  }`
 	
`  static void Main(string[] args) {`
`  Console.WriteLine(Log2(8));`
`  Console.WriteLine(Log2(0));`
`  }`
`}`
	`3`
	`Unbehandelte Ausnahme:`
	`System.ArgumentException:`
	`arg must be positive`

Weitere Einsatzorte für einen **throw**-Ausdruck:

- Ergebnisausdruck eines **switch**-Verzweigungsarms (siehe Abschnitt 15.1), z. B.:
  ```
  null => throw new ArgumentException(message: "No person found")
  ```
- Null-Sammeloperator (siehe Abschnitt 8.3), z. B.:
  ```
  static void Main(string[] args) {
    Console.WriteLine("Argument: " +
      args[0] ?? throw new ArgumentNullException("Argument missing"));
  }
  ```
- Methodendefinition mit Lambda-Operator (siehe Abschnitt 5.7.3), z. B.:
  ```
  static string GetPassword(string name) => throw new NotImplementedException();
  ```

13.6 Ausnahmen definieren

Mit Hilfe von Ausnahmeobjekten kann eine Methode beim Auftreten von Fehlern den Aufrufer ausführlich über Ursachen und Begleitumstände informieren. Dabei muss man sich nicht auf die in der BCL vorhandenen Ausnahmeklassen beschränken, sondern kann eigene Ausnahmen definieren, z. B.:

```
using System;

public class BadFaculArgException : Exception {
   int type, value = -1;
   string input;

   public BadFaculArgException() {
   }
   public BadFaculArgException(string message) : base(message) {
   }
   public BadFaculArgException(string message, Exception innerException)
      : base(message, innerException) {
   }
   public BadFaculArgException(string message, string input_, int type_, int value_)
      : this(message, input_, type_, value_, null) {
   }
   public BadFaculArgException(string message, string input_,
                               int type_, int value_, Exception innerException)
      : base(message, innerException) {
      input = input_;
      if (type_ >= 0 && type <= 3)
         type = type_;
      if (type_ == 4 && (value_ < 0 || value_ > 170)) {
         type = type_;
         value = value_;
      }
   }

   public string Input {get {return input;}}
   public int Type {get {return type;}}
   public int Value {get {return value;}}
}
```

Wir halten uns bei der Klasse `BadFaculArgException` an Microsofts Empfehlungen für selbst definierte Ausnahmeklassen:[1]

- Als Basisklasse sollte **System.Exception** verwendet werden, z. B.:
  ```
  public class BadFaculArgException : Exception { ...
  }
  ```
- Der Klassenname sollte mit dem Wort *Exception* enden.
- Die folgenden *allgemeinen Konstruktoren* sollten mit **public** - Verfügbarkeit implementiert werden:
  - Ein parameterfreier Konstruktor, z. B.:
    ```
    public BadFaculArgException() {
    }
    ```
  - Ein Konstruktor mit einem **string**-Parameter für die Fehlermeldung, z. B.:
    ```
    public BadFaculArgException(string message) : base(message) {
    }
    ```
  - Ein Konstruktor mit einem **string**- Parameter für die Fehlermeldung und einem **Exception**-Parameter für ein inneres Ausnahmeobjekt, das zuvor aufgefangen wurde und nun in ein informativeres Ausnahmeobjekt als Anlage aufgenommen wird, z. B.:
    ```
    public BadFaculArgException(string message, Exception innerException)
      : base(message, innerException) {
    }
    ```

Beim parameterlosen `BadFaculArgException`-Konstruktor beschränken wir uns auf den (impliziten) Aufruf des parameterlosen Basisklassenkonstruktors. Bei den restlichen Konstruktoren rufen

[1] https://docs.microsoft.com/de-de/dotnet/standard/exceptions/how-to-create-user-defined-exceptions

wir explizit eine passende Überladung des Basisklassenkonstruktors auf, um die Eigenschaften **Message** und **InnerException** zu initialisieren.

Durch ein Objekt der selbstdefinierten Ausnahmeklasse `BadFaculArgException` kann ausführlich über Probleme mit dem Argument für die Fakultätsberechnung informiert werden:

- In der Eigenschaft **Message** (geerbt von **Exception**) steht wie üblich eine Fehlermeldung.
- In der Eigenschaft `Input` steht die zu konvertierende Zeichenfolge.
- In der Eigenschaft `Type` wird ein numerischer Indikator für die Fehlerart angeboten:
  - 0: Unbekannt
  - 1: Argument hat den Wert **null**
  - 2: Zeichenfolge kann nicht konvertiert werden
  - 3: **int**-Überlauf
  - 4: **int**-Wert außerhalb [0, 170]
- In der Eigenschaft `Value` steht das Konvertierungsergebnis (falls vorhanden, sonst -1).

Die jüngste `Kon2Int()` - Version kümmert sich um alle von **ToInt32.Parse()** zu erwartenden Ausnahmen und wirft bei allen Fehlerursachen eine spezielle `BadFaculArgException`:

```
static int Kon2Int(ref String instr) {
   int arg;
   try {
      arg = Convert.ToInt32(instr);
   } catch (ArgumentNullException e) {
      throw new BadFaculArgException("Kein Argument vorhanden", null, 1, -1, e);
   } catch (FormatException e) when (Double.TryParse(instr, out double d)) {
      arg = (int)d;
      if (arg == d)
         instr = arg.ToString();
      else
         throw new BadFaculArgException("Fehler beim Konvertieren", instr, 2, -1, e);
   } catch (FormatException e) {
      throw new BadFaculArgException("Fehler beim Konvertieren", instr, 2, -1, e);
   } catch (OverflowException e) {
      throw new BadFaculArgException("Ganzzahl-Überlauf", instr, 3, -1, e);
   }

   if (arg < 0 || arg > 170)
      throw new BadFaculArgException("Wert außerhalb [0, 170]", instr, 4, arg);
   else
      return arg;
}
```

Den in **catch**-Blöcken geworfenen `BadFaculArgException`-Objekten wird das aufgefangene Ausnahmeobjekt beigefügt, um dem Aufrufer keine Information vorzuenthalten.

In der **Main()** - Methode des Beispielprogramms kann eine abgefangene Ausnahme nun präzise protokolliert werden:

```
static void Main(string[] args) {
   int argument = -1;

   if (args.Length == 0) {
      Console.WriteLine("Kein Argument angegeben");
      Console.Read();
      Environment.Exit(1);
   }

   try {
      argument = Kon2Int(ref args[0]);
      double fakul = 1.0;
      for (int i = 1; i <= argument; i++)
         fakul = fakul * i;
      Console.WriteLine("Fakultät von {0}: {1}", args[0], fakul);
      Console.Read();
   } catch (BadFakulArgException e) {
      Console.WriteLine($"Message:\t{e.Message}");
      Console.WriteLine($"Fehlertyp:\t{e.Type}\nEingabe:\t{e.Input} ");
      Console.WriteLine($"Wert:\t\t{e.Value}");
      if (e.InnerException != null)
         Console.WriteLine($"Orig. Message:\t{e.InnerException.Message}");
      Console.Read();
      Environment.Exit(1);
   }
}
```

Bei einem Programmstart mit dem Befehlszeilenargument „vier“ resultiert z. B. die Ausgabe:

```
Message:        Fehler beim Konvertieren
Fehlertyp:      2
Eingabe:        vier
Wert:           -1
Orig. Message:  Die Eingabezeichenfolge hat das falsche Format.
```

Eine eigene Ausnahmeklasse passt gut zur Strategie, Probleme aus diversen Teilschritten einer Aufgabenbearbeitung an einer Stelle zusammenzuführen, wo über das weitere Vorgehen nach einem Scheitern entschieden werden soll. An dieser Entscheidungsstelle muss dann nur *ein* Ausnahmeobjekt behandelt werden, das genügend Informationen über die Art des Problems enthält.

Zum Transport von Zusatzinformationen benötigt eine (selbst erstellte) Ausnahmeklasse nicht unbedingt zusätzliche Felder bzw. Eigenschaften. Alternativ kann man in der **Exception**-Eigenschaft **Data** eine beliebig lange Schlüssel-Wert - Liste mit Elementen vom Typ **DictionaryEntry** unterbringen (siehe Abschnitt 13.4). Bei der Klasse `BadFaculArgException` könnten wir uns auf die folgende Standarddefinition beschränken:

```
using System;

public class BadFaculArgException : Exception {
   public BadFaculArgException() {
   }
   public BadFaculArgException(String message) : base(message) {
   }
   public BadFaculArgException(String message, Exception innerException)
      : base(message, innerException) {
   }
}
```

In die **Data**-Liste eines `BadFaculArgException`-Objekts lassen sich Schlüssel-Wert - Einträge mit den benötigten Zusatzinformationen per **Add()** aufnehmen, z. B.:

```
. . .
try {
   arg = Convert.ToInt32(instr);
} catch (ArgumentNullException e) {
   var bfa = new BadFaculArgException("Kein Argument vorhanden", e);
   bfa.Data.Add("Input", instr);
   bfa.Data.Add("Type", 1);
   bfa.Data.Add("Value", -1);
   throw bfa;
} . . .
```

Das macht aber mehr Aufwand als die Verwendung der oben beschriebenen Ausnahmeklasse mit zusätzlichen Eigenschaften und passenden Konstruktoren, z. B.:

```
. . .
try {
   arg = Convert.ToInt32(instr);
} catch (ArgumentNullException e) {
   throw new BadFaculArgException("Kein Argument vorhanden", null, 1, -1, e);
} . . .
```

Trotzdem ist die **Data**-Liste eine nützliche Option:

- Sie kann ohne Nachteile für vorhandene, eine Ausnahme nutzende Klassen erweitert werden.
- Die Möglichkeit, eine Ausnahmeklasse aus der BCL mit individuellen Zusatzinformationen auszustatten, macht in vielen Fällen die Definition einer eigenen Ausnahmeklasse überflüssig.

Ein **catch**-Block kann auf die **DictionaryEntry**-Elemente in der **Data**-Liste eines `BadFaculArgException`-Objekts per Indexer

```
Console.WriteLine($"Wert = {e.Data["Value"]}");
```

oder mit Hilfe der **DictionaryEntry**-Eigenschaften **Key** und **Value**

```
foreach (DictionaryEntry de in e.Data)
   Console.WriteLine($"{de.Key}\t:\t{de.Value}");
```

zugreifen.

13.7 Übungsaufgaben zum Kapitel 13

1) Erstellen Sie ein Syntaxdiagramm zur **try** - Anweisung (vgl. Abschnitt 13.2.1).

2) Im Beispielprogramm zur Demonstration von möglichen Sequenzen bei der Ausnahmebehandlung (siehe Abschnitt 13.2.2) verzichtet die Methode `Calc()` darauf, die potentiell von der Methode **Convert.ToInt32()** zu erwartende **OverflowException** abzufangen. Bleibt die Ausnahme unbehandelt?

3) Beim Rechnen mit Gleitkommazahlen produziert C# in kritischen Situationen *keine* Ausnahmen, sondern operiert mit speziellen Werten wie **Double.POSITIVE_INFINITY** oder **Double.NaN**. Dieses Verhalten ist oft nützlich, kann aber die Fehlersuche erschweren, wenn mit den speziellen Funktionswerten weitergerechnet wird, und erst am Ende eines längeren Rechenwegs das Ergebnis **NaN** auftaucht (in der Ausgabe: **n. def.**). Im folgenden Beispiel wird eine Methode namens `Log2()` zur Berechnung des dualen Logarithmus[1] verwendet, welche auf die BCL-Methode **Math.Log()** zurückgreift und daher bei ungeeigneten Argumenten (≤ 0) als Rückgabewert **Double.NaN** liefert.

---

[1] Für eine positive Zahl *a* ist ihr Logarithmus zur Basis *b* (> 0) definiert durch:

Quellcode	Ausgabe

`using System;`
 	
`class Prog {`
`  static double Log2(double arg) {`
`  return Math.Log(arg) / Math.Log(2);`
`  }`
 	
`  static void Main() {`
`  double a = Log2(-1);`
`  double b = Log2(8);`
`  Console.WriteLine(a*b);`
`  }`
`}`
	`n. def.`

Erstellen Sie eine Version, die bei ungeeigneten Argumenten eine **ArgumentOutOfRange-Exception** wirft.

4) Erstellen Sie eine Variante der im Abschnitt 8.2.1 vorgestellten generischen Stapelverwaltungsklasse mit einer `Push()` - Methode, die bei besetztem Stapel eine **InvalidOperationException**-Ausnahme wirft, statt den Rückgabewert **false** zu liefern.

$$\log_b(a) := \frac{\ln(a)}{\ln(b)}$$

Dabei steht ln() für den natürlichen Logarithmus zur Basis e (Eulersche Zahl).

14 Attribute

An Typen (Klassen, Strukturen, Schnittstellen, usw.), Member (Methoden, Eigenschaften, usw.), Formalparameter, Typformalparameter und Rückgabewerte sowie an Assemblies und Module (vgl. Abschnitt 2.4.4) kann man *Attribute* anheften, um Metainformationen bereitzustellen, die beim Übersetzen und/oder zur Laufzeit berücksichtigt werden können. Attribute sind Objekte aus speziellen Klassen, die von der abstrakten Basisklasse **System.Attribute** abstammen. Der Compiler speichert die Attributobjekte als Metadaten im erzeugten Assembly. Bei einfachen Attributen besteht die Information über den Träger in der schlichten An- bzw. Abwesenheit des Attributs. Jedoch kann ein Attributobjekt auch Detailinformationen enthalten, die über Eigenschaften verfügbar sind.

Ein Attribut kann das Laufzeitverhalten eines Programms über seine Signalwirkung auf Methoden und/oder die CLR beeinflussen. Ein Beispiel ist das **SerializableAttribute**, mit dem für die Instanzen eines Typs das Serialisieren, d.h. das Speichern in einen Datenstrom (z. B. in eine Datei) erlaubt wird (vgl. Abschnitt 16.5.2). Fehlt dieses Attribut für eine Instanz, die in einen Serialisierungsstrom gerät (z. B. als direktes oder indirektes Mitglied eines anderen Typs), dann resultiert ein Ausnahmefehler.

Wir lernen mit den Attributen eine weitere Technik zur Kommunikation zwischen Programmbestandteilen kennen. Man kann die Attribute als Option zur *deklarativen Programmierung* auffassen. Sie ergänzen die im C# - Sprachumfang verankerten *Modifikatoren* (wie **public**, **abstract** etc.) für Typen, Methoden etc. und bieten dabei eine enorme Flexibilität.[1]

In komplexen objektorientierten Softwaresystemen (Frameworks) spielt generell die als *Reflexion* (engl.: *reflection*) bezeichnete Ermittlung von Informationen über Typen, Methoden usw. zur Laufzeit eine große Rolle. Dabei leisten Attribute einen wichtigen Beitrag.

Von *deklarativer Programmierung* sprachen wir auch bei der Gestaltung einer WPF-Bedienoberfläche per XAML-Code (siehe Abschnitt 5.12.2). Die aktuell behandelten, in den C# - Code zu platzierenden Attribute für Klassen, Methoden, Assemblies etc. sind streng von den Attributen von XAML-Elementen zu unterscheiden. Man kann aber durchaus im gemeinsamen deklarativen Ansatz der beiden Optionen zur Entwicklung von .NET - Software eine Ursache für die übereinstimmende Wortwahl sehen.

Wir behandeln zunächst die Vergabe und Auswertung von BCL-Attributen, beschäftigen uns später aber auch mit der Definition von eigenen Attributen.

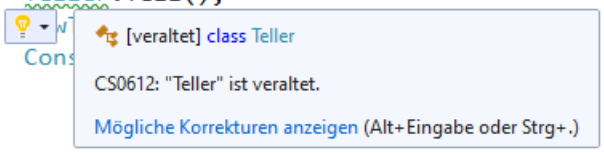
---

[1] Gelegentlich wird im Zusammenhang mit unserem aktuellen Thema von *benutzerdefinierten Attributen* gesprochen (engl.: *custom attributes*), siehe z. B.
https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/concepts/attributes/creating-custom-attributes
Wichtige Methoden zur Auswertung von Attributen (siehe Abschnitt 14.2) führen das Wort *Custom* im Namen. Damit sollen die von der **System.Attribute** abstammenden Klassen, deren Objekte vom Software-Entwickler in deklarativer Absicht an diverse C# - Elemente angeheftet werden können, von deklarativen Bestandteilen der Programmiersprache C# unterschieden werden. Während die zum Sprachumfang gehörenden Attribute (wie z. B. die Modifikatoren **sealed**, **static**, **public** usw.) über viele Jahre kaum ergänzt wurden, können **System.Attribute** - Ableitungen von den Software-Entwicklern jederzeit neu erstellt und verwendet werden. Das Manuskript orientiert sich bei der Begriffsverwendung an der C# - Sprachspezifikation (ECMA 2017, Kap. 22). Dort ist nur von *Attributen* die Rede, und dabei sind die von **System.Attribute** abstammenden Klassen gemeint.

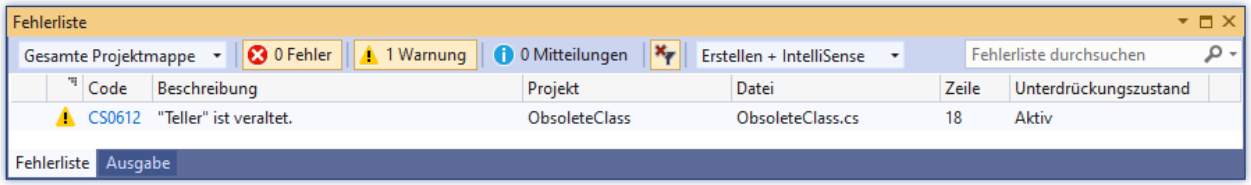
14.1 Attribute vergeben

Sollen z. B. die Benutzer einer Klassenbibliothek dazu gebracht werden, einer neuen Klasse den Vorzug gegenüber einer alten zu geben, dann kann man der alten Klassendefinition zwischen eckigen Klammern das Attribut **Obsolete** voranstellen, sodass der Compiler bei Verwendung dieser Klasse automatisch eine Warnung ausgibt. Dies geschieht z. B. beim Übersetzen der folgenden Quelle:

```
using System;

[Obsolete]
public class Teller {
    public static void Tell() {
        Console.WriteLine("Hallo!");
    }
}

public class NewTeller {
    public static void Tell() {
        Console.WriteLine("Hallo, Wilhelm!");
    }
}

class Prog {
    static void Main() {
        Teller.Tell();
        Cons
    }
}
```



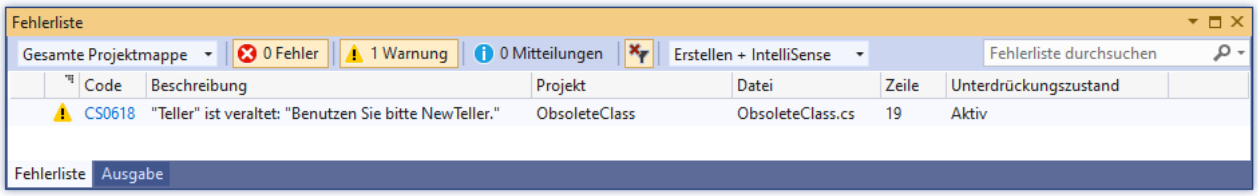
Im Editor der Entwicklungsumgebung wird der unerwünschte Zugriff auf die obsolete Klasse unterschlängelt und bei passender Mauszeigerpositionierung entsprechend kommentiert. In der **Fehlerliste** erscheint eine **Warnung**:



Wer als Klassenbibliotheksdesigner bzw. -renovierer diese Compiler-Meldung zu dürftig findet, kann zum Erstellen des **Obsolete**-Attributs eine alternative Konstruktorüberladung verwenden und die **Message**-Eigenschaft des Objekts mit einer Zeichenfolge versorgen, z. B.:

```
[Obsolete("Benutzen Sie bitte NewTeller.")]
```

Dann meldet der Compiler beim unerwünschten Zugriff:



Bei der Vergabe eines Attributes wird durch einen Konstruktor der Attributklasse ein Objekt erstellt, das die Metadaten der dekorierten Klasse ergänzt.

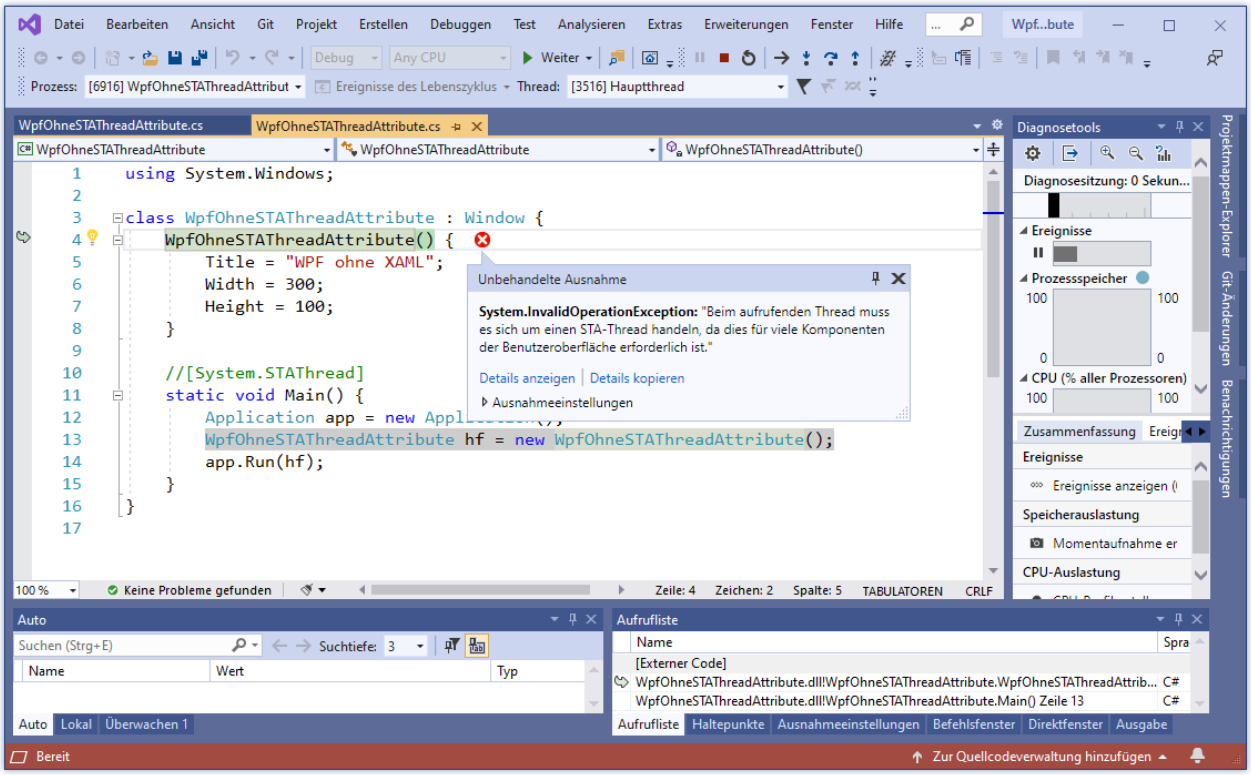
Je nach verwendeter Konstruktorüberladung sind Aktualparameter anzugeben, wobei eine leere Aktualparameterliste weggelassen werden kann. Als weitere syntaktische Besonderheit darf bei der Vergabe eines Attributs der Namensteil *Attribute* im Klassennamen entfallen. Im Beispiel kommt die Klasse **ObsoleteAttribute** (aus dem Namensraum **System**) zum Einsatz.

Neben Klassen können auch andere Programmbestandteile mit Attributen versehen werden, z. B. Methoden:

```
using System;

class Teller {
    [Obsolete("Benutzen Sie bitte TellEx().")]
    public static void Tell() {
        Console.WriteLine("Hallo!");
    }

    public static void TellEx() {
        Console.WriteLine("Hallo, Wilhelm!");
    }
}

class Prog {
    static void Main() {
        Teller.Tell();
    }
}
```

```
class Teller

CS0618: "Teller.Tell()" ist veraltet: "Benutzen Sie bitte TellEx()."

Mögliche Korrekturen anzeigen (Alt+Eingabe oder Strg+.)
```

Wie Sie aus dem Abschnitt 12.2.1 wissen, muss in einer WPF-Anwendung die Methode **Main()** generell das **STAThreadAttribute** erhalten. Anderenfalls scheitert schon die Ausführung des Fensterklassenkonstruktors an einer **InvalidOperationException**, z. B.:



Mit dem **STAThreadAttribute** wird signalisiert, dass bei der Kooperation mit dem Component Object Model (COM), der noch weit verbreiteten Windows-Komponententechnologie, das COM-Threading-Modell STA (*Singlethread-Apartment*) zum Einsatz kommen soll, um Synchronisationsprobleme bei Steuerelementen zu verhindern.

Seit C# 7.3 besteht die Möglichkeit, auch das private *backing field* einer automatisch implementierten Eigenschaft (siehe Abschnitt 5.5.2) mit Attributen zu dekorieren.[1] Dazu werden der Attributbezeichnung das Schlüsselwort **field** und ein Doppelpunkt vorangestellt. Im folgenden Beispiel geschieht dies bei einer Eigenschaft zur Erfassung der Stimmung einer Person, um die Speicherung oder Übertragung der Eigenschaftsausprägung im Rahmen einer binären Serialisierung (siehe Abschnitt 16.5.1) zu verhindern:

```
public class Person {
    . . .
   [field: NonSerialized]
   public int Mood { get; set; }
    . . .
}
```

Im weiteren Verlauf von Kapitel 14 wird noch klarer, dass man bei der Vergabe von Attributen in der Regel nicht nur den Quellcode kommentiert und den Compiler informiert, sondern signalisierend den Programmablauf beeinflusst, sofern andere Akteure (z. B. andere Typen oder die CLR) die Attribute kennen und bei ihrem Verhalten berücksichtigen.

14.2 Attribute per Reflexion auswerten

Die .NET - Plattform bietet leistungsfähige *Reflexionstechniken*, die es u. a. erlauben, die Attributausstattung von Programmbestandteilen zur Laufzeit zu analysieren. Mit der statischen Methode **IsDefined()** der Klasse **System.Attribute** kann man feststellen, ob ein bestimmtes Attribut vorhanden ist. Das folgende Programm prüft für eine Klasse und für eine Methode, ob das **ObsoleteAttribute** angeheftet ist:

```
using System;
using System.Reflection;

[Obsolete("Benutzen Sie bitte NewTeller.")]
public class Teller {
    [Obsolete("Benutzen Sie bitte TellEx().")]
    public void Tell() {
        Console.WriteLine("Hallo!");
    }
}

class Prog {
   static void Main() {
      Type typeTeller = typeof(Teller);
      Type typeObsolete = typeof(ObsoleteAttribute);

      if (Attribute.IsDefined(typeTeller, typeObsolete))
         Console.WriteLine("Der Typ {0}\n ist obsolet", typeTeller.Name);

      MemberInfo[] mi = typeTeller.FindMembers(MemberTypes.Method,
          BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public,
          Type.FilterName, "Tell*");
```

[1] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-7.3/auto-prop-field-attrs

```
      foreach (var m in mi)
         if (Attribute.IsDefined(m, typeObsolete))
            Console.WriteLine("\nDie Methode {0}\n ist obsolet", m.Name);
   }
}
```

Beide Prüflinge werden als obsolet erkannt:

```
Der Typ Teller
 ist obsolet

Die Methode Tell
 ist obsolet
```

Die verwendete **IsDefined()** - Überladung erwartet als ersten Parameter ein Objekt der Klasse **MemberInfo** aus dem Namensraum **System.Reflection**, von der auch die Klasse **Type** abstammt, die einen Datentyp (Klasse, Struktur, Schnittstelle etc.) repräsentiert. Im Beispiel wird **IsDefined()** zweimal aufgerufen:

- Im ersten Aufruf ist eine Klasse der potentielle Attributträger:
  `Attribute.IsDefined(typeTeller, typeObsolete)`
- Im zweiten Aufruf wird die Anwesenheit eines Methodenattributs untersucht:
  `Attribute.IsDefined(mi[0], typeObsolete)`

Als zweiter Parameter ist das **Type**-Objekt zur fraglichen Attributklasse anzugeben.

Im Beispiel werden Referenzen auf **Type**-Objekte per **typeof**-Operator gewonnen. Anders als bei der Attributvergabe (siehe Abschnitt 14.1) ist dabei der Name der Attributklasse vollständig (inkl. Namenbestandteil *Attribute*) zu schreiben, z. B.:

```
Type typeObsolete = typeof(ObsoleteAttribute);
```

Das im zweiten **IsDefined()** - Aufruf benötigte **MemberInfo**-Objekt zur `Teller`-Methode `Tell()` besorgt die Instanzmethode **FindMembers()** der Klasse **Type**. Sie liefert Information über die Member des befragten **Type**-Objekts in einem Array vom Typ **MemberInfo**:

- Im ersten **FindMembers()** - Parameter wählt man die Member-Kategorie.
- Mit dem zweiten Parameter lässt sich die Suche über eine ODER-Verknüpfung von Werten der Enumeration **BindingFlags** steuern. Im Beispiel werden Klassen- *und* Instanz-bezogene Member zugelassen, sofern diese als **public** deklariert sind:[1]
  `BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public`
- Als dritter Aktualparameter sorgt im Beispiel das durch die Eigenschaft **Type.FilterName** referenzierte Delegatenobjekt vom Typ **MemberFilter** für eine namensorientierte Filterung. Mit der Rückgabe **true** wird signalisiert, dass der Name eines Mitglieds dem im letzten **FindMembers()** - Parameter genannten Kriterium entspricht.

---

[1] Wie die folgende Webseite
https://docs.microsoft.com/en-us/dotnet/api/system.reflection.bindingflags
erläutert, muss zusammen mit **BindingFlags.Instance** und/oder **BindingFlags.Static** unbedingt auch **BindingFlags.Public** und/oder **BindingFlags.NonPublic** angegeben werden:
You must specify Instance or Static along with Public or NonPublic or no members will be returned.
Wenn z. B. **BindingFlags.Instance** angegeben wird, und eine **public**-deklarierte Methode vorhanden ist, welche die Filterbedingung erfüllt, dann muss auch **BindingFlags.Public** angegeben werden, damit die Methode gefunden wird.
Neben **Instance**, **Static**, **Public** und **NonPublic** hat die Enumeration **BindingFlags** noch zahlreiche weitere Werte, die nicht behandelt werden können.

- Im letzten Parameter spezifiziert man den geforderten Namen, wobei das Jokerzeichen ***** am Ende des Suchstrings erlaubt ist. Der vierte Parameter wird dem Delegatenobjekt des 3. Parameters als Aktualparameter übergeben.

Soll nicht nur die Existenz eines Attributs festgestellt, sondern auch sein Innenleben exploriert werden, eignet sich die statische Methode **GetCustomAttribute()** der Klasse **System.Attribute**. Sie rekonstruiert das angeheftete Objekt aus den Metadaten im Assembly, sodass öffentliche Felder und Eigenschaften zur Verfügung stehen. Wegen der Objektkreation sind die Kosten eines Aufrufs höher als bei der Methode **IsDefined()**. Die folgende Methode `ObsoleteMethodCheck()` prüft für alle öffentlichen Methoden eines Typs, ob sie als obsolet markiert sind:

```
static void ObsoleteMethodCheck(Type tt) {
 Attribute attrib;
 MemberInfo[] members = tt.FindMembers(MemberTypes.Method,
  BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public,
  Type.FilterName, "*");
 Console.WriteLine("Obsolete-Prüfung für die Methoden des Typs {0}:",
                   tt.FullName);
 foreach (MemberInfo mi in members) {
  attrib = Attribute.GetCustomAttribute(mi, typeof(ObsoleteAttribute));
  if (attrib != null) {
   Console.WriteLine("\nDie Methode {0}() ist obsolet.", mi.Name);
   Console.WriteLine("Message: " + (attrib as ObsoleteAttribute).Message);
  } else
   Console.WriteLine("\nDie Methode {0}() ist noch aktuell.", mi.Name);
 }
}
```

Per **GetCustomAttribute()** erhalten wir für jede Methode ggf. das angeheftete **ObsoleteAttribute**-Objekt und fragen dieses Objekt nach seiner **Message**-Eigenschaft.

Über die Klasse

```
class Teller {
   [Obsolete("Benutzen Sie bitte TellEx().")]
   public static void Tell() {
      Console.WriteLine("Hallo!");
   }
   public static void TellEx() {
      Console.WriteLine("Hallo, Wilhelm!");
   }
}
```

erhalten wir den Bericht:

```
Obsolete-Prüfung für die Methoden der Klasse Teller:

Die Methode Tell() ist obsolet
Message: Benutzen Sie bitte TellEx().

Die Methode TellEx() ist noch aktuell.

Die Methode ToString() ist noch aktuell.

Die Methode Equals() ist noch aktuell.

Die Methode GetHashCode() ist noch aktuell.

Die Methode GetType() ist noch aktuell.
```

In der **Main()** - Methode des nächsten Beispielprogramms werden mit der statischen **Attribute**-Methode **GetCustomAttributes()** für eine per **Type**-Objekt beschriebene Klasse *alle* angehefteten Attribute ermittelt, wobei *geerbte* Attribute nicht interessieren (Wert **false** für den Parameter **inherit**):

Quellcode	Ausgabe

`using System;`
 	
`[Obsolete] [Serializable]`
`class Teller {`
`  public static void Tell() {`
`  Console.WriteLine("Hallo!");`
`  }`
`}`
 	
`class Prog {`
`  static void Main() {`
`  Type type = typeof(Teller);`
`  Attribute[] atar =`
`  Attribute.GetCustomAttributes(type, false);`
`  foreach (Attribute at in atar)`
`  Console.WriteLine(at);`
`  }`
`}`
	`System.ObsoleteAttribute`
	`System.SerializableAttribute`

Mit alternativen Überladungen lassen sich Assemblies, Members, Parameter etc. analog untersuchen.

14.3 Attribute für Assemblies

Auch Assemblies können Attribute erhalten, wobei aber mangels syntaktischer Entsprechung für diese Übersetzungseinheiten der Bezug nicht durch die Platzierung der Attribute im Quellcode hergestellt werden kann. Stattdessen benutzt man Attribute mit expliziter Widmung, z. B.:

```
[assembly: AssemblyCompany("Marco Saft")]
[assembly: AssemblyProduct("YourTools")]
[assembly: AssemblyVersion("1.4.2.3")]
```

Hier wird das vom Compiler zu erzeugende Assembly als Träger des nachfolgenden Attributs festgelegt.

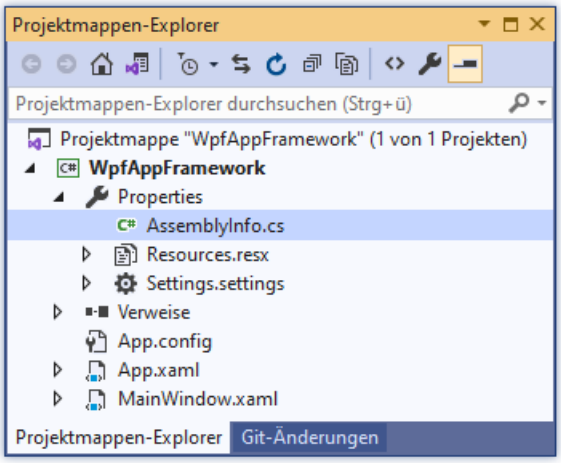
Bei der technischen Realisation geht Microsoft bei .NET Core (samt Nachfolger .NET 5.0) andere Wege als im .NET Framework.

14.3.1 Assembly-Attribute im .NET Framework

Traditionell werden Assembly-Attribute vom Entwickler in der Datei **AssemblyInfo.cs** deklariert und vom Compiler als Metadaten in das erstellte Assembly befördert. Zu einer neuen **WPF-App (.NET Framework)** legt das Visual Studio im Projektordner **Properties** die Datei **AssemblyInfo.cs** mit Vorschlägen für wichtige Assembly-Attribut - Deklarationen an, z. B. (**using**-Direktiven und Kommentare aus Platzgründen weggelassen):

```
[assembly: AssemblyTitle("WpfAppFramework")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("WpfAppFramework")]
[assembly: AssemblyCopyright("Copyright ©  2021")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
[assembly: ComVisible(false)]
[assembly: ThemeInfo(
    ResourceDictionaryLocation.None,
    ResourceDictionaryLocation.SourceAssembly
)]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

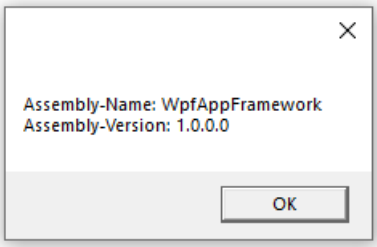
Zur Modifikation dieser Attribute öffnet man die Datei **AssemblyInfo.cs** über den Projektmappen-Explorer:



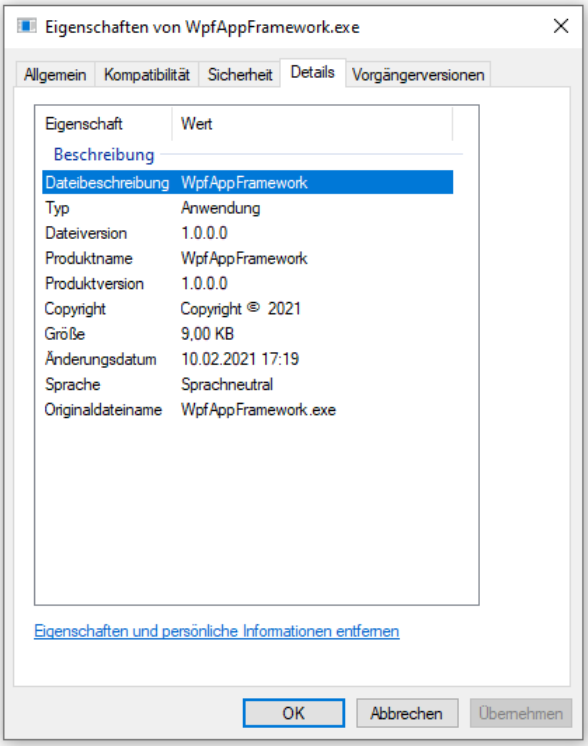
In einem Programm kann man auf einige Attribute des gerade ausgeführten Assemblies über Eigenschaften des zugehörigen **Assembly**-Objekts zugreifen, z. B.:

```
private void Button_Click(object sender, RoutedEventArgs e) {
 System.Reflection.Assembly ass = System.Reflection.Assembly.GetExecutingAssembly();
 MessageBox.Show("Assembly-Name: " + ass.GetName().Name+
                 "\nAssembly-Version: " + ass.GetName().Version);
}
```

Aufgrund der obigen Deklarationen in **AssemblyInfo.cs** erhält man die folgende MessageBox:



Außerdem erscheinen die Assembly-Attribute im Eigenschaftsdialog einer Assembly-Datei, z. B.:

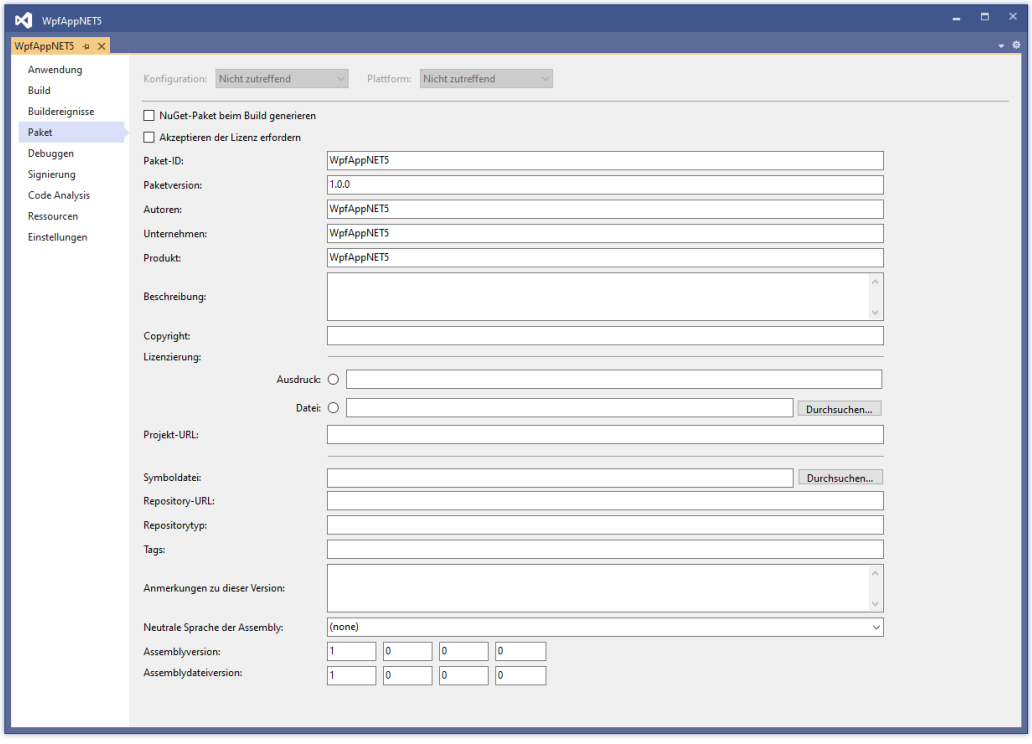


14.3.2 Assembly-Attribute in .NET 5.0

Auch in einer WPF-Anwendung für .NET 5.0 erstellt das Visual Studio eine Datei namens **AssemblyInfo.cs**, die sich im Hauptordner des Projekts befindet. Allerdings spielt diese Datei per Voreinstellung keine große Rolle bei der Verwaltung von Assembly-Attributen. Stattdessen sollen die Assembly-Attribute nach

Projekt > Eigenschaften > Paket

auf der folgenden Registerkarte vereinbart werden:



Sobald man die (überwiegend aus dem Projektnamen entstandenen) Voreinstellungen ändert, werden die neuen Werte in die Projektdatei geschrieben, bei einem Projekt mit dem Namen `WpfAppNET5` also in die Datei **WpfAppNET5.csproj** im Hauptordner des Projekts.

Bei jeder Erstellung des Projekts mit dem Namen `WpfAppNET5` und mit dem Zielframework .NET 5.0 werden die Assembly-Attribute in die folgende Datei geschrieben:

...\obj\Debug\net5.0-windows\WpfAppNET5.AssemblyInfo.cs

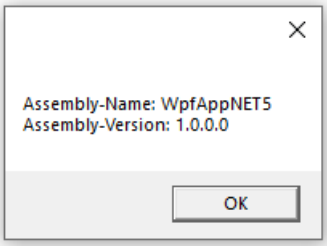
Es wäre nutzlos, in dieser Datei Änderungen vorzunehmen.

Einträge in der Datei **AssemblyInfo.cs** im Hauptordner des Projekts werden vom Compiler durchaus beachtet. Befinden sich dort die im .NET Framework üblichen Vereinbarungen von Assembly-Attributen, dann beschwert sich der Compiler über „doppelte Attribute“. Man kann der Datei **AssemblyInfo.cs** die altgewohnte alleinige Berechtigung zur Definition von Assembly-Attributen verschaffen durch das Element **GenerateAssemblyInfo** mit dem Wert **false** in der Projektdatei:

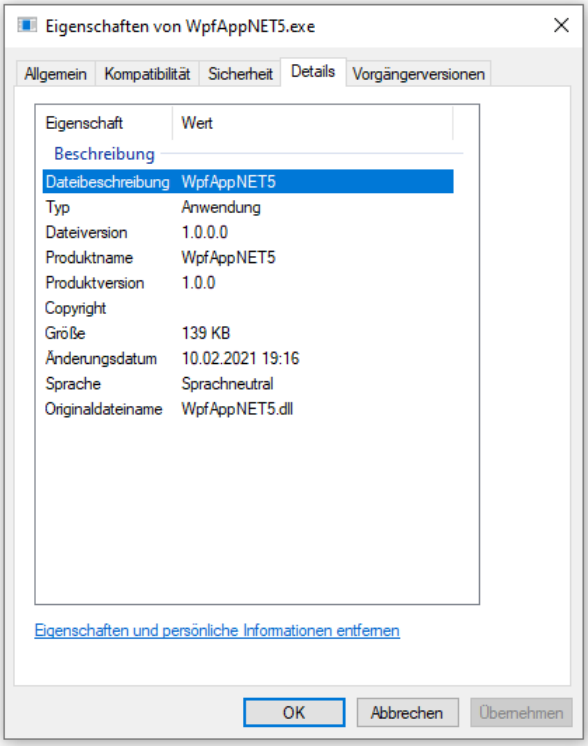
```
<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net5.0-windows</TargetFramework>
    <UseWPF>true</UseWPF>
    <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
  </PropertyGroup>

</Project>
```

Allerdings ist es wohl nicht empfehlenswert, in die ohnehin komplexe Assembly-Erstellung einzugreifen.

Beim Zugriff auf die Assembly-Attribute aus dem laufenden Programm



oder per Windows-Explorer



ändert sich in .NET 5.0 nichts im Vergleich zum .NET Framework.

14.4 Aufruferinformations-Attribute

Die Aufruferinformations-Attribute lassen sich an optionale Methodenparameter (mit Voreinstellungswerten, siehe Abschnitt 5.3.3.2) heften und veranlassen den Compiler, die Parameterwerte zu setzen:[1]

- **CallerMemberName**
  Der Compiler fügt den Namen des Aufrufers ein. Damit erfüllt das Attribut **CallerMemberName** für den Namen der aufrufenden Methode oder Eigenschaft dieselbe Funktion wie der **nameof**-Operator für einen Variablennamen. Man vermeidet Literale im Quellcode, der folglich auch nach einer Umbenennung per Refaktorierung (siehe Abschnitt 3.3.3) gültig bleibt.
- **CallerFilePath**
  Der Compiler fügt den Pfad der Quellcodedatei ein.
- **CallerLineNumber**
  Der Compiler fügt die Zeile der Quellcodedatei mit dem Methodenaufruf ein.

Die beiden Einsatzzwecke für diese Attribute sind:

- Implementation der Schnittstelle **INotifyPropertyChanged** zur Realisation von Datenbindungen (siehe Albahari & Johannsen 2020, S. 207f)
- Ablaufverfolgung, Debuggen und Erstellung von Diagnosewerkzeugen

Im folgenden Programm

[1] https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/attributes/caller-information

```
using System;
using System.Runtime.CompilerServices;

class Aufruferinformation {
   static void M ([CallerMemberName] string memberName = null,
                  [CallerFilePath] string filePath = null,
                  [CallerLineNumber] int lineNumber = 0) {
      Console.WriteLine("Called from " + memberName);
      Console.WriteLine("Defined in " + filePath);
      Console.WriteLine("At line " + lineNumber);
   }

   static void Main() {
      M(); // Zeile 14
   }
}
```

weiß die Methode `M()` einiges über ihren Aufrufer:

```
Called from Main
Defined in U:\Eigene Dateien\C#\BspUeb\Attribute\Aufruferinformation\Aufruferinformation.cs
At line 14
```

14.5 Weitere nützliche BCL-Attribute

Neben den im bisherigen Verlauf des Kapitels erwähnten BCL-Attributen für Klassen, Methoden, Eigenschaften und Assemblies enthält die BCL noch etliche weitere Attribute, die im Programmieralltag von Interesse sind. Anschließend wird eine kleine Auswahl vorgestellt.

14.5.1 Bedingte Methodenausführung per ConditionalAttribute

Wenn Kontrollausgaben oder andere Operationen nur in bestimmten Projektphasen (z. B. bei der Fehlersuche) erfolgen sollen, dann kann man folgendermaßen verfahren:

- Man definiert eine Methode mit den bedingt auszuführenden Operationen.
- Diese Methode erhält das Attribut **Conditional** aus dem Namensraum **System.Diagnostics**, wobei dem Konstruktor eine Bezeichnung der Bedingung als Aktualparameter zu übergeben ist, z. B.:

```
public class Trace {
   [Conditional("Trace")]
   public static void Log(string msg) {
      Console.WriteLine(msg);
   }
}
```

- Diese Methode wird wie gewohnt in Programmen aufgerufen.
- Damit die Aufrufe tatsächlich ausgeführt werden, muss im Quellcode das Symbol aus dem **Conditional**-Konstruktor per **Präprozessor-Kommando** definiert werden. z. B.:

```
#define Trace
using System;
using System.Diagnostics;
 . . .
```

Im folgenden Beispiel

```
class Condemo {
   static void Main() {
      int anzahl = 1_000;
      double zuf;
      Random ran = new Random();

      Trace.Log("Vor Beginn der Schleife");
      long vorher = DateTime.Now.Ticks;
      for (int i = 0; i < anzahl; i++) {
         zuf = ran.NextDouble();
         if (i % 100 == 0)
            Trace.Log(i.ToString());
      }
      long diff = DateTime.Now.Ticks - vorher;
      Trace.Log("Nach Beendigung der Schleife: " + diff / 1.0e4 + " Millisekunden");
   }
}
```

machen Kontrollausgaben eine Schleife beobachtbar:

```
Vor Beginn der Schleife
0
100
200
 . . .
900
Nach Beendigung der Schleife: 16,1531 Millisekunden
```

Die zur Fehlersuche nützlichen `Log()` - Methodenaufrufe sollen natürlich im normalen Programmeinsatz *nicht* ausgeführt werden.

Neben **#define** kennt C# noch etliche weitere Präprozessor-Kommandos.[1]

14.5.2 Bitfelder per FlagsAttribute

Bei einem Enumerationstyp (vgl. Abschnitt 6.4) signalisiert der Entwickler mit dem **System.FlagsAttribute**, dass ein Wert als *Bitfeld* interpretierbar ist, d.h.:

- Die ersten *k* Bits (mit dem niederwertigsten beginnend) des zugrunde liegenden Datentyps (meist **int**) stehen als unabhängige Informationsträger jeweils für ein dichotomes Merkmal (mit den Werten 0 und 1). Ein Enumerationswert kodiert also die Ausprägungen von *k* dichotomen Merkmalen. Bei der Enumeration **ModifierKeys** aus dem BCL-Namensraum **System.Windows.Input** stehen die ersten vier Bits für die Vorschalttasten **Alt**, **Strg**, **Umschalt** und **Windows**:[2]

```
[Flags]
 . . .
public enum ModifierKeys {
   None = 0,
   Alt = 1,
   Control = 2,
   Shift = 4,
   Windows = 8
}
```

---

[1] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/preprocessor-directives/

[2] Der Quellcode zur Enumeration **ModifierKeys** in der BCL zu .NET 5.0 kann über die Webseite https://source.dot.net/ inspiziert werden.

- Jede bitweise ODER-Kombination von zwei benannten Werten der Enumeration ergibt ein sinnvoll interpretierbares Bitfeld, also eine zulässige Kombination der dichotomen Einzelmerkmale. Mit dem folgenden **ModifierKeys**-Wert lässt sich z. B. prüfen, ob die **Strg**- und die Umschalttaste simultan gedrückt sind:

```
ModifierKeys.Control | ModifierKeys.Shift
```

Bei einem gewöhnlichen Enumerationstyp (ohne **FlagsAttribute**) …

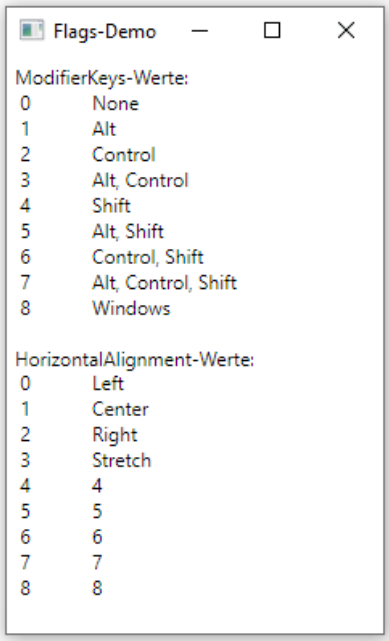
- stehen die Werte für die sich *gegenseitig ausschließenden* Ausprägungen *eines* Merkmals. Z. B. codieren bei der im Abschnitt 12.7.3.2 erwähnten Enumeration **HorizontalAlignment** (im Namensraum **System.Windows**) die ersten vier nicht-negativen **int**-Werte jeweils eine horizontale Orientierung eines WPF-Steuerelements gegenüber der umgebenden Containerzelle (**Left**, **Center**, **Right**, **Stretch**): [1]

```
public enum HorizontalAlignment {
   Left = 0,
   Center = 1,
   Right = 2,
   Stretch = 3
}
```

- Eine bitweise ODER-Verknüpfung der Werte ist zwar syntaktisch erlaubt, aber sinnlos.

Im folgenden WPF-Programm wird demonstriert, dass die Summe von zwei **ModifierKeys**-Werten wieder ein sinnvoller Wert dieses Typs ist, die Summe von zwei **HorizontalAlignment**-Werten hingegen nicht:

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;

class FlagsDemo : Window {
   FlagsDemo() {
      Label lbl = new();
      AddChild(lbl);
      string flags = "ModifierKeys-Werte:\n";
      for (ModifierKeys i = 0; (int)i <= 8; i++)
         flags = flags + (int)i + "\t" + i + "\n";
      flags += "\nHorizontalAlignment-Werte:\n";
      for (HorizontalAlignment i = 0; (int)i <= 8; i++)
         flags += (int)i + "\t" + i + "\n";
      lbl.Content = flags;
      Title = "Flags-Demo"; Width = 250; Height = 400;
   }

   [System.STAThread]
   static void Main() {
      new Application().Run(new FlagsDemo());
   }
}
```



Die informative Rückgabe der Methode **ToString()** kommt zustande, weil die Enumerationsbasisklasse **Enum** im .NET - Framework 4.x die Existenz des **Flags**-Attributs per **IsDefined()** - Aufruf (vgl. Abschnitt 14.2) überprüft[2]

---

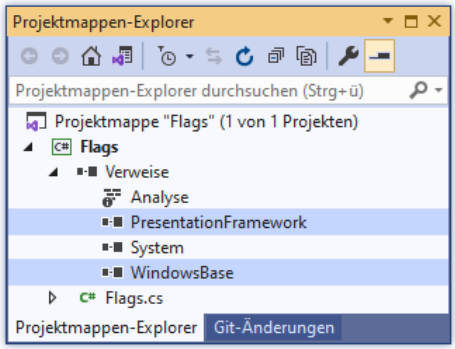
[1] Der Quellcode zur Enumeration **HorizontalAlignment** in der BCL zu .NET 5.0 kann über die Webseite https://source.dot.net/ inspiziert werden.

[2] Das Quellcodesegment gehört zur Enumerationsbasisklasse **Enum** in der Standardbibliothek zum .NET Framework 4.8 und kann hier inspiziert werden: https://referencesource.microsoft.com/

```
if (!eT.IsDefined(typeof(System.FlagsAttribute), false)) {...}
else {...}
```

und ggf. in der **ToString()** - Überschreibung für jeden Wert der Enumeration eine Komma-separierte Liste der Merkmale mit einem angeschalteten Bit ausgibt. Hier orientiert also eine Methode ihr Verhalten an der Anwesenheit eines Attributs.

Das Konsolenprogramm benötigt Verweise auf die Assemblies **PresentationFramework.dll** und **WindowsBase.dll**:



14.5.3 Unions per StructLayoutAttribute und FieldOffsetAttribute

Bei der im Abschnitt 4.3.5.1 zur Erläuterung der binären Gleitkommadarstellung benutzten (aber nicht erklärten) Anwendung `FloatBits` werden die Attribute **StructLayoutAttribute** und **FieldOffsetAttribute** aus dem Namensraum **System.Runtime.InteropServices** dazu verwendet, eine **Union** im Sinn der Programmiersprache C nachzubilden. In unseren Begriffen handelt es sich dabei um eine Struktur, deren Instanzvariablen im Speicher (zumindest teilweise) überlappen. In der Regel soll damit nicht etwa Speicherplatz gespart, sondern eine unterschiedliche Interpretation desselben Speicherinhalts ermöglicht werden.

In den folgenden Zeilen wird eine Struktur mit dem (frei gewählten) Namen `Union` sowie Feldern vom Typ **float** und **uint** definiert:

```
[StructLayout(LayoutKind.Explicit)]
public struct Union {
   [FieldOffset(0)]
   public float f;
   [FieldOffset(0)]
   public uint u;
}
```

Per **StuctLayoutAttribut** mit Konstruktor-Parameter **LayoutKind.Explicit** wird dem Compiler mitgeteilt, dass die Speicheradressen der beiden Felder explizit durch **FieldOffsetAttribut**e festgelegt werden sollen. So wird es möglich, die beiden Felder an derselben Anfangsadresse 0 beginnen zu lassen. Die beiden Typen (**float**, **uint**) haben denselben Platzbedarf von vier Bytes (siehe Abschnitt 4.3.4).

Das Programm `FloatBits` schreibt den vom Benutzer angegebenen **float**-Wert in das `f`-Feld einer `Union`-Instanz und liest anschließend über das `u`-Feld der Instanz aus demselben Speicherbereich einen **uint**-Wert, der über bitorientierte Operatoren (siehe Abschnitt 4.5.6) untersucht werden kann:

```
class FloatBits {
   static void Main() {
      Union uni = new Union();
      Console.Write("float: ");
      uni.f = Convert.ToSingle(Console.ReadLine());
      Console.WriteLine("\nBits:\n1 12345678 12345678901234567890123");
      for (int i = 31; i >= 0; i--) {
         if (i == 30 || i == 22)
            Console.Write(' ');
         Console.Write((1U << i & uni.u) >> i);
      }
   }
}
```

14.6 Attribute definieren

Wir müssen uns nicht auf die Vergabe und Auswertung von BCL-Attributen beschränken, sondern können auch eigene Attribute definieren. Bei einer eigenen Attributklasse sollte man …

- die Klasse **System.Attribute** oder eine Spezialisierung als Basisklasse verwenden,
- den Klassennamen mit dem Wort *Attribute* enden lassen.

Um den Compiler darüber zu informieren, welchen Programmbestandteilen das neue Attribut angeheftet werden darf, verwendet man ein Attribut aus der Klasse **AttributeUsageAttribute**. Im folgenden Beispiel erhält der Konstruktorparameter **validOn** den Wert **AttributeTargets.Class**, sodass nur *Klassen* das neu definierte `NonsenseAttribute` erhalten dürfen. Außerdem wird mit dem Wert **false** für die **AttributeUsageAttribute**-Eigenschaft **Inherited** verhindert, dass eine dekorierte Klasse das `NonsenseAttribute` an abgeleitete Klassen weitergibt:

```
[AttributeUsage(AttributeTargets.Class, Inherited=false)]
public class NonsenseAttribute : Attribute {
   public NonsenseAttribute(int level_) {
      Level = level_;
   }
   public int Level { get; }
}
```

Wie das Metaattribut **AttributeUsage** (ein Attribut, das an Attribute geheftet wird) demonstriert, kommt bei der Attributvergabe zur Eigenschaftsinitialisierung innerhalb der Konstruktor-Parameterliste eine spezielle Syntax für Namensparameter, die nach den regulären Parametern (Positionsparametern) in beliebiger Reihenfolge stehen dürfen, zum Einsatz. Diese Namensparameter werden nicht als Konstruktorargument definiert. Stattdessen kann jede öffentliche instanzbezogene Eigenschaft oder Variable als Namensparameter verwendet und auf diese Weise initialisiert werden. Unter der Bezeichnung *Objekt-* bzw. *Instanz-Initialisierer* steht seit C# 3.0 für beliebige Klassen eine ähnliche Initialisierungsmöglichkeit zur Verfügung (siehe Abschnitt 5.4.3.3).

Bei Positions- oder Namensparametern von **Attribut**-Konstruktoren sind ausschließlich die folgenden Datentypen erlaubt:

- **bool**, **byte**, **char**, **short**, **int**, **long**, **float**, **double**
- **Object**, **String**, **Type** (alle aus dem Namensraum **System**)
- Aufzählungstypen
- Eindimensionale Arrays mit einem Elementtyp aus der obigen Liste

In einer Übungsaufgabe zum aktuellen Kapitel sollen Sie das eben definierte Attribut auf eine Klasse anwenden und zur Laufzeit auswerten (siehe Abschnitt 14.7).

14.7 Übungsaufgaben zum Kapitel 14

1) Ergänzen Sie im Beispiel von Abschnitt 14.6 die ziemlich sinnlose Klasse `Dummy`

```
[Serializable] [NonsenseAttribute(13)]
class Dummy {
}
```

und eine Testklasse mit **Main()** - Methode, die alle benutzerdefinierten Attribute der Klasse `Dummy` und den `Level`-Wert des `NonsenseAttribut`-Objekts ausgibt.

15 Sonstige C# - Sprachbestandteile

In diesem Kapitel werden C# - Sprachbestandteile nachgetragen, die entweder zu keinem bisherigen Kapitel gepasst haben, oder an der thematisch passenden Stelle aufgrund ihrer hohen Komplexität didaktisch gestört hätten.

15.1 Mustervergleiche

Seit der C# - Version 7 wurden Kompetenzen zum Mustervergleich (engl.: *pattern matching*) in die Programmiersprache aufgenommen. In der **switch**-Anweisung wurde die seit C# 1.0 mögliche Detektion von Fällen über konstante Werte (nun als *konstantes Muster* bezeichnet) ergänzt durch die Falldetektion über sogenannte *Typmuster* (siehe Abschnitt 4.7.2.3.3). In den C# - Versionen 8 und 9 wurden die Optionen zum Mustervergleich sukzessive ausgebaut und an verschiedenen Stellen nutzbar gemacht:

- in der **switch**-Anweisung
- im **switch**-Ausdruck
- im **is**-Ausdruck

Microsoft möchte durch die Kompetenzen zum Mustervergleich die Eignung von C# für Aufgabenstellungen, die durch eine partielle Trennung von Daten und Funktionalität gekennzeichnet sind, verbessern:[1]

- Es sind unterschiedliche Typen zu verarbeiten, die keine hierarchischen Beziehungen aufweisen.
- Die zu realisierende Funktionalität gehört nicht in den Verantwortungsbereich der Klassen.

Ein Beispiel ist die Mautberechnung für unterschiedliche Fahrzeugklassen (private PKWs, Taxis, Busse, Lieferwagen) in Abhängigkeit von unterschiedlichen, nicht für alle Klassen gleichen Kriterien (z. B. Anteil der besetzten Plätze, Gewicht).[2]

Bei einer weniger programmiertheoretisch aufgeladenen Beschreibung geht es um Möglichkeiten, Fallunterscheidungen möglichst einfach vorzunehmen. Die neuen Optionen zum Mustervergleich haben es nicht immer leicht, ihre Überlegenheit gegenüber traditionellen Programmiertechniken zu beweisen (z. B. gegenüber bedingten Anweisungen mit geschickt formulierten logischen Ausdrücken).

Wir konzentrieren uns gleich auf die **switch**-Ausdrücke, die der Einsatzschwerpunkt für die neuen Optionen zum Mustervergleich sind. Obwohl auch die **switch**-Anweisung neuerdings den Mustervergleich beherrscht, kommt sie hier seltener zum Einsatz, denn:

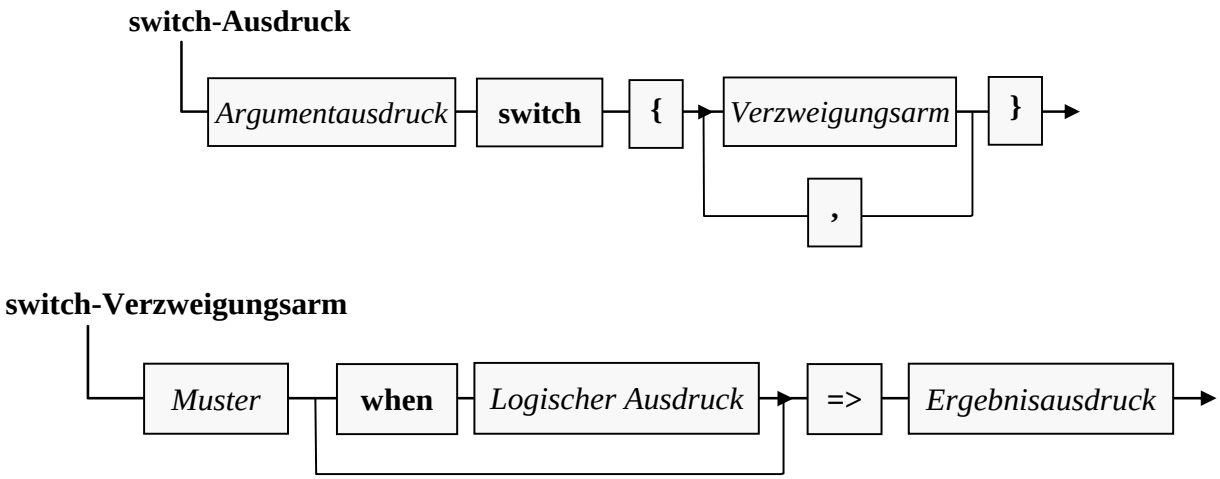
- Meist ist lediglich ein Ergebniswert in Abhängigkeit von der Fallzugehörigkeit zu produzieren.
- Die **switch**-Anweisung erfordert einen höheren syntaktischen Aufwand.

Der schon im Abschnitt 4.7.2.4 in Grundzügen beschriebene **switch**-Ausdruck hat den folgenden Aufbau:

---

[1] Hier geht es keinesfalls um die Rückkehr zur strukturierten Programmierung (vgl. Abschnitt 5.1.2), sondern um die Verarbeitung unterschiedlicher Klassen bei möglichst geringem syntaktischen Aufwand.

[2] https://docs.microsoft.com/de-de/dotnet/csharp/tutorials/pattern-matching



Im Ergebnisausdruck kann auch eine Ausnahme geworfen werden, weil C# seit der Version 7.0 neben der **throw**-Anweisung auch den **throw**-Ausdruck kennt (vgl. Abschnitt 13.5), z. B.:

```
null => throw new ArgumentException(message: "No person found")
```

Die Verzweigungsarme werden in der Definitionsreihenfolge abgearbeitet, und der Compiler verweigert die Übersetzung, wenn ein Verzweigungsarm nie erreicht werden kann.

Anschließend werden die unterstützten Muster beschrieben. Im gesamten Abschnitt 15.1 wird eine C# - Version ab 8 vorausgesetzt, ab Abschnitt 15.1.7 sogar eine Version ab 9.

15.1.1 Konstantenmuster

Das seit C# 1.0 im Rahmen der **switch**-Anweisung erlaubte Konstantenmuster, das aus einem schon zur Übersetzungszeit feststehenden Wert besteht, kann natürlich auch in einem **switch**-Ausdruck verwendet werden. Das folgende Beispiel wurde in ähnlicher Form schon im Abschnitt 4.7.2.4 (mit einer Kurzbeschreibung des in C# 8 eingeführten **switch**-Ausdrucks) vorgestellt:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  int charCode = 3;`
`  String charLabel = charCode switch {`
`  1 => "melancholisch",`
`  2 => "cholerisch",`
`  3 => "phlegmatisch",`
`  4 => "sanguinisch"`
`  };`
`  Console.WriteLine($"Charakter: {charLabel}");`
`  }`
`}`
	`Charakter: phlegmatisch`

15.1.2 Residuale Muster

Mit dem Unterstrich (in der englischen Literatur als *discard pattern* bezeichnet) werden alle bisher unbehandelten Fälle des Argumentausdrucks angesprochen. Er übernimmt im **switch**-Ausdruck offenbar die Rolle des **default**-Falls aus der **switch**-Anweisung. Im folgenden Beispiel liefert ein **switch**-Ausdruck zum Herkunftsland eines Kunden den Mehrwertsteuersatz:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static double Vat(string country) => country switch {`
`  "Deutschland" => 19.0,`
`  "Frankreich" => 20.0,`
`  "Luxemburg" => 17.0,`
`  _ => Double.NaN`
`  };`
 	
`  static void Main() {`
`  string country = "Lichtenstein";`
`  try {`
`  Console.WriteLine("Value added tax: " + Vat(country));`
`  } catch (Exception ex) {`
`  Console.WriteLine(ex.Message);`
`  }`
`  }`
`}`
	`Value added tax: NaN`

Als Alternative zum Unterstrichmuster kommt gelegentlich das **var**-Muster in Frage, das eigentlich ein spezielles Typmuster (vgl. Abschnitt 15.1.3) ist, z. B.:

```
var cvar => throw new ArgumentException("Unknown Country: " + cvar)
```

Weil **var** dank Typinferenz vom Typ des Argumentausdrucks ist, passt das **var**-Muster immer. Es liefert im Vergleich zum Unterstrichmuster eine Variable zur näheren Untersuchung. Diese Variable ist aber nicht unbedingt erforderlich, weil im Ergebnisausdruck auch das Argument verwendet werden kann, das denselben Typ hat wie die mit **var** deklarierte Variable, z. B.:

```
_ => throw new ArgumentException("Unknown Country: " + country)
```

Zum **var**- und zum Unterstrichmuster passt auch der Wert **null**, sodass eventuell eine **NullReferenceException** verhindert werden muss, z. B.:

```
var cvar => cvar != null ?
            throw new ArgumentException("Unknown Country: " + cvar) :
            throw new ArgumentException("No argument")
```

Im Beispiel ist übrigens das seit C# 7.0 mögliche Werfen von Ausnahmen in einem Konditionalausdruck zu sehen.

Die **NullReferenceException** lässt sich auch durch einen separaten Verzweigungsarm mit **null**-Muster abfangen, z. B.:

```
null => throw new ArgumentException("No argument")
```

Schließlich kann man mit dem **{}** - Muster, das eigentlich ein spezielles Eigenschaftsmuster ist (siehe Abschnitt 15.1.4) den Fall behandeln, dass der Argumentausdruck auf ein existentes Objekt zeigt, das aber von keinem vorherigen Verzweigungsarm behandelt wurde, z. B.:

```
{} => throw new ArgumentException("Unknown Country: "+ country)
```

Wenn auf ein residuales Muster verzichtet wird, und der Argumentausdruck einen unbehandelten Wert annimmt, dann wirft die CLR eine Ausnahme vom Typ **SwitchExpressionException** (Namensraum **System.Runtime.CompilerServices**) mit der Message:

```
Non-exhaustive switch expression failed to match its input.
Unmatched value was Lichtenstein.
```

15.1.3 Typmuster

Zum Typmuster gehört eine Typangabe und ein Variablenname. Im folgenden Beispiel wird für Objekte (oder Strukturinstanzen) zum Modellieren von Figuren die Fläche typabhängig bestimmt:

```
static double Flaeche(object figur) {
    return figur switch {
        Rechteck r => r.Breite * r.Hoehe,
        Kreis k => Math.PI * k.Radius * k.Radius,
        _ => 0.0
    };
}
```

Zumindest bei Strukturinstanzen kommt eine Flächenberechnung über die polymorphe Nutzung einer in der gemeinsamen Basisklasse (abstrakt) definierte Methode oder Eigenschaft nicht in Frage, sodass ein sinnvolles Anwendungsbeispiel für das Typmuster gefunden ist.

15.1.4 Merkmalsmuster

Wenn der Argumentausdruck eine Strukturinstanz ist oder auf ein Objekt zeigt, dann kann man über eine Merkmalsausprägung oder über mehrere, durch Komma getrennte Merkmalsausprägungen, die Anforderungen für einen Treffer definieren. Die Liste der geforderten Merkmalsausprägungen, die aus Feldern oder Eigenschaften stammen können, wird zwischen geschweiften Klammern formuliert, z. B.:

```
{ Passagiere: 1, GeradeNummer: true }
```

Es folgt ein Beispiel, das sich mit der Verkehrsregulierung in einer Stadt beschäftigt:

- PKWs mit mindestens 2 Passagieren und einem Kennzeichen mit gerader Nummer dürfen einfahren.
- Ist der Fahrer eines PKWs allein, oder hat das Kennzeichen eine ungerade Nummer, dann ist die Einfahrt verboten.
- LKWs dürfen bei einem Gewicht bis zu 7,5 t einfahren.

Im Beispiel sind die beiden Klassen `PKW` und `LKW` beteiligt, und dabei erweist es sich als Vorteil, dass man das Typ- und das Merkmalsmuster kombinieren darf:

```
using System;
using System.Collections;

class PKW {
    public int Passagiere;
    public bool GeradeNummer;
}

class LKW {
    public double Gewicht;
}
```

```
class Merkmalsmuster {
    static bool Zugelassen(object obj) =>
        obj switch {
            PKW { Passagiere: 1, GeradeNummer: true } => false,
            PKW { GeradeNummer: true } => true,
            PKW { GeradeNummer: false } => false,
            LKW lkw => lkw.Gewicht <= 7.5,
            _ => true
        };

    static void Main() {
        var p1 = new PKW { Passagiere = 1, GeradeNummer = true};
        var p2 = new PKW { Passagiere = 3, GeradeNummer = true};
        var p3 = new PKW { Passagiere = 4, GeradeNummer = false };
        var l1 = new LKW { Gewicht = 7 };
        var l2 = new LKW { Gewicht = 12 };
        var la = new ArrayList { p1, p2, p3, l1, l2 };
        for (int i = 0; i < la.Count; i++)
           Console.WriteLine($"Nr.: {i,2}, Zugelassen: {Zugelassen(la[i])}");
    }
}
```

Das Ergebnis:

```
Nr.:  0, Zugelassen: False
Nr.:  1, Zugelassen: True
Nr.:  2, Zugelassen: False
Nr.:  3, Zugelassen: True
Nr.:  4, Zugelassen: False
```

Besitzt eine Instanz eine Mitgliedsinstanz, dann können Merkmalsmuster geschachtelt werden. Im folgenden Beispiel besitzt die Struktur `Kreis` das Feld `Radius` vom elementaren Typ **double** und das Feld `Zentrum` vom Strukturtyp `Punkt`:

```
struct Punkt {
    internal double X;
    internal double Y;
}

struct Kreis {
    internal double Radius;
    internal Punkt Zentrum;
}
```

Per **is**-Operator wird für eine `Kreis`-Instanz getestet, ob sie den `Radius` 5 und das `Zentrum` (7, 8) besitzt:

```
Console.WriteLine(k is {Radius:5, Zentrum: {X:7, Y:8}});
```

Eine weitere Option des Merkmalsmusters besteht darin, Merkmalsausprägungen in Variablen zu speichern für eine anschließende Untersuchung. Im folgenden Beispiel wird für eine `Kreis`-Instanz getestet, ob der `Radius` gleich 5 ist, und ob das `Zentrum` auf der Hauptdiagonalen liegt:

```
Console.WriteLine(k is { Radius: 5, Zentrum: {X: var x, Y: var y}} && x==y);
```

15.1.5 Tupelmuster

Das Tupelmuster ist als mehrdimensionale Variante des Konstantenmusters dann von Interesse, wenn eine Abbildung vom Kreuzprodukt mehrerer Mengen ($A_1 \times A_2 \times ... \times A_k$) in eine Zielmenge B so komplex ist, dass anstelle der oft möglichen Beschreibung durch einen Ausdruck die Beschreibung durch eine Tabelle die beste Lösung ist. In einem künstlichen Beispiel betrachten wir 3

dichotome Variablen A_1, A_2 und A_3 (mit den Werten 0 und 1) und eine Abbildung in die Menge der booleschen Werte {**true**, **false**}, die ...

- den Wert **true** liefert, wenn zwei benachbarte Variablen denselben Wert (0 oder 1) haben, die dritte Variable jedoch einen abweichenden Wert besitzt,
- in allen anderen Fällen den Wert **false** liefert.

Die Funktionswertetabelle erfordert Fleißarbeit, ist aber übersichtlich:

```
static bool GenauZweiNachbarnGleich(int a1, int a2, int a3) =>
    (a1, a2, a3) switch {
        (0, _, 0) => false,
        (0, 0, 1) => true,
        (0, 1, 1) => true,
        (1, 0, 0) => true,
        (1, _, 1) => false,
        (1, 1, 0) => true
    };
```

Ist in einem Verzweigungsarm ein Tupelelement irrelevant, dann kann es durch einen Unterstrich ersetzt werden.

Funktional äquivalent, weniger Schreibaufwand verursachend, aber vielleicht auch weniger übersichtlich ist die folgende Lösung mit einem logischen Ausdruck:[1]

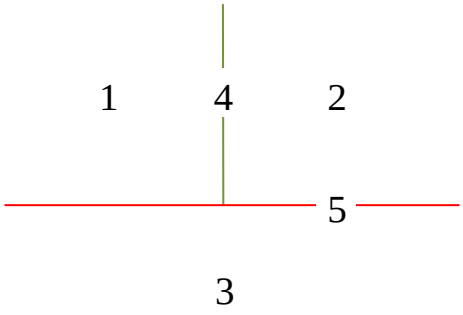
```
public static bool GenauZweiNachbarnGleich(int a1, int a2, int a3) =>
    a2 == a1 != (a2 == a3);
```

15.1.6 Positions- bzw. Dekonstruktionsmuster

Besitzt ein Typ eine **Deconstruct**-Methode (vgl. Abschnitt 6.6.6), dann lässt sich deren Tupel-Produktion als Eingabe für einen **switch**-Ausdruck verwenden. Zur Definition eines Verzweigungsarms können an den einzelnen Positionen des Tupels ...

- bestimmte Werte gefordert,
- Variablen zur näheren Beurteilung in einer **when**-Klausel abgelesen,
- oder durch die Verwendung des Unterstrichs beliebige Werte zugelassen werden.

Als Beispiel betrachten wir eine Klassifikation von Punkten der Zahlenebene:



Instanzen der folgenden `Punkt`-Struktur sollen einer von 5 Kategorien zugeordnet werden:

---

[1] Mathematiker werden vermutlich die Darstellung der Abbildung durch einen logischen Ausdruck gegenüber der Tupelmuster-Tabelle bevorzugen. Die genial kurze Lösung stammt von Paul Frischknecht, einem exzellenten Vertreter dieser Profession.

```
readonly struct Punkt {
    public readonly double X, Y;
    public Punkt(double x, double y) {
        X = x; Y = y;
    }
    public void Deconstruct(out double x, out double y) {
        x = X;
        y = Y;
    }
}
```

Im folgenden Programm

```
using System;
using System.Collections.Generic;

class Positionsmuster {
    public static int Zone(Punkt punkt) =>
         punkt switch {
            var (x, y) when x < 0 && y > 0 => 1,
            var (x, y) when x > 0 && y > 0 => 2,
            (_, var y) when y < 0 => 3,
            (0, var y) when y > 0 => 4,
            (_, 0) => 5
        };

    static void Main() {
        var p1 = new Punkt(-2, 2); var p2 = new Punkt( 2, 2);
        var p3 = new Punkt(-2,-2); var p4 = new Punkt( 0, 2);
        var p5 = new Punkt(-2, 0);
        var pl = new List<Punkt> { p1, p2, p3, p4, p5 };
        for (int i = 0; i < pl.Count; i++)
           Console.WriteLine($"({pl[i].X,2},{pl[i].Y,2}): Zone {Zone(pl[i])}");
    }
}
```

werden einige Punkte klassifiziert:

```
(-2, 2): Zone 1
( 2, 2): Zone 2
(-2,-2): Zone 3
( 0, 2): Zone 4
(-2, 0): Zone 5
```

15.1.7 Relationsmuster

Das seit C# 9 vorhandene Relationsmuster erlaubt übersichtliche Fallunterscheidungen durch die Verwendung der relationalen Operatoren **<**, **<=**, **>** und **>=**. Dieses Muster kann in **switch**-Ausdrücken oft die seit C# 7 verfügbare **when**-Klausel ersetzen. Im folgenden Beispiel wird für LKWs eine gewichtsabhängige Maut berechnet:

```
using System;

readonly struct LKW {
    public double Gewicht { get; }
    public LKW(double gew) => Gewicht = gew;
}
```

```
class Relationsmuster {
    public static decimal Maut(LKW lkw) =>
        lkw.Gewicht switch {
            <= 1000 => 10.0m,
            <= 2000 => 18.5m,
            <= 7500 => 25.2m,
            _       => 30.8m
        };

    static void Main() {
        Console.WriteLine("Maut: " + Maut(new LKW(1500)));
        Console.WriteLine("Maut: " + Maut(new LKW(8000)));
    }
}
```

15.1.8 Musterkombinatoren

Seit C# 9 ist es möglich, die Prüfergebnisse mehrerer Muster durch sogenannte *Kombinatoren* zusammenzuführen, wobei die neuen Schlüsselwörter **and**, **or** und **not** zu verwenden sind und Argumente durch runde Klammern zusammengefasst werden können. Im **is**-Ausdruck (siehe Abschnitt 7.6) des folgenden Programms werden mehrere Relationsmuster und ein Konstantenmuster kombiniert:

Quellcode	Ausgabe

`using System;`
`class Kombinatoren {`
`  static void Main() {`
`  char c = 'ä';`
`  Console.WriteLine(c is (>= 'a' and <= 'z' or 'ü' or 'ö' or 'ä')`
`  and not 'x');`
`  }`
`}`
	`True`

Es wird getestet, ob die lokale **char**-Variable `c` einen von ‚x' verschiedenen Kleinbuchstaben enthält.

15.2 Null-bedingter Operator

Vor dem Zugriff auf ein Instanzmitglied (z. B. Methode, Feld oder Array-Element) sollte die Existenz der Instanz zur Vermeidung von **NullReferenceException**-Laufzeitfehlern geprüft werden, was den Quellcode durch häufig auftretende Routine-Passagen belastet. Im folgenden Beispiel soll aus einem **String**-Array die Länge des Elements mit dem Index 1 ermittelt werden, was eine doppelte Existenzprüfung erfordert:

```
string[] ass = null;
 . . .
int len1 = (ass != null && ass[1] != null) ? ass[1].Length : -1;
Console.WriteLine("Länge: " + len1);
```

Der in C# 6.0 eingeführte **Null-bedingte Operator** (engl.: *null-conditional operator*) erleichtert den Instanzzugriff mit vorheriger Existenzprüfung. Für den Member- bzw. Indexzugriff sind zwei leicht verschiedene Syntaxvarianten zu unterscheiden:

- *objekt***?.***member*
  Als Datentyp hat der Ausdruck den Null-erweiterten Membertyp. Wenn das Objekt nicht existiert, liefert der Ausdruck den Wert **null** (Referenz auf Nichts bei einer Klasse oder undefinierte Ausprägung bei einem Werttyp). Im folgenden Beispiel resultiert der Typ **int?** (vgl. Abschnitt 8.3):
  ```
  string s = null;
  int? sl = s?.Length;
  ```
- *objekt***?[***index***]**
  Als Datentyp hat der Ausdruck den Null-erweiterten Elementtyp. Wenn das Objekt nicht existiert, liefert der Ausdruck den Wert **null** (Referenz auf Nichts bei einer Klasse oder undefinierte Ausprägung bei einem Werttyp). Im ersten Beispiel resultiert der Typ **String**:
  ```
  string[] ass = null;
  string ass1 = ass?[1];
  ```
  Im zweiten Beispiel resultiert der Typ **Nullable<int>**:
  ```
  int[] ai = null;
  int? ai1 = ai?[1];
  ```

Ein Null-bedingter Zugriff unterscheidet sich vom normalen Member- bzw. Indexzugriff nur im Fall einer gescheiterten Existenzprüfung: Dann wird keine **NullReferenceException** geworfen, sondern der Wert **null** abgeliefert. Der linke Operand des Null-bedingten Operators muss einen **null**-fähigen Typ haben, also einen Referenztyp oder einen Strukturtyp in **Nullable<T>** - Verpackung (vgl. Abschnitt 8.3).

Das Einstiegsbeispiel lässt sich mit zwei Null-bedingten Operatoren kürzer formulieren:

```
string[] ass = null;
. . .
int len1 = ass?[1]?.Length ?? -1;
Console.WriteLine("Länge: " + len1);
```

Damit ein Ergebnis mit Werttyp resultiert, kommt im Beispiel zusätzlich der Null-Sammeloperator zum Einsatz (vgl. Abschnitt 8.3).

Bei einer Sequenz von Null-bedingten Operatoren findet eine Kurzschlussauswertung statt: Die Auswertung wird abgebrochen, sobald eine Existenzprüfung zum negativen Ergebnis geführt hat.

Auch beim Aufruf eines Delegatenobjekts (vgl. Kapitel 10) kann man sich mit dem Null-bedingten Operator explizite Existenzprüfungen ersparen, z. B.:

```
delegate void DemoGate(int w);
 . . .
DemoGate demoVar = null;
demoVar?.Invoke(2);
```

Dabei ist ein expliziter **Invoke()** - Aufruf erforderlich (vgl. Abschnitt 10.1).[1]

15.3 Deklarierte null - (Un)zulässigkeit von Referenztypen

Bei manchen jüngeren Programmiersprachen wird die Vermeidung von **null**-Referenz - Laufzeitfehlern als wesentlicher Vorzug herausgestellt (z. B. bei Kotlin). C# ermutigt und unterstützt es seit der Version 8.0, im Quellcode bei der Deklaration einer Referenzvariablen festzulegen, ob für diese Variable der Wert **null** erlaubt sein soll oder nicht.

---

[1] https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/operators/null-conditional-operators

Die wesentliche Neuerung sind Referenzvariablen (mit einer Klasse oder Schnittstelle als Datentyp), die den Wert **null** *nicht* annehmen sollen bzw. können (siehe unten). Während bei Werttypen per **Nullable<T>** - Verpackung die **null**-Zulässigkeit nachgerüstet werden kann (siehe Abschnitt 8.3), sind nun Referenztypen im Angebot, denen die seit C# 1.0 selbstverständliche **null**-Zulässigkeit fehlt. Insgesamt steigt die Flexibilität für Programmierer, die sich nun situationsangemessen bei Referenz- und bei Werttypen für oder gegen die **null**-Zulässigkeit bzw. -Fähigkeit entscheiden können.

Bei Referenzvariablen mit **null**-Unzulässigkeit sollen *Warnungen* des Compiler dafür sorgen, dass ...

- beim Laden einer Klasse bzw. bei der Erstellung eines Objekts die Felder einen von **null** verschiedenen Wert erhalten (im Rahmen der Deklaration oder per Konstruktor),
- lokale Variablen bei der Initialisierung einen von **null** verschiedenen Wert erhalten,
- einer Variablen kein Wert zugewiesen wird, der gleich **null** sein könnte.

Man darf sich aber nicht darauf verlassen, dass durch Referenzvariablen mit deklarierter **null**-Unzulässigkeit und aufgrund der Detektionsfähigkeiten des Compilers **NullReferenceException** - Probleme ausgeschlossen sind. Schon bei Array-Elementen mit Referenztyp traut sich der Compiler keine sichere Diagnose zu und verzichtet trotz deklarierter **null**-Unzulässigkeit auf eine Warnung, z. B.:

```
var sar = new string[3];
Console.WriteLine(sar[1].Length);
```

Es ist nicht auszuschließen, dass bei Referenzvariablen mit deklarierter **null**-Unzulässigkeit durch reduzierte Sorgfalt im falschen Vertrauen auf die Warnungen des Compilers in manchen Projekten am Ende mehr **NullReferenceException** - Probleme auftreten.

Befindet sich eine Referenzvariablendeklaration in einem sogenannten *Anmerkungskontext*, dann ist der Wert **null** per Voreinstellung als unzulässig deklariert. Um den Wert **null** zu erlauben, setzt man hinter die Typbezeichnung ein Fragezeichen, z. B.:

```
string? s;
```

Dieselbe Syntax wird auch zur Deklaration von **null**-fähigen Werttypen verwendet (siehe Abschnitt 8.3).

Wie man den Anmerkungskontext für ein ganzes Projekt oder ein Quellcodesegment aktiviert, ist gleich zu erfahren. Bemerkt der Compiler bei aktiviertem Anmerkungskontext im Quellcode einen Verstoß gegen die Nullable-Deklaration, dann reagiert er mit Warnungen, wenn für das betroffene Quellcodesegment auch der sogenannte *Warnungskontext* aktiviert ist. Über eine Projektkonfiguration kann man dafür sorgen, dass bestimmte Warnungen das Gewicht von Fehlern erhalten und somit die Übersetzung des Quellcodes verhindern (siehe unten).

Die Behandlung eines Quellcodesegments durch den C# 8 - Compiler hängt also davon ab, ob der Nullable-Annotations- und/oder der Nullable-Warnungskontext (de)aktiviert ist.

Ist der Anmerkungskontext ...

- aktiviert, dann besteht für jede deklarierte Referenzvariable die **null**-Unzulässigkeit, sofern nicht in der Deklaration durch die **?**-Annotation des Typbezeichners der Wert **null** erlaubt wird.
- deaktiviert, dann besteht für jede deklarierte Referenzvariable die **null**-Zulässigkeit, und der Compiler kritisiert bei aktiviertem Warnungskontext die Verwendung der **?**-Annotation, z. B.:

```
string? s;
```

Zur Laufzeit gibt es *keinen* Unterschied zwischen Referenzdatentypen mit bzw. ohne **null**-Zulässigkeit, also z. B. zwischen den Datentypen **string?** und **string**. Demgegenüber verhält sich z. B. der **null**-fähige Werttyp **int?** erheblich anders als der zugrundeliegende Werttyp **int**.

Ist der Warnungskontext aktiviert, dann warnt der Compiler ...

- bei aktiviertem Anmerkungskontext z. B., wenn ...
  - ein Feld mit **null**-Verbot keinen von **null** verschiedenen Initialisierungswert erhält,
  - einer Variablen mit **null**-Verbot ein Wert zugewiesen wird, der nicht sicher von **null** verschieden ist,
  - für eine beliebige Referenzvariable (unabhängig vom **null**-Verbot) eine unsichere Dereferenzierung vorgenommen wird.

```
#nullable enable annotations
#nullable enable warnings
using System;
class Prog {
    string s;
    static void Main() {
        Prog p = new Prog();
        string? sl = null;
        p.s = sl;
        int len = p.s.Length;
        Console.WriteLine(len);
    }
}
```

- bei deaktiviertem Anmerkungskontext z. B., wenn ...
  - ein Datentyp eine **?**-Annotation erhält,
  - eine unsichere Dereferenzierung vorgenommen wird.

```
#nullable disable annotations
#nullable enable warnings
using System;
class Prog {
    string s;
    static void Main() {
        Prog p = new Prog();
        string? sl = null;
        p.s = sl;
        int len = p.s.Length;
        Console.WriteLine(len);
    }
}
```

Eine Warnung vor einer unsicheren Dereferenzierung kann mit dem durch ein Ausrufezeichen notierten **null**-Toleranz - Operator verhindert werden, z. B.:

```
int len = p.s!.Length;
```

Weil der Compiler eher zu selten als zu oft warnt, ist der **null**-Toleranz - Operator wohl kaum jemals sinnvoll.

Die Nullable-Kontexte (Anmerkung bzw. Warnung) lassen sich unabhängig voneinander ein- und ausschalten, wobei zwei alternative Einstellungsoptionen bestehen:[1]

- Das Element **Nullable** in der Projektkonfigurationsdatei (Namenserweiterung **.csproj**) nimmt eine Einstellung mit Gültigkeit für das gesamte Projekt vor, z. B.:

```
<PropertyGroup>
  <Nullable>enable</Nullable>
  . . .
</PropertyGroup>
```

[1] https://docs.microsoft.com/de-de/dotnet/csharp/nullable-references

- Das Präprozessor-Kommando **#nullable** nimmt eine Einstellung für die folgenden Quellcodezeilen vor, z. B.:
  ```
  #nullable enable warnings
  ```
  Es dominiert ggf. die Projekteinstellung und bleibt gültig bis zu einem abweichenden **#nullable** - Kommando.

Erlaubte Werte im **Nullable**-Element mit der Projekteinstellung:

- **enable**
  Der Annotationskontext und der Warnungskontext sind aktiviert.
- **disable**
  Der Annotationskontext und der Warnungskontext sind deaktiviert.
- **annotations**
  Der Annotationskontext ist aktiviert, und der Warnungskontext ist deaktiviert.
- **warnings**
  Der Annotationskontext ist deaktiviert, und der Warnungskontext ist aktiviert.

Das Präprozessor-Kommando **#nullable** kennt die folgenden Varianten:

- **#nullable enable**
  Der Annotationskontext und der Warnungskontext sind aktiviert.
- **#nullable disable**
  Der Annotationskontext und der Warnungskontext sind deaktiviert.
- **#nullable restore**
  Die Projekteinstellung (im **Nullable**-Element in der Projektkonfigurationsdatei) wird restauriert.
- **#nullable enable annotations,**
  #nullable disable annotations
  #nullable restore annotations
  Die Annotationen werden aktiviert, deaktiviert oder auf die Projekteinstellung gesetzt.
- **#nullable enable warnings,**
  #nullable disable warnings
  #nullable restore warnings
  Die Warnungen werden aktiviert, deaktiviert oder auf die Projekteinstellung gesetzt.

Per Voreinstellung sind der Anmerkungs- und der Warnungskontext abgeschaltet, was dem Zustand vor C# 8.0 entspricht.

Über das Element **WarningsAsErrors** in der Projektkonfigurationsdatei kann man aber dafür sorgen, dass bestimmte Warnungen das Gewicht von Fehlern erhalten und somit die Übersetzung des Quellcodes verhindern (Albahari & Johannsen 2020, S. 193), z. B.:

```
<PropertyGroup>
  <Nullable>enable</Nullable>
  <WarningsAsErrors>CS8601;CS8602;CS8618</WarningsAsErrors>
  . . .
</PropertyGroup>
```

Nun können die vom Compiler erkannten **null**-Risiken nicht mehr ignoriert werden:

```
using System;
class Prog {
    string s;
    static void Main() {
        Prog p = new Prog();
        string? sl = null;
        p.s = sl;
        int len = p.s.Length;
        Console.WriteLine(len);
    }
}
```

15.4 Der nameof-Operator

Der in C# 6.0 eingeführte **nameof**-Operator ist nützlich, wenn im Quellcode der Name eines Typs oder Members benötigt wird, z. B. der Name einer Eigenschaft im Konstruktor der Klasse **PropertyChangedEventArgs**:

public PropertyChangedEventArgs (String *propertyName***)**

Dieser Konstruktor ist beteiligt, wenn ein Ereignis mit dem Delegatentyp **PropertyChanged-EventHandler** aufgerufen wird, um registrierte Wertveränderungsinteressenten zu informieren. Diese erfahren über den zweiten **Invoke()** - Parameter, welche Eigenschaft geändert wurde, z. B.:

```
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Einkommen"));
```

Den Namen der Eigenschaft als Zeichenfolgenliteral anzugeben, ist wenig sinnvoll:

- Der Compiler kann Tippfehler nicht verhindern.
- Wenn sich der Name ändert, werden Zeichenfolgenliterale bei der Refaktorierung durch das Visual Studio nicht berücksichtigt.

Mit Hilfe des **nameof**-Operators lässt sich das Zeichenkettenliteral vermeiden:

```
new PropertyChangedEventArgs(nameof(Einkommen))
```

Wenn im Beispiel das Ereignis `Einkommen` per Refaktorierung umbenannt wird in `Income`, wird der Konstruktoraufruf automatisch aktualisiert:

```
new PropertyChangedEventArgs(nameof(Income))
```

Auch bei einer Fehlerbeschreibung (z. B. im **Message**-Parameter einer Ausnahme) kann man per **nameof**-Operator die beschriebenen Vorteile nutzen, z. B.:

```
new ArgumentException(string.Format(
    $"Falscher Wert {value} bei Eigenschaft {nameof(Einkommen)}"))
```

Die zur Demonstration verwendeten Anweisungen stammen aus einer Klasse namens `Person`, welche die Schnittstelle **INotifyPropertyChanged** implementiert und infolgedessen ein Ereignis namens **PropertyChanged** anbietet:[1]

[1] Die Anregung zu diesem Beispiel stammt von der Microsoft-Webseite:
https://docs.microsoft.com/de-de/dotnet/framework/wpf/data/how-to-implement-property-change-notification

```
using System;
using System.ComponentModel;
using System.Windows;

public class Person : INotifyPropertyChanged {
   public event PropertyChangedEventHandler PropertyChanged;
   decimal income;
   public decimal Income {
      get { return income; }
      set { if (value >= 0.0m) {
            income = value;
            PropertyChanged?.Invoke(this,new PropertyChangedEventArgs(nameof(Income)));
         } else
            throw new ArgumentException(string.Format(
                        $"Falscher Wert {value} bei Eigenschaft {nameof(Income)}"));
      }
   }

   public string FirstName { get; set; }
   public string LastName { get; set; }

   public Person(string fn, string ln, decimal income) {
      FirstName = fn; LastName = ln; Income = income;
   }
}
```

15.5 ref-Variablen und -Rückgabewerte

In diesem Abschnitt geht es um Optionen für die Definition von Methoden, die mit C# 7.0 zur Verbesserung der Sicherheit und Leistung eingeführt wurden. Für Einsteiger sind diese C# - Spracherweiterungen allerdings weniger zu empfehlen.

15.5.1 ref-Variablen in Methoden

Im Abschnitt 15.5 geht es primär um Methoden mit einem Verweis auf eine Strukturinstanz als Rückgabe. Weil eine solche Rückgabe oft in einer lokalen Variablen des Aufrufers abgelegt wird, beschäftigen wir uns zunächst mit **ref**-Variablen. Das sind ...

- lokale Variablen,
- die eine Referenz auf eine im aktuellen Kontext sichtbare Variable aufnehmen können.

Im Unterschied zu den bisher behandelten Referenzvariablen muss es sich beim Referenzziel nicht um ein komplettes Objekt auf dem Heap handeln. Alternative Referenzziele sind z. B.

- ein einzelnes Feld
- ein Array-Element
- eine Strukturinstanz auf dem Stack

Durch eine **ref**-Variable sind z. B. die folgenden Vorteile zu realisieren:

- Wenn nur eine einzelne Instanzvariable eines Objekts angesprochen werden kann, dann ist das Fehlerrisiko reduziert.
- Durch das Referenzieren von Strukturinstanzen können CPU-Zeit und Speicher belastende Kopiervorgänge eingespart werden (analog zu einem Methodenparameter mit **ref**-Modifikator, vgl. Abschnitt 5.3.1.3.2.1).

In einer Trockenübung wird zunächst demonstriert, dass mit dem Schlüsselwort **ref** eine lokale Variable deklariert werden kann, die auf ein Array-Element im selben Gültigkeitsbereich zeigt:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  int[] iar = { 1, 2, 3 };`
`  ref int loref = ref iar[2];`
`  loref = 4;`
`  Console.WriteLine(iar[2]);`
`  }`
`}`
	`4`

Das Schlüsselwort **ref** muss in der Variablendeklaration dem Datentyp *und* außerdem dem Referenzziel vorangestellt werden.

Es ist verboten, eine **ref**-Variable ohne Initialisierung zu deklarieren. Seit C# 7.3 ist es erlaubt, das Verweisziel einer **ref**-Variablen zu ändern:

```
loref = ref iar[0];
```

Seit C# 7.2 kann man eine lokale **ref**-Variable als **readonly** deklarieren, sodass nur lesende Zugriffe auf das Referenzziel möglich sind, z. B.:

```
int[] iar = { 1, 2, 3 };
ref readonly int loref = ref iar[2];
```

15.5.2 Methoden mit ref-Rückgabewert

Meist wird eine **ref**-Variable dazu verwendet, die von einer Methode als **ref**-Rückgabewert gelieferte Referenz aufzubewahren. Im folgenden Beispiel werden Instanzen der (als unveränderlich deklarierten) Struktur `Strecke`

```
readonly public struct Strecke {
    public int Li { get; }
    public int Re { get; }
    public Strecke(int li, int re) {
        Li = li; Re = re;
    }
}
```

durch eine Klasse namens `StreckenSammlung` verwaltet. In einem internen Array befinden sich 10 Strecken mit einem zufällig bestimmten Startpunkt im Intervall von 0 bis 95 und der Länge 5:

```
class StreckenSammlung {
    const int anz = 10, max = 100;
    readonly Strecke[] strecken;
    static Strecke nf = new Strecke(-1, -1);

    internal StreckenSammlung() {
        strecken = new Strecke[anz];
        Random zzg = new Random();
        for (int i = 0; i < anz; i++) {
            int start = zzg.Next(max - 4);
            strecken[i] = new Strecke(start, start + 5);
        }
    }
```

```
    internal ref Strecke FindeStrecke(int x) {
        for (int i = 0; i < anz; i++)
            if (x >= strecken[i].Li && x <= strecken[i].Re)
                return ref strecken[i];
        return ref nf;
    }
}
```

Die Methode `FindeStrecke()` sucht für ihr Argument die erste enthaltende Strecke und liefert diese als **ref**-Rückgabewert. Dazu muss das Schlüsselwort **ref** ...

- im Definitionskopf vor den Rückgabetyp
- und in jeder **return**-Anweisung vor den Rückgabewert

gesetzt werden.

Weitere Regeln:

- Die Gültigkeit der **ref**-Rückgabe muss die Ausführung der Methode überdauern. Das ist z. B. der Fall, wenn die Adresse eines Array-Elements geliefert wird.
- Eine Verwendung im Zusammenhang mit asynchronen Methoden ist *nicht* möglich (vgl. Kapitel 17).

In der Startmethode der folgenden Klasse wird einer **ref**-Variablen die **ref**-Rückgabe zugewiesen, wobei das Schlüsselwort **ref** in der Variablendeklaration dem Datentyp *und* dem Referenzziel vorangestellt werden muss:

```
using System;
class Prog {
    static StreckenSammlung strKoll = new StreckenSammlung();
    static void Main() {
        ref Strecke strecke = ref strKoll.FindeStrecke(12);
        if (strecke.Li != -1)
            Console.WriteLine("Treffer in (" + strecke.Li + ", " + strecke.Re + ")");
        else
            Console.WriteLine("Kein Treffer");
    }
}
```

Wenn der Aufrufer als Rückgabe keine *Referenz* auf das Original, sondern eine *Kopie* erhalten soll, lässt man in der initialisierenden Variablendeklaration die beiden **ref**-Schlüsselwörter weg, z. B.:

```
Strecke strecke = strKoll.FindeStrecke(12);
```

Seit C# 7.2 kann man eine **ref**-Rückgabe als **readonly** deklarieren, sodass nur lesende Zugriffe auf das Referenzziel möglich sind, z. B.:

```
static ref readonly int FindFirst(int k) {
    for (int i = 1; i < iar.Length; i++)
        if (iar[i] == k)
            return ref iar[i];
    return ref iar[0];
}
```

Damit einer lokalen **ref**-Variablen die **ref readonly** - Rückgabe zugewiesen werden kann, muss diese ebenfalls als **ref readonly** deklariert werden, z. B.:

```
ref readonly int lore = ref FindFirst(15);
```

15.5.3 ref-Rückgabe durch den Konditionaloperator

Seit C# 7.2 kann der Konditionaloperator eine **ref**-Rückgabe liefern. Im folgenden Beispiel wird in Abhängigkeit von einer Bedingung eine Referenz auf das erste oder zweite Element eines Arrays geliefert:

```
ref var r = ref (ar[1] % 2 == 0 ? ref ar[0] : ref ar[1]);
```

15.6 ref - Strukturen

Seit C# 7.2 kann bei der Definition einer Struktur durch den Modifikator **ref** verhindert werden, dass Instanzen der Struktur auf den Heap gelangen (z. B. per Boxing oder als Array-Elemente). Diese Verwendung des **ref**-Modifikators für eine eigene Struktur kommt vor allem dann in Frage, wenn dort eine Instanz mit BCL-Strukturtyp **Span<T>** zum Einsatz kommt. Dieser Typ wird in leistungskritischen Programmen dazu verwendet, um einen zusammenhängenden Speicherbereich auf dem Stack anzusprechen, und ist dementsprechend mit dem **ref**-Modifikator definiert.[1] Zu den zahlreichen Einschränkungen von **ref**-Strukturen gehört, dass sie in einer Klasse oder in einer normalen Struktur (ohne **ref**-Modifikator) nicht als Instanzvariablentyp erlaubt sind.[2] Folglich muss eine eigene Struktur bei der Verwendung einer Instanzvariablen vom Typ **Span<T>** ebenfalls den Modifikator **ref** erhalten.

15.7 Übungsaufgaben zum Kapitel 15

1) In einer Stadt werden nur PKWs zugelassen, die entweder mit mindestens drei Personen besetzt oder elektrisch angetrieben sind. Verwenden Sie z. B. die (wenig vorbildliche) Klasse

```
public class PKW {
    public int Passagiere;
    public bool Elektromobil;
    public bool Zugelassen() { ... }
}
```

und implementieren Sie die Methode `Zugelassen()`.

2) Auch im Umfeld des Null-bedingten Operators muss die Auswertungsreihenfolge von Ausdrücken (vgl. Abschnitt 4.5.10) eindeutig geklärt werden. Wird in der dritten Zeile des folgenden Codesegments der Ausdruck rechts vom Identitätsoperator ausgeführt, wenn entweder die Variable `ass` ins Leere zeigt, oder das Element 1 des **String**-Arrays gleich **null** ist?

```
string[] ass = . . .
int res = 0;
bool sl1ok = ass?[1]?.Length == ++res;
```

---

1 https://docs.microsoft.com/de-de/dotnet/csharp/write-safe-efficient-code

2 https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/builtin-types/struct

16 Dateiverarbeitung

Praktisch jedes Programm muss Daten aus externen Quellen einlesen und/oder Verarbeitungsergebnisse in externe Senken schreiben. Wir haben uns bisher auf die Eingabe per Tastatur sowie die Ausgabe auf den Bildschirm beschränkt und müssen allmählich alternative Quellen bzw. Senken kennenlernen (z. B. Dateien, Netzwerkverbindungen, Datenbankserver). Im aktuellen Kapitel werden auf der **Datenstrom**-Abstraktion basierende Verfahren vorgestellt, mit denen Werte elementarer Typen (z. B. **byte**, **int**, **double**), Zeichenfolgen oder beliebige Objekte in **Dateien** geschrieben bzw. von dort gelesen werden können. Außerdem werden wir uns mit der Verwaltung von Dateien und Verzeichnissen beschäftigen.

Mit dem Datenstromkonzept wird bezweckt, Anweisungen zur Ein- oder Ausgabe von Daten möglichst unabhängig von den Besonderheiten konkreter Datenquellen oder -senken formulieren zu können. Es lassen sich z. B. Klassen erstellen, die über Ströme kommunizieren, und in konkreten Anwendungen alternativ mit Dateien oder Netzverbindungen zusammenarbeiten.

Vorausblick auf zwei verwandte Themen:

- In einem späteren Kapitel werden Sie mit den **Netzwerkverbindungen** weitere, außerordentlich wichtige Datenquellen bzw. -senken kennenlernen und dabei von Ihren Kenntnissen über die Datenstromtechnik profitieren.
- Im Kapitel über **Datenbankprogrammierung** werden anspruchsvolle Datenverwaltungstechniken vorgestellt, die sich auch im Netzwerk- und Mehrbenutzerkontext bewähren. Dabei überlassen wir den direkten Kontakt mit Dateien einer speziellen Software, dem Datenbankmanagement-System (DBMS).

Die .NET - Klassen zur Datenein- und -ausgabe befinden sich im Namensraum **System.IO**, den wir folglich bei Quellcodedateien mit entsprechender Funktionalität in der Regel zu Beginn importieren:

```
using System.IO;
```

Die in den Abschnitten 16.1 und 16.3 beschriebenen Methoden zum Schreiben und Lesen durch die direkte Verwendung von Datenstromklassen bietet eine hohe Flexibilität, die speziell bei der Verarbeitung von *kleinen Dateien* oft überflüssig ist. In dieser Situation sollten zunächst die im Abschnitt 16.4 beschriebenen statischen Methoden der Klasse **File** in Betracht gezogen werden. Diese Methoden setzen im Hintergrund Datenstromklassen ein und vereinfachen deren Verwendung.

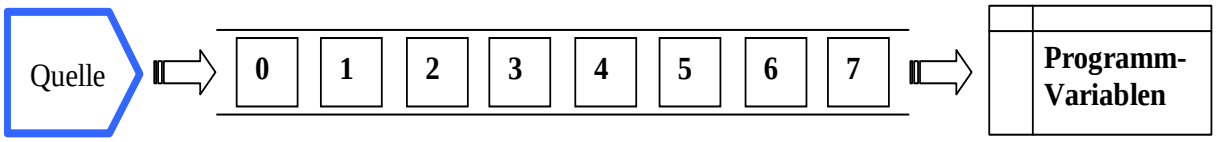
Im aktuellen Kapitel werden wir nur *synchrone* Schreib- und Lesemethoden verwenden, die den aktuellen Thread bis zur Beendigung eines Methodenaufrufs blockieren. Asynchrone Schreib- und Lesemethoden werden (leider nur kurz) im Kapitel 17 behandelt.

16.1 Datenströme aus Bytes

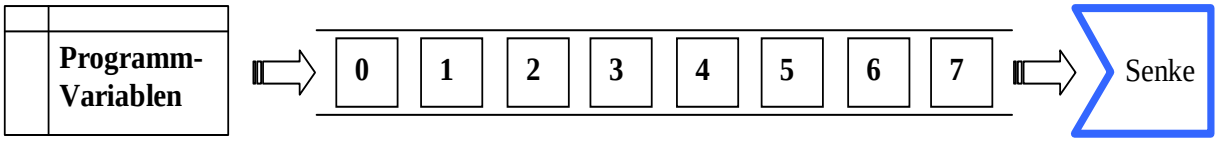
Alle Datentypen sind schlussendlich aus Bytes zusammengesetzt, und moderne Computer-Hardware ist so konstruiert, dass Bytes effizient verarbeitet werden können. Daher stützt sich ein beliebiger Datenstrom letztlich auf einen Strom aus Bytes, und wir befassen uns zunächst mit diesem Basisstrom.

16.1.1 Das Grundprinzip

Ein Programm **liest** Daten aus einem **Eingabestrom**, der aus einer Datenquelle (z. B. Datei, Eingabegerät, Netzwerkverbindung) gespeist wird:



Ein Programm **schreibt** Daten in einen **Ausgabestrom**, der die Werte von Programmvariablen zu einer Datensenke befördert (z. B. Datei, Ausgabegerät, Netzverbindung):



Ein- bzw. Ausgabeströme werden in .NET - Programmen durch Objekte aus Klassen im Namensraum **System.IO** repräsentiert, wobei die Auswahl u. a. von der angeschlossenen Datenquelle bzw. -senke (z. B. Datei versus Netzverbindung) sowie vom Typ der zu transportierenden Daten abhängt.

16.1.2 Beispiel

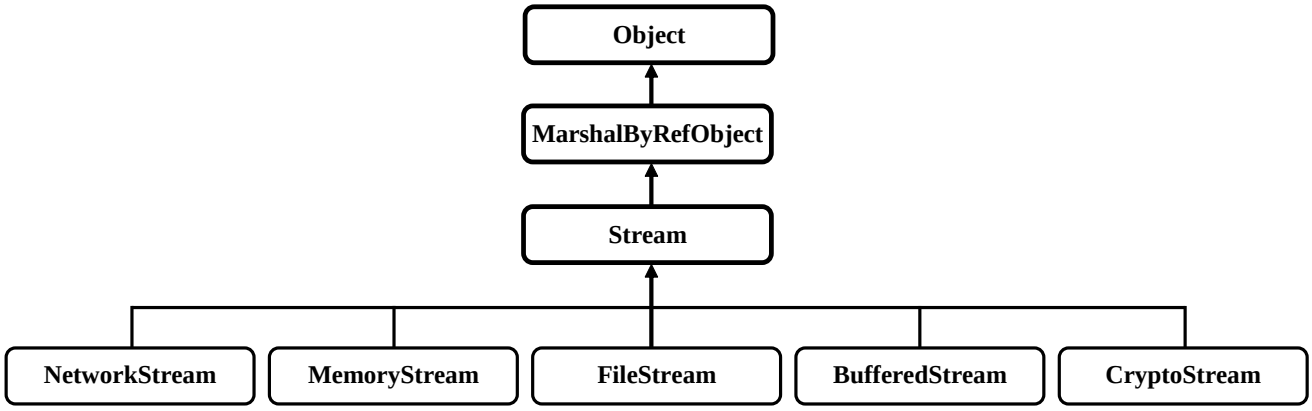
Nach so vielen allgemeinen bzw. abstrakten Bemerkungen wird es Zeit für ein konkretes Beispiel, wobei der Einfachheit halber auf Praxisnähe verzichtet wird. Das folgende Programm erstellt eine Datei namens **demo.bin**, schreibt einen **byte**-Array hinein, liest die Daten wieder zurück und löscht schließlich die Datei. In der Praxis kommt das Schreiben und Lesen von **byte**-Arrays durchaus vor, weil es z. B. Gründe gibt, ein Bild (ein Objekt der Klasse **Image**) als **byte**-Array zu behandeln. Über den Dateieröffnungsmodus **FileMode.CreateNew** (vgl. Abschnitt 16.1.4.1) wird dafür gesorgt, dass die Datei **demo.bin** neu anzulegen ist. Existiert im aktuellen Verzeichnis bereits eine Datei mit diesem Namen, dann endet das Programm mit einem Ausnahmefehler.

Quellcode	Ausgabe

```
using System;
using System.IO;

class FSDemo {
    static void Main() {
        string name = "demo.bin";
        byte[] arr = {0,1,2,3,4,5,6,7};
        using (FileStream fs = new(name, FileMode.CreateNew)) {
            fs.Write(arr, 0, arr.Length);
            fs.Position = 0;
            fs.Read(arr, 0, arr.Length);
            foreach(byte b in arr)
                Console.WriteLine(b);
        }
        File.Delete(name);
    }
}
```

```
0
1
2
3
4
5
6
7
```

Für die Ein- und die Ausgabe wird ein Objekt der Klasse **FileStream** eingesetzt. Diese Spezialisierung der abstrakten Basisklasse **Stream** implementiert das Stromkonzept für *Dateien*. Zur Einordnung ist hier ein kleiner Ausschnitt aus der **Stream**-Klassenhierarchie zu sehen:



Mit den Details des Beispielprogramms beschäftigen wir uns gleich.

16.1.3 Wichtige Methoden und Eigenschaften der Basisklasse Stream

Alle Ableitungen der abstrakten Basisklasse **Stream** verfügen u. a. über die folgenden Methoden und Eigenschaften:

- **public long Position {get; set;}**
  Über diese Eigenschaft wird die aktuelle Position im Strom angesprochen, an der das nächste Byte gelesen bzw. geschrieben wird. Im Beispielprogramm von Abschnitt 16.1.2 wird die Position der geöffneten Datei mit der folgenden Anweisung auf den Dateianfang zurückgesetzt:
  ```
  fs.Position = 0;
  ```
- **public int ReadByte()**
  Mit dieser Methode wird ein **Stream**-Objekt aufgefordert, *ein* Byte per Rückgabewert vom Typ **int** zu liefern und seine Position entsprechend zu erhöhen. Ist das Ende des Stroms erreicht, wird der Rückgabewert -1 geliefert.
- **public int Read(byte[]** *buffer***, int** *offset***, int** *count***)**
  Mit dieser Methode wird ein **Stream**-Objekt aufgefordert, *count* Bytes zu liefern, im **byte**-Array *buffer* in den Elementen ab Index *offset* abzulegen und seine Position entsprechend zu erhöhen. Als Rückgabewert erhält man die Anzahl der tatsächlich gelieferten Bytes, die bei unzureichendem Vorrat kleiner als *count* ausfallen kann.
- **public void WriteByte()**
  Mit dieser Methode wird ein **Stream**-Objekt aufgefordert, *ein* Byte zu schreiben und seine Position entsprechend zu erhöhen.
- **public void Write(byte[]** *buffer***, int** *offset***, int** *count***)**
  Mit dieser Methode wird ein **Stream**-Objekt aufgefordert, *count* Bytes zu schreiben, die sich im **byte**-Array *buffer* in den Elementen ab Index *offset* befinden, und seine Position entsprechend zu erhöhen.
- **public void Flush()**
  Viele **Stream**-Objekte verwenden beim Schreiben aus Performanzgründen einen Puffer, um die Anzahl der zeitaufwändigen Zugriffe auf eine angeschlossene Senke (z. B. Datei) möglichst gering zu halten. Mit der Methode **Flush()** verlangt man die sofortige Ausgabe des Puffers, sodass der komplette Inhalt für Abnehmer zur Verfügung steht. Bei der Klasse **FileStream** wird eine voreingestellte Puffergröße von 4096 Bytes benutzt.

- **public void Dispose()**
  public void Close()
  Über einen expliziten oder per **using**-Anweisung automatisierten Aufruf der Methode **Dispose()** werden alle mit einem **Stream**-Objekt verbundenen Ressourcen (z. B. eine geöffnete Datei) wieder freigegeben (siehe Abschnitt 16.2). Statt **Dispose()** kann man auch die Methode **Close()** aufrufen.
- **public long Length {get;}**
  Diese Eigenschaft enthält die Länge des Stroms (z. B. die Dateigröße) in Bytes.
- **public long Seek(long** *offset***, SeekOrign** *origin***)**
  Diese Methode fordert einen Strom auf, seine Position relativ zu einem per **SeekOrigin**-Wert festgelegten Bezugspunkt (**Begin**, **Current**, **End**) neu zu setzen, z. B. vom aktuellen Stand aus um vier Bytes zurück:

  ```
  fs.Seek(-4, SeekOrigin.Current);
  ```

  Als Rückgabe erhält man die neue Position.
- **public bool CanRead {get;}**, **public bool CanSeek {get;}**, **public bool CanWrite {get;}**
  Über diese Eigenschaften lässt sich feststellen, ob ein Strom das Lesen, Schreiben oder Positionieren erlaubt, z. B.:

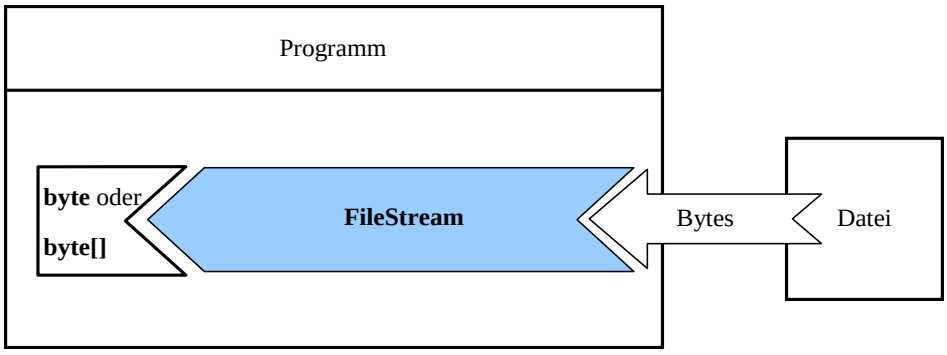
Quellcodesegment	Ausgabe

`Console.WriteLine("CanRead:  " + fs.CanRead);`
`Console.WriteLine("CanSeek:  " + fs.CanSeek);`
`Console.WriteLine("CanWrite: " + fs.CanWrite);`
	`CanRead:  True`
	`CanSeek:  True`
	`CanWrite: True`

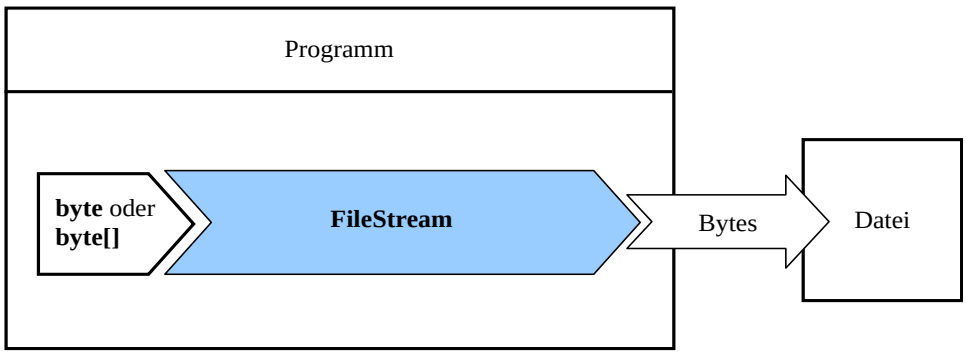
Die beschriebenen Ein-/Ausgabemethoden arbeiten synchron, blockieren also den aktuellen Thread bis zu ihrer Beendigung, was variable und oft inakzeptable Wartezeiten verursachen kann. Im Kapitel 17 werden asynchrone, nicht blockierende Ein-/Ausgabemethoden vorgestellt (siehe Abschnitt 17.5.9).

16.1.4 FileStream

Weil im aktuellen Kapitel vor allem die Anwendung des Datenstromkonzepts auf die *Dateiverarbeitung* interessiert, sehen wir uns die Klasse **FileStream** näher an. Mit einem **FileStream**-Objekt können Bytes aus einer Datei gelesen



oder dorthin geschrieben werden:



Ein **FileStream**-Objekt beherrscht prinzipiell *beide* Transportrichtungen, kann aber auch auf unidirektionalen Betrieb eingestellt werden.

Im folgenden **FileStream**-Konstruktoraufruf wird die Datei **demo.bin** erstellt und zum Lesen und/oder Schreiben geöffnet:[1]

```
FileStream fs = new FileStream("demo.bin", FileMode.Create);
```

Falls die Datei bereits existiert, wird sie überschrieben.

Für hinreichende Flexibilität bei der Ansprache und Behandlung von Dateien sorgen die Werte des im Beispiel benutzten Konstruktorparameters **FileMode** (siehe Abschnitt 16.1.4.1) sowie insgesamt 11 (nicht als veraltet deklarierte) Überladungen des Konstruktors.

FileStream-Objekte verwenden einen Puffer, um die Anzahl der Zugriffe auf die verbundene Datei möglichst gering zu halten. Beim Schließen einer Datei (durch den Garbage Collector oder einen expliziten **Dispose()** - Aufruf) werden gepufferte Schreibvorgänge automatisch ausgeführt. Einige Überladungen des Konstruktors bieten einen Parameter, um die voreingestellte Puffergröße von 4096 Bytes zu ändern.

16.1.4.1 Öffnungsmodus (FileMode)

Beim Erstellen eines **FileStream**-Objekts kann man den Öffnungsmodus über einen Wert des Enumerationstyps **FileMode** wählen:

[1] In dieser Anweisung wird die seit C# 9 bestehenden Möglichkeit nicht genutzt, im **new**-Ausdruck den Namen des Konstruktors wegzulassen, wenn der Compiler den Namen erschließen kann (zu zieltypisierten **new**-Ausdrücken siehe Abschnitt 5.4.3.2). Im Manuskript werden bewusst alternierend beide Schreibweisen genutzt, damit sich die Leser an die Koexistenz und die Äquivalenz gewöhnen.

Modus	Beschreibung
Append	Die Datei wird geöffnet und auf das Ende positioniert oder neu erzeugt.
Create	Ist die Datei noch nicht vorhanden, wird sie angelegt (wie bei **CreateNew**), anderenfalls wird sie überschrieben (wie bei **Truncate**).
CreateNew	Es wird eine neue Datei angelegt, oder eine **IOException** geworfen, falls eine Datei mit dem gewünschten Namen bereits existiert.
Open	Es wird eine vorhandene Datei geöffnet, oder eine **IOException** geworfen, falls keine Datei mit dem angegebenen Namen existiert.
OpenOrCreate	Es wird eine neue Datei erzeugt oder eine vorhandene geöffnet, jedoch im Unterschied zu **Create** *nicht* automatisch überschrieben.
Truncate	Es wird eine vorhandene Datei geöffnet und entleert, oder eine **IOException** geworfen, falls keine Datei mit dem gewünschten Namen existiert.

Beim Öffnungsmodus **Append** befindet sich der Dateizeiger am Ende der Datei (hinter dem letzten Byte), und es ist kein lesender Zugriff möglich. Bei den anderen Öffnungsmodi befindet sich der Dateizeiger am Dateianfang (ggf. vor dem ersten Byte).

Mit einem Öffnungsmodus wählt man ein Bündel von Einstellungen:

- Ist eine vorhandene Datei Voraussetzung oder Grund für einen Ausnahmefehler?
- Initiale Position des Dateizeigers
- Soll der aktuelle Inhalt einer vorhandenen Datei gelöscht oder beibehalten werden?
- Welche Zugriffsmöglichkeiten sind erlaubt (siehe Abschnitt 16.1.4.2)?

16.1.4.2 Zugriffsmöglichkeiten für das eigene FileStream-Objekt (FileAccess)

Bei einigen Überladungen des **FileStream**-Konstruktors lassen sich über einen Parameter vom Enumerationstyp **FileAccess** die Zugriffsmöglichkeiten für das eigene **FileStream**-Objekt vereinbaren, wobei auf die Verträglichkeit mit dem Eröffnungsmodus zu achten ist. Es sind folgende Alternativen verfügbar:

- **FileAccess.Read**
- **FileAccess.Write**
- **FileAccess.ReadWrite**

Im folgenden Beispiel wird eine Datei zum Lesen geöffnet:

```
FileStream fs = new FileStream("demo.bin", FileMode.Open,
                               FileAccess.Read);
```

Die im Abschnitt 16.1.4.1 beschriebenen Öffnungsmodi (**FileMode**-Ausprägungen) für Dateien sind nur eingeschränkt mit den **FileAccess**-Werten kombinierbar:

Öffnungsmodus	Erlaubte FileAccess-Werte	Voreinstellung
Append	Write	Write
Create	Write, ReadWrite	ReadWrite
CreateNew	Write, ReadWrite	ReadWrite
Open	Read, Write, ReadWrite	ReadWrite
OpenOrCreate	Read, Write, ReadWrite	ReadWrite
Truncate	Write, ReadWrite	ReadWrite

16.1.4.3 Zugriffsmöglichkeiten für andere Interessenten (FileShare)

Bei einigen **FileStream**-Konstruktorüberladungen kann man die Rechte für andere, simultan zugreifende Interessenten über einen Parameter vom Enumerationstyp **FileShare** regeln. Dabei sind u. a. die folgenden Alternativen verfügbar:

- **FileShare.None**
- **FileShare.Read**
- **FileShare.Write**
- **FileShare.ReadWrite**

Im folgenden Beispiel wird die gemeinsame Nutzung komplett verweigert:

```
FileStream fs = new FileStream("demo.bin", FileMode.Create,
                                           FileAccess.ReadWrite,
                                           FileShare.None);
```

Bei **FileStream**-Konstruktoren ohne **FileShare**-Parameter ist der Wert **FileShare.Read** eingestellt.

In der Regel befinden sich andere Interessenten für den Zugriff auf dieselbe Datei in einem anderen Programm (Prozess). Doch kann es auch sinnvoll sein, dass verschiedene Threads (Ausführungsfäden) *innerhalb* eines Programms jeweils über ein eigenes **FileStream**-Objekt auf dieselbe Datei zugreifen.

Während das simultane Lesen einer Datei durch mehrere Prozesse oder Threads unproblematisch und erlaubt ist, sind simultane Schreibzugriffe selbstverständlich verboten. Eine Datei zu lesen, in die ein anderer Prozess oder Thread gleichzeitig schreibt, kann z. B. bei einer Protokolldatei sinnvoll sein.

Beim Simultanzugriff auf eine Datei über zwei verschiedene **FileStream**-Objekte, ist zu beachten, dass *beide* Objekte über einen geeigneten **FileShare**-Parameterwert im Konstruktor den Partner berechtigen müssen. Wenn sich die beiden folgenden, auf dieselbe Datei zugreifenden **FileStream**-Objekte in verschiedenen Programmen befinden,

```
FileStream fs1 = new FileStream(name, FileMode.Create, FileAccess.Write);
  . . .
FileStream fs2 = new FileStream(name, FileMode.Open, FileAccess.Read);
```

dann scheitert beim versuchten simultanen Zugriff der jeweils zweite Interessent mit einem Ausnahmefehler:

```
Unhandled exception. System.IO.IOException: The process cannot access the file
'c:\temp\demo.bin' because it is being used by another process.
```

Weil `fs2` den voreingestellten **FileShare**-Wert **Read** verwendet, also das von `fs1` benötigte Schreibrecht nicht einräumt, wird der simultane Betrieb verhindert. So klappt es:

```
FileStream fs2 = new FileStream(name, FileMode.Open, FileAccess.Read,
                                      FileShare.Write);
```

16.2 Freigabe von Ressourcen

Nachdem ein Datenstromobjekt seine Aufgabe erfüllt hat, müssen die belegten Betriebssystem-Ressourcen (z. B. Datei-Handles, Netzwerk-Sockets) durch einen **Dispose()** - oder **Close()** - Aufruf freigegeben werden, um den Benutzer und andere Programme möglichst wenig zu behindern. Dabei wird ein vorhandener Schreibpuffer (z. B. bei der Klasse **FileStream**) durch einen automatischen **Flush()** - Aufruf entleert, sodass der komplette Inhalt für Abnehmer zur Verfügung steht. Außerdem muss sichergestellt werden, dass die Freigabe unter allen Umständen erfolgt, insbesondere auch nach einem Ausnahmefehler.

16.2.1 Dispose() oder Close()

Um die durch einen Datenstrom aus der **Stream**-Hierarchie belegten Ressourcen freizugeben, ruft man die **Dispose()** - oder die funktionsgleiche **Close()** - Methode auf, z. B.:

```
fs.Dispose();
```

Diese beiden parameterfreien Methoden rufen letztlich eine **Dispose()** - Überladung mit **bool**-Parameter auf, welche die eigentlichen Aufräumungsarbeiten (inklusive Entleeren des Puffers) verrichtet.

Neben der Klasse **Stream** implementieren auch viele andere BCL-Klassen das Interface **IDisposable** (vgl. Abschnitt 5.4.4), das die Methode **Dispose()** vorschreibt:

public void Dispose()

Nach einem **Dispose()** - bzw. **Close()** - Aufruf existiert das angesprochene **Stream**-Objekt weiterhin, doch führen Lese- bzw. Schreibversuche zu Ausnahmefehlern. Rufen Sie **Close()** bzw. **Dispose()** also *nicht* auf, wenn solche Ausnahmefehler möglich sind, weil im Programm noch weitere Referenzen auf das **Stream**-Objekt existieren.

16.2.2 Garbage Collector

Die eben beschriebene Panne ist ausgeschlossen, wenn man überflüssig gewordene Datenstromobjekte dem Garbage Collector überlässt, was Richter (2012, S. 535) empfiehlt. Über die Finalisierungsmethode (vgl. Abschnitt 5.4.4) der von **Stream** abgeleiteten Klassen ist sichergestellt, dass vor dem Entfernen eines Objekts per Garbage Collector alle verwendeten Ressourcen (z. B. Dateien, Netzwerkverbindungen) freigegeben werden, wobei die im Abschnitt 16.2.1 erwähnte **Dispose()** - Überladung mit **bool**-Parameter zum Einsatz kommt.

Bei den Aufräumungsarbeiten des Garbage Collectors sind allerdings Zeitpunkt und Reihenfolge unbestimmt. Setzt z. B. im Rahmen einer schreibenden Datenstrom-Verarbeitungskette ein Ausgabeobjekt mit eigenem Puffer (z. B. aus der Klasse **StreamWriter)** auf einem **Stream**-Objekt auf, darf man dem Garbage Collector das Schließen auf keinen Fall überlassen (siehe Abschnitt 16.3.2). Es kommt zu Datenverlusten wenn der Garbage Collector bei seinen Aufräumarbeiten (ohne garantierte Reihenfolge!) das **Stream**-Objekt *vor* dem **StreamWriter**-Objekt beseitigt.[1]

Unter ungünstigen Umständen kann die Finalisierungsmethode zu einem (Ausgabe-)Objekt von der CLR überhaupt nicht ausgeführt werden, weil z. B. der vorherige Aufruf einer Finalisierungsmethode blockiert.[2]

Außerdem ist es oft wichtig, die durch ein Stromobjekt belegten Ressourcen so früh wie möglich freizugeben, um (exklusive) Zugriffe durch andere Interessenten zu ermöglichen. Im Beispielprogramm `FSDemo` im Abschnitt 16.1.2 ist das Schließen der Datei erforderlich, um sie anschließend löschen zu können.

Daher ist es in der Regel sinnvoll, die durch Datenstromobjekte belegten Ressourcen explizit freizugeben, was durch die gleich zu beschreibende **using**-Anweisung erleichtert wird.

---

[1] https://docs.microsoft.com/en-us/archive/msdn-magazine/2000/november/garbage-collection-automatic-memory-management-in-the-microsoft-net-framework

[2] https://docs.microsoft.com/en-us/dotnet/api/system.object.finalize

16.2.3 using-Anweisung

Die **using**-Anweisung (nicht zu verwechseln mit der **using**-*Direktive* zum Importieren eines Namensraums) unterstützt eine einfache und garantierte Freigabe von Ressourcen (z. B. Dateiverbindungen). In der traditionellen Syntax folgt auf das Schlüsselwort **using** eine durch runde Klammern begrenzte Deklaration und eine Anweisung, wobei natürlich auch eine Blockanweisung erlaubt ist. Meist wird eine bzgl. der **using**-Anweisung lokale Variable von einem das Interface **IDisposable** implementierenden Klassen- oder Strukturtyp deklariert und initialisiert (zu **IDisposable** siehe Abschnitt 5.4.4). Die **IDisposable**-Variable kann in der (Block-)Anweisung verwendet, aber nicht verändert werden. Nach Ausführung der (Block-)Anweisung wird die **IDisposable**-Instanz automatisch per **Dispose()** aufgefordert, verwendete Ressourcen (z. B. eine Datei-, Datenbank- und Netzwerkverbindung) freizugeben. Statt *einer* lokalen Variablen dürfen auch *mehrere* vom selben Typ definiert werden.[1]

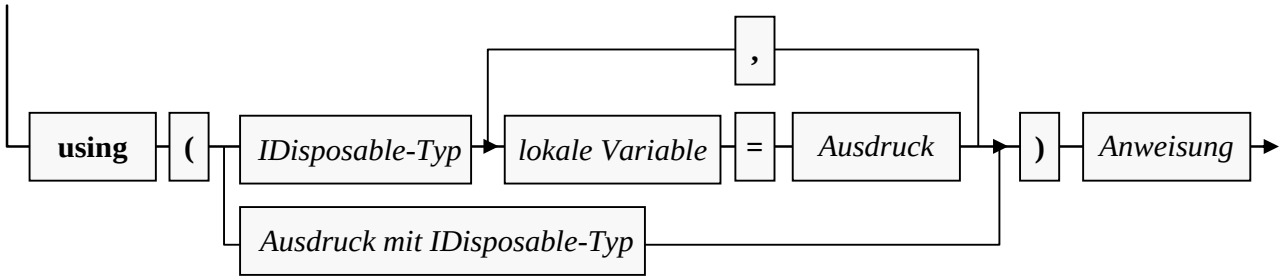
Im Deklarationsteil der **using**-Anweisung kann auch ein Ausdruck angegeben werden, der z. B. eine Referenz auf ein bereits vorhandenes Objekt liefert, das die Schnittstelle **IDisposable** implementiert.

Die drei beschriebenen Varianten für den Deklarationsteil einer **using**-Anweisung

- Deklaration einer lokalen Variablen mit **IDisposable**-Typ
- Deklaration von mehreren lokalen Variablen mit demselben **IDisposable**-Typ
- Ausdruck mit **IDisposable**-Typ

werden im folgenden Syntaxdiagramm dargestellt:

using-Anweisung mit Standardsyntax



In der **using**-Anweisung des im Abschnitt 16.1.2 vorgestellten Beispielprogramms `FSDemo` wird eine Variable vom Typ **FileStream** deklariert und initialisiert:

[1] https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/using-directive

```
using System;
using System.IO;

class FSDemo {
    static void Main() {
        String name = "demo.bin";
        byte[] arr = { 0, 1, 2, 3 , 4, 5, 6, 7 };
        using (FileStream fs = new(name, FileMode.CreateNew)) {
            fs.Write(arr, 0, arr.Length);
            fs.Position = 0;
            fs.Read(arr, 0, arr.Length);
            foreach (byte b in arr)
                Console.WriteLine(b);
        }
        File.Delete(name);
    }
}
```

Per **using**-Anweisung wird die Ressourcenfreigabe für einen Strom elegant gelöst, wobei der Compiler im Hintergrund eine **try**-Anweisung mit **finally**-Block erstellt (vgl. Abschnitt 13.2.1). Das letzte Beispielprogramm ist äquivalent zu:

```
using System;
using System.IO;

class UsingInterpretation {
    static void Main() {
        string name = "demo.bin";
        byte[] arr = { 0, 1, 2, 3, 4, 5, 6, 7 };
        {
            FileStream fs = new(name, FileMode.CreateNew);
            try {
                fs.Write(arr, 0, arr.Length);
                fs.Position = 0;
                fs.Read(arr, 0, arr.Length);
                foreach (byte b in arr)
                    Console.WriteLine(b);
            } finally {
                if (fs != null)
                    fs.Dispose();
            }
        }
        File.Delete(name);
    }
}
```

Ausnahmen, mit denen bei Dateisystemzugriffen jederzeit zu rechnen ist, werden von der **using**-Anweisung *nicht* behandelt, sondern an den Aufrufer weitergeleitet. Außerdem ist es nicht immer erwünscht, dass eine Ressourcen nutzende Instanz am Ende der **using** - (Block-)anweisung verworfen wird.[1] Mit Hilfe der im weiteren Verlauf des Abschnitts behandelten, seit C# 8.0 verfügbaren alternativen **using**-Syntax lässt sich der Zeitpunkt der Ressourcen-Freigabe etwas besser kontrollieren.

Oft bietet sich bei Datenstromobjekten die Verwendung von verschachtelten Konstruktoraufrufen an, und dabei kann aufgrund der eben beschriebenen Umsetzung der **using**-Anweisung durch den Compiler die automatische Ressourcen-Freigabe scheitern. Im folgenden Beispiel wird als Parameter in einem Konstruktor der Klasse **StreamReader**, die zum Lesen von Zeichenfolgen aus Textda-

[1] https://docs.microsoft.com/de-de/dotnet/standard/garbage-collection/using-objects

teien dient (siehe Abschnitt 16.3.2), per **new**-Operator ein Objekt der Klasse **FileStream** erstellt, das mit einer Datei verbunden ist:

```
using (StreamReader sr = new(
                            new FileStream(name, FileMode.Open, FileAccess.Read)))
    for (int i = 0; sr.Peek() >= 0; i++)
        Console.WriteLine("{0}:\t{1}", i, sr.ReadLine());
```

Der Compiler realisiert die **using**-Anweisung folgendermaßen:

```
{
    FileStream fs = new FileStream(name, FileMode.Open, FileAccess.Read);
    StreamReader sr = new StreamReader(fs);
    try {
        for (int i = 0; sr.Peek() >= 0; i++)
            Console.WriteLine("{0}:\t{1}", i, sr.ReadLine());
    } finally {
        if (sr != null)
            sr.Dispose();
    }
}
```

Wenn das **StreamReader**-Objekt zustande kommt und schlussendlich einen **Dispose()** - Aufruf erhält, dann wird dieser Aufruf an das **FileStream**-Objekt weitergeleitet, und alle Ressourcen werden freigegeben. Tritt aber im **StreamReader**-Konstruktor eine Ausnahme auf, dann kommt kein **StreamReader**-Objekt zustande, und das vorhandene **FileStream**-Objekt mit der Dateiverbindung erhält *keinen* **Dispose()** - Aufruf, sodass die Datei bis zum Programmende geöffnet bleibt.

Mit der folgenden Verschachtelung von **using**-Anweisungen ist das Risiko beseitigt:

```
using (FileStream fs = new(name, FileMode.Open, FileAccess.Read))
    using (StreamReader sr = new(fs))
        for (int i = 0; sr.Peek() >= 0; i++)
            Console.WriteLine("{0}:\t{1}", i, sr.ReadLine());
```

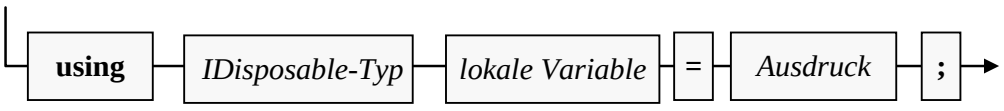
Dabei wird ausgenutzt, dass in der Klasse **FileStream** die Methode **Dispose()** konform zu einer Empfehlung von Microsoft auf wiederholte Aufrufe großzügig reagiert. Bei der nun gewählten Konstruktion erhält das **FileStream**-Objekt zunächst einen vom **StreamReader**-Objekt weitergereichten und danach einen separaten **Dispose()** - Aufruf.

Peterson (2014) behandelt neben dem dargestellten Risiko aufgrund eines **using**-Ausdrucks mit verschachtelten **IDisposable**-Konstruktoren noch weitere, ähnlich gelagerte Probleme der **using**-Anweisung.

Seit C# 8.0 ist für die **using**-Anweisung eine alternative Syntax verfügbar, wobei ...

- der Deklarationsteil *nicht* durch runde Klammern begrenzt wird,
- keine integrierte Anweisung vorhanden ist,
- die Freigabe der deklarierten Ressource beim Verlassen des Gültigkeitsbereichs erfolgt.

using-Anweisung mit alternativer Syntax



Damit lässt sich in manchen Fällen der Zeitpunkt der Ressourcenfreigabe flexibler festlegen als mit der traditionellen **using**-Syntax. Im folgenden Beispiel befindet sich die per **using** deklarierte Dateiverbindung im Block einer **if**-Anweisung:

```
if (!File.Exists(name) && arr.Length > 0) {
    using FileStream fs = new(name, FileMode.CreateNew);
    fs.Write(arr, 0, arr.Length);
     . . .
}
```

16.2.4 Ausnahmen behandeln

Weil Methodenaufrufe bei Datenstromobjekten diverse Ausnahmen produzieren können (z. B. **FileNotFoundException**, **UnauthorizedAccessException**, **IOException**), sind sie am besten in einem **try** - Block untergebracht. Damit ein **Dispose()** - Aufruf zur Freigabe von belegten Ressourcen (z. B. Dateien) auf jeden Fall ausgeführt wird, gehört er in den **finally**-Block der **try**-Anweisung. In der folgenden Variante des Einstiegsbeispiels von Kapitel 16 ist das Abfangen der Ausnahmen immerhin angedeutet:

```
using System;
using System.IO;

class FSDemoCatch {
   static void Main() {
      string name = "demo.bin";
      byte[] arr = { 0, 1, 2, 3, 4, 5, 6, 7 };
      FileStream fs = null;
      try {
         fs = new FileStream(name, FileMode.CreateNew);
         fs.Write(arr, 0, arr.Length);
         fs.Position = 0;
         fs.Read(arr, 0, arr.Length);
         foreach (byte b in arr)
            Console.WriteLine(b);
      } catch (Exception e) {
         Console.WriteLine(e.Message);
      } finally {
         if (fs != null)
            fs.Close();
      }
      try {
         File.Delete(name);
      } catch (Exception e) {
         Console.WriteLine(e.Message);
      }
   }
}
```

Mit Hilfe der **using**-Anweisung (siehe Abschnitt 16.2) lässt sich das Verhalten des Programms verbessern und dabei auch noch Schreibaufwand sparen:

```
static void Main() {
   string name = "demo.bin";
   byte[] arr = { 0, 1, 2, 3, 4, 5, 6, 7 };
   try {
      using (FileStream fs = new(name, FileMode.CreateNew)) {
         fs.Write(arr, 0, arr.Length);
         fs.Position = 0;
         fs.Read(arr, 0, arr.Length);
         foreach (byte b in arr)
            Console.WriteLine(b);
      }
      File.Delete(name);
   } catch (Exception e) {
      Console.WriteLine(e.Message);
   }
}
```

Nun steht der (implizite) **Dispose()** - Aufruf im **finally**-Block der vom Compiler aus der **using**-Anweisung erstellten **try**-Anweisung, und es sind zwei **try**-Anweisungen verschachtelt. Kommt es beim Öffnen der Ausgabedatei mit dem **FileMode.CreateNew** zu einem Fehler, weil bereits eine Datei mit diesem Namen vorhanden ist, dann wird die **Delete()** - Methode *nicht* aufgerufen. Von der vorherigen Programmversion wird eine vorgefundene Datei hingegen gelöscht.

16.3 Verarbeitung von Daten mit höherem Typ

Bisher haben wir uns auf das Schreiben und Lesen von *Bytes* beschränkt. Im Abschnitt 16.3 werden Verfahren beschrieben, um Daten mit einem beliebigen (aus mehreren Bytes bestehenden) Typ (z. B. **int**, **double**, **String**) in Dateien zu schreiben oder von dort zu lesen.

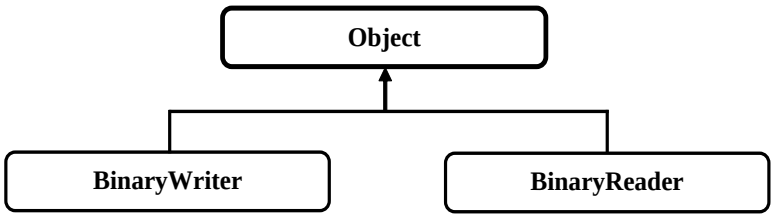
Bei den Dateien auf einem Rechner kann man unterscheiden:

- **Binärdateien**
  In einer Binärdatei werden Daten im Wesentlichen genauso dargestellt wie im Arbeitsspeicher eines Rechners, sodass Programme wenig Mühe dabei haben, Daten beliebigen Typs in eine Binärdatei zu schreiben oder aus einer Binärdatei mit bekanntem Aufbau zu lesen. Öffnet man eine Binärdatei mit einem Texteditor, ist eine Folge von (Sonder-)zeichen zu sehen, die von den meisten Menschen weder verstanden noch zielgerichtet geändert werden kann. Im Abschnitt 16.3.1 werden die zum Schreiben bzw. Lesen von Werten mit einem elementaren Datentyp (z. B. **int**, **double**) konstruierten Klassen **BinaryWriter** bzw. **BinaryReader** vorgestellt.
- **Textdateien**
  Diese Dateien können von Menschen mit Hilfe eines Texteditors gelesen und/oder bearbeitet werden, sofern der Texteditor die beim Erstellen der Datei verwendete Zeichencodierung versteht. Per Programm lassen sich numerische Daten und Zeichenfolgen leicht in eine Textdatei schreiben, doch ist das Lesen *numerischer* Daten aus einer Textdatei mit erhöhtem Aufwand verbunden. Im Abschnitt 16.3.2 werden die zum Schreiben bzw. Lesen von Zeichenfolgen konstruierten Klassen **StreamWriter** bzw. **StreamReader** vorgestellt.

Wie man komplette Instanzen mit einem Klassen- oder Strukturtyp in eine Binär- oder Textdatei schreibt bzw. von dort liest, wird im Abschnitt 16.5 über das sogenannte *Serialisieren* behandelt.

16.3.1 Schreiben und Lesen im Binärformat

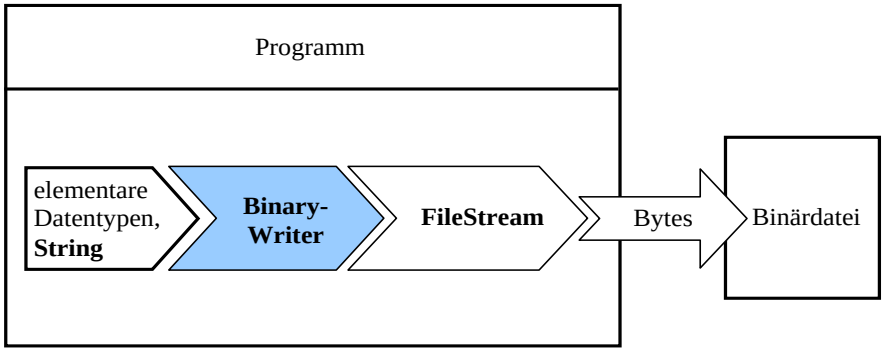
Um Werte von einem elementaren Datentyp (z. B. **int**, **double**) sowie Zeichenfolgen in einen binär organisierten Strom (z. B. in eine Binärdatei) zu schreiben bzw. aus einem Binärstrom mit bekanntem Aufbau zu lesen, verwendet man ein Objekt der Klasse **BinaryWriter** bzw. **BinaryReader**.



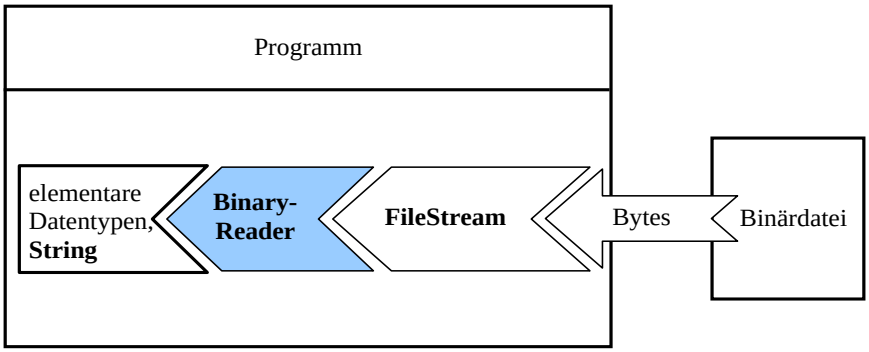
Die beiden Klassen stammen *nicht* von der im Abschnitt 16.1 behandelten Klasse **Stream** ab, verwenden aber für die Verbindung mit einer Datenquelle oder -senke ein **Stream**-Objekt, das im Konstruktor anzugeben ist, z. B.:

public BinaryWriter(Stream *ausgabestrom***)**

Im Unterschied zu den „bidirektionalen" **Stream**-Klassen sind für das Schreiben bzw. Lesen von Daten mit elementarem Typ „unidirektionale" Klassen zuständig. Schreibt man per **BinaryWriter** in eine Datei, entsteht die folgende Verarbeitungskette:



Beim *Lesen* aus einer Binärdatei reisen die Daten in umgekehrter Richtung:



Das folgende Beispielprogramm schreibt einen **int**- und einen **double**-Wert sowie eine Zeichenfolge per **BinaryWriter** über einen **FileStream** in eine Datei. Anschließend werden die Daten über eine **BinaryReader** - **FileStream** - Konstruktion eingelesen:

```
using System;
using System.IO;

class BinWrtRd {
    static void Main() {
    string name = "demo.bin";
        using (var fso = new FileStream(name, FileMode.Create))
        using (var bw = new BinaryWriter(fso)) {
            bw.Write(4711);
            bw.Write(3.1415926);
            bw.Write("Nicht übel");
        }
        using (var fsi = new FileStream(name, FileMode.Open, FileAccess.Read))
        using (var br = new BinaryReader(fsi))
            Console.WriteLine(br.ReadInt32() + "\n" +
                              br.ReadDouble() + "\n" +
                              br.ReadString());
  }
}
```

Im Beispiel wird für das Schreiben und das Lesen jeweils ein eigenes **FileStream**-Objekt mit passenden **FileMode**- bzw. **FileAccess**- Konstruktorparameterwerten verwendet, um den beim Schreiben erforderlichen exklusiven Dateizugriff möglichst kurzzeitig in Anspruch zu nehmen. Es wäre möglich, mit *einem* **FileStream**-Objekt zu arbeiten und dessen **Position**-Eigenschaft nach dem Schreiben wieder auf 0 zu setzen (siehe Beispiel im Abschnitt 16.1.1).

Die von beiden **FileStream**-Objekten verwendete Datei wird zunächst mit dem **FileMode.Create** geöffnet. Nach den Schreibzugriffen per **Write()** - Methode wird die Datei dank der **using**-Anweisungen, die vorsichtshalber verschachtelt sind (vgl. Abschnitt 16.2.3), automatisch geschlossen, sodass die Datei anschließend mit dem **FileMode.Open** geöffnet werden kann. Somit wird ...

- für eine sichere und möglichst frühzeitige Ressourcenfreigabe gesorgt,
- die für andere Datei-Interessenten besonders einschränkende Öffnungszeit mit Schreibzugriff möglichst kurz gehalten.

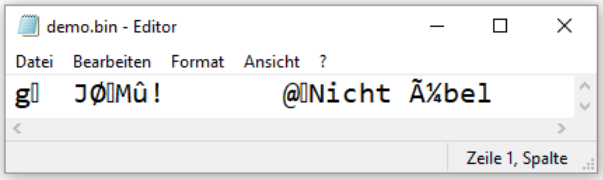
Durch den (implizit per **using** erfolgenden) **Dispose()** - Aufruf wird der Schreibpuffer des **FileStream**-Objekts `fso` geleert, sodass die geschriebenen Daten komplett in der Datei ankommen. Ein **BinaryWriter** verwaltet übrigens *keinen* eigenen Puffer, sondern reicht Schreibaufträge stets direkt an das angeschlossene **Stream**-Objekt weiter. Erhält ein **BinaryWriter**-Objekt einen **Flush()** - Aufruf zum Entleeren des Puffers, reicht es ihn an das verbundene **Stream**-Objekt weiter.

Während die Klasse **BinaryWriter** für alle unterstützten Datentypen eine eigene Überladung der Methode **Write()** besitzt, sind in der Klasse **BinaryReader** typspezifisch benannte Lesemethoden vorhanden (z. B. **ReadInt32()**, **ReadDouble()**).

Das Programm liefert die folgende Ausgabe:

```
4711
3,1415926
Nicht übel
```

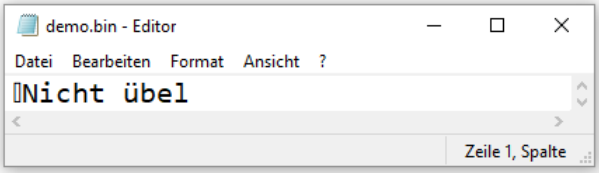
Es macht wenig Sinn, die vom Beispielprogramm erzeugte Binärdatei mit einem Texteditor zu öffnen:



Im Beispiel wird die vom **BinaryWriter** geschriebene Zeichenfolge allerdings vom Windows-Texteditor **notepad.exe** fast korrekt dargestellt. Bei der Ausgabe von **String**-Variablen ist die verwendete *Ausgabecodierung* der Zeichen relevant, die im Arbeitsspeicher eines Rechners bekanntlich in der Unicode-Codierung vorliegen. Die Klasse **BinaryWriter** verwendet per Voreinstellung dasselbe **UTF8Encoding** wie die später vorzustellenden **TextWriter**-Klassen. Offenbar ist die vom Windows-Texteditor unterstellte ANSI-Codierung bei den am häufigsten verwendeten Zeichen mit dem **UTF8Encoding** kompatibel.[1] Über einen **BinaryWriter**-Konstruktor mit entsprechendem Parameter stehen auch andere Codierungen zur Verfügung:

public BinaryWriter(Stream *Ausgabestrom***, Encoding** *Codierung***)**

Schreibt man per **BinaryWriter** an Stelle der gemischten Ausgaben ausschließlich Text, kann der Windows-Texteditor beim Öffnen der Ergebnisdatei die zugrunde liegende UTF-8 - Codierung übrigens erkennen, und das Ergebnis ist perfekt bis auf eine kleine Störung am Anfang, die gleich erklärt wird:



Trotz der gemeinsamen Codierungsvoreinstellung gibt es zwischen der Klasse **BinaryWriter** und den **TextWriter**-Klassen einen kleinen Unterschied bei der Textausgabe. Der **BinaryWriter** schreibt zur Unterstützung des **BinaryReader**s vor jede Zeichenfolge ihre Länge (Anzahl der Bytes) und verwendet dabei den Datentyp **uint** mit einer speziellen 7-Bit - Codierung:

- Von den 32 **uint**-Bits wird nur der signifikante Anteil als Byte-Sequenz ausgegeben. Führende Nullen werden also weggelassen.
- Ein Ausgabebyte enthält 7 **uint**-Bits (mit der niedrigsten Wertigkeit beginnend). Im achten Bit signalisiert eine Eins, dass noch ein weiteres Paket mit (7+1) Bits folgt.

Bei der Zeichenfolge „Nicht übel“ mit 11 Bytes Länge (neun Single-Byte-Zeichen und ein Double-Byte-Zeichen bei UTF-8 - Codierung, siehe Abschnitt 16.3.2) schreibt der **BinaryWriter** als Längenpräfix *ein* Byte. Bei einer Zeichenfolge mit 258 Bytes Länge resultiert ein Längenpräfix mit zwei Bytes:

Länge der Zeichenfolge in Bytes	**uint**-Bits	Längenpräfix
11	0…0 00000000 00001011	00001011
258	0…0..00000001 00000010	10000010 00000010

Es besteht übrigens *kein* Risiko, wenn ein Gespann aus einem **BinaryWriter**- und einem **FileStream**-Objekt dem Garbage Collector anheimfallen, obwohl beim automatischen Finalisieren (ohne garantierte Reihenfolge!) zuerst das **FileStream**-Objekt beseitigt werden könnte:

- **BinaryWriter**-Objekte besitzen keinen lokaler Puffer, der beim Abräumen geleert werden müsste.
- In der Klasse **BinaryWriter** ist keine Finalisierungsmethode vorhanden, sodass beim Abräumen kein Zugriff auf das zugrunde liegende (und eventuell nicht mehr existente) **Stream**-Objekt stattfinden kann.

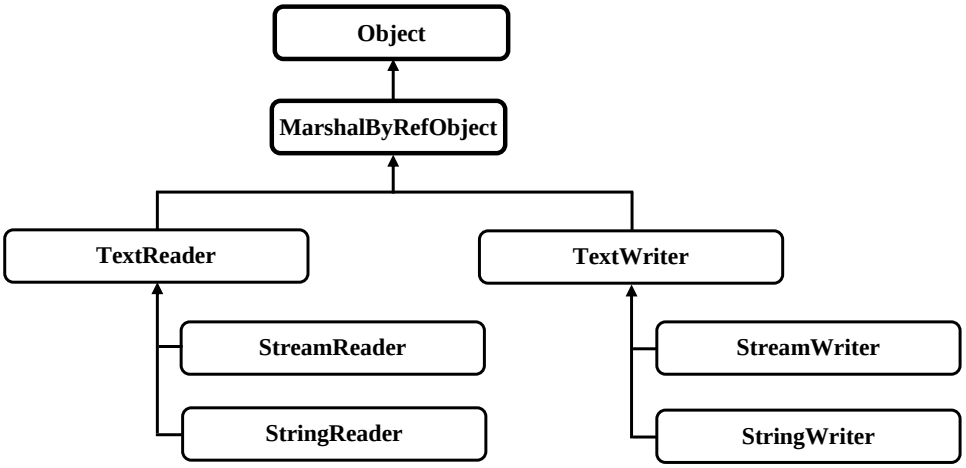
---

[1] Im Bereich von 0 bis 127 enthalten der Unicode und der ANSI-Code (*American National Standards Institute*) dieselben Zeichen wie der ASCII-Code (*American Standard Code for Information Interchange*) aus der Steinzeit der Datenverarbeitung.

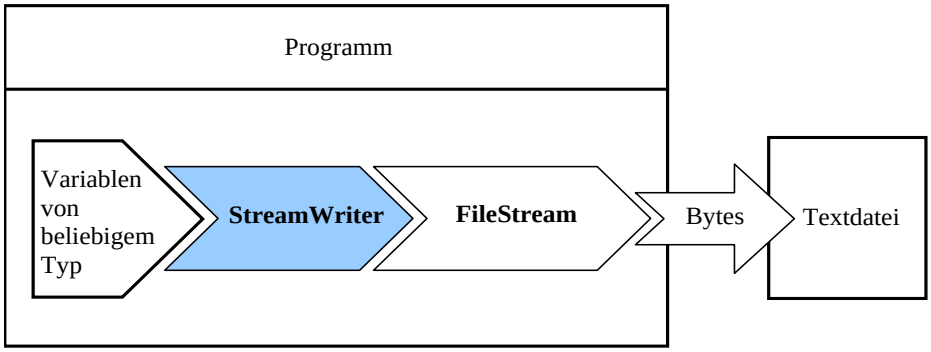
Objekte der anschließend behandelten Klasse **StreamWriter** müssen aufgrund ihres lokalen Puffers jedoch unbedingt *vor* dem zugrunde liegenden **Stream**-Objekt geschlossen werden, was nur durch einen **Dispose()** - Aufruf (oder einen äquivalenten **Close()** - Aufruf) an das **StreamWriter**-Objekt sichergestellt ist (explizit oder per **using**-Anweisung automatisiert, vgl. Abschnitt 16.2).

16.3.2 Schreiben und Lesen im Textformat

Mit einem **TextWriter**-Objekt kann man die Zeichenfolgenrepräsentation von Variablen beliebigen Typs ausgeben. Das Gegenstück **TextReader** liefert stets Zeichen ab, sodass bei der Versorgung von numerischen Variablen aus textuellen Eingabedaten etwas Eigeninitiative gefragt ist (siehe Übungsaufgabe im Abschnitt 16.7). Beide Klassen sind abstrakt, doch bietet die BCL auch konkrete Ableitungen für **Stream**- bzw. **String**-Objekte als Senken bzw. Quellen:

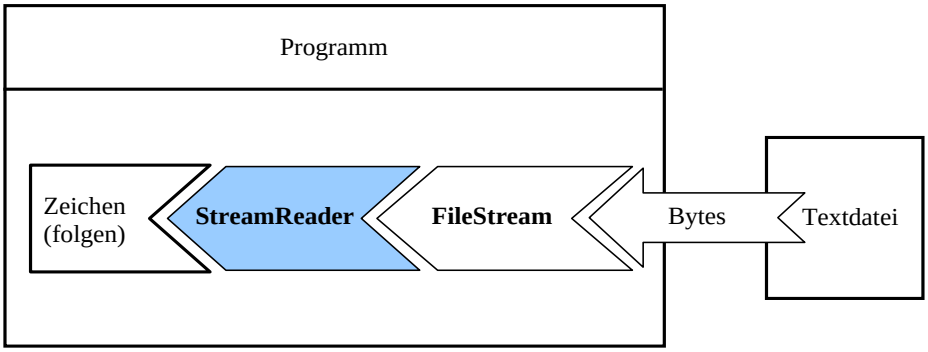


Um in eine Textdatei zu schreiben bzw. von dort zu lesen, verwendet man Objekte der Klassen **StreamWriter** bzw. **StreamReader**, die jeweils über eine Instanzvariable mit einem **FileStream**-Objekt verbunden sind (analog zu den Klassen **BinaryWriter** und **BinaryReader**).[1] Beim Schreiben haben wir also die folgende Situation:



Beim Lesen aus einer Textdatei reisen die Daten in umgekehrter Richtung:

[1] Die Bezeichnungen **StreamWriter** und **StreamReader** sind nicht ganz glücklich, weil auch ein **BinaryWriter** in ein **Stream**-Objekt schreibt und ein **BinaryReader** aus einem **Stream**-Objekt liest.



Man kann den Basisstrom für einen **StreamWriter** oder -**Reader** auch *implizit* erzeugen lassen, wenn die voreingestellten Kreationsparameter akzeptabel sind, z. B.:

```
StreamWriter sw = new StreamWriter("demo.txt");
```

Hier wird implizit ein **FileStream**-Objekt mit der voreingestellten Puffergröße 4096 erzeugt, das auf eine Datei mit dem Eröffnungsmodus **FileMode.Create** zugreift.

Das folgende Beispielprogramm schreibt einen **int**- und einen **double**-Wert sowie eine Zeichenfolge per **StreamWriter** über ein implizit erzeugtes **FileStream**-Objekt in eine Datei. Anschließend werden die Daten über einen **StreamReader** eingelesen, der sich auf ein explizit erzeugtes **FileStream**-Objekt stützt:

```
using System;
using System.IO;

class StreamWrtRd {
  static void Main() {
    string name = "demo.txt";
    using (var sw = new StreamWriter(name)) {
      sw.WriteLine(4711);
      sw.WriteLine(3.1415926);
      sw.WriteLine("Nicht übel");
    }
    using (var fs = new FileStream(name, FileMode.Open, FileAccess.Read))
    using (var sr = new StreamReader(fs)) {
      Console.WriteLine("Inhalt der Datei {0}:\n", ((FileStream)sr.BaseStream).Name);
      for (int i = 0; sr.Peek() >= 0; i++)
        Console.WriteLine("{0}:\t{1}", i, sr.ReadLine());
    }
  }
}
```

Natürlich gibt es auch eine **StreamReader**-Konstruktorüberladung mit einem Dateipfad als einzigem Parameter analog zur oben verwendeten **StreamWriter**-Konstruktorüberladung. Ein explizit per Konstruktor erstelltes **FileStream**-Objekt hat den Vorteil, dass man u. a. ...

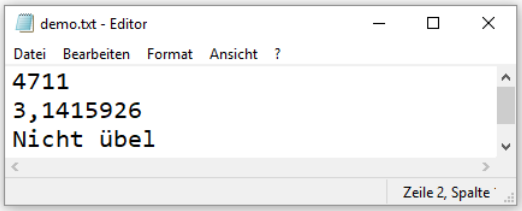
- den Öffnungsmodus der Datei (**FileMode**),
- die Zugriffsmöglichkeiten für das erstellte Objekt (**FileAccess**),
- die Zugriffsmöglichkeiten für andere Interessenten (**FileShare**)
- sowie die Puffergröße

einstellen kann (siehe Abschnitt 16.1.4).

Das Beispielprogramm liest so lange die nächste Dateizeile mit der **TextReader**-Methode **ReadLine()**, bis die **TextReader**-Methode **Peek()** durch die Rückgabe -1 das Dateiende signalisiert. Wir erhalten die folgende Ausgabe:

```
Inhalt der Datei U:\Eigene Dateien\C#\EA\StreamWrtRd\bin\Debug\demo.txt:

0:      4711
1:      3,1415926
2:      Nicht übel
```

Die erzeugte Textdatei ist ansehnlich:



Bei den **TextWriter**-Methoden **Write()** und **WriteLine()** treffen wir im Wesentlichen auf dieselben Überladungen wie bei den gleichnamigen **Console**-Methoden, die Ihnen aus zahlreichen Beispielen vertraut sind.

Im Unterschied zu einem **BinaryWriter** (siehe Abschnitt 16.3.1) besitzt ein **StreamWriter** einen lokalen Puffer (Datentyp: **char[]**, voreingestellte Größe: 1024). Daher muss ein **StreamWriter** unbedingt nach Gebrauch einen **Dispose()** - oder **Close()** - Aufruf erhalten. Dieser Aufruf wird automatisch an das verbundene **Stream**-Objekt weitergeleitet. Es wäre riskant, die beiden Objekte dem Garbage Collector zu überlassen, für den keine Arbeitsreihenfolge garantiert ist. Wenn er das **Stream**-Objekt *vor* dem **StreamWriter**-Objekt schließt, dann kann das **StreamWriter**-Objekt seinen Puffer nicht mehr ausgeben.[1]

Über die boolesche **StreamWriter**-Eigenschaft **AutoFlush** (Voreinstellung: **false**) wird festgelegt, ob die per **Write()** oder **WriteLine()** geschriebenen Zeichen sofort in den Ausgabestrom wandern (bei bestimmten Ausgabegeräten sinnvoll) oder zwischengepuffert werden sollen (höhere Performanz).[2]

Arbeitet ein **StreamWriter** mit einem **FileStream** zusammen, findet eine *Doppelpufferung* statt:

- Ein **StreamWriter**-Objekt enthält als Puffer einen **char**-Array (voreingestellte Größe: 1024).
- Ein **FileStream**-Objekt enthält als Puffer einen **byte**-Array (voreingestellte Größe: 4096).

Beim **Flush()** - Aufruf an ein **StreamWriter**-Objekt …

- wird zunächst der **StreamWriter**-Puffer in den Ausgabestrom geschrieben
- und dann ein **Flush**() - Aufruf an das **Stream**-Objekt gerichtet.

Hinweise zu einigen **TextReader**-Methoden:

---

[1] https://docs.microsoft.com/en-us/archive/msdn-magazine/2000/november/garbage-collection-automatic-memory-management-in-the-microsoft-net-framework

[2] Die folgende Implementierung der **StreamWriter**-Methode **Write()** mit **char**-Parameter aus der BCL zu .NET 5.0 zeigt, wie sich die öffentliche **AutoFlush**-Eigenschaft über das private **_autoFlush**-Feld auf das Pufferungsverhalten auswirkt:

```
public override void Write(char value) {
    . . .
    if (_autoFlush) {
        Flush(true, false);
    }
}
```

Der Quellcode stammt aus der Datei **StreamWriter.cs**, die über Microsofts .NET - Source Code - Webseite (https://source.dot.net/) inspiziert werden kann.

- **public String ReadLine()**
  Diese Methode liest eine Zeile, liefert das Ergebnis als **String**-Objekt ab und verschiebt die Position des Eingabestroms entsprechend. Unter einer *Zeile* ist eine Folge von Zeichen zu verstehen, auf die eine von den folgenden Terminierungen folgt:
  - das Steuerzeichen Carriage Return (0x000D),
  - das Steuerzeichen Line Feed (0x000A),
  - eine Sequenz aus den Sonderzeichen Carriage Return und Line Feed,
  - die in der statischen Eigenschaft **Environment.NewLine** hinterlegte Zeichenfolge,
  - das Ende des Stroms

  Im abgelieferten **String**-Objekt ist die Zeilenterminierung *nicht* enthalten. Enthält der Eingabestrom keine Zeichen mehr, liefert **ReadLine()** als Rückgabe den Wert **null**.
- **public int Read()**
  Diese Methode liefert als Rückgabewert die Unicode-Nummer des nächsten Zeichens und verschiebt die Position des Eingabestroms entsprechend. Wenn der Strom kein Zeichen mehr enthält, dann liefert **Read()** den Wert -1.
- **public int Peek()**
  Diese Methode liefert wie **Read()** die Unicode-Nummer des nächsten Zeichens oder den Wert -1, wenn der Strom kein Zeichen mehr enthält. Die Position des Stroms bleibt aber unverändert.

Per Voreinstellung schreiben bzw. lesen die **StreamWriter** bzw. -**Reader** Unicode-Zeichen unter Verwendung der Platz sparenden **UTF-8** - Codierung. Bei dieser Codierung werden die Unicode-Zeichen durch eine variable Anzahl von Bytes dargestellt. So können alle Unicode-Zeichen (2^{16} an der Zahl) ausgegeben werden, ohne eine Speicherplatzverschwendung durch führende Null-Bytes bei den sehr oft auftretenden Zeichen mit Unicode-Nummern ≤ 127 in Kauf nehmen zu müssen: [1]

Unicode-Zeichen		Anzahl Bytes
von	**bis**	
\u0000	\u0000	2
\u0001	\u007F	1
\u0080	\u07FF	2
\u0800	\uFFFF	3

Bei einigen Überladungen des **StreamWriter**-Konstruktors lassen sich auch alternative Codierungen einstellen, z. B.:

```
FileStream fs = new("unicode.txt", FileMode.Create);
StreamWriter swUnicode = new(fs, Encoding.Unicode);
```

Die statische Eigenschaft **Unicode** der Klasse **Encoding** im Namensraum **System.Text** zeigt auf ein Objekt der Klasse **UnicodeEncoding**, das die Codierung übernimmt. Es verzichtet auf Platzsparmaßnahmen und verwendet für *jedes* Zeichen 2 Bytes.

16.3.3 Textdateien mit der Codierung Windows-1252 lesen

Auch bei Objekten der Klasse **StreamReader** lässt sich die voreingestellte UTF-8 - Codierung per Konstruktorparameter ersetzen, was z. B. erforderlich ist beim Lesen von Textdateien mit der ANSI-Codierung (korrekte Bezeichnung: *Windows-1252*), die in der Windows-Welt noch häufig anzutreffen sind. Die Codierung Windows-1252 wird im .NET - Framework ohne weiteres unterstützt durch Verwendung der voreingestellten Codierung:

---

[1] Im Bereich von 0 bis 127 befinden sich im Unicode dieselben Zeichen wie im ASCII-Code (*American Standard Code for Information Interchange*) aus der Steinzeit der Datenverarbeitung.

```
var sr = new StreamReader(fs, Encoding.Default);
```

In .NET 5.0 wird hingegen nur die verwandte, aber nicht identische Codierung ISO-8859-1 (z. B. ohne € - Zeichen) direkt unterstützt.[1] Mit Hilfe der Klasse **CodePagesEncodingProvider** ist das Problem aber leicht zu lösen.[2] Im folgenden Programm

```
using System;
using System.IO;
using System.Text;

class AnsiTextLesen {
    static string name = "Ansi.txt";
    static void Main() {
        using var fs = new FileStream(name, FileMode.Open, FileAccess.Read);
        var sr = new StreamReader(fs);
        Console.WriteLine("Mit UTF-8 - Codierung gelesen:");
        while (sr.Peek() >= 0)
            Console.WriteLine(sr.ReadLine());
        fs.Position = 0;
        var enc1252 = CodePagesEncodingProvider.Instance.GetEncoding(1252);
        sr = new StreamReader(fs, enc1252);
        Console.WriteLine("\nMit ANSI - Codierung gelesen:");
        while (sr.Peek() >= 0)
            Console.WriteLine(sr.ReadLine());
    }
}
```

wird beim Lesen einer Textdatei zuerst die UTF-8 - und dann die ANSI-Codierung unterstellt. Bei einer Datei mit ANSI-Codierung und dem folgenden Inhalt

```
ANSI-kodierte Umlaute: üöä
```

resultiert die Ausgabe:

```
Mit UTF-8 - Codierung gelesen:
ANSI-kodierte Umlaute: ??

Mit ANSI - Codierung gelesen:
ANSI-kodierte Umlaute: üöä
```

In der folgenden Anweisung[3]

```
var enc1252 = CodePagesEncodingProvider.Instance.GetEncoding(1252);
```

wird von einem Objekt der Klasse **EncodingProvider**, auf das die statische Eigenschaft **Instance** der Klasse **CodePagesEncodingProvider** zeigt, als Rückgabe der Methode **GetEncoding()** die benötigte Codierung geliefert.

---

[1] Hier
https://docs.microsoft.com/de-de/dotnet/api/system.text.encoding?view=net-5.0
befindet sich eine Liste der in .NET 5.0 unterstützten Codierungen, die erheblich kürzer ausfällt als die analoge Liste für das .NET Framework.

[2] Seit dem 18.2.2021 ist die Klasse **CodePagesEncodingProvider** im SDK von .NET 5.0 vorhanden, sodass die Installation des NuGet-Pakets **System.Text.Encoding.CodePages** nicht mehr erforderlich ist.

[3] https://stackoverflow.com/questions/37870084/net-core-doesnt-know-about-windows-1252-how-to-fix

16.4 Schreiben und Lesen von kleinen Dateien mit Bytes oder Texten

Die in den Abschnitten 16.1 bzw. 16.3.2 beschriebenen Methoden zum Schreiben und Lesen von **byte**-Arrays bzw. Texten durch die mehr oder weniger explizite Verwendung von Datenstromklassen erlauben viel Flexibilität, z. B.:

- Man kann portionsweise in einen Strom schreiben bzw. von dort lesen, sodass nicht alle Daten gleichzeitig bereitstehen bzw. befördert werden müssen. Das ist z. B. sinnvoll, wenn kontinuierlich per Netzwerk eintreffende Daten in eine Datei geschrieben werden sollen. Zwischen zwei Schreib- bzw. Leseaufträgen werden die beteiligten Ströme offengehalten.
- Bei großen Dateien spricht der Hauptspeicherbedarf dagegen, die Dateien *komplett* einzulesen. Was mit den *kleinen Dateien* in der Überschrift des aktuellen Abschnitts konkret gemeint ist (z. B. Dateigröße < 100 MB), hängt auch vom verfügbaren Hauptspeicher ab.
- Über die Parameter **FileMode**, **FileAccess** und **FileShare** im **FileStream**-Konstruktor lassen sich der Eröffnungsmodus sowie die eigenen und die fremden Zugriffsrechte für eine Datei regeln (vgl. Abschnitt 16.1.4).
- Mit der Eigenschaft **Position** bzw. mit der Methode **Seek()** kann man den Dateizeiger absolut oder relativ zu einem Bezugspunkt (z. B. Start, Ende) neu setzen.

Diese Flexibilität ist aber speziell bei kleinen Dateien oft überflüssig, und dann bietet die Klasse **File** bequeme statische Methoden zum Lesen oder Schreiben von **byte**-Arrays oder Texten. Dabei kommen die Datenstromklassen (**FileStream**, **StreamWriter** und **StreamReader**) durchaus weiterhin zum Einsatz, doch führt die gekapselte Verwendung zu einem sehr einfachen und gut lesbaren Quellcode.

Die Kompetenzen der Klasse **File** zur Dateiverwaltung (z. B. Löschen, Kopieren, Existenzprüfung) werden im Abschnitt 16.6.1 behandelt.

Von den anschließend vorgestellten Methoden werden die beteiligten Dateien nach dem (gelungenen oder gescheiterten) Lesen bzw. Schreiben automatisch geschlossen.

Mit der statischen **File**-Methode **WriteAllBytes()** befördert man einen **byte**-Array in eine Datei:

public static void WriteAllBytes(String *path***, byte[]** *bytes***)**

Soll eine Datei komplett in einen **byte**-Array eingelesen werden, bietet die statische **File**-Methode **ReadAllBytes()** eine bequeme Lösung:

public static byte[] ReadAllBytes(String *path***)**

Das im Abschnitt 16.1.2 vorgestellte Beispielprogramm zum Schreiben und Lesen eines **byte**-Arrays lässt sich mit Hilfe der Klasse **File** vereinfachen:

Quellcode	Ausgabe

`using System;`
`using System.IO;`
 	
`class WriteReadAllBytes {`
`  static void Main() {`
`  string name = "demo.bin";`
`  byte[] arr = { 0, 1, 2, 3, 4, 5, 6, 7 };`
`  File.WriteAllBytes(name, arr);`
`  arr = File.ReadAllBytes(name);`
`  foreach (byte b in arr)`
`  Console.WriteLine(b);`
`  File.Delete(name);`
`  }`
`}`
	`0`
	`1`
	`2`
	`3`
	`4`
	`5`
	`6`
	`7`

Ein Blick in den Quellcode der **File**-Methode **WriteAllBytes()** aus der BCL von .NET 5.0[1]

```
using (FileStream fs = new FileStream(path, FileMode.Create, FileAccess.Write,
       FileShare.Read)) {
    fs.Write(bytes, 0, bytes.Length);
}
```

zeigt, ...

- dass die Klasse **FileStream** zum Einsatz kommt,
- wie der Eröffnungsmodus sowie die eigenen und die fremden Zugriffsrechte für die Datei geregelt sind. Der Eröffnungsmodus **FileMode.Create** sorgt dafür, dass eine vorhandene Datei überschrieben wird.

Mit der statischen **File**-Methode **WriteAllLines()** befördert man einen **String**-Array in eine Textdatei, wobei die UTF-8 - Codierung verwendet wird:

public static void WriteAllLines(String *path***, String[]** *lines***)**

Eine vorhandene Datei wird dabei überschrieben. Mit der Methode **AppendAllLines()** lässt sich eine vorhandene Datei erweitern.

Soll eine Textdatei mit Zeilenstruktur komplett in einen **String**-Array eingelesen werden, dann bietet die statische **File**-Methode **ReadAllLines()** eine bequeme Lösung:

public static String[] ReadAllLines (String *path***)**

Das im Abschnitt 16.3.2 vorgestellte Beispielprogramm zum Schreiben und Lesen von Textzeilen lässt sich mit Hilfe der Klasse **File** vereinfachen:

```
using System;
using System.IO;

class WriteReadAllLines {
    static void Main() {
        string name = "demo.txt";
        string[] text = { "Zeile 1", "Zeile 2", "Nicht übel" };
        File.WriteAllLines(name, text);

        text = File.ReadAllLines(name);
        foreach (var zeile in text)
            Console.WriteLine(zeile);
    }
}
```

[1] Der BCL-Quellcode ist hier zu finden: https://source.dot.net/

Bislang wurden synchrone, blockierende Ein-/Ausgabemethoden der Klasse **File** beschrieben. Dazu stehen asynchrone, nicht blockierende Alternativen zur Verfügung (**ReadAllBytesAsync()**, **ReadAllLinesAsync()**, **WriteAllBytesAsync()**, **WriteAllLinesAsync()**).

Zum Lesen und Schreiben von Werten mit elementarem Datentyp (**double**, **int**, etc.) enthält die Klasse **File** keine Alternativen zu den Methoden der im Abschnitt 16.3.1 beschriebenen Klassen **BinaryWriter** und **BinaryReader**.

16.5 Serialisieren von Instanzen

Wer objektorientiert programmiert, möchte natürlich auch objektorientiert speichern, laden und mit anderen Programmen kommunizieren. Erfreulicherweise können in C# Objekte und Strukturinstanzen tatsächlich in der Regel genauso einfach wie Werte mit einem elementaren Datentyp in eine Senke geschrieben bzw. von einer Quelle gelesen werden. Als Senke oder Quelle kommen beim Serialisieren in Frage:

- Dateien und Datenbanken
  So lassen sich Zustände von Instanzen über Programmeinsätze hinweg konservieren.
- Netzwerkverbindungen
  Instanzen können zwischen Anwendungen ausgetauscht werden, wobei dank offener Standards wie JSON (*JavaScript Object Notation*) auch unterschiedliche Anwendungsarchitekturen beteiligt sein können. Das ermöglicht z. B. den Datenaustausch zwischen einem in C# programmierten Server und einem in JavaScript programmierten Klienten.
- Variablen im Hauptspeicher
  Über eine solche Senke bzw. Quelle im Hauptspeicher, an die man beim Serialisieren nicht unbedingt zuerst denkt, lässt sich z. B. eine tiefe Kopie einer Instanz erstellen, wobei (im Unterschied zur flachen Kopie) auch alle direkten und indirekten Member-Objekte dupliziert werden.

Wegen der Aufgabenvielfalt und aus historischen Gründen enthält die .NET - Plattform gleich vier Serialisierungs-Lösungen. In der folgenden Liste werden jeweils die zentralen Klassen genannt:

- **BinaryFormatter** im Namensraum **System.Runtime.Serialization.Formatters**
  Diese traditionelle, performante und angenehm einfache Lösung darf wegen Sicherheitsrisiken nur noch in einem kontrollierten Kontext eingesetzt werden.
- **JsonSerializer** im Namensraum **System.Text.Json**
  Das JSON-Format ist aktuell (2021) bei der Plattform-übergreifenden Kooperation die erste Wahl.
- **XmlSerializer** im Namensraum **System.Xml.Serialization**
  Diese Klasse bietet sich an, wenn eine XML-basierte Ausgabe gefragt ist.
- **DataContractSerializer** im Namensraum **System.Runtime.Serialization**
  Diese Lösung gehört zur *Windows Communication Foundation* (WCF), einer Bibliothek für komplexe, Dienst-orientierte Netzwerklösungen, die von Microsoft mittlerweile aufgegeben wurde. In .NET Core (inklusive .NET 5.0) ist die WCF *nicht* mehr enthalten.[1]

Im Manuskript werden die beiden ersten Lösungen behandelt.

[1] https://docs.microsoft.com/de-de/dotnet/framework/wcf/whats-wcf

16.5.1 Binäres Serialisieren

Die ursprüngliche Serialisierungs-Lösung der .NET - Plattform basiert auf einem binären Datenformat, ist perfomant und angenehm einfach handhabbar. Wie man seit einigen Jahren weiß, bestehen leider gravierende Sicherheitsprobleme, die insbesondere das binäre Deserialisieren von Daten aus unsicheren Quellen betreffen.[1]

16.5.1.1 Sicherheit

Auffälligster Bestandteil in der BCL-Dokumentation zur zentralen Klasse **BinaryFormatter** der binären Serialisierung ist die folgende Warnung:[2]

> **BinaryFormatter is insecure and can't be made secure. For more information, see the BinaryFormatter security guide.**

Wer aus unsicherer Quelle stammende Byte-Ströme bedenkenlos deserialisiert, riskiert böswillige Angriffe durch manipulierte Ströme. Von diesem Problem ist nicht nur die .NET - Plattform betroffen, sondern z. B. auch Java (Bloch 2018, S. 339ff). Es wird empfohlen, so schnell wie möglich auf das binäre Serialisieren zu verzichten und stattdessen z. B. die textorientierte, offene JSON-Serialisierung zu verwenden (siehe Abschnitt 16.5.2).[3]

Allerdings ist das binäre Serialisieren auch in .NET 5.0 noch erlaubt und kann in vorhandenen Programmen weiterhin verwendet werden, wenn das Deserialisieren von Daten aus unsicheren Quellen ausgeschlossen ist.

16.5.1.2 Steuerung der (De-)Serialisierung

Das Serialisieren der Instanzen eines Typs muss explizit bei der Definition über das Typattribut **Serializable** erlaubt werden. Dies darf nicht unbedacht geschehen, z. B. weil die Serialisierbarkeit aller Datentypen von Feldern erforderlich ist. Aus naheliegenden Gründen haben die BCL-Designer das Attribut **Serializable** als *nicht* vererbbar definiert, sodass es nicht auf abgeleitete Klassen übertragen wird.

Bei Bedarf können einzelne Felder über das Attribut **NonSerialized** von der Serialisierung ausgeschlossen werden. Das kommt z. B. in Frage, wenn ...

- ein Feld aus Datenschutzgründen nicht in den Ausgabestrom gelangen soll,
- ein Feld temporäre Daten enthält, sodass ein Speichern überflüssig bzw. sinnlos ist,
- ein Feld einen nicht-serialisierbaren Datentyp hat.
  Wird ein solches Feld nicht von der Serialisierung ausgeschlossen, kommt es zu einer **SerializationException**.

Möchte ein Typdesigner bei der (De)serialisierung nicht auf die Automatik vertrauen, sondern selbst die Kontrolle übernehmen, muss er die Schnittstelle **ISerializable** implementieren, welche die Methode **GetObjectData()** vorschreibt.

Wir beschränken uns im weiteren Verlauf des Abschnitts auf das (De)serialisieren von Objekten (im Unterschied zu Strukturinstanzen). Die im folgenden Quellcode definierte Klasse `Kunde` ist als serialisierbar deklariert, wobei jedoch für das Feld `kredit` eine Ausnahme gemacht wird:

---

1 https://docs.microsoft.com/de-de/dotnet/standard/serialization/binary-serialization

2 https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.formatters.binary.binaryformatter?view=net-5.0

3 https://docs.microsoft.com/de-de/dotnet/standard/serialization/binaryformatter-security-guide

```
using System;
using System.Collections.Generic;

[Serializable]
public class Rechnung {
   public DateTime Datum { get; set; }
   public double Betrag { get; set; }
}

[Serializable]
public class Kunde {
   public int Nr { get; private set; }
   public string Vorname { get; set; }
   public string Name { get; set; }
   readonly List<Rechnung> Rechnungen = new();
   [NonSerialized]
   bool kredit;

   public Kunde(int nr, string vorname, string name, bool kredit) {
      Nr = nr;
      Vorname = vorname;
      Name = name;
      this.kredit = kredit;
   }

   public void Kaufen(DateTime datum, double betrag) {
      Rechnungen.Add(new Rechnung { Datum = datum, Betrag = betrag });
   }

   public void Prot() {
      Console.WriteLine("\nKundennummer: \t" + Nr);
      Console.WriteLine("Name: \t\t" + Vorname + " " + Name);
      Console.WriteLine("Kredit: \t" + kredit);
      Console.WriteLine("Rechnungen:");
      foreach (var re in Rechnungen)
         Console.WriteLine($" Datum: {re.Datum:d}, Betrag: {re.Betrag}");
   }
}
```

Ein Objekt vom Typ `Kunde` enthält ein Member-Objekt vom Kollektionstyp **List**<`Rechnung`>. Damit ein `Kunde`-Objekt serialisiert werden kann, muss auch die Klasse `Rechnung` das Attribut **Serializable** besitzen. Ist das nicht der Fall, kommt es beim Aufruf der **BinaryFormatter**-Methode **Serialize()** zu einem Ausnahmefehler:

```
Unhandled exception. System.Runtime.Serialization.SerializationException: Type
'Rechnung' in Assembly 'BinarySerialization, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null' is not marked as serializable.
```

Eigentlich sollte der Compiler warnen, wenn eine serialisierbare Klasse ein Member-Objekt mit einem nicht-serialisierbaren Typ enthält.

16.5.1.3 Die Klasse BinaryFormatter

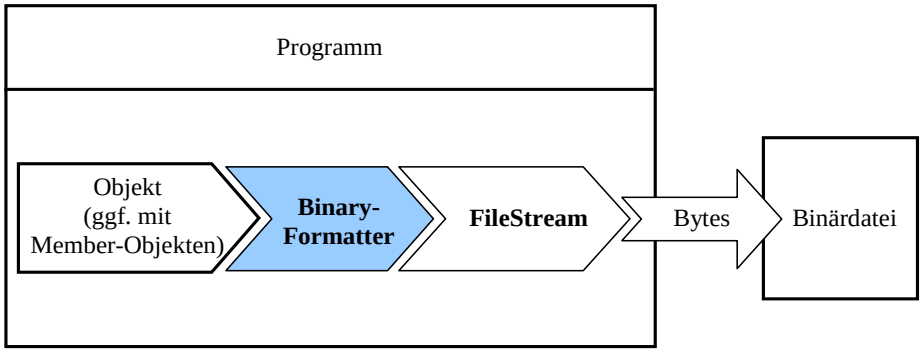
Beim binären (De)Serialisieren werden anspruchsvolle Leistungen erbracht:

- Beim Serialisieren einer Instanz werden alle Felder und die eventuell von Feldern referenzierten Objekte in einen Byte-Strom überführt.
- Beim Deserialisieren einer Instanz werden alle Felder sowie Member-Objekte wiederhergestellt und die Referenzen zwischen den Objekten restauriert.

Die BCL-Dokumentation spricht vom (De)serialisieren eines *Instanzdiagramms*.

Um diese Aufgaben kümmert sich ein Objekt aus einer Klasse, die das Interface **IFormatter** (aus dem Namensraum **System.Runtime.Serialization**) implementiert. Wir arbeiten anschließend mit der Klasse **BinaryFormatter** (aus dem Namensraum **System.Runtime.Serialization.Formatters.Binary**).

Beim Serialisieren in eine Datei besteht die folgende Verarbeitungskette:



Im folgenden Programm wird ein ...

- Kollektionsobjekt vom Typ **List**<Kunde>
- mit zwei Kunde-Elementobjekten
- mit den jeweils enthaltenen Instanzvariablen bzw. -objekten vom Typ **int**, **String** bzw. **List**<Rechnung>
- aber ohne die Instanzvariable stimmung

(de)serialisiert:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

class BinaryFormatterDemo {
    static string name = "demo.bin";
    static void Main() {
        var kunden = new List<Kunde> {
            new Kunde(1, "Fritz", "Orth", true),
            new Kunde(2, "Ludwig", "Knüller", false)
        };

        kunden[0].Kaufen(new DateTime(2021, 03, 28), 25.0);
        kunden[0].Kaufen(new DateTime(2021, 03, 30), 120.0);
        kunden[1].Kaufen(new DateTime(2021, 03, 29), 75.0);

        Console.WriteLine("Zu sichern:\n");
        foreach (Kunde k in kunden)
            k.Prot();

        var desKunden = new List<Kunde>();
        using (var fs = new FileStream(name, FileMode.Create)) {
            var bifo = new BinaryFormatter();
            bifo.Serialize(fs, kunden);
            fs.Position = 0;
            Console.WriteLine("\nRekonstruiert:");
            desKunden = (List<Kunde>) bifo.Deserialize(fs);
        }
        foreach (Kunde k in desKunden)
            k.Prot();
    }
}
```

Pro **Serialize()** - Aufruf wird *ein* Wurzelobjekt mitsamt den Werttyp-Feldern sowie den direkt oder indirekt referenzierten Objekten, also ein komplettes Instanzdiagramm, geschrieben. Um weitere Instanzdiagramme in denselben Datenstrom zu befördern, sind entsprechend viele Aufrufe erforderlich.

Beim Lesen eines Objekts durch die Methode **Deserialize()** wird zunächst die zugehörige Klasse festgestellt und in die Laufzeitumgebung geladen (falls noch nicht vorhanden). Dann wird das Objekt auf dem Heap angelegt, und die Instanzvariablen erhalten die rekonstruierten Werte, wobei *kein* Konstruktor aufgerufen wird. Das ganze wiederholt sich (ggf. auf mehreren Ebenen) für die referenzierten Objekte.

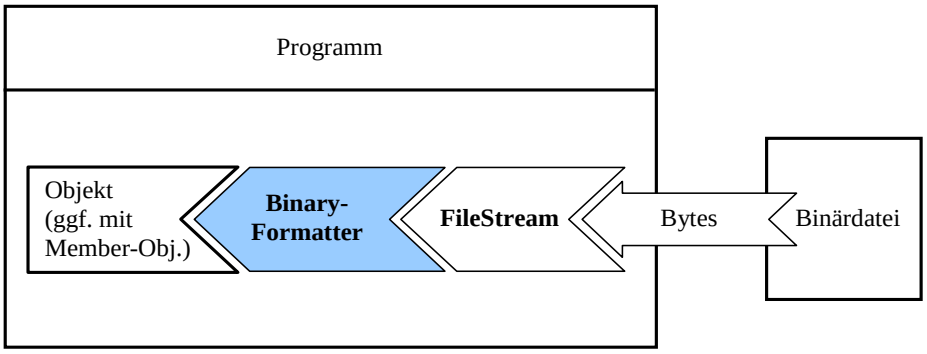
Weil **Deserialize()** den Rückgabewert **Object** hat, ist eine explizite Typumwandlung erforderlich. Ein **Deserialize()** - Aufruf liest das nächste im Datenstrom befindliche Wurzelobjekt samt Anhang (also *ein* Instanzdiagramm). Um weitere Wurzelobjekte aus demselben Datenstrom zu lesen, sind entsprechend viele Aufrufe erforderlich.

Das Beispielprogramm produziert die folgende Ausgabe:

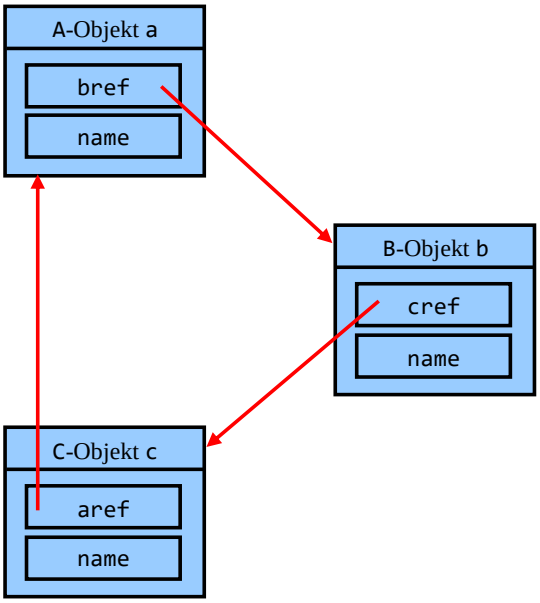
```
Zu sichern:

Kundennummer:   1
Name:           Fritz Orth
Kredit:         True
Rechnungen:
 Datum: 28.03.2021, Betrag: 25
 Datum: 30.03.2021, Betrag: 120

Kundennummer:   2
Name:           Ludwig Knüller
Kredit:         False
Rechnungen:
 Datum: 29.03.2021, Betrag: 75

Rekonstruiert:

Kundennummer:   1
Name:           Fritz Orth
Kredit:         False
Rechnungen:
 Datum: 28.03.2021, Betrag: 25
 Datum: 30.03.2021, Betrag: 120

Kundennummer:   2
Name:           Ludwig Knüller
Kredit:         False
Rechnungen:
 Datum: 29.03.2021, Betrag: 75
```

Die Instanzvariable `kredit` der eingelesenen Kunden besitzt den Initialwert **false**, während die übrigen Instanzvariablen bzw. Eigenschaften bei der (De)serialisierung ihre Werte behalten.

In der folgenden Abbildung wird die Rekonstruktion eines Wurzelobjekts skizziert:



Zirkuläre Referenzen wie in der folgenden Situation



bringen den **BinaryFormatter** nicht aus dem Tritt. Wenn man Objekte aus den Klassen `A`, `B` und `C` geeignet initialisiert

```
A a = new A(); B b = new B(); C c = new C();
a.bref = b; a.name = "a-Obj";
b.cref = c; b.name = "b-Obj";
c.aref = a; c.name = "c-Obj";
```

und anschließend das über die Referenzvariable `a` ansprechbare Objekt serialisiert,

```
using (var fs = new FileStream("demo.bin", FileMode.Create)) {
    IFormatter bifo = new BinaryFormatter();
    bifo.Serialize(fs, a);
     . . .
}
```

dann landen alle *drei* Objekte im Datenstrom. Beim Deserialisieren des Wurzelobjekts

```
fs.Position = 0;
A na = (A) bifo.Deserialize(fs);
Console.WriteLine("Rekonstruiert: " + na.bref.cref.aref.name);
```

werden auch die beiden anderen Objekte rekonstruiert, sodass der **WriteLine()** - Aufruf zu der folgenden Ausgabe führt:

```
Rekonstruiert: a-Obj
```

Beim Deserialisieren entstehen *neue* Objekte, sodass im Beispiel die Referenzvariable `b` *nicht* auf das restaurierte Objekt der Klasse `B` zeigt, und der Ausdruck

```
b == na.bref
```

den Wert **false** besitzt.

Über die vom **BinaryFormatter** unterstützte *Versions-tolerante Serialisierung* wird es möglich, eine Klasse weiterzuentwickeln, ohne dass konservierte Objekte mit einem älteren Versionsstand inkompatibel werden.[1]

[1] https://docs.microsoft.com/de-de/dotnet/standard/serialization/version-tolerant-serialization

16.5.2 Serialisieren im JSON-Format

Zum Austausch von kompletten Objekten bzw. Instanzen zwischen Anwendungen mit unterschiedlichen Architekturen hat sich der offene, textbasierte Standard JSON (*JavaScript Object Notation*) etabliert.[1] In der .NET - Szene wurde zur Realisation des JSON-Formats lange das NuGet-Paket **Newtonsoft.Json** bevorzugt (siehe Abschnitt 3.3.6.1.2). Es wird auch weiterhin in vielen Lehrbüchern als Standardlösung beschrieben (z. B.in Griffiths 2020). Mit .NET Core 3.0 hat Microsoft aber in den Namensräumen **System.Text.Json** und **System.Text.Json.Serialization** eine eigene Lösung eingeführt, die mit .NET 5.0 eine hohe Praxistauglichkeit erreicht hat und aktuell die besten Zukunftsaussichten besitzt.

Wir verwenden zur Erläuterung der JSON - Serialisierung weitgehend dasselbe Anwendungsbeispiel wie im Abschnitt 16.5.1 über das binäre Serialisieren. Im Beispiel wird ein Kollektionsobjekt vom Typ **List**<Kunde> serialisiert, wobei ein Objekt vom Typ Kunde wiederum ein Member-Objekt vom Kollektionstyp **List**<Rechnung> enthält:

```
using System;
using System.Collections.Generic;
using System.Text.Json.Serialization;

public class Rechnung {
   public DateTime Datum { get; set; }
   public double Betrag { get; set; }
}

public class Kunde {
    public int Nr { get; private set; }
    public string Vorname { get; set; }
    public string Name { get; set; }
    public List<Rechnung> Rechnungen { get; }
    [JsonIgnore] // Eigenschaft wird nicht serialisiert.
    public bool Kredit { get; private set; }

    public Kunde(int nr, string vorname, string name, bool kredit,
                 List<Rechnung> rechnungen) {
        Nr = nr;
        Vorname = vorname;
        Name = name;
        Kredit = kredit;
        Rechnungen = rechnungen;
    }

    [JsonConstructor] // Konstruktor wird beim Json-Deserialisieren verwendet.
    public Kunde(int nr, string vorname, string name, List<Rechnung> rechnungen) {
        Nr = nr;
        Vorname = vorname;
        Name = name;
        Rechnungen = rechnungen;
    }

    public void Kaufen(DateTime datum, double betrag) {
        Rechnungen.Add(new Rechnung { Datum = datum, Betrag = betrag });
    }
    public void Prot() {
        Console.WriteLine("\nKundennummer: \t" + Nr);
        Console.WriteLine("Name: \t\t" + Vorname + " " + Name);
        Console.WriteLine("Kredit: \t" + Kredit);
        Console.WriteLine("Rechnungen:");
        foreach (var re in Rechnungen)
            Console.WriteLine($" Datum: {re.Datum:d}, Betrag: {re.Betrag}");
    }
}
```

[1] https://www.json.org/

16.5.2.1 Serialisieren

In der **Main()** - Methode der Startklasse `JsonSerializerDemo` wird eine Kundenliste namens `kunden` angelegt und befüllt:

```
using System;
using System.Collections.Generic;
using System.Text.Json;

class JsonSerializerDemo {
    static void Main() {
        var kunden = new List<Kunde>();
        kunden.Add(new Kunde(1, "Fritz", "Orth", true, new List<Rechnung>()));
        kunden.Add(new Kunde(2, "Ludwig", "Knüller", false, new List<Rechnung>()));
        kunden[0].Kaufen(new DateTime(2021, 03, 28), 25.0);
        kunden[0].Kaufen(new DateTime(2021, 03, 30), 120.0);
        kunden[1].Kaufen(new DateTime(2021, 03, 29), 75.0);

        Console.WriteLine("Zu sichern:");
        foreach (Kunde k in kunden)
            k.Prot();

        var jsonOptions = new JsonSerializerOptions {
            WriteIndented = true,
            IncludeFields = true,
            PropertyNamingPolicy = JsonNamingPolicy.CamelCase
        };

        string kundenJS = JsonSerializer.Serialize(kunden, jsonOptions);

        Console.WriteLine("\nJson-String:\n" + kundenJS);

        var desKunden = JsonSerializer.Deserialize<List<Kunde>>(kundenJS, jsonOptions);
        Console.WriteLine("\nRekonstruiert:");
        foreach (Kunde k in desKunden)
            k.Prot();
    }
}
```

Die Klasse **JsonSerializer** spielt im Namensraum **System.Text.Json** die zentrale Rolle. Mit dem folgenden Aufruf ihrer statischen Methode **Serialize()** wird das Kollektionsobjekt `kunden` in die **String**-Variable `kundenJS` befördert:

```
string kundenJS = JsonSerializer.Serialize(kunden, jsonOptions);
```

Wir verwenden eine *Variable* als Senke (und gleich beim Deserialisieren als Quelle), was nicht ganz in das Kapitel über Dateiverarbeitung zu passen scheint. Allerdings wäre es nicht mehr als eine lästige Pflichtübung, die Zeichenfolge `kundenJS` in eine Datei zu befördern und von dort zurückzuholen (siehe Abschnitte 16.3.2 und 16.6.1).

Zum persistenten Speichern von JSON-Dokumenten kommen neben Textdateien auch Datenbanken in Frage. Üblicherweise werden die sogenannten *NoSQL* - Datenbanken (mit MongoDB als dem prominentesten Vertreter) zum Speichern von JSON-Dokumenten bevorzugt. Mittlerweile bieten aber auch einige relationale Datenbanken eine spezielle Unterstützung von JSON-Dokumenten (z. B. bei Abfragen) an.[1]

So sieht das Kollektionsobjekt mit den Kunden im JSON-Format aus:

[1] https://docs.microsoft.com/de-de/sql/relational-databases/json/store-json-documents-in-sql-tables

```
[
  {
    "nr": 1,
    "vorname": "Fritz",
    "name": "Orth",
    "rechnungen": [
      {
        "datum": "2021-03-28T00:00:00",
        "betrag": 25
      },
      {
        "datum": "2021-03-30T00:00:00",
        "betrag": 120
      }
    ]
  },
  {
    "nr": 2,
    "vorname": "Ludwig",
    "name": "Kn\u00FCller",
    "rechnungen": [
      {
        "datum": "2021-03-29T00:00:00",
        "betrag": 75
      }
    ]
  }
]
```

Die äußere Kollektion (mit den Kunden) und die innere Kollektion (mit den Rechnungen eines Kunden) sind als JSON-Arrays (begrenzt durch Paare eckiger Klammern) realisiert. Ein Objekt (aus den Klassen `Kunde` bzw. `Rechnung`) wird jeweils durch geschweifte Klammern begrenzt, und die Eigenschaften eines Objekts sind als (Name: Wert) - Paare notiert.[1]

16.5.2.2 Steuerung der (De-)Serialisierung

Zur Steuerung der JSON - (De-)Serialisierung kann man (teilweise alternativ) verwenden:

- Ein Objekt vom Typ der **JsonSerializerOptions**, das den **JsonSerializer**-Methoden **Serialize()**und **Deserialize()** als Aktualparameter übergeben wird.
- Attribute im Namensraum **System.Text.Json.Serialization**, mit denen Eigenschaften oder Konstruktoren der zu serialisierenden Typen dekoriert werden.

Übersichtliches Serialisierungs-Ergebnis durch Einrückungen

Die **JsonSerializer**-Methode **Serialize()** liefert per Voreinstellung ein Platz sparendes, relativ unübersichtliches Ergebnis, z. B.:

```
[{"nr":1,"vorname":"Fritz","name":"Orth","rechnungen":[{"datum":"2021-03-
28T00:00:00","betrag":25},{"datum":"2021-03-30T00:00:00","betrag":120}]}, ...]
```

Um ein ansehnlicheres Resultat mit Einrückungen zu erhalten (siehe Abschnitt 16.5.2.1), verwendet man beim Aufruf der **JsonSerializer**-Methode **Serialize()** ein Parameterobjekt vom Typ **JsonSerializerOptions**, das für die Eigenschaft **WriteIndented** den Wert **true** besitzt, z. B.:

```
var jsonOptions = new JsonSerializerOptions { WriteIndented = true };
```

Öffentliche Eigenschaften ohne öffentliche set-Methode in die Deserialisierung einbeziehen

Öffentliche Eigenschaften werden in die JSON - (De-)Serialisierung einbezogen. Wenn die **set**-Methode einer Eigenschaft wie im folgenden Beispiel

```
public int Nr { get; private set; }
```

---

[1] Zur JSON-Syntax siehe z. B. https://www.w3schools.com/js/js_json_syntax.asp

den Zugriffsmodifikator **private** besitzt, dann ist die Eigenschaft per Voreinstellung von der **De**serialisierung ausgeschlossen. Seit .NET 5.0 bestehen zwei Optionen, um diese Einschränkung aufzuheben:

- Die Eigenschaft wird mit dem Attribut **JsonInclude** dekoriert, z. B.:
  ```
  [JsonInclude]
  public int Nr { get; private set; }
  ```
- Die Eigenschaft wird beim Deserialisieren in einem öffentlichen, parametrisierten Konstruktor gesetzt. Sind mehrere parametrisierte Konstruktoren vorhanden, ist der beim Deserialisieren zu verwendende durch das Attribut **JsonConstructor** zu dekorieren (siehe unten).

Um öffentliche Eigenschaften ohne **set**-Methode (im Beispiel: `Rechnungen`) zu deserialisieren, wird ein parametrisierter Konstruktor benötigt. Sind mehrere parametrisierte Konstruktor-Überladungen vorhanden, ist die beim Deserialisieren zu verwendende Überladung durch das Attribut **JsonConstructor** zu dekorieren. Im Beispiel muss zum Deserialisieren diejenige Konstruktor-Überladung der Klasse `Kunde` verwendet werden, die *keinen* Parameter für die explizit durch das Attribut **JsonIgnore** vom Serialisieren ausgeschlossene Eigenschaft `Kredit` besitzt:

```
[JsonConstructor]
public Kunde(int nr, string vorname, string name, List<Rechnung> rechnungen) {
    Nr = nr;
    Vorname = vorname;
    Name = name;
    Rechnungen = rechnungen;
}
```

Öffentliche Felder in die (De-)Serialisierung einbeziehen

Per Voreinstellung werden nur öffentliche *Eigenschaften* in die JSON - (De-)Serialisierung einbezogen. Um auch öffentliche *Felder*

```
public int WaldWiese;
```

einzubeziehen, bestehen seit .NET 5.0 die folgenden Optionen:

- Ein betroffenes Feld wird mit dem Attribut **JsonInclude** dekoriert, z. B.:
  ```
  [JsonInclude]
  public int WaldWiese;
  ```
- Beim Aufruf der **JsonSerializer**-Methoden **Serialize()** und **Deserialize()** wird ein Parameterobjekt vom Typ **JsonSerializerOptions** verwendet, das den Wert **true** für die Eigenschaft **IncludeFields** besitzt, z. B.:
  ```
  var jsonOptions = new JsonSerializerOptions { IncludeFields = true };
   . . .
  string kundenJS = JsonSerializer.Serialize(kunden, jsonOptions);
   . . .
  var desKunden = JsonSerializer.Deserialize<List<Kunde>>(kundenJS, jsonOptions);
  ```

Eigenschaften von der Serialisierung ausschließen

Um eine öffentliche Eigenschaft von der JSON-Serialisierung auszuschließen, dekoriert man sie mit dem Attribut **JsonIgnore**, z. B.:

```
[JsonIgnore]
public bool Kredit { get; private set; }
```

Camel Casing im Serialisierungs-Ergebnis

Im Ergebnis der **JsonSerializer**-Methode **Serialize()** wird per Voreinstellung in den Namen von Eigenschaften und Feldern die Groß-/Kleinschreibung aus der Typdefinition übernommen, z. B.:

```
{
  "Nr": 1,
  "Vorname": "Fritz",
  "Name": "Orth",
  "Rechnungen": [
    {
      "Datum": "2021-03-28T00:00:00",
      "Betrag": 25
    },
    {
      "Datum": "2021-03-30T00:00:00",
      "Betrag": 120
    }
  ]
}
```

Im JSON-Format sollte allerdings für Namen das Camel Casing verwendet werden (vgl. Abschnitt 4.1.5).[1] Zur Anpassung an diese Konvention muss man beim Aufruf der **JsonSerializer**-Methode **Serialize()** ein Parameterobjekt vom Typ **JsonSerializerOptions** verwenden, das den Wert **JsonNamingPolicy.CamelCase** für die Eigenschaft **PropertyNamingPolicy** besitzt, z. B.:

```
var jsonOptions = new JsonSerializerOptions {
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase
};
```

Im Beispiel liefert die Methode **Serialize()** dann:

```
{
  "nr": 1,
  "vorname": "Fritz",
  "name": "Orth",
  "rechnungen": [
    {
      "datum": "2021-03-28T00:00:00",
      "betrag": 25
    },
    {
      "datum": "2021-03-30T00:00:00",
      "betrag": 120
    }
  ]
}
```

16.5.2.3 Deserialisieren

In der **Main()** - Methode der Klasse `JsonSerializerDemo` aus dem aktuellen Beispiel wird das Kundenlisten-Kollektionsobjekt durch den folgenden Aufruf der **JsonSerializer**-Methode **Deserialize()**

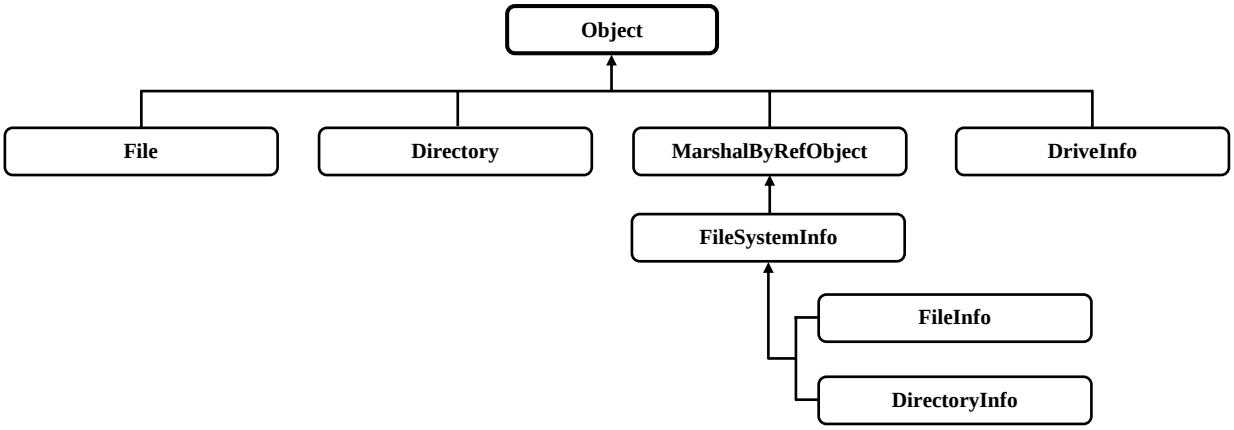
```
var desKunden = JsonSerializer.Deserialize<List<Kunde>>(kundenJS, jsonOptions);
```

in die **String**-Variable `desKunden` befördert.

16.6 Verwaltung von Dateien und Verzeichnissen

Zur Verwaltung von Dateien bzw. Verzeichnissen enthält die BCL jeweils eine Klasse mit statischen Methoden (**File** bzw. **Directory**) sowie eine Klasse mit Instanzmethoden (**FileInfo** bzw. **DirectoryInfo**):

[1] https://jsonapi.org/recommendations/



Mit Hilfe dieser Klassen kann man Dateien bzw. Verzeichnisse erstellen, kopieren, löschen, umbenennen und verschieben sowie diverse Datei- bzw. Verzeichnisattribute einsehen und verändern.

16.6.1 Dateiverwaltung

Im folgenden Beispielprogramm werden einige Methoden und Eigenschaften der Klassen **File** und **FileInfo** demonstriert:

```
using System;
using System.IO;

class Dateiverwaltung {
   static void Main() {
      const string pfad0 = @"U:\Eigene Dateien\C#\EA";
      const string pfad1 = @"U:\Eigene Dateien\C#\EA\demo.txt";
      const string pfad2 = @"U:\Eigene Dateien\C#\EA\kopie.txt";
      const string pfad3 = @"U:\Eigene Dateien\C#\EA\nn.txt";

      // Verzeichnis erstellen (setzt das LW U: voraus)
      Directory.CreateDirectory(pfad0);

      // File.CreateText(pfad1) entspricht new StreamWriter(pfad1).
      using (StreamWriter sw = File.CreateText(pfad1))
         sw.WriteLine("File-Demo");

      // Kopieren (Überschreiben erlaubt)
      File.Copy(pfad1, pfad2, true);
      Console.Write("Datei kopiert. Weiter mit Enter"); Console.ReadLine();

      File.Delete(pfad2);

      Console.WriteLine("\nExistiert " + pfad3 + "? " + File.Exists(pfad3));

      // Umbenennen (pfad1 u. pfad3 befinden sich im selben Ordner)
      File.Move(pfad1, pfad3);
      Console.WriteLine("\nDatei umbenannt. Weiter mit Enter"); Console.ReadLine();

      // Kreations- und Änderungsdatum setzen
      File.SetCreationTime(pfad3, new DateTime(2007, 12, 29, 22, 55, 44));
      File.SetLastWriteTime(pfad3, new DateTime(2005, 12, 29, 22, 55, 44));

      // Per FileInfo Dateieigenschaften ermitteln
      var fi = new FileInfo(pfad3);
      Console.WriteLine($"Die Datei {fi.Name} wurde\n  erstellt:\t\t{fi.CreationTime}" +
                        $"\n  zuletzt geändert:\t{fi.LastWriteTime}");
      Console.WriteLine("\nKreativdaten gesetzt. Weiter mit Enter"); Console.ReadLine();

      // Löschen per FileInfo-Instanzmethode
      fi.Delete();
   }
}
```

Wie im Abschnitt 4.3.10.5 über die Syntax von Zeichenkettenliteralen besprochen, wird mit dem Präfix „@“ vor einem Zeichenkettenliteral die Auswertung von Escape-Sequenzen abgeschaltet, sodass die in Windows-Pfadnamen üblichen Rückwärtsschrägstriche nicht mehr durch Verdoppeln von ihrer Sonderfunktion befreit werden müssen.

Von der Klasse **File** haben wir im Abschnitt 16.4 schon bequeme Verpackungsmethoden für die Datenstromklassen kennengelernt. Anschließend werden wichtige Dateiverwaltungsmethoden der Klasse **File** vorgestellt, die allesamt statisch sind und meist noch weitere Überladungen besitzen:

- **public static FileStream Create(String** *pfadname***)**
  Die **File**-Methode **Create()** erzeugt eine Datei und ein zugehöriges **FileStream**-Objekt, sodass die Datei anschließend mit dem **FileMode.Create** geöffnet ist. Die Anweisung
  ```
  FileStream fs = File.Create(pfad1)
  ```
  ist äquivalent mit
  ```
  FileStream fs = new FileStream(pfad1, FileMode.Create)
  ```
  bzw. (in der seit C# 9.0 erlaubten Schreibweise ohne Wiederholung der Klassennamens als Konstruktorname, siehe Abschnitt 5.4.3.2 über zieltypisierte **new**-Ausdrücke):
  ```
  FileStream fs = new(pfad1, FileMode.Create)
  ```
- **public static StreamWriter CreateText(String** *pfadname***)**
  Die **File**-Methode **CreateText()** erzeugt eine Datei und ein zugehöriges **StreamWriter**-Objekt (mit UTF-8 - Codierung). Es entsteht auch das vermittelnde **FileStream**-Objekt, und die Datei ist anschließend mit dem **FileMode.Create** geöffnet. Die Anweisung
  ```
  StreamWriter sw = File.CreateText(pfad1)
  ```
  ist äquivalent mit
  ```
  StreamWriter sw = new(pfad1)
  ```
- **public static bool Exists(String** *pfadname***)**
  Mit **File.Exists()** überprüft man die Existenz einer Datei.
- **public static void Delete(String** *pfadname***)**
  Mit **File.Delete()** löscht man eine Datei.
- **public static void Copy(String** *pfadQuelle***, String** *pfadZiel***, bool** *überschreiben***)**
  Bei dieser **Copy()** - Überladung erlaubt der Wert **true** des dritten Parameters das Überschreiben einer vorhandenen Zieldatei.
- **public static void Move(String** *pfadQuelle***, String** *pfadZiel***)**
  Zum Umbenennen oder Verschieben einer Datei verwendet man die **File**-Methode **Move()**. Bei identischem Ordner von Quelle und Ziel wird die Datei umbenannt, anderenfalls wird die Datei verschoben.
- **public static void SetCreationTime(String** *pfadname***, DateTime** *kreation***)**
  Mit der statischen **File**-Methode **SetCreationTime()** lässt sich für eine Datei das Kreationsdatum setzen.
- **public static void SetLastWriteTime(String** *pfadname***, DateTime** *letzteÄnderung***)**
  Mit der statischen **File**-Methode **SetLastWriteTime()** lässt sich für eine Datei das Datum der letzten Änderung setzen.

Zu praktisch allen **File**-Klassenmethoden finden sich Entsprechungen in der Klasse **FileInfo** als Instanzmethoden oder Eigenschaften. Im Beispiel wird das zu einer Datei erstellte **FileInfo**-Objekt dazu verwendet, ...

- um Eigenschaften der Datei zu ermitteln (z. B. **CreationTime** und **LastWriteTime**),
- um die Datei über die Instanzmethode **Delete()** zu löschen.

Wie die Ausgabe des Programms zeigt, lassen sich wichtige Dateieigenschaften leicht fälschen:

```
Die Datei nn.txt wurde
  erstellt:             29.12.2007 22:55:44
  zuletzt geändert:     29.12.2005 22:55:44
```

16.6.2 Ordnerverwaltung

Im folgenden Beispielprogramm werden einige Methoden der Klassen **Directory** und **DirectoryInfo** demonstriert:

```
using System;
using System.IO;

class Ordnerverwaltung {
   static void Main() {
      const string dir1 = @"U:\Eigene Dateien\C#\EA\";
      const string dir2 = @"U:\Eigene Dateien\C#\EA\Sub\";

      // Verzeichnis erstellen (setzt das LW U: voraus)
      Directory.CreateDirectory(dir2);

      Directory.SetCurrentDirectory(dir1);

      using (var sw = File.CreateText(dir1 + "demo.txt"))
         sw.WriteLine("Directory-Demo");

      // Über Dateien informieren (mit Filter)
      var di = new DirectoryInfo(".");
      FileInfo[] fia = di.GetFiles("*.txt");
      Console.WriteLine("txt-Dateien in {0}", di.FullName);
      Console.WriteLine("  {0, -20} {1, -20}", "Name", "Letzte Änderung");
      foreach (FileInfo fi in fia)
         Console.WriteLine("  {0, -20} {1, -20}", fi.Name, fi.LastWriteTime);

      // Über Unterordner informieren
      DirectoryInfo[] dia = di.GetDirectories();
      Console.WriteLine("\n\nOrdner in {0}", di.FullName);
      Console.WriteLine("  {0, -20} {1, -20}", "Name", "Letzte Änderung");
      foreach (DirectoryInfo die in dia)
         Console.WriteLine("  {0, -20} {1, -20}", die.Name, die.LastWriteTime);

      Directory.SetCurrentDirectory(dir1 + "\\..");
      Console.WriteLine("\nDer Ordner " + dir1 + " wird nach Enter gelöscht.");
      Console.ReadLine();

      // Ordner löschen
      Directory.Delete(dir1, true);
   }
}
```

Wichtige statische Methoden der Klasse **Directory** sind:

- **public static String GetCurrentDirectory()**
  public static void SetCurrentDirectory(String *pfadname***)**
  Mit **GetCurrentDirectory()** bzw. **SetCurrentDirectory()** kann man das aktuelle Verzeichnis zum laufenden Programm ermitteln bzw. setzen. Das im Aufruf von **SetCurrentDirectory()** angegebene Verzeichnis muss vorhanden sein.
- **public static bool Exists(String** *pfadname***)**
  Mit **Exists()** überprüft man die Existenz eines Ordners.

- **public static DirectoryInfo CreateDirectory(String** *pfadname***)**
  Mit der Methode **CreateDirectory()** lässt sich ein Ordner erstellen, sofern das angegebene Laufwerk vorhanden ist und die erforderlichen Zugriffsrechte bestehen. Bei Bedarf werden auch Zwischenordner im Pfad erstellt. Z. B. klappt der folgende Methodenaufruf auch dann, wenn auf dem Laufwerk **U:** noch *kein* Ordner namens **Eigene Dateien** vorhanden ist:
  ```
  Directory.CreateDirectory(@"U:\Eigene Dateien\C#\EA\Sub\");
  ```
- **public static void Delete(String** *pfadname***)**
  Mit dieser **Delete()** - Überladung lässt sich nur ein *leerer* Ordner löschen, sofern die erforderlichen Zugriffsrechte vorhanden sind. Bei einer alternativen Überladung mit Parameter vom Typ **bool** kann man auch ein rekursives Löschen von Unterverzeichnissen und Dateien erzwingen (siehe Beispielprogramm).

Im Konstruktor der Klasse **DirectoryInfo** ist ein Ordnerpfad anzugeben, wobei der aktuelle Pfad des Programms über einen Punkt angesprochen werden kann. Wichtige Instanzmethoden der Klasse **DirectoryInfo**:

- **public FileInfo[] GetFiles(String** *filter***)**
  Die **DirectoryInfo**-Instanzmethode **GetFiles()** liefert einen Array mit **FileInfo**-Objekten zu allen Dateien im Ordner mit einem zum Filter passenden Namen. Erlaubte Jokerzeichen im Dateiauswahlfilter sind:

Jokerzeichen	Bedeutung
?	ersetzt genau ein beliebiges Zeichen
*	steht für eine beliebige (eventuell leere) Zeichenfolge

- **public DirectoryInfo[] GetDirectories(String** *filter***)**
  Analog liefert die Instanzmethode **GetDirectories()** einen Array mit **DirectoryInfo**-Objekten zu den Unterordnern mit einem zum Filter passenden Namen.

Über die Eigenschaft **BaseDirectory** der sogenannten *Anwendungsdomäne* kann der Ordner mit dem ausgeführten Assembly ermittelt werden, um z. B. eine dort befindliche Datei anzusprechen:

```
AppDomain.CurrentDomain.BaseDirectory
```

Dieser Ordner ist nicht unbedingt identisch mit dem aktuellen Ordner der Anwendung, den man über die statische Methode **GetCurrentDirectory()** der Klasse **Directory** (siehe oben) oder über die statische Eigenschaft **CurrentDirectory** der Klasse **Environment** (im Namensraum **System**) ermittelt:

```
System.Environment.CurrentDirectory
```

16.6.3 Überwachung von Ordnern

Mit einem Objekt der Klasse **FileSystemWatcher** aus dem Namensraum **System.IO** lassen sich die Veränderungen in einem Ordner überwachen (Erzeugen, Löschen, Umbenennen von Einträgen). Das folgende Programm überwacht für die Textdateien (Namenserweiterung **.txt**) in dem per Befehlszeilenargument angegebenen Ordner die Änderungen beim Datum des letzten Zugriffs und beim Namen:

```
using System;
using System.IO;

class TxtWatcher {
   static void Main(string[] args) {
      using var watcher = new FileSystemWatcher(args[0]);

      // Zu Überwachen: Ändern und Umbenennen von Dateien
      watcher.NotifyFilter = NotifyFilters.LastWrite | NotifyFilters.FileName;
      // Filter für Dateinamen
      watcher.Filter = "*.txt";

      // Ereignisbehandlungsmethoden registrieren
      watcher.Changed += ChangedHandler;
      watcher.Created += ChangedHandler;
      watcher.Deleted += ChangedHandler;
      watcher.Renamed += RenamedHandler;

      // Überwachung aktivieren
      watcher.EnableRaisingEvents = true;

      Console.WriteLine("TxtWatcher gestartet. Beenden mit 'q'\n");
      Console.WriteLine("Überwachter Ordner: " + args[0] + "\n");
      ConsoleKeyInfo cki;
      do
         cki = Console.ReadKey(true);
      while (cki.KeyChar != 'q');
   }

   static void ChangedHandler(object source, FileSystemEventArgs e) {
      Console.WriteLine("Datei {0} {1}", e.Name, e.ChangeType);
   }

   static void RenamedHandler(object source, RenamedEventArgs e) {
      Console.WriteLine("Datei {0} umbenannt in {1}", e.OldName, e.Name);
   }
}
```

Im Übrigen demonstriert das Programm, dass Ereignisse nicht unbedingt von GUI-Komponenten stammen müssen, und dass auch Konsolenanwendungen auf Ereignisse reagieren können.

Eine Beispielausgabe:

```
TxtWatcher gestartet. Beenden mit 'q'

Überwachter Ordner: U:\Eigene Dateien\C#\EA

Datei Neues Textdokument.txt Created
Datei Neues Textdokument.txt Changed
Datei Neues Textdokument.txt umbenannt in test.txt
Datei test.txt Deleted
```

Mit der **Console**-Methode **ReadKey**() kann man sofort auf Tastendrücke reagieren und dabei die Ausgabe von Zeichen auf dem Bildschirm verhindern (Wert **true** für den ersten und einzigen Parameter). Man erhält eine Instanz der Struktur **ConsoleKeyInfo**, die u. a. das zu einer Taste gehörige Unicode-Zeichen kennt.

Während der Haupt-Thread des Konsolenprogramms auf Tastatureingaben lauert, werden die Dateisystemereignis-Behandlungsmethoden in einem separaten Thread ausgeführt.

Mit Hilfe einer **using**-Anweisung (siehe Abschnitt 16.2.3) wird dafür gesorgt, dass alle mit dem **FileSystemWatcher** verbundenen Ressourcen nach dem Beobachtungsende auf jeden Fall freigegeben werden. Im Beispiel wird dieses Muster demonstriert, um es in Erinnerung zu rufen. Weil die

using-Anweisung im Beispiel zum Block der **Main()** - Methode gehört, ist sie allerdings überflüssig, denn ...

- mit der Methode **Main()** endet auch der Haupt-Thread,
- und es sind keine weiteren Vordergrund-Threads vorhanden,
- sodass mit der **Main()** - Methode auch das Programm endet, und alle Ressourcen freigegeben werden.

16.7 Übungsaufgaben zum Kapitel 16

1) Erstellen Sie ein Statistikprogramm zur Berechnung des Mittelwerts, das als Eingabe eine Textdatei mit Daten akzeptiert, wobei das Semikolon als Trennzeichen dient. In folgender Beispieldatei liegen drei Variablen (Spalten) für fünf Fälle vor:

```
12;3;345
7;5;298
9;4;411
10;2;326
5;6;195
4;sieben;120
```

Die gültigen Werte der zweiten Spalte haben z. B. den Mittelwert 4,0. Ihr Programm sollte auf irreguläre Daten folgendermaßen reagieren:

- Warnung ausgeben
- mit den verfügbaren Werten rechnen

Für die obigen Daten sollte Ihr Programm ungefähr die folgende Ausgabe produzieren:

```
Mittelwertsberechnung für die Datei daten.txt

Warnung: Token 2 in Zeile 6 ist keine Zahl.
 "4;sieben;120"

            Variable           Mittelwert         Valide Werte
                   1                7,833                    6
                   2                4,000                    5
                   3              282,500                    6
```

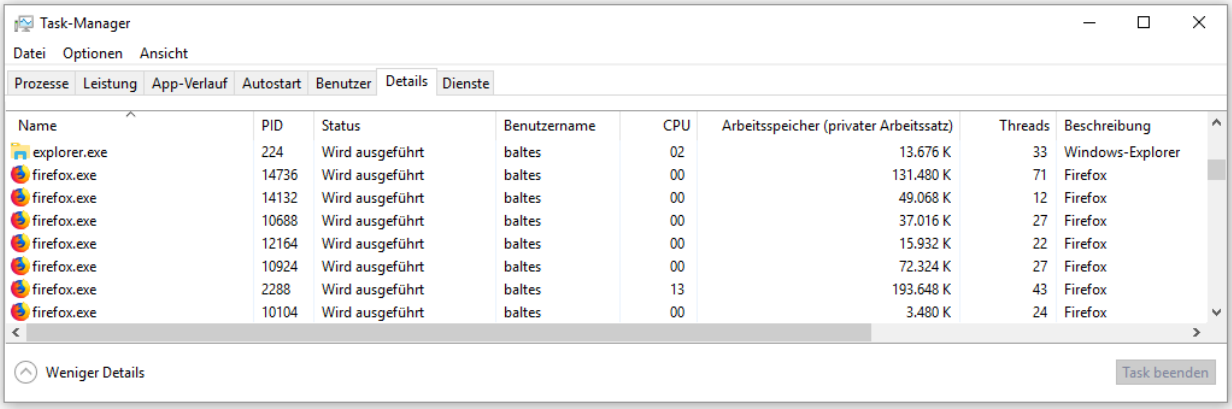
2) Wie kann man den Quellcode des folgenden Programms vereinfachen und dabei auch noch die Laufzeit erheblich reduzieren?

```
using System;
using System.IO;

class StreamWriterDemo {
  static void Main() {
    long zeit = DateTime.Now.Ticks;
    using (var sw = new StreamWriter("demo.txt")) {
      sw.AutoFlush = true;
      for (int i = 1; i < 30000; i++)
        sw.WriteLine(i);
    }
    Console.WriteLine("Zeit: " + ((DateTime.Now.Ticks - zeit) / 1.0e4) +
                      " Millisek.");
  }
}
```

17 Multithreading

Wir sind längst daran gewöhnt, dass moderne Betriebssysteme mehrere Programme (Prozesse) parallel betreiben können, sodass z. B. ein längerer Ausdruck keine Zwangspause des Benutzers zur Folge hat. Während der Druckertreiber die Ausgabeseiten aufbaut, kann z. B. ein C# - Programm entwickelt oder im Internet recherchiert werden. Weil in der Regel weniger Prozessoren bzw. virtuelle Prozessorkerne vorhanden sind als arbeitswillige Programme, muss das Betriebssystem die verfügbare CPU-Leistung nach einem Zeitscheibenverfahren auf die rechenwilligen Programme verteilen. Dadurch reduziert sich zwar die Ausführungsgeschwindigkeit jedes Programms, doch ist in den meisten Anwendungen trotzdem ein flüssiges Arbeiten möglich.

Als Ergänzung zum gerade beschriebenen **Multitasking**, das ohne Zutun der Anwendungsprogrammierer vom Betriebssystem bewerkstelligt wird, ist es oft sinnvoll oder gar unumgänglich, auch *innerhalb* einer Anwendung nebenläufige *Ausführungsfäden* zu realisieren, wobei man hier vom **Multithreading** spricht. Bei einem Internet-Browser muss man z. B. nach dem Anstoßen eines längeren Downloads nicht untätig den Fortschrittsbalken im Download-Fenster anstarren, sondern kann parallel mit anderen Fenstern arbeiten. Wie unter Windows die **Details**-Registerkarte im Task-Manager zeigt, sind z. B. bei einer typischen Verwendung des Internet-Browsers Firefox zahlreiche Threads aktiv, wobei die Anzahl ständig schwankt:[1]



Bei einer GUI-Anwendung sorgt die Multithreading-Technik dafür, dass die Bedienoberfläche auch dann noch flüssig auf Benutzereingaben reagiert, wenn im Hintergrund ein zeitaufwändiger Auftrag erledigt wird. Eine Server-Anwendung kann dank Multithreading mehrere Klienten simultan versorgen.

Die Multithreading-Technik kommt aber nicht nur dann in Frage, wenn eine Anwendung mehrere Aufgaben gleichzeitig erledigen soll, damit der Prozess nicht durch eine Aufgabe mit hohem Zeitaufwand blockiert wird. Sind auf einem Rechner mehrere Prozessoren oder Prozessorkerne verfügbar, dann sollten aufwändige Einzelaufgaben (z. B. das Rendern einer 3D-Ansicht, Virenanalyse einer kompletten Festplatte) in Teilaufgaben zerlegt und auf mehrere CPU-Kerne verteilt werden. Weil die CPU-Hersteller bei der Taktbeschleunigung an physische Grenzen gestoßen sind, konzentrieren sie sich seit vielen Jahren darauf, durch eine höhere Anzahl von CPU-Kernen eine Leistungssteigerung zu erzielen. Mittlerweile (2021) sind 4-6 *reale* Kerne zum Standard geworden, und viele CPUs der Hersteller AMD und Intel besitzen dank der SMT-Technik (*Simultaneous Multi-Threading*, bei Intel als *Hyper-Threading* bezeichnet) doppelt so viele *logische* CPU-Kerne. Multi-Core - CPUs erhöhen den Druck auf die Software-Entwickler, per Multithreading für gut skalierende Anwendungen zu sorgen, die ihre Leistung mit der Anzahl verfügbarer CPU-Kerne steigern.

---

[1] Wie das Bildschirmfoto zeigt, verwendet der Firefox-Browser zur Steigerung der Stabilität mehrere *Prozesse*; in jedem Prozess kommen wiederum zahlreiche Threads zum Einsatz.

Beim Multithreading ist allerdings eine sorgfältige Einsatzplanung erforderlich, weil auch Kosten und Risiken zu beachten sind:

- Das Erstellen, Terminieren, Blockieren und Reaktivieren von Threads kostet Zeit, sodass der Zeitaufwand durch die Verwendung von Multithreading sogar steigen kann.
- Threads belegen Speicher, sodass ihre Zahl nicht zu groß werden sollte.
- Die Multi-Thread - Programmierung ist erheblich anspruchsvoller als die Single-Thread - Programmierung. Das gilt vor allem dann, wenn mehrere Threads auf gemeinsame, variable Datenbestände zugreifen, sodass eine Synchronisation der Threads erforderlich ist (siehe Abschnitt 17.1.4). Hier kommt es oft zu Fehlern, wobei ein Programm ...
  - entweder hängt (Deadlock, siehe Abschnitt 17.1.7)
  - oder fehlerhafte Ergebnisse produziert, was noch weit gravierender ist.

  Eine fehlerhafte Thread-Synchronisation ist zudem aufgrund variabler Folgen schwer zu analysieren.

Während jeder *Prozess* einen eigenen Adressraum besitzt, laufen die in einem Prozess tätigen *Threads* im selben Adressraum ab, sodass sie gelegentlich auch als *leichtgewichtige Prozesse* bezeichnet werden. Sie verwenden einen gemeinsamen Heap-Speicher, wobei aber jeder Thread als selbständiger Kontrollfluss (Ausführungsfaden) einen eigenen Stack-Speicher benötigt.

Bei C# ist die Multithreading-Unterstützung in Sprache, Standardbibliothek und Laufzeitumgebung integriert. Folglich gehört diese Technik in C# nicht zum Guru-HighTech - Repertoire, sondern kann von *jedem* Programmierer ohne großen Aufwand genutzt werden.

Übrigens sind auch ohne unser Zutun in jeder .NET - Anwendung mehrere Threads aktiv. So läuft z. B. der Garbage Collector stets in einem eigenen Thread.

In der Praxis geht es darum, mit möglichst wenigen Threads eine gute Performanz zu erzielen und außerdem das aufwändige Kreieren und Terminieren von Threads so weit wie möglich zu vermeiden. Um diese Ziele zu unterstützen, verwaltet die CLR einen Threadpool. Hier befinden sich Threads in einer zur Hardware (zur Anzahl der logischen CPU-Kerne) passenden Zahl, die nach Erledigung einer Aufgabe nicht beendet werden, sondern auf neue Aufgaben warten. Statt für eine konkrete Aufgabe (z. B. Bedienung eines Webzugriffs) jeweils einen neuen Thread zeitaufwändig zu erzeugen und anschließend wieder zu zerstören, werden eingehende Aufgaben einem freien Pool-Thread zugeteilt oder in eine Warteschlange gestellt. Der Programmierer erstellt keine Threads, sondern *Aufgaben* und überlässt die Multithreading-Verwaltung der .NET - Laufzeitumgebung. Es wäre naheliegend, hier von *Multitasking* zu sprechen, wenn dieser Begriff nicht schon seit vielen Jahren für den Mehrprozessbetrieb auf der Ebene des Betriebssystems in Verwendung wäre.

Der Threadpool wird von diversen Klassen in der BCL verwendet, wobei sich im Laufe der .NET - Evolution unterschiedliche Programmiermuster etabliert haben. Das *Asynchronous Programming Model* (APM) war vom Anfang dabei und galt speziell bei der asynchronen Ein- und Ausgabe (Dateisystem, Netzwerk, Datenbankzugriff) lange als Standardlösung. Seit .NET 4.0 sollte zur performanten und skalierbaren Nutzung von Mehrkernsystemen die *Task Parallel Library* (TPL) bevorzugt werden. Seit C# 5 wird ihre Verwendung durch den Methodenmodifikator **async** und den Operator **await** erleichtert.

Wir erarbeiten uns in diesem Kapitel zunächst ein Multithreading-Basiswissen durch den Einsatz von dedizierten, für einen bestimmten Zweck vom Programmierer erstellten Threads, und bewegen uns dann langsam auf die TPL zu. Zu Recht stellt Microsoft in der Online-Dokumentation zur TPL fest:[1]

[1] https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl

> Although the TPL simplifies multithreaded scenarios, we recommend that you have a basic understanding of threading concepts, for example, locks, deadlocks, and race conditions, so that you can use the TPL effectively.

Wer über das aktuelle Kapitel hinaus weitere Informationen zum Multithreading in C# benötigt, findet diese z. B. in Agafonov & Koryavchenko (2015), Albahari & Johannsen (2020), Cleary (2014) und Griffiths (2013, 2020).

17.1 Threads

In C# wird ein Thread durch ein Objekt der gleichnamigen Klasse im Namensraum **System.Threading** realisiert.[1] Das direkte Erzeugen von Threads über Objekte aus der Klasse **Thread** wird zunehmend abgelöst durch die Nutzung von Bibliotheksklassen, die im Hintergrund mit Multithreading-Techniken arbeiten. Allerdings erleichtert es der traditionelle Direktkontakt mit Threads, grundlegende Eigenschaften der Technik kennenzulernen.

Der Abschnitt 17.1 startet elementar, behandelt im weiteren Verlauf aber auch komplexe Themen der direkten Multithreading-Programmierung. Die Erfahrung dieser Komplexität wird vermutlich viele Leser zu der meist empfehlenswerten Verwendung von Multithreading-Bibliotheken wie der TPL (*Task Parallel Library*) motivieren.

17.1.1 Threads erzeugen

Jede Instanz- oder Klassenmethode, die entweder den Delegatentyp

public delegate void ThreadStart()

oder den Delegatentyp

public delegate void ParameterizedThreadStart(Object *obj***)**

erfüllt, kann in einem eigenen Thread gestartet werden, indem ein zugehöriges Delegatenobjekt erzeugt und einem **Thread**-Konstruktor übergeben wird, z. B.:

```
Thread pt = new Thread(new ThreadStart(pro.Run));
```

Die explizite Delegaten-Konstruktion ist nicht erforderlich:

```
Thread pt = new Thread(pro.Run);
```

17.1.1.1 Produzent-Konsument - Beispiel

Die eben als Beispiel betrachtete Anweisung zur Thread-Kreation stammt aus einem „betriebswirtschaftlichen“ Programm mit einem Objekt aus einer Klasse namens `Produzent` und einem Objekt aus einer Klasse namens `Konsument`, die auf einen Lagerbestand einwirken, der von einem Objekt aus einer Klasse namens `Lager` gehütet wird. Produzent und Konsument entfalten ihre Tätigkeit jeweils im Rahmen einer Methode namens `Run()`, die in einem eigenen Thread ausgeführt wird.

17.1.1.2 Die Klasse Lager

Ein Objekt der Klasse `Lager` beherrscht die folgenden Methoden, um den Produzenten und den Konsumenten zu bedienen, solange nicht eine Maximalzahl von Lagerzugriffen überschritten ist:

---

[1] Ein Thread im Sinne der CLR (ein *verwalteter Thread*) muss nicht unbedingt 1:1 auf einen Thread im Sinne des zugrunde liegenden Betriebssystems abgebildet werden, siehe:
https://docs.microsoft.com/en-us/dotnet/standard/threading/managed-and-unmanaged-threading-in-windows

- `public bool Ergaenze(int add)`
  Diese Methode wird vom Produzenten genutzt, um Ware virtuell einzuliefern. Ist das Lager bereits geschlossen, wird **false** zurückgemeldet, sonst **true**.
- `public bool Liefere(int sub)`
  Diese Methode wird vom Konsumenten genutzt, um Ware virtuell zu beziehen. Ist das Lager bereits geschlossen, wird **false** zurückgemeldet, sonst **true**.
- `void Rumoren()`
  Diese private Methode dient dazu, Aufwand beim Ausführen der Aufträge zu simulieren.

In den Methoden `Ergaenze()` und `Liefere()` wird zur formatierten Zeitausgabe eine spezielle Überladung der **DateTime**-Methode **ToString()**

```
DateTime.Now.ToString("T", ci)
```

verwendet, wobei ein Objekt der Klasse **CultureInfo** (Namensraum **System.Globalization**) beteiligt ist:

```
System.Globalization.CultureInfo ci = new System.Globalization.CultureInfo("de-DE");
```

Ein `Lager`-Objekt verwaltet in den privaten Feldern `bilanz` und `anz` den aktuellen Lagerbestand (initialisiert mit der Konstanten `startKap`) und die Anzahl der bisherigen Lagerzugriffe (nach oben begrenzt durch die Konstante `maxAnz`):

```
using System;
using System.Threading;

public class Lager {
  int bilanz;
  int anz;
  const int maxAnz = 20;
  const int startKap = 100;
  System.Globalization.CultureInfo ci = new System.Globalization.CultureInfo("de-DE");

  public Lager(int start) {
    bilanz = start;
  }

  public bool Ergaenze(int add) {
    if (anz < maxAnz) {
      bilanz += add;
      anz++;
      Rumoren();
      Console.WriteLine("Nr. {0,2}: {1,10} ergänzt  {2,3} um {3} Uhr. Stand: {4}",
        anz, Thread.CurrentThread.Name, add, DateTime.Now.ToString("T", ci), bilanz);
      return true;
    } else {
      Console.WriteLine("\nLieber " + Thread.CurrentThread.Name +
        ", es ist Feierabend!");
      return false;
    }
  }
```

```
  public bool Liefere(int sub) {
    if (anz < maxAnz) {
      bilanz -= sub;
      anz++;
      Rumoren();
      Console.WriteLine("Nr. {0,2}: {1,10} entnimmt {2, 3} um {3} Uhr. Stand: {4}",
        anz, Thread.CurrentThread.Name, sub, DateTime.Now.ToString("T", ci), bilanz);
      return true;
    } else {
      Console.WriteLine("\nLieber " + Thread.CurrentThread.Name +
        ", es ist Feierabend!");
      return false;
    }
  }

  void Rumoren() {
    double d;
    for (int i = 0; i < 40_000; i++)
      d = i * i;
  }

  static void Main() {
    Lager lager = new Lager(startKap);
    Console.WriteLine("Das Lager ist offen (Bestand: {0})\n", startKap);
    Produzent pro = new Produzent(lager);
    Konsument kon = new Konsument(lager);
    Thread pt = new Thread(pro.Run);
    Thread kt = new Thread(kon.Run);
    pt.Name = "Produzent";
    kt.Name = "Konsument";
    pt.Start();
    kt.Start();
  }
}
```

17.1.1.3 Die Klassen `Produzent` und `Konsument`

Produzent und Konsument kommen mit einer simplen Klassendefinition aus:

```
using System;
using System.Threading;

public class Produzent {
   Lager pl;
   Random rand = new Random(1);

   public Produzent(Lager ptr) {
      pl = ptr;
   }

   public void Run() {
      while (pl.Ergaenze(5 + rand.Next(100)))
         Thread.Sleep(1000 + rand.Next(3000));
   }
}
```

```
public class Konsument {
   Lager pl;
   Random rand = new Random(2);

   public Konsument(Lager ptr) {
      pl = ptr;
   }

   public void Run() {
      while (pl.Liefere(5 + rand.Next(100)))
         Thread.Sleep(1000 + rand.Next(3000));
   }
}
```

Weil Produzent und Konsument mit dem `Lager`-Objekt kooperieren sollen, erhalten sie per Konstruktor-Parameter eine entsprechende Referenz.

Neben dem Konstruktor ist jeweils eine Methode namens `Run()` vorhanden, welche den Delegatentyp **ThreadStart** erfüllt und sich damit als Startmethode für einen Thread eignet. Die `Run()` - Methoden beschränken sich auf eine **while**-Schleife, wobei in jedem Durchgang ein Auftrag zum Ein- bzw. Auslagern einer zufallsbestimmten Menge an das `Lager`-Objekt geschickt wird. Der `Ergaenze()` - bzw. `Liefere()` - Methodenaufruf an das Lagerobjekt bildet den Ausdruck der **while**-Fortsetzungsbedingung, was aufgrund der Rückgabe vom Typ **bool** zulässig ist. Die **while**-Anweisung enthält einen Aufruf der statischen **Thread**-Methode **Sleep()**, die den aktuellen Thread vom Zustand **Running** in den Zustand **WaitSleepJoin** befördert (siehe Abschnitt 17.1.6.2). So entsteht eine Pause von zufallsabhängiger Dauer zwischen zwei Aufträgen an das Lager.

Mit festen Startwerten für die Pseudozufallszahlengeneratoren aus der Klasse **Random** soll dafür gesorgt werden, dass stets derselbe Lagerverlauf resultiert. Dies gelingt aber nur teilweise, weil der etwas früher gestartete Produzenten-Thread aufgrund von CLR-internen Vorgängen nicht unbedingt als erster beim Lager ankommt.

17.1.2 Threads starten

Die **Main()** - Methode der Klasse `Lager` erzeugt als Startmethode des Programms die beteiligten Objekte:

- einen Lageristen (Objekt `lager` aus der Klasse `Lager`)
  ```
  Lager lager = new Lager(startKap);
  ```
- einen Produzenten (Objekt `pro` aus der Klasse `Produzent`)
  ```
  Produzent pro = new Produzent(lager);
  ```
- einen Konsumenten (Objekt `kon` aus der Klasse `Konsument`)
  ```
  Konsument kon = new Konsument(lager);
  ```
- einen Thread, dessen Ausführung mit der `Run()` - Methode des Produzenten startet (Objekt `pt` aus der Klasse **Thread**)
  ```
  Thread pt = new Thread(pro.Run);
  ```
- einen Thread, dessen Ausführung mit der `Run()` - Methode des Konsumenten startet (Objekt `kt` aus der Klasse **Thread**)
  ```
  Thread kt = new Thread(kon.Run);
  ```

Schließlich erhalten die Threads einen Namen und werden gestartet:

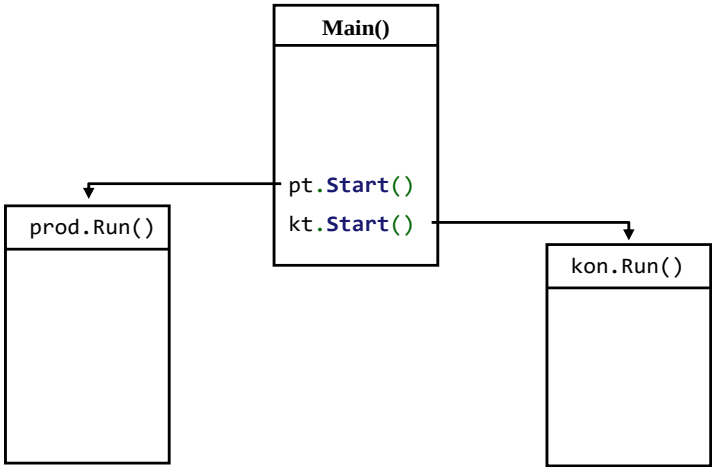
```
pt.Name = "Produzent";
kt.Name = "Konsument";
pt.Start();
kt.Start();
```

Der **Thread** `pt` startet mit der Ausführung der `Produzent`-Methode `Run()` durch das Objekt `pro` aus der Klasse `Produzent`, und der **Thread** `kt` startet mit der Ausführung der `Konsument`-Methode `Run()` durch das Objekt `kon` aus der Klasse `Konsument`.

Im Beispiel laufen der Produzenten- und der Konsumenten-Thread als sogenannte **Vordergrund-Threads**. Ein .NET - Programm bleibt aktiv, solange mindestens ein Vordergrund-Thread existiert. Ist ein Thread über seine Eigenschaft **IsBackground** als **Hintergrund-Thread** markiert, dann kann er das Programm nicht am Leben erhalten, sondern wird ggf. abrupt gestoppt, wenn der letzte Vordergrund-Thread endet. Wenn für einen Background-Thread das Abwürgen ohne die Gelegenheit für Aufräumungsarbeiten ausscheidet, dann muss dieser Thread durch einen Vordergrund-Thread geordnet beendet werden (siehe Abschnitt 17.1.5.2).

Unmittelbar vor dem Ende der **Main()** - Methode sind im Beispielprogramm **drei** Vordergrund-Threads aktiv:[1]

- Der **primäre** Thread des Programms lebt, solange die **Main()** - Methode läuft.
- Außerdem agieren zu diesem Zeitpunkt die beiden zusätzlich gestarteten **sekundären** Threads. Sie enden mit ihrer Startmethode (`pro.Run()` bzw. `kon.Run()`), sofern sie nicht zuvor abgebrochen werden (siehe unten).



Die Aufrufe der **Thread**-Methode **Start()** kehren praktisch unmittelbar zurück, und im Beispiel endet anschließend mit der **Main()** - Methode auch der primäre Thread. Die beiden sekundären Threads leben weiter, und das Programm endet mit seinem letzten Vordergrund-Thread.

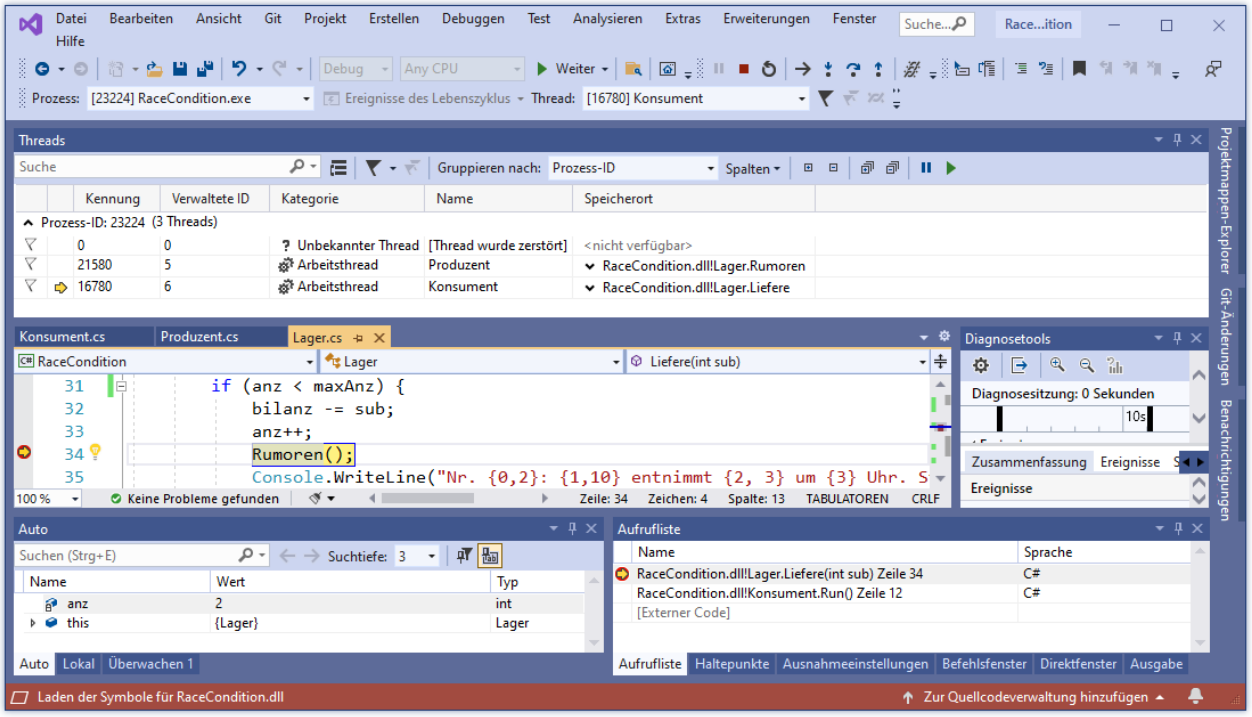
Um die in einem Programm bei der Ausführung einer bestimmten Anweisung aktiven Threads vom Visual Studio anzeigen zu lassen, setzt man einen Haltepunkt vor die Anweisung (vgl. Abschnitt 5.3.4), startet das Programm (über den Schalter ▶ oder die Funktionstaste **F5**) im Debug-Modus und öffnet das Fenster mit den Threads über den Menübefehl

Debuggen > Fenster > Threads

Im Beispiel sind die erwarteten drei Threads zu sehen, wobei der primäre Thread bereits beendet ist:

---

[1] Zu einem Programm gehören noch weitere Threads, die im Hintergrund von der CLR verwaltet werden (z. B. für den Garbage Collector). Endet der letzte Vordergrund-Thread eines Programms, werden seine Hintergrund-Threads automatisch gestoppt.



Es werden die folgenden **Kategorien** von Threads unterschieden:

- Hauptthread der Anwendung
- Bedienoberflächen-Threads (UI-Threads)
  Sie sind für die Erstellung und Verwaltung der Bedienelemente eines Fensters zuständig.
- Arbeits-Threads
  Sie sind *nicht* mit einem Fenster verbunden, sondern mit Hintergrundaufgaben beschäftigt. Im Sinne der obigen Definition kann es sich um Vorder- oder Hintergrund-Threads handeln.
- Remoteprozeduraufruf-Handler
  Remoteprozeduraufrufe dienen der Interprozesskommunikation, wobei die beteiligten Prozesse auf demselben Rechner oder auf verschiedenen Rechnern laufen können.

Ein Thread endet, wenn seine Startmethode abgearbeitet ist. Er befindet sich dann im Zustand **Stopped** und kann *nicht* erneut gestartet werden. Im Beispiel passiert das dem Produzenten- oder den Konsumenten-Thread, sobald die `Run()` - Methode zu Beginn eines **while** - Schleifendurchgangs ein geschlossenes Lager feststellt, wenn also der Methodenaufruf `lager.Ergaenze()` bzw. `lager.Liefere()` zum Rückgabewert **false** führt.

Der primäre Thread des Programms ist zu diesem Zeitpunkt bereits Geschichte, weil er mit der `Lager`-Methode **Main()** seine Tätigkeit einstellt. Folglich endet das Programm, wenn der Produzenten- und der Konsumenten-Thread sich verabschiedet haben.

Um den neben **ThreadStart** ebenfalls als **Thread**-Konstruktorparameter geeigneten Delegatentyp **ParameterizedThreadStart** vorzuführen, unterbrechen wir kurz das Produzent-Konsument - Beispiel. Im folgenden Programm wird eine statische Methode namens `TuWas()` definiert, welche den Delegatentyp **ParameterizedThreadStart** erfüllt, also den Rückgabetyp **void** besitzt sowie einen Parameter vom Typ **Object**, der eine Beeinflussung des Verhaltens erlaubt:

```
using System;
using System.Threading;

class PTS {
    static void TuWas(Object obj) {
        char c = (obj as String)[0];
        for (int i = 1; i < 100; i++) {
            Console.Write(c + " ");
            Thread.Sleep(10);
        }
    }
    static void Main() {
        Thread t = new Thread(new ParameterizedThreadStart(TuWas));
        t.Start("2");
        Thread.Sleep(500);
        Console.WriteLine("\nEnde Main");
    }
}
```

Unter Verwendung der Methode `TuWas()` wird ein **Thread**-Objekt erzeugt:

```
Thread t = new Thread(new ParameterizedThreadStart(TuWas));
```

Den `TuWas()` - Aktualparameter zur Beeinflussung des Thread-Verhaltens übergibt man beim Starten des Threads an eine Überladung der **Thread**-Methode **Start()**:

```
t.Start("2");
```

Der primäre Thread ruht eine halbe Sekunde und platziert seine Terminierungsbotschaft dann zwischen die vom sekundären Thread fleißig produzierten Zweier:

```
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
Ende Main
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

17.1.3 Klassen aus dem Anwendungsbereich und aus der Informatik

Das Produzent-Konsument - Beispiel demonstriert mit seinem Objekt-Ensemble, dass einige Objekte direkt aus der Abbildung des Anwendungsbereichs stammen (Lagerist, Produzent, Konsument), während die Objekte der Klasse **Thread** als Konstrukte der Informatik zur Parallelisierung von Aktivitäten dienen. Man kann einen Thread als *Ausführungsfaden* oder *Aktivitätsträger*[1] bezeichnen. Wir haben früher gelegentlich von *der objektorientierten Bühne* gesprochen und können nun zur Veranschaulichung von Multithreading zum Plural übergehen. Ein sekundärer Thread (Aktivitätsträger) wird als zusätzliche Bühne mit eigenem Handlungsablauf interpretiert.

Im Ausführungsfaden `pt` des Beispielprogramms wird die Startmethode `Run()` von einem Objekt aus der Klasse `Produzent` ausgeführt. Alle von einer Startmethode via Methodenaufruf direkt oder indirekt initiierten Aufträge an andere Objekte oder Klassen laufen ebenfalls im selben Thread (auf derselben Bühne) ab, sodass in einem Ausführungsfaden beliebig viele Akteure tätig werden können. Wir reden zwar vom *Produzenten*-Thread, weil dieser Aktivitätsträger mit der Ausführung der Methode `Run()` durch ein Objekt der Klasse `Produzent` startet, doch wird in diesem Thread auch das `Lager`-Objekt aktiv (über Aufrufe seiner `Ergaenze()` - Methode).

Andererseits kann ein Objekt in mehreren Threads tätig werden, wenn es entsprechende Botschaften erhält. Im Beispiel kommt das `Lager`-Objekt sowohl im Produzenten- als auch im Konsumenten-Thread zum Einsatz: Seine Methoden `Ergaenze()` und `Liefere()` erhöhen oder reduzieren den Lagerbestand, aktualisieren die Anzahl der Lagerveränderungen und protokollieren jede Maß-

[1] http://de.wikipedia.org/wiki/Thread_(Informatik)

nahme. Bei einem Protokolleintrag verwenden sie die statische **Thread**- Eigenschaft **CurrentThread()**, die eine Referenz auf den agierenden Thread enthält, und ermitteln dessen **Name**-Eigenschaft.

Wenn die Vorstellung eines Lageristen stört, der simultan in zwei Threads (auf zwei Bühnen) tätig ist, dann stelle man sich ein *Team* von Lagerarbeitern vor, was der Realität vieler Betriebe recht gut entspricht.

17.1.4 Threads koordinieren

In einem typischen Ablaufprotokoll des Produzent-Konsument - Programms zeigen sich einige Ungereimtheiten, verursacht durch das unkoordinierte Agieren des Produzenten- und des Konsumenten-Threads:

```
Das Lager ist offen (Bestand: 100)

Nr.  2:  Produzent ergänzt   29 um 02:01:58 Uhr. Stand: 47
Nr.  2:  Konsument entnimmt  82 um 02:01:58 Uhr. Stand: 47
Nr.  3:  Produzent ergänzt   51 um 02:02:00 Uhr. Stand: 98
Nr.  4:  Konsument entnimmt  21 um 02:02:01 Uhr. Stand: 77
Nr.  5:  Produzent ergänzt   70 um 02:02:03 Uhr. Stand: 147
Nr.  6:  Konsument entnimmt  15 um 02:02:04 Uhr. Stand: 132
Nr.  7:  Produzent ergänzt   40 um 02:02:05 Uhr. Stand: 172
Nr.  8:  Konsument entnimmt  85 um 02:02:06 Uhr. Stand: 87
Nr.  9:  Konsument entnimmt  27 um 02:02:09 Uhr. Stand: 60
Nr. 10:  Produzent ergänzt   15 um 02:02:09 Uhr. Stand: 75
Nr. 11:  Konsument entnimmt  81 um 02:02:10 Uhr. Stand: -6
Nr. 12:  Konsument entnimmt   5 um 02:02:11 Uhr. Stand: -11
Nr. 13:  Produzent ergänzt    7 um 02:02:12 Uhr. Stand: -4
Nr. 14:  Konsument entnimmt  43 um 02:02:13 Uhr. Stand: -47
Nr. 15:  Produzent ergänzt   37 um 02:02:14 Uhr. Stand: -10
Nr. 16:  Konsument entnimmt  75 um 02:02:15 Uhr. Stand: -85
Nr. 17:  Konsument entnimmt  78 um 02:02:17 Uhr. Stand: -163
Nr. 18:  Produzent ergänzt   73 um 02:02:18 Uhr. Stand: -90
Nr. 19:  Konsument entnimmt  13 um 02:02:18 Uhr. Stand: -103
Nr. 20:  Konsument entnimmt  37 um 02:02:20 Uhr. Stand: -140

Lieber Produzent, es ist Feierabend!

Lieber Konsument, es ist Feierabend!
```

U. a. fällt negativ auf:

- Im ersten Protokolleintrag wird berichtet, dass vom Startwert 100 ausgehend eine Lieferung von 29 Einheiten zu einem Bestand von 47 Einheiten geführt habe, und auch die Auftragsnummer 2 ist falsch.
- Im zweiten Eintrag wird behauptet, dass die Entnahme von 82 Einheiten ohne Effekt auf den Lagerbestand geblieben sei.
- Der Bestand wird negativ, was in einem realen Lager nicht passieren kann.

Wenn es sich nicht vermeiden lässt, dass mehrere Threads gemeinsame Daten verwenden und dabei auch *schreibend* zugreifen, sind Maßnahmen zur Koordination der Zugriffe erforderlich.

17.1.4.1 Zugriffsexklusivität

Während ein Thread auf gemeinsam genutzte Daten zugreift und diese potentiell verändert, müssen andere Threads am simultanen (lesenden oder schreibenden) Zugriff gehindert werden, um inkonsistente Zustände bzw. fehlerhafte Interpretationen zu vermeiden.

17.1.4.1.1 Die lock-Anweisung

Am Anfang des oben wiedergegebenen Ablaufprotokolls stehen zwei „wirre“ Einträge, die folgendermaßen durch eine sogenannte *Race Condition* zu erklären sind:

- Der etwas früher gestartete Produzenten-Thread kommt als erster beim Lager an, ruft die `Lager`-Methode `Ergaenze()` mit dem Parameterwert 29 auf und bringt mit den Anweisungen
  ```
  bilanz += add;
  anz++;
  ```
  die `bilanz` auf den Wert 129 sowie die Auftragsnummer auf den Wert 1.
- Dann unterbricht die CLR den Produzenten-Thread und aktiviert den Konsumenten-Thread.
- Dieser ruft die `Lager`-Methode `Liefere()` mit dem Parameterwert 82 auf und bringt mit den Anweisungen
  ```
  bilanz -= sub;
  anz++;
  ```
  die Lagerbilanz auf 47 sowie die Auftragsnummer auf den Wert 2.
- Nun kommt der Produzenten-Thread wieder zum Zug und schreibt seinen Protokolleintrag, wobei er die mittlerweile vom Konsumenten-Thread veränderten Werte von `anz` und `bilanz` verwendet.
- Dann schreibt auch der Konsumenten-Thread seine Protokollzeile. Allerdings ist der Stand von 47 nur dann nachvollziehbar, wenn man die vorherige, nicht korrekt protokollierte Lieferung berücksichtigt.

Es kann nicht nur zu wirren Protokolleinträgen kommen, sondern auch zu einem fehlerhaften `bilanz`-Wert. Scheinbar einschrittige Operationen wie die folgende Anweisung in der vom Produzenten-Thread aufgerufenen Methode `Ergaenze()`

```
bilanz += add;
```

haben in einem Rechner mehrere Teilschritte zur Folge, sind also nicht **atomar**, z. B.:

- aktuellen `bilanz`-Wert aus dem Hauptspeicher in ein CPU-Register einlesen
- Wert (der lokalen Kopie!) erhöhen
- Neuen Wert in den Hauptspeicher schreiben

In der vom Konsumenten-Thread aufgerufenen Methode `Liefere()` führt die Anweisung

```
bilanz -= sub;
```

analog zu folgenden Teilschritten:

- aktuellen `bilanz`-Wert aus dem Hauptspeicher in ein CPU-Register einlesen
- Wert (der lokalen Kopie!) reduzieren
- Neuen Wert in den Hauptspeicher schreiben

Durch unglückliche Thread-Wechsel kann es z. B. zu folgender Sequenz kommen:

- Der Produzent liest den Wert 100.
- Der Konsument liest den Wert 100.
- Der Produzent erhöht seine `bilanz`-Kopie um 10 auf 110 und schreibt das Ergebnis in den Hauptspeicher.
- Der Konsument reduziert seine `bilanz`-Kopie um 10 auf 90 und schreibt das Ergebnis in den Hauptspeicher. Damit ist der Beitrag des Produzenten verloren gegangen.

Auf einem Rechner mit einem 32-Bit - Betriebssystem kann es sogar passieren, dass ein Thread beim Schreiben eines **long**- oder **double**-Werts (64 Bit groß) unterbrochen wird, und dass schlussendlich die 64 Bits einer Variablen von zwei verschiedenen Threads stammen. Auf einem Rechner

mit 64-Bit-Betriebssystem werden **long**- und **double**-Werte atomar gelesen und geschrieben, nicht jedoch die 128 Bits großen Werte vom Typ **decimal** (vgl. Abschnitt 4.3.4 zum Speicherbedarf der elementaren Datentypen).

Im Beispielprogramm muss verhindert werden, dass während eines Lagerzugriffs ein Thread-Wechsel stattfindet. Die klassische Lösung besteht in C# darin, alle kritischen Quellcodesegmente per **lock**-Anweisung mit einer gemeinsamen Sperre zu versehen, z. B.:

```
Object lockObject = new Object();
  . . .
public bool Ergaenze(int add) {
 lock(lockObject) {
  if (anz < maxAnz) {
   bilanz += add;
   anz++;
   Rumoren();
   Console.WriteLine("Nr. {0,2}: {1,10} ergänzt  {2,3} um {3} Uhr. Stand: {4}",
    anz, Thread.CurrentThread.Name, add, DateTime.Now.ToString("T", ci), bilanz);
   return true;
  } else {
   Console.WriteLine("\nLieber " + Thread.CurrentThread.Name +
    ", es ist Feierabend!");
   return false;
  }
 }
}
```

Zum Sperren kann jedes beliebige Objekt verwendet werden. Richter (2012) rät aber zur Verwendung eines *privaten* Member-Objekts, weil ein öffentlich bekanntes Sperrobjekt durch eine unkoordinierte Verwendung zum Blockieren der Anwendung führen kann. Im Beispiel wird diese Empfehlung realisiert.

Ferner sollte zum Sperren *kein* Objekt einer *fremden* Klasse verwendet werden. Das ist auch dann riskant, wenn alle Referenzen auf das Objekt unter Kontrolle sind, weil die folgende, von Griffiths (2020) beschriebene Konstellation möglich ist:

- In der eigenen Klasse wird eine Methode des Sperrobjekts aufgerufen.
- Im unbekannten Quellcode dieser Methode wird das Objekt per **this**-Referenz für interne Sperrzwecke verwendet.

Im Produzent-Konsument - Beispiel muss auch der kritische Bereich in der Methode `Liefere()` durch *dasselbe* Objekt gesperrt werden:

```
public bool Liefere(int sub) {
 lock (lockObject) {
  if (anz < maxAnz) {
   bilanz -= sub;
   anz++;
   Rumoren();
   Console.WriteLine("Nr. {0,2}: {1,10} entnimmt {2, 3} um {3} Uhr. Stand: {4}",
    anz, Thread.CurrentThread.Name, sub, DateTime.Now.ToString("T", ci), bilanz);
   return true;
  } else {
   Console.WriteLine("\nLieber " + Thread.CurrentThread.Name +
    ", es ist Feierabend!");
   return false;
  }
 }
}
```

Ein Anweisungsblock mit **lock**-reguliertem Zugang wird auch als *synchronisiert* bezeichnet.

Beim Betreten eines noch freien geschützten Bereichs setzt ein Thread per **lock**-Anweisung die Sperre. Man kann sich vorstellen, dass er den einzigen Schlüssel für den geschützten, aus mehreren Quellcodesegmenten bestehenden Bereich erwirbt. Jedem anderen Thread wird der Zutritt verwehrt. Er wird in den Zustand **WaitSleepJoin** versetzt und muss warten. Beim Verlassen des geschützten Bereichs wird die Sperre aufgehoben, und ggf. kann ein wartender Thread seine Arbeit fortsetzen. Um andere Threads möglichst wenig zu behindern, muss ein Sperrobjekt so schnell wie möglich wieder freigegeben werden.

Aufgrund der Thread-Synchronisation per Sperrobjekt produziert unser Beispielprogramm keine wirren Protokolleinträge mehr, doch kann es nach wie vor zu einem negativen Lagerzustand kommen, z. B.:

```
Das Lager ist offen (Bestand: 100)

Nr.  1:  Produzent ergänzt   29 um 02:29:23 Uhr. Stand: 129
Nr.  2:  Konsument entnimmt  82 um 02:29:23 Uhr. Stand: 47
Nr.  3:  Produzent ergänzt   51 um 02:29:24 Uhr. Stand: 98
Nr.  4:  Konsument entnimmt  21 um 02:29:25 Uhr. Stand: 77
Nr.  5:  Produzent ergänzt   70 um 02:29:27 Uhr. Stand: 147
Nr.  6:  Konsument entnimmt  15 um 02:29:29 Uhr. Stand: 132
Nr.  7:  Produzent ergänzt   40 um 02:29:30 Uhr. Stand: 172
Nr.  8:  Konsument entnimmt  85 um 02:29:31 Uhr. Stand: 87
Nr.  9:  Konsument entnimmt  27 um 02:29:33 Uhr. Stand: 60
Nr. 10:  Produzent ergänzt   15 um 02:29:33 Uhr. Stand: 75
Nr. 11:  Konsument entnimmt  81 um 02:29:34 Uhr. Stand: -6
Nr. 12:  Konsument entnimmt   5 um 02:29:35 Uhr. Stand: -11
Nr. 13:  Produzent ergänzt    7 um 02:29:36 Uhr. Stand: -4
Nr. 14:  Konsument entnimmt  43 um 02:29:38 Uhr. Stand: -47
Nr. 15:  Produzent ergänzt   37 um 02:29:38 Uhr. Stand: -10
Nr. 16:  Konsument entnimmt  75 um 02:29:40 Uhr. Stand: -85
Nr. 17:  Konsument entnimmt  78 um 02:29:41 Uhr. Stand: -163
Nr. 18:  Produzent ergänzt   73 um 02:29:42 Uhr. Stand: -90
Nr. 19:  Konsument entnimmt  13 um 02:29:43 Uhr. Stand: -103
Nr. 20:  Konsument entnimmt  37 um 02:29:45 Uhr. Stand: -140

Lieber Produzent, es ist Feierabend!

Lieber Konsument, es ist Feierabend!
```

Es ist zu beachten, dass durch **lock**-Anweisungen ein geschützter, aus mehreren Segmenten bestehender *Code*-Bereich entsteht, nicht aber ein geschützter *Speicher*bereich. Damit ein geschützter Speicherbereich resultiert, müssen *alle* Codesegmente mit Zugriff auf diesen Speicherbereich in die Zone mit exklusivem Zugriff einbezogen werden (Morrison 2005a).

Wenn eine komplette Methode in den geschützten Bereich einbezogen werden soll, der bei Instanzmethoden vom handelnden Objekt (anzusprechen mit **this**) und bei statischen Methoden vom **Type**-Objekt zur Klasse (anzusprechen mit **typeof(***Klasse***)**) überwacht wird, dann stellt man der Methode ein **MethodImplAttribute** mit passendem Parameterwert voran, z. B.:[1]

[1] Diese Sperrtechnik entspricht dem Methoden-Modifikator **synchronized** in der Programmiersprache Java.

```
[MethodImplAttribute(MethodImplOptions.Synchronized)]
public bool Ergaenze(int add) {
 if (anz < maxAnz) {
  bilanz += add;
  anz++;
  Rumoren();
  Console.WriteLine("Nr. {0,2}: {1,10} ergänzt  {2,3} um {3} Uhr. Stand: {4}",
     anz, Thread.CurrentThread.Name, add, DateTime.Now.ToString("T", ci), bilanz);
  return true;
 } else {
  Console.WriteLine("\nLieber " + Thread.CurrentThread.Name +
   ", es ist Feierabend!");
  return false;
 }
}
```

Man überlässt dem Compiler das Erstellen der **lock**-Anweisung und muss zumindest bei statischen Methoden ein *öffentliches* Sperrobjekt in Kauf nehmen, das Blockaden durch ungeschickte oder böswillige Software ermöglicht.

17.1.4.1.2 Die Klasse Monitor

In diesem Abschnitt wird sich herausstellen:

- Für die Umsetzung der **lock**-Anweisung ist die Klasse **Monitor** zuständig.
- Die statischen **Monitor**-Methoden **Wait()** und **Pulse()** ermöglichen den beteiligten Threads eine intelligente Kooperation.

17.1.4.1.2.1 Realisation der lock-Anweisung

Für die Thread-Synchronisation per **lock**-Anweisung sorgt letztlich die Klasse **Monitor** aus dem Namensraum **System.Threading**. Sie eignet sich als statische Klasse weder zum Erzeugen von Objekten noch zum Ableiten neuer Klassen, spielt aber eine zentrale Rolle bei der Thread-Koordination. Die **lock**-Anweisung im folgenden Codesegment

```
var lockObj = new Object();
lock(lockObj) {
    . . .                                          // Synchronisierte Anweisungen
}
```

wird vom C# - Compiler der aktuellen Roslyn-Generation umgesetzt in:[1]

```
var lockObj = new Object();
bool lockTaken = false;
try {
   Monitor.Enter(lockObj, ref lockTaken);
    . . .                                            // Synchronisierte Anweisungen
} finally {
   if(lockTaken)
      Monitor.Exit(lockObj);
}
```

Hier kommen zwei statische **Monitor**-Methoden zum Einsatz:

---

[1] Dies ist einem Quellcode-Kommentar auf der folgenden Github-Webseite zu entnehmen: https://github.com/dotnet/roslyn/blob/56f605c41915317ccdb925f66974ee52282609e7/src/Compilers/CSharp/Portable/Lowering/LocalRewriter/LocalRewriter_LockStatement.cs

- **public static void Enter(Object** *obj***)**
  Das Sperrobjekt wird erworben, wobei der Verweisparameter `lockTaken` den Erfolg dokumentiert.
- **public static void Exit(Object** *obj***)**
  Von dieser Methode wird das Sperrobjekt zurückgegeben. Weil in der Methode **Enter()** eine Ausnahme aufgetreten sein könnte, wird **Exit()** nur dann aufgerufen, wenn `lockTaken` den Wert **true** besitzt.

Durch den **Exit()** - Aufruf im **finally**-Block ist sichergestellt, dass der kritische Block auch nach einem Ausnahmefehler verlassen wird.

Es ist durchaus erlaubt, die statische **Monitor**-Methode **Enter()** zum Erwerb eines Sperrobjekts direkt aufzurufen. Um einer Blockade vorzubeugen, kann sich ein Thread alternativ mit der statischen **Monitor**-Methode **TryEnter()** um ein Sperrobjekt bewerben:

public static bool TryEnter(Object *obj***)**

Diese Methode endet auf jeden Fall sofort und informiert mit einem Rückgabewert vom Typ **bool** darüber, ob die Berechtigung erteilt worden ist. Es sind Überladungen mit einem Timeout-Parameter vorhanden, sodass die maximale Wartezeit auf die Sperre festgelegt werden kann.

Wenn auf die **lock**-Anweisung und die damit verbundene Compiler-Unterstützung verzichtet und direkt mit der **Monitor**-Klasse gearbeitet wird (per **Enter()** oder **TryEnter()**), dann erhöhen sich die Flexibilität und die Verantwortung des Programmierers, insbesondere muss eine erworbene Sperre mit der statischen **Monitor**-Methode **Exit()** zurückgegeben werden.

17.1.4.1.2.2 Koordination per Wait() und Pulse()

Mit Hilfe der statischen **Monitor**-Methoden **Wait()** und **Pulse()** können in unserem betriebswirtschaftlichen Beispiel negative Lagerbestände verhindert werden. In der folgenden Variante der `Lager`-Methode `Liefere()` wird der Konsumenten-Thread mit der **Monitor**-Methode **Wait()** in den Zustand **WaitSleepJoin** versetzt, wenn seine Anfrage auf einen unzureichenden Lagerbestand trifft:

```
public bool Liefere(int sub) {
 nachfrage = true;
 lock (lockObject) {
  if (anz < maxAnz) {
   while (bilanz < sub) {
      Console.WriteLine("!!!!!!! {0,10} muss warten: " +
       "Keine {1, 3} Einheiten vorhanden um {2} Uhr.",
       Thread.CurrentThread.Name, sub, DateTime.Now.ToString("T", ci));
      Monitor.Wait(lockObject);
   }
   bilanz -= sub;
   anz++;
   Rumoren();
   Console.WriteLine("Nr. {0,2}: {1,10} entnimmt {2, 3} um {3} Uhr. Stand: {4}",
      anz, Thread.CurrentThread.Name, sub, DateTime.Now.ToString("T", ci), bilanz);
   nachfrage = false;
   return true;
  } else {
   Console.WriteLine("\nLieber " + Thread.CurrentThread.Name +
    ", es ist Feierabend!");
   nachfrage = false;
   return false;
  }
 }
}
```

Mit dem **Wait()** - Aufruf wird das exklusive Zutrittsrecht für den synchronisierten Block zurückgegeben, sodass im Beispiel der Produzenten-Thread zum Zug kommt und für Nachschub sorgen kann. Als Parameter ist im **Wait()** - Aufruf das synchronisierende Objekt anzugeben.

Um eine erfolgreiche Kooperation zu gewährleisten, muss der Produzenten-Thread nach jeder Lieferung die **Monitor**-Methode **Pulse()** mit dem synchronisierenden Objekt als Parameter aufrufen, um ggf. den wartenden Konsumenten-Thread zu reaktivieren, d.h. in den Zustand **Running** zu versetzen:

```
public bool Ergaenze(int add) {
 lock (lockObject) {
  if (anz < maxAnz || nachfrage) {
   bilanz += add;
   anz++;
   Rumoren();
   Console.WriteLine("Nr. {0,2}: {1,10} ergänzt  {2,3} um {3} Uhr. Stand: {4}",
    anz, Thread.CurrentThread.Name, add, DateTime.Now.ToString("T", ci), bilanz);
   Monitor.Pulse(lockObject);
   return true;
  } else {
   Console.WriteLine("\nLieber " + Thread.CurrentThread.Name +
    ", es ist Feierabend!");
   return false;
  }
 }
}
```

Der reaktivierte Konsumenten-Thread bewirbt sich wieder um Prozessorzeit und die Lagerzugangsberechtigung. Weil keinesfalls sicher ist, dass der Konsument nach der Reaktivierung einen ausreichenden Vorrat antrifft, findet sein **Wait()** - Aufruf in einer **while**-Schleife mit einleitender Bedingungsprüfung statt.

Mit Hilfe der zusätzlichen `Lager`-Instanzvariable `nachfrage`

```
bool nachfrage;
```

wird verhindert, dass sich der Konsumenten-Thread nach der geplanten Anzahl von Arbeitsvorgängen im Wartezustand befindet und nicht mehr per **Pulse()** befreit werden kann, weil der Produzenten-Thread bereits beendet ist. Das Ergebnis wäre ein dauerhaft blockiertes Programm. In der erweiterten Methode `Liefere()` meldet sich der Konsument beim Betreten des Lagers an und beim Verlassen wieder ab (siehe oben). In der Methode `Ergaenze()` wird dafür gesorgt, dass der Lagerist weiterarbeitet, solange sich der Konsument im Lager befindet. Schon das einfache Produzent-Lager-Konsument - Beispiel zeigt, dass die Thread-Koordination eine anspruchsvolle Aufgabe sein kann.

Nun produziert das Beispielprogramm nur noch realistische Lagerprotokolle, z. B.:

```
Das Lager ist offen (Bestand: 100)

Nr.  1:  Produzent ergänzt   41 um 03:51:00 Uhr. Stand: 141
Nr.  2:  Konsument entnimmt  78 um 03:51:00 Uhr. Stand: 63
!!!!!!!  Konsument muss warten: Keine  81 Einheiten vorhanden um 03:51:02 Uhr.
Nr.  3:  Produzent ergänzt  100 um 03:51:03 Uhr. Stand: 163
Nr.  4:  Konsument entnimmt  81 um 03:51:03 Uhr. Stand: 82
!!!!!!!  Konsument muss warten: Keine 103 Einheiten vorhanden um 03:51:07 Uhr.
Nr.  5:  Produzent ergänzt   33 um 03:51:07 Uhr. Stand: 115
Nr.  6:  Konsument entnimmt 103 um 03:51:07 Uhr. Stand: 12
Nr.  7:  Produzent ergänzt   66 um 03:51:09 Uhr. Stand: 78
Nr.  8:  Konsument entnimmt  70 um 03:51:09 Uhr. Stand: 8
!!!!!!!  Konsument muss warten: Keine  53 Einheiten vorhanden um 03:51:11 Uhr.
Nr.  9:  Produzent ergänzt   63 um 03:51:12 Uhr. Stand: 71
Nr. 10:  Konsument entnimmt  53 um 03:51:12 Uhr. Stand: 18
!!!!!!!  Konsument muss warten: Keine  95 Einheiten vorhanden um 03:51:14 Uhr.
```

```
Nr. 11:  Produzent ergänzt   36 um 03:51:15 Uhr. Stand: 54
!!!!!!!  Konsument muss warten: Keine  95 Einheiten vorhanden um 03:51:15 Uhr.
Nr. 12:  Produzent ergänzt   22 um 03:51:18 Uhr. Stand: 76
!!!!!!!  Konsument muss warten: Keine  95 Einheiten vorhanden um 03:51:18 Uhr.
Nr. 13:  Produzent ergänzt   75 um 03:51:20 Uhr. Stand: 151
Nr. 14:  Konsument entnimmt  95 um 03:51:20 Uhr. Stand: 56
Nr. 15:  Produzent ergänzt   46 um 03:51:23 Uhr. Stand: 102
Nr. 16:  Konsument entnimmt  79 um 03:51:24 Uhr. Stand: 23
!!!!!!!  Konsument muss warten: Keine  33 Einheiten vorhanden um 03:51:25 Uhr.
Nr. 17:  Produzent ergänzt   44 um 03:51:25 Uhr. Stand: 67
Nr. 18:  Konsument entnimmt  33 um 03:51:25 Uhr. Stand: 34
Nr. 19:  Produzent ergänzt   22 um 03:51:26 Uhr. Stand: 56
!!!!!!!  Konsument muss warten: Keine  95 Einheiten vorhanden um 03:51:28 Uhr.
Nr. 20:  Produzent ergänzt   26 um 03:51:29 Uhr. Stand: 82
!!!!!!!  Konsument muss warten: Keine  95 Einheiten vorhanden um 03:51:29 Uhr.
Nr. 21:  Produzent ergänzt   39 um 03:51:31 Uhr. Stand: 121
Nr. 22:  Konsument entnimmt  95 um 03:51:31 Uhr. Stand: 26

Lieber Produzent, es ist Feierabend!

Lieber Konsument, es ist Feierabend!
```

Wenn nach der geplanten Anzahl von Arbeitsvorgängen noch eine unversorgte Konsumentenanforderung ansteht, dann muss der Lagerist so lange weiterarbeiten, bis der Produzent für einen ausreichenden Lagerbestand gesorgt hat, um den Konsumenten bedienen zu können. Um den relativ seltenen Fall von mehr als 20 Lagerzugriffen beobachten zu können, werden die **Random**-Objekte in den Klassen `Produzent` und `Konsument` nun über den parameterfreien Konstruktor erstellt, der einen zeitabhängigen Startwert für den Pseudozufallszahlengenerator verwendet.

Zu jedem Sperrobjekt gehören zwei Warteschlangen:

- In der **Ready**-Warteschlange befinden sich arbeitswillige Threads, die auf das Sperrobjekt warten.
- In der **Wait**-Warteschlange befinden sich Threads, die auf eine Zustandsänderung angewiesen sind und danach das Sperrobjekt erwerben möchten. Ohne Benachrichtigung über eine Zustandsänderung verbleiben die wartenden Threads auch dann passiv, wenn das Sperrobjekt frei ist. Der **Wait()** - Aufruf kehrt erst dann zurück (mit dem Rückgabewert **true**), wenn der aufrufende Thread reaktiviert worden ist und das Sperrobjekt wieder erworben hat.

Die Methode **Pulse()** informiert aus der **Wait**-Warteschlange den Thread mit dem ältesten **Wait()** - Aufruf über eine Zustandsänderung und befördert ihn in die **Ready**-Warteschlange. Als Alternative ist die Methode **PulseAll()** verfügbar, die *alle* wartenden Threads in die **Ready**-Warteschlange befördert.

Zu **Wait()** sind Überladungen mit Timeout Parameter vorhanden. Wird ein wartender Thread nicht innerhalb der Timeout-Zeitspanne per **Pulse()** oder **PulseAll()** reaktiviert, wechselt er in den Zustand **Ready**, bewirbt sich also erneut um das Sperrobjekt.

Die **Monitor**-Methoden **Wait()**, **Pulse()** und **PulseAll()** dürfen nur in einem synchronisierten Block aufgerufen werden.

17.1.4.1.3 SpinLock-Sperre statt lock-Anweisung

Versucht ein Thread vergeblich, über die **Monitor**-Methode **Enter()** ein Sperrobjekt zu erwerben (entweder durch den direkten Aufruf der Methode oder per **lock**-Anweisung), dann wird er blockiert. Dieser Übergang in den Zustand **WaitSleepJoin** verursacht durch den Kontakt mit der Thread-Verwaltung des Betriebssystems und den erforderlichen Kontextwechsel einen nennenswerten Zeitaufwand. Stattdessen könnte der Thread aktiv bleiben und sich in einer Schleife wiederholt um den Zugang zum geschützten Bereich bemühen. Er könnte z. B. eine Methode zur Beantragung

der Sperre, die per **bool**-Rückgabe den Erfolg meldet in einer **while**-Schleife mit leerer Anweisung bis zur positiven Rückmeldung aufrufen:

```
while (!DeliverLock()) ;
```

Dabei wird zwar auch CPU-Zeit verbraucht, doch kann diese Technik dem Blockieren des Threads überlegen sein, wenn mit sehr kurzen Wartezeiten zu rechnen ist. Das Warten auf ein exklusives Zugangsrechts per Schleife statt Blockade wird in der BCL durch die Struktur **SpinLock** im Namensraum **System.Threading** unterstützt.

Allerdings ist es mir nicht gelungen, ein Anwendungsbeispiel für den potentiellen Einspareffekt der Warteschleifentechnik zu konstruieren. Auch das Demonstrationsprogramm auf der folgenden Microsoft-Seite

https://docs.microsoft.com/de-de/dotnet/standard/threading/how-to-use-spinlock-for-low-level-synchronization

liefert keine Überlegenheit der **SpinLock**-Sperre gegenüber der **lock**-Anweisung. Lässt man es in unveränderter Form ausführen, scheint sich ein Vorteil für die **SpinLock**-Technik zu zeigen:

```
elapsed ms with lock: 162
elapsed ms with spinlock: 50
```

Es liegt jedoch ein Reihenfolgeeffekt vor, der das zuerst getestete Verfahren benachteiligt. In einem „Nachtest“ schneidet die **lock**-Anweisung besser ab:

```
elapsed ms with lock: 35
```

Weil die Verwendung der **SpinLock**-Struktur (im Unterschied zur **lock**-Anweisung) nicht direkt durch den Compiler unterstützt wird, muss der Programmierer außerdem einen Mehraufwand betreiben, z. B.:

Verwendung einer **lock**-Anweisung	Verwendung einer **SpinLock**-Sperre

`static void UpdateWithLock(Data d, int i) {`
`  lock (_lock) {`
`  _queue.Enqueue(d);`
`  }`
`}`
	`private static void UpdateWithSpinLock(Data d, int i) {`
	`  bool lockTaken = false;`
	`  try {`
	`  _spinlock.Enter(ref lockTaken);`
	`  _queue.Enqueue(d);`
	`  } finally {`
	`  if (lockTaken) _spinlock.Exit(false);`
	`  }`
	`}`

Folglich lohnt sich die Verwendung der **SpinLock**-Sperre wohl kaum.

17.1.4.1.4 ReaderWriterLockSlim

Anders als bei einer Sperre per **Monitor** oder **SpinLock** bietet eine Sperre mit Hilfe der Klasse **ReaderWriterLockSlim** die Option, lese- und schreibwillige Threads unterschiedlich zu behandeln:

- Mehrere lesende Threads können sich simultan im geschützten Bereich aufhalten.
- Ein schreibender Thread benötigt den exklusiven Zutritt.

Ein Einsatz der Klasse **ReaderWriterLockSlim** verspricht ein höheres Maß an Parallelität, doch führen die komplexen Spielregeln für die Vergabe von Zutrittsrechten zu einem erhöhten Aufwand der CLR bei der Synchronisation.[1] Folglich dauert der Erwerb einer Zugangsberechtigung länger als bei der Klasse **Monitor**, und vor einer Verwendung der Klasse **ReaderWriterLockSlim** sollten vergleichende Leistungsmessungen durchgeführt werden.

---

[1] https://docs.microsoft.com/de-de/dotnet/api/system.threading.readerwriterlockslim

Am ehesten ist ein Nutzen der Klasse **ReaderWriterLockSlim** zu erwarten, wenn die zu synchronisierenden Threads auf die gemeinsame Ressource meist lesend und nur selten schreibend zugreifen:

- Nachdem ein schreibwilliger Thread die Methode **EnterWriteLock()** der Klasse **ReaderWriterLockSlim** oder die mit einem Timeout-Parameter ausgestattetet Alternative **TryEnterWriteLock()** erfolgreich aufgerufen hat, ist kein weiterer Zugriff möglich, bis die Schreibberechtigung mit einem Aufruf der **ReaderWriterLockSlim**-Methode **ExitWriteLock()** zurückgegeben wird.
- Nachdem ein lesewilliger Thread die Methode **EnterReadLock()** der Klasse **ReaderWriterLockSlim** oder die mit einem Timeout-Parameter ausgestattete Alternative **TryEnterReadLock()** erfolgreich aufgerufen hat, ist kein Schreibzugriff möglich, während simultane Lesezugriffe durch andere Threads erlaubt sind. Hat jeder Thread mit Leseberechtigung die **ReaderWriterLockSlim**-Methode **ExitReadLock()** aufgerufen, dann kann wieder eine Schreibberechtigung erworben werden.

Im folgenden Beispiel greift *ein* Thread schreibend auf eine Ressource (einen **decimal**-Wert) zu, während drei andere Threads nur lesend zugreifen:

```
using System;
using System.Threading;

class ReaderWriterLockSlimDemo {
  static decimal dwert = 13.0M;
  static readonly ReaderWriterLockSlim rwls = new ReaderWriterLockSlim();

  static void WM() {
    for (int i = 1; i <= 3; i++) {
      rwls.EnterWriteLock();
      dwert++;
      Console.WriteLine("\n*** Schreibzugriff " + i + ". Neuer Wert: " + dwert +
        ", aktive Leser: " + rwls.CurrentReadCount + " (" + DateTime.Now.ToString()+")");
      Thread.Sleep(2000);
      rwls.ExitWriteLock();
      Thread.Sleep(1000);
    }
  }

  static void RM() {
    for (int i = 1; i <= 3; i++) {
      rwls.EnterReadLock();
      Console.WriteLine(Thread.CurrentThread.Name + " ermittelt Wert " + dwert +
        ", aktive Leser: " + rwls.CurrentReadCount + " (" + DateTime.Now.ToString()+")");
      Thread.Sleep(500);
      rwls.ExitReadLock();
      Thread.Sleep(1000);
    }
  }

  static void Main() {
    new Thread(WM).Start();
    Thread rt;
    for (int i = 0; i < 3; i++) {
      Thread.Sleep(50);
      rt = new Thread(RM);
      rt.Name = "RT" + Convert.ToString(i);
      rt.Start();
    }
  }
}
```

Wie das folgende Ablaufprotokoll zeigt, behindern sich die 3 lesenden Threads nicht gegenseitig. Sie müssen aber warten, während der schreibende Thread zugreift:

```
*** Schreibzugriff 1. Neuer Wert: 14,0, aktive Leser: 0 (04.04.2021 13:46:17)
RT2 ermittelt Wert 14,0, aktive Leser: 1 (04.04.2021 13:46:19)
RT1 ermittelt Wert 14,0, aktive Leser: 2 (04.04.2021 13:46:19)
RT0 ermittelt Wert 14,0, aktive Leser: 3 (04.04.2021 13:46:19)

*** Schreibzugriff 2. Neuer Wert: 15,0, aktive Leser: 0 (04.04.2021 13:46:20)
RT0 ermittelt Wert 15,0, aktive Leser: 3 (04.04.2021 13:46:22)
RT1 ermittelt Wert 15,0, aktive Leser: 3 (04.04.2021 13:46:22)
RT2 ermittelt Wert 15,0, aktive Leser: 3 (04.04.2021 13:46:22)

*** Schreibzugriff 3. Neuer Wert: 16,0, aktive Leser: 0 (04.04.2021 13:46:23)
RT2 ermittelt Wert 16,0, aktive Leser: 2 (04.04.2021 13:46:25)
RT0 ermittelt Wert 16,0, aktive Leser: 3 (04.04.2021 13:46:25)
RT1 ermittelt Wert 16,0, aktive Leser: 2 (04.04.2021 13:46:25)
```

Die aktuelle Zahl der mit Leserecht versorgten Threads kann über die **ReaderWriterLockSlim**-Eigenschaft **CurrentReadCount** ermittelt werden.

Auf keinen Fall sollte die ältere Klasse **ReaderWriterLock** verwendet werden, die nur noch aus Kompatibilitätsgründen vorhanden ist. Sie ist bekannt für eine schlechte Performanz und eine hohe Deadlock-Neigung.[1]

17.1.4.1.5 Mutex und Semaphore

Die Klasse **Mutex** (engl. *mutual exclusion*) erbringt ähnliche Leistungen wie die Klasse **Monitor** und erlaubt dabei auch eine Thread-Synchronisation über Prozessgrenzen hinweg (z. B. zwischen verschiedenen Programmen). Diese Flexibilität ist allerdings mit einem erheblichen Zeitaufwand verbunden, was speziell bei häufigen Thread-Wechseln von Bedeutung ist.

Über die Klasse **Semaphore** kann die Zutrittsexklusivität insofern relativiert werden, dass nicht nur *ein* Thread den geschützten Bereich betreten darf, sondern eine im Konstruktor festgelegte Anzahl. Wird der **WaitOne()** - Antrag eines Threads positiv entschieden, reduziert sich die Zahl der verbleibenden „Eintrittskarten". Trifft ein **WaitOne()** - Antrag auf den Zählerstand 0, dann wird der anfragende Thread blockiert. Per **Release()** kann ein Thread seine Eintrittskarte zurückgeben. Wie die Klasse **Mutex** kann auch die Klasse **Semaphore** prozessübergreifend operieren. Das ist wie bei der Klasse **Mutex** mit einem erheblichen Zeitaufwand zu bezahlen. Ist nur eine Prozess-interne Verwaltung von *k* simultanen Zutrittsberechtigungen erforderlich, dann sollte die Klasse **SemaphoreSlim** verwendet werden.

17.1.4.2 Atomare Operationen und volatile Variablen

Im Abschnitt 17.1.4.1.1 wurde erläutert, dass auch eine einfache Operation wie das Inkrementieren einer Ganzzahlvariablen *nicht atomar* ist, also durch Thread-Wechsel unterbrochen werden kann, wobei es zu irregulären Variableninhalten und damit zu einem sehr schwer aufzuklärenden Programmfehler kommen kann. Durch statische Methoden der Klasse **Interlocked** im Namensraum **System.Threading** erhält man atomare Varianten von diversen Operationen, z. B.:

- **public static long Add(ref long** *location***, long** *value***)**
  Zur **long**-Variablen *location* wird der Wert des zweiten Parameters addiert.
- **public static long Increment(ref long** *location***)**
  Die **long**-Variable *location* wird inkrementiert.

---

[1] Bei einem Deadlock blockieren sich Threads gegenseitig (siehe Abschnitt 17.1.7).

Eine solche Sicherung gegen Thread- Synchronisationsfehler verursacht geringere Kosten als eine **lock**-Anweisung. Sie wird oft als *lock-free* bezeichnet, wobei die Bezeichnung *low-lock* zutreffender ist. Morrison (2005b) nennt zwei Nachteile der low-lock - Synchronisation:

- Schmaler Anwendungsbereich
- Erhebliches Fehlerrisiko
  Es sind Details im Speichersystem und Compiler zu beachten, die normalerweise für Programmierer irrelevant sind.

Insgesamt empfehlen Griffiths (2020) und Morrison (2005b), die low-lock - Techniken nur dann einzusetzen, wenn unproblematischere Synchronisationstechniken wie die **lock**-Anweisung zu hohe Kosten verursachen.

Im Zusammenhang mit der Klasse **Interlocked** wird oft der für (statische) Felder erlaubte Modifikator **volatile** diskutiert, z. B.:

```
static volatile int ivar;
```

Er sorgt dafür, dass der Compiler bei Mehrkernprozessoren darauf verzichtet, CPU-lokalen Cache-Speicher zur Leistungssteigerung einzusetzen. Bei so ausgezeichneten Variablen sehen alle Threads zu einem Zeitpunkt denselben Speicherinhalt. Eine Thread-Synchronisation wird jedoch im Allgemeinen *nicht* erreicht. Es kann z. B. weiterhin passieren, dass ein Thread beim Inkrementieren einer Variablen zwischen dem Lesen des alten und dem Schreiben des neuen Werts unterbrochen wird, sodass schlussendlich ein falscher Wert im Speicher landet. Nur unter speziellen Umständen kann der Modifikator **volatile** Thread- Synchronisationsfehler verhindern, z. B.:

- Auf eine Variable greift nur *ein* Thread schreibend zu.
- Zugriffe auf diese Variable erfolgen atomar (siehe oben).

17.1.4.3 Signalisierungsobjekte

Mit den **Monitor**-Methoden **Wait()**, **Pulse()** und **PulseAll()** lässt sich ein koordiniertes Agieren und eine Kommunikation von Threads realisieren, doch bestehen zwei wesentliche Einschränkungen:

- Die Methoden dürfen nur in einem synchronisierten Block aufgerufen werden.
- Die Methoden **Pulse()** - und **PulseAll()** haben den Zweck, auf ein konkretes Sperrobjekt wartende Threads ggf. zu wecken. Folglich verhallt ein Aufruf ohne Effekt, wenn gerade kein Thread auf das Sperrobjekt wartet.

Anschließend wird beschrieben, wie ein Thread an einer beliebigen Stelle ein mehr oder weniger dauerhaftes Signal für andere Threads setzen kann.

Wenn ein Thread ein „auf Grün“ stehendes Signalisierungsobjekt der Klasse **AutoResetEvent** benötigt, dann kann er sich durch den Aufruf der Methode **WaitOne()** darum bewerben. Dieser Methodenaufruf endet bei gesetzem Signal mit dem Rückgabewert **true** und blockiert anderenfalls, sodass der aufrufende Thread in den Zustand **WaitSleepJoin** versetzt wird.

Ein anderer Thread ist dafür zuständig, das Signal (**AutoResetEvent**-Objekt) über die Methode **Set()** „auf Grün“ zu stellen. Wenn gerade ein Thread auf das Signal wartet, dann wird er reaktiviert. Es profitieren aber auch Threads, deren Interesse am Signal erst später entsteht.

Wie bei einer Drehkreuztür mit Einzeldurchlass kann pro **Set()** - Aufruf nur *ein* Thread profitieren, wobei das gesetzte Signal automatisch verbraucht wird (daher der Klassenname **AutoResetEvent**).

Soll stattdessen eine Tür für *alle* von einem Signal abhängige Threads geöffnet werden, dann verwendet man statt der Klasse **AutoResetEvent** das Gegenstück **ManualResetEvent**.[1] Bei einem Signalisierungsobjekt dieser Klasse bleibt ein gesetztes Signal erhalten bis zur Stornierung über die Methode **Reset()**.

Im folgenden Programm wird über ein **AutoResetEvent**-Objekt dafür gesorgt, dass zwei Threads streng alternierend tätig werden:

```
using System;
using System.Threading;

class AutoResetEventDemo {
 static Thread t1, t2;
 static AutoResetEvent are = new AutoResetEvent(false); // Signal ist initial aus.

 static void M() {
  int t = 1;
  if (Thread.CurrentThread == t2) {
   t = 2;
   are.WaitOne(); // t2 geht als erster in den Wartezustand.
  }
  for (int j = 1; j <= 2; j++) {
   Console.Write("\n" + Thread.CurrentThread.Name + " ist am Zug: ");
   for (int i = 1; i <= 5; i++) {
    Console.Write(t + " ");
    Thread.Sleep(200);
   }
   Console.WriteLine("\n" + Thread.CurrentThread.Name +
                     " Setzt das Signal und dann sich selbst zur Ruhe.");
   are.Set();
   Thread.Sleep(100); // Nötig, damit der andere Thread das Signal verbraucht.
   are.WaitOne();
  }
  Console.WriteLine("\n"+Thread.CurrentThread.Name+" endet.");
  are.Set(); // Nötig, damit der wartende Kollege zum letzten Auftritt geweckt wird.
 }

 static void Main() {
  t1 = new Thread(M); t2 = new Thread(M);
  t1.Name = "t1";
  t2.Name = "t2";
  t1.Start();
  t2.Start();
 }
}
```

Ein Programmlauf produziert die folgenden Ausgaben:

[1] Bei den Klassen **AutoResetEvent** und **ManualResetEvent** hat der Namensbestandteil *Event* keinen Bezug zu .NET - Ereignissen im Sinne von Kapitel 10.2. Hintergrund der Benennung ist ein Bezug zu der im Win32-API für die Thread- Synchronisation verfügbaren Funktion **CreateEvent()**.

```
t1 ist am Zug: 1 1 1 1 1
t1 Setzt das Signal und dann sich selbst zur Ruhe.

t2 ist am Zug: 2 2 2 2 2
t2 Setzt das Signal und dann sich selbst zur Ruhe.

t1 ist am Zug: 1 1 1 1 1
t1 Setzt das Signal und dann sich selbst zur Ruhe.

t2 ist am Zug: 2 2 2 2 2
t2 Setzt das Signal und dann sich selbst zur Ruhe.

t1 endet.

t2 endet.
```

Das verwendete **AutoResetEvent** - Objekt erhält per Konstruktorparameter einen inaktiven Startzustand:

```
static AutoResetEvent are = new AutoResetEvent(false);
```

Hat der momentan aktive Thread eine Etappe bewältigt, setzt er das Freigabesignal und wartet ein Weilchen, damit das Signal vom wartenden Kollegen verbraucht wird. Dann stellt er sich selbst wieder hinter der Drehtür an:

```
are.Set();
Thread.Sleep(100); // Nötig, damit der andere Thread das Signal verbraucht.
are.WaitOne();
```

Wenn mehrere Threads aufgrund eines **WaitOne()** - Aufrufs vor einer Drehkreuztür vom Typ **AutoResetEvent** warten, dann kann beim Setzen des Signals genau *einer* passieren, wobei das Signal automatisch abgeschaltet wird.

Während ein Aufruf der **Monitor**-Methode **Pulse()** ungehört und effektfrei verhallen kann, weil gerade kein Thread auf das zugehörige Sperrobjekt wartet, ermöglicht ein Aufruf der **AutoResetEvent**-Methode **Set()** unabhängig von der aktuellen Nachfragesituation bei der überwachten Drehtür eine Einzelpasssage. Ist das Signal gerade gesetzt, bleibt ein **Set()** - Aufruf allerdings ebenfalls folgenlos.

In der gemeinsamen Basisklasse **EventWaitHandle** von **AutoResetEvent** und **ManualResetEvent** sind die statischen Methoden **WaitAny()** und **WaitAll()** verfügbar. In einem Aufruf dieser Methoden kann ein Thread einen ganzen Array mit Signalisierungsobjekten angeben, um geweckt zu werden, wenn *irgendein* Signal gesetzt worden ist bzw. wenn *alle* Signale gesetzt worden sind. Allerdings wird die Methode **WaitAll()** nicht unterstützt, wenn unter Windows ein Thread im **STAThread** - Modus läuft.[1] Das gilt z. B. für die Threads im Rahmen einer WPF-Anwendung, weil deren **Main()** - Methode mit dem **STAThreadAttribute** dekoriert werden muss (siehe Abschnitt 12.2.1).

Die Klassen **AutoResetEvent** und **ManualResetEvent** erlauben eine Thread-Koordination über Prozessgrenzen hinweg. Mit dieser (in der Regel nicht benötigten) Option ist ein nennenswerter Zeitaufwand verbunden, sodass ein Einsatz der **EventWaitHandle**-Ableitungen beim kurz getakteten Start-Stopp - Betrieb *nicht* ratsam ist.

Die Klasse **ManualResetEventSlim**, die von **System.Object** abstammt, bietet eine zur Klasse **ManualResetEvent** analoge Funktionalität und arbeitet dabei aus zwei Gründen etwas flotter:

---

[1] https://docs.microsoft.com/de-de/dotnet/api/system.threading.waithandle.waitall

- Bevor ein Anwärter-Thread blockiert, versucht er mehrmals sein Glück (analog zur Struktur **SpinLock**, siehe Abschnitt 17.1.4.1.3), sodass gelegentlich der relativ zeitaufwändige Kontakt mit der Thread-Verwaltung des Betriebssystems eingespart wird. Nach einer begrenzten Zahl von gescheiterten Anträgen, begibt sich ein **ManualResetEventSlim**-Objekt aber doch in den blockierten Zustand.
- Es wird auf die Option zur Überschreitung von Prozessgrenzen verzichtet. Wo diese Option benötigt wird, ist weiterhin die Klasse **ManualResetEvent** zu verwenden.

Zur Klasse **AutoResetEvent** existiert *keine* Slim-Variante.

Über ein Objekt der Klasse **CountdownEvent**, kann sich ein Thread reaktivieren lassen, wenn ein Signal *k* mal gesetzt worden ist (siehe Übungsaufgabe im Abschnitt 17.7).

17.1.4.4 Einfache Verfahren zur Thread-Koordination

In diesem Abschnitt werden zwei einfache Verfahren beschrieben, die es einem Thread erlauben, auf eine Bedingung zu warten:

- Auf den Ablauf einer Wartezeit
- Auf die Beendigung eines anderen Threads

17.1.4.4.1 Ein Schläfchen in Ehren

Der Vollständigkeit halber soll die schon mehrfach benutzte statische **Thread**-Methode **Sleep()** noch einmal erwähnt werden, mit der sich der aufrufende Thread für eine per Parameter bestimmte Anzahl von Millisekunden in den Zustand **WaitSleepJoin** versetzt und den restlichen Threads die CPU-Leistung überlässt, z. B.:

```
Thread.Sleep(1000);
```

17.1.4.4.2 Weck mich, wenn Du fertig bist

Will ein Thread `t1` in den Wartezustand wechseln, bis der Thread `t2` seine Tätigkeit beendet hat, dann kann er diesen Plan durch einen Aufruf der **Thread**-Methode **Join()** realisieren, z. B.:

```
using System;
using System.Threading;

class JoinDemo {
   static Thread t1, t2;

   static void M1() {
      Console.WriteLine("t1 in M1");
      Thread.Sleep(100);
      Console.WriteLine("\nt1 beginnt seine Arbeit:");
      for (int i = 1; i <= 3; i++) {
         Console.Write(1 + " ");
         Thread.Sleep(500);
      }
      Console.WriteLine("\nt1 endet");
   }
```

```
    static void M2() {
      Console.WriteLine("t2 in M2, wartet auf t1");
      t1.Join();
      Console.WriteLine("\nt2 beginnt seine Arbeit:");
      for (int i = 1; i <= 3; i++) {
        Console.Write(2 + " ");
        Thread.Sleep(500);
      }
      Console.WriteLine("\nt2 beendet");
    }

    static void Main() {
      t1 = new Thread(new ThreadStart(M1));
      t2 = new Thread(new ThreadStart(M2));
      t1.Start(); t2.Start();
    }
  }
```

Das Programm produziert die folgende Ausgabe:

```
t1 in M1
t2 in M2, wartet auf t1

t1 beginnt seine Arbeit:
1 1 1
t1 endet

t2 beginnt seine Arbeit:
2 2 2
t2 beendet
```

17.1.5 Threads vorzeitig beenden

Ein ungestört agierender Thread endet mit der zugrundeliegenden Startmethode. Gelegentlich soll ein Thread aber schon vorher gestoppt werden, weil z. B. der Benutzer per **Abbrechen**-Schalter eine entsprechende Absicht bekundet hat. Im aktuellen Abschnitt werden zwei Möglichkeiten beschrieben, einen Thread vorzeitig zu beenden:

- Die statische **Thread**-Methode **Abort()**
  Diese Technik zum Beenden eines Threads ist nur dann akzeptabel, wenn ein Thread sich selbst beendet. In .NET 5.0 wird die Methode nicht mehr unterstützt.
- Das seit .NET Framework 4.0 verfügbare kooperative Terminierungsverfahren

17.1.5.1 Abort()

Vor der statischen **Thread**-Methode **Abort()** ist über viele Jahre gewarnt worden, und seit .NET Core wird ihre Verwendung nun wirksam unterbunden. In .NET 5.0 warnt der Compiler:

```
"Thread.Abort()" ist veraltet: "Thread.Abort is not supported and throws
PlatformNotSupportedException."
```

Wird die Methode in .NET 5.0 trotzdem aufgerufen, wird die Warnung zur Realität:

```
System.PlatformNotSupportedException: Thread abort is not supported on this platform.
```

Im .NET Framework ist die Methode **Abort()** aber noch erlaubt. Damit lässt sich ein Thread in fast jedem Zustand stoppen (Albahari 2014). Insbesondere ist die Methode wirksam bei den Thread-Zuständen **Running** und **WaitSleepJoin**, auf die wir unsere Betrachtungen beschränken. Ein **Abort()** - Aufruf befördert den angesprochenen Thread in den Zustand **AbortRequested** und löst eine **ThreadAbortException** aus, sodass per Ausnahmebehandlung für einen sinnvollen Abgang gesorgt werden kann.

Am Ende eines behandelnden **catch**-Blocks wird die **ThreadAbortException** automatisch erneut ausgelöst, sofern dies nicht über einen Aufruf der Methode **ResetAbort()** verhindert wird (siehe unten). Eine weitere Besonderheit der **ThreadAbortException** besteht darin, dass sie *nicht* zur Beendigung des Programms führt, wenn sie unbehandelt bleibt. Während der betroffene Thread endet, laufen die anderen Threads des Programms weiter, wenn es sich bei dem abgewürgten Thread nicht um den letzten Vordergrund-Thread des Programms gehandelt hat.

Wir beschäftigen uns zunächst mit dem Fall, dass sich ein Thread per **Abort()** - Aufruf selbst beendet. Dabei könnte nur ein Programmierfehler zu einer schwierigen Lage führen. Als Beispiel betrachten wir eine kunden*un*freundliche Variante des Produzent-Lager-Konsument - Programms: Die `Lager`-Methode `Liefere()` terminiert den Konsumenten-Thread, sobald dieser mit einem Wunsch über den Lagerbestand hinausgeht:

```
public bool Liefere(int sub) {
 lock (lockObject) {
  if (anz < maxAnz) {
   if (bilanz < sub) {
    Console.WriteLine("!! {1,10} fordert  {2, 3}" +
                      " um {3} Uhr und wird ausgeschlossen (Abbruch des Threads).",
     anz, Thread.CurrentThread.Name, sub, DateTime.Now.ToString("T", ci));
    Thread.CurrentThread.Abort();
   }
   anz++;
   bilanz -= sub;
   Rumoren();
   Console.WriteLine("Nr. {0,2}: {1,10} entnimmt {2, 3} um {3} Uhr. Stand: {4}",
    anz, Thread.CurrentThread.Name, sub, DateTime.Now.ToString("T", ci), bilanz);
   return true;
  } else {
   Console.WriteLine("\nLieber " + Thread.CurrentThread.Name +
    ", es ist Feierabend!");
   return false;
  }
 }
}
```

Der Konsument nutzt die Ausnahmebehandlung für eine Beschwerde:

```
public void Run() {
   Random rand = new Random(2);
   try {
      do {
         offen = pl.Liefere((5 + rand.Next(100)));
         Thread.Sleep(1000 + rand.Next(3000));
      } while (offen);
   } catch (ThreadAbortException) {
      Console.WriteLine("Als Kunde muss ich mir so etwas " +
         "nicht gefallen lassen!");
   }
}
```

Nach der Ausnahmebehandlung gelangt der Konsumenten-Thread in der Zustand **Stopped**. Das Programm läuft hingegen weiter, weil noch ein Vordergrund-Thread existiert, und im Lager taucht nur noch der Produzent auf:

```
Das Lager ist offen (Bestand: 100)

Nr.  1:  Produzent ergänzt   29 um 03:22:27 Uhr. Stand: 129
Nr.  2:  Konsument entnimmt  82 um 03:22:27 Uhr. Stand: 47
Nr.  3:  Produzent ergänzt   51 um 03:22:29 Uhr. Stand: 98
Nr.  4:  Konsument entnimmt  21 um 03:22:30 Uhr. Stand: 77
Nr.  5:  Produzent ergänzt   70 um 03:22:32 Uhr. Stand: 147
Nr.  6:  Konsument entnimmt  15 um 03:22:34 Uhr. Stand: 132
Nr.  7:  Produzent ergänzt   40 um 03:22:34 Uhr. Stand: 172
Nr.  8:  Konsument entnimmt  85 um 03:22:36 Uhr. Stand: 87
Nr.  9:  Konsument entnimmt  27 um 03:22:38 Uhr. Stand: 60
Nr. 10:  Produzent ergänzt   15 um 03:22:38 Uhr. Stand: 75
!!  Konsument fordert   81 um 03:22:39 Uhr und wird ausgeschlossen (Abbruch des Threads).
Als Kunde muss ich mir so etwas nicht gefallen lassen!
Nr. 11:  Produzent ergänzt    7 um 03:22:41 Uhr. Stand: 82
Nr. 12:  Produzent ergänzt   37 um 03:22:43 Uhr. Stand: 119
Nr. 13:  Produzent ergänzt   73 um 03:22:47 Uhr. Stand: 192
Nr. 14:  Produzent ergänzt   33 um 03:22:50 Uhr. Stand: 225
Nr. 15:  Produzent ergänzt   75 um 03:22:53 Uhr. Stand: 300
Nr. 16:  Produzent ergänzt   99 um 03:22:56 Uhr. Stand: 399
Nr. 17:  Produzent ergänzt   21 um 03:22:57 Uhr. Stand: 420
Nr. 18:  Produzent ergänzt   84 um 03:22:59 Uhr. Stand: 504
Nr. 19:  Produzent ergänzt   84 um 03:23:01 Uhr. Stand: 588
Nr. 20:  Produzent ergänzt   87 um 03:23:03 Uhr. Stand: 675

Lieber Produzent, es ist Feierabend!
```

Bevor ein abgebrochener Thread in den Zustand **Stopped** gelangt, werden natürlich alle beteiligten **finally**-Blöcke ausgeführt, was den Abbruch hinauszögern kann.

Ein Thread im Zustand **AbortRequested** kann durchaus in den Zustand **Running** zurückkehren. Dazu muss lediglich in einer **ThreadAbortException**-Ausnahmebehandlung die **Thread**-Methode **ResetAbort()** aufgerufen werden, was der Konsument in der folgenden Variante unseres Beispiels tut:

```
public void Run() {
   Random rand = new Random(2);
   do {
      try {
         offen = pl.Liefere((5 + rand.Next(100)));
      } catch (ThreadAbortException) {
         Console.WriteLine("Als Kunde muss ich mir so etwas " +
            "nicht gefallen lassen!");
         Thread.ResetAbort();
      }
      Thread.Sleep(1000 + rand.Next(3000));
   } while (offen);
}
```

Wenn ein Thread sich selbst unterbricht, ist der Programmablauf unter Kontrolle. Wenn einem Thread hingegen von einem *anderen* Thread eine **ThreadAbortException** - Ausnahme aufgezwungen wird, dann sind der Unterbrechungspunkt und die möglichen Schäden unkalkulierbar (siehe z. B. Albahari 2014). Außerdem ist nicht garantiert, dass die **ThreadAbortException** tatsächlich zur Beendigung des Threads führt:

- Der Thread kann die Methode **ResetAbort()** aufrufen
- oder in einem **finally**-Block eine lang dauernde Aktivität entfalten.

Statt einen „fremden“ Thread per **Abort()** zu beenden, sollte ein auf Kooperation basierendes Terminierungsverfahren verwendet werden (siehe Abschnitt 17.1.5.2). Generell sollte man einen Thread nur dann starten, wenn seine kooperative Reaktion auf ein Terminierungssignal sichergestellt ist.[1]

---

[1] Stack Overflow - Beitrag von Eric Lippert (2010): http://stackoverflow.com/questions/2251964/c-sharp-thread-termination-and-thread-abort

17.1.5.2 Kooperative Terminierung

Wird ein *fremder* Thread per **Abort()** gestoppt, was nur im .NET Framework erlaubt ist, dann sind der Unterbrechungspunkt und die möglichen Schäden unkalkulierbar. Im folgenden Beispielprogramm besteht der Schaden allerdings nur darin, dass eine Bildschirmausgabe an einer unpassenden Stelle abgebrochen wird:

Quellcode	Ausgabe

```
using System;
using System.Threading;

public class AbortForeign {
    void Write(string s) {
        for (int i = 0; i < s.Length; i++) {
            Console.Write(s[i]);
            Thread.Sleep(100);
        }
        Console.WriteLine();
    }

    public void Run() {
        try {
            for (int i = 0; i < 5; i++) {
                Write("Rumoren im Runner, i = " + i);
                Thread.Sleep(1000);
            }
        } catch (ThreadAbortException) {
            Console.WriteLine("Autsch!");
        }
    }

    static void Main() {
        var runner = new AbortForeign();
        var t = new Thread(runner.Run);
        t.Start();
        Thread.Sleep(8000);
        t.Abort();
    }
}
```

```
Rumoren im Runner, i = 0
Rumoren im Runner, i = 1
Rumoren im RAutsch!
```

Der mit **Abort()** abgebrochene Thread basiert auf der Instanzmethode `Run()`, die eine Methode namens `Write()` zur Bildschirmausgabe verwendet. Weil sich `Write()` für einen Ausgabeauftrag viel Zeit lässt, ist eine Unterbrechung gut zu beobachten.

Mit dem .NET Framework 4.0 wurde unter dem Namen *Unified Cancellation Framework* (dt.: *kooperatives Abbruchmodell*) ein auf Kooperation basierendes Verfahren etabliert, um Threads oder Aufgaben (siehe Abschnitt 17.4) abzubrechen.[1] Mit Hilfe der Typen **CancellationTokenSource** und **CancellationToken** kann ein Thread auf standardisierte Weise gebeten werden, seine Tätigkeit einzustellen. Ein Objekt der Klasse **CancellationTokenSource**

```
var cts = new CancellationTokenSource();
```

stellt über seine **Token**-Eigenschaft eine Instanz vom Strukturtyp **CancellationToken** als „Teilnahmekärtchen“ am Terminierungsverfahren zur Verfügung.

Das kooperative Abbruchmodell wurde primär zur Verwaltung von Aufgaben im Sinne der Task Parallel Library (siehe Abschnitt 17.4) entwickelt, doch lässt sich dieses Muster über die **Thread**-

[1] https://docs.microsoft.com/de-de/dotnet/standard/threading/cancellation-in-managed-threads

Konstruktorüberladung mit einem Parameter vom Delegatentyp **ParameterizedThreadStart** auch bei der Verwaltung von klassischen **Thread**-Objekten verwenden. In der folgenden Anweisung

```
var t = new Thread(new ParameterizedThreadStart(runner.Run));
```

wird ein **Thread** basierend auf einer Methode mit **Object**-Parameter erstellt. An diese Methode wird beim Thread-Start das **CancellationToken** übergeben:

```
t.Start(cts.Token);
```

Die folgende Startmethode des Threads führt eine Schleife aus und prüft bei jedem Durchlauf, ob für das **CancellationToken** die Beendigung angefordert worden ist:

```
public void Run(object obj) {
   var ct = (CancellationToken)obj;
   for (int i = 0; i < 5; i++) {
      Write("Rumoren im Runner, i = " + i);
      Thread.Sleep(1000);
      if (ct.IsCancellationRequested) {
         Console.WriteLine(" Tschüss!");
         break;
      }
   }
}
```

Sobald der Wert **true** für die Eigenschaft **IsCancellationRequested** festgestellt wird, enden die Schleife, die Startmethode und der Thread. Dabei kommt es aber nicht zu einer abgebrochenen Ausgabe, sondern der kooperative Thread macht bei der nächsten passenden Gelegenheit einen geordneten Abgang, z. B.:

```
Rumoren im Runner, i = 0
Rumoren im Runner, i = 1
Rumoren im Runner, i = 2
 Tschüss!
```

Nach Bedarf sollten vor dem Thread-Ende erforderliche Aufräumungsarbeiten ausgeführt werden.

Wenn die Eigenschaft **IsCancellationRequested** einer **CancellationToken**-Instanz den Wert **true** erhalten hat, lässt sich dieser Wert nicht mehr auf **false** zurücksetzen.

Um die Unterbrechungsaufforderung für einen Thread zu setzen, ruft man die **Cancel()** - Methode des **CancellationTokenSource**-Objekts auf, z. B.:

```
cts.Cancel();
```

Ein **CancellationTokenSource**-Objekt verwendet unverwaltete Ressourcen, die auf jeden Fall freigegeben werden sollten. Daher implementiert die Klasse das Interface **IDisposable**, sodass ihre Objekte die Methode **Dispose()** zur Freigabe der Ressourcen beherrschen. Mit Hilfe der **using**-Anweisung kann man auf einfache Weise erreichen, dass die **Dispose()** - Methode unter allen Umständen aufgerufen wird, z. B.:

```
static void Main() {
    using var cts = new CancellationTokenSource();
    var runner = new ThreadCancellationPolling();
    var t = new Thread(new ParameterizedThreadStart(runner.Run));
    t.Start(cts.Token);
    Thread.Sleep(8000);
    cts.Cancel();
}
```

Wenn ein zu unterbrechender Thread eine Schleife mit passender Wiederholfrequenz verwendet, sollte er regelmäßig die **IsCancellationRequested**-Eigenschaft des Tokens überprüfen, was keine nennenswerten Kosten verursacht. Diese periodische Abfrage bezeichnet man als *Polling*.

Das Polling-Verfahren eignet sich nicht zum Unterbrechen eines Threads, der gerade auf ein Synchronisierungsobjekt wartet (vgl. Abschnitt 17.1.4.3). In diesem Fall muss man dafür sorgen, dass der Thread sowohl auf das Signalisierungsobjekt als auch auf eine Unterbrechungsaufforderung achtet.[1] Das lässt sich z. B. über die Klasse **ManualResetEventSlim** realisieren, die seit dem .NET - Framework 4.0 verfügbar ist und das Cancellation Framework unterstützt.

Abschließend sollen die Bestandteile der kooperativen Thread-Terminierung noch einmal im Überblick beschrieben werden:

- Es wird ein Objekt der Klasse **CancellationTokenSource** erstellt, z. B.:
  ```
  var cts = new CancellationTokenSource();
  ```
  Über seine **Token**-Eigenschaft ist eine Instanz vom Strukturtyp **CancellationToken** verfügbar, die an potentiell zu terminierende Threads übergeben wird. Das **CancellationTokenSource**-Objekt beherrscht die Methode **Cancel()** zur Übermittlung der Terminierungsaufforderung.
- Die Startmethode für den potentiell zu unterbrechenden Thread muss den Delegatentyp **ParameterizedThreadStart** erfüllen, z. B.:
  ```
  public void Run(Object obj) {
     var ct = (CancellationToken)obj;
       . . .
  }
    . . .
  var runner = new ThreadCancellationPolling();
  var t = new Thread(new ParameterizedThreadStart(runner.Run));
  ```
- An die **Thread**-Methode **Start()** wird das Token über die **CancellationTokenSource**-Eigenschaft **Token** übergeben, z. B.:
  ```
  t.Start(cts.Token);
  ```
- In den Methoden des potentiell zu unterbrechenden Threads muss ein Verfahren zur Detektion einer Unterbrechungsaufforderung implementiert werden. Eine einfache und typische Lösung besteht darin, die **IsCancellationRequested**-Eigenschaft der erhaltenen **CancellationToken**-Instanz zyklisch abzufragen, z. B.:
  ```
  if (ct.IsCancellationRequested) {
     Console.WriteLine(" Tschüss!");
     break;
  }
  ```
  Stellt ein Thread eine Terminierungsaufforderung fest, sollte er bei der nächsten passenden Gelegenheit notwendige Aufräumungsarbeiten ausführen und dann seine Tätigkeit einstellen.
- Um eine Terminierungsaufforderung an einen Thread zu übermitteln, richtet man einen **Cancel()** - Methodenaufruf an das **CancellationTokenSource**-Objekt, z. B.:
  ```
  cts.Cancel();
  ```

Über weitere Details des kooperativen Terminierungsverfahrens wird im Zusammenhang mit der Task Parallel Library berichtet, für die es primär entwickelt worden ist (siehe Abschnitt 17.4.10).

---

[1] https://docs.microsoft.com/de-de/dotnet/standard/threading/cancellation-in-managed-threads

17.1.6 Thread-Lebensläufe

In diesem Abschnitt wird zunächst die Vergabe von Arbeitsberechtigungen für konkurrierende Threads durch die Laufzeitumgebung behandelt. Dann fassen wir unsere Kenntnisse über die verschiedenen Zustände eines Threads und über Anlässe für Zustandswechsel zusammen.

17.1.6.1 Scheduling und Prioritäten

Die Zuweisung von Rechenzeit auf den CPU-Kernen eines Rechners an die Threads im Zustand **Running** orientiert sich u. a. an den **Prioritäten** der Threads, die sich über ihre **Priority**-Eigenschaft ermitteln und verändern lassen. Erlaubt sind die folgenden Werte des Enumerationstyps **ThreadPriority** (im Namensraum **System.Threading**):

- **Highest**
- **AboveNormal**
- **Normal**
- **BelowNormal**
- **Lowest**

Per Voreinstellung haben Threads die Priorität **Normal**.

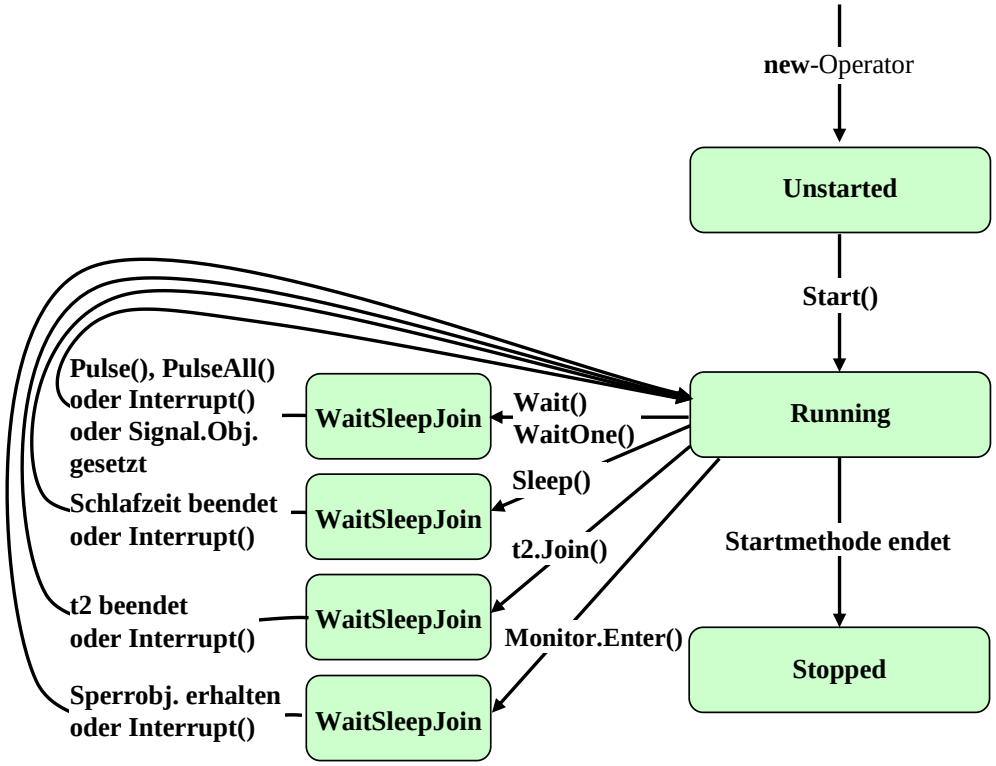
Die Zuweisung von Rechenzeit an arbeitswillige Threads hängt auch vom zugrundeliegenden Betriebssystem ab. Unter Windows gelten die folgenden Regeln:[1]

- Threads mit höherer Priorität werden in einem *strengen* Sinn bevorzugt. Ein Thread kann nur dann Zugang zu einem CPU-Kern erhalten, wenn *kein* Thread mit höherer Priorität arbeitswillig ist. Ein *Verhungern* (engl. *starvation*) von Threads mit niedriger Priorität, die permanent den Kürzeren ziehen, wird also *nicht* verhindert. Daher sollte die Priorität eines Threads nicht leichtfertig erhöht werden, und in einem Thread mit erhöhter Priorität sollten keine zeitaufwändigen Arbeiten ausgeführt werden.
- Bei *gleicher* Priorität kommt ein Zeitscheibenverfahren zum Einsatz. Die Threads gleicher Priorität werden reihum (*Round-Robin*) jeweils für eine festgelegte Zeitspanne ausgeführt (bei einer Desktop-Version von Windows auf einem x64 - Prozessor ca. 30 Millisekunden). Hat ein Thread A sein Zeitquantum verbraucht, dann wird geprüft, ob ein Thread mit höherer Priorität oder ein anderer Thread mit derselben Priorität arbeitswillig ist. Je nach Prüfergebnis wechselt die Rechenerlaubnis, oder der Thread A erhält ein weiteres Zeitquantum.

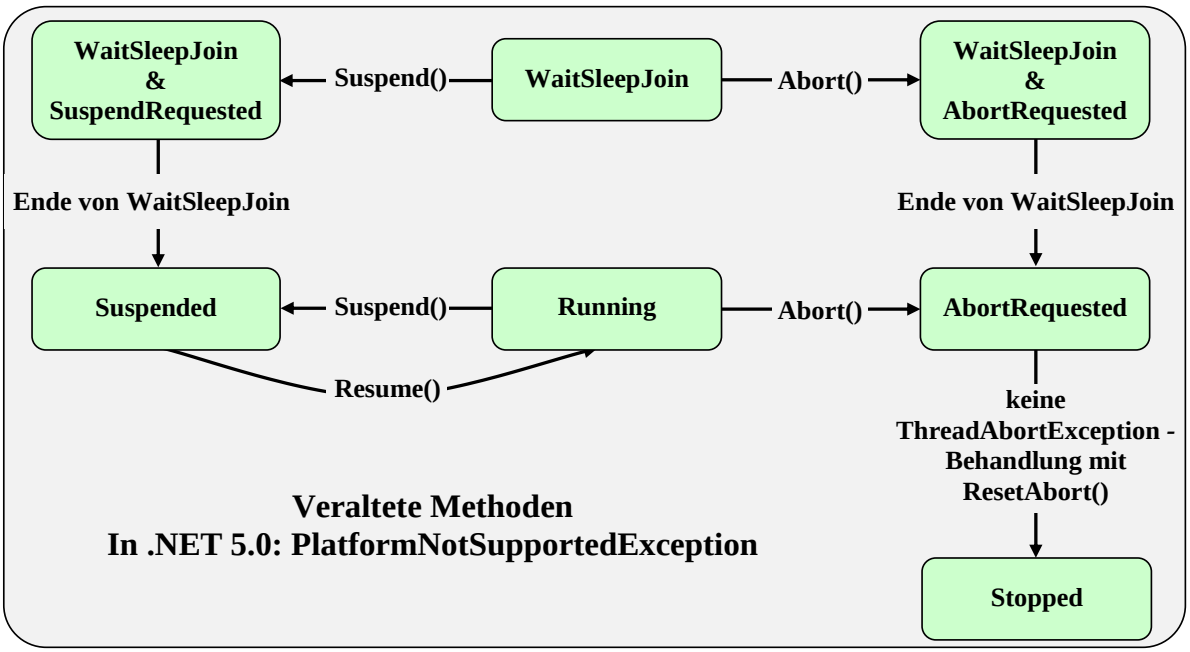
17.1.6.2 Zustände von Threads

In der folgenden Abbildung werden wichtige Thread-Zustände (bezeichnet durch Werte der Enumeration **ThreadState**) und Anlässe für Zustandsübergänge dargestellt:

[1] https://www.microsoftpressstore.com/articles/article.aspx?p=2233328&seqNum=7



Vom Abbrechen (siehe Abschnitt 17.1.5.1) und vom Unterbrechen eines Threads wird schon seit längerer Zeit abgeraten. In .NET 5.0 produzieren die beteiligten Methoden **Abort()**, **Suspend()** und **Resume()** eine **PlatformNotSupportedException**. Die in der folgenden Abbildung beschriebenen Methoden sind allesamt veraltet und in .NET 5.0 nicht mehr nutzbar:



Einige in den Abbildungen enthaltene **Thread**-Instanzmethoden wurden bisher noch nicht behandelt:

- **Interrupt()**
  Diese Instanzmethode dient dazu, einen Thread vom Zustand **WaitSleepJoin** in den Zustand **Running** zu versetzen. Dazu wird dem angesprochenen Thread eine **ThreadInterrupted-Exception** zugeworfen, wenn er sich (jetzt oder später) im **WaitSleepJoin**-Zustand befindet. Ein **Interrupt()** - Aufruf kann also auch prophylaktisch erfolgen. Begibt sich der angesprochene Thread nie in den Zustand **WaitSleepJoin**, bleibt ein **Interrupt()** - Aufruf ohne Folgen. Um auf eine **ThreadInterruptedException** vorbereitet zu sein, sollten die zum Zustand **WaitSleepJoin** führenden Methoden in einer **try-catch** - Anweisung aufgerufen werden, z. B.:
  ```
  try {
     Console.WriteLine("T1 möchte 10 Minuten schlafen.");
     Thread.Sleep(600000);
  } catch {
     Console.WriteLine("T1 beim Schlafen gestört");
  }
  ```
- **Suspend()**, **Resume()**
  Mit diesen als *veraltet* (engl.: *deprecated*) eingestuften Methoden kann man einen Thread anhalten bzw. wieder starten. Die Methoden sind in Misskredit geraten, weil ihr Einsatz zu Deadlock-Situationen (siehe Abschnitt 17.1.7) führen kann. Wenn ein Thread suspendiert wird, der über eine Sperre verfügt, werden alle anderen Threads blockiert, welche dieselbe Sperre beanspruchen. In .NET 5.0 produzieren die Methoden **Suspend()** und **Resume()** eine **PlatformNotSupportedException**.

Weitere Hinweise:

- Im Zustand **Running** ist ein arbeitswilliger Thread auch dann, wenn er gerade auf die Zuteilung eines CPU-Kerns wartet.
- Ein Thread ist auch dann im Zustand **WaitSleepJoin**, wenn er (z. B. nach dem Aufruf der Methode **WaitOne()**) auf ein Signalisierungsobjekt (z. B. aus der Klasse **AutoResetEvent**) wartet (siehe Abschnitt 17.1.4.3).[1]
- Einige Thread-Zustände können *gleichzeitig* bestehen. Erhält z. B. ein Thread im Zustand **WaitSleepJoin** einen **Abort()** - Aufruf, dann bestehen simultan die Zustände **WaitSleepJoin** und **AbortRequested** bis der Thread den Zustand **WaitSleepJoin** verlässt und dann mit der **ThreadAbortException** konfrontiert wird.

17.1.7 Deadlock

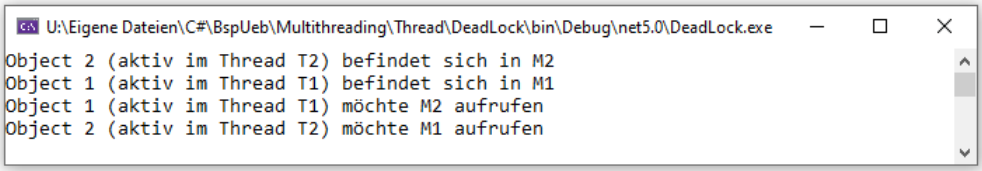
Wer sich beim Einsatz von Sperrobjekten zur Thread-Synchronisation ungeschickt anstellt, kann einen sogenannten *Deadlock* produzieren, wobei sich Threads gegenseitig blockieren. Im folgenden Beispiel begeben sich die Threads mit den Namen `T1` und `T2` jeweils in einen exklusiven Block, der durch die Objekte `lock1` bzw. `lock2` geschützt ist:

```
using System;
using System.Threading;

class MMDeadLock {
   bool warInM1, warInM2;
   static object lock1 = new object();
   static object lock2 = new object();
```

[1] https://docs.microsoft.com/de-de/dotnet/api/system.threading.threadstate

```
    void M1(object nr) {
       int nri = (int)nr;
       lock (lock1) {
          warInM1 = true;
          Console.WriteLine("Object " + nri + " (aktiv im Thread " +
             Thread.CurrentThread.Name + ") befindet sich in M1");
          Thread.Sleep(100);
          if (!warInM2) {
             Console.WriteLine("Object " + nri + " (aktiv im Thread " +
             Thread.CurrentThread.Name + ") möchte M2 aufrufen");
             M2(nr);
          }
       }
    }

    void M2(object nr) {
       int nri = (int)nr;
       lock (lock2) {
          warInM2 = true;
          Console.WriteLine("Object " + nri + " (aktiv im Thread " +
             Thread.CurrentThread.Name + ") befindet sich in M2");
          Thread.Sleep(100);
          if (!warInM1) {
             Console.WriteLine("Object " + nri + " (aktiv im Thread " +
                Thread.CurrentThread.Name + ") möchte M1 aufrufen");
             M1(nr);
          }
       }
    }

    static void Main() {
       MMDeadLock mm12 = new();
       MMDeadLock mm21 = new();
       Thread t1 = new(new ParameterizedThreadStart(mm12.M1));
       Thread t2 = new(new ParameterizedThreadStart(mm21.M2));
       t1.Name = "T1"; t1.Start(1);
       t2.Name = "T2"; t2.Start(2);
    }
}
```

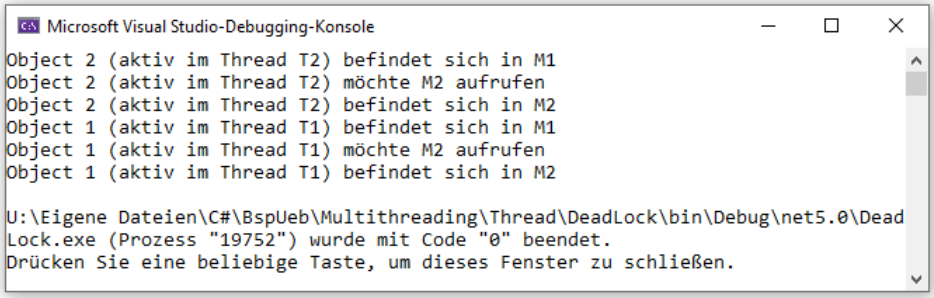
Durch kurze Schläfchen ist dafür gesorgt, dass beide Threads ihren „eigenen“ synchronisierten Block ungestört betreten können. Anschließend versuchen beide, in den jeweils anderen Block zu gelangen, und das Programm hängt fest:



Das Problem hat die folgende Struktur:

- Der Thread `T1` besetzt den vom Objekt `lock1` geschützten Block.
- Der Thread `T2` besetzt den vom Objekt `lock2` geschützten Block.
- Der Thread `T1` möchte den von `lock2` geschützten Block betreten, wartet also darauf, dass Thread `T2` diesen Block verlässt.
- Dies wird Thread `T2` aber nicht tun, bevor er in dem von `lock1` geschützten Block gewesen ist. Der Thread `T2` wartet also darauf, dass Thread `T1` diesen Block verlässt.

Die beiden Threads können übrigens erfolgreich agieren, wenn sie in derselben Reihenfolge vorgehen (z. B. erst `M1()` aufrufen, dann `M2()`), wobei sich ein Thread etwas gedulden muss, z. B.:



17.1.8 Unbehandelte Ausnahmen in sekundären Threads

Als Ergebnis einer unbehandelten Ausnahme wird die Anwendung von der CLR gestoppt. Das gilt für den primären Thread, einen explizit gestarteten sekundären Thread und auch für einen Pool-Thread mit traditioneller Technik (siehe Abschnitt 17.2.1).[1]

Für die bei Verwendung der TPL (*Task Parallel Library*) gestarteten Pool-Threads gilt leider, dass hier auftretende unbehandelte Ausnahmen unbeachtet bleiben können (siehe Abschnitt 17.4.6).

Im .NET Framework vor der Version 2.0 hatte die CLR eine Anwendung nach einer unbehandelten Ausnahme in explizit gestarteten (sekundären) Threads oder in einem Pool-Thread weiterlaufen lassen. Bei einer Konsolenanwendung erschien das Ausnahmeprotokoll mit Typangabe, Message und Stacktrace. Bei einer GUI-Anwendung erfuhr der Anwender nichts von dem Problem.

Dieses als riskant beurteilte Verhalten kann in einem C# - Projekt für die aktuelle (und finale) .NET Framework - Version 4.8 über das folgende Anwendungskonfigurationselement (in der Datei **app.config**) angeordnet werden:

```
<configuration>
  <runtime>
    <legacyUnhandledExceptionPolicy enabled="1" />
  </runtime>
</configuration>
```

In .NET 5.0 kann diese Toleranz gegenüber unbehandelten Ausnahmen *nicht* reaktiviert werden.

Als zentrale Einsprungstelle zur Reaktion auf unbehandelte Ausnahmen, die in einem beliebigen Thread aufgetreten sind, bietet die Klasse **AppDomain** das Ereignis **UnhandledException** an.[2] Das folgende Programm registriert eine Behandlungsmethode für das Ereignis **UnhandledException** und startet dann einen sekundären Thread, der nach kurzer Ruhe eine Ausnahme wirft und unbehandelt lässt:

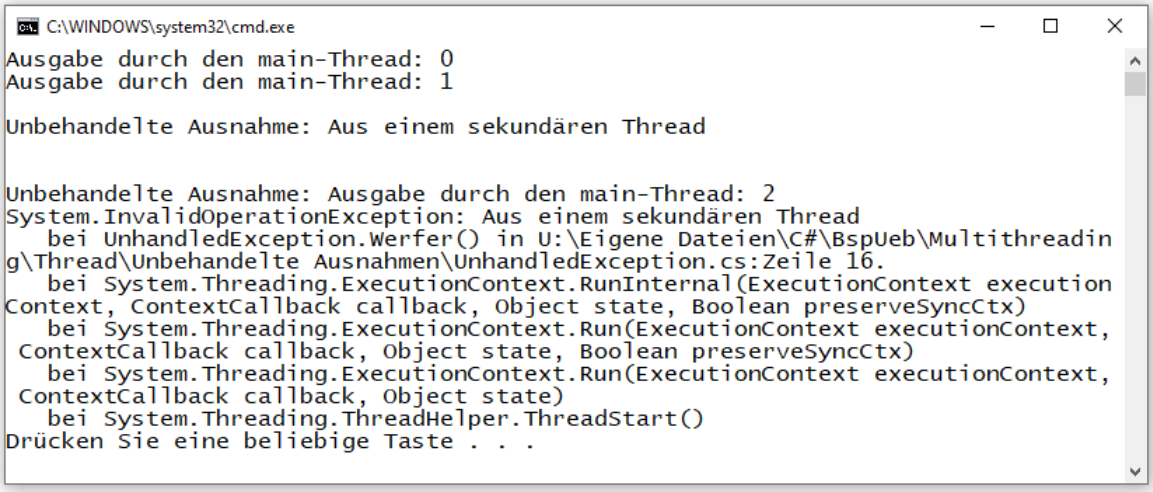
---

[1] Aus der Reihe tanzt nur die **ThreadAbortException**, die auch dann *nicht* zur Beendigung *des Programms* führt, wenn sie unbehandelt bleibt (siehe Abschnitt 17.1.5.1). Allerdings kann dieser Fall in .NET 5.0 nicht auftreten, weil die auslösende Methode **Abort()** in dieser .NET - Implementation nicht mehr erlaubt ist.

[2] Im .NET Framework kann eine Anwendung neben der voreingestellten Anwendungsdomäne noch weitere Domänen besitzen, die im Speicher getrennt sind und separat entladen werden können, um die Stabilität der Anwendung zu erhöhen. Dieses Konzept fehlt in .NET Core (also auch in .NET 5.0). Im Kurs bzw. Manuskript werden Anwendungsdomänen nicht behandelt. Die Klasse **AppDomain** existiert auch in .NET Core, hat aber hier eine eingeschränkte Funktionalität.

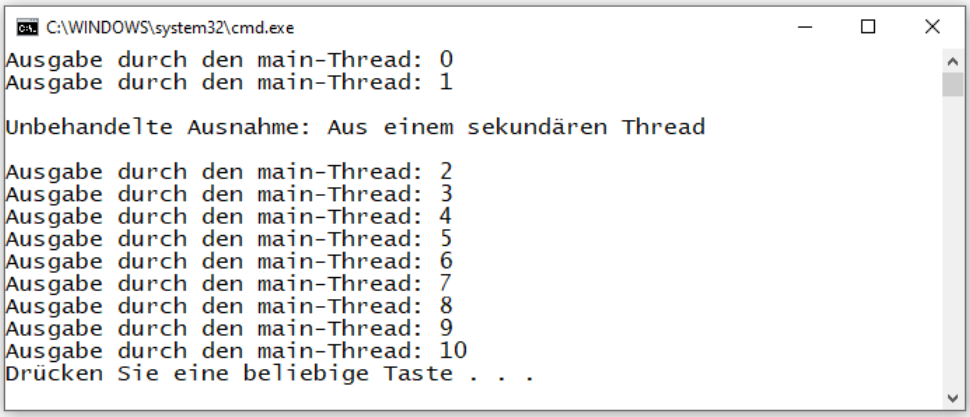
```
using System;
using System.Threading;

class UnhandledException {
   static void Werfer() {
      Thread.Sleep(3000);
      throw new InvalidOperationException("Aus einem sekundären Thread");
   }

   static void UnHandler(object sender, UnhandledExceptionEventArgs e) {
      Console.WriteLine("\nUnbehandelte Ausnahme: " +
                        $"{((Exception) e.ExceptionObject).Message}\n");
      // Protokoll in eine Logdatei schreiben
   }

   public static void Main() {
      AppDomain.CurrentDomain.UnhandledException += UnHandler;
      new Thread(Werfer).Start();
      for(int i = 0; ; i++) {
         Thread.Sleep(1000);
         Console.WriteLine("Ausgabe durch den Main-Thread: "+ i);
      }
   }
}
```

Im .NET Framework wird seit der Version 2.0 per Voreinstellung die **UnhandledException**-Ereignisbehandlungsmethode ausgeführt und anschließend die Anwendung gestoppt. Die Ereignisbehandlungsmethode kann z. B. einen Logdatei-Eintrag schreiben oder eine Datenrettung versuchen, aber das Anwendungsende nicht verhindern, z. B.:



Dasselbe Verhalten zeigt (ohne Änderungsmöglichkeit) auch ein Programm für .NET 5.0.

Durch das oben beschriebene Element **legacyUnhandledExceptionPolicy** in der Anwendungskonfigurationsdatei **app.config** kann man im .NET Framework 4.8 hingegen eine Ausnahmebehandlung *ohne* Anwendungsstopp erzwingen, z. B.:



17.2 Threadpool

Durch eine große Zahl von Threads mit kurzer Lebensdauer wird eine Anwendung eher ausgebremst als beschleunigt. Statt für viele einzelne Aufgaben jeweils einen neuen Thread zeitaufwändig zu erzeugen und anschließend wieder zu zerstören, sollte der vom .NET-Laufzeitsystem verwaltete Pool von Arbeits- und Ein-/Ausgabe - Threads genutzt werden. Ankommende Aufträge gelangen in eine Warteschlange und werden vom nächsten freien Thread aus dem „Bereitschaftsteam“ übernommen.

Als Vorteile von Pool-Threads im Vergleich zu selbst erstellten Threads sind zu nennen:

- Die Leistung einer Anwendung kann gesteigert werden.
- Indem der Programmierer von der Thread-Verwaltung entlastet wird, sinken Aufwand und Fehlerrisiko.

In einem Threadpool befinden sich nur *Hintergrund*-Threads, die nach dem Ende des letzten Vordergrund-Threads von der CLR automatisch gestoppt werden. Vor dem Beenden des letzten Vordergrund-Threads muss man also eventuell prüfen, ob ein Pool-Thread noch Aufgaben ausführt, die nicht abgebrochen werden dürfen.

Anstelle eines Threadpool - Auftrags ist in einigen Situationen jedoch ein explizit erstellter Thread erforderlich bzw. empfehlenswert, z. B.:[1]

- Es wird ein *Vordergrund*-Thread benötigt.
- Es wird eine spezielle Thread-Priorität bevorzugt.
  Alle Pool-Threads haben eine *normale* Priorität (siehe Abschnitt 17.1.6.1).
- Es ist mit einer langen Bearbeitungsdauer zu rechnen.
  In dieser Situation sollte der traditionelle Thread-Pool (siehe Abschnitt 17.2.1) nicht verwendet werden. Statt gem. Abschnitt 17.1 einen eigenen Thread zu starten, kann man aber eine Aufgabe mit der **TaskFactory** - Methode **StartNew()** erstellen und mit dem Parameterwert **TaskCreationOptions.LongRunning** dafür sorgen, dass die Anzahl der Pool-Threads wegen des Langläufers erhöht wird (siehe Abschnitt 17.2.2).

Der Threadpool passt seine „Personalstärke“ dynamisch an den Bedarf an und bevorzugt dabei die Anzahl der verfügbaren (logischen) Prozessorkerne, sofern so viele Threads ausgelastet werden können. Wenn allerdings Threads auf das Dateisystem oder auf das Netzwerk warten müssen, also blockiert sind, dann nimmt die CLR zusätzliche Threads in Betrieb (Griffiths 2013, S. 609). Selbstverständlich wird die Anzahl der Pool-Threads auch nach unten an die Anzahl zu erledigender Aufträge angepasst, um keine Ressourcen zu verschwenden. Es ist nur selten sinnvoll, mit den stati-

[1] https://docs.microsoft.com/en-us/dotnet/standard/threading/the-managed-Threadpool

schen Methoden **SetMaxThreads()** bzw. **SetMinThreads()** der Klasse **ThreadPool** (im Namensraum System.Threading) die Entscheidung der CLR über die angemessene Zahl von Pool-Threads zu beeinflussen.[1]

In jedem Prozess existiert genau ein Threadpool.

Im Threadpool sind Spezialisten für zwei Aufgabenbereiche tätig:

- **Worker Threads**
  Die mit Hilfe der Klassen **ThreadPool** (siehe Abschnitt 17.2.1) oder **Task** (siehe Abschnitt 17.2.2) übergebenen Delegatenobjekte werden durch Worker Threads abgearbeitet.[2]
- **I/O Completion Port Threads** (IOCP-Threads)
  Diese Threads kommen im Hintergrund zum Einsatz, um in Zusammenarbeit mit dem Betriebssystem und mit den zuständigen Gerätetreibern Ein- bzw. Ausgaben bei Datei- oder Netzwerkzugriffen asynchron abzuwickeln. Allerdings müssen die IOCP-Threads nur Erfolgsmeldungen verarbeiten, sodass auch bei einer großen Zahl von Ein- bzw. Ausgaben nur wenige IOCP-Threads benötigt werden.[3]

Im .NET Framework ab Version 4.0 und in .NET Core (inklusive .NET 5.0) wird die Nutzung des Threadpools über die Task Parallel Library (TPL) erheblich vereinfacht. Wir beschäftigen uns im Abschnitt 17.2.1 kurz mit der noch in vielen Programmen anzutreffenden traditionellen Threadpool-Technik und dann im Abschnitt 17.2.2 ausführlich mit der Pool-Nutzung im Rahmen der Task Parallel Library.

17.2.1 Traditionelle Threadpool-Technik

Bei der traditionellen Threadpool-Technik ist nur *eine* Warteschlange vorhanden, und die Arbeitsaufträge werden in der Reihenfolge ihres Eingangs abgearbeitet (FIFO, First-In-First-Out). Wenn viele Aufträge anfallen und gleichzeitig mehrere (logische) CPU-Kerne verfügbar sind, dann kann das Leistungspotential der Hardware durch die traditionelle Threadpool-Technik nicht optimal ausgeschöpft werden. Folglich sollte die traditionelle Threadpool-Technik nicht mehr benutzt werden. Sie wird trotzdem anschließend beschrieben, weil entsprechender Code noch oft anzutreffen ist, der verstanden, gepflegt und eventuell modernisiert werden muss.

Soll eine Methode im Hintergrund durch den traditionell verwalteten Threadpool ausgeführt werden, dann muss sie dem Delegatentyp **WaitCallback**

public delegate void WaitCallback(Object *state***)**

entsprechen, z. B.

```
static void HintergrundAktion(object anzahl) {
   Random zzg = new Random();
   int anz = (int)anzahl;
   double summe;
   for (int i = 0; i < anz; i++) {
      summe = 0.0;
      for (int j = 0; j < 10_000_000; j++)
         summe += zzg.Next(100);
      Console.WriteLine("Aktuelle Zufallssumme: " + summe);
   }
}
```

---

1 https://docs.microsoft.com/en-us/dotnet/api/system.threading.threadpool.setminthreads

2 Worker Threads sind uns schon im Abschnitt 17.1.2 (unter der Bezeichnung *Arbeits-Threads*) begegnet.

3 https://blog.stephencleary.com/2013/11/there-is-no-thread.html

Diese Methode berechnet eine wählbare Anzahl von Summen, jeweils ermittelt aus reichlich vielen Zufallszahlen, die von einem **Random**-Objekt geliefert werden.

Um einen **WaitCallback**-Arbeitsauftrag zur traditionellen Verarbeitung an den Threadpool zu übergeben, verwendet man die statische Methode **QueueUserWorkItem()** der Klasse **ThreadPool** in einer von den folgenden Überladungen:

public static bool QueueUserWorkItem(WaitCallback *callBack*)
public static bool QueueUserWorkItem(WaitCallback *callBack*, Object *state*)

Der *state*-Parameter der zweiten Überladung ist für Daten vorgesehen, die der **WaitCallback**-Methode beim Aufruf übergeben werden sollen. Im Beispiel lässt sich so festlegen, wie viele Zufallssummen berechnet werden sollen:

```
ThreadPool.QueueUserWorkItem(HintergrundAktion, 5);
```

Das kooperative Terminierungsverfahren lässt sich auch bei der traditionellen Threadpool-Nutzung einsetzen (siehe Abschnitt 17.1.5.2).

17.2.2 Threadpool 4.0

Mit dem Erscheinen von .NET Framework 4.0 wurde die *Task Parallel Library* (TPL) eingeführt, um die parallele Programmierung zu reformieren (siehe Abschnitt 17.4). Zur Unterstützung der TPL wurde gleichzeitig der Threadpool überarbeitet mit dem Ziel, die in Mehrkernprozessoren für jeden Kern vorhandenen lokalen Zwischenspeicher optimal zu nutzen und Zugriffe auf den langsameren Hauptspeicher zu reduzieren (Griffiths 2013, S. 607f):

- Für jeden (logischen) Prozessorkern wird eine separate Warteschlange verwendet, damit ein Auftrag möglichst vom selben Kern erledigt wird, der ihn erstellt hat. So besteht eine hohe Wahrscheinlichkeit, dass sich benötigte Daten im lokalen Cache des Kerns befinden.
- Die einem (logischen) Prozessorkern zugeordneten Aufgaben werden nach dem LIFO-Prinzip (Last-In-First-Out) erledigt, um den Durchsatz zu erhöhen. Bei den zuletzt eingelieferten Arbeitsaufträgen ist nämlich die Chance am größten, dass sich relevante Daten noch im lokalen Cache des Prozessorkerns befinden.
- Bei Beschäftigungsmangel bedient sich ein Prozessorkern aus den Warteschlangen der Kollegen, wobei ältere Aufträge bevorzugt werden, weil sich die von ihnen benötigten Daten mit geringerer Wahrscheinlichkeit im lokalen Cache des jeweiligen Prozessorkerns befinden. In der englischsprachigen Literatur spricht man vom *work stealing*.

Mit der im Abschnitt 17.2.1 beschriebenen **ThreadPool**-Methode **QueueUserWorkItem()** lassen sich die Threadpool-Optimierungen *nicht* nutzen. Sie sollte daher ab .NET Framework 4.0 (insbesondere auch in .NET 5.0) durch die gleich vorzustellenden Methoden unter Verwendung der Klasse **Task** ersetzt werden. Wir verzichten vorläufig auf eine flexible Aufgabenverwaltung und verwenden den leistungsoptimierten Threadpool nur für Aufgaben nach dem Motto „Fire & Forget“.

Zur Übergabe eines Arbeitsauftrags verwenden wir die Methode **StartNew()** der Klasse **TaskFactory**, die sich (wie auch die Klasse **Task**) im Namensraum **System.Threading.Tasks** befindet. Wir beschränken uns auf die folgende Überladung:

public Task StartNew(Action<Object> *action*, **Object** *state*)

Als erster Parameter ist ein Objekt vom Delegatentyp **Action<Object>** zu übergeben:

public delegate void Action<Object>(Object *state*)

Offenbar verlangt dieser Delegatentyp von einer Methode nichts anderes als der Delegatentyp **WaitCallback**, den die Methode **QueueUserWorkItem()** als Datentyp für ihren ersten Parameter verwendet (siehe Abschnitt 17.2.1).

Das per *state*-Parameter an **StartNew()** übergebene Objekt wird beim Aufruf der Methode **Action<Object>()** weitergereicht.

Über die statische Eigenschaft **Factory** der Klasse **Task** ist ein einsatzfähiges Objekt der Klasse **TaskFactory** ansprechbar, z. B.:

```
Task.Factory.StartNew(G1Task, i);
```

Es folgt ein Beispielprogramm, das die zeitökonomische Auftragsbearbeitung durch den modernen Threadpool demonstriert, aber keine verwertbare Leistung erbringt. Die Methode `G1Task()` erfüllt den Delegatentyp **Action<Object>**, kann also als erster Parameter an **StartNew()** übergeben werden:

```
static long start;
 . . .
static void G1Task(object nr) {
   for (int i = 0; i < 4; i++) {
      int[] ids = new int[2];
      ids[0] = (int)nr;
      ids[1] = i;
      Task.Factory.StartNew(G2Task, ids);
   }
   long dauer = DateTime.Now.Ticks - start;
   Console.WriteLine($"G1Task {nr}, Zeit seit Start: {(dauer/1.0e7),6:f4} Sek., " +
                     $"ThreadId: {Thread.CurrentThread.ManagedThreadId}");
}
```

Im Rahmen der Auftragsbearbeitung startet `G1Task()` vier Unteraufträge unter Verwendung der Methode `G2Task()`, die ebenfalls den Delegatentyp **Action<Object>** erfüllt. An `G2Task()` wird als *state*-Parameter ein **int**-Array mit den Nummern für den Hauptauftrag (`ids[0]`) und den Unterauftrag (`ids[1]`) übergeben:

```
Task.Factory.StartNew(G2Task, ids);
```

Am Ende eines `G1Task()` - Aufrufs wird die Zeitdifferenz seit dem Programmstart und die Kennung des ausführenden Pool-Threads ausgegeben.

In `G2Task()` wird Rechenzeit verbraucht durch das Aufsummieren der Quadratwurzeln von ziemlich vielen Zufallszahlen. Abschließend werden die Zeitdifferenz seit dem Programmstart und die Kennung des ausführenden Pool-Threads ausgegeben:

```
static void G2Task(object state) {
   Random zzg = new Random();
   int[] ids = (int[])state;
   double summe = 0.0;
   for (int i = 0; i < 10_000_000; i++)
      summe += Math.Sqrt(zzg.NextDouble());
   long dauer = DateTime.Now.Ticks - start;
   Console.WriteLine($"G2Task ({ids[0]},{ids[1]})" +
           $", Zeit seit Start: {(dauer / 1.0e7),6:f4} Sek., " +
           $"ThreadId: {Thread.CurrentThread.ManagedThreadId}");
}
```

Im der **Main()** - Methode des folgenden Programms werden vier `G1Task()` - basierte Aufträge gestartet:

```
using System;
using System.Threading;
using System.Threading.Tasks;

class TaskPool {
   static long start;

   static void G1Task(Object nr) {
       . . .
   }

   static void G2Task(Object state) {
       . . .
   }

   static void Main() {
      start = DateTime.Now.Ticks;
      const int anz = 4;
      for (int i = 0; i < anz; i++) {
         Task.Factory.StartNew(G1Task, i);
      }
      Console.ReadLine(); // Hält den Vordergrund-Thread aktiv
   }
}
```

Die auf eine per Enter abgeschickte Zeile lauernde Methode **ReadLine()** verhindert das Ende der Methode **Main()** und damit das Ende des Haupt-Threads und das Ende des Programms. Die im Threadpool tätigen Hintergrund-Threads könnten das Programm nicht am Leben erhalten.

Auf einem Rechner mit vier logischen Prozessorkernen hat sich das folgende Ablaufprotokoll ergeben:

```
G1Task 0, Zeit seit Start: 0,0167 Sek., ThreadId: 4
G1Task 1, Zeit seit Start: 0,0166 Sek., ThreadId: 5
G1Task 2, Zeit seit Start: 0,0166 Sek., ThreadId: 6
G1Task 3, Zeit seit Start: 0,0167 Sek., ThreadId: 7
G2Task (2,3), Zeit seit Start: 0,2995 Sek., ThreadId: 6
G2Task (3,3), Zeit seit Start: 0,3028 Sek., ThreadId: 7
G2Task (1,3), Zeit seit Start: 0,3095 Sek., ThreadId: 5
G2Task (0,3), Zeit seit Start: 0,3099 Sek., ThreadId: 4
G2Task (0,2), Zeit seit Start: 0,5851 Sek., ThreadId: 4
G2Task (1,2), Zeit seit Start: 0,5984 Sek., ThreadId: 5
G2Task (2,2), Zeit seit Start: 0,6068 Sek., ThreadId: 6
G2Task (3,2), Zeit seit Start: 0,6113 Sek., ThreadId: 7
G2Task (0,1), Zeit seit Start: 0,8498 Sek., ThreadId: 4
G2Task (2,1), Zeit seit Start: 0,8565 Sek., ThreadId: 6
G2Task (1,1), Zeit seit Start: 0,8609 Sek., ThreadId: 5
G2Task (3,1), Zeit seit Start: 0,8638 Sek., ThreadId: 7
G2Task (0,0), Zeit seit Start: 1,1173 Sek., ThreadId: 4
G2Task (3,0), Zeit seit Start: 1,1226 Sek., ThreadId: 7
G2Task (2,0), Zeit seit Start: 1,1229 Sek., ThreadId: 6
G2Task (1,0), Zeit seit Start: 1,1287 Sek., ThreadId: 5
```

Die vier Aufgabenpakete (jeweils einer `G1Task` zugehörig) werden von den vorhandenen vier logischen Prozessorkernen quasi-parallel ausgeführt. In den Warteschlangen, die fest mit einem Thread und vermutlich auch mit einem Prozessorkern assoziiert sind, werden die „lokalen“ Aufträge nach dem LIFO-Prinzip abgearbeitet:

Queue zur ThreadId 4	Queue zur ThreadId 5	Queue zur ThreadId 6	Queue zur ThreadId 7
(0, 0)	(1, 0)	(2, 0)	(3, 0)
(0, 1)	(1, 1)	(2, 1)	(3, 1)
(0, 2)	(1, 2)	(2, 2)	(3, 2)
(0, 3)	(1, 3)	(2, 3)	(3, 3)

Um das Beispielprogramm auf die Nutzung der traditionellen Threadpool-Technik umzustellen, sind die **StartNew()** - Aufrufe

```
Task.Factory.StartNew(G1Task, i);
 . . .
Task.Factory.StartNew(G2Task, ids);
```

zu ersetzen durch **QueueUserWorkItem()** - Aufrufe (siehe Abschnitt 17.2.1):

```
ThreadPool.QueueUserWorkItem(G1Task, i);
. . .
ThreadPool.QueueUserWorkItem(G2Task, ids);
```

Auf einem Rechner mit vier logischen Prozessorkernen hat sich das folgende Ablaufprotokoll ergeben:

```
G1Task 0, Zeit seit Start: 0,0167 Sek., ThreadId: 4
G1Task 1, Zeit seit Start: 0,0167 Sek., ThreadId: 5
G1Task 2, Zeit seit Start: 0,0167 Sek., ThreadId: 6
G1Task 3, Zeit seit Start: 0,0168 Sek., ThreadId: 7
G2Task (0,1), Zeit seit Start: 0,2976 Sek., ThreadId: 5
G2Task (0,2), Zeit seit Start: 0,2986 Sek., ThreadId: 6
G2Task (0,3), Zeit seit Start: 0,3032 Sek., ThreadId: 7
G2Task (0,0), Zeit seit Start: 0,3051 Sek., ThreadId: 4
G2Task (2,0), Zeit seit Start: 0,5727 Sek., ThreadId: 5
G2Task (2,1), Zeit seit Start: 0,5805 Sek., ThreadId: 6
G2Task (2,2), Zeit seit Start: 0,5816 Sek., ThreadId: 7
G2Task (2,3), Zeit seit Start: 0,5963 Sek., ThreadId: 4
G2Task (1,1), Zeit seit Start: 0,8371 Sek., ThreadId: 6
G2Task (1,2), Zeit seit Start: 0,8377 Sek., ThreadId: 7
G2Task (1,0), Zeit seit Start: 0,8379 Sek., ThreadId: 5
G2Task (1,3), Zeit seit Start: 0,8520 Sek., ThreadId: 4
G2Task (3,0), Zeit seit Start: 1,0944 Sek., ThreadId: 7
G2Task (3,1), Zeit seit Start: 1,0952 Sek., ThreadId: 5
G2Task (3,2), Zeit seit Start: 1,1018 Sek., ThreadId: 6
G2Task (3,3), Zeit seit Start: 1,1043 Sek., ThreadId: 4
```

Die Aufgaben stehen in *einer* Warteschlange und werden nach dem FIFO-Prinzip abgearbeitet. Bei der Verteilung auf die vier Threads im Pool wird die Herkunft der Unteraufträge *nicht* beachtet:

Reihenfolge Fertigstellung	Task	ThreadId
1	(0, 1)	5
2	(0, 2)	6
3	(0, 3)	7
4	(0, 0)	4
5	(2, 0)	5
6	(2, 1)	6
7	(2, 2)	7
8	(2, 3)	4
9	(1, 1)	6
10	(1, 2)	7
11	(1, 0)	5
12	(1, 3)	4
13	(3, 0)	7
14	(3, 1)	5
15	(3, 2)	6
16	(3, 3)	4

Weil die vier logischen Prozessorkerne unterschiedlich leistungsfähig bzw. verfügbar sind, sind die Aufgaben nicht in der perfekten FIFO-Reihenfolge erledigt.

Im Beispiel führt die verbesserte Threadpool-Verwaltung im Rahmen der TPL *nicht* zu einer Leistungssteigerung. Das sollte aber niemanden davon abhalten, in der Regel der folgenden, von Microsofts Experten zur parallelen Programmierung mit .NET ausgesprochenen Empfehlung zu folgen:[1]

> Task is now the preferred way to queue work to the thread pool.

Die im Beispiel verwendete **StartNew()** - Überladung

> **public Task StartNew(Action<Object>** *action***, Object** *state***)**

verwendet für den ersten Parameter den konkretisierten generischen Delegatentyp **Action<Object>**. Eine realisierende Methode muss mit dem Parametertyp **Object** arbeiten, was eine explizite Typumwandlung erzwingt und keine perfekte Typüberwachung durch den Compiler erlaubt, z. B.:

```
static void G2Task(object state) {
   Random zzg = new Random();
   int[] ids = (int[])state;
    . . .
}
```

Nach einem Vorschlag von Griffiths (2013, S. 607) lässt sich der Parametertyp **Object** folgendermaßen vermeiden:

- Man verwendet die **StartNew()** - Überladung mit einem einzigen Parameter vom Delegatentyp **Action**:
  public Task StartNew(Action *action***)**
- Eine realisierende Methode muss den Delegatentyp **Action** erfüllen:
  public delegate void Action()

[1] https://devblogs.microsoft.com/pfxteam/choosing-between-the-task-parallel-library-and-the-threadpool/

- Man verwendet ein per Lambda-Notation definiertes Delegatenobjekt (vgl. Abschnitt 10.1.6), das in seinem Ausdruck bzw. Anweisungsblock den eigentlichen Arbeitsauftrag enthält, z. B.:
  ```
  Task.Factory.StartNew(() => G2Task(ids));
  ```
  Von großem Nutzen ist dabei die Möglichkeit, im Ausdruck bzw. Anweisungsblock eines per Lambda-Notation realisierten Delegatenobjekts auf lokale Variablen der umgebenden Methode zuzugreifen.
- Die als Task auszuführende Methode kann den gewünschten Parametertyp verwenden, z. B.:
  ```
  static void G2Task(int[] ids) {
      . . .
  }
  ```

Bei einigen **StartNew()** - Überladungen kann man das Verhalten des TPL-Aufgabenplaners (siehe Abschnitt 17.4.7) durch einen Parameter vom Enumerationstyp **TaskCreationOptions** beeinflussen, z. B.:

public Task StartNew(Action<Object> *action***, Object** *state***, TaskCreationOptions** *options***)**

Mit dem häufig verwendeten Wert **TaskCreationOptions.LongRunning** kündigt man einen Langläufer an und ermuntert den Aufgabenplaner, die Anzahl der Pool-Threads zu erhöhen.

Seit dem .NET Framework 4.5 beherrscht die Klasse **Task** die statische Methode **Run()** in diversen Überladungen, z. B.:

public static Task Run(Action *action***)**

Damit kann man sich den „Umweg“ über ein Objekt der Klasse **Factory** sparen. Wie Sie bereits wissen, ist es trotz des parameterfreien Delegatentyps **Action** mit Hilfe der Lambda-Notation möglich, an eine per Pool-Thread auszuführende Methode Daten zu übergeben:

```
Task.Run(() => G2Task(ids));
```

Leider hat der optimierte Threadpool den Nachteil, dass die in einem Pool-Thread aufgetretenen und nicht behandelten Ausnahmen unbeachtet bleiben können (siehe Abschnitt 17.4.6).

17.2.3 Threadpool-Nutzung in einer WPF-Anwendung

Um eine konsistente Bedienoberfläche zu garantieren, verwendet das WPF-Framework eine **Single-Thread - Architektur**. Auf ein UI-Element darf nur derjenige Thread zugreifen, der das Element erzeugt hat. Andererseits müssen zeitaufwändige Arbeiten aus dem UI-Thread herausgehalten werden, um die Reaktionsfähigkeit der Bedienoberfläche sicherzustellen. Wenn nun z. B. ein Pool-Thread eine Aufgabe erledigt hat, dann muss eine Möglichkeit geschaffen werden, das Ergebnis unter Beachtung der Single-Thread - Regel auf der Bedienoberfläche anzuzeigen. Genau dies ermöglicht die Klasse **SynchronizationContext** aus dem Namensraum **System.Threading**:

- In einer WPF-Ereignisbehandlungsmethode erhält man über die statische **SynchronizationContext**-Eigenschaft **Current** ein **SynchronizationContext**-Objekt, das mit dem UI-Thread assoziiert ist. Wir werden später noch wiederholt vom *UI-Synchronisierungskontext* oder kurz vom *UI-Kontext*) sprechen (siehe z. B. Abschnitt 17.4.7).
- Durch einen Aufruf der **SynchronizationContext**-Methode **Post()** mit einem Delegatenobjekt vom Typ **SendOrPostCallback** als Aktualparameter sorgt man dafür, dass das Delegatenobjekt (genauer: die realisierende Methode) im UI-Thread ausgeführt wird.

Wir nutzen den Threadpool unter Verwendung der im Abschnitt 17.2.2 vorgeschlagenen TPL-basierten Technik, um eine Schwäche des RSS-Feed - Readers, den wir im Abschnitt 6.8 erstellt haben, zu beheben.[1] Während eine angeforderte Feed-Datei aus dem Internet geladen wird, können viele Sekunden vergehen, und in dieser Zeit reagiert der RSS-Feed - Reader aus dem Abschnitt 6.8 nicht auf Benutzerwünsche, sodass z. B. das Anwendungsfenster nicht verschoben werden kann. Zur Beseitigung des Problems sind lediglich einige kleine Änderungen in der **Click**-Behandlungsmethode zum Befehlsschalter der Anwendung vorzunehmen:

```
private void button_Click(object sender, RoutedEventArgs e) {
   SynchronizationContext contextUI = SynchronizationContext.Current;

   var waitInfo = new List<RssItem>()
                  { new RssItem() { Title = " Feed wird geladen. Bitte warten ..." } };
   listBox.ItemsSource = waitInfo;
   button.IsEnabled = false;

   String url = textBox.Text;
   Task.Factory.StartNew(() => {
      try {
         XDocument feed = XDocument.Load(url);
         var items = new List<RssItem>();
         IEnumerable<XElement> xDocItems = feed.Descendants("item");
         RssItem rit;
         foreach (XElement item in xDocItems) {
            rit = new RssItem() {
               Title = item.Element("title").Value,
               Description = Regex.Replace(item.Element("description").Value, "<[^>]*>",
                                           String.Empty, RegexOptions.IgnoreCase).Trim(),
               Url = item.Element("link").Value
            };
            items.Add(rit);
         }
         contextUI.Post(notUsed => {listBox.ItemsSource = items;}, null);
      } catch (Exception ex) {
         contextUI.Post(notUsed => {listBox.ItemsSource = null;
                                    MessageBox.Show(this, ex.ToString(), ex.Message);}, null);
      } finally {
         contextUI.Post(notUsed => {button.IsEnabled = true;}, null);
      }
   });
}
```

Weil die Methode `button_Click()` im UI-Thread ausgeführt wird, resultiert aus der folgenden Anweisung ein **SynchronizationContext**-Objekt, das mit dem UI-Thread assoziiert ist:

```
SynchronizationContext contextUI = SynchronizationContext.Current;
```

Die **try**-Anweisung, welche einen Internet-Zugriff und die bei einem umfangreichen Feed eventuell zeitaufwändige Aufbereitung der Daten enthält, soll vom Threadpool ausgeführt werden. Daher erstellen wir per Lambda-Notation ein Delegatenobjekt, das den Delegatentyp **Action** realisiert, mit der **try**-Anweisung im Rumpf. Dieses Delegatenobjekt wird an die **Task.Factory** - Methode **StartNew()** als Aktualparameter übergeben:

[1] Im Abschnitt 17.5 wird eine noch modernere und bequemere Lösung vorgestellt für das Problem, eine zeitaufwändige Aufgabe per Threadpool zu erledigen und anschließend die Bedienoberfläche zu aktualisieren (asynchrone Programmierung mit den Schlüsselwörtern **async** und **await**).

```
Task.Factory.StartNew(() => {
   try {
      . . .
   } catch (Exception ex) {
      . . .
   } finally {
      . . .
   }
});
```

Weil die **try**-Anweisung in einem Pool-Thread ausgeführt wird, darf hier kein (lesender oder schreibender) Zugriff auf den UI-Thread stattfinden. Daher wird die **Text**-Eigenschaft der **TextBox** noch im UI-Thread extrahiert.

Um das fertig aufbereitete **List<`RssItem`>** - Objekt als Wert der **ListBox**-Eigenschaft **ItemsSource** festzulegen, wird die **Post()** - Methode des mit dem UI-Thread assoziierten **SynchronizationContext**-Objekts aufgerufen mit einem per Lambda-Notation realisierten **SendOrPostCallback**-Delegaten

public delegate void SendOrPostCallback(object *state***)**

als erstem Parameter:

```
contextUI.Post(notUsed => {listBox.ItemsSource = items;}, null);
```

Der zweite **Post()** - Parameter dient zur Übergabe von Daten an den Delegaten (siehe Definition des Delegatentyps **SendOrPostCallback**). Weil dieser Parameter im Beispiel ungenutzt bleibt, geben wir als Wert das Referenzliteral **null** an.

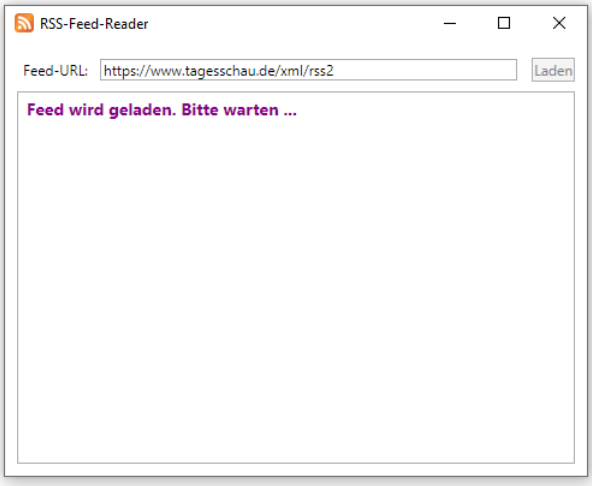
Auch die im **catch**- und im **finally**-Block befindlichen GUI-relevanten Anweisungen müssen an den UI-Thread delegiert werden. U. a. wird der am Anfang der Ereignisbehandlung aus naheliegenden Gründen deaktivierte Befehlsschalter im **finally**-Block wieder nutzbar gemacht:

```
contextUI.Post(notUsed => {button.IsEnabled = true;}, null);
```

Damit der Benutzer während der Wartezeit darüber informiert ist, was das Programm gerade tut, erhält die **ListBox** einen provisorischen Inhalt

```
var waitInfo = new List<RssItem>()
                  { new RssItem() { Title = " Feed wird geladen. Bitte warten ..." } };
listBox.ItemsSource = waitInfo;
button.IsEnabled = false;
```

mit Erläuterung:



Den früher während des Ladens angezeigten Wait-Cursor brauchen wir nicht mehr.

17.3 Timer

Wenn eine Aufgabe in festen Zeitabständen wiederholt ausgeführt werden muss (z. B. Aktualisierung einer Zeitanzeige in einem WPF-Programm, Überprüfung der Funktionstüchtigkeit eines Servers), dann beauftragt man ein Objekt aus einer Timer-Klasse der BCL, eine Methode mit der entsprechenden Funktionalität regelmäßig aufzurufen. Anschließend werden drei Timer-Klassen vorgestellt:

- In den Namensräumen **System.Threading** und **System.Timers** ist jeweils eine Klasse namens **Timer** vorhanden. Beide Klassen sind geeignet, wenn die regelmäßig auszuführende Aktion aufwändig ist und in einem Pool-Thread ausgeführt werden soll.
- Die Klasse **DispatcherTimer** im Namensraum **System.Windows.Threading** ist im Rahmen von WPF-Anwendungen besonders bequem einsetzbar. Allerdings laufen die regelmäßig auszuführenden Methoden im UI-Thread ab, sodass bei hohem Zeitbedarf eine unergonomisch zäh reagierende Bedienoberfläche resultieren kann.

Die Präzision der Timer-Lösungen in .NET - Anwendungen hängt vom zugrunde liegenden Betriebssystem ab und bewegt sich in der Regel im Bereich von 10-20 Millisekunden, was in den meisten Fällen genügt. Durch eine direkte Verwendung von Diensten des Betriebssystems lässt sich unter Windows die Ungenauigkeit auf ca. 1 Millisekunde reduzieren (Albahari & Johannsen 2020, S. 920).

17.3.1 Multithreading-Timer

Den beiden in diesem Abschnitt beschriebenen **Timer**-Klassen ist gemeinsam, dass die regelmäßig auszuführende Arbeit von einem Pool-Thread erledigt wird.

Es werden unverwaltete Ressourcen belegt, um die regelmäßige Ausführung zu ermöglichen. Die beiden Klassen implementieren das Interface **IDisposable**, sodass ihre Objekte die Methode **Dispose()** zur Freigabe der Ressourcen beherrschen. Diese Methode sollte möglichst früh aufgerufen werden, z. B. implizit mit Hilfe der **using**-Anweisung.

Mit Hilfe der angenehm einfach konstruierten Klasse **Timer** im Namensraum **System.Threading** kann man die CLR beauftragen, eine Methode regelmäßig aufzurufen. Im folgenden Beispiel

```
tim = new Timer(new TimerCallback(HintergrundAktion), null, 0, 10000);
```

kommt eine Konstruktor-Überladung

public Timer(TimerCallback *callback***, Object** *state***, long** *dueTime***, long** *period***)**

mit den folgenden Parameter-Datentypen zum Einsatz:

- **TimerCallback** *callback*
  Dieser Delegatentyp verlangt für die regelmäßig auszuführende Methode die folgende Bauart:

  public delegate void TimerCallback(Object *state***)**
- **Object** *state*
  Die regelmäßig auszuführende Methode erhält beim Aufruf das im zweiten Konstruktorparameter anzugebende Objekt, sodass ihr Verhalten gesteuert werden kann. Ist (wie im Beispiel) kein Parameter erforderlich, übergibt man eine **null**-Referenz.
- **long** *dueTime*
  Mit diesem Parameter wird die Zeit bis zum *ersten* Aufruf in Millisekunden festgelegt.

- **long** *period*
  Durch diesen Parameter wird die Zeit zwischen zwei Aufrufen in Millisekunden festgelegt. Soll es bei *einem* Aufruf bleiben, versorgt man den Parameter *period* mit dem Wert **Timeout.Infinite**.

Bei aufwändigen Arbeiten kann es in Abhängigkeit von der gewählten Wartezeit zwischen zwei Aufrufen dazu kommen, dass die **TimerCallback**-Methode von mehreren Threads simultan ausgeführt wird. Wenn die Methode gemeinsame Daten (z. B. Instanz- oder Klassenvariablen) verändert, kommt eine Thread-Synchronisation in Frage (siehe Abschnitt 17.1.4). Eine ständig wachsende Zahl von simultanen Ausführungen der **TimerCallback**-Methode muss natürlich verhindert werden. Neben der Wahl eines passenden *period*-Konstruktorparameters besteht auch die Möglichkeit, für ein bereits aktives **Timer**-Objekt mit der Methode **Change()** die Intervalldauer zu verändern, z. B.:

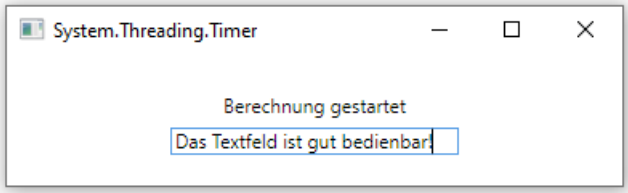
```
tim.Change(0, 2000);
```

Wir stellen uns die Aufgabe, in einer GUI-Anwendung eine umfangreiche Aktivität regelmäßig per Threadpool im Hintergrund erledigen zu lassen, wobei die Benutzeroberfläche verzögerungsfrei bedienbar bleiben soll. Im folgenden Programm (eine handgestrickte WPF-Anwendung ohne XAML, vgl. Abschnitt 12.2.1) berechnet die Methode `HintergrundAktion` alle 10 Sekunden die Summe aus 200 Millionen Zufallszahlen:

```
using System;
using System.Windows;
using System.Threading;
using System.Windows.Controls;

class ThreadingTimer : Window {
   Label anzeige;
   Timer tim;
   Random zzg = new Random();

   ThreadingTimer() {
      Height = 120; Width = 400;
      Title = "System.Threading.Timer";

      StackPanel lm = new StackPanel();
      lm.VerticalAlignment = VerticalAlignment.Center;
      Content = lm;

      anzeige = new Label();
      anzeige.HorizontalAlignment = System.Windows.HorizontalAlignment.Center;
      lm.Children.Add(anzeige);

      tim = new Timer(new TimerCallback(HintergrundAktion), null, 0, 10000);

      TextBox eingabe = new TextBox();
      eingabe.Width = 180;
      lm.Children.Add(eingabe);
   }

   void Melde (String s) {
      Action action = delegate () {anzeige.Content = s;};
      anzeige.Dispatcher.BeginInvoke(action);
   }

   void HintergrundAktion(Object info) {
      double zs = 0;
      Melde("Berechnung gestartet");
      for (int i = 0; i < 200_000_000; i++)
         zs += zzg.Next(100);
      Melde("Aktuelle Zufallssumme: " + zs.ToString() +
            " berechnet um " + DateTime.Now.ToLongTimeString());
   }
```

```
   [STAThreadAttribute]
   static void Main() {
      Application app = new();
      app.Run(new ThreadingTimer());
   }
}
```

Trotz des erheblichen CPU-Zeitverbrauchs durch die Hintergrundaktivität reagiert die Benutzeroberfläche des Programms jederzeit verzögerungsfrei:



In einer WPF-Anwendung müssen Zugriffe auf Steuerelemente demjenigen Thread vorbehalten bleiben, der sie erzeugt hat. Zugriffe durch fremde Threads werden von der CLR verhindert. Im Abschnitt 17.2.3 wurde die **SynchronizationContext**-Methode **Post()** dazu verwendet, um aus einem Hintergrund-Thread heraus die Bedienoberfläche zu aktualisieren. Im aktuellen Beispiel wird ein alternatives Verfahren verwendet, weil an das **SynchronizationContext**-Objekt zum UI-Thread nicht ganz leicht heranzukommen ist. Der UI-Thread kann über die **Invoke()** - Methode oder die **BeginInvoke()** - Methode des zuständigen **Dispatcher**-Objekts veranlasst werden, eine Delegatenmethode auszuführen. An dieses Objekt kommt man über die **Dispatcher**-Eigenschaft eines Steuerelements heran.

Beim **Invoke()** - Aufruf wartet der Hintergrund-Thread, bis der UI-Thread die Aktualisierungsmethode beendet hat. Beim **BeginInvoke()** - Aufruf wartet der Hintergrund-Thread *nicht* auf das Ende der Aktualisierung durch den UI-Thread. In der Regel ist die asynchrone Variante **BeginInvoke()** zu bevorzugen. In unserem Beispiel wird so für das möglichst frühzeitige Ende der **Timer-Callback**-Methode `HintergrundAktion()` gesorgt, die einen Pool-Thread belegt.[1]

Wir verwenden die folgende **BeginInvoke()** - Überladung:

public DispatcherOperation BeginInvoke(Delegate *method***, params Object[]** *args***)**

Zum Formalparameter mit dem abstrakten Typ **Delegate** liefern wir ein Objekt vom Delegatentyp **Action**, das durch eine anonyme Methode realisiert wird (vgl. Abschnitt 10.1.5):

```
void Melde (String s) {
   Action action = delegate () {anzeige.Content = s;};
   anzeige.Dispatcher.BeginInvoke(action);
}
```

Im zweiten **BeginInvoke()** - Parameter gibt man eine passende Zahl von Argumenten für die Delegatenmethode an, in unserem Fall also keines (siehe Abschnitt 5.3.1.3.3 zu Serienparametern).

Um die delegierte Änderung der Bedienoberfläche zu realisieren, muss in der von einem Pool-Thread ausgeführten Methode `Melde()` auf die **Dispatcher**-Eigenschaft des **Label**-Steuerelements zugegriffen werden, wobei es sich um einen Zugriff auf ein **DispatcherObject** handelt (vgl. Abschnitt 12.2.2). In der folgenden, oft zitierten Aussage aus der BCL-Dokumentation sollte der Klarheit halber zwischen erlaubten und verbotenen Zugriffen differenziert werden:[2]

---

[1] Während generell zu einem **BeginInvoke()** - Aufruf an ein *Delegatenobjekt* auch der zugehörige **EndInvoke()** - Aufruf erfolgen sollte, ist im vorliegenden Fall *kein* **EndInvoke()** - Aufruf erforderlich (siehe Abschnitt 17.6.1.1).

[2] https://docs.microsoft.com/en-us/dotnet/api/system.windows.threading.dispatcher.begininvoke

In WPF, only the thread that created a **DispatcherObject** may access that object.

Während dem Hintergrund-Thread der unvermeidliche Zugriff auf die **Dispatcher**-Eigenschaft des **Label**-Objekts erlaubt ist, führt der Zugriff auf die **Content**-Eigenschaft zum erwarteten Ausnahmefehler:

```
void Melde (String s) {
    anzeige.Content = "Nicht möglich";
    Action action = delegate () {anzeige.Content = s;};
    anzeige.Dispatcher.BeginInvoke(acti
}

void HintergrundAktion(Object info) {
    double zs = 0;
    Melde("Berechnung gestartet");
    for (int i = 0; i < 200_000_000; i+
        zs += zzg.Next(100);
    Melde("Aktuelle Zufallssumme: " + zs.ToString() +
          " berechnet um " + DateTime.Now.ToLongTimeString());
}
```

Vom Benutzer nicht behandelte Ausnahme

System.InvalidOperationException: "Der aufrufende Thread kann nicht auf dieses Objekt zugreifen, da sich das Objekt im Besitz eines anderen Threads befindet."

Details anzeigen | Details kopieren

Ausnahmeeinstellungen

Auf der Klasse **Timer** im Namensraum **System.Threading** baut eine namensgleiche Klasse im Namensraum **System.Timers** auf (Albahari 2014). Hier sind Zusatzleistungen vorhanden, die z. B. die situationsangepasste Steuerung erleichtern:

- Die Klasse bietet ein Ereignis namens **Elapsed** an, bei dem regelmäßig auszuführende Methoden registriert werden können. Damit ist es möglich, die Liste der regelmäßig auszuführenden Methoden im laufenden Betrieb zu verändern.
- Der Timer lässt sich flexibel (de)aktivieren über ...
  - die Eigenschaft **Enabled** (Voreinstellung **false**)
  - oder über die Methoden **Start()** und **Stop()**.

17.3.2 DispatcherTimer

Im Namensraum **System.Windows.Threading** ist eine Klasse namens **DispatcherTimer** vorhanden, die ereignisorientiert arbeitet und bequem in eine GUI-Anwendung einzubinden ist, z. B.:

```
using System;
using System.Windows;
using System.Windows.Threading;
using System.Windows.Controls;

class WPFTimer : Window {
   Label anzeige;
   DispatcherTimer tim;
   Random zzg = new Random();

   WPFTimer() {
      Height = 120; Width = 400;
      Title = "DispatcherTimer";

      var lm = new StackPanel();
      lm.VerticalAlignment = VerticalAlignment.Center;
      Content = lm;

      anzeige = new Label();
      anzeige.HorizontalAlignment = System.Windows.HorizontalAlignment.Center;
      lm.Children.Add(anzeige);

      tim = new DispatcherTimer();
      tim.Tick += new EventHandler(TimerAktion);
      tim.Interval = new TimeSpan(0, 0, 10);
      tim.Start();
```

```
      TextBox eingabe = new TextBox();
      eingabe.Width = 180;
      lm.Children.Add(eingabe);
   }

   void TimerAktion(Object myObject, EventArgs ea) {
      double zs = 0;
      anzeige.Content = "Berechnung gestartet"; // Bleibt ohne Effekt!
      for (int i = 0; i < 200_000_000; i++)
         zs += zzg.Next(100);
      anzeige.Content = "Aktuelle Zufallssumme: " + zs + " berechnet um " +
                        DateTime.Now.ToLongTimeString();
   }

   [STAThreadAttribute]
   static void Main() {
      new Application().Run(new WPFTimer());
   }
}
```

Diesmal laufen die vom Timer angestoßenen Methodenaufrufe jedoch im UI-Thread ab, und bei hohem Zeitaufwand resultiert eine zähe Bedienoberfläche.

Das aktuelle Beispiel ist dem Vorbild aus dem Abschnitt 17.3.1 nachempfunden und lässt ebenfalls alle 10 Sekunden die Summe aus 200 Millionen Zufallszahlen berechnen. Das hat aber nun zur Folge, dass nach jedem Tick-Ereignis einige Sekunden lang keine Eingaben in das Textfeld möglich sind.

Außerdem ist zu beachten, dass die Anzeige des Fensters nicht aktualisiert werden kann, solange eine **DispatcherTimer**-Ereignismethode läuft. Im Beispiel ist es daher *nicht* möglich, die **Label**-Anzeige „unterwegs“ zu aktualisieren:

```
anzeige.Content = "Berechnung gestartet"; // Bleibt ohne Effekt!
```

Bei der im Abschnitt 17.3.1 vorgestellten Lösung mit dem **Timer** aus dem Namensraum **System.Threading** war das möglich.

Für das regelmäßige Starten von Methoden mit geringem Rechenzeitbedarf ist die Klasse **DispatcherTimer** durchaus geeignet, weil beim Zugriff auf UI-Elemente keine Thread-Grenzen zu überwinden sind.

17.4 Aufgabenbasierte asynchrone Programmierung (Task Parallel Library)

Die mit dem .NET Framework 4.0 eingeführte *Task Parallel Library* (TPL) mit etlichen Typen im Namensraum **System.Threading.Tasks** wurde zur performanten Nutzung von Multi-Core - CPUs bei der parallelen bzw. asynchronen Programmierung konzipiert. Microsoft spricht bei der TPL-Nutzung auch vom *Task-based Asynchronous Pattern* (TAP). Für jede asynchron auszuführende Aufgabe wird ein Objekt aus der Klasse **Task** oder aus ihrer generischen Spezialisierung **Task<TResult>** erstellt. Die Aufgaben werden meist von Pool-Threads erledigt, wobei die ebenfalls mit dem .NET Framework 4.0 eingeführten Optimierungen des Threadpools für eine flotte, gut skalierende Abwicklung sorgen (siehe Abschnitt 17.2). Im Rahmen einer GUI-Anwendung muss gelegentlich dafür gesorgt werden, dass eine Aufgabe im UI-Thread ausgeführt wird.

Neben der Leistungsoptimierung bietet die TPL eine gute Kontrolle über die **Task** - bzw. **Task<TResult>** - Objekte (z. B. Warten auf Beendigung, Fortsetzungsaufgaben, zusammengesetzte Aufgaben, Abbrechen, Ausnahmebehandlung), sodass **Task** - bzw. **Task<TResult>** - Objekte in vielen Situationen gegenüber der direkten Verwendung von **Thread**-Objekten (siehe Abschnitt 17.1) und auch gegenüber einer Verwendung des traditionellen Threadpools (siehe Abschnitt 17.2.1) zu bevorzugen sind. Im Abschnitt 17.2.2 über den beim Erscheinen von .NET Framework

4.0 renovierten Threadpool haben wir bereits mit **Task** - Objekten gearbeitet. Allerdings wurden die Aufgaben nach dem Motto „Fire & Forget“ verwendet, sodass die nun vorzustellenden Optionen zur Aufgabenverwaltung keine Rolle spielten.

Microsoft empfiehlt, in Anwendungen für eine Zielplattform ab .NET Framework 4.0 (also insbesondere in Anwendungen für .NET Core und für .NET ab Version 5.0) die nebenläufige Programmierung möglichst per TPL zu realisieren.[1]

It's the recommended approach to asynchronous programming in .NET.

17.4.1 Aufgaben ohne bzw. mit Rückgabe erstellen

Für eine Aufgabe ohne bzw. mit Rückgabe verwendet man ein Objekt aus der Klasse **Task** bzw. aus der generischen Spezialisierung **Task<TResult>**. Es sprechen einige Argumente dagegen, eigene Spezialisierungen dieser Klassen zu definieren, u. a.:[2]

- Die im Abschnitt 17.4.1.3 beschriebenen Fabrikmethoden können nur **Task**- bzw. **Task<TResult>** - Objekte erstellen.
- Die im Abschnitt 17.4.8 beschriebenen Methoden der Aufgabenfortsetzung können nur **Task**- bzw. **Task<TResult>** - Objekte erstellen.

Um die Klasse **Task** bzw. **Task<TResult>** auch ohne Vererbungstechnik mit zusätzlichen Verhaltenskompetenzen auszustatten, kommen Erweiterungsmethoden in Frage (siehe Abschnitt 7.13).

17.4.1.1 Erstellung per Konstruktor

17.4.1.1.1 Die Klasse Task für Aufgaben ohne Rückgabe

Wir beschränken uns zunächst auf asynchron auszuführende Methoden, die keinen Parameter besitzen und keine Rückgabe liefern, also den folgenden Delegatentyp **Action** (im Namensraum **System**) erfüllen:

public delegate void Action()

Der einfachste Konstruktor der Klasse **Task** nimmt ein Parameterobjekt vom Typ **Action** entgegen:

public Task(Action *action***)**

An alternative Konstruktor-Überladungen kann man zur Steuerung der Aufgabenabwicklung übergeben:

- Eine Instanz der Struktur **CancellationToken**, um das kooperative Terminieren der Aufgabe zu ermöglichen (siehe Abschnitt 17.4.10)
- Eine Instanz der Enumeration **TaskCreationOptions**, um die Ausführung der Aufgabe zu beeinflussen (siehe Abschnitt 17.4.1.2)

Für asynchron auszuführende Methoden, die den Delegatentyp **Action<Object>** erfüllen und über einen Parameter vom Typ **Object** mit Daten versorgt werden können, bietet die Klasse **Task** passende Konstruktor-Überladungen an (siehe Abschnitt 17.4.3).

---

1 https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/

2 https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-based-asynchronous-programming

Im folgenden Programm entsteht aus der Methode `SampleMean()`, die den Mittelwert aus einer Anzahl von gleichverteilten Zufallszahlen aus dem Intervall [0; 1) auf der Konsole ausgibt und keine Rückgabe liefert, ein Objekt der Klasse **Task**:

```
using System;
using System.Threading.Tasks;

class TaskDemo {
   const int sampleSize = 100_000_000;

   static void SampleMean() {
      var zzg = new Random();
      double erg = 0.0;
      for (int j = 0; j < sampleSize; j++)
         erg += zzg.NextDouble();
      Console.WriteLine("Stichprobenmittel " + (erg / sampleSize));
   }

   static void Main() {
      var task = new Task(SampleMean);
      task.Start();
      Console.WriteLine("Task gestartet");
      // Parallele Arbeiten erledigen
      task.Wait();
   }
}
```

Das **Task**-Objekt wird per Konstruktor erstellt und anschließend gestartet:

```
var task = new Task(SampleMean);
task.Start();
```

Die Aufgabe wird von einem Pool-Thread ausgeführt, der als Hintergrund-Thread die Anwendung nicht am Leben erhalten kann. In der **Main()** - Methode wird daher durch die später vorzustellende **Task**-Methode **Wait()** dafür gesorgt, dass der Haupt-Thread auf die Fertigstellung der Aufgabe wartet:

```
task.Wait();
```

Anderenfalls würden mit der Methode **Main()** der Haupt-Thread und damit das Programm sehr schnell enden, und die von einem Hintergrund-Thread auszuführende Aufgabe könnte nicht fertiggestellt werden.

Damit ein Nutzen durch Parallelität entsteht, muss natürlich der Haupt-Thread während der Hintergrundbearbeitung selbst tätig werden, statt nur abzuwarten.

17.4.1.1.2 Die Klasse Task<TResult> für Aufgaben mit einer Rückgabe

Soll eine Aufgabe bzw. die zugrundeliegende Methode einen Rückgabewert liefern, erzeugt man ein Objekt der generischen Klasse **Task<TResult>**, die von der Klasse **Task** abstammt und für den Rückgabetyp einen Typparameter besitzt. Im folgenden Programm

```
using System;
using System.Threading.Tasks;

class TaskResultDemo {
   const int sampleSize = 100_000_000;

   static double SampleMean() {
      var zzg = new Random();
      double erg = 0.0;
      for (int j = 0; j < sampleSize; j++)
         erg += zzg.NextDouble();
      return erg / sampleSize;
   }

   static void Main() {
      var task = new Task<double>(SampleMean);
      task.Start();
      Console.WriteLine("Task gestartet");
      // Parallele Arbeiten erledigen
      Console.WriteLine("Stichprobenmittel = " + task.Result);
   }
}
```

wird aus der Methode `SampleMean()`, die den Delegatentyp **Func<out TResult>**

public delegate TResult Func<out TResult>()

mit **double** als **TResult**-Konkretisierung erfüllt und für eine Stichprobe mit gleichverteilten Zufallszahlen aus dem Intervall [0; 1) den Mittelwert berechnet, ein Objekt der Klasse **Task<double>** erstellt und gestartet:

```
var task = new Task<double>(SampleMean);
task.Start();
```

Wie die Klasse **Task** besitzt auch ihre generische Ableitung **Task<TResult>** etliche Konstruktor-Überladungen zur Steuerung der Aufgabenabwicklung. Mögliche Parameter sind:

- Eine Instanz der Struktur **CancellationToken**, um das kooperative Terminieren der Aufgabe zu ermöglichen (siehe Abschnitt 17.4.10)
- Eine Instanz der Enumeration **TaskCreationOptions**, um die Ausführung der Aufgabe zu beeinflussen (siehe Abschnitt 17.4.1.2)

Für asynchron auszuführende Methoden mit Rückgabe, die den Delegatentyp **Func<Object, TResult>** erfüllen und über einen Parameter vom Typ **Object** mit Daten versorgt werden können, bietet die Klasse **Task<TResult>** passende Konstruktor-Überladungen an (siehe Abschnitt 17.4.3).

Task<TResult> - Objekte machen ihr Ergebnis über die Eigenschaft **Result** verfügbar, z. B.:

```
Console.WriteLine("Stichprobenmittel = " + task.Result);
```

Ein Zugriff auf die **Result**-Eigenschaft einer noch nicht abgeschlossenen Aufgabe blockiert den anfragenden Thread. Er sollte sich so lange mit anderen Arbeiten beschäftigen, bis er das Ergebnis der Aufgabenbearbeitung benötigt, und sich eventuell über die (nicht blockierende) **Task**-Eigenschaft **IsCompleted** über die Fertigstellung der Aufgabe informieren (siehe Abschnitt 17.4.4).

Ist bei der asynchronen Aufgabenbearbeitung eine unbehandelte Ausnahme aufgetreten, so wird diese beim **Result**-Zugriff neu geworfen (siehe Abschnitt 17.4.6).

Ein Objekt der Klasse **Task<TResult>** wird in der englischsprachigen Literatur gelegentlich als *future* bezeichnet, weil es ein Ergebnis repräsentiert, das im weiteren Verlauf eventuell verfügbar wird (Albahari & Johannsen 2020, S. 595).

17.4.1.2 Erstellungsoptionen

Bei einigen Konstruktor- bzw. Fabrikmethodenüberladungen zur Erstellung einer Aufgabe wird ein Parameter vom Typ **TaskCreationOptions** akzeptiert, mit dem sich einige Verhaltensmerkmale der erstellten Aufgabe beeinflussen lassen. Die Enumeration **TaskCreationOptions** besitzt u. a. die folgenden Werte

- **PreferFairness**
  Dieser Wert signalisiert, dass für die Aufgabe eine faire Auftragsbearbeitung nach dem FIFO-Prinzip (First-In-First-Out) bevorzugt wird (Voreinstellung: LIFO).
- **LongRunning**
  Mit diesem Wert kündigt man einen Langläufer an und ermuntert den TPL-Aufgabenplaner (siehe Abschnitt 17.4.7), die Anzahl der Pool-Threads zu erhöhen.
- **AttachedToParent**
  Wenn der in einer primären Aufgabe ausgeführte Code weitere (sekundäre) Aufgaben erstellt und dabei die Option **AttachedToParent** verwendet, dann endet die primäre Aufgabe erst dann, wenn alle sekundären Aufgaben abgeschlossen sind. Per Voreinstellung sind die sekundären Aufgaben (in Englisch oft als *child tasks* bezeichnet) abgetrennt (engl.: *detached*), sodass die Fertigstellung der primären Aufgabe *nicht* von der Fertigstellung der sekundären Aufgaben abhängig ist.

Weil an die Enumeration **TaskCreationOptions** das **FlagsAttribute** angeheftet ist (vgl. Abschnitt 14.5.2), sind ihre Werte kombinierbar, z. B.:

```
var meep = new Task(LongRunner,
               TaskCreationOptions.LongRunning | TaskCreationOptions.PreferFairness);
meep.Start();
```

17.4.1.3 Erstellung durch Fabrikmethoden

In diesem Abschnitt werden zwei, schon im Abschnitt 17.2.2 verwendete Fabrikmethoden behandelt, die das Erstellen und Starten einer Aufgabe in *einem* Aufruf ermöglichen:

- Die Instanzmethode **StartNew()** der Klasse **TaskFactory** erlaubt in diversen Überladungen eine detaillierte Konfiguration.
- Die statische Methode **Run()** der Klasse **Task** empfiehlt sich als vereinfachte Variante von **StartNew()** für Standardaufgaben.

Beide Methoden besitzen Überladungen, welche ...

- die Klassen **Task** bzw. **Task<TResult>** unterstützen,
- eine Instanz der Struktur **CancellationToken** entgegennehmen, um das Abbrechen der Aufgabe zu ermöglichen (siehe Abschnitt 17.4.10).

Statt ein **Task**-Objekt per Konstruktor zu erstellen und anschließend mit der **Start()** - Methode zu aktivieren,

```
Task task = new Task(SampleMean);
task.Start();
```

kann man die **StartNew()** - Methode der Klasse **TaskFactory** verwenden, um Kreation und Start in einem Aufruf zu erledigen. Ein einsatzbereites Objekt der Klasse **TaskFactory** ist über die statische **Task**-Eigenschaft **Factory** ansprechbar, z. B.:

```
Task task = Task.Factory.StartNew(SampleMean);
```

Im Unterschied zur gleich zu beschreibenden Methode **Run()** besitzt die Methode **StartNew()** Überladungen, welche die folgenden Argumente entgegennehmen:

- Eine Instanz der Enumeration **TaskCreationOptions**, um das Verhalten der erstellten Aufgabe zu beeinflussen (siehe Abschnitt 17.4.1.1)
- Ein **Object** mit Daten, die der Aufgabe zur Verfügung stehen sollen (siehe Abschnitt 17.4.3)
- Ein Objekt der Klasse **TaskScheduler**, das die Vergabe der erstellten Aufgabe an einen Thread regeln soll

Damit bietet **StartNew()** sogar mehr Steuerungsmöglichkeiten als die Konstruktoren der Klassen **Task** bzw. **Task<TResult>** und sollte in der Regel gegenüber der Sequenz aus einem Konstruktor- und einem **Start()** - Aufruf bevorzugt werden.[1]

Statt das über die statische **Task**-Eigenschaft **Factory** ansprechbare **TaskFactory**-Objekt zu verwenden, kann man eine eigene **TaskFactory** per Konstruktor erstellen und dabei Optionen festlegen, die bei jeder späteren **Task**-Produktion durch diese Fabrik verwendet werden sollen (**CancellationToken**, **TaskCreationOptions**, **TaskContinuationOptions**, **TaskScheduler**).[2]

Seit dem .NET Framework 4.5 ist die statische **Task**-Methode **Run()** verfügbar, die ebenfalls das Erstellen und Starten einer Aufgabe in *einem* Aufruf ermöglicht, z. B.:

```
Task task = Task.Run(SampleMean);
```

Es ist auch eine generische Methode **Run<TResult>()** vorhanden, die ein **Task<TResult>** - Objekt erstellt und startet, z. B.:

```
Task<double> task = Task.Run<double>(SampleMean);
```

An ein **Task**- oder **Task<TResult>** - Objekt gelangt man auch durch den Aufruf von etlichen BCL-Methoden, z. B. aus der Klasse **Stream** (vgl. Abschnitt 17.5.9):

- **public Task WriteAsync(byte[]** *buffer***, int** *offset***, int** *count***)**
  Vom angesprochenen **Stream**-Objekt wird ein **byte**-Array im Rahmen einer asynchron ausgeführten Aufgabe geschrieben. Die Methode **WriteAsync()** kehrt sofort zurück, blockiert also nicht den aufrufenden Thread.
- **public Task<int> ReadAsync(byte[]** *buffer***, int** *offset***, int** *count***)**
  Das angesprochene **Stream**-Objekt liest asynchron Daten in einen **byte**-Array. Vom zurückgelieferten **Task<int>** - Objekt ist die Zahl der tatsächlich gelesenen Bytes über seine **Result**-Eigenschaft zu erfahren.

17.4.2 Sekundäre Aufgaben (child tasks)

Wird im Code einer primären Aufgabe eine neue sekundäre Aufgabe erstellt, so stehen die beiden Aufgaben per Voreinstellung in keiner besonderen Beziehung zueinander.[3] Mit einigen Überladungen des **Task**-Konstruktors oder der **TaskFactory**-Methode **StartNew()** lässt sich aber eine sekundäre Aufgabe (engl.: *child task*) mit einem Abhängigkeitsverhältnis erstellen, indem der Parameter vom Enumerationstyp **TaskCreationOptions** einen Wert mit **AttachedToParent**-Flag erhält.[4] Das hat für die primäre Aufgabe die folgenden Konsequenzen:

---

1 https://docs.microsoft.com/de-de/dotnet/api/system.threading.tasks.taskfactory-1.startnew

2 https://docs.microsoft.com/de-de/dotnet/standard/parallel-programming/task-based-asynchronous-programming

3 Seit der .NET Framework - Version 4.0 wird eine sekundäre Aufgabe allerdings bevorzugt vom selben logischen Prozessorkern abgearbeitet wie die initiierende primäre Aufgabe (siehe Abschnitt 17.2.2).

4 Der Enumerationstyp **TaskCreationOptions** ist durch das **FlagsAttribute** dekoriert, sodass ein Wert für mehrere binäre Einzelattribute steht (siehe Abschnitt 14.5.2).

- Die primäre Aufgabe wird nur dann als abgeschlossen betrachtet, wenn alle sekundären Aufgaben abgeschlossen sind.
- Die beim Zugriff auf die primäre Aufgabe (per **Wait()** - Methode oder **Result**-Eigenschaft, siehe Abschnitt 17.4.6.1) nach dem Auftreten eines Ausnahmefehlers geworfene **AggregateException** enthält ggf. auch alle in sekundären Aufgaben aufgetretenen unbehandelten Ausnahmen (siehe Abschnitt 17.4.6.1.2).

Natürlich kann eine sekundäre Aufgabe ebenfalls neue Aufgaben erstellen. Primär bzw. sekundär ist eine Aufgabe also nicht in einem absoluten Sinn, sondern in Relation zu einer anderen Aufgabe.

17.4.3 Versorgung von Aufgaben mit Daten

Soll eine Aufgabe bzw. die zugrundeliegende Methode in Abhängigkeit von einem Parameter tätig werden bzw. Daten verwenden, dann ist ein alternativer Delegatentyp zu verwenden. Bei einer Aufgabe *ohne* Rückgabe ist dies:

public delegate void Action<Object>(Object *arg***)**

Bei einer parametrisierten Aufgabe *mit* Rückgabe ist der folgende Delegatentyp zu verwenden:

public delegate TResult Func<Object, TResult>(Object *arg***)**

Der erste Typformalparameter von **Action<T>** bzw. **Func<T, TResult>** muss durch **Object** konkretisiert werden.

Eine mit Daten zu versorgende Aufgabe ohne bzw. mit Rückgabe kann analog zu einer Aufgabe ohne Parameter auf unterschiedliche Weise gestartet werden (siehe Abschnitt 17.4.1). Für eine parameterabhängige Aufgabe *ohne* Rückgabe eignet sich z. B. der folgende Konstruktor der Klasse **Task**:

public Task(Action<Object> *action***, Object** *state***)**

Für eine parameterabhängige Aufgabe *mit* Rückgabe eignet sich z. B. der folgende Konstruktor der Klasse **Task<TResult>**:[1]

public Task(Func<Object, TResult> *method***, Object** *arg***)**

Alternativ zum **Task<TResult>** - Konstruktor kann bei einer Aufgabe mit Eingabedaten und Rückgabe die folgende Überladung der **TaskFactory**-Methode **StartNew<TResult>()** verwendet werden:

public Task<TResult> StartNew<TResult>(Func<Object, TResult> *method***, Object** *arg***)**

Die im Abschnitt 17.4.1 als Beispiel verwendete Methode `SampleMean()` zur Berechnung des Mittelwerts aus einer Serie von Zufallszahlen verwendet für den Pseudozufallszahlengenerator den voreingestellten, aus der Systemzeit abgeleiteten Startwert. Bei der folgenden `SampleMean()` - Variante wird ein per Parameter übergebener Startwert benutzt:

```
static double SampleMean(object seed) {
   var zzg = new Random((int)seed);
   double erg = 0.0;
   for (int j = 0; j < sampleSize; j++)
      erg += zzg.NextDouble();
   return erg / sampleSize;
}
```

Sie erfüllt den Delegatentyp

---

[1] Wer im Kopf des **Task<TResult>** - Konstruktors hinter dem Klassennamen den Typformalparameter vermisst, sei daran erinnert, dass bei generischen Typen in einer Konstruktordefinition generell so verfahren wird (siehe Abschnitt 8.2.1).

public delegate double Func<Object, double>(Object *arg***)**

und kann daher als Basis für ein parameterabhängiges **Task<double>** - Objekt verwendet werden, z. B.:

```
Task<double> task = Task.Factory.StartNew<double>(SampleMean, 13);
```

Das an eine Aufgabe übergebenen Daten-**Object** kann später über die **Task**-Eigenschaft **AsyncState** angesprochen werden, z. B.:

```
static void Main() {
    Task<double> task = Task.Factory.StartNew<double>(SampleMean, 13);
    Console.WriteLine("Task gestartet");
    // Parallele Arbeiten erledigen
    Console.WriteLine("Stichprobenmittel = " + task.Result);
    Console.WriteLine("Pseudozufall-Startwert = " + task.AsyncState);
}
```

17.4.4 Zustände einer Aufgabe

In welchem Zustand sich ein **Task**-Objekt befindet, erfährt man über seine **Status**-Eigenschaft mit den folgenden möglichen Werten aus der Enumeration **TaskStatus**:

- **Created**
  Der Auftrag wurde initialisiert, aber noch nicht zur Bearbeitung eingeplant. In diesem Zustand befindet sich ein Auftrag z. B. nach der Erstellung per Konstruktor:
  ```
  var task = new Task(SampleMean);
  ```
- **Canceled**
  Die Aufgabe wurde abgebrochen (siehe Abschnitt 17.4.10).
- **Faulted**
  Die Bearbeitung wurde durch einen unbehandelten Ausnahmefehler abgebrochen (siehe Abschnitt 17.4.6).
- **RanToCompletion**
  Die Aufgabe wurde erfolgreich zu Ende geführt.
- **Running**
  Der Auftrag wird gerade ausgeführt und ist noch nicht fertiggestellt.
- **WaitingForActivation**
  In diesem Zustand befindet sich z. B. eine Fortsetzungsaufgabe (siehe Abschnitt 17.4.8) vor dem Ende ihres Vorgängers.
- **WaitingForChildrenToComplete**
  Die Aufgabe ist erledigt, wartet aber noch auf die Fertigstellung von sekundären Aufgaben (siehe Abschnitt 17.4.9).
- **WaitingToRun**
  Der Auftrag wartet darauf, von einem freien Thread ausgeführt zu werden.

Informationen über den Status einer Aufgabe lassen sich auch mit den folgenden booleschen **Task**-Eigenschaften ermitteln:

- **IsCanceled**
  Diese Eigenschaft hat den Wert **true**, wenn die Aufgabe abgebrochen worden ist (siehe Abschnitt 17.4.10).
- **IsCompleted**
  Diese Eigenschaft hat den Wert **true**, wenn die Aufgabe fehlerfrei beendet worden ist.
- **IsFaulted**
  Diese Eigenschaft hat den Wert **true**, wenn die Bearbeitung an einem Ausnahmefehler gescheitert ist (siehe Abschnitt 17.4.6).

17.4.5 Auf die Fertigstellung von Aufgaben warten

17.4.5.1 Warten auf eine einzelne Aufgabe

Um auf die Fertigstellung eines **Task**-Objekts zu warten, ruft man dessen **Wait()** - Methode auf, wobei der aktuelle Thread blockiert wird. Als Gründe kommen in Frage:[1]

- Bei vielen Algorithmen hängt das weitere Vorgehen vom Ergebnis einer Aufgabe ab, sodass deren Beendigung abgewartet werden muss.
- Eine Aufgabe wird von einem Pool-Thread, also von einem Hintergrund-Thread ausgeführt. Weil ein Hintergrund-Thread eine Anwendung nicht am Leben erhalten kann, muss der Haupt-Thread vor seiner Beendigung auf eine Aufgabe warten, um deren vollständige Ausführung zu ermöglichen.
- Ein **Wait()** - Aufruf ist eine von mehreren Möglichkeiten, um zu veranlassen, dass eine bei der Aufgabenbearbeitung aufgetretene unbehandelte Ausnahme neu geworfen wird. Das klappt auch dann, wenn die im **Wait()** - Aufruf angesprochene Aufgabe bereits beendet ist. So wird verhindert, dass die Ausnahme unbeobachtet bleibt (siehe Abschnitt 17.4.6).

Ein **Wait()** - Aufruf kehrt sofort zurück, wenn die angesprochene Aufgabe bereits beendet ist (im Zustand **Canceled**, **Faulted** oder **RanToCompletion**).

Wir haben die Methode **Wait()** bereits im Abschnitt 17.4.1.1.1 verwendet, allerdings unvorsichtig ohne Einbindung in eine **try-catch** - Anweisung:

```
Task task = Task.Factory.StartNew(SampleMean);
// Parallele Arbeiten erledigen
task.Wait();
```

Zur unbegrenzt lange blockierenden **Wait()** - Überladung existiert eine Alternative mit Timeout-Parameter:

public bool Wait(TimeSpan *timeout***)**

Über den Rückgabewert vom Typ **bool** erfährt man, ob die Aufgabe in der festgelegten Zeitspanne fertig geworden ist.

Eine weitere Möglichkeit zur Begrenzung der Wartezeit bietet die folgende **Wait()** - Überladung mit einem Parameter vom Typ **CancellationToken**:

public void Wait(CancellationToken *token***)**

Ein Aufruf dieser Methode kehrt zurück, ...

---

[1] https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-based-asynchronous-programming#waiting-for-tasks-to-finish

- wenn die angesprochene Aufgabe endet,
- oder wenn für das **CancellationToken** ein Abbruch angeordnet ist (siehe Abschnitt 17.4.10).

Hat die angesprochene Aufgabe beim **Wait(CancellationToken** *token***)** - Aufruf das Signal zum Abbrechen erhalten, aber noch nicht beachtet, dann bewirkt der Aufruf eine Ausnahme vom Typ **System.OperationCanceledException**. Folglich sollte ein **Wait(CancellationToken** *token***)** - Aufruf im Rahmen einer **try-catch** - Anweisung vorgenommen werden.

Durch die folgende **Wait()** - Überladung erhält man für den Wartezustand sowohl eine zeitliche Begrenzung als auch eine **CancellationToken**-Bedingung:

public bool Wait(int *milliseconds***, CancellationToken** *token***)**

17.4.5.2 Warten auf mehrere Aufgaben

Bei *mehreren* abzuwartenden Aufgaben kommen die statischen **Task**-Methoden **WaitAll()** und **WaitAny()** zum Einsatz, die z. B. in den folgenden Überladungen vorhanden sind:[1]

- **public static void WaitAll(params Task[]** *tasks***)**
  Es wird gewartet, bis *alle* Aufgaben im Parameter-Array abgeschlossen sind.
- **public static void WaitAll(params Task[]** *tasks***, int** *millisec***, CancellationToken** *token*)
  Es wird gewartet, bis alle Aufgaben im ersten Parameter abgeschlossen sind, oder die im zweiten Parameter angegebene Zahl von Millisekunden verstrichen ist, oder für das Token im dritten Parameter ein Abbruch angeordnet ist (siehe Abschnitt 17.4.10).
- **public static void WaitAny(params Task[]** *tasks***)**
  Es wird gewartet, bis irgendeine von den Aufgaben im Parameter-Array abgeschlossen ist.
- **public static void WaitAny(params Task[]** *tasks***, int** *millisec***, CancellationToken** *token*)
  Es wird gewartet, bis irgendeine von den Aufgaben im ersten Parameter abgeschlossen ist, oder die im zweiten Parameter angegebene Zahl von Millisekunden verstrichen ist, oder für das Token im dritten Parameter ein Abbruch angeordnet ist (siehe Abschnitt 17.4.10).

Indem eine durch die statische **Task**-Methode **Delay()** erstellte Aufgabe einbezogen wird, sorgt man dafür, dass bei **WaitAll()** oder **WaitAny()** auf jeden Fall eine bestimmte minimale Wartezeit eingehalten wird. Von der folgenden **Delay()** - Überladung erhält man eine Aufgabe, die nach der im Parameter angegebenen Anzahl von Millisekunden beendet ist:

public static Task Delay(int *milliseconds***)**

17.4.6 Unbehandelte Ausnahmen bei der Aufgabenbearbeitung

In diesem Abschnitt geht es um unbehandelte Ausnahmen, die bei einer Aufgabenerledigung aufgetreten sind. Man sollte sich darum kümmern, was aber von der CLR *nicht* erzwungen wird.

17.4.6.1 Beobachtete Ausnahmen

Ist bei einer Aufgabenbearbeitung eine unbehandelte Ausnahme aufgetreten, dann wird diese ...

---

[1] Mit dem Schlüsselwort **params** wird ein Serienparameter deklariert (siehe Abschnitt 5.3.1.3.3). Man kann also beim Aufruf der Methoden ...

- einen **Task**-Array als Aktualparameter übergeben,
- oder beliebig viele einzelne **Task**-Objekte auflisten.

- beim Aufruf der **Task**-Methoden **Wait()**, **WaitAll()** und **WaitAny()**
- oder beim Zugriff auf eine **Task<TResult>** - Produktion per **Result**-Eigenschaft

erneut geworfen.

Es erfolgt generell eine Verpackung in ein **AggregateException**-Objekt, um für die bei Beteiligung von mehreren Aufgaben bestehende Möglichkeit von mehreren aufgetretenen Ausnahmen vorbereitet zu sein.

Das Warten auf die Aufgabenbearbeitung bzw. der Ergebniszugriff sollte im Rahmen einer **try-catch** - Anweisung stattfinden, damit die **AggregateException** behandelt werden kann. Im Manuskript wird diese Empfehlung gelegentlich der Einfachheit halber *nicht* umgesetzt.

17.4.6.1.1 Primäre Aufgabe

Im folgenden Beispielprogramm beschränkt sich die Aufgabenmethode auf das Werfen einer Ausnahme, wobei eine von zwei selbst definierten Ausnahmeklassen verwendet wird (vgl. Abschnitt 13.6).

```
using System;
using System.Threading.Tasks;

class HarmlessException : Exception {
   public HarmlessException(String message) : base(message) {}
}

class SeriousException : Exception {
   public SeriousException(String message) : base(message) {}
}

class AggregateExceptionDemo {
   static void Werfer() {
      throw new HarmlessException("Harmlos");
      //throw new SeriousException("Ernstfall");
   }

   static bool InternalExceptionHandler(Exception e) {
      if (e is HarmlessException) {
         Console.WriteLine(e.Message);
         return true;
      } else
         return false;
   }

   static void Main() {
      Task t = Task.Factory.StartNew(Werfer);
      try {
         t.Wait();
      } catch (AggregateException ae) {
          ae.Handle(InternalExceptionHandler);
      }
      Console.WriteLine("Normales Ende");
   }
}
```

Zur Analyse der erhaltenen **AggregateException** wird deren Methode **Handle()** aufgerufen, die als Parameter ein Delegatenobjekt vom Typ **Func<Exception, bool>** erwartet. Es ist also eine Methode verlangt, die einen Parameter vom Typ **Exception** entgegennimmt und eine Rückgabe vom Typ **bool** liefert. Diese Methode wird sequentiell für jede im **AggregateException**-Objekt enthaltene Ausnahme aufgerufen und muss per Rückgabewert darüber informieren, ob eine Behandlung stattgefunden hat (**true**) oder nicht (**false**). Wird für eine Ausnahme der Wert **true** gemeldet, gilt sie als *beobachtet*. Beim Rückgabewert **false** für mindestens eine Ausnahme wirft **Handle()** eine neue

AggregateException mit den unbehandelten Ausnahmen. Kommt es bei einem Aufruf der **Func<Exception, bool>** - Methode zu einer Ausnahme, stellt **Handle()** seine Tätigkeit ein und leitet diese Ausnahme weiter.

Die Flexibilität und Komplexität eines **AggregateException**-Objekts sind nur relevant, wenn *mehrere* Aufgaben im Spiel sind (z. B. beim Aufruf von **WaitAll()**, bei Existenz von angehefteten sekundären Aufgaben oder bei den im Abschnitt 17.4.9 beschriebenen zusammengesetzten Aufgaben). Im aktuellen Beispiel kann das **AggregateException**-Objekt nur eine einzelne Ausnahme enthalten. Ist diese vom Typ `HarmlessException`, findet eine Behandlung statt, und das Programm wird normal weitergeführt:

```
Harmlos
Normales Ende
```

Im Falle einer `SeriousException` unterbleibt eine Behandlung, und das Programm wird von der CLR beendet.

Man kann sich über einen Ausnahmefehler bei der Auftragsbearbeitung informieren, ohne dass dabei die Ausnahme neu geworfen wird:

- Die **Task**-Eigenschaft **IsCanceled** informiert darüber, ob eine **OperationCanceledException** geworfen wurde (vgl. Abschnitt 17.4.10).
- Die **Task**-Eigenschaft **IsFaulted** informiert darüber, ob eine beliebige andere Ausnahme geworfen wurde.

Eine unbehandelte Ausnahme, die zum Task-Zustand **Faulted** geführt hat, wird als beobachtet eingestuft, wenn auf die **Exception**-Eigenschaft der sich im Zustand **Faulted** befindenden Task zugegriffen wird. Für einen solchen Zugriff ist die im Abschnitt 17.4.8 beschriebene Fortsetzungsaufgabe sehr gut geeignet, sodass wir einen Vorgriff auf diesen Abschnitt in Kauf nehmen. Im folgenden Programm vereinbaren wir in einem **ContinueWith()** - Aufruf an eine zu analysierende, garantiert mit einer unbehandelten Ausnahme endende Aufgabe eine Fortsetzungsaufgabe. Weil der **ContinueWith()** - Parameter *continuationOptions* den Wert **OnlyOnFaulted** aus der Enumeration **TaskContinuationOptions** erhält, wird die Fortsetzungsaufgabe ausschließlich nach dem Auftreten eines Ausnahmefehlers in der angesprochenen Aufgabe ausgeführt. In der Fortsetzungsaufgabe findet lediglich ein lesender Zugriff auf die **Exception**-Eigenschaft der gescheiterten Aufgabe statt mit dem Effekt, dass die „beobachtete“ Ausnahme beim späteren **Wait()** - Aufruf *nicht* neu geworfen wird:

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class TaskException {
    static void Main() {
        var t1 = Task.Factory.StartNew(() => {
            Console.WriteLine("Thrower started");
            throw new Exception("Bang");
        }).ContinueWith(t => {
            Console.WriteLine(t.Exception.InnerException.Message);
        }, TaskContinuationOptions.OnlyOnFaulted);

        Thread.Sleep(100);
        t1.Wait();
        Console.WriteLine("No exception observed");
    }
}
```

Es resultiert die Ausgabe:

```
Thrower started
Bang
No exception observed
```

Die Klasse **AggregateException** stammt von **Exception** ab und erbt daher die Eigenschaft **InnerException**, mit der ein **Exception**-Objekt auf ein ursprüngliches **Exception**-Objekt (auf den primären Unfall) verweisen kann. Die Klasse **AggregateException** besitzt eine Eigenschaft namens **InnerExceptions**, die auf eine Kollektion mit den in beteiligten Aufgaben aufgetretenen Ausnahmeobjekten verweist. Das erste Element in dieser Kollektion ist auch über die Eigenschaft **InnerException** ansprechbar, was in einigen Beispielprogrammen geschieht, z. B.:

```
t.Exception.InnerException
```

17.4.6.1.2 Sekundäre Aufgabe

Wenn in einer primären Aufgabe eine sekundäre Aufgabe (engl. *child task*) mit der Option **TaskCreationOptions.AttachedToParent** erstellt wird und dort eine unbehandelte Ausnahme auftritt, dann wird diese nach den obigen Erläuterungen in ein **AggregateException**-Objekt verpackt an die primäre Aufgabe übergeben. Die primäre Aufgabe erstellt ihrerseits ein **AggregateException**-Objekt zur Weiterleitung von unbehandelten Ausnahmen an den aufrufenden Thread. Auf diese Weise kommt es zu einer Verschachtelung von Ausnahmeobjektlisten, die beliebig komplex werden kann. Mit der **AggregateException**-Methode **Flatten()** hebt man die Verschachtelung auf, sodass alle Ausnahmen in einer einzigen Schleife behandelt werden können:[1]

public AggregateException Flatten()

Im folgenden Beispielprogramm iteriert die **AggregateException**-Methode **Handle()** über alle bei der Aufgabenbearbeitung unbehandelt gebliebenen Ausnahmen, die zuvor per **Flatten()** in *einer* Liste zusammengefasst wurden:

```
using System;
using System.Threading.Tasks;

class HarmlessException : Exception {
  public HarmlessException(String message) : base(message) { }
}
class SeriousException : Exception {
  public SeriousException(String message) : base(message) { }
}

class FlattenDemo {
  static void WithChildTasks() {
    Task t1 = Task.Factory.StartNew(
      () => { int i = 0; int j = 12 / i; },
      TaskCreationOptions.AttachedToParent);
    Task t2 = Task.Factory.StartNew(
      () => {throw new HarmlessException("Harmlos"); },
      TaskCreationOptions.AttachedToParent);
    throw new SeriousException("Ernstfall");
  }

  static bool InternalExceptionHandler(Exception e) {
    Console.WriteLine(e.Message);
    return true;
  }
```

[1] https://docs.microsoft.com/de-de/dotnet/api/system.aggregateexception.flatten

```
    static void Main() {
      Task t = Task.Factory.StartNew(WithChildTasks);
      try {
        t.Wait();
      } catch (AggregateException ae) {
        ae.Flatten().Handle(InternalExceptionHandler);
      }
    }
}
```

Es wird von einer Ausnahme in der primären Aufgabe und von zwei Ausnahmen in sekundären Aufgaben berichtet:

```
Ernstfall
Harmlos
Es wurde versucht, durch 0 (null) zu teilen.
```

Werden sekundäre Aufgaben im voreingestellten abgetrennten Modus erstellt (also *ohne* die Erstellungsoption **TaskCreationOptions.AttachedToParent**), dann werden ihre unbehandelten Ausnahmen *nicht* an die primäre Aufgabe weitergeleitet. Im Beispiel hat die entsprechend geänderte Methode `WithChildTasks()`

```
static void WithChildTasks() {
    Task t1 = Task.Factory.StartNew(
        () => { int i = 0; int j = 12 / i; });
    Task t2 = Task.Factory.StartNew(
        () => {throw new HarmlessException("Harmlos"); });
    throw new SeriousException("Ernstfall");
}
```

einen unvollständigen Unfallbericht zur Folge:

```
Ernstfall
```

Die Ausnahmen aus den sekundären Aufgaben bleiben unbeobachtet, was durchaus negative Konsequenzen haben kann.

17.4.6.2 Unbeobachtete Ausnahmen

Tritt in einer Aufgabe eine unbehandelte Ausnahme auf, die weder ...

- beim Aufruf der **Task**-Methoden **Wait()**, **WaitAll()** und **WaitAny()**
- oder beim Zugriff auf eine **Task<TResult>** - Produktion per **Result**-Eigenschaft

erneut geworfen und dann abgefangen wird, noch durch Zugriff auf die **Task**-Eigenschaft **Exception** beobachtet wird, dann liegt eine sogenannte *unbeobachtete* Ausnahme vor.

Das passiert auch dann, wenn eine Aufgabe einen **Wait()** - Aufruf mit Timeout-Parameter erhält, und in dieser Aufgabe *nach* Ablauf der Wartezeit eine Ausnahme auftritt. Im folgenden Beispielprogramm wird das statische Ereignis **UnobservedTaskException** der Klasse **TaskScheduler** dazu verwendet, um die Existenz einer unbeobachteten Ausnahme nachzuweisen:

```
using System;
using System.Threading;
using System.Threading.Tasks;

class ExceptionAfterTimeout {
   static void Werfer() {
      Thread.Sleep(500);
      throw new Exception();
   }
```

```
    static void Main() {
       TaskScheduler.UnobservedTaskException +=
          (object sender, UnobservedTaskExceptionEventArgs eventArgs) => {
             Console.WriteLine("UnobservedTaskException-Handler");
             //eventArgs.SetObserved();
          };

       Task t = Task.Factory.StartNew(Werfer);

       try {
          t.Wait(50);
       } catch {
          Console.WriteLine("Diese Ausgabe wird nie erscheinen.");
       }
       Thread.Sleep(1000); // Die Aufgabe erhält Zeit zum Werfen der Ausnahme
       t = null;
       GC.Collect();
       for (int i = 0; i < 5; i++) {
          Thread.Sleep(500);
          Console.Write(i + " ");
       }
    }
}
```

Dabei ist zu beachten:

- Das statische **TaskScheduler**-Ereignis **UnobservedTaskException** wird nur dann ausgelöst, wenn die **Release**-Konfiguration des Projekts eingestellt ist.
- Eine unbeobachtete Ausnahme wird beim Abräumen des **Task**-Objekts per Garbage Collector von der CLR diagnostiziert. Im Beispiel wird die einzige Referenz auf das **Task**-Objekt aufgehoben und dann mit der statischen Methode **Collect()** der Klasse **GC** die Tätigkeit des Müllsammlers angeordnet:

  ```
  t = null;
  GC.Collect();
  ```

In der **Release**-Konfiguration ausgeführt, belegt das Programm die Existenz einer unbeobachteten Ausnahme:

```
UnobservedTaskException-Handler
```

Das Ereignis **UnobservedTaskException** bietet eine letzte Gelegenheit, sich um eine bei der Aufgabenbearbeitung aufgetretene unbehandelte Ausnahme zu kümmern (Griffiths 2013, S. 642f). Insbesondere kann die Ausnahme durch die Methode **SetObserved()** der Klasse **UnobservedTaskExceptionEventArgs** als beobachtet deklariert und damit neutralisiert werden. Geschieht dies nicht, hängt der weitere Ablauf von der .NET - Implementation und (im .NET Framework) von der Anwendungskonfiguration ab:[1]

[1] https://docs.microsoft.com/de-de/dotnet/framework/configure-apps/file-schema/runtime/throwunobservedtaskexceptions-element

- Ab .NET Framework 4.5 wird eine unbeobachtete Ausnahme **ignoriert**, wenn diese Voreinstellung nicht durch das Konfigurationselement **ThrowUnobservedTaskExceptions** in der Datei **app.config** geändert wird:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <runtime>
    <ThrowUnobservedTaskExceptions enabled="true"/>
  </runtime>
</configuration>
```

- In .NET 5.0 ist die Projektkonfiguration alternativ organisiert und die Datei **app.config** entfernt worden. Weil zum Konfigurationselement **ThrowUnobservedTaskExceptions** kein Analogon vorhanden ist, wird eine unbeobachtete Ausnahme auf jeden Fall **ignoriert**.

Das obige Programm zeigt also bei aktiver **Release**-Konfiguration in Abhängigkeit von der .NET-Implementation bzw. -Konfiguration ein stark abweichendes Verhalten:

a) .NET Framework 4.8 *ohne* **ThrowUnobservedTaskExceptions** - Element in **app.config**
Die Ausnahme wird ignoriert, und das Programm wird fortgesetzt:

```
UnobservedTaskException-Handler
0 1 2 3 4
```

b) .NET Framework 4.8 *mit* **ThrowUnobservedTaskExceptions** - Element in **app.config**
Die Ausnahme wird geworfen, und das Programm wird beendet:

```
UnobservedTaskException-Handler

Unbehandelte Ausnahme: System.AggregateException: Ausnahmen einer Aufgabe wurden
nicht überwacht (entweder wegen Wartens auf die Aufgabe oder wegen des Zugriffs auf
die Ausnahmeeigenschaft. Daher wurde die nicht überwachte Ausnahme vom Finalizer-
Thread erneut ausgelöst. ---> System.Exception: Eine Ausnahme vom Typ
"System.Exception" wurde ausgelöst.
   bei ExceptionAfterTimeout.Werfer() in U:\Eigene
Dateien\C#\BspUeb\Multithreading\TPL\Ausnahmebehandlung\ExceptionAfterTimeout (.NET
Framework)\ExceptionAfterTimeout.cs:Zeile 8.
   bei System.Threading.Tasks.Task.Execute()
   --- Ende der internen Ausnahmestapelüberwachung ---
   bei System.Threading.Tasks.TaskExceptionHolder.Finalize()
0
```

c) .NET 5.0
Die Ausnahme wird ignoriert, und das Programm wird fortgesetzt:

```
UnobservedTaskException-Handler
0 1 2 3 4
```

Bei Verwendung der **Debug**-Konfiguration zeigt das Programm unter den drei beschriebenen Konstellationen dasselbe Verhalten:

```
0 1 2 3 4
```

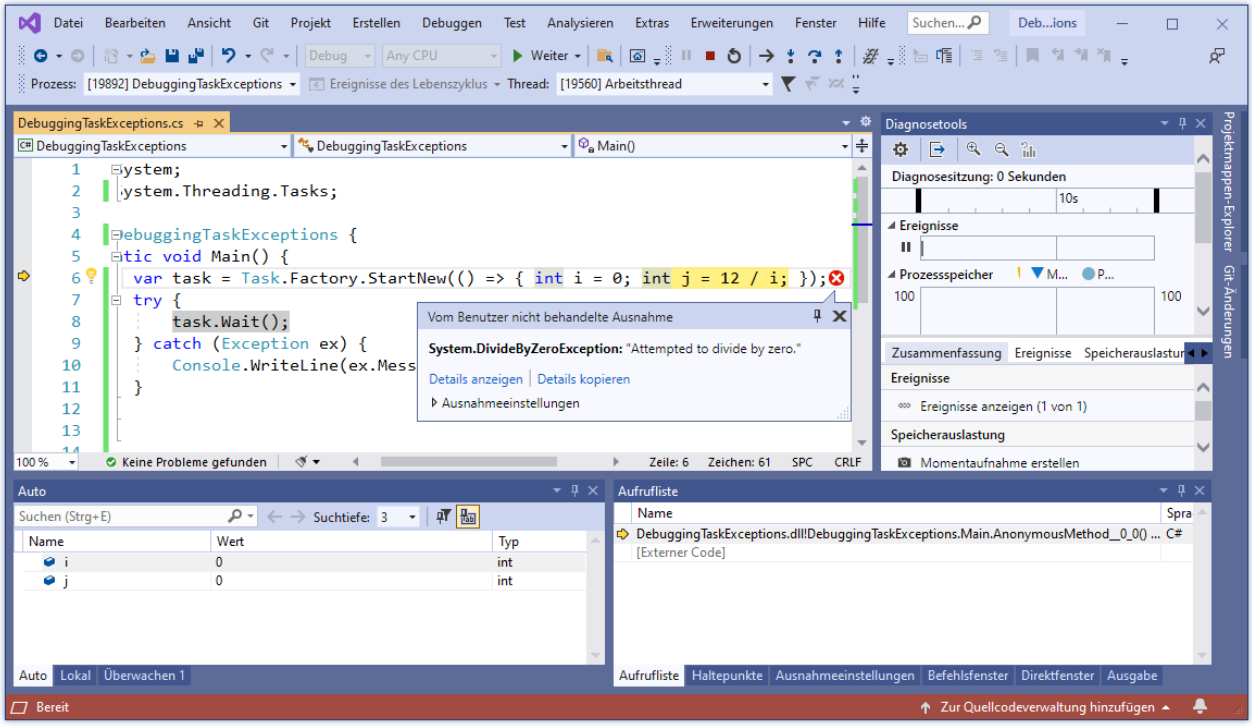
Das Ignorieren von Ausnahmen ist in der Regel *keine* sinnvolle Option. Also sollten alle bei der Aufgabenbearbeitung aufgetretenen und unbehandelt gebliebenen Ausnahmen beobachtet werden (siehe Abschnitt 17.4.6.1). Weil z. B. aufgrund der unvermeidlichen Nutzung von Bibliotheksmethoden unbehandelte und unbeobachtete Ausnahmen bei der Aufgabenbearbeitung nicht ausgeschlossen werden können, sind Vorsorgemaßnahmen sinnvoll, damit kein Unfall im Verborgenen bleibt:

- Es sollte ein **UnobservedTaskException**-Handler eingerichtet werden, der diagnostische Informationen liefert.
- Im .NET Framework sollte über das Anwendungsattribut **ThrowUnobservedTaskExceptions** dafür gesorgt werden, dass eine unbeobachtete Ausnahme im Finalisierer-Thread erneut geworfen wird.

Das Ereignis **UnobservedTaskException** sowie das erneute Werfen einer unbeobachteten Ausnahme finden allerdings erst im Rahmen der Finalisierung des betroffenen **Task**-Objekts statt, also mit erheblicher Verzögerung oder überhaupt nicht. Diese Reaktion auf eine unbeobachtete **Task**-Ausnahme unterscheidet sich also deutlich von der im Abschnitt 17.1.8 beschriebenen Reaktion auf eine unbehandelte Ausnahme in einem sekundären Thread.

17.4.6.3 Task-Ausnahmen im Debug-Modus der Entwicklungsumgebung

Wenn ein Programm (z. B. mit **F5**) im Debug-Modus der Entwicklungsumgebung ausgeführt wird, und Task-intern eine unbehandelte Ausnahme auftritt, dann stoppt das Visual Studio das Programm an der Unfallstelle, z. B.:[1]

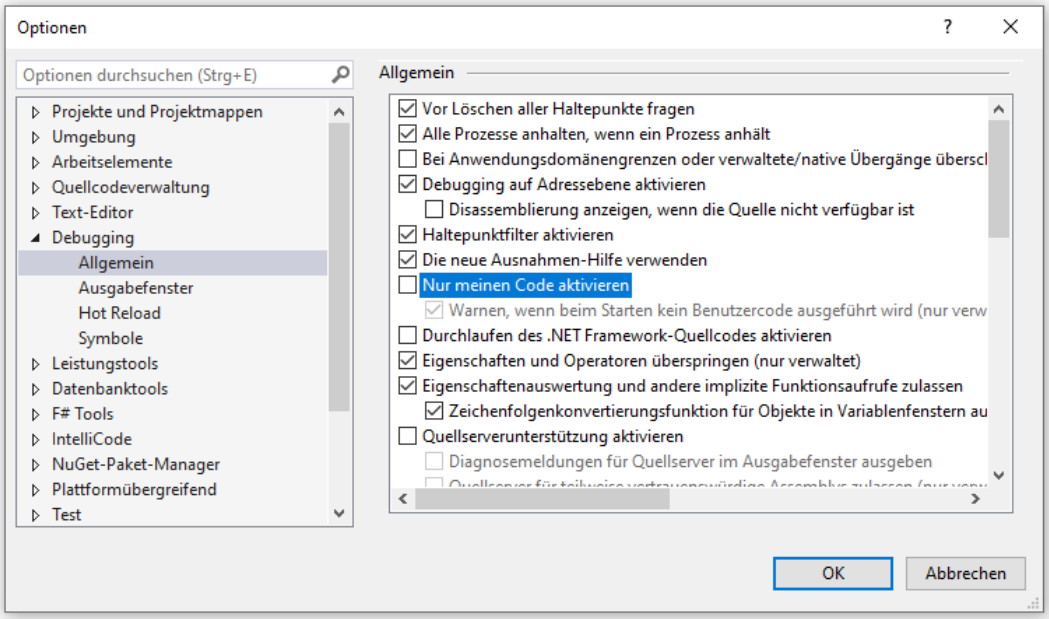


Wenn das *nicht* erwünscht ist, muss man nach

Extras > Optionen > Debugging > Allgemein

die Markierung zum Kontrollkästchen **Nur meinen Code aktivieren** entfernen:

[1] Die Programmausführung im Debug-Modus ist von der Debug-Konfiguration des Projekts zu unterscheiden.



17.4.7 Aufgabenplaner und Synchronisierungskontext

TPL-Aufgaben werden von einem Objekt der Klasse **TaskScheduler** (dt.: *Aufgabenplaner*) auf Threads verteilt. Der voreingestellte Aufgabenplaner benutzt den Threadpool-Synchronisierungskontext und verwendet dementsprechend zur Durchführung der Aufgaben den Threadpool, wobei die mit dem .NET Framework 4.0 eingeführten Threadpool-Optimierungen für eine gute Performanz sorgen (siehe Abschnitt 17.2.2).[1]

Bei GUI-Anwendungen ist es aber oft erforderlich, dass eine Aufgabe im UI-Thread ausgeführt wird, weil sie Änderungen an Steuerelementen vornimmt. Eine solche Aufgabe muss einem Aufgabenplaner übergeben werden, der dem UI-Synchronisierungskontext zugeordnet ist. Man erhält einen solchen Aufgabenplaner über einen im UI-Thread vorgenommenen Aufruf der statischen **TaskScheduler**-Methode **FromCurrentSynchronization()**.

Aus der folgenden Anweisung mit der Deklaration und Initialisierung einer Instanzvariablen in einer Fensterklasse einer WPF-Anwendung resultiert ein **TaskScheduler**-Objekt mit dem UI- Synchronisierungskontext:

```
TaskScheduler uiTS = TaskScheduler.FromCurrentSynchronizationContext();
```

In der folgenden Ereignisbehandlungsmethode wird eine per Pool-Thread auszuführende Aufgabe gestartet. An das von **StartNew<double>()** gelieferte **Task<double>** - Objekt wird per **ContinueWith()** eine Fortsetzungsaufgabe angehängt (siehe Abschnitt 17.4.8.1), die im UI-Thread auszuführen ist, und daher an ein **TaskScheduler**-Objekt mit dem UI-Synchronisierungskontext übergeben wird:

```
private void Button_Click(object sender, RoutedEventArgs e) {
   label.Content = "";
   Task<double> task = Task.Factory.StartNew<double>(SampleMean);
   task.ContinueWith(t => { label.Content = t.Result.ToString(); }, uiTS);
}
```

Für .NET - Software sind in der Regel die folgenden Synchronisierungskontexte relevant (Cleary 2014, S. 4):

---

[1] https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskscheduler

- Threadpool-Kontext
  Hier kommen mehrere Threads zum Einsatz.
- UI-Kontext
  Hier wird nur der UI-Thread verwendet.
- Request-Kontext (bei einer ASP.NET - Anwendung)

17.4.8 Fortsetzungsaufgaben

17.4.8.1 Fortsetzung zu einer einzelnen Aufgabe

Über diverse Instanzmethoden der Klassen **Task** und **Task<TResult>** kann man eine Aufgabe erstellen, die gestartet wird, sobald eine andere Aufgabe abgeschlossen ist. Durch die folgende Überladung der **Task<TResult>** - Methode **ContinueWith()**, wird z. B. eine Fortsetzungsaufgabe erstellt, der eine **Action<Task<TResult>>** - Methode zugrunde liegt:

public Task ContinueWith(Action<Task<TResult>> *continuation***, TaskScheduler** *sched***)**

Eine Methode mit dem Delegatentyp **Action<Task<TResult>>** erhält per Parameter ein **Task<TResult>** - Objekt und liefert keine Rückgabe, sodass ein **Task**-Objekt resultiert. Mit dem **ContinueWith()** - Parameter vom Typ **TaskScheduler** wird dafür gesorgt, dass die Fortsetzungsaufgabe einem bestimmten Aufgabenplaner übergeben wird.

In der folgenden **Click**-Behandlungsmethode zu einem **Button**-Objekt in einer WPF-Anwendung

```
private void Button_Click(object sender, RoutedEventArgs e) {
   label.Content = "";
   Task<double> task = Task.Factory.StartNew<double>(SampleMean);
   task.ContinueWith(t => { label.Content = t.Result.ToString(); }, uiTS);
}
```

wird in einer Aufgabe vom Typ **Task<double>** (basierend auf der Methode `SampleMean`) eine Hintergrundberechnung vorgenommen. Um das Ergebnis anschließend an ein WPF-Steuerelement zu übergeben, wird durch die eben erläuterte **ContinueWith()** - Überladung eine im UI-Thread auszuführende Fortsetzungsaufgabe vom Typ **Task** erstellt:

- Als erster **ContinueWith()** - Parameter wird eine per Lambda-Notation realisierte Methode übergeben, welche den Delegatentyp **Action<Task<double>>** erfüllt (Rückgabetyp **void**, ein Parameter vom Typ **Task<double>**).
- Als zweiter Parameter wird ein dem UI-Synchronisierungskontext zugeordnetes **TaskScheduler**-Objekt verwendet (zur Erstellung siehe Abschnitt 17.4.7).

Ist der Vorgänger beim **ContinueWith()** - Aufruf bereits beendet, startet die Fortsetzungsaufgabe sofort.

Für die eventuell von der Vorgängeraufgabe erstellten Sekundäraufgaben gilt:

- Unter Verwendung der Option **TaskCreationOptions.AttachedToParent** angekoppelte Sekundäraufgaben müssen beendet sein, bevor die Vorgängeraufgabe als abgeschlossen gilt. Auf diese Aufgaben wird also gewartet.
- Auf abgekoppelte Sekundäraufgaben wird nicht gewartet.

Eine Fortsetzungsaufgabe wird per Voreinstellung *auf jeden Fall* ausgeführt, also unabhängig davon, ob die vorherige Aufgabe erfolgreich beendet wurde, mit einem Ausnahmefehler gescheitert ist (siehe Abschnitt 17.4.6) oder abgebrochen wurde (siehe Abschnitt 17.4.10). Bei einigen **ContinueWith()** - Überladungen kann durch einen Parameter vom Typ **TaskContinuationOptions** dafür gesorgt werden, dass die Fortsetzungsaufgabe nur dann ausgeführt wird, ...

- wenn der Vorgänger erfolgreich war (**OnlyOnRanToCompletion**),
- wenn der Vorgänger an einem Ausnahmefehler gescheitert ist (**OnlyOnFaulted**)
- oder wenn der Vorgänger abgebrochen wurde (**OnlyOnCanceled**).

Neben den Fortsetzungsbedingungen enthält die Enumeration **TaskContinuationOptions** noch weitere Werte, u. a.:

- Alle Werte der im Abschnitt 17.4.1.2 vorgestellten Enumeration **TaskCreationOptions**
- **ExecuteSynchronously**
  Die Fortsetzungsaufgabe wird im selben Thread ausgeführt wie der Vorgänger. Das spart etwas Verwaltungsaufwand, wird von Microsoft aber nur für *sehr kleine* Fortsetzungsaufgaben empfohlen.[1] Als Voreinstellung wäre die Option **ExecuteSynchronously** ungeeignet, weil zu einer Aufgabe durchaus *mehrere* Fortsetzungsaufgaben definiert werden können. In dieser Situation sollte die Parallelisierung nicht behindert werden.

Weil an die Enumeration **TaskContinuationOptions** das **FlagsAttribute** angeheftet ist, sind ihre Werte kombinierbar (vgl. Abschnitt 14.5.2).

Liegt bei Beendigung der Vorgängeraufgabe eine im **ContinueWith()** - Aufruf vereinbarte Fortsetzungsoption nicht vor, erhält die Fortsetzungsaufgabe den Status **Canceled**. Denselben Status erhält die Fortsetzungsaufgabe auch, wenn sie bereits eine Abbruchaufforderung per **CancellationToken** erhalten hat (siehe Abschnitt 17.4.10).

Um eine unbehandelte Ausnahme in einer Vorgängeraufgabe muss man sich explizit kümmern, damit sie nicht übersehen wird. Eine typische Lösung besteht darin, die beim Zugriff auf die **Result**-Eigenschaft der Vorgängeraufgabe ggf. neu geworfene Ausnahme per **catch**-Block zu behandeln, z. B.:

```
task.ContinueWith(t => {
    try {
        label.Content = t.Result.ToString();
    } catch (Exception ex) {
        label.Content = ex.InnerException.Message;
    }
}, uiTS);
```

Alternativ kann man durch einen **ContinueWith()** - Aufruf mit dem **TaskContinuationOptions**-Wert **OnlyOnFaulted** eine spezielle Fortsetzungsaufgabe definieren, die nur nach dem Auftreten eines Ausnahmefehlers ausgeführt wird, z. B.:[2]

```
task.ContinueWith(t => { label.Content = t.Exception.InnerException.Message; },
                  token, TaskContinuationOptions.OnlyOnFaulted, uiTS);
task.ContinueWith(t => { label.Content = t.Result.ToString(); },
                  token, TaskContinuationOptions.OnlyOnRanToCompletion, uiTS);
```

Informationen über den Status des Vorgängers machen es möglich, in einer Fortsetzungsaufgabe differenziert zu reagieren, z. B.:

```
task.ContinueWith(t => {
   if (t.IsFaulted)
      label.Content = t.Exception.InnerException.Message;
   else
      label.Content = t.Result.ToString();
}, uiTS);
```

---

[1] https://docs.microsoft.com/de-de/dotnet/api/system.threading.tasks.taskcontinuationoptions

[2] Der zweite Parameter der in den Beispielen verwendeten **ContinueWith()** - Überladung ist vom Typ **CancellationToken**, der im Abschnitt 17.4.10 behandelt wird.

17.4.8.2 Fortsetzung zu einer Aufgabenserie

Über die **TaskFactory**-Methode **ContinueWhenAll**() kann man eine Aufgabe erstellen, die gestartet wird, sobald aus einem Array von Aufgaben alle Elemente abgeschlossen sind. Von den 16 Überladungen der Methode **ContinueWhenAll**() soll die folgende gleich in einem Beispielprogramm verwendet werden. Die abzuwartenden Aufgaben liefern jeweils ein Ergebnis vom Typ **TAntecedentResult**:

public Task ContinueWhenAll<TAntecedentResult>(Task<TAntecedentResult>[] *tasks*,
Action<Task<TAntecedentResult>[]> *continuationAction*)

Die der Fortsetzungsaufgabe zugrundeliegende Methode erhält beim Aufruf als Parameter einen Array vom Typ **Task<TAntecedentResult>[]** mit den Vorgängeraufgaben und kann folglich bei jeder Vorgängeraufgabe über die **Result**-Eigenschaft das Ergebnis abfragen.

Um die Methode zu demonstrieren, setzen wir unser Beispiel zur Untersuchung von Zufallszahlen fort (siehe z. B. Abschnitt 17.4.2) und erzeugen einen Array mit Elementen vom Typ **Task<double>**, die jeweils einen Stichprobenmittelwert berechnen:

```
static readonly int number = 1_000;
static readonly int size = 100;
 . . .
var sampleMeanTaskArray = new Task<double>[number];
for (int i = 0; i < number; i++)
   sampleMeanTaskArray[i] = Task.Factory.StartNew<double>(SampleMean, i);
```

Dank moderner PC-Rechenleistung und TPL sollte es kein Problem darstellen, z. B. aus 1.000 Stichproben vom jeweiligen Umfang 100 bei optimaler Parallelisierung den Mittelwert zu berechnen. Wenn alle Stichprobenmittel vorliegen, sollen diese in einer Fortsetzungsaufgabe untersucht werden. Die zugrunde liegende Methode `Summary()`

```
static void Summary(Task<double>[] sampleMeanTaskArray) {
   double x;
   int n = sampleMeanTaskArray.Length;
   double sum = 0.0;
   double sq = 0.0;
   for (int j = 0; j < n; j++) {
      x = sampleMeanTaskArray[j].Result;
      sum += x;
      sq += x*x;
   }
   double mean = sum / n;
   double variance = (sq - (sum * sum / n)) / (n - 1);
   double sd = Math.Sqrt(variance);
   Console.WriteLine("\nAnzahl = {0,25:d}\nUmfang = {1,25:d}\nMittel = {2,25:f7}"+
                     "\nVarianz = {3,24:f7}\nStandardabweichung = {4,13:f7}",
                     n, size, mean, variance, sd);
}
```

erfüllt den Delegatentyp **Action<Task<double>[]>**. Sie erhält per Parameter den Array mit den Aufgaben zum Berechnen der einzelnen Stichprobenmittelwerte und ermittelt daraus das Gesamtmittel sowie die Varianz und die Standardabweichung der Stichprobenmittelwerte. So lässt sich z. B. beobachten, wie die Standardabweichung der Stichprobenmittelwertverteilung (also der geschätzte Standardfehler) von der Stichprobengröße abhängt.

Aus dem folgenden Aufruf der **TaskFactory**-Methode **ContinueWhenAll<double>()** resultiert eine neue Aufgabe, welche asynchron die Methode `Summary()` ausführt, sobald alle Aufgaben im Array `sampleMeanTaskArray` erledigt sind:

```
Task sTsk = Task.Factory.ContinueWhenAll<double>(sampleMeanTaskArray, Summary);
```

Eine Analyse von 1.000 Stichproben vom Umfang 100 aus einer Population mit gleichverteilten Werten auf dem Intervall [0, 1) zeigt, dass die Mittelwerte aus derartigen Stichproben nur noch sehr wenig um den Erwartungswert 0,5 variieren (Standardabweichung: 0,029):

```
Anzahl =                     1000
Umfang =                      100
Mittel =                0,5019044
Varianz =               0,0008442
Standardabweichung =    0,0290554

Zeit im Sek.: 0,0445402
```

Über die **TaskFactory**-Methode **ContinueWhenAny**() kann man eine Aufgabe erstellen, die gestartet wird, sobald *irgendein* Element aus einem Array mit Aufgaben abgeschlossen ist. Von den 16 Überladungen der Methode **ContinueWhenAny**() liefert die folgende

public Task ContinueWhenAny<TResult>(Task<TResult>[] *tasks*,
Action<Task<TResult>> ***continuationAction*)**

eine Fortsetzungsaufgabe, die über den Parameter der zugrundeliegenden Methode vom Delegatentyp **Action<Task<TResult>>** von der zuerst abgeschlossenen Vorgängeraufgabe erfährt und per **Result**-Eigenschaft auf deren Ergebnis zugreifen kann.

17.4.9 Metaaufgaben

Werden in einer primären Aufgabe sekundäre Aufgaben erstellt unter Verwendung der Option **TaskCreationOptions.AttachedToParent**, dann wird die primäre Aufgabe nur dann als abgeschlossen betrachtet, wenn alle sekundären Aufgaben abgeschlossen sind (siehe Abschnitt 17.4.2). Auch mit den statischen Methoden **WhenAll**() und **WhenAny**() der Klasse **Task** lässt sich eine Metaaufgabe erstellen, deren Status von *mehreren* Einzelaufgaben abhängt. Es sind u. a. die folgenden Überladungen vorhanden:[1]

- **public static Task WhenAll(params Task[]** *tasks***)**
  Die resultierende Metaaufgabe ist abgeschlossen, wenn *alle* Elemente im Parameter-Array (mit Aufgaben ohne Ergebnislieferung) abgeschlossen sind.
- **public static Task<TResult[]> WhenAll<TResult>(params Task<TResult>[]** *tasks***)**
  Die resultierende Metaaufgabe ist abgeschlossen, wenn *alle* Elemente im Parameter-Array, der Aufgaben mit Ergebnislieferung enthält, abgeschlossen sind. Über ihre **Result**-Eigenschaft stellt die Metaaufgabe einen Array mit den Ergebnissen der Elementaufgaben zur Verfügung.
- **public static Task<Task> WhenAny(params Task[]** *tasks***)**
  Die resultierende Metaaufgabe ist abgeschlossen, wenn *irgendein* Element im Parameter-Array (mit Aufgaben ohne Ergebnislieferung) abgeschlossen ist. Über ihre **Result**-Eigenschaft informiert die Metaaufgabe darüber, welche Elementaufgabe zuerst abgeschlossen war.

---

[1] Mit dem Schlüsselwort **params** wird ein Serienparameter deklariert (siehe Abschnitt 5.3.1.3.3). Man kann also beim Aufruf der Methoden ...

- einen **Task**-Array als Aktualparameter übergeben,
- oder beliebig viele einzelne **Task**-Objekte auflisten.

- **public static Task<Task<TResult>> WhenAny<TResult>(params Task<TResult>[]** *tasks***)**
  Die resultierende Metaaufgabe ist abgeschlossen, wenn *irgendein* Element im Parameter-Array, der Aufgaben mit Ergebnislieferung enthält, abgeschlossen ist. Das Ergebnis der Metaaufgabe ist die abgeschlossene Elementaufgabe, die wiederum ein Ergebnis vom Typ **TResult** enthält.

Wird per **Wait()** - Methode auf eine per **WhenAll()** erstellte Metaaufgabe gewartet, dann wird ggf. eine **AggregateException** geworfen, die alle in Einzelaufgaben aufgetretenen unbehandelten Ausnahmen enthält, z. B.:

Quellcode	Ausgabe

```
using System;
using System.Threading;
using System.Threading.Tasks;

class WhenAllDemo {
    static bool InternalExceptionHandler(Exception e) {
        Console.WriteLine(e.Message);
        return true;
    }

    static void Main() {
        Task t1 = Task.Run(() => {
            Thread.Sleep(1000);
            throw new Exception("Task 1 Faulted");
        });
        Task t2 = Task.Run(() => {
            Thread.Sleep(1000);
        });
        Task t = Task.WhenAll(t1, t2);
        try {
            t.Wait();
        } catch (AggregateException ae) {
            ae.Handle(InternalExceptionHandler);
        }
        Console.WriteLine("Status von t: " + t.Status);
    }
}
```

```
Task 1 Faulted
Status von t: Faulted
```

Endet mindestens eine Aufgabe mit einem Ausnahmefehler, dann hat die von **WhenAll()** erstellte Metaaufgabe den Status **Faulted** (siehe Beispiel).

Demgegenüber endet eine per **WhenAny()** erstellte Metaaufgabe stets mit dem Status **RanToCompletion**. Das gilt auch dann, wenn ihr Ergebnis, also die zuerst beendete Aufgabe den Status **Faulted** oder **Canceled** besitzt. Man muss also über die **Result**-Eigenschaft der **WhenAny()** - Rückgabe die Ergebnisaufgabe ermitteln und deren Status überprüfen, z. B.:

Quellcode	Ausgabe

`using System;`
`using System.Threading;`
`using System.Threading.Tasks;`
 	
`class WhenAnyDemo {`
`  static void SleepAndThrow(Object ii) {`
`  int i = (int)ii;`
`  Thread.Sleep(1000);`
`  throw new Exception("Task " + i + "Faulted");`
`  }`
 	
`  static void Main() {`
`  Task t1 = Task.Factory.StartNew(SleepAndThrow, 1);`
`  Task t2 = Task.Factory.StartNew(SleepAndThrow, 2);`
`  Task<Task> t = Task.WhenAny(t1, t2);`
`  t.Wait();`
`  Console.WriteLine("Status von t: "+t.Status);`
`  Console.WriteLine("Zuerst fertig: "+t.Result.AsyncState+`
`  "\n  mit Status: " + t.Result.Status);`
`  }`
`}`
	`Status von t: RanToCompletion`
	`Zuerst fertig: 2`
	`  mit Status: Faulted`

Die **Wait()** - Methode in einer **try-catch** - Anweisung aufzurufen, ist nicht erforderlich, wenn die ausschließlich zu erwartenden Ausnahmen aus den Klassen **ArgumentNullException** und **ArgumentException** auszuschließen sind.

Nachdem eine per **WhenAny()** erstellte Metaaufgabe abgeschlossen ist, lohnt es sich eventuell, die nicht mehr benötigten Aufgaben abzubrechen (siehe Abschnitt 17.4.10).

Statt zu einer **WhenAll()** - oder **WhenAny()** - Rückgabe eine Fortsetzungsausgabe zu definieren, kann man die **TaskFactory**-Methoden **ContinueWhenAll**() und **ContinueWhenAny**() verwenden (siehe Abschnitt 17.4.8.2).

Nützlich sind die Methoden **WhenAll()** und **WhenAny()** z. B. im Operanden des im Abschnitt 17.5.1 vorgestellten **await**-Operators. Dort ist *eine* Aufgabe zulässig, und die Metaaufgaben sorgen für die Möglichkeit zur flexiblen Beschreibung einer abzuwartenden Konstellation aus mehreren Aufgaben, z. B.:

```
await Task.WhenAll(t1, t2);
```

17.4.10 Aufgaben abbrechen

Eine ungestört ausgeführte Aufgabe endet mit der zugrundeliegenden Delegatenmethode. Vielleicht wird eine Aufgabe aber nicht mehr benötigt, weil z. B. der Benutzer auf einen zugehörigen **Abbrechen**-Schalter geklickt hat. Die Klasse **Task** unterstützt das mit dem .NET Framework 4.0 eingeführte kooperative Abbruchmodell (engl.: *Cooperative Cancellation Framework*), dessen Anwendung auf Threads wir im Abschnitt 17.1.5.2 kennengelernt haben. Mit Hilfe der TPL-Typen **CancellationTokenSource** und **CancellationToken** kann eine Aufgabe aufgefordert werden, ihre Tätigkeit einzustellen. Ein Objekt der Klasse **CancellationTokenSource**

```
var cts = new CancellationTokenSource();
```

stellt über seine **Token**-Eigenschaft eine Instanz vom Strukturtyp **CancellationToken** zur Verwendung in einem Terminierungsverfahren zur Verfügung:

```
CancellationToken ct = cts.Token;
```

Dieses Token kann einer neuen Task per Konstruktor oder beim Starten über die Methoden **StartNew()** bzw. **Run()** zugeordnet werden, z. B.:

```
Task task = Task.Factory.StartNew(Punktieren, ct, ct);
```

Im Beispiel kommt die folgende Überladung der **TaskFactory**-Methode **StartNew()**

public Task StartNew(Action<Object> *action***, Object** *state***, CancellationToken** *token***)**

mit drei Parametern zum Einsatz:

- **Action<Object>** *action*
  Als erster Parameter wird eine Delegatenmethode mit dem Rückgabetyp **void** und einem Parameter vom Typ **Object** erwartet.
- **Object** *state*
  Beim Aufruf der Delegatenmethode wird dieses Parameterobjekt übergeben. Der zur Versorgung von Aufgaben mit beliebigen Daten gedachte Parameter (siehe Abschnitt 17.4.3) wird im Beispiel zur Übergabe der **CancellationToken**-Instanz verwendet.
- **CancellationToken** *token*
  Die Aufgabe wird mit der **CancellationToken**-Instanz assoziiert. Dies ist keine Voraussetzung für die kooperative Terminierung, hat aber wichtige Effekte auf das Verhalten der TPL:
  - Aufgabenstatus nach dem Abbrechen durch das Werfen einer **OperationCanceledException**
  - Reaktion auf ein schon vor dem Start gesetztes Terminierungssignal
  - Fortsetzungsaufgaben mit der Bedingung **OnlyOnCanceled**

  Die Details werden gleich erläutert.

Um alle ein **CancellationToken** als Abbruchsignal beachtende Aufgaben zu stoppen, ruft man die **CancellationTokenSource** - Methode **Cancel()** auf:

```
cts.Cancel();
```

Daraufhin wird die **IsCancellationRequested**-Eigenschaft der **CancellationToken**-Instanz auf den Wert **true** gesetzt, und jede betroffene Aufgabe sollte Ihre Tätigkeit einstellen. Eine Aufgabe kann sich durchaus betroffen fühlen und das Signal beachten, obwohl ihr das **CancellationToken** *nicht* zugeordnet worden war.

Bei näherer Betrachtung ist die Realisation von **CancellationToken** als Werttyp irritierend. Weil eine Methode per Wertparameter von einer Strukturinstanz keine Referenz erhält sondern eine Kopie, drängt sich die Frage auf, ob sich eine Änderung der Eigenschaft **IsCancellationRequested** auf alle Kopien auswirkt. Die beruhigende Antwort lautet: *Ja*. Alle Kopien der von einem **CancellationTokenSource**-Objekt abstammenden **CancellationToken**-Instanz kennen nämlich das **CancellationTokenSource** - *Objekt* und fragen dort nach dem Status einer Unterbrechungsanforderung, wie der BCL-Quellcode der Struktur **CancellationToken** zeigt:[1]

```
private readonly CancellationTokenSource _source;
public bool IsCancellationRequested => _source != null &&
                                       _source.IsCancellationRequested;
```

Ein **CancellationToken** mit einer Aufgabe zu assoziieren, ist weder eine Voraussetzung noch eine Garantie dafür, dass die Aufgabe auf ein Terminierungssignal reagiert, hat aber einen Einfluss auf den Aufgabenstatus nach dem Abbruch. Reagiert eine Aufgabe mit dem empfohlenen Aufruf der **CancellationToken**-Methode **ThrowIfCancellationRequested()** (siehe unten) dann ...

[1] https://source.dot.net/

- hat sie anschließend den Status **Canceled**, wenn ihr das verwendete **CancellationToken** zugeordnet worden war,
- hat sie anschließend den Status **Faulted**, wenn ihr das verwendete **CancellationToken** *nicht* zugeordnet worden war.

Seit dem .NET Framework 4.5 besitzt die Klasse **CancellationTokenSource** eine Konstruktorüberladung mit Timeout-Parameter:

public CancellationTokenSource(int *millisecondsTimeout***)**

Mit dem zugehörigen Token assoziierte Aufgaben erhalten automatisch ein Abbruchsignal, wenn seit dem Konstruktor-Aufruf die angegebene Wartezeit vergangen ist.

Ist das Terminierungssignal schon vor Beginn der Bearbeitung gesetzt worden (z. B. während des Wartens auf andere Aufgaben), dann wird eine mit dem Signal assoziierte Aufgabe durch die TPL-Logik vom Zustand **WaitingToRun** sofort in den Zustand **Canceled** befördert, ohne je den Zustand **Running** erlebt zu haben. Eine Aufgabe, die nicht mit dem Signal assoziiert ist, das Signal aber kennt und beachtet, wird hingegen starten und sich nach dem ersten Zugriff auf das Signal wieder beenden. Je nach Terminierungsverfahren befindet sich eine solche Aufgabe dann im Zustand **RanToCompletion** (nach dem Verlassen der Aufgabenmethode per **return**) oder **Faulted** (nach dem Abbruch per Ausnahme).

Eine durch den bereits vorgestellten **StartNew()** - Aufruf

```
Task task = Task.Factory.StartNew(Punktieren, ct, ct);
```

erstellte und gestartete Aufgabe ist mit dem **CancellationToken** `ct` assoziiert. Außerdem erhält die zugehörige Methode `Punktieren()`, die den Delegatentyp **Action<Object>** erfüllt, das kooperativ zu beachtende Terminierungssignal per Parameter:

```
static void Punktieren(object token) {
   CancellationToken ct = (CancellationToken) token;
   Console.WriteLine("Aufgabe in Bearbeitung");
   for (int i = 0; i < maxIt; i++) {
      Thread.Sleep(500);
      Console.Write(".");
      if (ct.IsCancellationRequested) {
         Console.WriteLine("\nSignal zum Abbrechen erhalten\n");
         ct.ThrowIfCancellationRequested();
      }
   }
}
```

Die Methode `Punktieren()` prüft regelmäßig, ob die **IsCancellationRequested**-Eigenschaft der zuständigen **CancellationToken**-Instanz den Wert **true** besitzt. In diesem Fall stellt sie durch einen Aufruf der **CancellationToken**-Methode **ThrowIfCancellationRequested()** ihre Tätigkeit ein. Wie der Methodenname andeutet, wird eine Ausnahme geworfen, wobei die **Exception**-Spezialisierung **OperationCanceledException** Verwendung findet. Dem Ausnahmeobjekt wird das **CancellationToken** der Aufgabe mit auf den Weg gegeben, sodass der Ausnahmefänger den Grund der Terminierung ermitteln kann.

Weil die Aufgabe mit dem **CancellationToken** assoziiert ist, befindet sie sich anschließend im Zustand **Canceled**, sodass ihre Eigenschaft **IsCanceled** den Wert **true** besitzt. Außerdem werden Fortsetzungsaufgaben mit der Bedingung **OnlyOnCanceled** (siehe Abschnitt 17.4.8.1) gestartet.

Wer nicht die Methode **ThrowIfCancellationRequested()** aufrufen, sondern die **OperationCanceledException** direkt werfen möchte, muss dem Konstruktor das **CancellationToken** übergeben, damit die beschriebenen Folgen für assoziierte Aufgaben auftreten (Status **Canceled**, Start von Fortsetzungsaufgaben mit der Bedingung **OnlyOnCanceled**). Dass beim Aufruf der Methode **ThrowIfCancellationRequested()** eine erneute Überprüfung des Terminierungssignals (der Eigen-

schaft **IsCancellationRequested**) stattfindet, ist kein Grund gegen ihre Verwendung, weil diese Prüfung praktisch keinen Aufwand verursacht. Ein Aufruf der Methode **ThrowIfCancellationRequested()** ist praktisch immer gegenüber der direkt geworfenen **OperationCanceledException** zu bevorzugen.[1]

Es ist erforderlich, die einer Aufgabe zugrundeliegende Methode per Parameter über eine zu beachtende **CancellationToken**-Instanz zu informieren. Statt wie im Beispiel das **CancellationToken** über den **StartNew()** - Parameter *state* vom Typ **Object** in die Delegatenmethode vom Typ **Action<Object>** zu befördern, kann man eine Methode mit einem Parameter vom Typ **CancellationToken** definieren

```
static void Punktieren(CancellationToken ct) {
    Console.WriteLine("Aufgabe in Bearbeitung");
    for (int i = 0; i < maxIt; i++) {
        Thread.Sleep(500);
        Console.Write(".");
        if (ct.IsCancellationRequested) {
            Console.WriteLine("\nSignal zum Abbrechen erhalten\n");
            ct.ThrowIfCancellationRequested();
        }
    }
}
```

und bei der **Task**-Kreation per Lambda-Notation eine Delegatenmethode vom Typ **Action** realisieren, z. B.:

```
Task task = Task.Factory.StartNew(() => Punktieren(ct), ct);
```

Die Klasse **CancellationTokenSource** implementiert das Interface **IDisposable**, und Microsoft ermahnt dazu, für ein **CancellationTokenSource**-Objekt nach Gebrauch die **Dispose()** - Methode aufzurufen, um die belegten unverwalteten Ressourcen freizugeben.[2]

In der folgenden **Main()** - Methode wird eine Aufgabe basierend auf der Methode `Punktieren()` gestartet, und der Benutzer kann den Zeitpunkt des Abbruchs bestimmen. Anschließend wird die **Wait()** - Methode des **Task**-Objekts aufgerufen, um das **AggregateException**-Ausnahmeobjekt mit Detailinformationen über das Terminierungsverfahren abzufangen:

```
static void Main() {
    using var cts = new CancellationTokenSource();
    CancellationToken ct = cts.Token;

    Task task = Task.Factory.StartNew(Punktieren, ct, ct);
    Console.WriteLine("Aufgabe gestartet. Zustand:     " + task.Status +
                      ".\nStoppen mit Enter\n");
    Console.ReadLine();

    Console.WriteLine("\nZustand der Aufgabe vor Cancel: " + task.Status);
    cts.Cancel();
    Console.WriteLine("Signal zum Abbrechen gesetzt.\n");
```

[1] https://docs.microsoft.com/de-de/dotnet/standard/threading/how-to-listen-for-cancellation-requests-by-polling

[2] https://docs.microsoft.com/de-de/dotnet/api/system.threading.cancellationtokensource

```
        try {
            task.Wait();
        } catch (AggregateException ae) {
           foreach (Exception ie in ae.InnerExceptions)
                Console.WriteLine("Innere Ausnahme: " + ie.GetType());
        }
        Console.WriteLine("Aufgabe abgebrochen. Zustand:   " + task.Status);
    }
```

Beim folgenden Programmeinsatz hatte sich der Benutzer schon nach ca. 30 Sekunden an den Punkten satt gesehen:

```
Aufgabe in Bearbeitung
Aufgabe gestartet. Zustand:     WaitingToRun.
Stoppen mit Enter

.....................................................

Zustand der Aufgabe vor Cancel: Running
Signal zum Abbrechen gesetzt.

.
Signal zum Abbrechen erhalten

Innere Ausnahme: System.Threading.Tasks.TaskCanceledException
Aufgabe abgebrochen. Zustand:   Canceled
```

In der **AggregateException**, die der zum Abbruch auffordernde und mit **Wait()** auf die Beendigung der Aufgabe wartende Thread nach dem Abbruch sieht, steckt *keine* **OperationCanceledException**, sondern ein Objekt der von **OperationCanceledException** abgeleiteten Klasse **TaskCanceledException** (siehe Ausgabe). Dafür sorgt das TPL-Framework, wenn ...

- Aufgaben-intern eine **OperationCanceledException** geworfen wird,
- in diesem Ausnahmeobjekt das mit der Aufgabe assoziierte **CancellationToken** enthalten ist,
- und die Eigenschaft **IsCancellationRequested** des enthaltenen Tokens den Wert **true** besitzt.

Das TPL-Framework geht dann von einer erfolgreichen Terminierung aus, und die **Exception**-Eigenschaft der Aufgabe besitzt den Wert **null**.

Wenn eine Aufgabe wegen einer Ausnahme endet, und die drei genannten Bedingungen nicht alle erfüllt sind, dann ...

- hat die Aufgabe den Status **Faulted**,
- steckt in der beim **Wait()** - Aufruf geworfenen **AggregateException** keine innere Ausnahme vom Typ **TaskCanceledException**,
- zeigt die **Exception**-Eigenschaft der Aufgabe auf die **AggregateException**.

Wenn das den Abbruch signalisierende Token nicht mit der Aufgabe assoziiert ist, dann steckt in der **AggregateException** die ursprünglich geworfene **OperationCanceledException**, die nun wie eine gewöhnliche Ausnahme behandelt wird.[1]

Eine sinnvolle Reaktion auf eine Terminierungsaufforderung kann in einfachen Fällen auch schlicht darin bestehen, die Aufgabe zu beenden, also auf einen Aufruf der Methode **ThrowIfCancellationRequested()** zu verzichten. Die betroffene **Task**-Instanz wechselt dann nicht in den Zustand **Canceled**, sondern in den Zustand **RanToCompletion**.

---

[1] https://docs.microsoft.com/de-de/dotnet/standard/parallel-programming/task-cancellation

17.4.11 Aufgaben serienweise starten über die Klasse Parallel

Über die statische Methode **Invoke()** der Klasse **Parallel** aus dem Namensraum **System.Threading.Tasks** startet man eine Serie von Aufgaben. Man übergibt die Aufgaben per Serienparameter vom Typ **Action[]**, wobei **Action** bekanntlich ein Delegatentyp mit der folgenden Signatur ist:

public delegate void Action()

Im folgenden Programm erfüllt die Methode `SampleMean()`, die den Mittelwert aus 10 Millionen Pseudozufallszahlen berechnet und auf der Konsole protokolliert, den Delegatentyp **Action**:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks;

class ParallelInvoke {
 const int number = 16;
 const int sampleSize = 10_000_000;
 static HashSet<int> tids = new HashSet<int>();

 static void SampleMean() {
  var zzg = new Random();
  double erg = 0.0;
  for (int j = 0; j < sampleSize; j++)
   erg += zzg.NextDouble();
  Console.WriteLine("Stichprobenmittel " + (erg / sampleSize) +
            " berechnet von Thread " + Thread.CurrentThread.ManagedThreadId);
  lock (tids) {tids.Add(Thread.CurrentThread.ManagedThreadId);}
 }

 static void Main() {
  var av = new Action[number];
  for (int i = 0; i < number; i++)
   av[i] = new Action(SampleMean);
  var stopWatch = new Stopwatch();
  stopWatch.Start();
  try {
   Parallel.Invoke(av);
  } catch (Exception e) {
   Console.WriteLine(e);
  }
  stopWatch.Stop();
  Console.WriteLine("\nBenötigte Zeit: " + stopWatch.Elapsed +
                    "\nAnzahl eingesetzter Threads: " + tids.Count);
 }
}
```

In der Methode **Main()** wird ein **Action**-Array mit Delegatenobjekten gefüllt, deren Aufgabe darin besteht, die Methode `SampleMean()` auszuführen. Per **Invoke()** - Aufruf werden **Task**-Objekte erstellt und durch eine geeignete Anzahl von Threads ausgeführt.

Dass ein **Invoke()** - Aufruf aus jedem Delegatenobjekt im Parameter-Array eine Aufgabe erstellt und startet, bleibt im Beispielprogramm ohne großen Nutzen, weil in der zuvor ausgeführten **for**-Schleife statt der Kreation der Delegatenobjekte auch ein Start der Aufgaben mit Hilfe der **Task**-Methode **Run()** hätte erfolgen können.

Ein **Invoke()** - Aufruf startet die Aufgaben nicht nur, sondern wartet auch auf deren Fertigstellung, denn der Aufruf endet erst dann, wenn alle Aufgaben abgeschlossen sind.

Es ist keine Reihenfolge der Bearbeitung garantiert. Wenn in mindestens einer Aufgabe eine Ausnahme auftritt, dann endet der **Invoke()** - Aufruf mit einer **AggregateException**.

Beim anschließend protokollierten Programmeinsatz auf einem Rechner mit vier virtuellen Kernen (Intel Core i3 mit Hyper-Threading) sind 16 **Task**-Objekte auf 5 Threads verteilt worden:

```
Stichprobenmittel 0,500100023702583 berechnet von Thread 1
Stichprobenmittel 0,500022002839959 berechnet von Thread 3
Stichprobenmittel 0,499972074918739 berechnet von Thread 4
Stichprobenmittel 0,500067046997573 berechnet von Thread 6
Stichprobenmittel 0,499972074918739 berechnet von Thread 5
Stichprobenmittel 0,499940154294905 berechnet von Thread 3
Stichprobenmittel 0,499940154294905 berechnet von Thread 1
Stichprobenmittel 0,500018147236211 berechnet von Thread 4
Stichprobenmittel 0,499991719314938 berechnet von Thread 5
Stichprobenmittel 0,500018147236211 berechnet von Thread 6
Stichprobenmittel 0,500018468337515 berechnet von Thread 3
Stichprobenmittel 0,500129547474901 berechnet von Thread 1
Stichprobenmittel 0,49994795555673 berechnet von Thread 4
Stichprobenmittel 0,500155598691146 berechnet von Thread 5
Stichprobenmittel 0,499928677828538 berechnet von Thread 6
Stichprobenmittel 0,499840447713619 berechnet von Thread 3

Benötigte Zeit: 00:00:01.1572925
Anzahl eingesetzter Threads: 5
```

Um die Zahl der beteiligten Threads zu ermitteln, werden Kennungen über die **Thread**-Eigenschaft **ManagedThreadId** bestimmt und in einem **HashSet<int>** - Objekt gesammelt. Die aus mehreren Threads stammenden Zugriffe auf das Kollektionsobjekt werden per **lock**-Anweisung synchronisiert (vgl. Abschnitt 17.1.4.1.1).[1]

Zur Zeitmessung verwenden wir die Klasse **Stopwatch** aus dem Namensraum **System.Diagnostics**, die im Vergleich zur bisher meist verwendeten Klasse **DateTime** mehr Komfort bietet.

Muss eine Aufgabe für eine Folge von Indexwerten, aber nicht unbedingt in der natürlichen Reihenfolge, ausgeführt werden, dann erlaubt die statische **Parallel**-Methode **For()** eine Parallelisierung. Es ist eine Methode zu definieren, die den Delegatentyp **Action<int>** erfüllt, also von der folgenden Bauart ist:

public void Action(**int** *index***)**

Ein auf dieser Methode basierendes Delegatenobjekt wird zusammen mit einem Start- und einem Endwert für den Schleifenindex als Parameter an die Methode **For()** übergeben, z. B.:

```
Parallel.For(1, number, SampleMean);
```

Die Delegatenmethode wird für jeden Indexwert aufgerufen und erhält diesen als Aktualparameter. Man darf sich aber nicht darauf verlassen, dass die Aufrufe in der Reihenfolge der Indexwerte stattfinden.

Die folgende Variante des obigen Programms zeigt den **For()** - Aufruf im Kontext:

```
using System;
using System.Threading;
using System.Threading.Tasks;
using System.Collections.Generic;
using System.Diagnostics;
```

[1] Im Namensraum **System.Collections.Concurrent** mit Thread-sicheren Kollektionsklassen fehlt leider eine für den Multithreading-Betrieb geeignete Alternative zu **HashSet<T>**. Man kann eine solche Klasse aus der vorhandenen Klasse **ConcurrentDictionary<TKey, TValue>** erstellen.

```
class ParallelFor {
   const int number = 16;
   const int sampleSize = 10_000_000;
   static HashSet<int> tids = new HashSet<int>();

   static void SampleMean(int i) {
      var zzg = new Random(i);
      double erg = 0.0;
      for (int j = 0; j < sampleSize; j++)
         erg += zzg.NextDouble();
      Console.WriteLine($"Mittelwert {(erg / sampleSize), 8:f5} mit Index {i, 2} " +
                        $"berechnet von Thread {Thread.CurrentThread.ManagedThreadId}");
      lock (tids) {tids.Add(Thread.CurrentThread.ManagedThreadId);}
   }

   static void Main() {
      var stopWatch = new Stopwatch();
      try {
         stopWatch.Start();
         ParallelLoopResult plr = Parallel.For(1, number, SampleMean);
         stopWatch.Stop();
         Console.WriteLine("\nParallelLoopResult fertiggestellt: " + plr.IsCompleted +
                           "\nBenötigte Zeit: " + stopWatch.Elapsed +
                           "\nAnzahl eingesetzter Threads: " + tids.Count);
      } catch (Exception e) {
         Console.WriteLine(e);
      }
   }
}
```

Zur Berechnung von 16 Mittelwerten aus jeweils 10.000.000 Zahlen werden 5 Threads verwendet:

```
Mittelwert  0,49984  mit Index 10 berechnet von Thread 6
Mittelwert  0,49978  mit Index  1 berechnet von Thread 1
Mittelwert  0,50004  mit Index  4 berechnet von Thread 4
Mittelwert  0,49989  mit Index  7 berechnet von Thread 5
Mittelwert  0,50000  mit Index 13 berechnet von Thread 7
Mittelwert  0,50015  mit Index  2 berechnet von Thread 1
Mittelwert  0,50004  mit Index 11 berechnet von Thread 6
Mittelwert  0,49996  mit Index 14 berechnet von Thread 7
Mittelwert  0,50014  mit Index  5 berechnet von Thread 4
Mittelwert  0,49991  mit Index  8 berechnet von Thread 5
Mittelwert  0,50019  mit Index  3 berechnet von Thread 1
Mittelwert  0,49984  mit Index 15 berechnet von Thread 7
Mittelwert  0,49996  mit Index  6 berechnet von Thread 4
Mittelwert  0,49999  mit Index 12 berechnet von Thread 6
Mittelwert  0,49997  mit Index  9 berechnet von Thread 5

ParallelLoopResult fertiggestellt: True
Benötigte Zeit: 00:00:01.0703399
Anzahl eingesetzter Threads: 5
```

Ein **For()** - Aufruf endet, wenn alle Aufgaben erledigt sind. Wenn in mindestens einer Aufgabe eine Ausnahme auftritt, dann endet der **For()** - Aufruf mit einer **AggregateException**.

Die **Parallel**-Methode **ForEach()** erstellt eine Serie von **Task**-Objekten aus einer Methode (bzw. aus einem Delegatenobjekt) und passenden Argumenten in einer Kollektion. Zu der folgenden Überladung existieren mehrere Alternativen mit zusätzlichen Parametern:

public static ParallelLoopResult ForEach<TSource>(IEnumerable<TSource> *source*, **Action<TSource>** *action***)**

Wir gelangen auf einfache Weise zu einem Demonstrationsbeispiel für die **Parallel**-Methode **ForEach()**, indem wir im letzten Programm den **For()** - Aufruf mit Start- und Stoppindex

```
ParallelLoopResult plr = Parallel.For(1, number, SampleMean);
```

ersetzen durch einen **ForEach()** - Aufruf mit einem **List<int>** - Parameterobjekt, das eine analoge Serie mit ganzen Zahlen enthält

```
ParallelLoopResult plr = Parallel.ForEach(iList, SampleMean);
```

Die als erster **ForEach()** - Aktualparameter geforderte Datenquelle ...

- muss die Schnittstelle **IEnumerable<TSource>** erfüllen,
- also eine Methode namens **GetEnumerator()** besitzen
- mit einem Enumerator als Rückgabe,
- der sukzessive Instanzen vom Typ **TSource** liefert.

Für jede **TSource**-Produktion des Enumerators startet die TPL eine Aufgabe basierend auf der im zweiten **ForEach()** - Aktualparameter angegebenen Methode, die den jeweiligen **TSource**-Wert als Aktualparameter erhält. Man darf sich aber nicht darauf verlassen, dass die Aufgaben in der Enumerator-Reihenfolge gestartet werden.

Die Änderungen am Programm beschränken sich auf die Methode **Main()**:

```
static void Main() {
    var iList = new List<int>();
    for (int i = 1; i <= number; i++)
        iList.Add(i);
    var stopWatch = new Stopwatch();
    try {
        stopWatch.Start();
        ParallelLoopResult plr = Parallel.ForEach(iList, SampleMean);
        stopWatch.Stop();
        Console.WriteLine("\nParallelLoopResult fertiggestellt: " + plr.IsCompleted +
                          "\nBenötigte Zeit:\t" + stopWatch.Elapsed +
                          "\nAnzahl eingesetzter Threads:\t" + tids.Count);
    } catch (Exception e) {
        Console.WriteLine(e);
    }
}
```

Ein **ForEach()** - Aufruf endet, wenn alle Aufgaben erledigt sind.

Von den Methoden **Invoke()**, **For()** bzw. **ForEach()** wird ggf. eine **AggregateException** mit den bei der Aufgabenbearbeitung aufgetretenen unbehandelten Ausnahmen geworfen. Diese **AggregateException** sollte im Rahmen einer **try**-Anweisung abgefangen werden. Zur Analyse kann man die **AggregateException**-Methode **Handle()** verwenden (siehe Abschnitt 17.4.6.1.1).

17.5 C# - Sprachunterstützung für das asynchrone Programmieren

Obwohl die TPL im Vergleich zur direkten Nutzung der Klassen **Thread** oder **ThreadPool** einige Erleichterungen für die asynchrone Programmierung bietet, verbleibt doch viel Aufwand beim Programmierer bei der Realisierung einer Parallelverarbeitung. Durch die in C# 5.0 eingeführten Schlüsselwörter **async** und **await** soll die Nutzung der asynchronen Programmierung nochmals erleichtert werden, um ihre Verbreitung zu fördern. Unter Verwendung der neuen Schlüsselwörter müssen Programmierer beim Umstieg von synchronem auf asynchronen Code nur wenige Änderungen vornehmen, während der Compiler nach Möglichkeit für eine parallele Ausführung sorgt und dabei im Wesentlichen die TPL-Typen benutzt. Wer verstehen möchte, was der Compiler mit

dem Code anstellt, muss allerdings trotz der versprochenen Vereinfachung einige Arbeit investieren.[1]

Die folgenden Ziele sollen realisiert werden:

- Die für asynchrone Programmierung sorgende Syntax soll sich nur wenig von der synchronen Variante unterscheiden.
- Die verfügbaren Multithreading-Ressourcen (z. B. CPU-Kerne) sollen gut ausgenutzt werden.
- Die Fehlerbehandlung soll nicht über mehrere Methoden verteilt, sondern in *einer* Methode konzentriert werden.

Ein Hauptanwendungsfeld sind die Ereignisbehandlungsmethoden von GUI-Anwendungen, wobei zu den eben genannten generellen Zielen noch die folgenden speziellen Ziele hinzukommen:

- Der UI-Thread soll wenig belastet werden, damit die Bedienoberfläche flüssig reagiert.
- Von Hintergrund-Threads ermittelte Ergebnisse sollen bequem in den UI-Thread (also in Steuerelemente) kanalisiert werden.

Aber auch bei Server-basierten Anwendungen ist die C# - Sprachunterstützung der asynchronen Programmierung durch die Schlüsselwörter **async** und **await** von hoher Relevanz. Hier geht es darum, das Blockieren von Threads zu minimieren, um die Performanz zu steigern (siehe Griffiths 2013, S. 651ff).

17.5.1 Die Schlüsselwörter async und await

Durch den Modifikator **async** signalisiert eine Methode, ...

- dass sie asynchron (in einem anderen Thread auszuführende) Aufgaben starten wird,
- dass sie die Kontrolle an ihren Aufrufer zurückgeben wird, während sie auf die Fertigstellung von asynchron ausgeführten Aufgaben wartet,
- dass sie nach der Beendigung abzuwartender Aufgaben ihre Tätigkeit fortsetzen wird.

Eine asynchrone (also eine mit dem Modifikator **async** dekorierte) Methode darf den Operator **await** verwenden, um einen Suspendierungspunkt zu setzen. Der unäre **await**-Operator erhält als Operanden in der Regel eine Aufgabe, also ein Objekt vom Typ **Task** oder **Task<TResult>**.[2] Im Fall einer Aufgabe mit Ergebnis liefert der **await**-Operator einen Wert vom Ergebnistyp.

Per **await** wird der Compiler darüber informiert, dass die Methode nicht weiterarbeiten kann, bevor die Aufgabe im Operanden erfolgreich beendet wurde. Ist die abzuwartende Aufgabe noch nicht erledigt, wird die Kontrolle an den Aufrufer der asynchronen Methode zurückgegeben, sodass der aktuelle Thread nicht blockiert wird. Die asynchrone Methode ist aber nicht beendet, sodass z. B. vorhandene **finally**-Blöcke jetzt noch nicht ausgeführt werden. Stellt der **await**-Operator fest, dass die abzuwartende Aufgabe bereits erledigt ist, macht die asynchrone Methode weiter, ohne die Kontrolle an den Aufrufer zurückzugeben.

Der auf den **await**-Ausdruck folgende Rest der asynchronen Methode wird bei einem Kontrollwechsel zum Code einer **Fortsetzungsaufgabe**, die nach Beendigung der abzuwartenden Aufgabe ausgeführt wird. Per Voreinstellung wird dabei derselbe Thread verwendet, in dem die asynchrone Methode begonnen wurde.

---

[1] Das Motto „Magie, mach was du willst“ ist bei der Software-Entwicklung nicht ganz ungefährlich.

[2] Der **await**-Operand ist in der Regel ein **Task**- bzw. **Task<TResult>** - Objekt, doch ist von seinem Typ nur das Implementieren bestimmter Methoden bzw. Schnittstellen gefordert (siehe Abschnitt 17.5.4)

17.5.2 Anwendungsbeispiel

Die folgende, als **async** deklarierte Methode `GetHtmlTitleAsync()` ermittelt zu einer Webadresse das **title**-Element des dort abrufbaren HTML-Codes und liefert es als Zeichenfolge:[1]

```
static HttpClient client = new();
 . . .
private async Task<String> GetHtmlTitleAsync(String url) {
   string s = await client.GetStringAsync(url).ConfigureAwait(false);
   Match m = Regex.Match(s, @"<title>\s*(.+?)\s*</title>");
   if (m.Success)
      return m.Groups[1].Value;
   else
      return null;
}
```

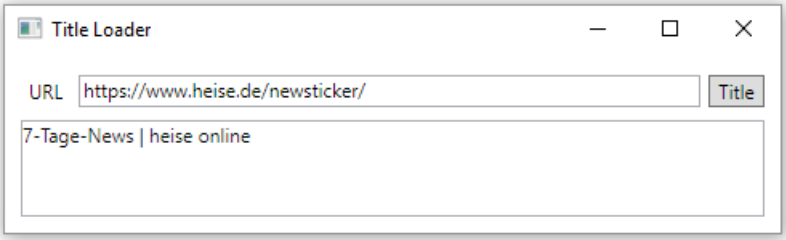
Über die Methode **GetStringAsync()** der Klasse **HttpClient** wird eine Aufgabe vom Typ **Task<String>** erstellt und gestartet, die asynchron (in einem Pool-Thread) den HTML-Code der Webseite beschafft:[2]

public Task<String> GetStringAsync (String *uri***)**

Durch Nutzung des **await**-Operators wartet `GetHtmlTitleAsync()` auf diese Aufgabe, ohne den aktuellen Thread zu blockieren.

Von den **return**-Anweisungen in `GetHtmlTitleAsync()` wird ein **String**-Objekt (oder **null**) geliefert, obwohl **Task<String>** als Rückgabetyp der Methode deklariert ist.

Die Methode `GetHtmlTitleAsync()` gehört zu einem Programm, das die Eingabe einer Webadresse erlaubt und auf Knopfdruck das zugehörige **title**-Element liefert:



Auch die Ereignisbehandlungsmethode zum Befehlsschalter des Programms besitzt den Modifikator **async**, weil sie per **await** auf die Ausführung der Methode `GetHtmlTitleAsync()` wartet, ohne den UI-Thread zu blockieren:

---

[1] In der Methode wird ein regulärer Ausdruck verwendet, der von der folgenden Webseite stammt: https://www.dotnetperls.com/title-html.

[2] Mit der Klasse **HttpClient** (aus dem Namensraum **System.Net.Http**) werden wir uns im Abschnitt 21.2 näher beschäftigen. Dort wird u. a. begründet, warum für eine Anwendung nur ein einziges **HttpClient**-Objekt erforderlich ist. Infolgedessen wird eine statische Referenzvariable verwendet und kein **Dispose()** - Aufruf vorgenommen. Im .NET Framework ist für die Verwendung der Klasse **HttpClient** ein Verweis auf das GAC-Assembly **System.Net.Http** erforderlich.

```
private async void ButtonTitle_Click(Object sender, RoutedEventArgs e) {
   String temp = null;
   try {
      buttonTitle.IsEnabled = false;

      Task<String> ts = GetHtmlTitleAsync(textBox.Text);

      temp = textBlock.Text;
      textBlock.Text = "Bitte warten ....";

      String s = await ts;
      if (s != null && s.Length > 0)
         textBlock.Text = s;
      else
         textBlock.Text = "Kein vorzeigbares Ergebnis";
   } catch (Exception ex) {
      textBlock.Text = temp;
      MessageBox.Show(this, ex.ToString(), ex.Message);
   } finally {
      buttonTitle.IsEnabled = true;
   }
}
```

Der **await**-Operator darf nur in einer Methode mit **async**-Deklaration verwendet werden.

Das Beispiel demonstriert eine typische Vorgehensweise bei der asynchronen Programmierung in WPF-Anwendungen mit Hilfe der Schlüsselwörter **async** und **await**:

- Die im UI-Thread ausgeführte Methode `ButtonTitle_Click()` deaktiviert zunächst den Befehlsschalter, damit der Benutzer während der Auftragsbearbeitung keinen neuen Auftrag erteilen kann. Dann wird die asynchrone Methode `GetHtmlTitleAsync()` aufgerufen, deren Anfang (der synchrone Teil) ebenfalls im UI-Thread ausgeführt wird.
- `GetHtmlTitleAsync()` veranlasst durch einen Aufruf der **HttpClient**-Methode **GetStringAsync()** die Erstellung und Ausführung einer Aufgabe vom Typ **Task<String>**. Diese von einem Pool-Thread ausgeführte Aufgabe beschafft den HTML-Code der Webseite. `GetHtmlTitleAsync()` wartet per **await**-Operator auf das Ergebnis, sodass die Kontrolle zur Methode `ButtonTitle_Click()` zurückkehrt.
- Hier werden (im UI-Thread) Arbeiten erledigt, die parallel zum Internet-Zugriff durch **GetStringAsync()** ablaufen: Der bisherige Inhalt eines **TextBlock**-Steuerelements wird gesichert und durch eine Warteinstruktion ersetzt.
- Jetzt benötigt `ButtonTitle_Click()` das Resultat von `GetHtmlTitleAsync()` und wartet daher auf die Fertigstellung der zugehörigen Aufgabe:
  ```
  String s = await ts;
  ```
- Die Kontrolle wird an den Aufrufer von `ButtonTitle_Click()` abgegeben, sodass weitere Arbeiten in der Nachrichtenwarteschlange des UI-Threads erledigt werden können. Es ist z. B. möglich, das Fenster des Programms zu verschieben.
- Nachdem **GetStringAsync()** seine Aufgabe erledigt hat, wird `GetHtmlTitleAsync()` fortgesetzt. Weil für den **await**-Operanden mit `ConfigureAwait(false)` angeordnet worden war, dass der Synchronisierungskontext *nicht* konserviert werden soll (siehe Abschnitt 17.5.5),
  ```
  String s = await client.GetStringAsync(url).ConfigureAwait(false);
  ```
  läuft der Methodenrest *nicht* im UI-Thread ab, sondern in einem Pool-Thread.

- Nach der Beendigung von `GetHtmlTitleAsync()` wird das von dieser asynchronen Methode gelieferte **Task<String>** - Objekt auf den Zustand **RanToCompletion** gesetzt. Nun setzt `ButtonTitle_Click()` seine Tätigkeit im UI-Thread fort und zeigt den ermittelten Webseiten-Titel an, wenn ein vorzeigbares Ergebnis geliefert wurde.
- Alle Ausnahmen können in `ButtonTitle_Click()` behandelt werden, auch die von **GetStringAsync()** in einem anderen Thread (!) geworfenen Ausnahmen (siehe Abschnitt 17.5.6). Die **try**-Anweisung in `ButtonTitle_Click()` besitzt einen **finally**-Block, um den Befehlsschalter zu reaktivieren.

17.5.3 Rückgabetyp einer async-Methode und andere technische Details

Erlaubte Rückgabetypen für eine **async**-Methode sind:

- **Task**
  Dieser Typ eignet sich, wenn zu einer vormals synchronen Methode mit dem Rückgabetyp **void** eine asynchrone Variante erstellt wird. Damit kann die asynchrone Methode einen **await**-Operanden liefern und zum abzuwartenden Auftrag werden.
- **Task<TResult>**
  Dieser Typ eignet sich, wenn zu einer vormals synchronen Methode mit dem Rückgabetyp **TResult** eine asynchrone Variante erstellt wird. Im Vergleich zum Rückgabetyp **Task** hat der Aufrufer die zusätzliche Option, ein Ergebnis entgegenzunehmen.
- **void**
  Der Rückgabetyp **void** ist bei **async**-Methoden vor allem deshalb erlaubt, um Ereignisbehandlungsmethoden, für die meist der Rückgabetyp **void** vorgeschrieben ist, als **async** deklarieren zu können. Allerdings bestehen für eine solche **async**-Methode wesentliche Einschränkungen:[1]
  - Weil sie keinen **await**-Operanden liefert, kann keine abzuwartende Aufgabe entstehen.
  - Es kann kein Aufgabenstatus festgestellt, kein Resultat abgefragt und keine Fortsetzungsaufgabe definiert werden.
  - Die für **async**-Methoden vorgesehene Kanalisierung von Ausnahmen klappt nicht, weil bei Verwendung des Rückgabetyps **void** kein **Task**-Objekt existiert, dem die Ausnahme zugeordnet werden könnte.

  Daher sollten **async**-Methoden mit dem Rückgabetyp **void** nur zur Ereignisbehandlung verwendet werden.
- Typ mit realisiertem Awaiter-Muster
  Seit C# 7.0 ist jeder (z. B. selbst definierte) Typ erlaubt, der eine Methode namens **GetAwaiter()** besitzt, die ein Objekt der Klasse **TaskAwaiter<TResult>** liefert (siehe Abschnitt 17.5.4). Das leistet z. B. auch die Struktur **ValueTask<TResult>** im Namensraum **System.Threading.Tasks**.

Für eine asynchrone Methode mit dem Rückgabetyp **Task** ist keine **return**-Anweisung mit Rückgabe zu verwenden. Von den **return**-Anweisungen einer asynchronen Methode mit dem Rückgabetyp **Task<TResult>** ist ein Ausdruck vom Typ **TResult** zu liefern.

Weitere Details zu **async** und **await**:

---

[1] https://docs.microsoft.com/en-us/archive/msdn-magazine/2013/march/async-await-best-practices-in-asynchronous-programming

- Per Konvention endet der Name einer asynchronen Methode mit **Async**, sofern es sich nicht um eine Ereignisbehandlungsmethode handelt.
- Der Modifikator **async** hat keinen Einfluss auf die Signatur einer Methode.
- In einer mit **async** dekorierten Methode darf **await** nicht als Bezeichner verwendet werden, anderenorts aber schon. Es ist (wie auch **async**) ein kontextbezogenes Schlüsselwort (vgl. Abschnitt 4.1.5).
- In der Regel enthält eine asynchrone Methode einen oder mehrere **await**-Operatoren. Während die Verwendung des **await**-Operators in einer nicht mit **async** dekorierten Methode verboten ist und einen Compiler-Fehler nach sich zieht, hat eine **async**-Methode ohne **await**-Operator lediglich eine Compiler-Warnung zur Folge.
- Auch anonyme Methoden (mit traditioneller Syntax oder Lambda-Syntax) lassen sich als asynchron definieren.
- Konstruktoren sowie die Zugriffsmethoden von Eigenschaften und Indexern dürfen *nicht* als asynchron deklariert werden (Mössenböck 2019, S. 241).
- Lange durfte die **Main()** - Methode eines Programms nicht als asynchron deklariert werden. Dieses Verbot ist seit C# 7.1 aufgehoben, wenn der Rückgabetyp **Task** verwendet wird.
- Seit C# 6.0 darf **await** auch in **catch**- und **finally**-Blöcken verwendet werden.
- Eine asynchrone Methode darf keine **ref**- oder **out**-Parameter besitzen, denn die Methode kehrt ja in der Regel zum Aufrufer zurück, bevor sie vollständig ausgeführt worden ist. Ein **ref**- oder **out**-Parameter hätte eventuell zu diesem Zeitpunkt noch keinen Wert erhalten.
- Wie bei jeder anderen Nutzung des Threadpools ist zu beachten, dass hier Hintergrund-Threads tätig sind, die abgebrochen werden, wenn der letzte Vordergrund-Thread eines Programms endet (vgl. Abschnitt 17.2). Damit daraus kein Problem resultiert, muss ein Vordergrund-Thread (z. B. per **await**) auf die Beendigung der asynchronen Methode warten.

Bislang haben wir per **await**-Operator auf *eine* asynchron ausgeführte Aufgabe gewartet. Über die statischen Methoden **WhenAll**() und **WhenAny**() der Klasse **Task** kann man eine Metaaufgabe erstellen und somit *mehrere* Aufgaben auf die Beobachtungsliste setzen (siehe Abschnitt 17.4.9).

17.5.4 Code-Transformation durch den Compiler

In der Regel liefert eine asynchrone Methode ein **Task** - oder **Task<TResult>** - Objekt zurück, z. B. die Methode **GetStringAsync()** der BCL-Klasse **HttpClient** im Namensraum **System.Net.Http**:

public async Task<String> GetStringAsync(String *uri***)**

Im Beispiel von Abschnitt 17.5.2 ruft die per Nachrichtenwarteschlange gestartete asynchrone Methode `ButtonTitle_Click()`

```
private async void ButtonTitle_Click(Object sender, RoutedEventArgs e) { ... }
```

die asynchrone Methode `GetHtmlTitleAsync()` auf:

```
private async Task<String> GetHtmlTitleAsync(String url) { ... }
```

In `GetHtmlTitleAsync()` fungiert ein Aufruf der asynchronen Methode `GetStringAsync()` als **await**-Operand:[1]

```
String s = await client.GetStringAsync(url).ConfigureAwait(false);
```

Der Compiler nimmt Code-Transformationen nach dem **await**-Muster vor (vgl. Griffiths 2013, S. 662ff), um möglichst viel Parallelität zu ermöglichen. Dazu muss der Typ des **await**-Operanden

---

[1] Man könnte eine Verschachtelungsebene einsparen und `GetStringAsync()` direkt in `ButtonTitle_Click()` aufrufen. Dann würden allerdings mehr Verarbeitungsschritte im UI-Thread stattfinden, und es könnten weniger Details des **await**-Musters demonstriert werden.

bestimmte Verhaltenskompetenzen besitzen. Diese Kompetenzen sind in den Klassen **Task** und **Task<TResult>** vorhanden, haben aber bei der Behandlung der TPL im Abschnitt 17.4 keine Rolle gespielt.

Der Compiler fordert beim **await**-Operanden durch einen Aufruf seiner Methode **GetAwaiter()** ein Objekt der Klasse **TaskAwaiter** bzw. **TaskAwaiter<TResult>** (aus dem Namensraum **System.Runtime.CompilerServices**) an, das im weiteren Verlauf als *Awaiter* bezeichnet werden soll. Beim Awaiter erkundigt sich der Compiler über die Eigenschaft **IsCompleted**, ob die abzuwartende Aufgabe bereits abgeschlossen ist. Bei einer positiven Auskunft wird die Methode fortgesetzt. Bei der Methode `GetHtmlTitleAsync()` würde in einem solchen Fall der komplette Rest ausgeführt, weil dort kein weiterer **await**-Operator vorhanden ist. Allerdings wird der Fall kaum auftreten, weil das zugrundeliegende **Task<String>** - Objekt im **await**-Operanden gerade erst erstellt und gestartet worden ist.

Hat **IsCompleted** den Wert **false**, gibt die Methode `GetHtmlTitleAsync()` die Kontrolle an ihren Aufrufer `ButtonTitle_Click()` zurück. Der Compiler erstellt eine Callback-Methode, die den Rest von `GetHtmlTitleAsync()` erledigt und übergibt diese als Delegaten-Aktualparameter an die Methode **OnCompleted()** des Awaiters.

Die für den Rest von `GetHtmlTitleAsync()` zuständige Delegatenmethode wird per Voreinstellung im selben Synchronisierungskontext (vgl. Abschnitt 17.4.7) ausgeführt wie die Methode **OnCompleted()**, was z. B. die Aktualisierung von Steuerelementen nach Beendigung einer Hintergrundaufgabe vereinfacht. Weil es im Beispiel nicht erforderlich bzw. wünschenswert ist, dass die Fortsetzungsmethode im selben Synchronisierungskontext (hier: UI-Kontext) läuft, wird über die **Task<TResult>** - Methode **ConfigureAwait()** dafür gesorgt, dass die Fortsetzung in einem Pool-Thread stattfindet.

Wenn die Methode `GetHtmlTitleAsync()` auf das Ende von `GetStringAsync()` wartet und die Kontrolle an ihren Aufrufer `ButtonTitle_Click()` zurückgibt, werden dort Arbeiten ausgeführt, die parallel zur Beschaffung der HTML - Titelzeile ablaufen können:

```
temp = textBlock.Text;
textBlock.Text = "Bitte warten ....";
```

Sobald in `ButtonTitle_Click()` keine vor Ankunft der Titelzeile zu erledigenden Arbeiten mehr anstehen, begibt sich diese Methode in den Wartezustand, ohne den UI-Thread zu blockieren:

```
String s = await ts;
```

Die Kontrolle kehrt nun zur Nachrichtenwarteschlangenverwaltung zurück, sodass andere Aufgaben erledigt werden können (z. B. eine Verschiebung des Fensters). Der Compiler richtet auch für den Rest der Methode `ButtonTitle_Click()` eine Fortsetzungsaufgabe ein, die nach Fertigstellung des **Task<String>** - Objekts ausgeführt wird. Dass die Fortsetzung per Voreinstellung im selben Synchronisierungskontext (also im UI-Kontext) erfolgt, ist diesmal sinnvoll und sogar erforderlich.

Der **await**-Operator lässt sich als Option zur bequemen Vereinbarung einer Aufgabenfortsetzung auffassen.

Wie das folgende Beispiel zeigt, kann die Aufforderung **GetAwaiter()** auch vom Programmierer an ein **Task**- oder ein **Task<TResult>** - Objekt gerichtet werden, um eine Fortsetzungsmethode einzurichten:

```
private void Button_Click(object sender, RoutedEventArgs e) {
    label.Content = "";
    Task<double> task = Task.Factory.StartNew<double>(SampleMean);
    var awaiter = task.GetAwaiter();
    awaiter.OnCompleted(() => {
        try {
            label.Content = awaiter.GetResult().ToString();
        } catch (Exception ex) {
            label.Content = ex.Message;
        }
    });
}
```

In der BCL-Dokumentation wird allerdings davon abgeraten:[1]

> This method is intended for compiler use rather than for use in application code.

Wie das Beispiel zeigt, kann die an **OnCompleted()** übergebene Delegatenmethode über die Awaiter-Methode **GetResult()** auf das Ergebnis der gerade erledigten Aufgabe zugreifen. Eine in der asynchron ausgeführten Aufgabe aufgetretene und nicht behandelte Ausnahme wird beim Ergebniszugriff mit **GetResult()** neu geworfen.

Der tatsächlich vom Compiler erstellte Code ist u. a. deswegen ziemlich kompliziert, weil eine asynchrone Methode *mehrere* **await**-Ausdrücke enthalten kann.

17.5.5 Thread-Affinität

Eine asynchrone Methode startet (wie jede andere Methode) synchron im aktuellen Thread. Wenn **await** auf ein **Task**- bzw. **Task<TResult>** - Objekt wartet, wird per Voreinstellung der beim Starten der asynchronen Methode aktuelle Synchronisierungskontext (der Wert von **SynchronizationContext.Current**) konserviert und (bei einem Wert ungleich **null**) vor der Ausführung einer Fortsetzungsmethode restauriert. Ist der Wert von **SynchronizationContext.Current** vor **await** ungleich **null** (z. B. bei einer asynchronen Ereignisbehandlungsmethode in einem GUI-Framework), dann läuft die Fortsetzungsmethode im selben Synchronisierungskontext. Ist der Wert von **SynchronizationContext.Current** vor **await** hingegen gleich **null**, dann läuft die Fortsetzungsmethode in einem Pool-Thread, zur Vermeidung von überflüssigem Aufwand meist im selben, der die abgewartete Aufgabe erledigt hat.

In einer WPF-Ereignisbehandlungsmethode sorgt die Thread-Restaurierung dafür, dass die nach einer asynchron erledigten Aufgabe fälligen Änderungen von Steuerelementen im UI-Thread ablaufen. Während wir im Abschnitt 17.2.3 über die Threadpool-Nutzung in einer WPF-Anwendung einige Mühe hatten, um beim RSS-Feed-Reader sicherzustellen, dass alle Zugriffe auf Steuerelemente im UI-Thread stattfinden, wird sich im Abschnitt 17.5.7 beim selben Beispiel zeigen, dass die asynchrone Programmierung mit **async** und **await** speziell bei der GUI-Aktualisierung erheblich bequemer ist.

Wenn allerdings *keine* Notwendigkeit dafür besteht, eine Fortsetzungsmethode im ursprünglichen Synchronisierungskontext auszuführen, dann ist der voreingestellte Kontexttransfer *nicht* sinnvoll. Es könnte eine Flaschenhalskonstruktion provoziert und speziell der UI-Thread behindert werden. Soll z. B. ein per Befehlsschalter initiierter HTTP-Transfer letztlich zu einer Dateiausgabe (statt zu einer GUI-Veränderung) führen, dann ist es sinnvoll, durch einen **ConfigureWait()** - Aufruf mit dem Aktualparameter **false** an das per **GetStreamAsync()** angeforderte **Task<Steam>** - Objekt die Verbindung mit dem UI-Thread zu kappen, z. B.:

```
System.IO.Stream str = await client.GetStreamAsync(url).ConfigureAwait(false);
```

---

[1] https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.getawaiter

Im Ergebnis wird die Fortsetzungsmethode in einem Pool-Thread ausgeführt.

17.5.6 Unbehandelte Ausnahmen in abzuwartenden Aufgaben

Das bei der C# - Spracherweiterung um die Schlüsselwörter **async** und **await** angestrebte Ziel, bei asynchronem Code mit wenig Abweichungen von der synchronen Variante auszukommen, wird auch bei der Behandlung von Ausnahmen realisiert. Wenn eine **async**-Methode namens `M1()` per **await**-Ausdruck auf eine noch nicht abgeschlossene Aufgabe wartet, dann verschwindet sie vom Stack und gibt die Kontrolle an ihren Aufrufer zurück. Tritt nun in der abzuwartenden Aufgabe eine unbehandelte Ausnahme auf, besteht die Herausforderung für den Compiler darin, diese Ausnahme in dem per Callback ausgeführten Rest von `M1()` sichtbar zu machen. Genau dort ist nämlich im Normalfall die Ausnahmebehandlung kodiert.

Ist eine abzuwartende Aufgabe beendet, wertet der **await**-Operator ihren Status aus und unterscheidet folgende Fälle:

- Die Aufgabe wurde normal beendet.
- Während der Auftragsbearbeitung ist ein unbehandelter Ausnahmefehler aufgetreten. In diesem Fall wird die Ausnahme vom **await**-Operator erneut ausgelöst. Das geschieht *ohne* die Verpackung in eine **AggregateException** (siehe Abschnitt 17.4.5).
- Der Auftrag wurde abgebrochen (siehe Abschnitt 17.4.10). In diesem Fall löst der **await**-Operator eine **OperationCanceledException** aus.

Damit kann die Ausnahme aus der Sicht des Programmierers genauso behandelt werden wie bei einer synchronen Lösung. Weitere Details zur Weiterleitung von unbehandelten Ausnahmen bei der asynchronen Programmierung mit **async** und **await** beschreiben Albahari & Johannsen (2020, S. 619f).

Es ist zu beachten, dass *alle* in einer **async**-Methode mit **Task**- bzw. **Task<TResult>** - Rückgabe geworfene unbehandelte Ausnahmen dadurch kommuniziert werden, dass der Status des Aufgabenobjekts auf **Faulted** gesetzt wird. Das passiert auch mit den im synchron ausgeführten Teil (vor dem ersten **await**) auftretenden unbehandelten Ausnahmen, die also *nicht* wie bei einer normalen Methode an den Aufrufer weitergeleitet werden (Griffiths 2013, S. 669). Die im aktuellen Abschnitt beschriebene Kommunikation von Ausnahmen stellt *kein* Problem dar, wenn eine **async**-Methode per **await**-Ausdruck in den Programmablauf einbezogen wird. Dann lassen sich dort aufgetretene unbehandelte Ausnahmen in der aufrufenden Methode abfangen.

Wenn man per **await** auf eine kombinierte, z. B. über die statische **Task**-Methode **WhenAll()** erstellte Aufgabe wartet (vgl. Abschnitt 17.4.9), dann können *mehrere* unbehandelte Ausnahmen auftreten, von denen **await** aber nur die erste weiterleitet. Lösungsmöglichkeiten für dieses Problem sind in Griffiths (2013, S. 670ff) zu finden.

17.5.7 RSS-Feed-Reader mit async und await

Der schon mehrfach als Beispiel verwendete RSS-Feed-Reader wurde im Abschnitt 17.2.3 wesentlich verbessert, indem das Laden der RSS-Datei aus dem Internet und die Aufbereitung der Listenelemente in einen Pool-Thread verlagert wurden. Um die Ergebnisse des Pool-Threads auf der Bedienoberfläche anzuzeigen, mussten wir uns mit dem Synchronisierungskontext beschäftigen und einigen Aufwand betreiben. Mit Hilfe der Schlüsselwörter **async** und **await** lässt sich der Aufwand reduzieren. In der folgenden Lösung wird das **List**`<RssItem>` - Kollektionsobjekt, das als Wert der **ListBox**-Eigenschaft **ItemsSource** dient, von der Methode `GetItemsSource` erstellt:

```
private List<RssItem> GetItemsSource(Object obj) {
 String url = (String)obj;
 XDocument feed = XDocument.Load(url);
 var items = new List<RssItem>();
 IEnumerable<XElement> xDocItems = feed.Descendants("item");
 RssItem rit;
 foreach (XElement item in xDocItems) {
  rit = new RssItem() {
   Title = item.Element("title").Value,
   Description = Regex.Replace(item.Element("description").Value, "<[^>]*>",
                               String.Empty, RegexOptions.IgnoreCase).Trim(),
   Url = item.Element("link").Value};
  items.Add(rit);
 }
 return items;
}
```

Aus dieser Methode, die den Delegatentyp **Func<Object, List<RssItems>>** erfüllt, entsteht ein Objekt der Klasse **Task<List<RssItem>>**. Das geschieht zu Beginn der Ereignisbehandlungsmethode `button_Click()` per **StartNew()** - Aufruf, sodass sofort ein Pool-Thread mit der Arbeit loslegt. Nachdem die als **async** deklarierte Methode `button_Click()` parallel einige Arbeiten erledigt hat, wartet sie per **wait**-Operation auf die Hintergrundaufgabe, ohne dabei den UI-Thread zu blockieren:

```
private async void button_Click(object sender, RoutedEventArgs e) {
 Task<List<RssItem>> gis = Task.Factory.StartNew<List<RssItem>>(GetItemsSource,
                                                                textBox.Text);
 var waitInfo = new List<RssItem>()
                { new RssItem() { Title = "Feed wird geladen. Bitte warten ..." } };
 listBox.ItemsSource = waitInfo;
 button.IsEnabled = false;
 try {
  listBox.ItemsSource = await gis;
 } catch (Exception ex) {
  listBox.ItemsSource = null;
  MessageBox.Show(this, ex.ToString(), ex.Message);
 } finally {
  button.IsEnabled = true;
 }
}
```

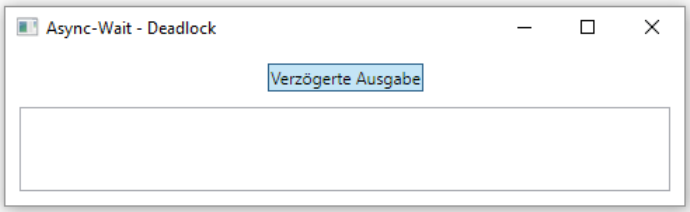
Nach dem Abschluss der Hintergrundaufgabe werden deren Ergebnisse im Rahmen einer Fortsetzungsaufgabe, die der Compiler zum Rest von `button_Click()` erstellt, ohne jede Vorkehrung korrekt im UI-Thread dargestellt. Die eventuell während der Aufgabenerledigung aufgetretenen unbehandelten Ausnahmen werden von **await** erneut geworfen, sodass die Behandlung in `button_Click()` erfolgen kann.

17.5.8 Tücken

Das von einer **async**-Methode gelieferte **Task**- oder **Task<TResult>** - Objekt sollte per **await**-Ausdruck in den Programmablauf integriert werden. Die **Wait()** - Methode oder die **Result**-Eigenschaft eines solchen Aufgabenobjekts zu verwenden, kann im UI-Synchronisierungskontext zum Deadlock führen, wie das folgende Beispiel nach Cleary (2014, S. 6) zeigt:

```
private async Task WaitAsync() {
   await Task.Delay(1000);
}

private void Button_Click(object sender, RoutedEventArgs e) {
   Task task = WaitAsync();
   task.Wait();
   text.Text = "Ready";
}
```

Nach einem Klick auf den Befehlsschalter



startet `Button_Click()` und ruft `WaitAsync()` auf. Dort wird mit **await** auf das Ablaufen einer durch die statische **Task**-Methode **Delay()** realisierten Verzögerung gewartet. Damit erhält `Button_Click()` die Kontrolle zurück und wartet mit **Wait()** auf die Beendigung des **Task**-Objekts zu `WaitAsync()`. Allerdings blockiert `Button_Click()` den UI-Thread. Nach Ablauf der **Delay()** - Zeit wartet die vom **await**-Operator erstellte Fortsetzungsmethode auf die Ausführung, kommt aber im UI-Thread nicht zum Zug.

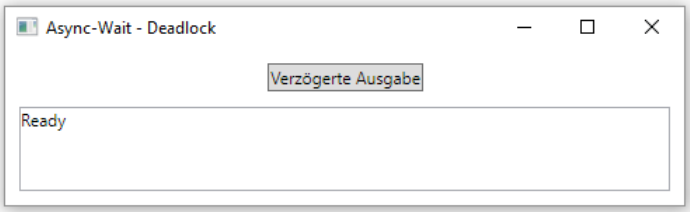
Zur Lösung des Problems bestehen zwei Optionen:

- In `WaitAsync()` wird veranlasst, dass die Fortsetzungsmethode vom Threadpool ausgeführt werden darf (vgl. Abschnitt 17.5.5):
  ```
  await Task.Delay(1000).ConfigureAwait(false);
  ```
- In `Button_Click()` wird per **await** - Operator auf das Ende von `WaitAsync()` gewartet:
  ```
  await task;
  ```
  Dann gibt `Button_Click()` im Wartezustand den UI-Thread frei, sodass die Fortsetzungsmethode zu `WaitAsync()` und danach die Fortsetzungsmethode zu `Button_Click()` ausgeführt werden können.

Mit dieser Lösung

```
private async Task WaitAsync() {
   await Task.Delay(1000);
}

private async void button_Click(object sender, RoutedEventArgs e) {
   await WaitAsync();
   text.Text = "Ready";
}
```

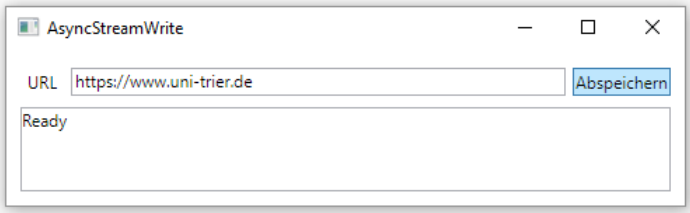
verhält sich das Programm sinnvoll:



Die statische **Task**-Methode **Delay()** ist übrigens in vielen Situationen gegenüber der **Thread**-Methode **Sleep()** zu bevorzugen. Als **await**-Operand macht sie es möglich, einen Algorithmus für eine Zeitspanne anzuhalten, ohne den Thread lahmzulegen.

17.5.9 Async-Methoden der Klasse Stream

Seit der .NET Framework - Version 4.5 unterstützt die Klasse **Stream** die Task Parallel Library (TPL) durch die Methoden **ReadAsync()** und **WriteAsync()**. Das folgende Programm



befördert auf Knopfdruck den Inhalt einer Webseite in eine lokale Datei.

Die Ereignisbehandlungsmethode zum Befehlsschalter ruft die asynchrone Methode `GetHtmlCodeAsync()` auf, die ein **Task**-Objekt als Rückgabe liefert. Dann führt die Ereignisbehandlungsmethode einige vom **Task**-Objekt unabhängige Arbeiten aus und wartet schließlich per **await**-Ausdruck auf die Fertigstellung der Aufgabe, ohne den UI-Thread zu blockieren:

```
private async void Button_Click(object sender, RoutedEventArgs e) {
   String temp = null;
   try {
      button.IsEnabled = false;
      Task ts = GetHtmlCodeAsync(textBox.Text);

      temp = result.Text;
      result.Text = "Bitte warten ....";

      await ts;
      if (ts.Status == TaskStatus.RanToCompletion)
         result.Text = "Ready";
   } catch (Exception ex) {
      result.Text = temp;
      MessageBox.Show(this, ex.ToString(), ex.Message);
   } finally {
      button.IsEnabled = true;
   }
}
```

In der asynchronen Methode `GetHtmlCodeAsync()` wird zunächst mit der **HttpClient**-Methode **GetStringAsync()** der HTML-Code asynchron bezogen und in einer **String**-Variablen gespeichert.[1] Anschließend wird aus der Zeichenfolge unter Verwendung der UTF-8 - Codierung ein **byte**-Array erstellt, der schließlich mit der asynchronen **Stream**-Methode **WriteAsync()** in eine Datei geschrieben wird:

[1] Mit der Klasse **HttpClient** (aus dem Namensraum **System.Net.Http**) werden wir uns im Abschnitt 21.2 näher beschäftigen. Dort wird u. a. begründet, warum für eine Anwendung nur ein einziges **HttpClient**-Objekt erforderlich ist. Infolgedessen wird eine statische Referenzvariable verwendet und kein **Dispose()** - Aufruf vorgenommen. Im .NET Framework ist für die Verwendung der Klasse **HttpClient** ein Verweis auf das GAC-Assembly **System.Net.Http** erforderlich.

```
static HttpClient client = new();
 . . .
private async Task GetHtmlCodeAsync(string uri) {
 try {
  string s = await client.GetStringAsync(uri).ConfigureAwait(false);
  using var stream = new FileStream("demo.html", FileMode.Create, FileAccess.Write,
                                    FileShare.None, 4096, useAsync: true);
  var utf8enc = new System.Text.UTF8Encoding();
  byte[] sb = utf8enc.GetBytes(s);
  await stream.WriteAsync(sb, 0, sb.Length).ConfigureAwait(false);
 } catch (Exception ex) {
  MessageBox.Show(this, ex.ToString(), ex.Message);
 }
}
```

Für die **Task<String>** - Rückgabe von **GetStringAsync()** sowie für die **Task**-Rückgabe von **WriteAsync()** wird per `ConfigureAwait(false)` angeordnet, dass die zugehörige Fortsetzungsmethoden *nicht* in dem beim Methodenstart gültigen Synchronisierungskontext (im Beispiel: UI-Thread) ausgeführt werden sollen (siehe Abschnitt 17.5.5).

Beim Erstellen eines **Stream**-Objekts ist darauf zu achten, dass die Fähigkeiten des Betriebssystems zur Delegation von Ein-/Ausgabeoperationen an die Gerätetreiber genutzt werden. Dann wird *kein* Pool-Thread benötigt und eine erhebliche Beschleunigung (bis zum Faktor 10) erzielt, wenn ausschließlich eine asynchrone Nutzung stattfindet. Allerdings sollten in diesem Modus keine synchronen Zugriffe stattfinden, weil dann statt einer Beschleunigung eine Verzögerung in derselben Größenordnung zu erwarten ist.[1] Bei der Klasse **FileStream** wird die asynchrone Ein-/Ausgabe auf Betriebssystemebene im Konstruktor (je nach Überladung) durch die Parameter **useAsync** oder **options** aktiviert, z. B.:

```
FileStream(name, FileMode.Create, FileAccess.Write, FileShare.None, 4096, useAsync: true)
```

Die oben vorgestellte Methode `GetHtmlCodeAsync()` dient u. a. zur Demonstration der asynchronen **Stream**-Methode **WriteAsync()**. Zur Lösung der Aufgabe, den HTML-Code einer Webseite in eine lokale Datei zu befördern, ist allerdings die asynchrone **Stream**-Methode **CopyToAsync()** besser geeignet, wie die folgende Variante der Methode `GetHtmlCodeAsync()` zeigt:

```
private async Task GetHtmlCodeAsync(string uri) {
 try {
  using Stream stream = await client.GetStreamAsync(uri).ConfigureAwait(false);
  using var fileStream = new FileStream("demo.html", FileMode.Create,
                            FileAccess.Write, FileShare.None, 4096, useAsync: true);
  await stream.CopyToAsync(fileStream).ConfigureAwait(false);
 } catch (Exception ex) {
  MessageBox.Show(this, ex.ToString(), ex.Message);
 }
}
```

Über die asynchrone **HttpClient**-Methode **GetStreamAsync()** wird der HTML-Code als Datenstrom bezogen. Dieser wird anschließend durch die asynchrone **Stream**-Methode **CopyToAsync()** in das **FileStream**-Objekt umgeleitet, also in die angebundene Datei geschrieben.

---

[1] https://docs.microsoft.com/de-de/dotnet/api/system.io.filestream.-ctor

17.6 Weitere Multithreading-Techniken

In diesem Abschnitt haben sich Techniken angesammelt, die ...

- als veraltet gelten, aber wegen ihrer großen Tradition und Verbreitung erwähnt werden sollen (APM, EAP),
- für neue Projekte relevant sind, aber aus Zeitgründen nur erwähnt werden können (PLINQ, TPL-Datenflussbibliothek, Thread-sichere Kollektionsklassen).

17.6.1 Asynchronous Programming Model (APM)

Das APM-Entwurfsmuster (*Asynchronous Programming Model*) wurde schon in .NET Framework 1.0 eingeführt und war lange die empfohlene Technik zur Verwendung von Pool-Threads. Im Vergleich zur direkten Verwendung der **Threadpool**-Methode **QueueUserWorkItem()** (siehe Abschnitt 17.2.1) bietet das APM u. a. die folgenden Vorteile:

- Statt der Einschränkung auf den Delegatentyp **WaitCallback** können Methoden mit Rückgabe per Pool-Thread ausgeführt werden.
- Der initiierende Thread kann sich über den Abschluss der Hintergrundtätigkeit informieren.

Mittlerweile erlaubt jedoch die im Abschnitt 17.4 vorgestellte Task Parallel Library (TPL) eine weitaus flexiblere Nutzung des Threadpools, die von C# durch die Schlüsselwörter **async** und **await** sehr gut unterstützt wird. Daher urteilt Microsoft über das APM:[1]

> This pattern is no longer recommended for new development.

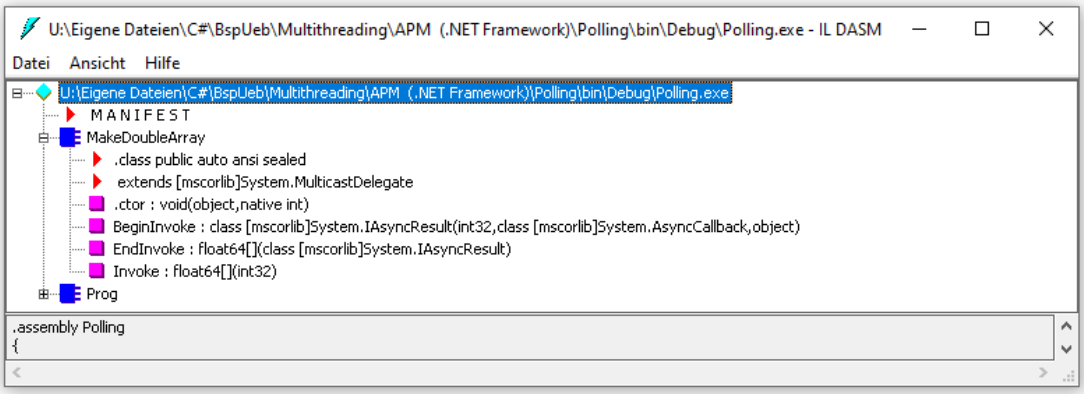
In .NET Core (also auch in .NET 5.0) wird das APM nicht mehr unterstützt, und ein Aufruf der zentralen Methode **BeginInvoke()** führt zu einer **PlatformNotSupportedException**. Weil in zu pflegenden .NET Framework - Bestandsprojekten APM-Lösungen für asynchrone Operationen noch anzutreffen sind, folgt eine kurze Beschreibung.

17.6.1.1 Asynchrone CPU-Nutzung

Zu jeder Delegatendefinition, z. B.:

```
public delegate double[] MakeDoubleArray(int anz);
```

erstellt der Compiler eine Klasse mit den Methoden **BeginInvoke()** und **EndInvoke()**, wie die folgende **ILDasm**-Ausgabe für den Typ `MakeDoubleArray` zeigt:



Um das recht komplexe Zusammenspiel verschiedener Methoden im APM anschaulich erläutern zu können, betrachten wir eine konkrete (nicht sonderlich sinnvolle) Methode, welche den Delegatentyp `MakeDoubleArray` erfüllt:

---

[1] https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/

```
static double[] RandomSums(int anz) {
   var erg = new double[anz];
   for (int i = 0; i < anz; i++) {
      for (int j = 0; j < 10_000_000; j++)
         erg[i] += zzg.Next(100);
   }
   return erg;
}
```

Sie liefert einen Array mit **double**-Elementen, die jeweils eine Summe von 10.000.000 Zufallszahlen enthalten. Über den Parameter wählt man die Anzahl der Array-Elemente. Wir bezeichnen `RandomSums()` später als *Arbeitsmethode*.

Wir erstellen aus `RandomSums()` ein Delegatenobjekt namens `rs` vom Typ `MakeDoubleArray` (gleich als *Arbeitsdelegat* bezeichnet) und beauftragen dieses Objekt, seine Methode **BeginInvoke()** auszuführen:

```
var rs = new MakeDoubleArray(RandomSums);
IAsyncResult ar = rs.BeginInvoke(3, null, null);
```

Als Rückgabewert liefert **BeginInvoke()** ein Objekt vom Typ **IAsyncResult**, das die asynchron ausgeführte Operation identifiziert. Wir bezeichnen es gleich als *Operationsobjekt*. Es …

- muss der Arbeitsdelegatenmethode **EndInvoke()** (siehe unten) als Parameter übergeben werden, um das Ergebnis der abgeschlossenen Operation zu erhalten, z. B. die **double[]** - Rückgabe der Arbeitsmethode):
  ```
  double[] da = rs.EndInvoke(ar);
  ```
- kann mit seiner booleschen Eigenschaft **IsCompleted** über den Abschluss der Arbeiten informieren.
- enthält ein Signalisierungsobjekt vom Typ **WaitHandle** (vgl. Abschnitt 17.1.4.3), das über die Eigenschaft **AsyncWaitHandle** angesprochen werden kann.

BeginInvoke() besitzt alle Parameter der Delegatendefinition (im Beispiel: einen Parameter vom Typ **int**) und außerdem am Ende seiner Parameterliste noch zwei Parameter mit den folgenden Typen:

- **AsyncCallback**
  Dieser Delegatentyp hat die folgende Signatur:

  public delegate void AsyncCallback(IAsyncResult *ar***)**

  Beim **BeginInvoke()** - Aufruf kann eine Rückrufmethode angegeben werden, die bei Beendigung der Arbeitsmethode aufgerufen wird und das Operationsobjekt als Parameter erhält. In der ersten Variante des Beispielprogramms wird noch auf eine Rückrufmethode verzichtet und dem **BeginInvoke()** - Aufruf **null** an Stelle eines Rückruf-Delegatenobjekts übergeben.
- **Object**
  Dieser **BeginInvoke()** - Parameter ist in der resultierenden **IAsyncResult**-Rückgabe (im Operationsobjekt) über die Eigenschaft **AsyncState** ansprechbar. Wir benutzen diesen Parameter später, um der Rückrufmethode den Arbeitsdelegaten bekannt zu machen. In der ersten Variante des Beispielprogramms wird auf diese Option verzichtet und dem **BeginInvoke()** - Aufruf **null** als dritter Parameter übergeben.

Durch den **BeginInvoke()** - Aufruf an den Arbeitsdelegaten gelangt über die **ThreadPool**-Methode **QueueUserWorkItem()** ein Arbeitsauftrag in die Warteschlange des Threadpools.

In der ersten Variante des Beispielprogramms stellen wir über die Eigenschaft **IsCompleted** des Operationsobjekts fest, ob der bearbeitende Hintergrund-Thread fertig ist:

```
int sek = 0;
while (!ar.IsCompleted) {
 Console.WriteLine("Warte seit {0} Sekunden auf den Hintergrund-Thread", sek++);
 Thread.Sleep(1000);
}
```

Alternativ lässt sich derselbe Zweck auch über einen **WaitOne()** - Aufruf an das im Operationsobjekt enthaltene **WaitHandle** erreichen:

```
int sek = 0;
while (!ar.AsyncWaitHandle.WaitOne(1000))
   Console.WriteLine("Warte seit {0} Sekunden auf den Hintergrund-Thread", ++sek);
```

Hat der Pool-Thread seine Arbeit abgeschlossen, kann man beim Arbeitsdelegaten per **EndInvoke()** - Aufruf die Ergebnisse anfordern, wobei das Operationsobjekt als Parameter zu übergeben ist:

```
double[] da = rs.EndInvoke(ar);
```

Die **EndInvoke()** - Methode hat denselben Rückgabewert wie die Arbeitsmethode (im Beispiel mit dem Typ **double[]**).

Neben der Übergabe der Arbeitsergebnisse des Pool-Threads erfüllt die **EndInvoke()** - Methode noch weitere Aufgaben und sollte daher auf jeden Fall aufgerufen werden:

- Ist bei der Hintergrundaktivität eine Ausnahme aufgetreten, wird diese von **EndInvoke()** erneut geworfen, sodass sie vom aufrufenden Thread zur Kenntnis genommen und bearbeitet werden kann.
- Die von der CLR zur Verwaltung der asynchronen Operation benutzten Ressourcen werden freigegeben. Ohne Aufruf von **EndInvoke()** entsteht ein Ressourcenleck (Richter 2006, S. 621).

Das folgende Programm lässt 10 Summen von Zufallszahlen von einem Pool-Thread berechnen

```
using System;
using System.Threading;

public delegate double[] MakeDoubleArray(int anz);

class Prog {
 static Random zzg = new Random();

 static double[] RandomSums(int anz) {
  . . .
 }

 static void Main() {
  Console.WriteLine("Der Arbeitsdelegat soll per Poolthread 10 " +
  "Summen von Zufallszahlen ermitteln.\n");
  var rs = new MakeDoubleArray(RandomSums);
  IAsyncResult ar = rs.BeginInvoke(10, null, null);

  int sek = 0;
  while (!ar.IsCompleted) {
  Console.WriteLine("Warte seit {0} Sekunden auf den Hintergrund-Thread", sek++);
  Thread.Sleep(1000);
  }

  Console.WriteLine("\nReport der ermittelten Summen:");
  foreach (double zs in rs.EndInvoke(ar))
   Console.WriteLine(" " + zs);
 }
}
```

und wartet unproduktiv auf das Ergebnis:

```
Der Arbeitsdelegat soll per Poolthread 10 Summen von Zufallszahlen ermitteln.

Warte seit 0 Sekunden auf den Hintergrund-Thread
Warte seit 1 Sekunden auf den Hintergrund-Thread

Report der ermittelten Summen:
 494930329
 495003294
 494964374
 495035327
 495165614
 494972809
 495014438
 494966394
 494866876
 495084050
```

Statt in einer **while**-Schleife zwischen zwei **IsCompleted**-Abfragen den primären Thread schlafen zu legen, sollte die Wartezeit für nützliche Arbeiten verwendet werden.

Eine wenig empfehlenswerte Alternative besteht darin, ohne Prüfung des Bearbeitungszustands die **EndInvoke()** - Methode des Arbeitsdelegaten aufzurufen, weil diese Methode auf das Auftragsende wartet und damit den aktuellen Thread blockiert.

Mit der **Rückruftechnik** kommen wir ohne wiederholtes Nachfragen (engl.: *Polling*) und ohne Warten an die Ergebnisse einer asynchronen Auftragsabwicklung heran. Dazu definieren wir eine Rückrufmethode, die den Delegatentyp **AsyncCallBack** (siehe oben) erfüllt, z. B.:

```
static void Report(IAsyncResult ar) {
   MakeDoubleArray z = (MakeDoubleArray)ar.AsyncState;
   Console.WriteLine("Report der ermittelten Summen:");
   double[] da = z.EndInvoke(ar);
   foreach (double zs in da)
      Console.WriteLine(" "+zs);
}
```

Sobald die Arbeitsmethode beendet ist, wird (immer noch im Pool-Thread) die Rückrufmethode aufgerufen und dabei per Aktualparameter mit einer Referenz auf das Operationsobjekt versorgt. Im Beispiel soll die Rückrufmethode `Report()` das Operationsobjekt in einem **EndInvoke()** - Aufruf an den Arbeitsdelegaten verwenden, um die Ergebnisse anzufordern. Die nötige Referenz auf den Arbeitsdelegaten wird folgendermaßen in die Rückrufmethode transportiert:

- Im **BeginInvoke()** - Aufruf an den Arbeitsdelegaten wird seine eigene Adresse als dritter Parameter übergeben:
  ```
  rs.BeginInvoke(3, new AsyncCallback(Report), rs);
  ```
- Wie oben erläutert, ist dieser Parameter im Operationsobjekt enthalten und über die Eigenschaft **AsyncState** ansprechbar.

Das folgende Programm

```
using System;
using System.Threading;

public delegate double[] MakeDoubleArray(int anz);

class Prog {
   static Random zzg = new Random();

   static double[] RandomSums(int anz) {
      . . .
   }
   static void Report(IAsyncResult ar) {
      . . .
   }

   static void Main() {
      MakeDoubleArray rs = new MakeDoubleArray(RandomSums);
      rs.BeginInvoke(3, new AsyncCallback(Report), rs);
      Console.WriteLine("Der Arbeitsdelegat soll per Pool-Thread 3 " +
         "Summen von Zufallszahlen ermitteln.\n");
      Console.WriteLine("Der primäre Thread schläft 5 Sekunden.\n");
      Thread.Sleep(5000);
      Console.WriteLine("\nDer primäre Thread ist aufgewacht.\n");
   }
}
```

demonstriert die APM-Rückruftechnik, verzichtet aber der Übersichtlichkeit halber auf eine sinnvolle Aktivität im Vordergrund-Thread.

```
Der Arbeitsdelegat soll per Pool-Thread 3 Summen von Zufallszahlen ermitteln.

Der primäre Thread schläft 5 Sekunden.

Report der ermittelten Summen:
 495128107
 495006556
 495034464

Der primäre Thread ist aufgewacht.
```

Bei Richter (2006, S. 621) finden sich die folgenden Empfehlungen zum APM-Einsatz:

- Auf einen **BeginInvoke()** - Aufruf sollte unbedingt ein **EndInvoke()** - Aufruf mit dem zugehörigen Operationsobjekt als Parameter folgen, weil ansonsten von der asynchronen Operation belegte CLR-Ressourcen nicht mehr freigegeben werden. Wie das obige Beispiel zeigt, kann der **EndInvoke()** - Aufruf in der Rückrufmethode und somit im Hintergrund-Thread erfolgen, sodass der aufrufende Thread nicht warten muss.[1]
- Für jedes Operationsobjekt sollte **EndInvoke()** nur *einmal* aufgerufen werden, weil diese Methode eventuell Aufräumungsarbeiten ausführt, sodass ein erneuter Aufruf scheitern könnte.

[1] Eine Ausnahme von der Pflicht zum **EndInvoke()** - Aufruf besteht beim **BeginInvoke()** - Aufruf an das **Dispatcher**-Objekt des UI-Threads (siehe Abschnitt 17.3.1).

17.6.1.2 Asynchrone Schreib- und Leseoperationen

Als asynchrone Alternative zu den im Abschnitt 16.1.3 vorgestellten synchronen Methoden **Read()** und **Write()** unterstützt die Klasse **Stream** das APM-Muster durch die Methodenpaare

- **BeginRead()** und **EndRead()**
- **BeginWrite()** und **EndWrite()**

Allerdings ist mir auf diesem Weg in einem Beispielprogramm *keine* asynchrone Dateiausgabe mit Hilfe der Klasse **FileStream** gelungen. Den vermutlichen Grund für das beobachtete synchrone (blockierende) Verhalten von **BeginWrite()** verrät ein Kommentar im Quellcode der BCL-Klasse **Stream**:[1]

```
// To avoid a race with a stream's position pointer & generating ----
// conditions with internal buffer indexes in our own streams that
// don't natively support async IO operations when there are multiple
// async requests outstanding, we will block the application's main
// thread if it does a second IO request until the first one completes.
```

Seit .NET Framework 4.5 unterstützt **Stream** auch die Task Parallel Library (TPL) durch die Methoden **ReadAsync()** und **WriteAsync()**, denen bei neuen Projekten der Vorzug gegeben werden sollte (siehe Beispiel im Abschnitt 17.5.9).

17.6.2 Event-based asynchronous Pattern (EAP)

Klassen, die das im .NET Framework 2.0 eingeführte *Event-based asynchronous Pattern* (EAP) implementieren, bieten Methoden mit dem Namensausklang **Async** an, welche die asynchrone Ausführung von zeitaufwändigen Operationen erlauben, z. B.:

- Die als EAP-Vielzweck-Implementation geltende Klasse **BackgroundWorker** besitzt die Methode **RunWorkerAsync()** und bietet das Ereignis **DoWork** an (Albahari 2014).
- Die Klasse **WebClient** besitzt die Methode **DownloadFileAsync()**, die eine Datei asynchron per Pool-Thread herunterlädt.
- Die Klasse **SmtpClient** besitzt die Methode **SendAsync()** zum asynchronen Versenden einer Mail.

Die Beendigung einer **Async**-Methode meldet ein Ereignis mit dem Namensausklang **Completed**, das denselben Namensanfang besitzt wie die Methode, z. B.:

- Das zu **DownloadFileAsync()** gehörige Ereignis heißt **DownloadFileCompleted**.
- Das zu **SendAsync()** gehörige Ereignis heißt **SendCompleted**.

Mittlerweile wird das EAP als veraltete und überflüssige Technik angesehen, z. B. auch von Microsoft:[2]

> It's no longer recommended for new development.

Die als Beispiele für das EAP erwähnten Methoden wurden übrigens mittlerweile durch Alternativen ergänzt, die mit der TPL arbeiten:

- Mit dem .NET Framework 4.5 hat die Klasse **WebClient** die TPL-Methode **DownloadFileTaskAsync()** erhalten.
- Seit .NET 5.0 besitzt die Klasse **SmtpClient** die TPL-Methode **SendMailAsync()**.

---

[1] https://source.dot.net/#System.Private.CoreLib/Stream.cs
Siehe auch: https://www.codeproject.com/tips/575618/avoiding-deadlocks-with-system-io-stream-beginread

[2] https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/

17.6.3 PLINQ und andere aktuelle Multithreading-Lösungen

Im Unterschied zu den bisher im Abschnitt 17.6 erwähnten Techniken sind die folgenden Verfahren aktuell, können aber leider aus Zeitgründen im Manuskript nicht behandelt werden:

- PLINQ
  Das mit dem .NET Framework 4.0 eingeführte PLINQ (paralleles LINQ) kann die LINQ-Standardabfrageoperationen für Kollektionen (LINQ to Objects) durch Parallelität beschleunigen.[1]
- TPL-Datenflussbibliothek
  Die komplexe, mit dem .NET Framework 4.5 eingeführte TPL-Datenflussbibliothek erweitert die TPL um die Möglichkeit zur Modellierung von Arbeitsabläufen mit einem möglichst hohen Grad an Parallelität.[2]
- Thread-sichere Kollektionsklassen
  Im Namensraum **System.Collections.Concurrent** bietet die BCL Kollektionsklassen mit Thread-Sicherheit beim Einfügen und Entnehmen von Elementen an. Bekannte Klassen aus diesem Namensraum sind **BlockingCollection<T>**, **ConcurrentDictionary<K, V>**, **ConcurrentQueue<T>** und **ConcurrentStack<T>**. Im Vergleich zu Sperrmaßnahmen durch den Benutzer-Code (siehe Abschnitt 17.1.4.1) wird eine höhere Performanz versprochen.[3]

17.7 Übungsaufgaben zum Kapitel 17

1) Welche von den folgenden Aussagen sind richtig?

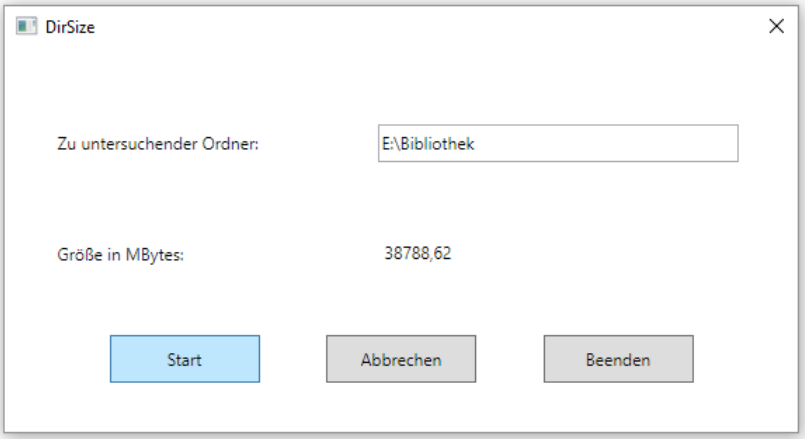
1. Ein Thread im Zustand **Stopped** lässt sich mit der Methode **Start()** reaktivieren.
2. WPF-Steuerelemente können nur in dem Thread angesprochen werden, der sie erstellt hat.
3. Das Cooperative Cancellation Framework (mit den BCL-Typen **CancellationTokenSource** und **CancellationToken**) sollte bei Threads und Aufgaben (**Task**-Objekten) zur vorzeitigen Beendigung verwendet werden.
4. Durch die asynchrone Programmierung mit Hilfe der Schlüsselwörter **async** und **await** lässt sich die Fehlerbehandlung in *einer* Methode konzentrieren.

2) Objekte der Signalisierungsklasse **CountdownEvent** enthalten einen Zähler und starten mit einem positiven Wert, der sich bei jedem Aufruf der Instanzmethode **Signal()** um eins verringert. Hat sich ein Thread per **Wait()** - Aufruf an das Signalisierungsobjekt in Wartestellung begeben, wird er beim Zählerstand null reaktiviert. Verwenden Sie die Klasse im Rahmen eines Konsolenprogramms für einen simulierten Raketenstart, der in einem eigenen Thread abläuft, sobald ein **CountdownEvent** 10mal gesetzt worden ist. Die Kontrollausgaben des Programms könnten ungefähr so aussehen:

```
Thread-Rakete bereit, wartet auf CountdownEvent.

Countdown läuft:
10 9 8 7 6 5 4 3 2 1

Thread-Rakete startet:
.. . .  .  .   .   .    .    .     .     .      .      .       .
```

[1] https://docs.microsoft.com/de-de/dotnet/standard/parallel-programming/parallel-linq-plinq
[2] https://docs.microsoft.com/de-de/dotnet/standard/parallel-programming/dataflow-task-parallel-library
[3] https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/data-structures-for-parallel-programming

3) Erstellen Sie eine WPF-Anwendung, welche die Größe eines Ordners (inklusive aller Unterordner) in einer unterbrechbaren Aufgabe berechnet, sodass die Bedienelemente des Formulars stets verzögerungsfrei reagieren, z. B.:



18 Datenbankprogrammierung mit ADO.NET

Die Verwaltung großer Datenbestände (z. B. Personaldaten einer Firma, Bestellungen eines Webshops) ist eine häufige und oft unternehmenswichtige Aufgabe, wobei der direkte Kontakt mit den Daten meist einem spezialisierten Datenbankmanagementsystem (DBMS) überlassen wird. In diesem Kapitel steigen wir in die Datenbankprogrammierung in C# ein und behandeln dabei folgende Themen:

- Grundlagen der relationalen Datenbanktechnik (inkl. SQL)
  Grundlegende Merkmale von relationalen Datenbanken werden ebenso behandelt wie die seit langen Jahren für lesende und schreibende Datenbankzugriffe verwendete *Structured Query Language* (SQL).
- Microsofts Datenbankprogramme
  Dazu zählt der **Microsoft SQL Server** (ein in verschiedenen Varianten verfügbares DBMS) und das **SQL Server Management Studio**.
- Elementare Klassen in der ADO.NET - Bibliothek
  Zur Kooperation mit relationalen Datenbankverwaltungsprogrammen enthält die BCL eine elementare Bibliothek namens **ADO.NET**, die von den meisten .NET-Programmen mit Datenbankzugriff direkt oder indirekt verwendet wird. Die ersten drei Buchstaben des Namens wurden vom Windows-Programmiermodell *ADO* (*ActiveX Data Objects*) übernommen. Es gibt aber kaum noch technische Gemeinsamkeiten, und insbesondere ist ADO.NET *keine* ActiveX-Technik.
- Typisierte DataSets
  Durch automatisch erstellte datenbank-spezifische Spezialisierungen der ADO.NET - Klassen wird dafür gesorgt, dass im Quellcode eines Anwendungsprogramms Datenbankelemente (z. B. Tabellen, Spalten) über C# - Eigenschaften angesprochen werden können. Im Vergleich zu der in traditionellen ADO.NET - Anwendungen üblichen Ansprache von Datenbankelementen über **String**-Argumente von Indexern ...
  - kann der Compiler die Typsicherheit bei Wertzuweisungen garantieren,
  - kann die Entwicklungsumgebung mit der IntelliSense-Technik das Codieren unterstützen.

Im Kapitel 20 über das *Entity Framework Core*, das als ORM-Lösung (*Object Relational Mapping*) die Kluft zwischen objektorientierter Programmierung und relationaler Datenbanktechnik überbrückt, wird eine moderne Variante der Datenbankprogrammierung vorgestellt, die speziell für komplexe Projekte empfehlenswert ist.

Wer erwägt, das Kapitel 18 auszulassen und sofort zur Behandlung der modernen Datenzugriffstechnik in Kapitel 20 zu springen, der sollte die folgenden Lektürehinweise beachten:

- Der Abschnitt 18.6 über die elementaren Klassen in der ADO.NET - Bibliothek ist sehr anstrengend und nicht unbedingt erforderlich für das Verständnis von Kapitel 20. Auch der Abschnitt 18.7 ist ein Streichkandidat, weil die dort beschriebenen typisierten DataSets nicht zu den ganz aktuellen Datenzugriffstechniken zählen und im weiteren Verlauf des Manuskripts nicht mehr auftauchen. Wer aber mit Projekten in Berührung kommt, die elementare ADO.NET - Klassen oder typisierte DataSets verwenden, der kommt eine gründliche Beschäftigung mit diesen Techniken nicht herum.
- Das Kapitel 19 über die an SQL angelehnte Abfragetechnik namens *LINQ* (*Language Integrated Query*) muss unbedingt vor dem Kapitel 20 gelesen werden, weil das Entity Framework Core diese Technik intensiv verwendet.

18.1 Datenbankmanagementsysteme

18.1.1 Anforderungen und technische Lösungen

Für den Begriff *Datenbank* schlägt Ebner (2000, S. 21) die folgende Definition vor:

> *Eine Datenbank ist eine Sammlung von nicht-redundanten Daten, die von mehreren Anwendungen benutzt werden kann.*

Redundanz würde sich z. B. in der Datensammlung eines Online-Shops einstellen, wenn man für jede Bestellung einen Datensatz anlegen und dabei auch die Adresse des Bestellers einbeziehen würde. Sobald von einem Kunden mehrere Bestellungen vorlägen, wäre seine Adresse mehrfach vorhanden. Neben dem unsinnigen Erfassungsaufwand und der Platzverschwendung hat die Redundanz einen weiteren Nachteil: die Gefahr **inkonsistenter** Daten. In einer Datenbank mit der dominanten relationalen Technik (siehe Abschnitt 18.2) wird das Problem gelöst durch ...

- eine Tabelle mit den Kunden, die als *Master-Tabelle* bezeichnet wird,
- eine Tabelle mit den Bestellungen, die als *Details-Tabelle* bezeichnet wird,
- eine Beziehung zwischen den Tabellen.
  Eine Zeile in der Tabelle mit den Bestellungen enthält keine vollständige Kundeninformation, sondern nur einen Verweis auf den betroffenen Kunden (z.B. über eine Kundennummer).

Mit dem zweiten Kriterium aus obiger Definition (*Benutzbarkeit durch mehrere Anwendungen*) ist in erster Linie gemeint, dass Anwendungsprogramme nicht direkt auf die Datenbestände zugreifen sollen, sondern nur über ein spezielles **Datenbankmanagementsystem** (**DBMS**). Diese Software ist für die

- fehlerfreie,
- effiziente (nicht-redundante, schnelle),
- gegen Datenverluste gesicherte
- und vor illegitimen Zugriffen geschützte

Verwaltung der Daten verantwortlich und ermöglicht simultane Zugriffe durch mehrere Anwendungsprogramme, sofern entsprechende Rechte vorhanden sind. Bekannte Beispiele für diese Software-Gattung sind:[1]

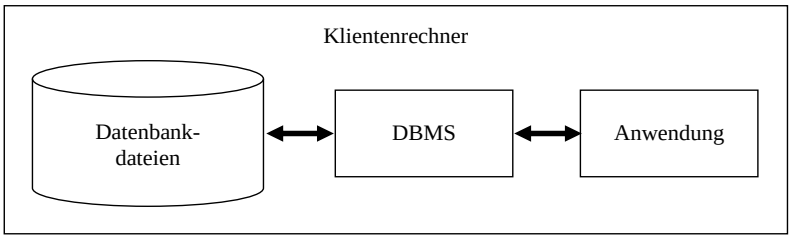
- Oracle Database
- MySQL
- Microsoft SQL Server
- PostgreSQL
- MongoDB
- DB2 von IBM

Auch bei Datenbeständen, die voraussichtlich nur von einer einzigen Anwendung benutzt werden sollen, kann es sinnvoll (bequem und sicher) sein, die Verwaltung der Daten an ein DBMS zu delegieren. Man kann zwei Datenbankeinsatzszenarien unterscheiden, wobei sich für den Datenbankzugriff durch ein Anwendungsprogramm in C# praktisch keine Unterschiede ergeben:

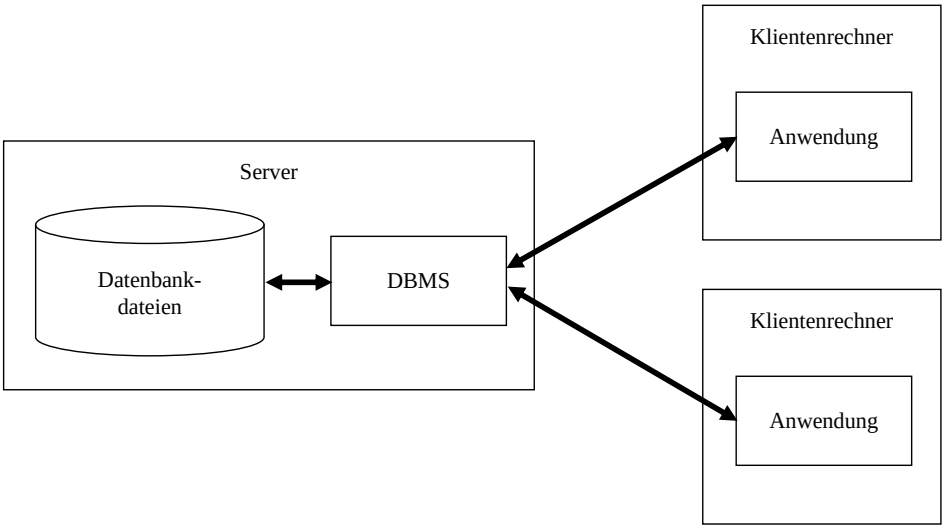
---

[1] Die Anordnung in der Liste folgt der hier dokumentierten Popularitäts-Rangordnung aus 2020: https://www.statista.com/statistics/809750/worldwide-popularity-ranking-database-management-systems/

- **Einzelbenutzerdatenbank**
  Anwendungsprogramm und DBMS laufen auf demselben Rechner.



- **Mehrbenutzerdatenbank (Client-Server - Lösung)**
  Das DBMS befindet sich auf einem Server und bedient die Zugriffe durch Anwendungsprogramme auf verschiedenen Klientenrechnern.



18.1.2 Microsoft SQL Server Express

Wir werden im Kurs bzw. Manuskript zwei Varianten der Express-Edition von Microsofts SQL-Server einsetzen:

- **Microsoft SQL Server Express Edition**
  Das durch (normalerweise automatisch gestartete) Dienste realisierte Programm erlaubt die simultane Nutzung durch mehrere Anwendungsprogramme im Client-Server - Betrieb mit wenigen (für uns irrelevanten) Einschränkungen gegenüber den kostenpflichtigen Varianten von Microsofts SQL Server.
- **Microsoft SQL Server Express LocalDB**
  Wenn nur eine Einzelbenutzerdatenbank benötigt wird, aber der volle Funktionsumfang von Microsofts SQL Server gewünscht ist, dann kommt die LocalDB-Variante in Frage, die von Microsoft als vereinfachte Variante der SQL Server Express - Edition beschrieben wird.[1]
  Die LocalDB-Variante läuft als Prozess, der bei Bedarf gestartet wird.

Beide Varianten der Express-Edition dürfen kostenlos produktiv genutzt werden und bieten die vollständige SQL Server Engine.[2]

Bei den meisten Themen des aktuellen Kapitels spielt die eingesetzte SQL Server - Variante keine Rolle, sodass die etwas aufwändigere Installation und Konfiguration der Client-Server - Variante

---

[1] https://docs.microsoft.com/de-de/sql/database-engine/configure-windows/sql-server-express-localdb

[2] Mit der SQL Server Compact Edition hat Microsoft noch eine besonders schlanke Einbenutzerdatenbank im Angebot, die aber nicht die vollständige SQL Server Engine bietet. Wer eine platzsparende Einbenutzerdatenbank sucht, sollte auch das Open Source - Programm SQLite in Erwägung ziehen.

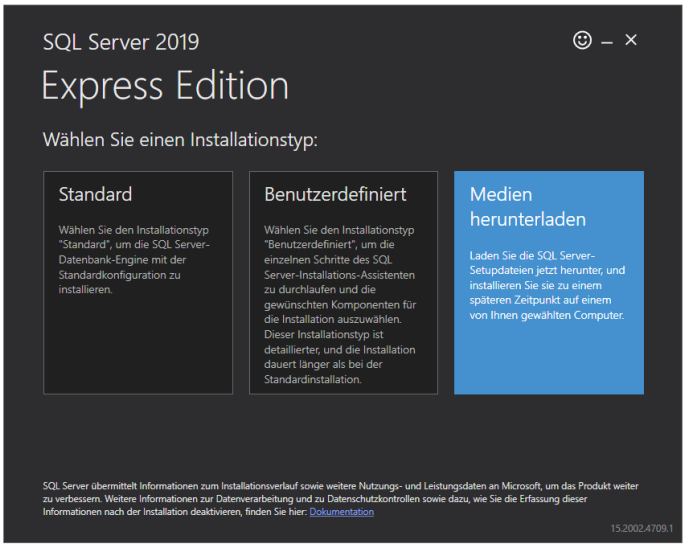
erst im Abschnitt 18.8 behandelt wird. Die LocalDB-Variante hat den Vorteil einer bequemen Installation zusammen mit den **SQL Server Data Tools** im Visual Studio 2019.

Um die LocalDB-Variante zur Verwendung durch ein selbst entwickeltes Programm auf einem Kundenrechner zu installieren, kann man so vorgehen:

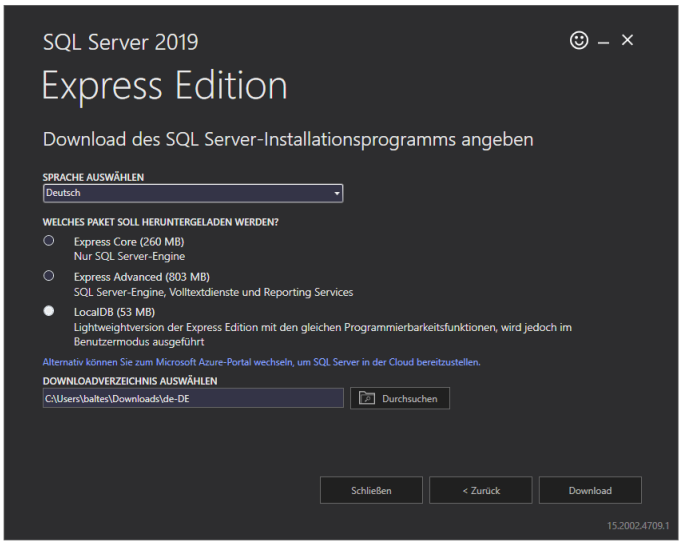
- Installationsprogramm **zur** Express-Edition von Microsofts SQL-Server herunterladen. Aktuell (im Mai 2021) ist über die folgende Webseite

  https://www.microsoft.com/de-de/sql-server/sql-server-downloads

  das Installationsprogramm **SQL2019-SSEI-Expr.exe** zu beziehen.
- Installationsprogramm starten und **Medien herunterladen** wählen:



- Im nächsten Dialog erfährt man u. a. die Größenunterschiede zwischen den angebotenen Varianten des SQL-Servers:



- Bei einer Entscheidung für die Option **LocalDB** erhält man eine einzelne MSI-Datei namens **SqlLocalDB.msi**, die sich bequem zusammen mit einer eigenen Anwendung ausliefern und in eine Installationsprozedur einbinden lässt.
- Eine per Doppelklick auf die Datei **SqlLocalDB.msi** gestartete Installation verläuft unspektakulär. Im Ergebnis befindet sich die LocalDB-Edition des SQL-Servers im Installationsordner

  C:\Program Files\Microsoft SQL Server

18.2 Relationale Datenbanken

Heute arbeiten die meisten Datenbankmanagementsysteme mit der **relationalen Datenbankstruktur**, wenngleich sich auch die sogenannten *NoSQL*-Datenbanken etabliert haben, die zur Verwaltung von weniger streng oder weniger statisch strukturierten Daten konzipiert wurden (z. B. MongoDB). Es sind zahlreiche relationale DBMS-Produkte (RDBMS-Programme) mit einem hohen Reifegrad verfügbar. Erfreulicherweise ist der Zugriff auf relationale Datenbanken über die Abfragesprache SQL (siehe Abschnitt 18.5) weitgehend standardisiert.

18.2.1 Tabellen

Bei einer relationalen Datenbank sind die Daten in **Tabellen**[1] angeordnet, wobei die **Zeilen** auch als **Datensätze** (engl.: *records*), **Tupel** oder **Fälle** und die **Spalten** auch als **Felder**, **Attribute**, **Merkmale** oder **Variablen** bezeichnet werden:

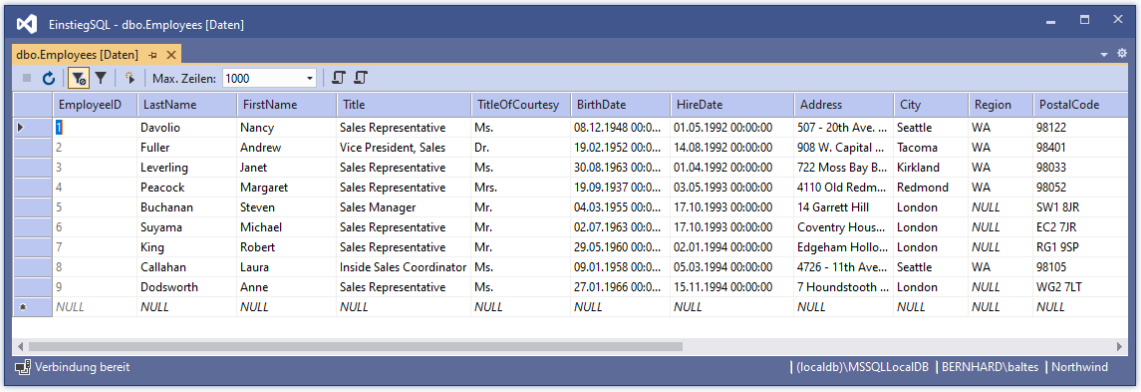
- Jede Tabelle enthält Zeilen eines bestimmten Typs (z. B. Kunden, Artikel, Lieferanten, Reklamationen). Welche Tabellen benötigt werden, hängt vom Anwendungsbereich ab. Jede Zeile der Tabelle enthält die zu einem Fall verfügbaren Informationen.
- Jede Spalte (jedes Feld) enthält für alle Fälle die Werte zu einem Attribut. Alle Werte in einer Spalte besitzen denselben **Datentyp** (z. B. Ganzzahl, Datum).
- Eine Tabelle besitzt in der Regel einen **Primärschlüssel** (engl.: *primary key*) mit den folgenden Eigenschaften:
  - Jeder Datensatz besitzt einen eindeutigen Wert (engl.: *unique constraint*).
  - Jeder Datensatz *muss* einen gültigen Wert besitzen (engl.: *not null constraint*).

  Oft dient eine *einzelne* Spalte als Primärschlüssel; man kann aber auch *zusammengesetzte Schlüssel* verwenden, die auf *mehreren* Spalten basieren. Bei großen Datenbanken ist eine *einzelne* Primärschlüsselspalte mit *numerischem* Datentyp aus Performanzgründen zu bevorzugen. Oft wird mit dem RDBMS vereinbart, die Werte einer solchen Primärschlüsselspalte ausgehend von einem Startwert automatisch zu erhöhen (engl.: *auto increment*).
- Man kann auch für beliebige andere Spalten die eben für Primärschlüssel genannten und andere **Restriktionen** (engl.: *constraints*) formulieren, z. B.:
  - Eindeutigkeit der Werte
  - Verbot fehlender Werte
  - Fremdschlüsselrestriktion (engl.: *foreign key constraint*)
    Einschränkung der Werte auf die Primärschlüsselwerte einer anderen Tabelle
- Für einen Primärschlüssel ist ein **Index** vorhanden, der die Suche nach bestimmten Werten dank Sortierung beschleunigt. Zusätzlich können in einer Tabelle zu weiteren Spalten Indizes angelegt werden. Weil für den Aufbau und die Pflege der Indizes ein gewisser Aufwand erforderlich ist, sollte man sich auf die zur Beschleunigung von Suchanfragen erforderlichen Indizes beschränken.

---

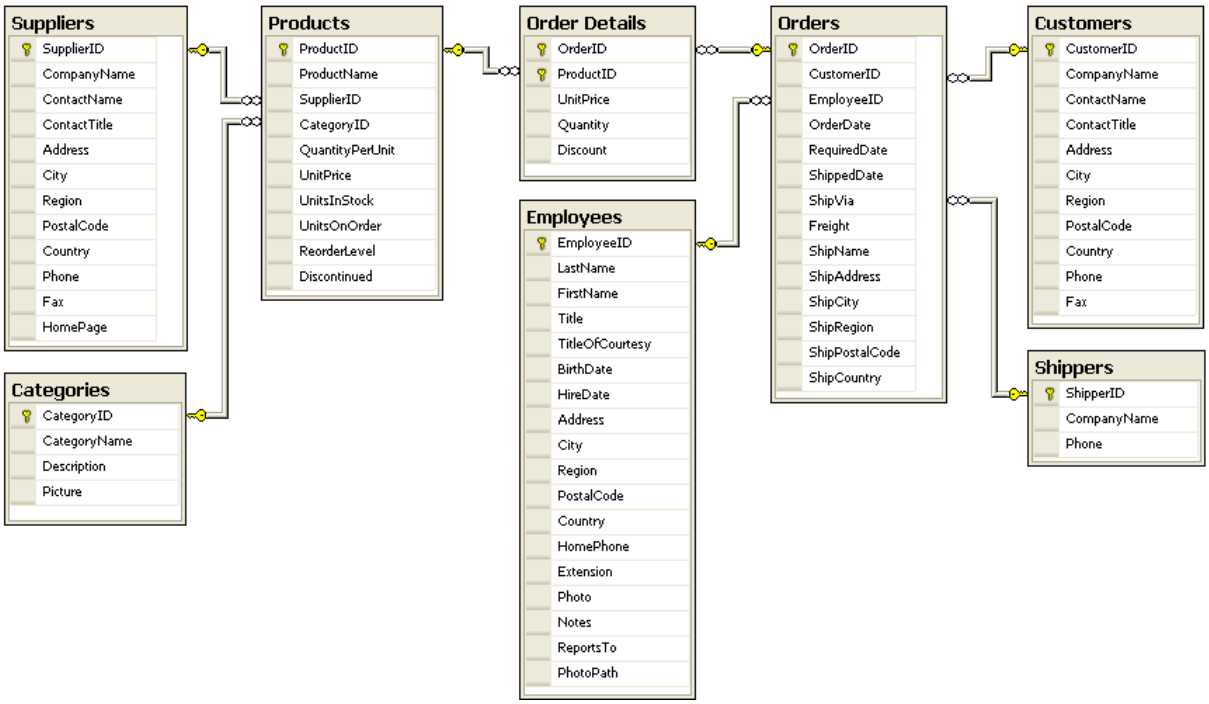
[1] In der Bezeichnung *relationale Datenbank* ist der erste Namensbestandteil im Sinn der Mathematik gemeint, wo man unter einer *Relation* eine Teilmenge des kartesischen Produkts aus mehreren Mengen versteht. Mit dieser etwas abstrakten Definition ist der im IT-Alltag üblichere Begriff *Tabelle* durchaus konsistent: Jede von den *m* Variablen (Spalten) einer Tabelle steuert die Menge ihrer potentiellen Ausprägungen bei. Ein Element des kartesischen Produkts dieser *m* Mengen ist ein *m*-Tupel mit *m* speziellen Variablenausprägungen, also eine Tabellenzeile. Eine Tabelle mit *n* Zeilen kann also in der Tat als Teilmenge des kartesischen Produkts (sprich: *als Relation*) aufgefasst werden. Bei einer mathematischen Relation sind (wie bei jeder Menge) alle Elemente verschieden. Bei den Tabellen eines relationalen Datenbankmanagementsystems verhindert in der Regel eine spezielle Variable mit einer Eindeutigkeitsforderung (der Primärschlüssel) das Auftreten von identischen Zeilen. In manchen Texten zur relationalen Datenbanktechnik wird unter einer *Relation* allerdings keine *Tabelle*, sondern eine *Beziehung* zwischen zwei Tabellen verstanden (siehe unten).

Wir betrachten im Kapitel 18 die von Microsoft angebotene Beispieldatenbank zu einer erfundenen Firma namens **Northwind** (siehe Abschnitt 18.3.2 zum Bezug und zur Installation der Datenbank). In der **Northwind**-Datenbank befindet sich u. a. eine Tabelle mit dem Personal der Firma:



EmployeeID	LastName	FirstName	Title	TitleOfCourtesy	BirthDate	HireDate	Address	City	Region	PostalCode
1	Davolio	Nancy	Sales Representative	Ms.	08.12.1948 00:0...	01.05.1992 00:00:00	507 - 20th Ave. ...	Seattle	WA	98122
2	Fuller	Andrew	Vice President, Sales	Dr.	19.02.1952 00:0...	14.08.1992 00:00:00	908 W. Capital ...	Tacoma	WA	98401
3	Leverling	Janet	Sales Representative	Ms.	30.08.1963 00:0...	01.04.1992 00:00:00	722 Moss Bay B...	Kirkland	WA	98033
4	Peacock	Margaret	Sales Representative	Mrs.	19.09.1937 00:0...	03.05.1993 00:00:00	4110 Old Redm...	Redmond	WA	98052
5	Buchanan	Steven	Sales Manager	Mr.	04.03.1955 00:0...	17.10.1993 00:00:00	14 Garrett Hill	London	NULL	SW1 8JR
6	Suyama	Michael	Sales Representative	Mr.	02.07.1963 00:0...	17.10.1993 00:00:00	Coventry Hous...	London	NULL	EC2 7JR
7	King	Robert	Sales Representative	Mr.	29.05.1960 00:0...	02.01.1994 00:00:00	Edgeham Hollo...	London	NULL	RG1 9SP
8	Callahan	Laura	Inside Sales Coordinator	Ms.	09.01.1958 00:0...	05.03.1994 00:00:00	4726 - 11th Ave...	Seattle	WA	98105
9	Dodsworth	Anne	Sales Representative	Ms.	27.01.1966 00:0...	15.11.1994 00:00:00	7 Houndstooth ...	London	NULL	WG2 7LT
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Das folgende Datenbankdiagramm zeigt die wichtigsten **Northwind**-Tabellen mit den zugehörigen Feldern und den Primärschlüsseln (einfach oder zusammengesetzt):



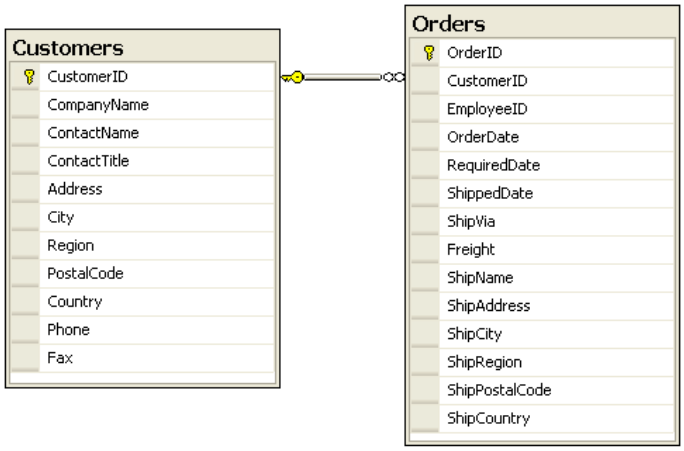
Durch Verbindungslinien sind die Beziehungen zwischen den Tabellen dargestellt, wobei wir uns auf die *Master-Details*- bzw. (1:n) - Beziehung beschränken (vgl. Abschnitt 18.2.2).

Ein redundanzfreier Datenbankentwurf erfordert bei den meisten Aufgabenstellungen das Verteilen der Daten auf *mehrere* Tabellen. Man spricht hier vom *Normalisieren* einer Datenbank.

18.2.2 Beziehungen zwischen Tabellen

Zwischen zwei Tabellen (z. B. `Customers` und `Orders`) kann über korrespondierende Felder mit bestimmten Eigenschaften (z. B. über das in beiden Tabellen vorhandene Feld `CustomerID`) eine **Beziehung** hergestellt werden. Wir verzichten auf eine generelle Behandlung der Thematik und beschränken uns auf den wichtigen Spezialfall der sogenannten **Master-Details**- bzw. (1:n) - Beziehung. Im Beispiel sind einem Fall der `Customers`-Tabelle mit einem bestimmten Wert beim Primärschlüsselfeld `CustomerID` (also einem bestimmten Kunden) alle Datensätze der `Orders`-

Tabelle zugeordnet, die denselben Wert im Feld `CustomerID` besitzen (also alle Bestellungen dieses Kunden):



Folglich kann man z. B. beim Auflisten von Bestellungen beliebige Daten der jeweils betroffenen Kunden (z. B. Adresse) einblenden, wobei das Prinzip der redundanzfreien Datenspeicherung gewährleistet ist.

Aus der Master-Tabelle wirkt der Primärschlüssel bei der Beziehung mit. Die zur Verknüpfung herangezogene Spalte der Details-Tabelle wird als *Fremdschlüssel* bezeichnet. Der Fremdschlüssel muss denselben Datentyp haben wie der zugehörige Primärschlüssel, aber nicht unbedingt denselben Namen.

Man kann eine Master-Details - Beziehung zwischen zwei Tabellen **A** und **B** durch eine Konstellation von Spalten und zugehörigen Restriktionen definieren:

- Tabelle **A** enthält eine Primärschlüsselspalte, die z. B. den Namen **AId** besitzt.
- **AId** erfüllt die Primärschlüsselrestriktionen:
  - Eindeutigkeit der Werte
  - Verbot fehlender Werte
- Tabelle **B** enthält eine Spalte, die denselben Datentyp hat wie **AId** und in der Regel auch denselben Namen besitzt.
- Die **AId**-Spalte in Tabelle **B** erfüllt die Fremdschlüssel-Restriktion. Es sind also nur Werte erlaubt, die in der Primärschlüsselspalte der Tabelle **A** vorhanden sind.

In der Northwind-Beispieldatenbank sind zahlreiche Master-Details - Beziehungen vorhanden. Bei Betrachtung in umgekehrter Richtung wird daraus jeweils eine Details-Master - bzw. (n:1) - Beziehung.

Aufgrund der im Datenbankentwurf vereinbarten Beziehungen sorgt ein RDBMS für **referentielle Integrität**:

- In einer Details-Tabelle werden bei einem Fremdschlüssel nur solche Werte akzeptiert, die im Primärschlüssel der verknüpften Master-Tabelle vorhanden sind (Fremdschlüssel-Restriktion).
- Eine Zeile der Master-Tabelle, auf die sich Zeilen einer Details-Tabelle beziehen, kann nicht gelöscht werden.

Das sogenannte **Schema** einer relationalen Datenbank beinhaltet für jede Tabelle ihren Namen und die folgenden Informationen über ihre Spalten:

- Name
- Datentyp und Wertebereich
- Ggf. die Primärschlüsselrolle und ein vorhandener Index
- Restriktionen (Eindeutigkeit, Null-Verbot, Fremdschlüssel)

18.3 Datenbankzugriff im Visual Studio

18.3.1 SQL Server-Objekt-Explorer versus Server-Explorer

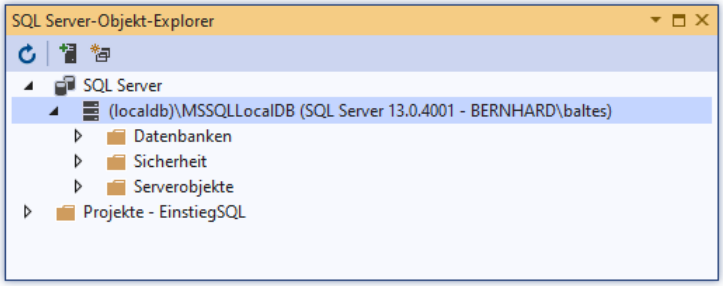
Wenn gemäß der Beschreibung im Abschnitt 3.3.2.3 die **SQL Server Data Tools** installiert worden sind, steht im Visual Studio der **SQL Server-Objekt-Explorer** bereit, dessen Fenster bei Bedarf mit

Ansicht > SQL Server-Objekt-Explorer

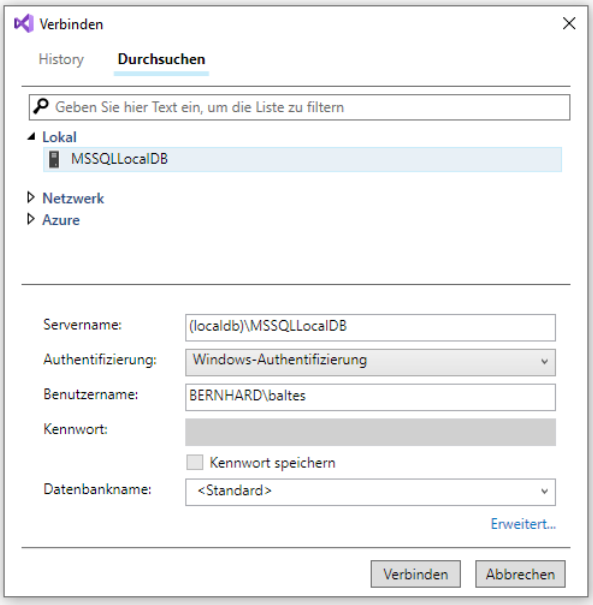
geöffnet werden kann. Hier sollte der **SQL Server** mit dem Namen

(localdb)\MSSQLLocalDB

zu sehen sein:

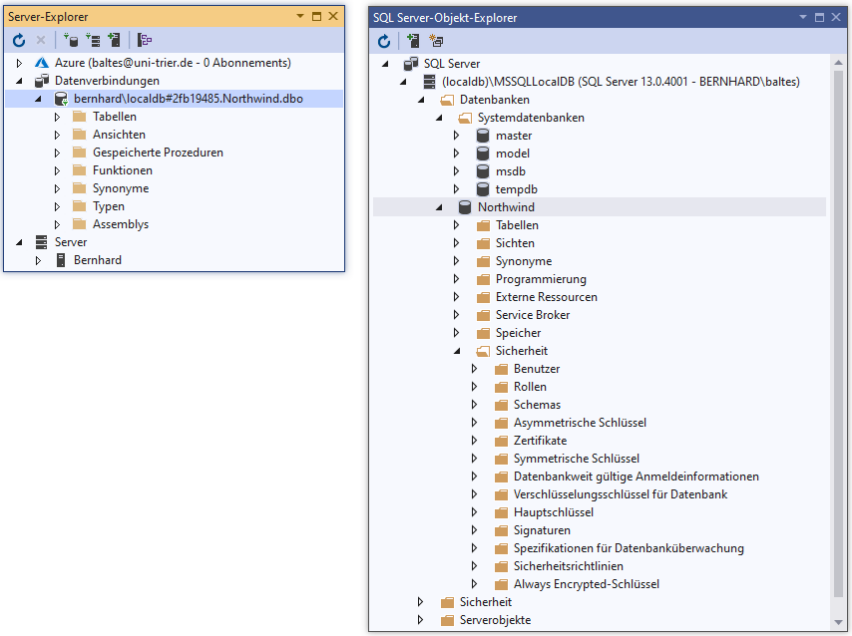


Nötigenfalls kann man über das Kontextmenü zum Knoten **SQL Server** oder nach einem Mausklick auf das Symbol einen **SQL Server hinzufügen**:



Der **SQL Server-Objekt-Explorer** ist zur Kooperation mit einem Microsoft SQL Server spezialisiert und damit aktuell für uns nützlicher als der im Visual Studio ebenfalls vorhandene **Server-Explorer**, der ein breiteres Einsatzspektrum besitzt und z. B. auch für die Kommunikation mit Microsofts Cloud-Dienst **Azure** zuständig ist. Der **SQL Server-Objekt-Explorer** kommt nahe an

die Funktionen des **SQL Server Management Studios** heran (siehe Abschnitt 18.4) und informiert z. B. auch über die für eine Datenbank vereinbarten Zugriffsrechte von Benutzern:



18.3.2 Beispieldatenbank Northwind erstellen

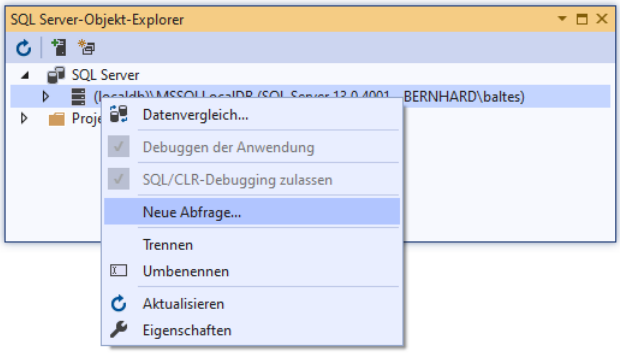
Mit dem **SQL Server-Objekt Explorer** kann man u. a.:

- Datenbanken anlegen
- Das Schema einer Datenbank modifizieren (z. B. Tabellen hinzufügen)
- Abfragen ausführen

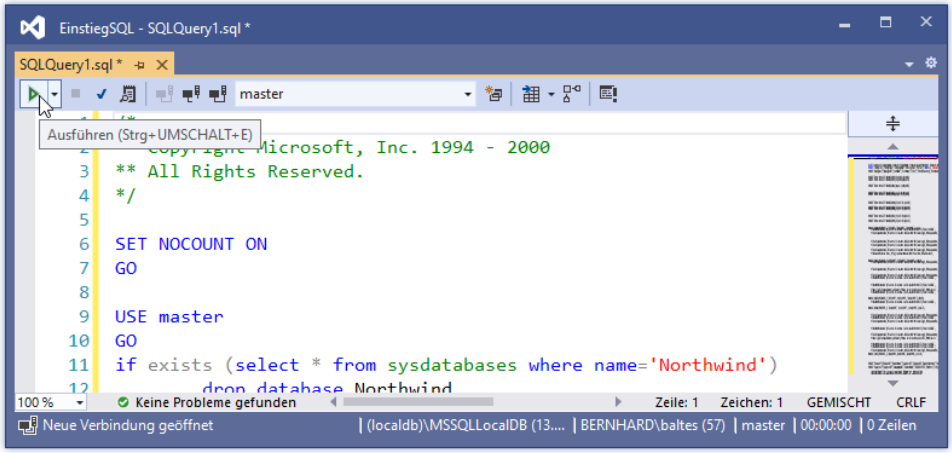
Wir erstellen über eine Abfrage (ein Programm in der Sprache SQL) die Beispieldatenbank **Northwind**, die trotz ihres stattlichen Alters von ca. 20 Jahren für unsere Zwecke gut geeignet ist, was die Langlebigkeit der SQL-Datenbanktechnik belegt. Auf der folgenden Webseite

https://github.com/Microsoft/sql-server-samples/tree/master/samples/databases/northwind-pubs

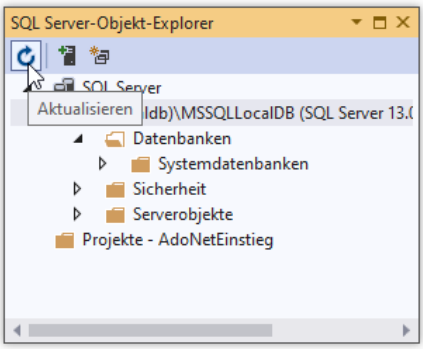
bietet Microsoft in der Datei **instnwnd.sql** ein SQL-Skript an, das die **Northwind**-Datenbank erstellt. Wählen Sie im **SQL Server-Objekt-Explorer** aus dem Kontextmenü zur Datenbankverbindung das Item **Neue Abfrage**,



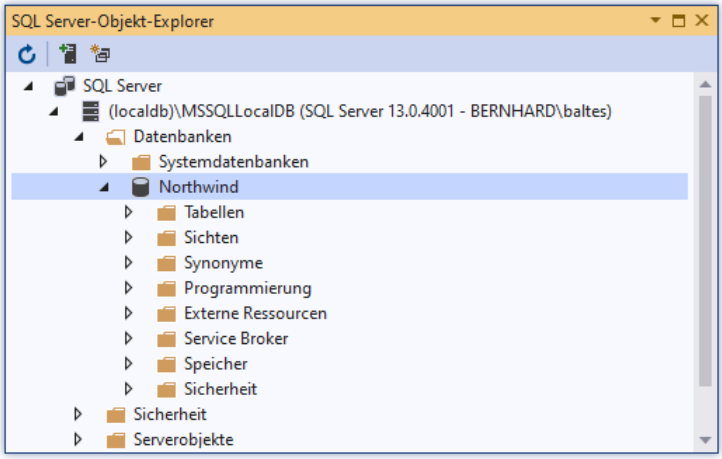
und übertragen Sie den Inhalt der Datei **instnwnd.sql** in das erscheinende Editor-Fenster **SQLQuery1.sql**. Starten Sie das SQL-Skript bzw. die Abfrage mit einem Klick auf das Symbol ▶ (**Ausführen**):



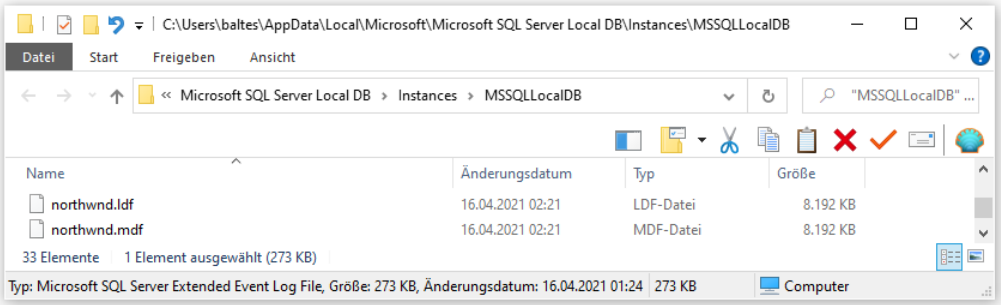
Nach dem erfolgreichen Ausführen der Abfrage und einem Mausklick auf den Schalter **Aktualisieren**



ist die Datenbank **Northwind** im **SQL Server-Objekt-Explorer** zu sehen:

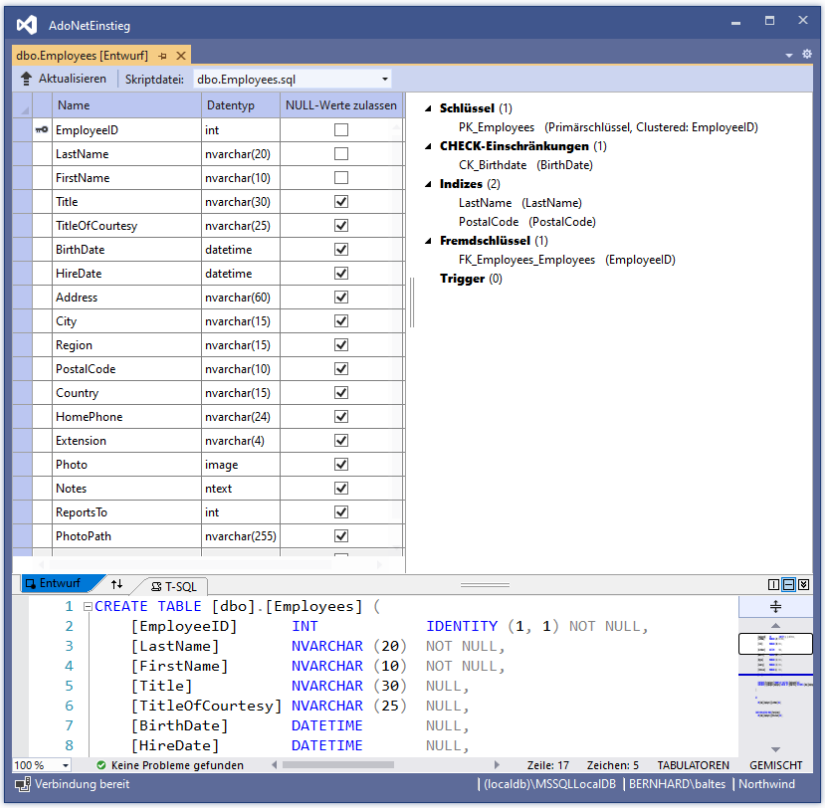


Die Dateien zur neu angelegten Datenbank befinden sich im Windows-Profilordner des Benutzers:



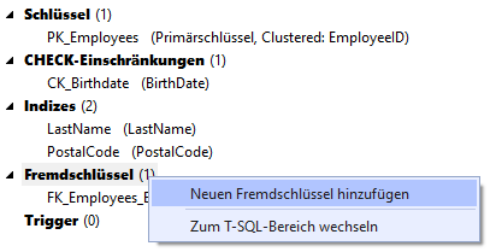
18.3.3 Tabellendesigner

Der **SQL Server-Objekt Explorer**, den unsere Entwicklungsumgebung aufgrund der Installation der **SQL Server Data Tools** besitzt (siehe Abschnitt 3.3.2.3), enthält einen Tabellendesigner, mit dem das Schema einer Datenbank inspiziert und modifiziert werden kann. Erweitern Sie z. B. zur Datenbank **Northwind** den Zweig **Tabellen**, und öffnen Sie per Doppelklick die Tabelle **Employees** zur Bearbeitung im Tabellen-Designer:



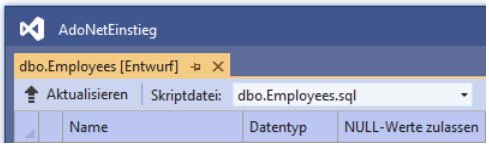
Der Tabellen-Designer besitzt drei Zonen:

- Merkmalstabelle (oben links)
  Hier lassen sich Merkmale ergänzen (neue Zeile am Ende der Merkmalstabelle), konfigurieren (z. B. Datentyp, Null-Werte - Verbot) oder löschen (per Kontextmenü).
- Kontextbereich (oben rechts)
  Hier werden über Kontextmenüs z. B. (Fremd)schlüssel und Restriktionen erstellt und gelöscht, z. B.:

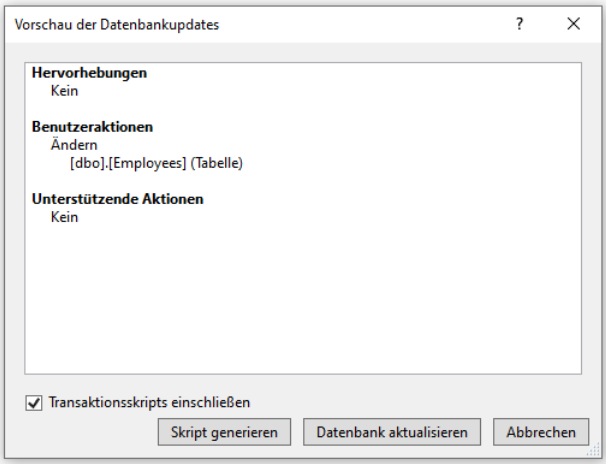


- Skriptbereich
  Hier befindet sich ein Skript in Microsofts SQL-Dialekt T-SQL, das die im Tabellen-Designer vorgenommenen Änderungen an den SQL Server übertragen kann.

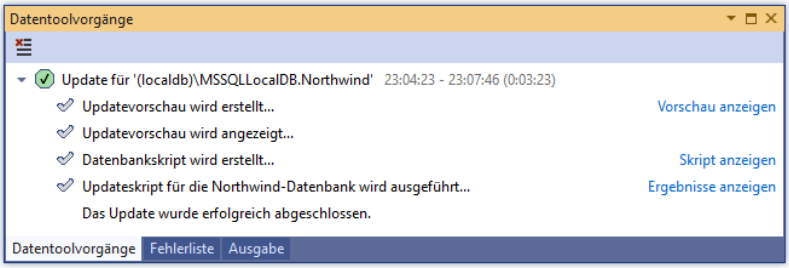
Um die im Tabellen-Designer vorgenommenen Änderungen zur Datenbank zu übertragen, klickt man auf den Schalter **Aktualisieren** über dem Spaltenbereich:



Es erscheint ein Fenster mit der **Vorschau des Datenbankupdates**, z. B.:

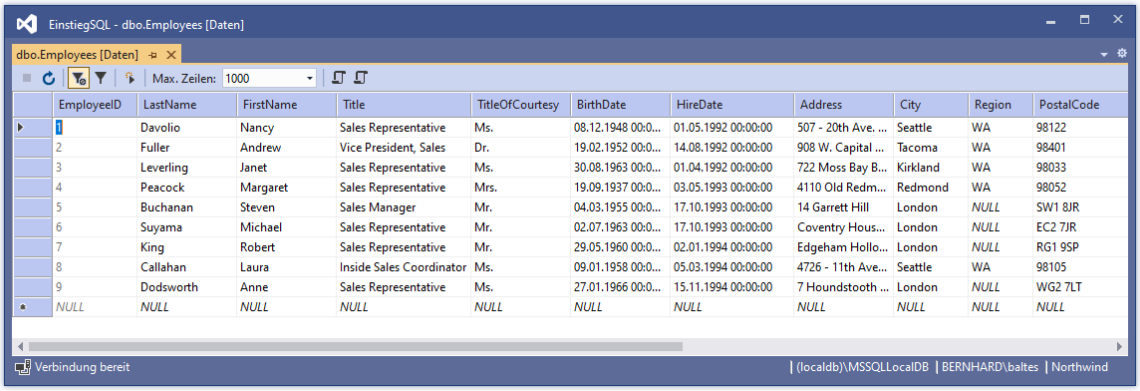


Nach einem Klick auf den Schalter **Datenbank aktualisieren** erfährt der SQL Server von den gewünschten Änderungen und führt sie in der Regel aus:



18.3.4 Dateneditor

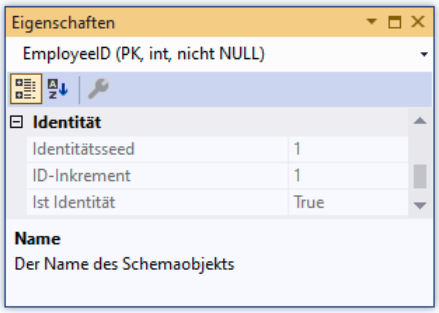
Über die Option **Daten anzeigen** im Kontextmenü zu einer Tabelle kann man deren Daten im Dateneditor einsehen und ändern, z. B.:



Im Beispiel wird für die Werte in der Spalte `EmployeeID` durch die graue Schriftfarbe signalisiert, dass *keine* Änderung möglich ist. Es handelt sich um Werte einer **Identitätsspalte**, die vom SQL-Server (mit dem Startwert und dem Inkrement 1) berechnet werden. Wie das SQL-Kommando **CREATE TABLE** zeigt, dient die Identitätsspalte `EmployeeID` in der Tabelle `Employees` als Primärschlüssel:

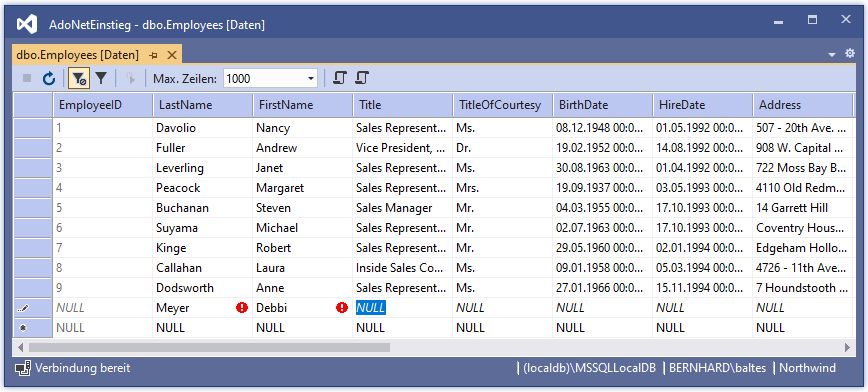
```
CREATE TABLE [dbo].[Employees] (
    [EmployeeID] INT IDENTITY (1, 1) NOT NULL,
     . . .
    CONSTRAINT [PK_Employees] PRIMARY KEY CLUSTERED ([EmployeeID] ASC),
     . . .
);
```

Das Visual Studio informiert im **Eigenschaften**-Fenster darüber, dass `EmployeeID` eine Identitätsspalte ist:

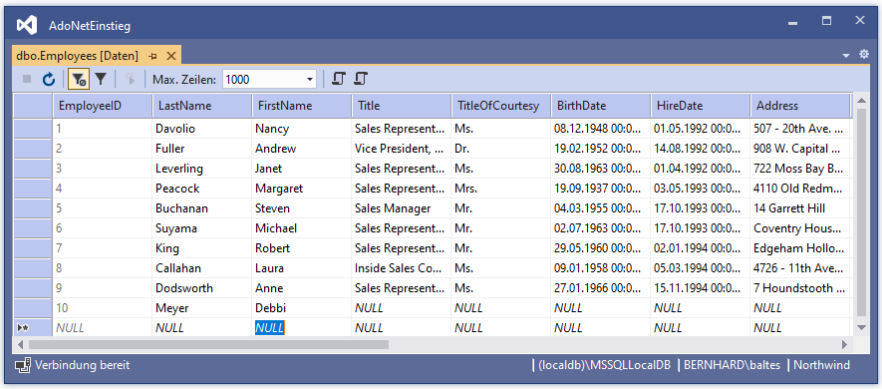


Bei einem neuen Fall wird der Wert für die Identitätsspalte nicht eingegeben, sondern vom Datenbankeditor berechnet.

Vor der Übertragung an die Datenbank sind geänderte Daten an einem Ausrufezeichen auf rotem Grund zu erkennen, z. B.:

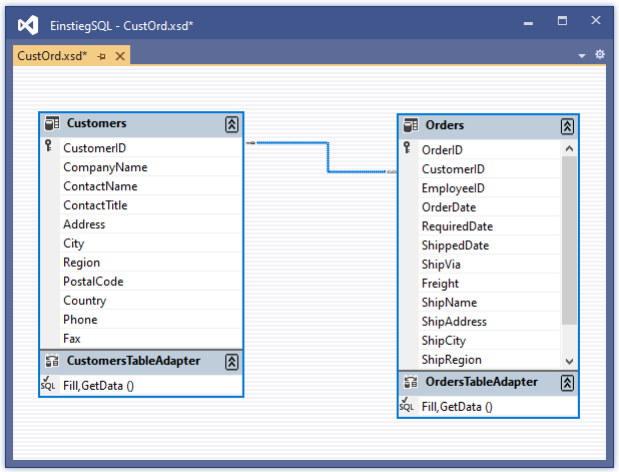


Eine neue oder geänderte Zeile wird an die Datenbank übertragen, sobald sich die Einfügemarke nicht mehr darin befindet:



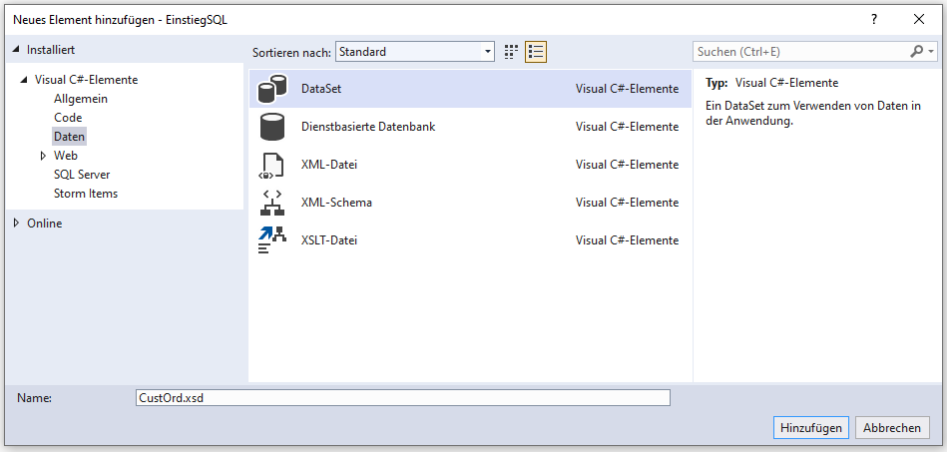
18.3.5 Datenbankdiagramm erstellen

Mit Hilfe des **DataSet-Designers**, dessen vollen Funktionsumfang wir im Abschnitt 18.7 kennenlernen werden, lassen sich Datenbankdiagramme erstellen, z. B.:

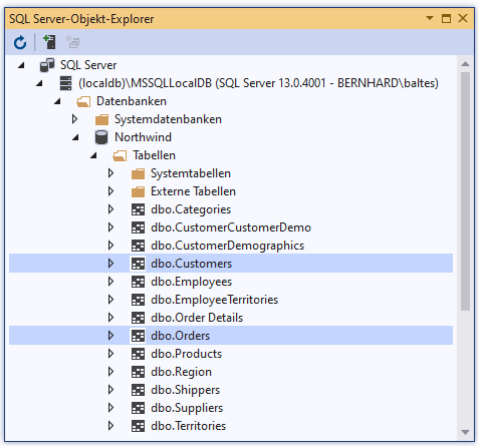


Die erforderlichen Arbeitsschritte im Visual Studio:

- Im Projektmappen-Explorer aus dem Kontextmenü zum Projekt wählen:
  Hinzufügen > Neues Element > Daten > DataSet



- Tabellen aus dem **SQL Server-Objekt-Explorer**



  auf das erstellte **xsd**-Dokument ziehen.

Anschließend lässt sich das Diagramm in der Designer-Zone der Entwicklungsumgebung durch Verschieben von Tabellen und Beziehungspfeilen gestalten.

Im Projektmappen-Explorer erscheint das komplexe Arbeitsergebnis des DataSet-Designers als **xsd**-Datei mit untergeordneten Einträgen, wobei wir aktuell ausschließlich am Datenbankdiagramm interessiert sind.

Wer sich die Mühe macht, das SQL Server Management-Studio zu installieren, hat anschließend u. a. einen leistungsfähigeren Datenbankdiagramm-Designer zur Verfügung (siehe Abschnitt 18.4.3).

18.4 SQL Server Management Studio

Das **SQL Server Management Studio** erleichtert die Verwaltung von SQL Server - Instanzen und von Datenbanken. Auf der folgenden Webseite

https://docs.microsoft.com/de-de/sql/ssms/download-sql-server-management-studio-ssms

wird das Programm beschrieben und zum Download angeboten.

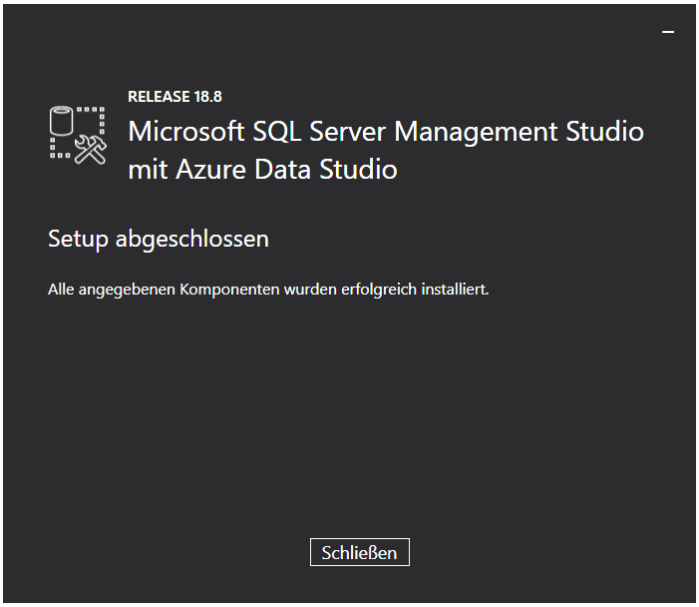
Viele Funktionen des Management Studios lassen sich auch mit den **SQL Server Data Tools** im Visual Studio realisieren (siehe Abschnitt 18.3). Wir werden das Management Studio an zwei Stellen als leistungsfähigere und bequemere Lösung bevorzugen:

- bei der Erstellung eines Datenbankdiagramms (siehe Abschnitt 18.4.3)
- bei der Rechteverwaltung für den SQL Server Express (siehe Abschnitt 18.8.4.1)

18.4.1 Installation

Nachdem das Installationsprogramm **SSMS-Setup-DEU.exe** zum SQL Server Management Studio

erfolgreich ausgeführt wurde,

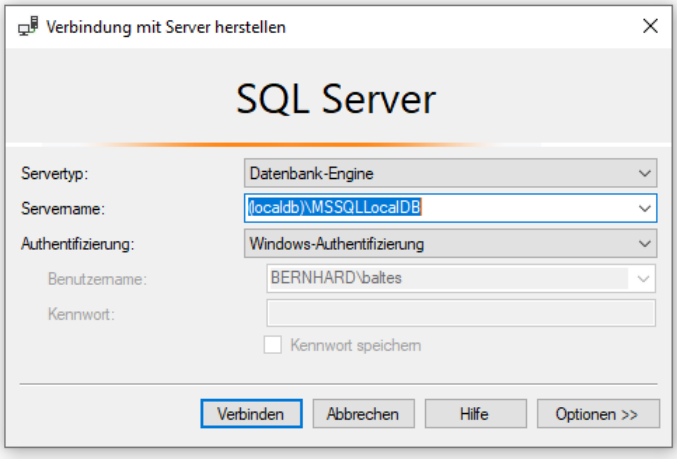


befindet sich der Link zum Starten des Management Studios in der Startmenü-Gruppe

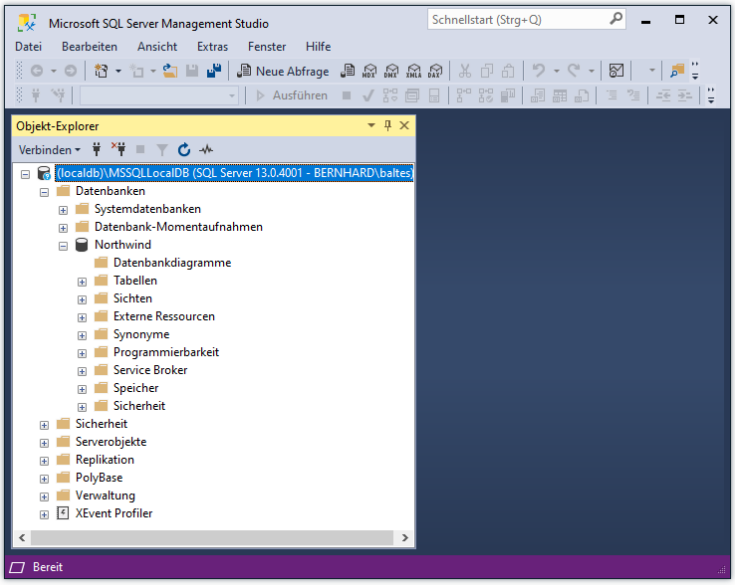
Microsoft SQL Server Tools 18

18.4.2 Mit einem SQL Server verbinden

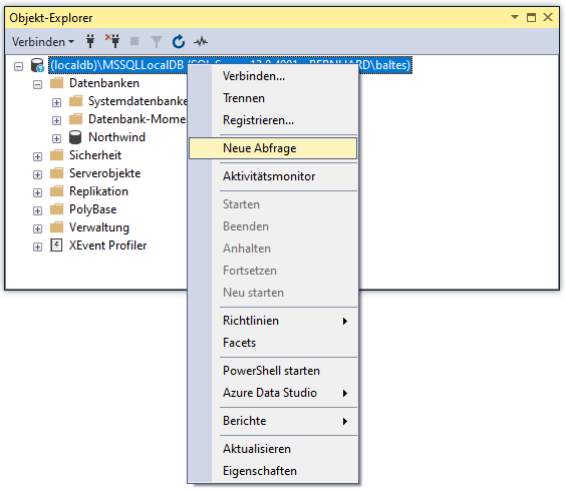
Das Management Studio sucht beim Start nach installierten SQL Servern. Wenn gemäß der Beschreibung im Abschnitt 3.3.2.3 die **SQL Server Data Tools** installiert worden sind, sollte der LocalDB-Server gefunden werden:



Nach einem Klick auf **Verbinden** können wir das Innenleben des SQL Servers explorieren:

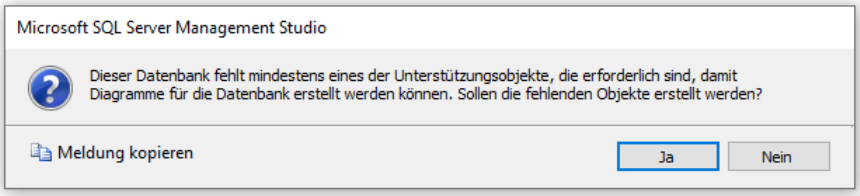


Wenn die Beispieldatenbank **Northwind** nicht schon aufgrund der Installation im Visual Studio (siehe Abschnitt 18.3.2) vorhanden wäre, könnten wir das SQL-Skript **instnwnd.sql** auch im Management Studio als Abfrage ausführen lassen, um die Datenbank zu erstellen. Im Kontextmenü zum Server befindet sich zu diesem Zweck das Item **Neue Abfrage**:



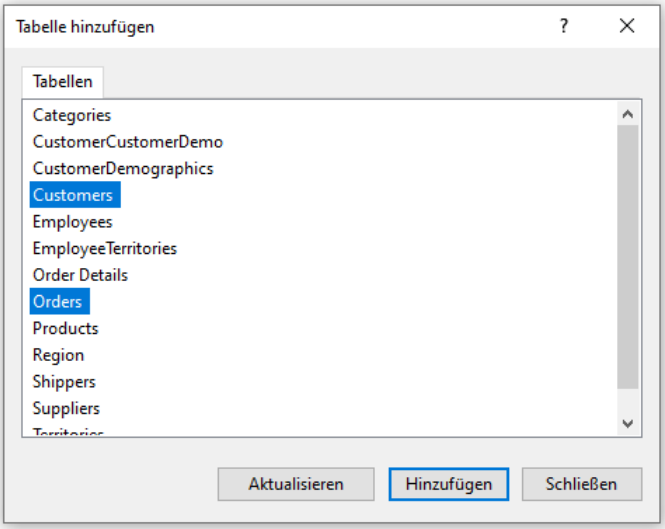
18.4.3 Datenbankdiagramm erstellen

Das Management Studio ermöglicht die Erstellung von Datenbankdiagrammen. In der Baumansicht mit den Bestandteilen einer Datenbank befindet sich der Zweig **Datenbankdiagramme**, und sein Kontextmenü enthält das Item **Neues Datenbankdiagramm**. Nötigenfalls lässt man durch eine zustimmende Antwort auf die folgende Frage

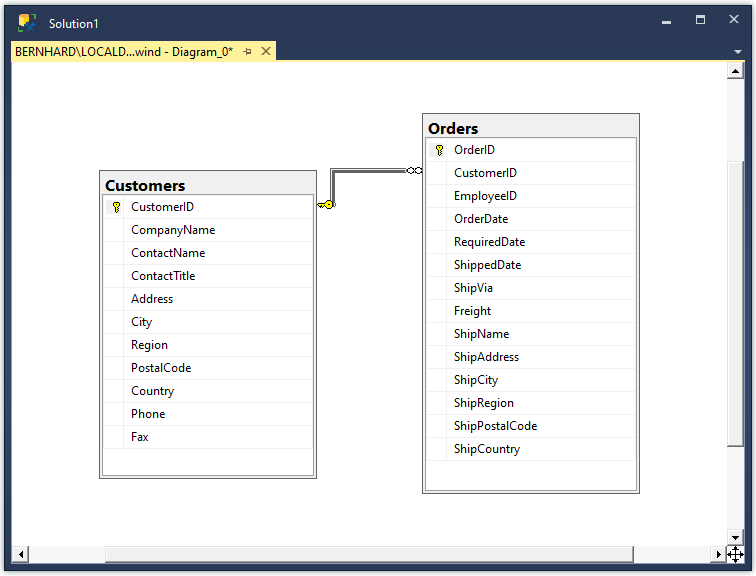


ein Hindernis für die Diagrammerstellung automatisch beseitigen.

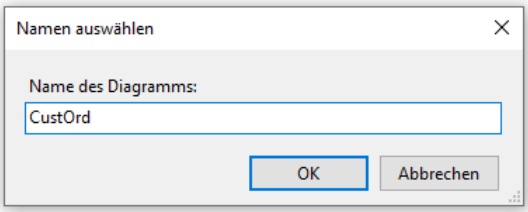
Es erscheint ein Dialog zur Auswahl der Tabellen, die samt Beziehungen dargestellt werden sollen, z. B.:



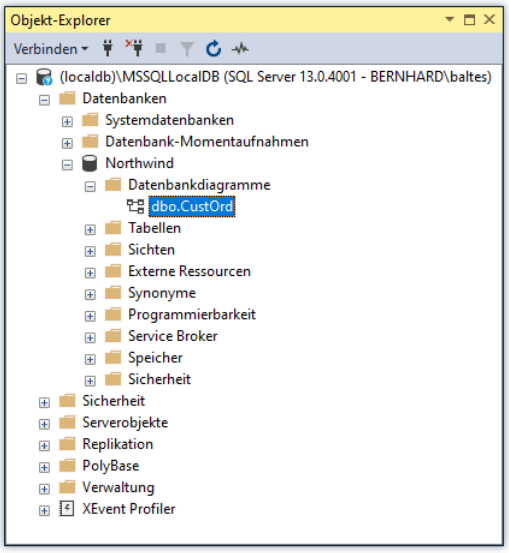
Klicken Sie auf **Hinzufügen** und nach Fertigstellung des Diagramms auf **Schließen**. Anschließend lässt sich das Diagramm in der Designer-Zone des Management Studios durch Verschieben von Tabellen und Beziehungspfeilen gestalten, z. B.:



Beim Speichern des Diagramms (per **Strg-S** oder Symbolschalter angefordert) legt man seinen Namen fest, z. B.:

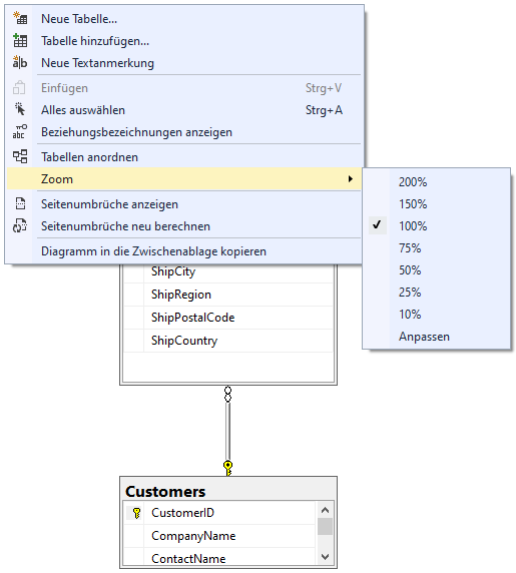


Das Diagramm kann später erneut geöffnet werden:



Das Kontextmenü zum Hintergrund des Datenbankdiagramm-Designers enthält nützliche Items:

- **Tabellen anordnen**
  Dieses Item bewirkt eine Anordnung der Tabellen.
- **Zoom**
  Das Item **Zoom** bietet neben festen Zoom-Stufen die Möglichkeit zur **Anpassung** des Diagramms an die verfügbare Fläche:

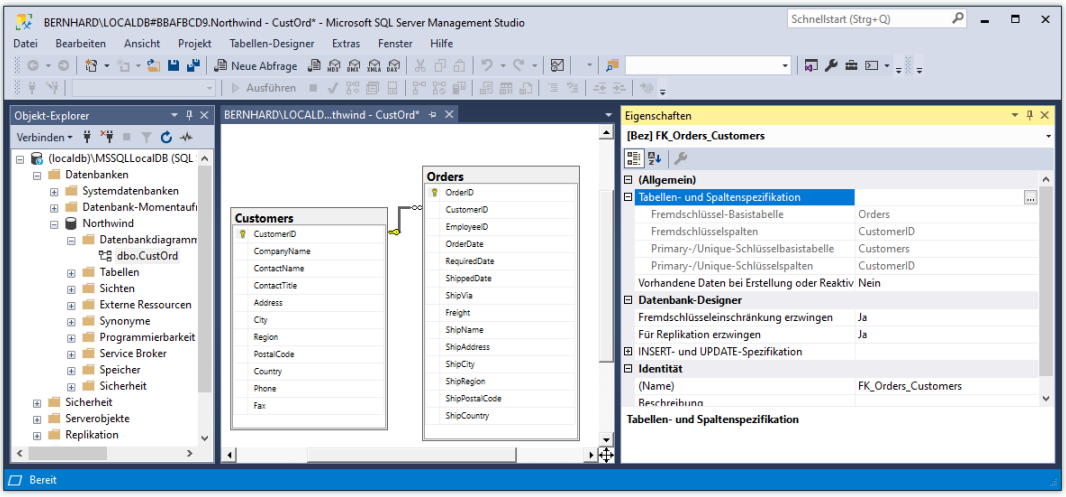


  Bei gedrückter **Strg**-Taste kann mit dem Mausrad eine Zoom-Stufe gewählt werden.
- **Diagramm in die Zwischenablage kopieren**

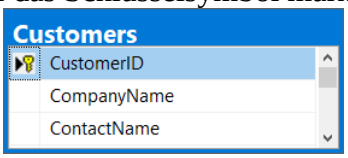
18.4.4 Datenbankschema modifizieren

Über ein Datenbankdiagramm lassen sich Primärschlüssel definieren und Beziehungen bearbeiten bzw. definieren:

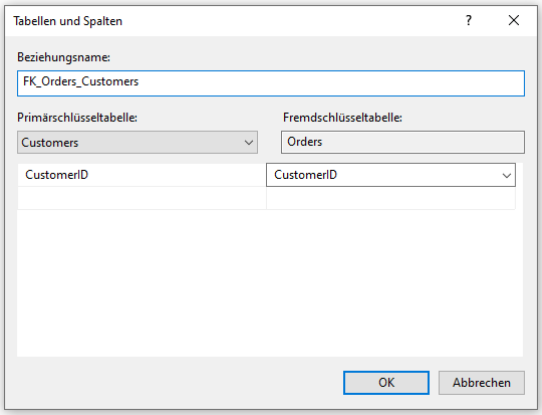
- Um eine Variable als Primärschlüssel für eine Tabelle festzulegen, wählt man aus ihrem Kontextmenü das Item **Primärschlüssel festlegen**.
- Eine markierte Beziehung lässt sich über das **Eigenschaften**-Fenster verwalten:



- Eine Beziehung kann über ihr Kontextmenü gelöscht werden.
- So lässt sich eine neue Master-Details - Beziehung definieren:
  - Primärschlüssel per Klick auf das Schlüsselsymbol markieren, z. B.:



  - Maus bei gedrückter linker Taste auf das Feld mit dem Fremdschlüssel ziehen und Taste loslassen. Über einem geeigneten Ziel wird der Mauszeiger durch ein Pluszeichen ergänzt.
  - In der Dialogbox **Tabellen und Spalten** lässt sich die **Fremdschlüsselspalte** festlegen:



18.5 SQL

Ein RDBMS interagiert mit anderen Softwaresystemen über die **Structured Query Language** (SQL). Microsofts SQL-Server bieten unter der Bezeichnung **T-SQL** (*Transact-SQL*) einen Dialekt mit etlichen Erweiterungen im Vergleich zum ISO/ANSI - Standard.[1]

[1] https://en.wikipedia.org/wiki/Transact-SQL

Bei einer 2021 von Stack Overflow, einer sehr populären Selbsthilfe-Webseite für Software-Entwickler, durchgeführten Befragung

https://insights.stackoverflow.com/survey/2021

landete die SQL auf Position 4 in der Rangordnung der populärsten Techniken. Das spricht dafür, dass sich die Beschäftigung mit der SQL trotz der Popularität von ORM-Lösungen (*Object Relational Mapping*, siehe Kapitel 20) noch lohnt.

18.5.1 Überblick

Wie der Name *SQL* nahe legt, ist die *Abfrage* von Informationen klarer Einsatzschwerpunkt, doch deckt der Sprachumfang alle Aufgaben der Datenverwaltung ab, wobei sich die folgenden Aufgabenbereiche unterscheiden lassen:

- **DDL** (**D**ata **D**efinition **L**anguage)
  Mit den DDL-Befehlen (z. B. **CREATE TABLE**, **CREATE INDEX**, **DROP TABLE**) definiert oder verändert man das Schema einer Datenbank. Man kann hier von *Definitionsabfragen* sprechen.
- **Abfragen per SELECT**
  Der außerordentlich wichtige **SELECT**-Befehl ermöglicht das Abrufen, Auswählen, Suchen und Auswerten von Datensätzen. Als Abfrageergebnis erhält man eine Menge von Datenzeilen (engl. *rowset*), die sich strukturell von den Zeilen der Datenbanktabellen unterscheiden können, z. B. wenn Daten aus mehreren Tabellen zusammengeführt werden. Beim **SELECT**-Befehl spricht man von einer *Auswahlabfrage*.
- **DML** (**D**ata **M**anipulation **L**anguage)
  Mit den DML-Befehlen werden Datenbanktabellen verändert:
  - **INSERT**
    Fügt neue Zeilen in eine Tabelle ein
  - **DELETE**
    Löscht Tabellenzeilen
  - **UPDATE**
    Ändert die Werte in Tabellenzeilen

  Man spricht hier auch von *Aktionsabfragen*.

Während in einer ADO.NET - Anwendung „eigenhändig“ formulierte **SELECT**-Kommandos nicht unüblich sind, kann man das Erstellen von zugehörigen **INSERT**-, **DELETE**- und **UPDATE**-Kommandos meist einem **CommandBuilder**-Objekt überlassen (vgl. Abschnitt 18.6.9).

Auf der Webseite

https://docs.microsoft.com/de-de/sql/t-sql/tutorial-writing-transact-sql-statements

bietet Microsoft ein T-SQL - Tutorial an.

18.5.2 Abfragen per SELECT-Anweisung

Die folgende Syntaxbeschreibung zum **SELECT**-Befehl beschränkt sich auf die anschließend besprochenen Ausdrucksmittel:

```
SELECT  {* | spalten}
   FROM tabellen
   [WHERE logischer_ausdruck]
   [ORDER BY sortierkriterium [{ASC | DESC}] [, sortierkriterium [{ASC | DESC}] ...]]
   [GROUP BY gruppierungs_spalte [, gruppierungs_spalte ...]]
```

Hier werden spezielle Beschreibungstechniken verwendet, die aus naheliegenden Gründen für die C# - Syntax nicht verwendbar wären:

- Eckige Klammern begrenzen optionale Elemente.
- Zwischen geschweiften Klammern und durch senkrechte Striche getrennt stehen Optionen, von denen genau *eine* zu wählen ist. Bei einer *optionalen* Auswahlliste ist der Voreinstellungswert unterstrichen.
- Durch drei aufeinanderfolgende Punkte wird zum Ausdruck gebracht, dass eine Liste beliebig verlängert werden darf.

Im Unterschied zu C# ist in SQL die Groß-/Kleinschreibung irrelevant. Es hat sich aber eingebürgert, die Schlüsselwörter groß zu schreiben.

18.5.2.1 Spalten einer Tabelle abrufen

Sollen per **SELECT**-Befehl *alle* Spalten einer Tabelle abgerufen werden, dann ist ein Stern zu setzen (*). Mehrere einzelne Spalten sind durch Kommata getrennt aufzulisten.

In der **FROM**-Klausel wird die Tabelle genannt, aus der die abgerufenen Spalten stammen.

Beispiel:

```
SELECT EmployeeID, FirstName, LastName FROM Employees
```

18.5.2.2 Fälle auswählen

Mit der **WHERE**-Klausel der **SELECT**-Anweisung kann über einen logischen Ausdruck eine Teilmenge von Zeilen ausgewählt werden, z. B.:

```
SELECT EmployeeID, FirstName, LastName FROM Employees
  WHERE City='London'
```

Wie das Beispiel zeigt, sind gemäß ISO/ANSI - Standard Zeichenkettenliterale durch *einfache* Anführungszeichen zu begrenzen. Um ein Hochkomma zum Bestandteil einer Zeichenfolge zu machen, verdoppelt man es. Bei der Verwendung von Anführungszeichen halten sich aber nicht alle Datenbankverwaltungssysteme strikt an den Standard.

Treten in *Namen* (z. B. von Tabellen oder Spalten) Schlüsselwörter, Sonderzeichen oder Leerzeichen auf, dann sind diese Namen durch *doppelte* Anführungszeichen zu begrenzen, z. B.:

```
SELECT EmployeeID, "First Name", "Last Name" FROM Employees
  WHERE City='London'
```

In dem von Microsofts SQL-Server verwendeten SQL-Dialekt (T-SQL) werden Namen mit kritischen Bestandteilen durch eckige Klammern begrenzt, z. B.:

```
SELECT EmployeeID, [First Name], [Last Name] FROM Employees
  WHERE City='London'
```

Für die Suche nach einem Zeichenfolgenmuster bietet SQL den Vergleichsoperator **LIKE**, wobei die folgenden Jokerzeichen zur Verfügung stehen:

%	ersetzt null, ein oder mehrere beliebige Zeichen
_	ersetzt genau ein beliebiges Zeichen

Der folgende Befehl spürt alle Personen auf, deren Nachname mit „P“ beginnt:

```
SELECT FirstName, LastName FROM Employees
  WHERE LastName LIKE 'P%'
```

Leere Feldinhalte lassen sich mit dem Schlüsselwort **NULL** ansprechen, z. B.:

```
SELECT FirstName, LastName FROM Employees
  WHERE Region IS NULL
```

Um die Fälle mit einem *vorhandenen* Wert auszuwählen, setzt man den Operator **NOT** vor das Schlüsselwort **NULL**, z. B.:

```
SELECT FirstName, LastName FROM Employees
  WHERE Region IS NOT NULL
```

Mit dem Vergleichsoperator **IN** stellt man für einen Ausdruck fest, ob seine aktuelle Ausprägung in einer Liste von Trefferwerten auftritt, z. B.:

```
SELECT FirstName,LastName,City FROM Employees
  WHERE City IN ('London','Seattle')
```

18.5.2.3 Daten aus mehreren Tabellen zusammenführen

Für zwei Tabellen, die zueinander in einer Master-Details- Beziehung stehen (vgl. Abschnitt 18.2.2), benötigt man oft eine Abfrageergebnistabelle, die alle Detail-Zeilen mit zusätzlichen Master-Daten enthält. Im folgenden Beispiel wird zu jeder Zeile aus der Detail-Tabelle `Orders` das Merkmal `CompanyName` aus der Master-Tabelle `Customers` ergänzt:

```
SELECT Customers.CompanyName, Orders.OrderDate
  FROM Customers, Orders
  WHERE Customers.CustomerID = Orders.CustomerID
```

Sobald die **FROM**-Klausel *mehrere* Tabellen enthält, muss den Namen der abgerufenen Spalten ein Tabellenname vorangestellt werden. Der bei Beteiligung mehrerer Tabellen anfallende Schreibaufwand lässt sich durch Abkürzungen (Aliasnamen) begrenzen, z. B.:

```
SELECT C.CompanyName, B.OrderDate FROM Customers C, Orders B
 WHERE C.CustomerID = B.CustomerID
```

Zwischen einem Tabellennamen und seinem Aliasnamen kann optional das Schlüsselwort AS stehen, z. B.:

```
SELECT C.CompanyName, B.OrderDate FROM Customers AS C, Orders AS B
 WHERE C.CustomerID = B.CustomerID
```

Ohne **WHERE**-Klausel werden im Beispiel die beiden Tabellen nach einem zwar systematischen, aber wohl nur selten sinnvollen Verfahren zusammengeführt: Jeder Fall der ersten Tabelle wird mit jedem Fall der zweiten Tabelle kombiniert. Die obige **WHERE**-Klausel sorgt dafür, dass aus einer Zeile der `Customers` - Tabelle und einer Zeile der `Orders` - Tabelle nur dann eine Zeile im Abfrageergebnis entstehen soll, wenn beide Zeilen im Feld `CustomerID` denselben Wert haben. Folglich enthält das Abfrageergebnis für jede Bestellung eine Zeile, in der neben dem Bestelldatum auch die Firmenbezeichnung des Kunden auftritt.

Dieselbe Ergebnismenge erhält man auch über die Operation **INNER JOIN**, z. B.:

```
SELECT Customers.CompanyName, Orders.OrderDate
  FROM Customers INNER JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID
```

18.5.2.4 Abfrageergebnis sortieren

Mit der Klausel **ORDER BY** lässt sich das Abfrageergebnis (auf- oder absteigend) sortieren, wobei Felder oder numerische Ausdrücke als Sortierkriterien in Frage kommen, z. B.:

```
SELECT C.CompanyName, B.OrderDate FROM Customers C, Orders B
  WHERE C.CustomerID = B.CustomerID
  ORDER BY B.OrderDate DESC
```

18.5.2.5 Auswertungsfunktionen

Auf die in der **SELECT**-Anweisung angeforderten Spalten können Auswertungsfunktionen angewendet werden. Im folgenden Beispiel wird der mittlere Preis aller Fälle in der **Northwind**-Tabelle `Products` festgestellt:

```
SELECT AVG(UnitPrice) FROM Products
```

Wichtige SQL-Auswertungsfunktionen sind:

Funktion	Beschreibung
COUNT(ALL *col*)	Liefert die Anzahl der von **NULL** verschiedenen Werte in der Spalte *col*
COUNT(DISTINCT *col*)	Liefert die Anzahl der unterschiedlichen und von **NULL** verschiedenen Werte in der Spalte *col*
COUNT(*)	Liefert die Anzahl der Zeilen
SUM(*col*)	Summiert über die Spalte *col*
AVG(*col*)	Mittelt über die Spalte *col*
MIN(*col*)	Ermittelt das Minimum in der Spalte *col*
MAX(*col*)	Ermittelt das Maximum in der Spalte *col*

Verwendet man die Auswertungsfunktionen zusammen mit einer *Gruppierung* (siehe nächsten Abschnitt), dann erhält man das Auswertungsergebnis für jede Gruppe (statt für die gesamte Ergebnistabelle).

18.5.2.6 Daten gruppieren

Über die Klausel **GROUP BY** werden die Zeilen einer Tabelle in Gruppen eingeteilt, die jeweils durch einen eindeutigen Wert der zur Gruppierung (man sagt auch: zum *Aggregieren*) herangezogenen Spalte definiert sind. Die im Abschnitt 18.5.2.5 beschriebenen Funktionen werden dann für jede Gruppe ausgewertet. Im folgenden Beispiel wird die Variable `Country` zur Gruppierung verwendet, um die Kundenzahlen in den einzelnen Herkunftsländern festzustellen:

```
SELECT Country, COUNT(*) FROM Customers GROUP BY Country
```

18.5.3 Datenmanipulation

18.5.3.1 INSERT

Mit dem **INSERT**-Kommando trägt man eine neue Zeile in eine Tabelle ein, wobei zwei korrespondierende Listen mit Feldnamen und typkompatiblen Werten anzugeben sind, z. B.:

```
INSERT INTO Employees (LastName, FirstName) VALUES ('Meyer', 'Danny')
```

Beim Einfügen neuer Tabellenzeilen erhalten Spalten ohne explizite Versorgung den im **CREATE TABLE** - Kommando per **DEFAULT**-Schlüsselwort definierten Voreinstellungswert. Soll ein Wert explizit fehlen, dann ist das Schlüsselwort **NULL** anzugeben, was bei Spalten mit der Restriktion **NOT NULL** allerdings verboten ist. Ist für eine Spalte das automatische Generieren der Werte eingestellt (z. B. beim Primärschlüssel `EmployeeID`), dann wird ein INSERT-Kommando mit Wertangabe als fehlerhaft abgelehnt.

Um mit einem **INSERT**-Kommando *mehrere* Zeilen anzulegen, lässt man entsprechend viele Wertelisten durch Kommata getrennt aufeinander folgen, z. B.:

```
INSERT INTO Employees (LastName, FirstName)
  VALUES ('Meyer', 'Danny'), ('Biden', 'Joe')
```

Ob neue Zeilen grundsätzlich am Ende einer Tabelle angehängt werden, oder ob z. B. vorhandene und als gelöscht markierte Zeilen ggf. überschrieben werden, das ist ein nebensächliches Implementierungsdetail.

18.5.3.2 DELETE

Mit dem **DELETE**-Kommando löscht man Zeilen aus einer Tabelle, z. B.:

```
DELETE FROM Orders WHERE EmployeeID > 10
```

Hier werden alle Zeilen mit einer `EmployeeID` oberhalb von 10 aus der Tabelle `Orders` gelöscht bzw. als gelöscht markiert.

Anschließend könnten übrigens die betroffenen Mitarbeiter ohne Verstoß gegen die referentielle Integrität aus der `Employees`-Tabelle gelöscht werden:

```
DELETE FROM Employees WHERE EmployeeID > 10
```

Mit der **WHERE**-Klausel haben wir uns im Abschnitt 18.5.2.2 beschäftigt.

18.5.3.3 UPDATE

Mit dem **UPDATE**-Kommando verändert man die Werte bestimmter Spalten, wobei der Effekt meist per **WHERE**-Klausel (siehe Abschnitt 18.5.2.2) auf bestimmte Zeilen eingeschränkt wird, z. B.:

```
UPDATE Employees SET LastName = 'Fuller' WHERE EmployeeID = 5;
```

Hier wird der Familienname eines Mitarbeiters (vermutlich wegen Eheschließung) verändert.

In der **SET**-Klausel dürfen auch *mehrere* Wertezuweisungen (Name-Wert - Paare) durch Kommata getrennt aufeinander folgen.

18.6 ADO.NET

Die ADO.NET - Bibliothek enthält essentielle Klassen für Datenbankzugriffe durch .NET - Programme. Man verwendet diese Klassen ...

- entweder direkt mit viel Transparenz und Handarbeit (z.B. bei der Formulierung von SQL-Kommandos),
- oder mit Hilfe von datenbank-spezifisch durch einen Assistenten generierten Spezialisierungen (siehe Abschnitt 18.7 über typisierte DataSets),
- oder als kaum noch sichtbare technische Basis der ORM-Lösung (*Object Relational Mapping*) Entity Framework Core (siehe z. B. Kapitel 20).

Im aktuellen Abschnitt lernen wir die wesentlichen ADO.NET - Klassen im traditionellen Direktkontakt kennen. Auf diese Weise lassen sich auch heute noch konkurrenzfähige Anwendungen erstellen, wobei auf dem Weg zur Beherrschung der Materie eine Monographie mit einer hohen dreistelligen Seitenzahl von Nutzen ist (z.B. Hamilton & MacDonald 2003).

18.6.1 Überblick

18.6.1.1 Verbindungsloses versus verbindungsorientiertes Arbeiten

Beim Datenbankzugriff ist in der Regel das *verbindungslose* Arbeiten zu bevorzugen, wobei die relevanten Daten vom DBMS bezogen, in einem lokalen **DataSet**-Objekt gespeichert, bearbeitet und ggf. anschließend im geänderten Zustand wieder zur Datenbank übertragen werden.

Beim Durchsuchen sehr großer Datenbestände ist es aber weniger sinnvoll, mit einer lokalen Kopie zu arbeiten. Für solche Anwendungen bieten die ADO.NET – Provider (siehe Abschnitt 18.6.1.2) eine Implementierung der Schnittstelle **IDataReader** (z. B. **SqlDataReader**). Diese Klassen erlauben ein lesendes, sequentielles Durchlaufen der angeschlossenen Datenbank. Im Unterschied zum **DataSet**-Einsatz besteht eine permanente Verbindung zur Datenbank.

18.6.1.2 Provider

Für die Kooperation mit einem bestimmten DBMS wird ein sogenannter *Provider* benötigt, worunter eine Sammlung von .NET - Klassen zu verstehen ist, die gemeinsam den Zugriff auf ein bestimmtes DBMS unterstützen. Im .NET - Ökosystem sind Provider für viele wichtige Datenbankmanagementsysteme verfügbar. Wir werden im Kurs nur den Provider **SqlClient** verwenden, der für Microsofts SQL Server (inkl. Express Edition und LocalDB-Variante) zuständig ist.

Die Datenbank-Provider sind entweder in der Standardinstallation der .NET - Laufzeitumgebung enthalten (beim .NET Framework oft der Fall) oder als NuGet-Pakete verfügbar. Der Provider **SqlClient** ist sogar doppelt vorhanden:[1]

- Im Namensraum **System.Data.SqlClient** befindet sich die klassische Variante mit der vollen Unterstützung der ADO.NET - Technik (inkl. **DbDataAdapter**-Klasse).
- Im Namensraum **Microsoft.Data.SqlClient** befindet sich eine modernisierte, ganz auf das Entity Framework Core (siehe Kapitel 20) eingestellte Variante (*ohne* **DbDataAdapter**-Klasse).

Nach Rücksprache mit erfahrenen IT-Praktikern wird im aktuellen Kapitel 18 dem klassischen, vollständigen Provider der Vorzug gegeben. Microsoft plante ursprünglich, die **DbDataAdapter**-Klassen aus .NET Core fernzuhalten, musste diese Entscheidung aber unter dem Druck der Praxis revidieren.

Zum Provider **SqlClient** aus dem Namensraum **System.Data.SqlClient** gehören die folgenden Klassen:

- **SqlConnection**
  Diese Klasse erlaubt das Konfigurieren und Öffnen einer Verbindung zum SQL Server (siehe Abschnitt 18.6.3).
- **SqlCommand**
  Diese Klasse repräsentiert ein SQL-Kommando (siehe Abschnitt 18.6.4).
- **SqlDataAdapter**
  Diese Klasse vermittelt zwischen der Datenbank und den lokal (in einem **DataSet**-Objekt) zwischengespeicherten Daten (siehe Abschnitt 18.6.5).
- **SqlDataReader**
  Diese Klasse erlaubt ein lesendes, sequentielles Durchlaufen der angeschlossenen Datenbank (siehe Abschnitt 18.6.12).

---

[1] https://devblogs.microsoft.com/dotnet/introducing-the-new-microsoftdatasqlclient/

- **SqlParameter**
  Diese Klasse kommt zum Einsatz bei parametrisierten Abfragen (siehe Abschnitt 18.6.4.2) und bei der Übergabe von Parametern an gespeicherte Prozeduren im DBMS.[1]
- **SqlTransaction**
  Wenn für eine Sequenz von Datenbank-Modifikationen sichergestellt ist, dass entweder *alle* Modifikationen erfolgreich ausgeführt werden, oder *überhaupt keine* Datenbankveränderung stattfindet, dann bezeichnet man diese Sequenz als *Transaktion*.[2]

Weil die Klassen eines Providers jeweils eine bestimmte Schnittstelle zu implementieren haben (z. B. **IDbConnection** bei **SqlConnection**), ist der Wechsel zu einem anderen Provider bei geschickter Programmierung im Prinzip mit relativ wenigen Quellcodeänderungen verbunden. In der Praxis kann die Migration aber doch durch unterschiedliche SQL-Implementationen der Datenbankmanagementsysteme erschwert werden.

18.6.1.3 Provider-unabhängige Klassen

Im Namensraum **System.Data** befinden sich provider*un*abhängige Klassen, die meist mit der klientenseitigen, verbindungslosen Verarbeitung von Daten beschäftigt sind:

- **DataSet**
  Repräsentiert im lokalen Speicher Tabellen, Restriktionen und Beziehungen, die in der Regel aus *einer* Datenbank stammen, grundsätzlich aber auch aus verschiedenen Quellen zusammengeführt werden können.
- **DataTable, DataTableCollection**
  Ein **DataTable**-Objekt repräsentiert eine Tabelle. Die **DataSet**-Eigenschaft **Tables** zeigt auf ein Objekt der Klasse **DataTableCollection** mit allen im **DataSet**-Objekt enthaltenen Tabellen.
- **DataColumn, DataColumnCollection**
  Ein **DataColumn**-Objekt repräsentiert eine Tabellenspalte und enthält z. B. in seiner **DataType**-Eigenschaft den Typ der enthaltenen Daten. Die **DataTable**-Eigenschaft **Columns** zeigt auf ein Objekt der Klasse **DataColumnCollection** mit allen im **DataTable**-Objekt enthaltenen Spalten. In der **DataColumnCollection** befinden sich also nicht die Daten einer Tabelle, sondern die Schema-Informationen der Tabelle.
- **DataRow, DataRowCollection**
  Ein **DataRow**-Objekt repräsentiert eine Tabellenzeile. Die **DataTable**-Eigenschaft **Rows** zeigt auf ein Objekt der Klasse **DataRowCollection** mit allen im **DataTable**-Objekt enthaltenen Zeilen. In der **DataRowCollection** befinden sich also die Daten einer Tabelle. Auf die einzelnen Werte einer Tabellenzeile kann man über die **DataRow**-Eigenschaft **ItemArray** mit dem Datentyp **Object[]** zugreifen. Die Zuweisung von Werten mit fehlerhaftem Datentyp wird erst zur Laufzeit anhand der Schema-Informationen in der **DataColumnCollection** bemerkt.

---

[1] Gespeicherte Prozeduren, die aus Zeitgründen im Manuskript leider nicht behandelt werden können, sind aus Performanzgründen zu empfehlen, wenn identische SQL-Anweisungen wiederholt benötigt werden.

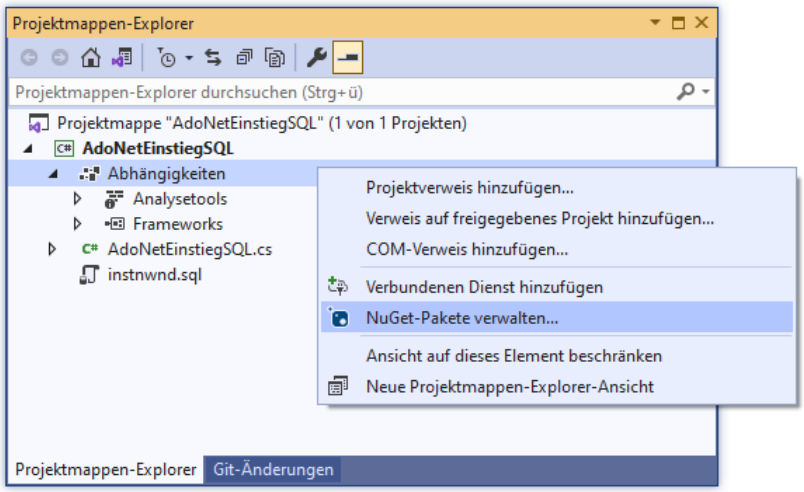
[2] Im Manuskript können Transaktionen aus Zeitgründen leider nicht behandelt werden.

- **DataRelation, DataRelationCollection**
  Ein **DataRelation**-Objekt repräsentiert eine Beziehung zwischen zwei **DataTable**-Objekten (vgl. Abschnitt 18.2.2). Die **DataTable**-Eigenschaft **ChildRelations** zeigt auf ein Objekt der Klasse **DataRelationCollection** mit allen **DataRelation**-Objekten, an denen das **DataTable**-Objekt in der Master-Rolle beteiligt ist. Die **DataTable**-Eigenschaft **ParentRelations** zeigt auf ein Objekt der Klasse **DataRelationCollection** mit allen **DataRelation**-Objekten, an denen das **DataTable**-Objekt in der Details-Rolle beteiligt ist.
- **Constraint, ConstraintCollection**
  Ein **Constraint**-Objekt repräsentiert eine Regel zur Sicherung der Datenintegrität. Spezialisierte Unterklassen sind **ForeignKeyConstraint** und **UniqueConstraint**. Die **DataTable**-Eigenschaft **Constraints** zeigt auf ein Objekt der Klasse **ConstraintCollection** mit allen für das **DataTable**-Objekt definierten Regeln.

In einer .NET - Quellcodedatei mit Datenbankzugriff ist es meist sinnvoll, den Namensraum **System.Data** sowie den Namensraum zum eingesetzten Provider (z. B. **System.Data.SqlClient**) zu importieren.

18.6.2 Beispielprogramm

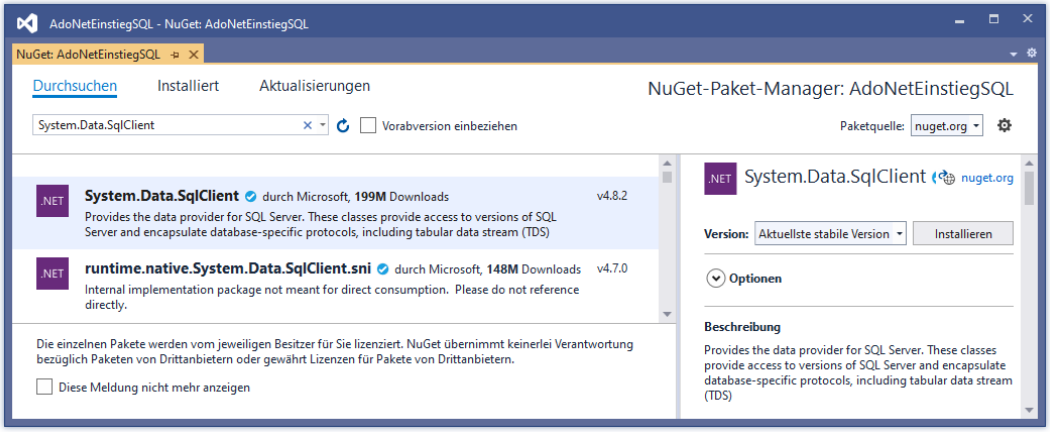
Bevor die Sache zu komplex und abstrakt wird, betrachten wir ein einfaches Beispielprogramm in einem Konsolenprojekt. Weil wir mit .NET 5.0 arbeiten und den Provider **System.Data.SqlClient** verwenden wollen, muss das passende NuGet-Paket installiert werden. Dazu wählen wir im **Projektmappen-Explorer** aus dem Kontextmenü zu den **Abhängigkeiten** das Item **NuGet-Pakete verwalten**:



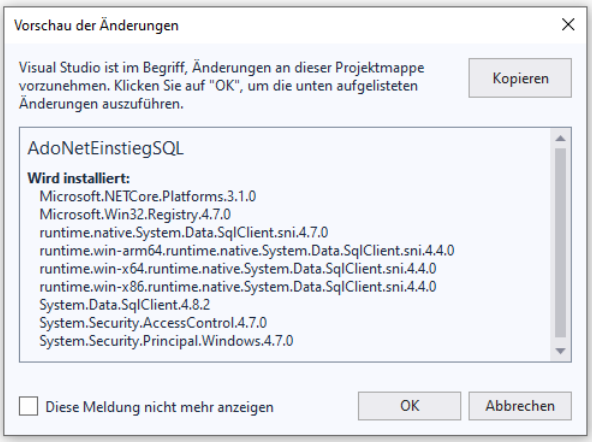
Daraufhin erscheint der **NuGet-Paket-Manager**, der folgende Leistungen anbietet:

- Die Webseite https://www.nuget.org/ nach Paketen **durchsuchen**
- Die im Projekt **installierten** Pakete verwalten
- Die Pakete des Projekts **aktualisieren**

Mit Hilfe des Suchfelds ist das benötigte Paket schnell gefunden:



Wir veranlassen das **Installieren** der Version 4.8.2 und bestätigen per Klick auf **OK**:



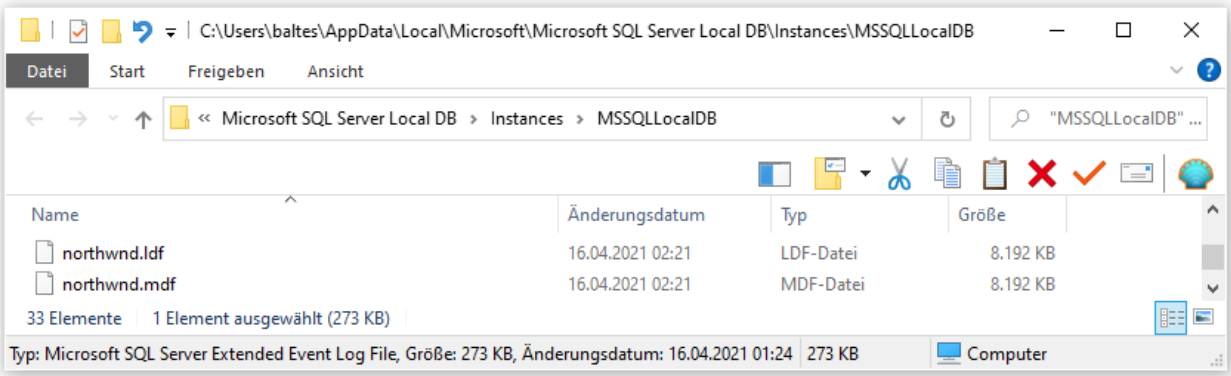
In der Projektdatei (Namenserweiterung **.csproj**) erscheint daraufhin das folgende **PackageReference**-Element:

```
<ItemGroup>
  <PackageReference Include="System.Data.SqlClient" Version="4.8.2" />
</ItemGroup>
```

Wegen der im **SqlConnection**-Konstruktor verwendeten Verbindungszeichenfolge (siehe Abschnitt 18.6.3.1)

```
string cs = "Data Source=(LocalDb)\\MSSQLLocalDB; " +
            "Initial Catalog=Northwind; Integrated Security=true";
using SqlConnection dbConnection = new(cs);
```

öffnet das Programm die im voreingestellten Datenbankordner des LocalDB-Servers befindliche Version der **Northwind**-Datenbank (abgelegt in den Dateien **northwnd.mdf** (*Main Database File*) und **northwnd.ldf** (*Log Data File*), siehe Abschnitt 18.3.2 zur Installation der Datenbank):



Unter Beteiligung der Klassen **SqlConnection**, **SqlCommand**, **SqlDataAdapter** und **DataSet** wird ein SELECT-Kommando in der Sprache SQL (siehe Abschnitt 18.5) an das DBMS geschickt, um drei Spalten aus der Datenbanktabelle `Employees` abzurufen:

```
SqlCommand selCommand =
  new("SELECT EmployeeID, FirstName, LastName FROM Employees", dbConnection);
SqlDataAdapter dbAdapter = new();
dbAdapter.SelectCommand = selCommand;
dbAdapter.TableMappings.Add("Table", "Employees");
DataSet ds = new();
dbConnection.Open();
dbAdapter.Fill(ds);
dbConnection.Close();
```

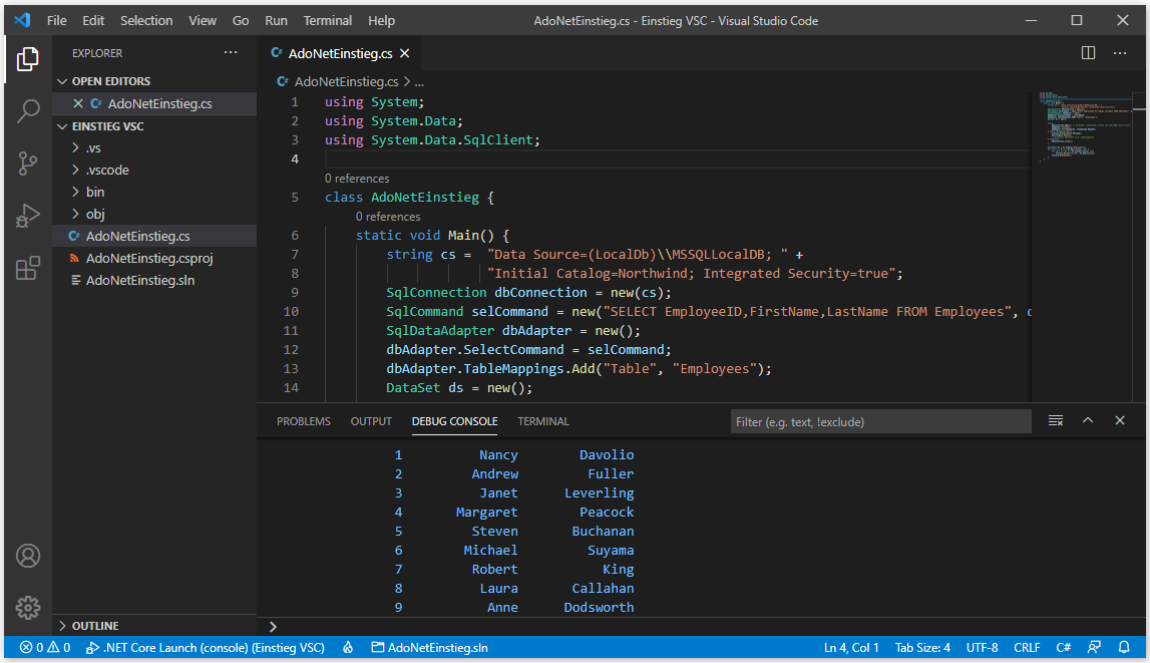
Schlussendlich resultiert ein Objekt der Klasse **DataTable** mit den Spalten `EmployeeID`, `FirstName` und `LastName` aus der Datenbanktabelle `Employees`. Auf die Zellen in dieser Tabelle kann über die Indexer-Syntax zugegriffen werden, was im folgenden Programm, das später noch näher erläutert wird, zu einer Konsolenausgabe genutzt wird:

```
using System;
using System.Data;
using System.Data.SqlClient;

class AdoNetEinstieg {
  static void Main() {
    string cs = "Data Source=(LocalDb)\\MSSQLLocalDB; " +
                "Initial Catalog=Northwind; Integrated Security=SSPI";
    using SqlConnection dbConnection = new(cs);

    SqlCommand selCommand = new("SELECT EmployeeID, FirstName, LastName FROM Employees",
                                dbConnection);
    SqlDataAdapter dbAdapter = new();
    dbAdapter.SelectCommand = selCommand;
    dbAdapter.TableMappings.Add("Table", "Employees");
    DataSet ds = new();

    try {
      dbConnection.Open();
      dbAdapter.Fill(ds);
      dbAdapter.FillSchema(ds, SchemaType.Mapped);
    } catch (Exception ex) {
      Console.WriteLine(ex.Message);
      Environment.Exit(1);
    } finally {
      dbConnection.Close();
    }

    DataTable dt = ds.Tables["Employees"];
    for (int i = 0; i < dt.Rows.Count; i++) {
      for (int j = 0; j < dt.Columns.Count; j++)
        Console.Write("{0,15}", dt.Rows[i][j]);
      Console.WriteLine();
    }
  }
}
```

In der Ausgabe erscheinen die aus der Datenbank kopierten Informationen:

```
1          Nancy        Davolio
2         Andrew         Fuller
3          Janet      Leverling
4       Margaret       Peacock
5         Steven       Buchanan
6        Michael         Suyama
7         Robert           King
8          Laura       Callahan
9           Anne      Dodsworth
```

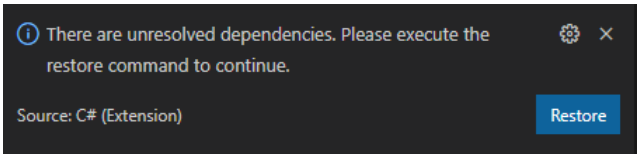
Das mit dem Visual Studio 2019 erstellte Projekt lässt sich ohne Änderung auch vom Visual Studio Code öffnen und ausführen:



Dazu muss ...

- die LocalDB-Edition von Microsofts SQL-Server installiert sein (siehe Abschnitt 18.1.2),
- das NuGet-Paket mit dem ADO.NET - Provider **System.Data.SqlClient** installiert sein,
- die **Northwind**-Datenbank im erwarteten Ordner vorhanden sein.

Auf ein fehlendes NuGet-Paket macht das VS Code aufmerksam:



Sollte die freundliche Aufforderung unterbleiben, kann man in der internen Console (bei englischer Bedienoberfläche zu öffnen mit **View > Terminal**) das folgende Kommando abschicken, um fehlende NuGet-Pakete zu installieren:

```
> dotnet restore
```

Als Ablageort für die Datenbankdateien kann statt der LocalDB-Voreinstellung ein beliebiger Ordner verwendet werden, wobei die Verbindungszeichenfolge abzuändern ist. Wenn sich die Dateien **northwnd.mdf** (*Main Database File*) und **northwnd.ldf** (*Log Data File*) z. B. im Ordner

U:\Eigene Dateien\C#\BspUeb\ADO.NET\Northwind

befinden, dann eignet sich die folgende Verbindungszeichenfolge:

```
string dbFile = @"U:\Eigene Dateien\C#\BspUeb\ADO.NET\Northwind\northwnd.mdf";
string cs = "Data Source=(LocalDb)\\MSSQLLocalDB; " +
    "Database=Northwind; AttachDbFilename=" + dbFile  + "; Integrated Security=true";
```

18.6.3 DbConnection

Wir verwenden als DBMS die Microsoft SQL Server Express Edition, zunächst in der LocalDB-Variante (Einbenutzerdatenbank) und später auch in der Client-Server - Variante. In beiden Fällen kommt der ADO.NET - Provider **SqlClient** zum Einsatz, und für die Verbindung zur Datenbank ist die von **DbConnection** abstammende Klasse **SqlConnection** aus dem Namensraum **System.Data.SqlClient** zuständig.[1] Andere Provider enthalten analoge **DbConnection**-Ableitungen.

18.6.3.1 Verbindungszeichenfolge

Das **SqlConnection**-Objekt muss wissen, ...

- welcher SQL-Server kontaktiert werden soll,
- welche dort verwaltete Datenbank benutzt werden soll,
- mit welchem Benutzerkonto der Zugriff erfolgen soll.

Die erforderlichen Informationen werden in der sogenannten *Verbindungszeichenfolge* zusammengestellt.

Das im Abschnitt 18.6.2 vorgestellte Beispielprogramm enthält die folgenden Anweisungen zur Deklaration und Initialisierung einer Instanzvariablen vom Typ **SqlConnection**:

```
string cs = "Data Source=(LocalDb)\\MSSQLLocalDB; " +
            "Initial Catalog=Northwind; Integrated Security=true";
using SqlConnection dbConnection = new(cs);
```

Die verwendete **SqlConnection**-Konstruktorüberladung erhält als Parameter die Verbindungszeichenfolge, die schlussendlich in der **SqlConnection**-Eigenschaft **ConnectionString** landet.

Weil bei Verwendung der LocalDB-Variante das DBMS und das Anwendungsprogramm auf demselben Rechner laufen, genügt eine einfach aufgebaute Verbindungszeichenfolge mit wenigen, jeweils durch ein Semikolon getrennten (Parameter=Wert) - Paaren:

- **SQL-Server**
  Von einem SQL-Server können auf einem Rechner mehrere Instanzen vorhanden sein, und die anzusprechende Instanz ist mit der folgenden Syntax zu benennen:

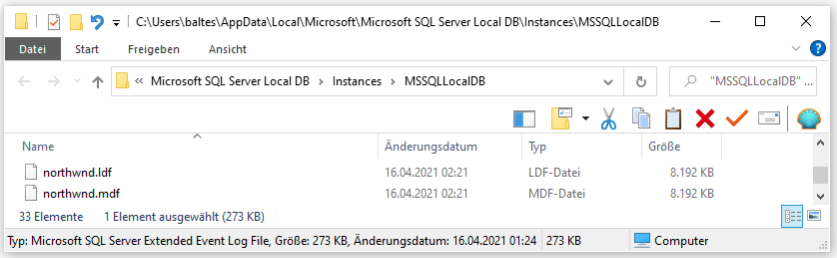
  Data Source=(LocalDb)*Instanzname*

  Wir haben bei der LocalDB-Installation den voreingestellten Instanznamen MSSQLLocalDB beibehalten:

  ```
  Data Source=(LocalDb)\MSSQLLocalDB
  ```

[1] Zur Entscheidung gegen den alternativen **SqlClient**-Provider im Namensraum **Microsoft.Data** siehe Abschnitt 18.6.1.2.

- **Datenbank**
  Die im voreingestellten Datenbankordner des LocalDB-Servers befindliche Version der **Northwind**-Datenbank mit den Dateien **northwnd.mdf** (*Main Database File*) und **northwnd.ldf** (*Log Data File*)



  kann über ihren logischen Namen angesprochen werden:

```
Initial Catalog=Northwind
```

  Der Parametername **Initial Catalog** kann ersetzt werden durch **Database**:

```
Database=Northwind
```

- **Benutzer-Authentifizierung**
  Microsofts SQL-Server unterstützen die Windows - und die SQL Server - Authentifizierung. Um die von Microsoft nachdrücklich empfohlene Windows - Authentifizierung in einem Datenbank-Klientenprogramm zu nutzen, setzt man den Parameter **Integrated Security** auf den Wert **true** oder **SSPI**:

```
Integrated Security=true
```

  Dann muss man sich nicht beschäftigen, wie sich sensible Daten (z. B. per **Secret Manager**) aus der Verbindungszeichenfolge heraushalten lassen.[1]

- **Pfad zur Datenbankdatei**
  Wenn sich die Datenbankdatei *nicht* im voreingestellten Datenbankordner des LocalDB-Servers befindet, dann ist der Pfadname als Wert des Parameters **AttachDbFilename** anzugeben, z. B.:

```
string dbFile = @"U:\Eigene Dateien\C#\BspUeb\ADO.NET\Northwind\northwnd.mdf";
dbConnection.ConnectionString = @"Data Source=(LocalDb)\MSSQLLocalDB; " +
                                "Database=Northwind; AttachDbFilename=" +
                                dbFile + "; Integrated Security=true";
```

Die zum ADO.NET - Provider **SqlClient** vorhandene Klasse **SqlConnectionStringBuilder** kann die Erstellung einer Verbindungszeichenfolge nicht entscheidend erleichtern, z. B.:

```
SqlConnectionStringBuilder csb = new();
csb.DataSource = "(LocalDb)\\MSSQLLocalDB";
csb.IntegratedSecurity = true;
csb.InitialCatalog = "Northwind";
Console.WriteLine(csb.ConnectionString);
```

Im Beispiel resultiert die Ausgabe

```
Data Source=(LocalDb)\MSSQLLocalDB;Initial Catalog=Northwind;Integrated Security=True
```

Weitere Hinweise zur Verbindungszeichenfolge:

- Die Groß-/Kleinschreibung ist irrelevant.
- Die Eigenschaft **ConnectionString** darf im Programm nur bei geschlossener Datenbankverbindung geändert werden.

---

[1] https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets

18.6.3.2 Eigenschaften, Ereignisse und Methoden

Einige Eigenschaften der **DbConnection**-Klassen erlauben einen Lesezugriff auf einzelne Parameter der Verbindungszeichenfolge. Bei der Klasse **SqlConnection** können auf diese Weise z. B. die get-only - Eigenschaften **DataSource** und **Database** abgefragt werden. Über die ebenfalls auf Lesezugriffe beschränkte Eigenschaft **State** vom Typ **ConnectionState** ist von einem **SqlConnection**-Objekt u. a. zu erfahren, ob die Verbindung momentan offen oder geschlossen ist (Werte **Open** bzw. **Closed**).

Über die Ereignisse **InfoMessage** bzw. **StateChanged** informiert ein **DbConnection**-Objekt über Probleme mit dem SQL-Server bzw. über Änderungen im Verbindungsstatus.

Wenn Sie eine Datenbankverbindung explizit öffnen (über die **SqlConnection**-Methode **Open()**), dann müssen Sie diese Verbindung möglichst frühzeitig wieder schließen. Das kann explizit über die Methoden **Close()** oder **Dispose()** sowie implizit mit Hilfe der **using**-Anweisung geschehen.

Bei vielen Klassen, die wie **SqlConnection** das Interface **IDisposable** implementieren, sind die Methoden **Dispose()** und **Close()** äquivalent (z. B. bei der Klasse **FileStream**), und ein Objekt ist nicht mehr verwendbar, nachdem es einen Aufruf von **Dispose()** oder **Close()** erhalten hat. Bei der Klasse **SqlConnection** sind die Methoden **Dispose()** und **Close()** hingegen *nicht* äquivalent:

- Beide Methoden schließen die Verbindung zur Datenbank. Ein **Dispose()** - Aufruf impliziert einen **Close()** - Aufruf.
- Nach einem **Dispose()** - Aufruf ist ein **SqlConnection**-Objekt nicht mehr verwendbar.
- Eine per **Close()** - Aufruf geschlossene Datenbankverbindung kann per **Open()** - Aufruf erneut geöffnet werden.

Beim Einsatz einer **DataReader**-Klasse (z. B. **SqlDataReader**, vgl. Abschnitt 18.6.12) ist ein explizites Öffnen der Datenbankverbindung per **Open()** - Aufruf unumgänglich. Beim *verbindungslosen* Arbeiten mit der Klasse **SqlDataAdapter** (siehe Abschnitt 18.6.5) ist das explizite Öffnen und Schließen der Datenbankverbindung hingegen vermeidbar. So ist z. B. bei den folgenden Anweisungen zum Füllen eines **DataSet**-Objekts

```
SqlDataAdapter dbAdapter = new();
dbAdapter.SelectCommand = selCommand;
DataSet ds = new();
try {
   dbConnection.Open();
   dbAdapter.Fill(ds);
} finally {
   dbConnection.Close();
}
```

etlicher Aufwand überflüssig. Denselben Effekt erreicht man auch so:

```
SqlDataAdapter dbAdapter = new();
dbAdapter.SelectCommand = selCommand;
DataSet ds = new();
dbAdapter.Fill(ds);
```

Es ist davon auszugehen, dass die Methode **Fill()** eine von ihr geöffnete Datenbankverbindung auch dann schließt, wenn in **Fill()** ein Ausnahmefehler auftritt. Ist eine Verbindung schon *vor* dem **Fill()** - Aufruf geöffnet, dann wird sie von **Fill()** *nicht* geschlossen, weder bei einer regulären, noch bei einer fehlerhaften Ausführung.

Bei einer *Sequenz* von Methodenaufrufen mit Verbindungsautomatismus (z. B. **Fill()**, **FillSchema()**) kann es sinnvoll sein, die Datenbankverbindung vorher explizit zu öffnen und so ein automatisches zwischenzeitliches Schließen zu vermeiden (siehe Beispielprogramm im Abschnitt 18.6.2).

Der ADO.NET - Provider **SqlClient** verwendet per Voreinstellung das **Verbindungspooling**, damit die zeitaufwändige Erstellung einer Datenbankverbindung möglichst selten erforderlich ist. Beim **Close()** - Aufruf an ein **SqlConnection**-Objekt wird eine Verbindung nicht beendet, sondern an den Pool zurückgegeben, sodass sie beim nächsten **Open()** - Aufruf wiederverwendet werden kann. Für jede Verbindungszeichenfolge wird ein separater Verbindungspool verwaltet.

Ein **SqlConnection**-Objekt kann über seine Methode **CreateCommand()** ein Objekt der Klasse **SqlCommand** erstellen, das die **SqlConnection** für die Verbindung zum DBMS verwendet:

```
SqlCommand selCommand = dbConnection.CreateCommand();
selCommand.CommandText = "SELECT EmployeeID, FirstName, LastName FROM Employees";
```

Im Vergleich zur Verwendung des **SqlCommand**-Konstruktors wird aber kein Spareffekt erzielt:

```
SqlCommand selCommand = new("SELECT EmployeeID, FirstName, LastName FROM Employees",
                            dbConnection);
```

18.6.4 DbCommand

Für SQL-Anweisungen an ein DBMS (und auch für die im Kurs nicht behandelten gespeicherten Prozeduren) ist beim ADO.NET - Provider **SqlClient** die von **DbCommand** abstammende Klasse **SqlCommand** zuständig. Andere Provider bieten analog benannte Klassen.

18.6.4.1 Eigenschaften und Methoden

Die wichtigsten Eigenschaften eines **SqlCommand**-Objekts:

- **Connection**
  Über diese Eigenschaft ermittelt oder setzt man das **SqlConnection**-Objekt, das für die Verbindung zum DBMS zuständig ist. In der Regel wird das **SqlConnection**-Objekt schon per **SqlCommand**-Konstruktor festgelegt (siehe Beispiel). Es ist möglich, aber selten erforderlich, das **SqlConnection**-Objekt später noch einmal zu ändern.
- **CommandText**
  Über diese Eigenschaft ermittelt oder setzt man die SQL-Anweisung (oder die gespeicherte Prozedur), die ausgeführt werden soll.
- **CommandTimeout**
  Diese Eigenschaft legt fest, wie lange die Ausführung eines Kommandos maximal dauern darf.
- **Parameters**
  Diese Eigenschaft zeigt auf eine Kollektion mit dem Datentyp **SqlParameterCollection** und wird im Abschnitt 18.6.4.2 über parametrisierte Abfragen behandelt.

Beispiel:

```
string cs = "Data Source=(LocalDb)\\MSSQLLocalDB; " +
            "Initial Catalog=Northwind; Integrated Security=true";
using SqlConnection dbConnection = new(cs);
SqlCommand selCmd = new("SELECT EmployeeID, FirstName, LastName FROM Employees",
                        dbConnection);
```

Command-Objekte tun meist als Helfer eines **DbDataAdapter**-Objekts ihren Dienst (siehe Abschnitt 18.6.5), doch kommt bei einigen Methoden der **DbCommand**-Klassen auch der direkte Aufruf in Frage:

- **ExecuteReader()**
  Die **SqlCommand**-Methode
  public SqlDataReader ExecuteReader()
  sendet den **CommandText** an das eingebundene **SqlConnection**-Objekt und liefert als Rückgabewert einen **SqlDataReader**, der beim ausschließlich lesenden Zugriff auf eine große Datenbank als Alternative zur verbindungslosen Arbeitsweise (unter Verwendung einer lokalen Kopie) in Frage kommt (siehe Abschnitt 18.6.12).
- **ExecuteNonQuery()**
  Mit der **SqlCommand**-Methode
  public int ExecuteNonQuery()
  lässt man SQL-Kommandos aus der DDL (***D**ata **D**efinition **L**anguage*) wie z. B. **CREATE TABLE** oder aus der DML (***D**ata **M**anipulation **L**anguage*) (**UPDATE**, **INSERT**, **DELETE**) ausführen. Als Rückgabe wird die Anzahl der betroffenen Zeilen gemeldet.
  Beispiel:
  ```
  try {
     string inSt = "INSERT INTO Employees (LastName, FirstName, HireDate)" +
                   "VALUES('Meyer', 'Danny', '2010-01-01 12:00')";
     SqlCommand insCmd = new(inSt, dbConnection);
     dbConnection.Open();
     insCmd.ExecuteNonQuery();
  } finally {
     dbConnection.Close();
  }
  ```
- **ExecuteScalar()**
  Die **SqlCommand**-Methode
  public object ExecuteScalar()
  eignet sich zur Ausführung von **SELECT**-Kommandos, die einen *einzelnen Wert* zurückliefern, z. B. den mittleren Preis aller Produkte:
  ```
  SELECT AVG(UnitPrice) FROM Products
  ```
  In dieser Situation ist die **SqlCommand**-Methode **ExecuteScalar()** aufgrund des geringeren Aufwands gegenüber der **SqlDataAdapter**-Methode **Fill()** zu bevorzugen (vgl. Abschnitt 18.6.5.2). Von einem „breiteren“ oder „längeren“ Abfrageergebnis liefert **ExecuteScalar()** nur die erste Spalte der ersten Zeile. Weil der Rückgabewert vom Typ **Object** ist, muss er in der Regel einer expliziten Typanpassung unterworfen werden.
  Beispiel:
  ```
  try {
     SqlCommand selCmd = new("SELECT AVG(UnitPrice) FROM Products", dbConnection);
     dbConnection.Open();
     Console.WriteLine(Convert.ToDouble(selCmd.ExecuteScalar()));
  } finally {
     dbConnection.Close();
  }
  ```

Weil die Klasse **SqlCommand** (als Ableitung von **DbCommand**) die Schnittstelle **IDisposable** implementiert, sollte eigentlich vorsichtshalber für jedes **SqlCommand**-Objekt nach der Verwendung die Methode **Dispose()** aufgerufen werden, was im Rahmen einer **using**-Anweisung ohne großen Aufwand geschehen kann. In der ADO.NET - Dokumentation der Firma Microsoft wird diese Empfehlung allerdings nicht befolgt, und das Manuskript schließt sich gelegentlich der laxen Handhabung an, die zumindest beim Provider **SqlClient** keine negativen Konsequenzen hat. Die leidige Frage nach der Notwendigkeit von **Dispose()** - Aufrufen bei den ADO-NET - Klassen mit **IDisposable**-Implementation wird im Abschnitt 18.6.11 ausführlich behandelt.

18.6.4.2 Parametrisierte Abfragen

Wenn sich bei wiederholt benötigten Abfragen nur einzelne Bestandteile ändern, definiert man Parameter und ändert deren Werte, statt das komplette Kommando jeweils neu zu erzeugen. Auf diese Weise reduziert man den Aufwand des Datenbankverwaltungssystems, das sich auf jedes SQL-Kommando mit einem Ausführungsplan vorbereiten muss.[1] Im folgenden Programm

```
using System;
using System.Data;
using System.Data.SqlClient;

class ParamQuery {
  static void Main() {
    using SqlConnection dbConnection = new("Data Source=(LocalDb)\\MSSQLLocalDB; " +
                                           "Initial Catalog=Northwind; Integrated Security=true");
    SqlCommand selAvgSupp = new("SELECT AVG(UnitPrice) FROM Products WHERE SupplierID = @Supp",
                                dbConnection);
    selAvgSupp.Parameters.Add("@Supp", SqlDbType.Int);
    dbConnection.Open();
    while (true) {
      Console.Write("\nNummer das Anbieters (Beenden mit 0): ");
      int supp = Convert.ToInt32(Console.ReadLine());
      if (supp == 0)
        break;
      selAvgSupp.Parameters["@Supp"].Value = supp;
      Console.WriteLine("Mittlerer Preis aller Produkte von Anbieter " + supp +
                        ": " + Convert.ToDouble(selAvgSupp.ExecuteScalar()));
    }
  }
}
```

entscheidet der Anwender, für welche Lieferanten der Firma **Northwind** der mittlere Produktpreis ermittelt werden soll. Im **CommandText** steht an Stelle einer festen Lieferantennummer der Parameter @Supp:

```
selAvgSupp = new("SELECT AVG(UnitPrice) FROM Products WHERE SupplierID = @Supp",
                 dbConnection);
```

Das **SqlCommand**-Objekt wird über den Parameter informiert, wobei Name und Datentyp anzugeben sind:

```
selAvgSupp.Parameters.Add("@Supp", SqlDbType.Int);
```

Die **SqlCommand**-Eigenschaft **Parameters** zeigt auf eine Kollektion mit dem Datentyp **SqlParameterCollection**, welche Objekte der Klasse **SqlParameter** verwaltet und die üblichen Kollektionsmethoden beherrscht (z. B. **Add()**). Bei der gewählten **Add()** - Überladung sind ein Parametername und ein Datentyp (als Wert der Enumeration **SqlDbType**) anzugeben.

Vor einem Aufruf der Methode **ExecuteScalar()** wird per Indexer-Zugriff auf die **Parameters**-Kollektion der beteiligte Parameter auf den gewünschten Wert gesetzt:

```
selAvgSupp.Parameters["@Supp"].Value = supp;
```

---

[1] https://docs.microsoft.com/en-us/archive/msdn-magazine/2005/may/data-points-data-access-strategies-using-ado-net-and-sql

18.6.5 DbDataAdapter

Bei dem in der Regel zu bevorzugenden *verbindungslosen* Datenbankzugriff wird vom DBMS eine Kopie der relevanten Daten bezogen und in einem lokalen **DataSet**-Objekt gespeichert. Die lokale Kopie wird bearbeitet und ggf. anschließend im geänderten Zustand wieder zur Datenbank übertragen. Dabei spielt die Klasse **DbDataAdapter** bzw. ihre provider-spezifische Ableitung eine zentrale Rolle.

Zum Erstellen eines **DbDataAdapter**-Objekts besitzen die ADO-NET - Provider in der Regel mehrere Konstruktorüberladungen. Häufig wird dem Konstruktor per Parameter ein **SqlCommand** übergeben, das ein **SELECT**-Kommando kapselt, z. B. beim Provider **SqlClient**:

public SqlDataAdapter (SqlCommand *selectCommand***)**

18.6.5.1 Zuständigkeiten

Beim verbindungslosen Datenbankzugriff erledigt ein **DbDataAdapter**-Objekt die folgenden Aufgaben:

- Es führt mit Hilfe von **DbCommand**-Objekten SQL-Kommandos aus und vermittelt zwischen der Datenbank und den lokal (in einem **DataSet**-Objekt) zwischengespeicherten Daten. Dabei kommen die folgenden SQL-Kommandos zum Einsatz:
  - Das Kommando **SELECT** zur Anforderung von Datenbankinhalten
  - Die DML-Kommandos **UPDATE**, **INSERT** und **DELETE** zur Änderung der Datenbank
- Es öffnet zur Ausführung eines Kommandos (z. B. **Fill()**, **Update()**, siehe unten) bei Bedarf automatisch eine Verbindung zum DBMS und schließt diese Verbindung automatisch wieder zum frühestmöglichen Zeitpunkt. Eine geöffnet vorgefundene Verbindung wird jedoch nach Ausführung eines Kommandos *nicht* geschlossen.

Beim Provider **SqlClient** ist die Klasse **SqlDataAdapter** aus dem Namensraum **System.Data.SqlClient** zuständig; andere Provider bieten analog benannte Klassen.[1]

Für die Verbindung zu den beteiligten **DbCommand**-Objekten sorgen die folgenden **DbDataAdapter**-Eigenschaften

- **SelectCommand**
- **UpdateCommand**
- **InsertCommand**
- **DeleteCommand**

Während das **SELECT**-Kommando in der Regel vom Programmierer formuliert wird, kann die Erstellung der DML-Kommandos (**UPDATE**, **INSERT**, **DELETE**) in vielen Fällen einem Objekt der Klasse **CommandBuilder** überlassen werden (siehe Abschnitt 18.6.9).

---

[1] Zum **SqlClient**-Provider im Namensraum **Microsoft.Data.SqlClient** siehe Abschnitt 18.6.1.2.

18.6.5.2 Datentransfer von der Datenbank zum DataSet-Objekt

Beim Aufruf der **DbDataAdapter**-Methode **Fill()**, die das zu füllende **DataSet**-Objekt (siehe Abschnitt 18.6.6) per Parameter erfährt, wird das **SelectCommand** ausgeführt, das eine oder mehrere Tabellen liefert, die in der per **Tables**-Eigenschaft ansprechbaren **DataTableCollection** des **DataSet**-Objekts landen. Mehrere Tabellen kommen z. B. dann zu Stande, wenn der **CommandText** des **SelectCommand**-Objekts eine Serie von **SELECT**-Kommandos enthält, z. B.:

```
SELECT * FROM Employees; SELECT * FROM Customers
```

In der Regel vermeidet man aber diese Konstellation und übergibt dem **DbDataAdapter**-Objekt ein **SelectCommand**-Objekt, das nur *eine* Ergebnistabelle erstellt, z. B.:

```
SELECT EmployeeID, FirstName, LastName FROM Employees
```

Auf der Basis dieses SQL-Kommandos resultiert aus dem folgenden **Fill()** - Aufruf, der im ersten Parameter das zu füllende **DataSet**-Objekt erfährt, *eine* Tabelle (ein **DataTable**-Objekt) mit dem Namen `Employees`:

```
dbAdapter.Fill(ds, "Employees");
```

Bei Verzicht auf die Angabe eines Tabellennamens erhält die **TableName**-Eigenschaft zum ersten **DataTableCollection**-Element

```
ds.Tables[0].TableName
```

den Wert `Table`. Eine Ergebnistabelle kann in Abhängigkeit vom **SELECT**-Kommando auch durch das Zusammenführen von mehreren Datenbanktabellen entstehen, und das Adapter-Objekt versucht daher nicht, aus dem **SELECT**-Kommando einen Tabellennamen zu ermitteln.

Ein **DbDataAdapter** bietet über seine Eigenschaft **TableMappings** eine Möglichkeit zur individuellen Benennung von *mehreren* **Fill()** - Ergebnistabellen, z. B.:

```
dbAdapter.TableMappings.Add("Table", "Employees");
dbAdapter.TableMappings.Add("Table1", "Customers");
```

Die Korrespondenz zwischen den Standardnamen `Table`, `Table1`, … und den gewünschten **DataSet**-Tabellennamen wird über die per **TableMappings**-Eigenschaft ansprechbare **DataTableMappingCollection** vorgenommen. Nach diesen Vorbereitungen resultieren aus dem **Fill()** - Aufruf

```
dbAdapter.Fill(ds);
```

auf der Basis eines **SelectCommand**-Objekts mit dem **CommandText**

```
SELECT * FROM Employees; SELECT * FROM Customers
```

die Tabellennamen `Employees` und `Customers`. Allerdings sollte in der Regel ein **SelectCommand** verwendet werden, das ...

- *ein* **DataTable**-Objekt
- mit Informationen aus *einer* Datenbanktabelle

liefert, damit die zur Datenbankänderung erforderlichen SQL-Aktionskommandos (**UPDATE**, **INSERT**, **DELETE**) automatisch aus dem **SELECT**-Kommando erzeugt werden können (siehe Abschnitte 18.6.6.1 und 18.6.9).

Ist der Primärschlüssel einer Tabelle bekannt (z. B. aufgrund eines **FillSchema()** - Aufrufs, siehe Abschnitt 18.6.5.4), dann überschreibt die **Fill()** - Methode ggf. in der Ergebnistabelle bereits vorhandene Fälle. Ohne Wissen um den Primärschlüssel werden die vom **Fill()** - Aufruf (also vom beteiligten **SELECT**-Kommando) gelieferten Zeilen am Ende der Ergebnistabelle angehängt.

Als Rückgabewert liefert **Fill()** die Anzahl der hinzugefügten oder aktualisierten Zeilen. Bei einigen **Fill()** - Überladungen kann man sich auf eine Teilmenge aus dem **SELECT**-Abfrageergebnis beschränken und dazu die (nullbasierte) Nummer der ersten Datenzeile sowie die Anzahl der Datenzeilen angeben, z. B.:

```
dbAdapter.Fill(ds, 2, 5, "Employees");
```

18.6.5.3 Datentransfer vom DataSet-Objekt zur Datenbank

Beim Aufruf der **DbDataAdapter**-Methode **Update()** werden die DML-Befehle in **UpdateCommand**, **InsertCommand** und **DeleteCommand** ausgeführt, um die am lokalen **DataSet**-Objekt vorgenommenen Änderungen zur Datenbank zu übertragen, z. B.:

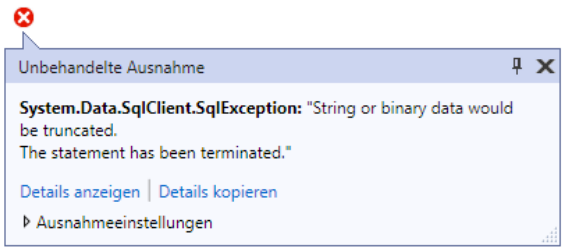
```
dbAdapter.Update(dt);
```

Im Abschnitt 18.6.9 ist zu erfahren, wie die DML-Befehle per **CommandBuilder** entstehen, und was bei **DataSet**-Objekten mit mehreren Tabellen zu beachten ist.

18.6.5.4 Schematransfer von der Datenbank zum DataSet-Objekt

Die im Schema einer Datenbank definierten Regeln für Tabellen und Spalten (z. B. Primärschlüssel, Fremdschlüssel, maximale Länge einer Zeichenfolgenvariablen, Null-Verbot) werden durch einen **Fill()** - Aufruf per Voreinstellung *nicht* zum lokalen **DataSet**-Objekt übertragen. Bei einem ausschließlich *lesenden* Zugriff auf das **DataSet**-Objekt ist das auch nicht unbedingt erforderlich. Wird ein **DataTable**-Objekt allerdings inkompatibel verändert und anschließend via **Update()** zur Datenbank übertragen, kommt es zu einem Laufzeitfehler, z. B.:

```
void UpdateData() {
    dbAdapter.Update(ds, "Employees");
}
```



Um aus dem Schema der Datenbank die Gültigkeitsregeln und Primärschlüsseldefinitionen zu den abgefragten Tabellen in ein lokales **DataSet** zu übertragen, kann man die **DbDataAdapter**-Methode **FillSchema()** nutzen. Durch die folgende Überladung wird das Schema einer einzelnen Tabelle übertragen:

public DataTable FillSchema(DataTable *table***, SchemaType** *type***)**

Im folgenden Codesegment wird ein **DataTable**-Objekt per **Fill()** - Aufruf mit Daten befüllt und benannt. Im anschließenden **FillSchema()** - Aufruf wird das **DataTable**-Objekt mit Schemainformationen versorgt:

```
string cs = "Data Source=(LocalDb)\\MSSQLLocalDB; " +
            "Initial Catalog=Northwind; Integrated Security=true";
string selCmd = "SELECT EmployeeID, FirstName, LastName FROM Employees";
SqlDataAdapter dbAdapter = new();
DataSet ds = new DataSet();
DataTable dt;
 . . .
void GetData() {
   using SqlConnection dbConnection = new(cs);
   dbAdapter.SelectCommand = new(selCmd, dbConnection);
   dbConnection.Open();
   dbAdapter.Fill(ds, "Employees");
   dt = ds.Tables["Employees"];
   dbAdapter.FillSchema(dt, SchemaType.Source);
}
```

Im Beispiel benötigt und kennt der **SqlDataAdapter** keine **TableMappings**-Einträge, sodass der zweite **FillSchema()** - Parameter (vom Datentyp **SchemaTyp**e) den Wert **SchemaType.Source** erhalten hat. Anderenfalls ist der Wert **SchemaType.Mapped** zu verwenden.

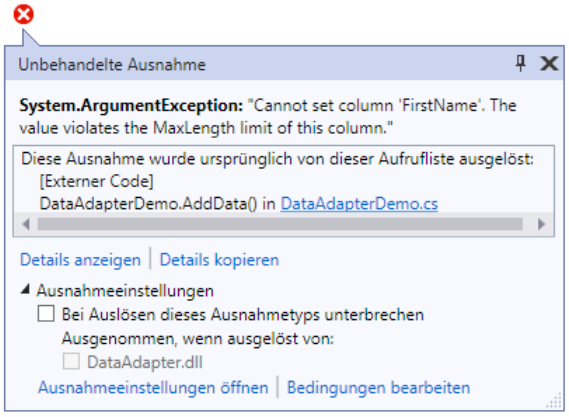
Zur Aufnahme der Schemainformationen werden in der Tabelle **DataColumn**-Objekte erzeugt und konfiguriert über die folgenden Eigenschaften:

- **AllowDBNull**
- **AutoIncrement**
  Dabei werden die Eigenschaften **AutoIncrementSeed** und **AutoIncrementStep** aber nicht gesetzt.
- **MaxLength**
- **ReadOnly**
- **Unique**

Weitere Schemainformationen landen ggf. in den **DataTable**-Eigenschaften **PrimaryKey** und **Constraints**.

Zwar führt ein Regelverstoß auch nach dem Schemaimport in das lokale **DataSet** zu einem Ausnahmefehler, doch wird das Problem nun ohne zeitaufwändige Datenbankverbindung erkannt:

```
void AddData() {
    DataRow dr = dt.NewRow();
    dr["FirstName"] = "VeryVeryVeryVeryVeryVeryVeryVeryVeryVeryLongFirst";
    dr["LastName"] = "Last";
    dt.Rows.Add(dr);
}
```



Statt die **DbDataAdapter**-Methode **FillSchema()** aufzurufen, kann man die **DbDataAdapter**-Eigenschaft **MissingSchemaAction** auf den Wert **AddWithKey** der Enumeration **MissingSchemaAction** setzen, z. B.:

```
dbAdapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;
```

Daraufhin findet bei jedem **Fill()** - Aufruf ein Abgleich zwischen den Schemainformationen in der Datenbank und im lokalen **DataSet** statt. Erhält die Eigenschaft **MissingSchemaAction** den Wert **AddWithKey**, dann profitieren alle **Fill()** - Aufrufe, und man spart sich den **FillSchema()** - Aufruf. Allerdings hat die Verwendung der **MissingSchemaAction** - Eigenschaft **AddWithKey** auch Nachteile:

- Bei wiederholten **Fill()** - Aufrufen für dieselbe Tabelle entsteht überflüssiger Aufwand.
- Zu den Spalten werden nur die Eigenschaften **AllowDBNull** und **MaxLength** mit Werten versorgt. Außerdem wird der Primärschlüssel der Tabelle gesetzt. Die Eigenschaften bzw. Einschränkungen **AutoIncrement**, **ReadOnly** und **Unique** bleiben hingegen unversorgt.

Daraus ergeben sich folgende Empfehlungen:

- Die **FillSchema()** - Methode ist sinnvoll, wenn ...
  - Tabellen *mehrfach* durch **Fill()** - Aufrufe gelesen werden,
  - oder *alle* Gültigkeitsregeln zu den Spalten benötigt werden.
- Die **MissingSchemaAction** - Eigenschaft **AddWithKey** ist zu bevorzugen, wenn ...
  - die **Fill()** - Methode pro Tabelle nur einmal aufgerufen wird, und die eingeschränkten Schemainformationen genügen.
  - das **SelectCommand** mehrere Tabellen liefert, weil die Methode **FillSchema()** in dieser Situation nur die erste Tabelle berücksichtigt (Hamilton & MacDonald 2003, S. 166).

Erfreulich ist bei den **DbDataAdapter**-Methoden **Fill()**, **FillSchema()** und **Update()** die Fähigkeit, bei Bedarf eine Datenbankverbindung automatisch zu öffnen und frühestmöglich wieder zu schließen (vgl. Abschnitt 18.6.3.2).

Man kann die Schemainformationen zu einem **DataSet**-Objekt außerdem ...

- komplett durch C# - Anweisungen definieren,
- oder mit Hilfe der **DataSet**-Methode **ReadXmlSchema()** aus einer XML-Datei einlesen, die z. B. zuvor mit Hilfe der **DataSet**-Methode **WriteXmlSchema()** erstellt worden ist.

In der Liste der **DataColumn**-Eigenschaften, die per **FillSchema()** - Aufruf an den **DbDataAdapter** einen Wert erhalten (**AllowDBNull**, **AutoIncrement**, **MaxLength**, **ReadOnly** und **Unique**) fehlen übrigens die wichtigen Eigenschaften **ColumnName** und **DataType**. Diese Eigenschaften werden schon durch den **Fill()** - Aufruf an den **DbDataAdapter** mit Werten versorgt, wobei eine Abbildung zwischen den beiden Typsystemen (.NET und DBMS) vorgenommen werden muss.[1] Z. B. resultiert aus der Spalte `BirthDate` in der `Employees`-Tabelle der Datenbank **Northwind** ein **DataColumn**-Objekt mit dem Datentyp **DateTime**.

Allerdings kann der Compiler trotzdem nicht für Typsicherheit sorgen und den Versuch unterbinden, das Geburtsdatum eines Falles auf einen ungeeigneten Wert zu setzen. Die folgende Zeile wird vom Compiler *nicht* kritisiert,

```
dtEmployees.Rows[1][1] = 13;
```

weil die zum Zugriff auf die Spaltenwerte in einem **DataRow**-Objekt zu verwendenden Indexer den Datentyp **Object** besitzen, z. B.:

public Object this[int columnIndex] { get; set; }

Zur Laufzeit verursacht die fehlerhafte Anweisung aber eine **ArgumentException**:

```
Unhandled exception. System.ArgumentException: Invalid cast from 'Int32' to
'DateTime'.Couldn't store <13> in BirthDate Column.  Expected type is DateTime.
```

18.6.6 DataSet und andere provider-unabhängige Klassen

In diesem Abschnitt werden wichtige Klassen aus dem provider-unabhängigen Namensraum **System.Data** behandelt. Diese Klassen speichern beim verbindungslosen Arbeiten lokale Kopien der angeforderten Datenbanktabellen samt Restriktionen (Gültigkeitsregeln) und Beziehungen.

18.6.6.1 DataSet

Im folgenden Quellcodesegment werden aus den **Northwind**-Tabellen `Customers` und `Orders` jeweils einige Spalten in ein **DataSet**-Objekt mit zwei Tabellen übernommen:

---

[1] https://docs.microsoft.com/de-de/dotnet/framework/data/adonet/sql-server-data-type-mappings

```
static DataSet ds = new();
static DataTable dtCustomers, dtOrders;
static string selCus = "SELECT CustomerID, CompanyName, Country FROM Customers",
              selOrd = "SELECT OrderID, CustomerID, ShipCountry FROM Orders",
              cs     = "Data Source=(LocalDb)\\MSSQLLocalDB; " +
                       "Initial Catalog=Northwind; Integrated Security=true";
 . . .
static void ReadData(string cs, string selCus, string selOrd) {
   using SqlConnection dbConnection = new(cs);
   using SqlCommand selCommandCus = new(selCus, dbConnection);
   using SqlCommand selCommandOrd = new(selOrd, dbConnection);
   using SqlDataAdapter dbAdapterCus = new(selCommandCus);
   using SqlDataAdapter dbAdapterOrd = new(selCommandOrd);

   dbAdapterCus.MissingSchemaAction = MissingSchemaAction.AddWithKey;
   dbAdapterOrd.MissingSchemaAction = MissingSchemaAction.AddWithKey;

   try {
      dbConnection.Open();
      dbAdapterCus.Fill(ds, "Customers");
      dbAdapterOrd.Fill(ds, "Orders");
   } finally {
      dbConnection.Close();
   }
   dtCustomers = ds.Tables["Customers"];
   dtOrders = ds.Tables["Orders"];
}
```

Die **Tables**-Eigenschaft eines **DataSet**-Objekts zeigt auf ein **DataTableCollection**-Objekt mit den enthaltenen Tabellen (**DataTable**-Objekten). Im Beispielprogramm werden **DataTable**-Referenzvariablen zu den beiden Tabellen angelegt, um später bequem darauf zugreifen zu können.

Durch den Aufruf der **Fill()** - Methode eines **DbDataAdapter**-Objekts wird das **SelectCommand** ausgeführt, das im folgenden Beispiel eine Tabelle (ein **DataTable**-Objekt) mit dem Namen `Customers` in dem per Parameter benannten **DataSet**-Objekt erstellt und befüllt:

```
dbAdapterCust.Fill(ds, "Customers");
```

Im Beispiel kommen *zwei* **SqlDataAdapter**-Objekte zum Einsatz, damit die zur Datenbankänderung erforderlichen SQL-Aktionskommandos (**UPDATE**, **INSERT**, **DELETE**) automatisch aus den **SELECT**-Kommandos für die beiden beteiligten Datenbanktabellen erzeugt werden können (siehe Abschnitt 18.6.9).

Ein **DataSet**-Objekt verwaltet über seine **Relations**-Eigenschaft eine Kollektion von **DataRelation**-Objekten, die jeweils eine Beziehung zwischen zwei Tabellen beschreiben (siehe Abschnitt 18.6.7).

Die Klasse **DataSet** ist nicht immer erforderlich für den verbindungslosen Datenbankzugriff mit ADO.NET. Man kann die unverzichtbaren **DataTable**-Objekte auch ohne **DataSet**-Container verwalten und dabei einige Quellcodezeilen einsparen, sofern keine Beziehungen zwischen Tabellen bestehen. Bei der Arbeit mit **DataRelation**-Objekten sind beim Verzicht auf den **DataSet**-Container aber Ausnahmefehler zu erwarten, z. B.:

```
Unhandled exception. System.Data.InvalidConstraintException: Cannot create a DataRelation if
Parent or Child Columns are not in a DataSet.
```

18.6.6.2 DataTable, DataRow und DataColumn

Die **Rows**- bzw. **Columns**-Eigenschaft eines **DataTable**-Objekts zeigt auf das **DataRowCollection**- bzw. auf das **DataColumnCollection**-Objekt mit den Zeilen bzw. Spalten der Tabelle:

- Die **DataRow**-Objekte in der **DataRowCollection** enthalten jeweils für einen Fall alle Merkmalsausprägungen.
- Die **DataColumn**-Objekte in der **DataColumnCollection** enthalten für jeweils eine Spalte die Schemainformationen (Gültigkeitsregeln, Constraints).

Per **Count**-Eigenschaft lässt sich die Anzahl von Zeilen bzw. Spalten feststellen.

Weil die beiden Kollektionen über einen Indexer verfügen, kann man mit einer bequemen Indexsyntax auf die Elemente zugreifen. Im folgenden Quellcodesegment werden die Zeilen (**DataRow**-Objekte) in der Tabelle `dtOrders` über den numerischen Index angesprochen:

```
Console.WriteLine("{0,15} {1,15} {2,20}",
   "OrderID", "CustomerID", "ShipCountry");
Console.WriteLine("{0,15} {1,15} {2,20}\n",
   dtOrders.Rows[0]["OrderID"].GetType(),
   dtOrders.Rows[0]["CustomerID"].GetType(),
   dtOrders.Rows[0]["ShipCountry"].GetType());
for (int i = 0; i < 5; i++)
   Console.WriteLine("{0,15} {1,15} {2,20}", dtOrders.Rows[i][0],
      dtOrders.Rows[i][1], dtOrders.Rows[i]["ShipCountry"]);
```

Bei den Spalten einer Tabelle gelingt der Zugriff alternativ über den nullbasierten numerischen Index oder über den Spaltennamen (Eigenschaft **ColumnName**). So kann man die Merkmalsausprägungen einer einzelnen Zeile oder auch die kompletten **DataColumn**-Objekte ansprechen (siehe unten).

Wie die Ausgabe

```
          OrderID      CustomerID          ShipCountry
    System.Int32   System.String        System.String

            10248           VINET              France
            10249           TOMSP             Germany
            10250           HANAR              Brazil
            10251           VICTE              France
            10252           SUPRD             Belgium
```

zeigt, werden die SQL-Feldtypen (hier: **int**, **nchar(5)** und **nvarchar(15)**) auf .NET - Datentypen (hier: **System.Int32** und **System.String**) der **DataColumn**-Objekte abgebildet. Auf der folgenden Webseite

https://docs.microsoft.com/de-de/dotnet/framework/data/adonet/sql-server-data-type-mappings

findet sich eine umfangreiche Tabelle mit den *SQL Server Data Type Mappings*.

Mit dem folgenden Methodenaufruf

```
Console.WriteLine(" MaxLength(CompanyName)     = " +
   dtCustomers.Columns["CompanyName"].MaxLength +
   "\n Primärschlüssel(Customers) = " + dtCustomers.PrimaryKey[0].ColumnName);
```

kann man sich davon überzeugen, dass durch die Anweisung

```
dbAdapterCust.MissingSchemaAction = MissingSchemaAction.AddWithKey;
```

wichtige Schemainformationen der Datenbanktabelle `Customers` in die **DataColumn**-Objekte des zugehörigen **DataTable**-Objekts `dtCustomers` übernommen worden sind. Man erhält die Ausgabe:

```
MaxLength(CompanyName)     = 40
Primärschlüssel(Customers) = CustomerID
```

Insbesondere ist die **DataTable**-Eigenschaft **PrimaryKey**, die auf einen **DataColumn**-Array zeigt, versorgt worden.

Die **Constraints**-Eigenschaft eines **DataTable**-Objekts zeigt auf ein **ConstraintCollection**-Objekt, das **UniqueConstraint**- oder **ForeignKeyConstraint**-Objekte enthalten kann. Im folgenden Codesegment werden die Restriktionen zum **DataTable**-Objekt `dtCustomers` aufgelistet:

```
foreach (Constraint c in dtCustomers.Constraints) {
   Console.WriteLine(" " + c.ConstraintName + "    " + c.GetType());
   if (c.GetType() == typeof(UniqueConstraint))
      Console.WriteLine("  Spalte zur UniqueConstraint: " +
         (c as UniqueConstraint).Columns[0].ColumnName);
}
```

Es liefert die Ausgabe:

```
Constaints zur Customers-Tabelle:
 Constraint1   System.Data.UniqueConstraint
  Spalte zur UniqueConstraint: CustomerID
```

Leider kann ADO.NET die Beziehungsstruktur einer Datenbank nicht automatisch übernehmen (z. B. per **FillSchema()**), sodass **ForeignKeyConstraint**-Objekte (z. B. zum **DataTable**-Objekt `dtOrders`) erst dann erscheinen, nachdem im Programm eine entsprechende **DataRelation** (zur Tabelle `dtCustomers`) angelegt worden ist (siehe Abschnitt 18.6.7).

18.6.7 Beziehungen zwischen Tabellen vereinbaren

Ein **DataSet**-Objekt verwaltet über seine **Relations**-Eigenschaft eine Kollektion von **DataRelation**-Objekten, die jeweils eine Beziehung zwischen zwei Tabellen beschreiben. Meist handelt es sich um Master-Details - Beziehungen (siehe Abschnitt 18.2.2). Von einer beteiligten Tabelle aus sind die **DataRelation**-Objekte über die **ChildRelations**- bzw. über die **ParentRelations**-Eigenschaft ansprechbar:

- Die **DataTable**-Eigenschaft **ChildRelations** zeigt auf ein Objekt der Klasse **DataRelationCollection** mit allen **DataRelation**-Objekten, an denen das **DataTable**-Objekt in der Master-Rolle beteiligt ist.
- Die **DataTable**-Eigenschaft **ParentRelations** zeigt auf ein Objekt der Klasse **DataRelationCollection** mit allen **DataRelation**-Objekten, an denen das **DataTable**-Objekt in der Details-Rolle beteiligt ist.

Leider kann ADO.NET die Beziehungsstruktur einer Datenbank nicht automatisch übernehmen (z. B. per **FillSchema()**). Wird etwa im Beispielprogramm von Abschnitt 18.6.6.1 das **DataSet**-Objekt `ds` (mit den Tabellen `dtCustomers` und `dtOrders`) nach der Anzahl seiner **DataRelation**-Objekte befragt,

```
Console.WriteLine("ds.Relations.Count: " + ds.Relations.Count);
```

resultiert das Ergebnis

```
ds.Relations.Count: 0
```

Mit den folgenden Anweisungen entsteht im **DataSet**-Objekt `ds` eine Master-Details - Beziehung mit dem Namen `CustomersOrders` zwischen der Parent-Spalte `CustomerID` in der `Customers`-Tabelle und der gleichnamigen Child-Spalte in der `Orders`-Tabelle:

```
DataColumn parent = dtCustomers.Columns["CustomerID"];
DataColumn child = dtOrders.Columns["CustomerID"];
coRel = new DataRelation("CustomersOrders", parent, child);
ds.Relations.Add(coRel);
```

Zu einem **DataRelation**-Objekt erstellt ADO.NET automatisch **Constraint**-Objekte für die beteiligten Tabellen. Mit den folgenden Anweisungen:

```
DataRelation coRel = ds.Relations["CustomersOrders"];
Console.WriteLine("\nParentKeyConstraint zur Beziehung \"CustomersOrders\":"+
   "\n Tabelle: " + coRel.ParentKeyConstraint.Table.TableName +
   "\n Feld:    " + coRel.ParentKeyConstraint.Columns[0].ColumnName +
   "\n Typ:     " + coRel.ParentKeyConstraint.GetType());
Console.WriteLine("\nChildKeyConstraint zur Beziehung \"CustomersOrders\":"+
   "\n Tabelle: " + coRel.ChildKeyConstraint.Table.TableName +
   "\n Feld:    " + coRel.ChildKeyConstraint.Columns[0].ColumnName +
   "\n Typ:     " + coRel.ChildKeyConstraint.GetType());
```

werden Informationen zu den Gültigkeitsregeln anfordert, die aus der Beziehung `Customers-Orders` resultieren:

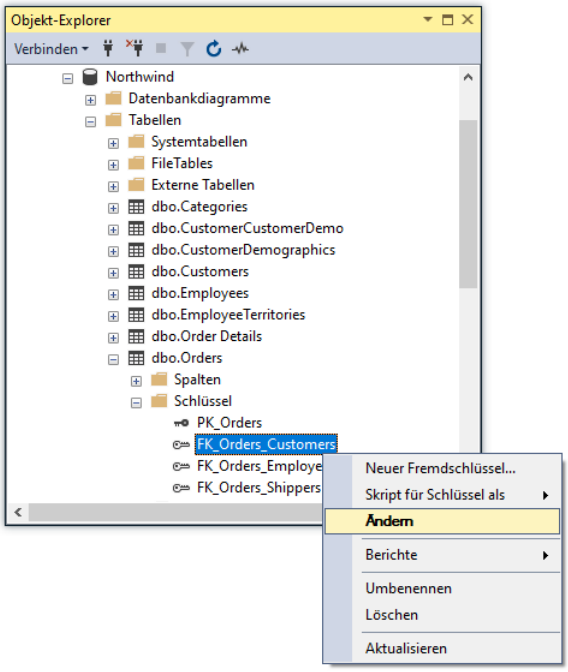
```
ParentKeyConstraint zur Beziehung "CustomersOrders":
 Tabelle: Customers
 Feld:    CustomerID
 Typ:     System.Data.UniqueConstraint

ChildKeyConstraint zur Beziehung "CustomersOrders":
 Tabelle: Orders
 Feld:    CustomerID
 Typ:     System.Data.ForeignKeyConstraint
```

Die aus einer Beziehung resultierenden Einschränkungen für die beteiligten Tabellen sind natürlich auch über deren **Constraints**-Eigenschaft ansprechbar, die auf eine Kollektion von **Constraint**-Objekten zeigt. So gehört im Beispiel das eben protokollierte **ForeignKeyConstraint**-Objekt auch zur **Constraints**-Kollektion der **Orders**-Tabelle.

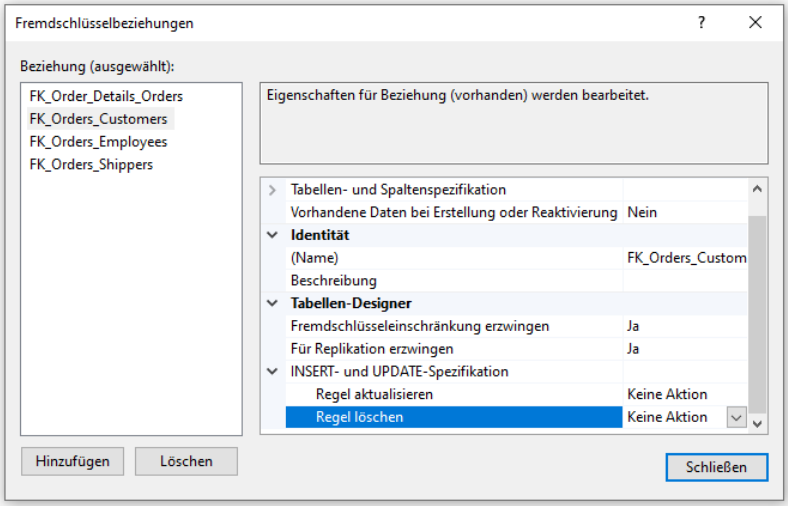
Für Objekt der Klasse **ForeignKeyConstraint** ist durch eine Eigenschaft namens **DeleteRule** mit Werten aus der Enumeration **Rule** im Namensraum **System.Data** geregelt, was beim Löschen einer Master-Zeile mit den zugehörigen Details-Zeilen geschehen soll. Die **Rule**-Ausprägungen haben die folgenden Bedeutungen:

- **Cascade** (= Voreinstellung)
  Die Details-Zeilen werden ebenfalls gelöscht.
- **None**
  Beim Versuch, eine Master- bzw. Parent-Zeile zu löschen, wird eine Ausnahme geworfen. Dies entspricht der typischen Einstellung eines SQL-Servers.
- **SetDefault**
  Wird eine Master- bzw. Parent-Zeile gelöscht, dann ...
  - erhalten die zugehörigen Details- bzw. Child-Zeilen für die Fremdschlüsselspalte den Voreinstellungswert, wenn ein solcher definiert ist.
  - wird eine Ausnahme geworfen, wenn kein Voreinstellungswert definiert ist.
- **SetNull**
  Wird eine Master- bzw. Parent-Zeile gelöscht, dann ...
  - erhalten zugehörige Details- bzw. Child-Zeilen für die Fremdschlüsselspalte den Wert **null**, wenn die Eigenschaft **AllowDBNull** den Wert **true** besitzt.
  - wird eine Ausnahme geworfen, wenn die Eigenschaft **AllowDBNull** den Wert **false** besitzt.

Die **DeleteRule**-Voreinstellung **Cascade** widerspricht der typischen, auch bei der **Northwind**-Datenbank realisierten, Einstellung eines SQL-Servers für die Verwaltung von Master-Details - Beziehungen. Eine Inspektion mit dem SQL Server Management Studio



ergibt für die Master-Details - Beziehung der Tabelle `Customers` zur Tabelle `Orders`, dass eine Master-Zeile *nicht* gelöscht werden darf, wenn zugehörige Details-Zeilen vorhanden sind:



Folglich kommt es beim versuchten Datenbank-Update zu einem Ausnahmefehler vom Typ **SqlException**:

```
void UpdateData() {
    dbAdapterCust.Update(dtCustomers);
}
```

Daher sollte die Eigenschaft **DeleteRule** den Wert **None** erhalten, um das Löschen von Master-Zeilen zu verhindern, sofern abhängige Details-Zeilen vorhanden sind:

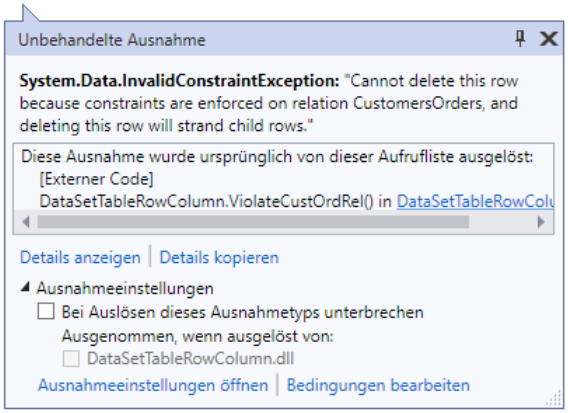
```
(coRel.ChildKeyConstraint as ForeignKeyConstraint).DeleteRule = Rule.None;
```

Eine analoge Aussage gilt für die Eigenschaft **UpdateRule**, sodass auch hier die Voreinstellung **Cascade** durch den alternativen Wert **None** ersetzt werden sollte:

```
(coRel.ChildKeyConstraint as ForeignKeyConstraint).UpdateRule = Rule.None;
```

Sind die Fremdschlüsselregeln der Datenbank im Programm bekannt, dann können Verstöße lokal per **InvalidConstraintException** erkannt und behandelt werden, z. B.

```
void ViolateCustOrdRel() {
    DataRow dr = dtCustomers.Rows.Find("ALFKI");
    dr.Delete();
}
```



Man muss also nicht den SQL-Server belästigen und Wartezeiten in Kauf nehmen.

Über die **DataRows**-Methode **GetChildRows()** ermittelt man die zu einer Tabellenzeile aufgrund einer Master-Details - Beziehung gehörigen Details-Zeilen, z. B.:

```
DataRow[] crs;
 . . .
crs = dtCustomers.Rows[i].GetChildRows(ds.Relations["CustomersOrders"]);
```

18.6.8 Änderungen an den DataTable-Objekten

Änderungen an den **DataTable**-Objekten im lokalen **DataSet**-Objekt werden später über die **DbDataAdapter**-Methode **Update()** (siehe Abschnitt 18.6.9) zur Datenbank übertragen.

18.6.8.1 Werte in vorhandenen DataRow-Objekten ändern

Um einzelne Werte einer **DataTable**-Zeile (eines **DataRow**-Objekts) zu ändern, kann man die zugehörigen Zellen per Indexer-Zugriff ansprechen, wobei ...

- die Zeilen über einen nullbasierten numerischen Index,
- die Spalten alternativ über ihren Namen oder einen nullbasierten numerischen Index

adressiert werden können, z. B.:

```
DataRow dr = dtCustomers.Rows[0];
dr["Country"] = "Lummerland";
dr[2] = "Lummerland";
```

Zur Identifikation einer Zeile kann bei einem **DataTable**-Objekt mit bekanntem Primärschlüssel die **Find()** - Methode der Klasse **DataRowCollection** verwendet werden, die einen Primärindexwert als Aktualparameterwert erwartet, z. B.:

```
DataRow dr = dtCustomers.Rows.Find("ALFKI");
```

Gelegentlich soll ein Wert gelöscht werden, was durch die Zuweisung des statischen Felds **Value** der Klasse **DBNull** (im Namensraum **System**) möglich ist, z. B.:

```
dr["Country"] = DBNull.Value;
```

Über die **DataRow**-Eigenschaft **ItemArray** (mit Typ **Object[]**) kann man alle Spalten eines **DataRow**-Objekts gemeinsam mit neuen Werten versorgen, z. B.:

```
dr.ItemArray = new Object[] {null, "Alfreds Hummerkiste", "Lummerland"};
```

Soll dabei eine Spalte ihren Wert behalten, verwendet man an der betroffenen Stelle das **Array**-Element **null**. Aus den Ausgangswerten

```
ALFKI   Alfreds Futterkiste   Hummerland
```

resultiert durch die obige Anweisung ein Ergebnis mit *unveränderter* `CustomerID`:

```
ALFKI   Alfreds Hummerkiste   Lummerland
```

Weil die **DataRow**-Eigenschaft **ItemArray** den Typ **Object[]** besitzt, kann der Compiler sinnlose Zuweisungen wie im folgenden Beispiel nicht verhindern:

```
dr.ItemArray = new Object[] { null, 13, DateTime.Now };
```

Auch die oben in Wertzuweisungen verwendeten Indexer der Klasse **DataRow** besitzen den Datentyp **Object**, z. B.:

public Object this[int columnIndex] { get; set; }

Daher wird die folgende Zuweisung vom Compiler *nicht* kritisiert:

```
dr["Country"] = DateTime.Now;
```

Zur Laufzeit verursacht die fehlerhafte Anweisung aber (noch vor der Datenbank-Aktualisierung) eine **ArgumentException**:

```
Unhandled exception. System.ArgumentException: Cannot set column 'Country'. The value violates
the MaxLength limit of this column.
```

Der Einsatz der sogenannten *untypisierten* DataSets ist offenbar fehleranfällig, und wir werden noch zwei Verbesserungen kennenlernen:

- Typisierte DataSets (siehe Abschnitt 18.7)
- Entity Framework Core (siehe Kapitel 20)

Hatte ein **DataRow**-Objekt den Ausgangszustand **Unchanged** (siehe Abschnitt 18.6.8.6), gelangt es durch eine Änderung von Werten in den Zustand **Modified** und durch einen anschließenden Aufruf der **DataSet**-, **DataTable**- oder **DataRow**-Methode **AcceptChanges()** (siehe Abschnitt 18.6.8.6) wieder zurück in den Zustand **Unchanged**.

18.6.8.2 Ereignisse bei DataTable-Änderungen

Ein **DataTable**-Objekt feuert bei Veränderungen diverse Ereignisse, die z. B. zur Überprüfung von Werten oder zur Neuberechnung abgeleiteter Werte durch registrierte Behandlungsmethoden dienen können. In der folgenden Tabelle werden die zur Validierung oder Verarbeitung neuer Werte geeigneten **DataTable**-Ereignisse beschrieben:

Ereignis	Beschreibung
ColumnChanging	Das Ereignis tritt ein, nachdem für eine Zelle eine Wertänderung angefordert worden ist. Eine registrierte Methode erfährt durch das per Parameter übergebene **DataColumnChangeEventArgs**-Objekt: • die betroffene Zeile (Eigenschaft **Row**) • die betroffene Spalte (Eigenschaft **Column**) • den vorgeschlagenen Wert (Eigenschaft **ProposedValue**) und kann über die **DataRow**-Methode **SetColumnError()** einen später auszuwertenden Fehlerindikator setzen (siehe Abschnitt 18.6.8.3).
ColumnChanged	Das Ereignis tritt ein, sobald der Wert einer Zelle erfolgreich geändert worden ist.
RowChanging	Das Ereignis tritt ein, wenn bei einer Zeile ein Zellenwert oder die **RowState**-Eigenschaft geändert werden soll. Eine registrierte Methode erfährt durch das per Parameter übergebene Ereignisbeschreibungsobjekt vom Typ **DataRowChangeEventArgs**: • die betroffene Zeile (Eigenschaft **Row**) Somit kann über die **DataRow**-Methode **SetColumnError()** ein später auszuwertender Fehlerindikator gesetzt werden (siehe Abschnitt 18.6.8.3). • die beabsichtigte Aktion (z.B. Veränderung oder Einfügen einer Zeile)
RowChanged	Das Ereignis tritt ein, nachdem bei einer Zeile ein Zellenwert oder die **RowState**-Eigenschaft erfolgreich geändert worden ist.
RowDeleting	Das Ereignis tritt ein, wenn eine Zeile als **Deleted** festgelegt werden soll.
RowDeleted	Das Ereignis tritt ein, nachdem eine Zeile als **Deleted** festgelegt worden ist.

Die Ereignisse sind für die folgenden Verwendungen konzipiert:

- **ColumnChanging**, **RowChanging** und **RowDeleting** erlauben eine Validierung und Stornierung von geplanten Veränderungen.
- **ColumnChanged**, **RowChanged** und **RowDeleted** erlauben eine Reaktion auf durchgeführte Veränderungen.
- **ColumnChanging** und **ColumnChanged** beziehen sich auf einzelne Wertänderungen.
- **RowChanging** und **RowChanged** berichten über Änderungen an Zeilen. Obwohl das Ereignisbeschreibungsobjekt nicht direkt über die geplanten neuen Werte informiert, kann eine Behandlungsmethode diese Werte über die **Row**-Eigenschaft in Erfahrung bringen und darauf reagieren.

In den folgenden Ereignismethoden finden allerdings keine Validierungen bzw. Verarbeitungen von Werten statt, sondern nur diagnostische Ausgaben:

```
static void CustomersOnColumnChanging(Object sender, DataColumnChangeEventArgs e) {
   DataTable table = (DataTable)sender;
   Console.WriteLine("ColumnChanging,\tSpalte: " + e.Column.Caption +
      "\n  vorgeschl. Wert: " + e.ProposedValue + ", akt. Wert: " + e.Row[e.Column]);
}

static void CustomersOnColumnChanged(Object sender, DataColumnChangeEventArgs e) {
   DataTable table = (DataTable)sender;
   Console.WriteLine("ColumnChanged,\tSpalte: " + e.Column.Caption +
      "\n  vorgeschl. Wert: " + e.ProposedValue + ", akt. Wert: " + e.Row[e.Column]);
}
```

```
static void CustomersOnRowChanging(Object sender, DataRowChangeEventArgs e) {
   DataTable table = (DataTable)sender;
   Console.WriteLine("RowChanging (ID: " + e.Row.ItemArray[0] +
        "), Action: " + e.Action + ", RowState: " + e.Row.RowState +
        "\n  CompanyName: " + e.Row["CompanyName"]);
}

static void CustomersOnRowChanged(Object sender, DataRowChangeEventArgs e) {
   DataTable table = (DataTable)sender;
   Console.WriteLine("RowChanged (ID: " + e.Row.ItemArray[0] +
        "), Action: " + e.Action + ", RowState: " + e.Row.RowState +
        "\n  CompanyName: " + e.Row["CompanyName"]);
}
```

Die Methoden werden bei den zugehörigen Ereignissen des **DataTable**-Objekts `dtCustomers` registriert:

```
dtCustomers.ColumnChanging += CustomersOnColumnChanging;
dtCustomers.ColumnChanged += CustomersOnColumnChanged;
dtCustomers.RowChanging += CustomersOnRowChanging;
dtCustomers.RowChanged += CustomersOnRowChanged;
```

Aus den Wertänderungen

```
dr["CompanyName"] = "Alfreds Kutterkiste";
dr["Country"] = "Hummerland";
```

resultieren diese Kontrollausgaben:

```
ColumnChanging, Spalte: CompanyName
  vorgeschl. Wert: Alfreds Kutterkiste, akt. Wert: Alfreds Futterkiste

ColumnChanged,  Spalte: CompanyName
  vorgeschl. Wert: Alfreds Kutterkiste, akt. Wert: Alfreds Kutterkiste

ColumnChanging, Spalte: Country
  vorgeschl. Wert: Hummerland, akt. Wert: Kummerland

ColumnChanged,  Spalte: Country
  vorgeschl. Wert: Hummerland, akt. Wert: Hummerland

RowChanging (ID: ALFKI), Action: Change, RowState: Unchanged
  Wert: Alfreds Kutterkiste

RowChanged (ID: ALFKI), Action: Change, RowState: Modified
  Wert: Alfreds Kutterkiste
```

Es zeigt sich u. a.:

- Für jede Wertveränderung erhält man vier Ereignisse in der folgenden Reihenfolge: **ColumnChanging**, **ColumnChanged**, **RowChanging** und **RowChanged**.
- Bei der **ColumnChanging**-Behandlung ist der vorgeschlagene Wert noch vom aktuellen verschieden, während beide bei der **ColumnChanged**-Behandlung übereinstimmen.
- Bei der ersten **RowChanging**-Behandlung hat die betroffene Zeile noch den **RowState**-Wert **Unchanged**, während die Spalte `CompanyName` schon den neuen Wert besitzt. In **RowChanging** ist also eine Validierung und Stornierung von geplanten Veränderungen möglich.
- Im Zusammenhang mit dem **RowChanged**-Ereignis wechselt der Zeilenstatus auf **Modified**. Mit den möglichen **RowState**-Werten einer Zeile werden wir uns im Abschnitt 18.6.8.6 beschäftigen.

Eine validierende Methode könnte einen vorgeschlagenen Wert bei Missfallen auf den aktuellen Wert setzen, z. B.:

```
if (e.Column.ToString().Equals("Country"))
   if ((e.ProposedValue as String).Equals("Kummerland"))
      e.ProposedValue = e.Row[e.Column];
```

Allerdings empfiehlt Microsoft zum Verwerfen von Änderungsvorschlägen eine andere Vorgehensweise, die sich z. B. durch den anschließend beschriebenen Bearbeitungsmodus für Datenzeilen realisieren lässt.

18.6.8.3 Der Bearbeitungsmodus für DataRow-Objekte

Die auf eine Einleitung durch die **DataRow**-Methode **BeginEdit()** folgenden Änderungen an den Werten einer Zeile kann man entweder durch einen **EndEdit()** - Aufruf quittieren oder durch einen **CancelEdit()** - Aufruf verwerfen, z. B.:

```
dr.BeginEdit();
dr["CompanyName"] = "Alfreds Hummerkiste";
dr["Country"] = "Kummerland";
if (!dr.HasErrors)
   dr.EndEdit();
else
   dr.CancelEdit();
```

Ein guter Grund für das Verwerfen vorgeschlagener Änderungen per **CancelEdit()** - Aufruf liegt z. B. dann vor, wenn die **DataRow**-Eigenschaft **HasErrors** der betroffenen Zeile den Wert **true** besitzt, z. B. aufgrund eines **SetColumnError()** - Aufrufs in einer Validierungsmethode:

```
static void CustomersOnColumnChanging(Object sender, DataColumnChangeEventArgs e) {
   DataTable table = (DataTable)sender;
   Console.WriteLine("\nColumnChanging,\tSpalte: " + e.Column.Caption +
      "\n  vorgeschl. Wert: " + e.ProposedValue + ", akt. Wert: " + e.Row[e.Column]);
   if (e.Column.ColumnName.Equals("Country"))
      if ((e.ProposedValue as String).Equals("Kummerland"))
         e.Row.SetColumnError(e.Column, "Error");
}
```

Man kann sich darauf verlassen, dass die **DataRow**-Eigenschaft **HasErrors** erst nach der Beendigung von allen registrierten Validierungsmethoden ausgewertet wird.

Der Bearbeitungsmodus deaktiviert **RowChanging**- und **RowChanged**-Ereignisse sowie die Kontrolle auf verletzte Eindeutigkeitsrestriktionen bis zum **EndEdit()** - Aufruf. Geänderte Spaltenwerte in einer Zeile führen dann unabhängig von ihrer Anzahl zu genau *einem* **RowChanging**- bzw. **RowChanged**-Ereignis.

Eine im Bearbeitungsmodus geänderte Zeile bleibt im Zustand **Unchanged** und wechselt erst beim **EndEdit()** - Aufruf in den Zustand **Modified**. Durch die **DataSet**-, **DataTable**- oder **DataRow**-Methode **AcceptChanges()** werden **DataRow**-Objekte vom Zustand **Modified** in den Zustand **Unchanged** befördert (siehe Abschnitt 18.6.8.6 zu den möglichen Zuständen einer Zeile).

18.6.8.4 DataRow-Objekt hinzufügen

Um eine neue Tabellenzeile anzulegen, …

- erstellt man mit der **DataTable**-Methode **NewRow()** eine Zeile mit dem Schema der Tabelle,
- versorgt die Pflichtfelder (Null-Verbot) mit Werten,
- und nimmt die Zeile schließlich per **Add()** - Methode in die Zeilen-Kollektion der Tabelle auf.

Hier wird die Tabelle `dtCustomers` um eine Zeile erweitert:

```
DataRow drem = dtCustomers.NewRow();
drem["CustomerID"] = "NEWCU";
drem["CompanyName"] = "Rempremerding & CO KG";
drem["Country"] = "Newland";
dtCustomers.Rows.Add(drem);
```

Das neue **DataRow**-Objekt befindet sich zunächst im Zustand **Detached**, d. h.:

- das Objekt existiert,
- gehört aber zu keiner **DataRowCollection**.

Durch einen **Add()** - Aufruf gelangt das neue **DataRow**-Objekt in den Zustand **Added**.

Bei einem späteren Datenbank-Update (siehe Abschnitt 18.6.9) erhalten die unversorgten Felder der neuen Datenbankzeile den Wert **NULL**, sofern dieser erlaubt ist.

18.6.8.5 DataRow-Objekt zum Löschen vormerken

Um eine **DataTable**-Zeile im Zustand (d.h. mit dem **RowState**-Wert) **Unchanged** oder **Modified** zu löschen, ruft man die **DataRow**-Methode **Delete()** auf, z. B.:

```
DataRow drem = dtCustomers.Rows.Find("NEWCU");
if (drem != null)
   drem.Delete();
```

Das **DataRow**-Objekt wird nicht aus der **DataRowCollection** beseitigt, sondern durch den **RowState**-Wert **Deleted** (siehe unten) zum Entfernen gekennzeichnet, sodass beim späteren Datenbank-Update die zugehörige Zeile der Datenbanktabelle (falls vorhanden) gelöscht wird. Der Vollständigkeit halber soll noch erwähnt werden, dass ein **DataRow**-Objekt im Zustand **Added** per **Delete()** in den Zustand **Detached** versetzt, d. h. aus der **DataRowCollection** der Tabelle entfernt wird, wobei die Daten verloren gehen.

Durch die nach einem Datenbank-Update automatisch aufgerufene **DataTable**-Methode **AcceptChanges()** werden **DataRow**-Objekte im Zustand **Deleted** aus der **DataRowCollection** der Tabelle entfernt (in den Zustand **Detached** versetzt), wobei die Daten verloren gehen.

Mit der **DataRowCollection**-Methode **Remove()** kann man ein **DataRow**-Objekt aus der **DataRowCollection** eines **DataTable**-Objekts entfernen und seine Daten löschen, z. B.:

```
dtCustomers.Rows.Remove(drem);
```

Das **DataRow**-Objekt gelangt in den Zustand **Detached** und hat bei einem späteren Datenbank-Update *keinen* Effekt, sodass eine korrespondierende Tabellenzeile der Datenbank unbehelligt bleibt.

18.6.8.6 Zeilenstatus und Zeilenversion

Für jedes **DataRow**-Objekt verwaltet ADO.NET seinen aktuellen Zustand, abfragbar über die Eigenschaft **RowState** vom Enumerationstyp **DataRowState**:

DataRowState	Beschreibung
Unchanged	Die Zeile wurde nicht geändert, seit … • der Übernahme aus einer Datenbank über die **DbDataAdapter**-Methode **Fill()** • dem letzten Aufruf der **DataSet**-, **DataTable**- oder **DataRow**-Methode **AcceptChanges()** Eine Zeile im Zustand **Modified** oder **Added** wird per **AcceptChanges()** - Aufruf in den Zustand **Unchanged** befördert. Eine Zeile im Zustand **Deleted** gelangt per **RejectChanges()** - Aufruf in den Zustand **Unchanged**.
Modified	Die zuvor im Zustand **Unchanged** in der Tabelle enthaltene Zeile wurde geändert ohne anschließenden **AcceptChanges()** - Aufruf.
Added	Die Zeile wurde der Tabelle hinzugefügt ohne anschließenden **AcceptChanges()** - Aufruf.
Detached	Die Zeile wurde entweder neu angelegt und noch nicht per **Add()** in die Tabelle aufgenommen, oder sie wurde aus der **DataRowCollection** der Tabelle entfernt (per **Remove()** oder durch eine **Delete()** - **AcceptChanges()** - Sequenz).
Deleted	Die zuvor im Zustand **Unchanged** oder **Modified** in der Tabelle enthaltene Zeile wurde per **Delete()** gelöscht (siehe oben). Sie kann nun per **AcceptChanges()** aus der Tabelle entfernt (in den Zustand **Detached** gebracht) oder per **RejectChanges()** in den Zustand **Unchanged** versetzt werden.

Je nach Bearbeitungsstand und **RowState**-Wert existieren zudem von einer Zeile verschiedene Versionen, wobei die Existenz einer bestimmten Version über die **DataRow**-Methode **HasVersion()** mit einem Parameter vom Enumerationstyp **DataRowVersion** festgestellt werden kann:

DataRowVersion	Beschreibung
Current	Diese Version enthält die aktuellen Werte einer Tabellenzeile.
Original	Diese Version enthält die Werte aus dem letzten **Unchanged**-Zustand der Zeile.
Proposed	Diese Version enthält mindestens einen *vorgeschlagenen* Wert. Sie ist für Tabellenzeilen im Bearbeitungsmodus (zwischen **BeginEdit()** und **EndEdit()** bzw. **CancelEdit()**) sowie bei neuen Zeilen vor der Aufnahme in die **DataRowCollection** (Zustand **Detached**) verfügbar.
Default	Die Standardversion hängt vom **RowState**-Wert ab: **DataRowState** / **Default**-Version ist gleich **Unchanged** / **Original** **Modified** / **Current** **Added** / **Current** **Detached** / **Proposed** **Deleted** / **Current**

Zum Zugriff auf eine bestimmte Zeilenversion dient eine Indexer-Überladung mit Versions-Parameter, z. B.

```
Console.WriteLine("Nach Fill():\t\t" + dr0[1] + "\tRowState: " + dr0.RowState);
dr0[1] = "Alfreds Hummerkiste";
Console.WriteLine("Nach Änderung:\t\t" + dr0[1] + "\tRowState: " + dr0.RowState);
Console.WriteLine("Original-Vers.:\t" + dr0[1, DataRowVersion.Original]);
```

Nach der Änderung unterscheiden sich die Versionen **Current** und **Original**:

```
Nach Fill():            Alfreds Kutterkiste     RowState: Unchanged
Nach Änderung:          Alfreds Hummerkiste     RowState: Modified
Original-Vers.:         Alfreds Kutterkiste
```

Bei der Tabellenmodifikation spielen die für **DataSet**-, **DataTable**- und **DataRow**-Objekte definierten Methoden **AcceptChanges()** und **RejectChanges()** eine wichtige Rolle:

- **AcceptChanges()** wirkt sich folgendermaßen auf **DataRow**-Objekte aus:

RowState vorher	**RowState** nachher	Wirkung
Modified	**Unchanged**	Die **Original**-Version wird durch die **Current**-Version überschrieben.
Added	**Unchanged**	Die **Current**-Version wird in die **Original**-Version kopiert.
Deleted	**Detached**	Das **DataRow**-Objekt wird aus der **DataRowCollection** der Tabelle entfernt. Seine Daten werden gelöscht.

  Durch einen **AcceptChanges()** - Aufruf wird nötigenfalls der Bearbeitungsmodus für **DataRow**-Objekte über einen impliziten **EndEdit()** - Aufruf abgeschlossen (siehe Abschnitt 18.6.8.3). Ein Aufruf der **DbDataAdapter**-Methode **Update()** hat einen Aufruf der **DataTable**-Methode **AcceptChanges()** zur Folge.

- **RejectChanges()** wirkt sich folgendermaßen auf **DataRow**-Objekte aus:

RowState vorher	**RowState** nachher	Wirkung
Unchanged	**Unchanged**	
Modified	**Unchanged**	Die **Current**-Version wird durch die **Original**-Version überschrieben.
Added	**Detached**	Das **DataRow**-Objekt wird aus der **DataRowCollection** der Tabelle entfernt. Seine Daten werden gelöscht.
Deleted	**Unchanged**	Die **Original**-Version wird in die **Current**-Version kopiert.

18.6.9 Datenbank-Update

Für die Übertragung der an einem **DataTable**-Objekt vorgenommenen Änderungen zur Datenbank ist die **DbDataAdapter**-Methode **Update()** zuständig, z. B.:

```
dbAdapterCust.Update(dtCustomers);
```

Von der hier gewählten Überladung wird für jede geänderte, ergänzte oder gelöschte Zeile im **DataTable**-Objekt `dtCustomers` das passende SQL-Kommando (vgl. Abschnitt 18.5) an das DBMS geschickt:

- ein **UPDATE**-Kommando für geänderte Zeilen (**RowState** hat den Wert **Modified**)
- ein **INSERT**-Kommando für ergänzte Zeilen (**RowState** hat den Wert **Added**)
- ein **DELETE**-Kommando für zu löschende Zeilen (**RowState** hat den Wert **Deleted**)

Man kann die beteiligten SQL-Kommandos selbst erstellen und jeweils verpackt in ein **SqlCommand**-Objekt der **UpdateCommand**-, **InsertCommand**- bzw. **DeleteCommand**-Eigenschaft des **DbDataAdapter**-Objekts zuweisen. Bequemer ist es jedoch, zum **DbDataAdapter**-Objekt ein passendes **CommandBuilder**-Objekt zu erzeugen, das die SQL-DML - Kommandos (sofern noch nicht vorhanden) automatisch produziert, z. B.:

```
new SqlCommandBuilder(dbAdapterCust);
```

Das klappt aber nur unter bestimmten Voraussetzungen:

- Das **DataTable**-Objekt mit den zu sichernden Änderungen ist aus *einer* Datenbanktabelle entstanden. Aus zwei oder mehr Datenbanktabellen durch **JOIN**-Operationen entstandene Ergebnistabellen werden nicht unterstützt.
- Das **DataTable**-Objekt verfügt über einen Primärschlüssel oder über eine Spalte mit Eindeutigkeitsrestriktion.
- Wenn das **SelectCommand**-Objekt des **DbDataAdapter**-Objekts *mehrere* Tabellen liefert, dann wird nur die *erste* Tabelle vom **CommandBuilder**-Objekt unterstützt, d. h. nur für die erste Tabelle werden DML-Kommandos erstellt. Wie gleich zu sehen ist, kann diese Einschränkung aber leicht durch die Verwendung von mehreren **SelectCommand**-Objekten überwunden werden.

Außerdem muss der **CommandBuilder** das **SelectCommand** ausführen, um vom DBMS erforderliche Informationen zum Erstellen der DML-Kommandos zu beschaffen. Wenn Performanz eine große Rolle spielt, dann kann die Vermeidung dieses Datenbankzugriffs durch das manuelle Erstellen der DML-Kommandos eine sinnvolle Option sein.[1]

Sind in einem **DataSet**-Objekt *mehrere* **DataTable**-Objekte mit zu sichernden Änderungen vorhanden, kann man trotzdem die **CommandBuilder**-Bequemlichkeit nutzen, indem man für jede Datenbanktabelle ein separates Trio aus **DbDataAdapter**, **SelectCommand** und **CommandBuilder** erstellt, z. B.:

```
static DataTable dtCustomers, dtOrders;
static string selCus = "SELECT CustomerID, CompanyName, Country FROM Customers",
              selOrd = "SELECT OrderID, CustomerID, ShipCountry FROM Orders",
              cs =     "Data Source=(LocalDb)\\MSSQLLocalDB; " +
                       "Initial Catalog=Northwind; Integrated Security=true" ;
. . .
using SqlConnection dbConnection = new(cs);
using SqlCommand selCommandCus = new(selCus, dbConnection);
using SqlCommand selCommandOrd = new(selOrd, dbConnection);
using SqlDataAdapter dbAdapterCus = new(selCommandCus);
using SqlDataAdapter dbAdapterOrd = new(selCommandOrd);
. . .
new SqlCommandBuilder(dbAdapterCus);
new SqlCommandBuilder(dbAdapterOrd);
```

Dabei kann man weiterhin mit *einem* **DataSet**-Objekt arbeiten, z. B.:

```
DataSet ds = new DataSet();
 . . .
dbAdapterCust.Fill(ds, "Customers");
dbAdapterOrd.Fill(ds, "Orders");
```

Zum Aktualisieren der Datenbank sind bei der vorgeschlagenen Konstruktion *zwei* **Update()** - Aufrufe erforderlich:

[1] https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/generating-commands-with-commandbuilders

```
static void UpdateData() {
   . . .
   try {
      dbConnection.Open();
      dbAdapterCust.Update(dtCustomers);
      dbAdapterOrd.Update(dtOrders);
   } finally {
      dbConnection.Close();
   }
}
```

Analog zur **Fill()** - Methode stellt auch die **Update()** - Methode bei Bedarf kurzzeitig eine Datenbankverbindung her (vgl. Abschnitt 18.6.3.2). Bei einer *Sequenz* von Methodenaufrufen mit Verbindungsautomatismus kann es aber performanter sein, die Datenbankverbindung vorher explizit zu öffnen, um so ein automatisches zwischenzeitliches Schließen zu vermeiden.

Am Ende eines **Update()** - Aufrufs wird automatisch die **DataTable**-Methode **AcceptChanges()** aufgerufen, um **RowState**-Anpassungen zu veranlassen (siehe Abschnitt 18.6.8.6). Weil eine Tabelle unmittelbar nach einem **AcceptChanges()** - Aufruf nur noch Zeilen im Zustand **Unchanged** enthält, ist in dieser Situation ein **Update()** - Aufruf wirkungslos.

Stehen zwei Tabellen in der Master-Details - Beziehung zueinander, dann ist die Reihenfolge der Update-Aufrufe relevant, z. B.:

- Wurden Details-Zeilen zu einer neuen Master-Zeile ergänzt, dann muss zuerst die Master-Tabelle aktualisiert werden.
- Wurde eine Master-Zeile zusammen mit den zugehörigen Details-Zeilen entfernt, dann muss zuerst die Details-Tabelle aktualisiert werden.

Im Kapitel 18 ignorieren wir aus Zeitgründen das Problem simultaner schreibender Datenbankzugriffe durch mehrere Benutzer. Solange eine Einzelbenutzerdatenbank wie Microsofts SQL Server Express LocalDB als DBMS zum Einsatz kommt, braucht man sich um simultane Datenbankzugriffe natürlich nicht zu kümmern. Im Zusammenhang mit dem Entity Framework Core, das für anspruchsvollere Datenbankanwendungen oft empfehlenswert ist, werden wir Maßnahmen zur Parallelitätskontrolle behandeln (siehe Abschnitt 20.7.4).

18.6.10 Zusammenspiel der ADO.NET - Klassen beim verbindungslosen Arbeiten

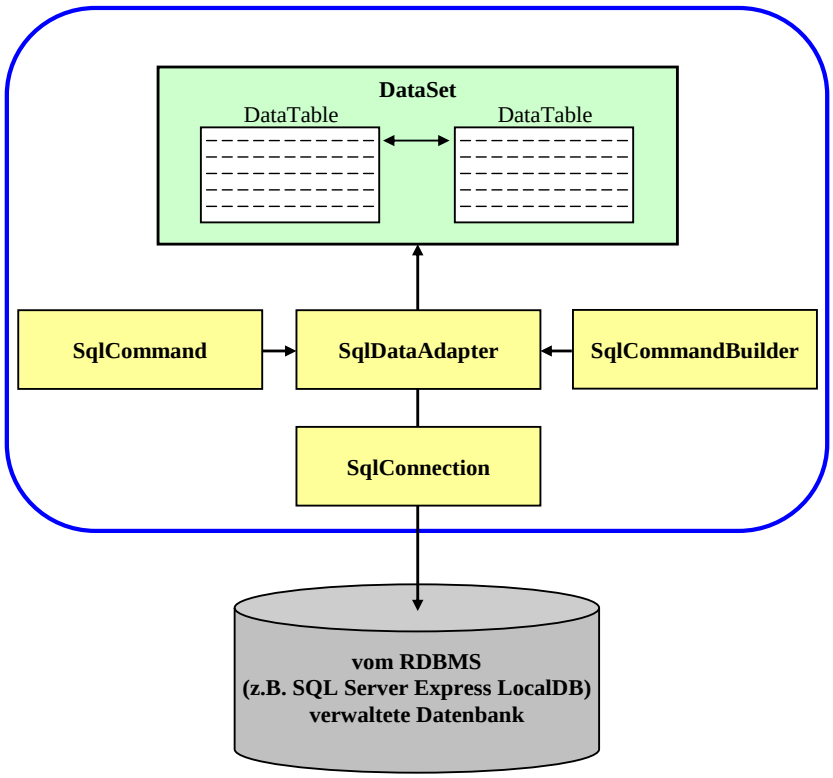
Im bisherigen Verlauf von Abschnitt 18.6 wurden zahlreiche ADO.NET - Klassen vorgestellt, wobei deren Verwendung meist durch Quellcodesegmente illustriert wurde. Jetzt soll noch einmal ein komplettes Beispielprogramm vorgeführt werden. Es liest im Sinne der verbindungslosen Datenbankbearbeitung aus der von einer LocalDB-Instanz der SQL Server Express - Edition verwalteten Datenbank **Northwind** die beiden Tabellen `Customers` und `Orders` in **DataTable**-Objekte ein, die zu einem gemeinsamen **DataSet**-Objekt gehören. Dabei kommen die Klassen **SqlConnection**, **SqlCommand** und **SqlDataAdapter** des **SqlClient** - Providers zum Einsatz. Nach einer Änderung der lokalen **DataTable**-Objekte wird die Klasse **SqlCommandBuilder** des **SqlClient** - Providers dazu benutzt, um die SQL-Kommandos (**UPDATE**, **INSERT** und **DELETE**) zur Übertragung der Änderungen an die Datenbank zu erstellen. Zur Ausführung dieser SQL-Kommandos wird schließlich die **SqlDataAdapter**-Methode **Update()** aufgerufen:

```
using System;
using System.Data;
using System.Data.SqlClient;
```

```
class Verbindungslos {
  static DataTable dtCustomers, dtOrders;
  static string selCus = "SELECT CustomerID, CompanyName, Country FROM Customers",
                selOrd = "SELECT OrderID, CustomerID, ShipCountry FROM Orders",
                cs =     "Data Source=(LocalDb)\\MSSQLLocalDB; " +
                         "Initial Catalog=Northwind; Integrated Security=true" ;

  static void ReadData(string cs, string selCus, string selOrd) {
    using SqlConnection dbConnection = new(cs);
    using SqlCommand selCommandCus = new(selCus, dbConnection);
    using SqlCommand selCommandOrd = new(selOrd, dbConnection);
    using SqlDataAdapter dbAdapterCus = new(selCommandCus);
    using SqlDataAdapter dbAdapterOrd = new(selCommandOrd);

    DataSet ds = new();

    // Das explizite Öffnen und Schließen der Datenbankverbindung vermeidet
    // die wiederholte Ausführung der Operationen durch Fill() und FillSchema()
    try {
      dbConnection.Open();
      dbAdapterCus.Fill(ds, "Customers");
      dbAdapterOrd.Fill(ds, "Orders");
      dbAdapterOrd.FillSchema(ds, SchemaType.Source);
    }
    finally {
      dbConnection.Close();
    }

    // Referenzvariablen für den einfachen Zugriff auf die Tabellen
    dtCustomers = ds.Tables["Customers"];
    dtOrders = ds.Tables["Orders"];
  }

  static void PrintData() {
    Console.WriteLine("Customers:\n{0,10} {1,40} {2,20}\n",
                      "CustomerID", "Company", "Country");
    for (int i = 0; i < 5; i++)
      Console.WriteLine("{0,10} {1,40} {2,20}", dtCustomers.Rows[i][0],
        dtCustomers.Rows[i][1], dtCustomers.Rows[i]["Country"]);
    Console.WriteLine("\n\nOrders:\n{0,10} {1,40}\n", "OrderID", "CustomerID");
    for (int i = 0; i < 5; i++)
      Console.WriteLine("{0,10} {1,40}", dtOrders.Rows[i][0],
        dtOrders.Rows[i][1]);
  }

  static void ChangeData() {
    dtCustomers.Rows[0]["Country"] = "Lummerland";
    dtOrders.Rows[0]["CustomerID"] = "TOMSP";
  }

  static void UpdateData() {
    using SqlConnection dbConnection = new(cs);
    using SqlCommand selCommandCus = new(selCus, dbConnection);
    using SqlCommand selCommandOrd = new(selOrd, dbConnection);
    using SqlDataAdapter dbAdapterCus = new(selCommandCus);
    using SqlDataAdapter dbAdapterOrd = new(selCommandOrd);

    // DML-Kommandos für beide DbDataAdapter automatisch erstellen lassen
    new SqlCommandBuilder(dbAdapterCus);
    new SqlCommandBuilder(dbAdapterOrd);
```

```
      try {
        dbConnection.Open();
        dbAdapterCus.Update(dtCustomers);
        dbAdapterOrd.Update(dtOrders);
      }
      finally {
        dbConnection.Close();
      }
    }

    static void Main() {
      ReadData(cs, selCus, selOrd);
      Console.WriteLine("Ausgangszustand:\n");
      PrintData();
      Console.WriteLine("\nWeiter mit Enter"); Console.ReadLine();
      ChangeData();
      Console.WriteLine("Nach der Änderung:\n");
      PrintData();
      UpdateData();
    }
  }
```

In der folgenden Abbildung wird die Kooperation der beteiligten Klassen skizziert:



18.6.11 Dispose

Die Frage, für welche ADO.NET-Objekte nach der Verwendung die **Dispose()** - Methode aufgerufen werden muss, wird im Internet intensiv diskutiert.[1] Weil die Klassen **DbConnection**, **DbDataAdapter** und **DbCommand** die Schnittstelle **IDisposable** implementieren, ist für jedes Objekt aus einer dieser Klassen nach der Verwendung ein **Dispose()** - Aufruf fällig. Das gilt auch für die abgeleiteten Klassen **SqlConnection**, **SqlDataAdapter** und **SqlCommand** des ADO-NET - Providers **SqlClient**. In den Beispielen aus der offiziellen ADO.NET - Dokumentation, die häufig

[1] https://stackoverflow.com/questions/18205560/do-i-need-to-explicitly-dispose-sqldataadapter

den Provider **SqlClient** verwenden, wird jedoch in der Regel nur für Objekte der Klasse **SqlConnection** (meist implizit per **using**-Anweisung) die Methode **Dispose()** aufgerufen, z. B.:[1]

```
private static DataSet SelectRows(DataSet dataset,
                                  string connectionString, string queryString) {
   using (SqlConnection connection = new SqlConnection(connectionString)) {
      SqlDataAdapter adapter = new SqlDataAdapter();
      adapter.SelectCommand = new SqlCommand(queryString, connection);
      adapter.Fill(dataset);
      return dataset;
   }
}
```

Im Manuskript wird gelegentlich der Einfachheit halber die Praxis aus Microsofts ADO.NET - Dokumentation übernommen. Ob die Unterlassung von **Dispose()** - Aufrufen an ADO.NET - Provider-Objekte tatsächlich ungestraft bleibt, hängt allerdings vom konkreten Provider ab. Microsofts ADO.NET - Dokumentation taugt also nur in Verbindung mit dem Provider **SqlClient** als Quasi-Legitimation für die nachlässige Praxis, ausschließlich für **DbConnection**-Objekte nach Gebrauch die **Dispose()** - Methode aufzurufen.

Wer die **IDisposable**-Regel strikt einhalten möchte, muss allerdings keinen allzu großen Aufwand in Kauf nehmen, weil für die **using**-Anweisung mit C# 8.0 eine sehr bequeme Syntaxvariante hinzugekommen ist (vgl. Abschnitt 16.2.3), z. B.:

```
private static DataSet SelectRows(DataSet dataset,
                                  string connectionString, string queryString) {
    using SqlConnection connection = new(connectionString);
    using SqlCommand command = new(queryString, connection);
    using SqlDataAdapter adapter = new(command);
    adapter.Fill(dataset);
    return dataset;
}
```

18.6.12 DbDataReader

Ist ausschließlich ein unidirektional-sequentieller Lesezugriff auf eine große Datenbank gefragt, dann kommt an Stelle der verbindungslosen Arbeitsweise mit einer lokalen Kopie der relevanten Datenbankinhalte der Einsatz einer Provider-spezifischen **DbDataReader**-Ableitung in Frage. Im Vergleich zur verbindungslosen Arbeitsweise (Befüllen eines **DataTable**-Objekts) ...

- werden Zeit und Speicherplatz gespart,
- muss die Verbindung zur Datenbank während des gesamten Lesevorgangs offengehalten werden.

Das folgende Beispielprogramm verwendet ein Objekt der zum Provider **SqlClient** gehörenden Klasse **SqlDataReader** dazu, um aus der `Customers`-Tabelle der **Northwind**-Datenbank alle Firmennamen aufzulisten:

```
using System;
using System.Data;
using System.Data.SqlClient;
```

[1] https://docs.microsoft.com/de-de/dotnet/api/system.data.sqlclient.sqldataadapter

```
class DbDataReaderDemo {
  static void Main() {
    using SqlConnection dbConnection = new("Data Source=(LocalDb)\\MSSQLLocalDB; " +
                                "Initial Catalog=Northwind; Integrated Security=true");
    SqlCommand command = new("SELECT CompanyName FROM Customers", dbConnection);

    // ExecuteReader benötigt eine offene Verbindung.
    dbConnection.Open();
    Console.WriteLine("\nNorthwind-Kunden:");
    using (SqlDataReader reader = command.ExecuteReader(CommandBehavior.CloseConnection))
      while (reader.Read()) {
        Console.WriteLine(reader["CompanyName"]);
      }

    // Beim Schließen des Readers wird die Verbindung automatisch ebenfalls geschlossen.
    Console.WriteLine("\nVerbindung: " + dbConnection.State);
  }
}
```

Zur Klasse **SqlDataReader** existiert kein öffentlicher Konstruktor. Im Beispiel erzeugt die **SqlCommand**-Methode **ExecuteReader()**

public SqlDataReader ExecuteReader(CommandBehavior *behavior***)**

den **SqlDataReader** und liefert eine Referenz zurück. Im Beispiel hat der Wert **CloseConnection** für den Parameter vom Typ **CommandBehavior** zur Folge, dass beim Schließen des **SqlDataReader**s auch die benutzte Datenbankverbindung geschlossen wird.

Ein **Read()** - Aufruf an den **SqlDataReader** beschafft die nächste vom **SELECT**-Kommando definierte Datenzeile und signalisiert ggf. mit dem Rückgabewert **false**, dass keine weitere Zeile verfügbar war.

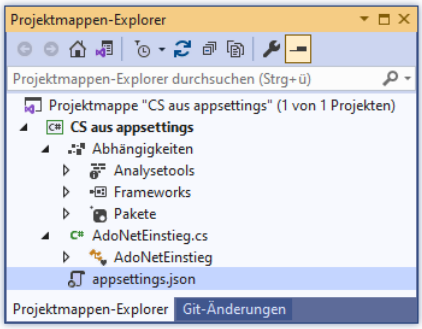
Solange ein **SqlDataReader** geöffnet ist, verwendet er das eingebundene **SqlConnection**-Objekt exklusiv. Es wird erst durch das Schließen des **DataReader**s für eine andere Verwendung frei.

18.6.13 Konfigurationsdatei mit der Verbindungszeichenfolge

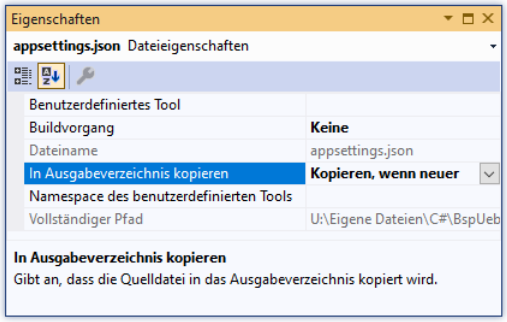
Eine praxisgerechte Datenbankanwendung sollte die Verbindungszeichenfolge nicht im Quellcode aufbewahren, sondern z. B. in einer Konfigurationsdatei, damit die Anwendung nach einer Änderung der Verbindungszeichenfolge nicht neu übersetzt werden muss. Wir erweitern das ADO.NET - Einstiegsbeispiel aus dem Abschnitt 18.6.2 um eine Anwendungskonfigurationsdatei im JSON-Format mit dem Namen **appsettings.json**. Diese Textdatei enthält die Verbindungszeichenfolge als Wert zum Attribut `AdoNetEinstiegdDb`:

```
{
 "ConnectionStrings": {
  "AdoNetEinstiegdDb":
   "Data Source=(LocalDb)\\MSSQLLocalDB; Initial Catalog=Northwind; Integrated Security=true"
 }
}
```

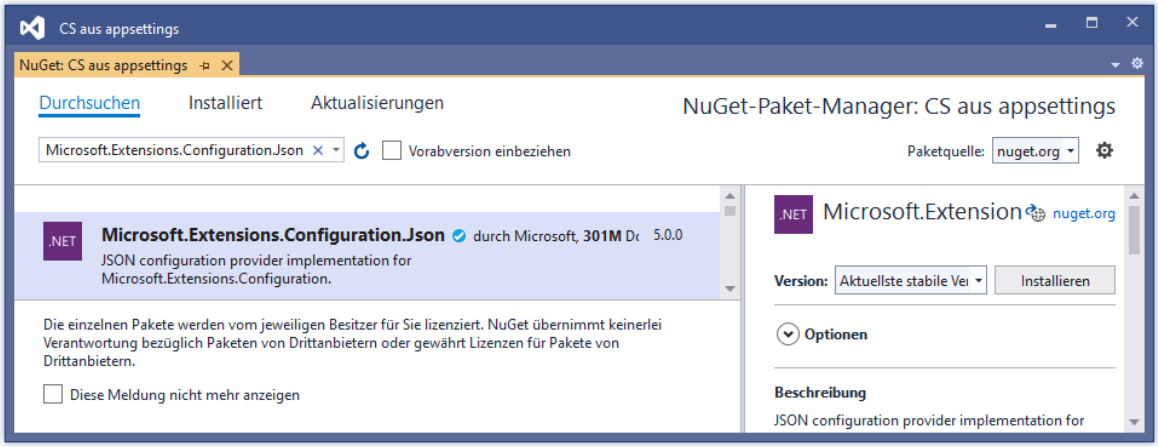
Das Visual Studio zeigt die im Anwendungsordner angelegte Datei spontan im Projektmappen-Explorer an:



Wir sorgen im **Eigenschaften**-Fenster dafür, dass die Konfigurationsdatei im Erstellungsprozess in den Ordner mit dem ausführbaren Programm kopiert wird:



Um die Einstellung mit Hilfe des **JSON-Konfigurationsanbieters**[1] laden zu können, installieren wir das NuGet-Paket **Microsoft.Extensions.Configuration.Json**:



Im Quellcode wird der Namensraum **`Microsoft.Extensions.Configuration`** importiert:

```
using Microsoft.Extensions.Configuration;
```

Nun kann die Verbindungszeichenfolge mit Hilfe eines **IConfigurationRoot** - Objekts gelesen werden:

```
IConfigurationRoot config = new ConfigurationBuilder()
   .AddJsonFile("appsettings.json")
   .Build();
string cs = config.GetConnectionString("AdoNetEinstiegdDb");
```

[1] https://docs.microsoft.com/en-us/dotnet/core/extensions/configuration-providers#file-configuration-provider

18.7 Typisierte DataSets

Bei der im Abschnitt 18.6 beschriebenen Datenbankprogrammierung mit der ADO.NET - Bibliothek besteht Verbesserungsbedarf:

- Im Quellcode werden oft Datenbankelemente über Zeichenfolgen angesprochen, z. B. die **Northwind**-Datenbanktabelle `Customers` und die dort vorhandene Spalte `Country`:

  ```
  DataTable dtCustomers = ds.Tables["Customers"];
  DataRow dr = dtCustomers.Rows[0];
  dr["Country"] = "Lummerland";
  ```

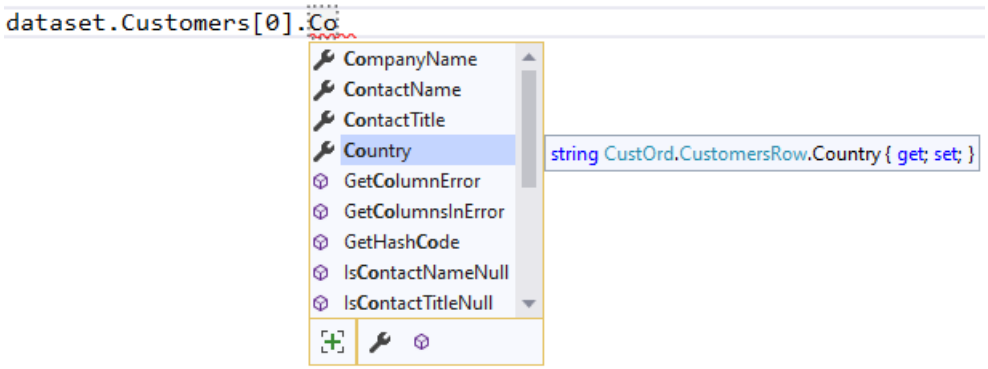
  Das Visual Studio kann mit seiner IntelliSense - Technik *keine* Schreibhilfe bieten, und der Compiler kann Tippfehler nicht aufdecken, sodass es leicht zu Laufzeitfehlern kommt.
- Es besteht ein Paradigmenbruch zwischen der objektorientierten Programmierung einerseits und der Bearbeitung von Tabellenzeilen andererseits, den die angelsächsische Literatur gelegentlich als *impedance mismatch* (dt.: *Unverträglichkeit*) bezeichnet.

Das erste Problem hat Microsoft mit dem sogenannten **typisierten DataSet** behoben, wobei durch einen Assistenten der Entwicklungsumgebung (den **DataSet-Designer**) zu einer Datenbankabfrage (z. B. basierend auf mehreren **SELECT**-Kommandos) spezifische C# - Klassen definiert werden, die in enger Beziehung zu den im Abschnitt 18.6 beschriebenen (untypisierten) ADO.NET - Klassen für das verbindungslose Arbeiten stehen.

Beim Zugriff auf die Spalte `Country` in der Tabelle `Customers` sind nun dem Compiler bekannte C# - Eigenschaften im Spiel:

```
CustOrd dataset = new();
dataset.Customers[0].Country = "Lummerland";
```

IntelliSense hilft beim bequemen und tippfehlerfreien Codieren:



Für das zweite Problem (*impedance mismatch*) hat Microsoft mit dem *Entity Framework Core* (siehe Kapitel 20) eine attraktive ORM-Lösung (*Object Relational Mapping*) entwickelt. Man kann also das typisierte DataSet als mittlerweile überholte Übergangslösung betrachten. Weil es lange Zeit die beste Option zur Datenbankentwicklung mit der .NET - Plattform war, ist es aber in vielen Projekten anzutreffen und wird daher im Manuskript behandelt.

18.7.1 Zu einer Datenbankabfrage automatisch definierte Klassen

Ein *typisiertes DataSet* beinhaltet Spezialisierungen der ADO.NET - Klassen **DataSet**, **DataTable** und **DataRow**, die durch zusätzliche Methoden und Eigenschaften den für eine Anwendung relevanten Teil des Schemas einer bestimmten Datenbank abbilden. Infolgedessen kann man Tabellen und Spalten typsicher durch C# - Eigenschaften ansprechen statt durch Indexer-Argumente vom Zeichenfolgentyp. Es resultiert ein stabiler, leicht zu lesender und leicht zu pflegender Quellcode.

Außerdem ist für jede beteiligte Datenbanktabelle noch eine typisierte Klasse im Spiel, welche die Aufgaben der ADO.NET - Klasse **DbDataAdapter** übernimmt.

Eine Klassenfamilie mit den beschriebenen Eigenschaften kann selbstverständlich jeder begabte und fleißige Programmierer auf der Basis der ADO.NET - Klassen erstellen. Beim Einsatz eines typisierten DataSets werden diese Klassen aber vom DataSet-Designer im Visual Studio von einer Datenbankabfrage ausgehend (z. B. basierend auf mehreren **SELECT**-Kommandos) *automatisch* erstellt.

Wir betrachten im Abschnitt 18.7 als Beispiel eine Kollektion von typisierten Klassen, die vom DataSet-Designer aus einer Abfrage an die Datenbank **Northwind** unter Beteiligung der Tabellen `Customers` und `Orders` erstellt worden sind.

Die Bezeichnung *typisiertes DataSet* für die gleich näher zu beschreibende Klassenfamilie ist etwas unglücklich gewählt und wird auch nicht konsistent verwendet. Zur Rechtfertigung der Bezeichnung kann man vorbringen, dass der DataSet-Designer eine typisierte **DataSet**-Ableitung als Top-Level - Klasse definiert, welche typisierte **DataTable**- und **DataRow**-Spezialisierungen als innere Klassen enthält. Allerdings sind zusätzlich typisierte Klassen mit **DbDataAdapter**-Aufgaben beteiligt, die nicht als DataSet-Bestandteile aufzufassen sind, und die vom DataSet-Designer dementsprechend als Top-Level - Klassen definiert werden.

Nach diesen terminologischen Randbemerkungen werden nun die vom DataSet-Designer zu einer Datenbankabfrage automatisch definierten typisierten Klassen beschrieben:

- Die typisierte **DataSet**-Klasse
  Es wird eine typisierte **DataSet**-Klasse passend zu den beteiligten Bestandteilen (z. B. Tabellen und Beziehungen) einer Datenbank definiert, wobei die Klasse **DataSet** als Basisklasse dient. Die folgende Klasse `CustOrd` ist das Ergebnis einer Abfrage unter Einbeziehung der Tabellen `Customers` und `Orders` in der Datenbank **Northwind**:[1]

  ```
  public partial class CustOrd : global::System.Data.DataSet {
      private CustomersDataTable tableCustomers;
      private OrdersDataTable tableOrders;
      private global::System.Data.DataRelation relationFK_Orders_Customers;
      . . .
  }
  ```

  Es werden datenbank-spezifische Methoden und Eigenschaften definiert. So enthält die typisierte **DataSet**-Klasse für jede zu bearbeitende Datenbanktabelle eine Eigenschaft, z. B.:

  ```
  public OrdersDataTable Orders {
      get {
          return this.tableOrders;
      }
  }
  ```

  Damit kann die Tabelle über eine dem Compiler bekannte Eigenschaft angesprochen werden. Tippfehler im Tabellennamen werden per IntelliSense minimiert bzw. vom Compiler reklamiert, statt einen Laufzeitfehler zu verursachen.

[1] Mit dem Schlüsselwort **global** und dem **:: - Operator** wird eine im globalen Namensraum beginnende Namensauflösung angeordnet (vgl. Abschnitt 2.6).

- Typisierte **DataTable**- und **DataRow**-Klassen
  Die eben vorgeführte Eigenschaft `Orders` ist vom Typ `OrdersDataTable`, der speziell zur **Orders**-Tabelle der **Northwind**-Datenbank vom Visual Studio als innere Klasse der typisierten **DataSet**-Ableitung `CustOrd` definiert worden ist:

```
public partial class OrdersDataTable :
         global::System.Data.TypedTableBase<OrdersRow> {
    private global::System.Data.DataColumn columnOrderID;
    private global::System.Data.DataColumn columnCustomerID;
    private global::System.Data.DataColumn columnEmployeeID;
    . . .
}
```

  Zur Konkretisierung des Typparameters der generischen Basisklasse **TypedTableBase<T>** dient eine als innere Klasse der typisierten **DataSet**-Ableitung `CustOrd` definierte **DataRow**-Ableitung, z. B.:

```
public partial class OrdersRow : System.Data.DataRow {
    . . .
  public int OrderID {
      get { . . . }
      set { . . . }
  }
    . . .
  public string CustomerID {
      get { . . . }
      set { . . . }
  }
    . . .
  public int EmployeeID {
      get { . . . }
      set { . . . }
  }
    . . .
}
```

  Über einen Indexer der inneren Klasse `OrdersDataTable`

```
public OrdersRow this[int index] {
    get {
        return ((OrdersRow)(this.Rows[index]));
    }
}
```

  lassen sich die Zeilen der Tabelle als Objekte vom Typ `CustOrd.OrdersRow` ansprechen, z. B.:

```
dataset.Orders[0].ShippedDate = DateTime.Now;
```

  Die Spalten sind dabei bequem und typsicher über C# Eigenschaften ansprechbar, sodass die Bezeichnung *typisiertes DataSet* gerechtfertigt ist. In einem untypisierten DataSet bietet sich die Ansprache der Spalten über einen Indexer mit **String**-Parameter an, um für einen gut lesbaren Quellcode zu sorgen, z.B.:

```
dtOrders.Rows[0]["ShippedDate"] = "Yesterdy";
```

  Dabei drohen allerdings Tippfehler, und vor allem kann der Compiler nicht für Typsicherheit bei der Wertzuweisung sorgen, weil alle Items in einer untypisierten **DataRow**-Klasse den Typ **Object** besitzen.

  Außerdem besitzt die Klasse `OrdersDataTable` für jede Spalte der zugehörigen Datenbanktabelle eine Eigenschaft mit dem Datentyp **DataColumn**, und das referenzierte Objekt enthält die Schemainformationen zur Tabellenspalte, z. B.:

```
this.columnOrderID.AutoIncrement = true;
this.columnOrderID.AllowDBNull = false;
this.columnOrderID.ReadOnly = true;
this.columnOrderID.Unique = true;
this.columnCustomerID.MaxLength = 5;
```

Im Vergleich zur Verwendung der untypisierten ADO.NET - Klassen muss sich der Anwendungsprogrammierer nicht darum kümmern, die Schemainformationen aus der Datenbank zu beschaffen (vgl. Abschnitt 18.6.5.4).

Analog zu `OrdersDataTable` und `OrdersRow` sind in `CustOrd` auch die inneren Klassen `CustomersDataTable` und `CustomersRow` vorhanden.

- Typisierte Klassen mit **DbDataAdapter**-Aufgaben
  Der DataSet-Designer erstellt neben der typisierten **DataSet**-Ableitung mit inneren, typisierten **DataTable**- und **DataRow**-Klassen auch noch typisierte Klassen, welche die Aufgaben der ADO.NET - Klasse **DbDataAdapter** übernehmen (z.B. durch Methoden **Fill()** und **Update()**, vgl. Abschnitt 18.6.5). Allerdings ist z. B. die Klasse `OrdersTableAdapter` nicht als Ableitung der provider-spezifischen Klasse **SqlDataAdapter** definiert, sondern sie enthält ein Member-Objekt aus dieser Klasse:

```
public partial class OrdersTableAdapter : System.ComponentModel.Component {
    private global::System.Data.SqlClient.SqlDataAdapter _adapter;
    . . .
    protected internal global::System.Data.SqlClient.SqlDataAdapter Adapter {
        get {
            if ((this._adapter == null)) {
                this.InitAdapter();
            }
            return this._adapter;
        }
    }
    . . .
}
```

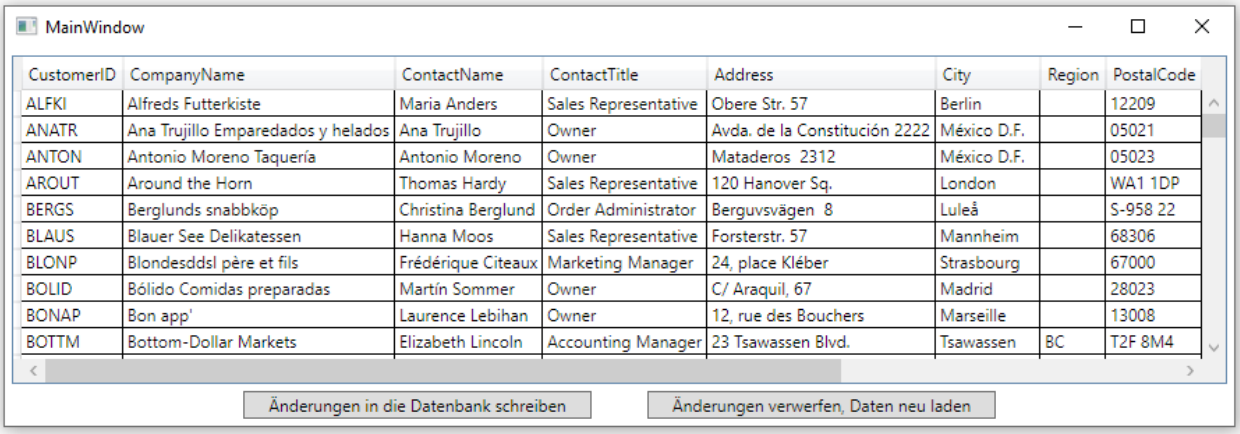
Ein `OrdersTableAdapter` kann trotzdem als typisierter, für eine bestimmte Tabelle zuständiger **DbDataAdapter** betrachtet werden, der somit für das Befüllen des typisierten **DataTable**-Objekts und für die Datenbank-Aktualisierung zuständig ist, z. B.:

```
public virtual int Fill(CustOrd.OrdersDataTable dataTable) {
    this.Adapter.SelectCommand = this.CommandCollection[0];
    if ((this.ClearBeforeFill == true)) {
        dataTable.Clear();
    }
    int returnValue = this.Adapter.Fill(dataTable);
    return returnValue;
}
```

Aufgrund der zur Entwurfszeit erforderlichen Definitionen ist ein typisiertes DataSet statisch. Werden *dynamisch* definierte Abfragen benötigt, dann müssen die *untypisierten* ADO.NET - Klassen (**DataSet**, **DataTable**, etc.) verwendet werden.

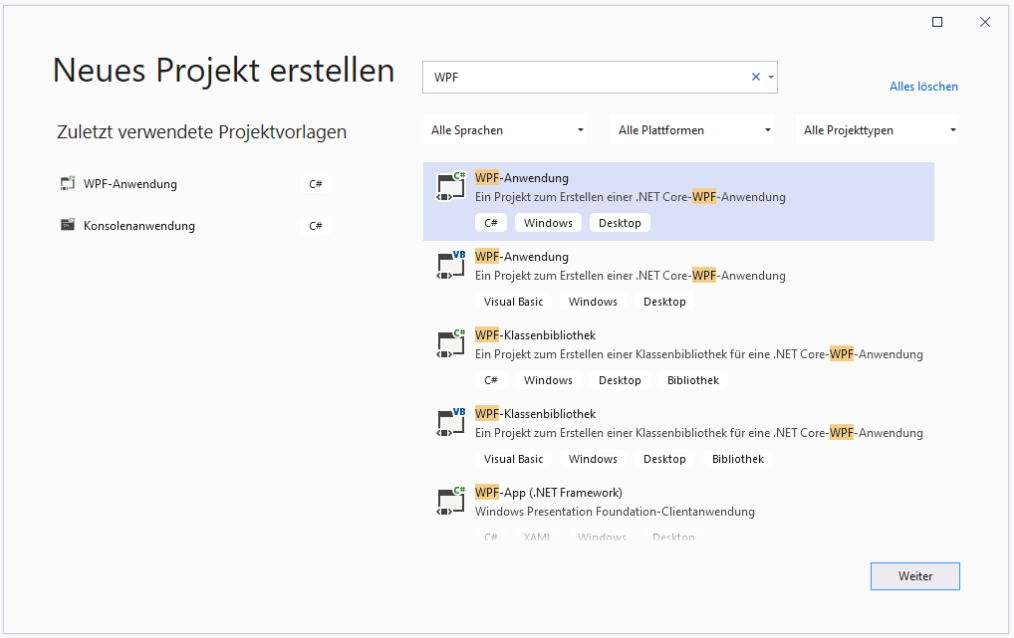
18.7.2 Anwendungsbeispiel

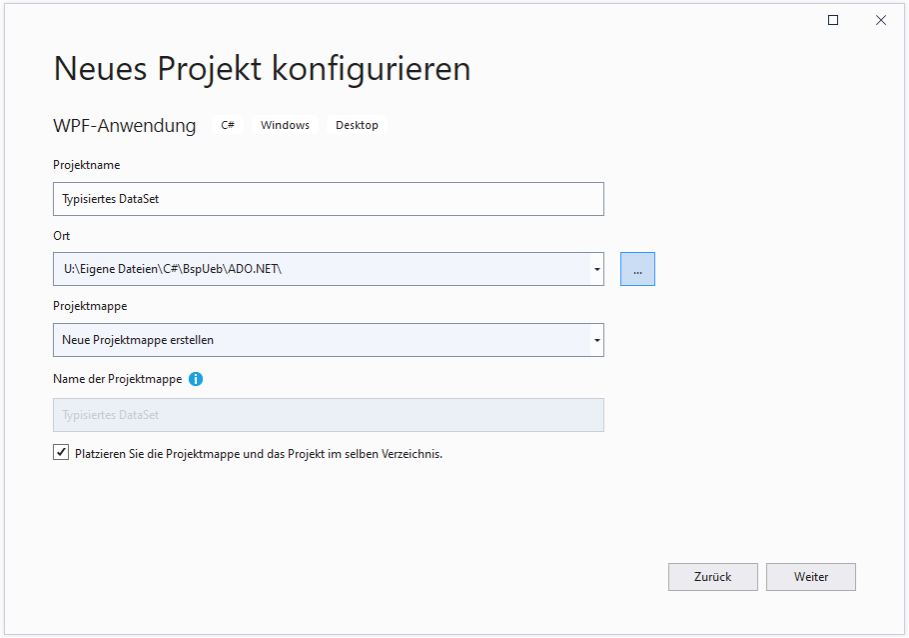
Wir erstellen eine WPF-Anwendung für .NET 5.0 mit Datenbankzugriff über ein typisiertes DataSet. Das Programm wird ein benutzerfreundliches Editieren der **Northwind**-Tabelle `Customers` erlauben,

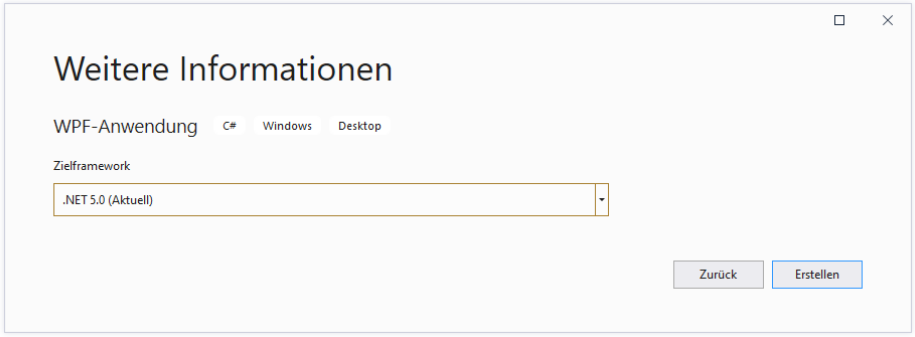


und viele Tausend Zeilen Quellcode enthalten, von denen wir die wenigsten selbst verfassen müssen.

Legen Sie dazu ein neues Projekt auf Basis der Vorlage **WPF-Anwendung** (für .NET Core) an, z. B.:

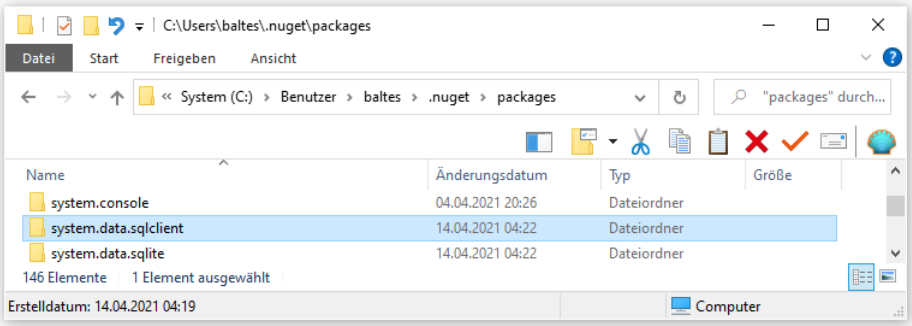






18.7.2.1 ADO.NET - Provider per NuGet installieren

Da wir die LocalDB-Variante von Microsofts SQL-Server Express verwenden wollen, installieren wir für das neue Projekt gemäß Abschnitt 18.6.2 per NuGet den ADO.NET-Provider **System.Data.SqlClient**. Vermutlich befindet sich das Paket bereits auf Ihrem Rechner, z. B.:



Daher ist die projektbezogene Installation schnell erledigt.

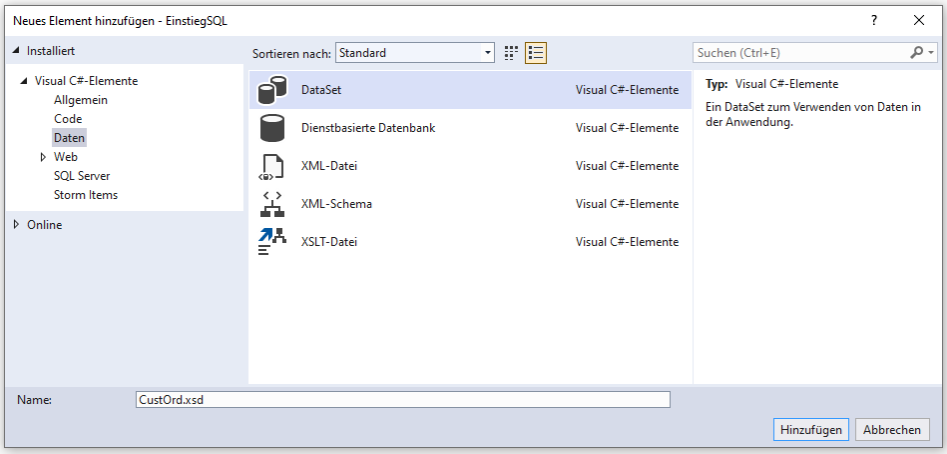
18.7.2.2 Typisiertes DataSet definieren

Wir erstellen ein typisiertes DataSet mit dem **DataSet-Designer**:[1]

- Im **Projektmappen-Explorer** wählen wir aus dem Kontextmenü zum Projekt:

  Hinzufügen > Neues Element > Daten > DataSet

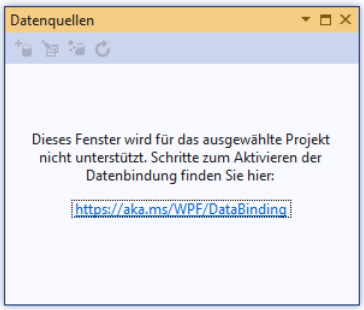
  Weil aus der **Northwind**-Datenbank die Tabellen `Customers` und `Orders` einbezogen werden sollen, verwenden wir den Namen **CustOrd.xsd**:[2]



- Mit dem **SQL Server-Objekt-Explorer**, der nötigenfalls per **Ansicht** - Menü aktiviert wird,

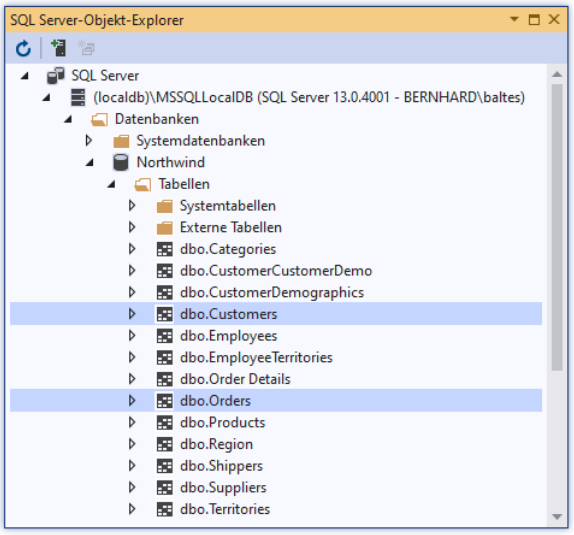
---

[1] Leser mit Erfahrungen bei der Datenbankprogrammierung mit dem .NET *Framework* vermissen eventuell den Begriff der *Datenquelle* und das **Datenquellen**-Fenster. Beim Versuch, in unserem bewusst für .NET 5.0 erstellten Projekt über den Menübefehl **Ansicht > Weitere Fenster > Datenquellen** das Datenquellenfenster zu öffnen, wird über die fehlende Unterstützung informiert:

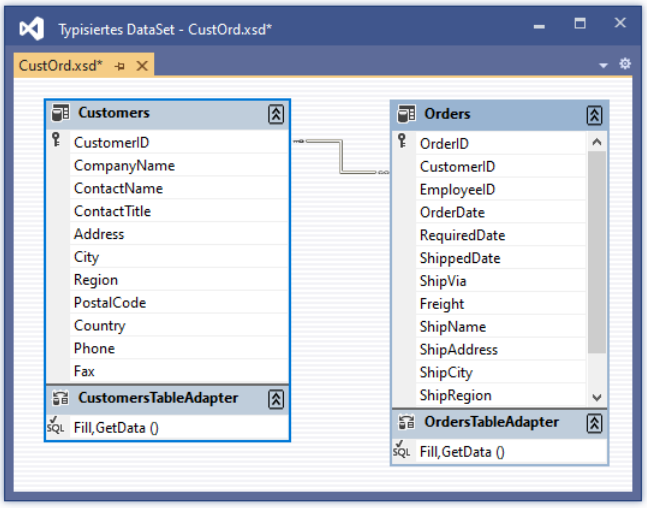


Statt das nicht mehr weiterentwickelte .NET Framework 4.8 (und damit C# 7.3) zu verwenden, verzichten wir auf das **Datenquellen**-Fenster.

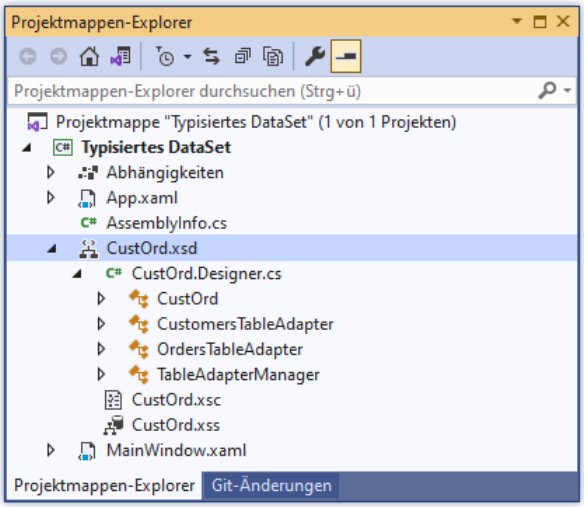
[2] Für die zu Beginn von Abschnitt 18.7.2 präsentierte Ausbaustufe des Programms ist die Tabelle `Orders` überflüssig. Sie wird trotzdem einbezogen, um einige Optionen des DataSet-Designers demonstrieren zu können. Bei der Erweiterung des Programms unter Verwendung der Tabelle `Orders` sind Ihrer Phantasie kaum Grenzen gesetzt.



befördern wir die Tabellen `Customers` und `Orders` auf die Entwurfszone des DataSet-Designers:



Im **Projektmappen-Explorer** erscheint das Arbeitsergebnis des DataSet-Designers als **xsd**-Datei mit untergeordneten Einträgen:

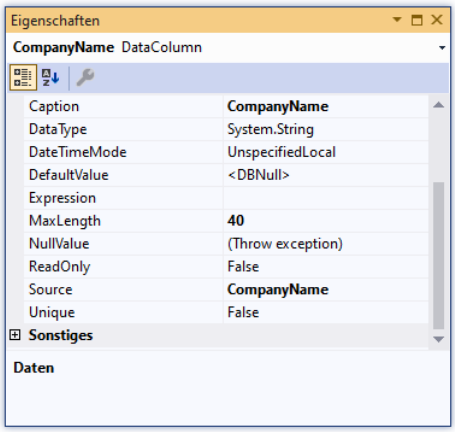


Die im Abschnitt 18.7.1 beschriebenen Typen werden in der Datei **CustOrd.Designer.cs** (mit einem Umfang von fast 4000 Zeilen) definiert.

Hier ist auch die Verbindungszeichenfolge zu finden, falls sie geändert oder (besser) aus einer Anwendungskonfigurationsdatei bezogen werden soll:

```
private void InitConnection() {
    this._connection = new global::System.Data.SqlClient.SqlConnection();
    this._connection.ConnectionString = "Data Source=(localdb)\\MSSQLLocalDB;" +
                        "Initial Catalog=Northwind;Integrated Security=True";
}
```

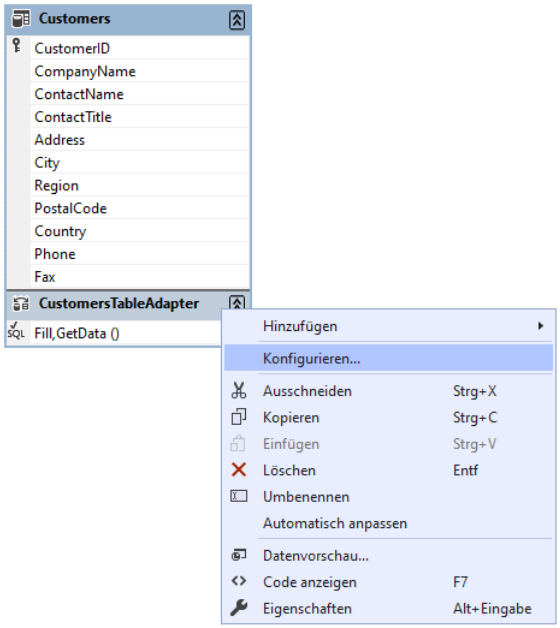
Man kann das Schema zum typisierten DataSet **CustOrd.xsd** über das **Eigenschaften**-Fenster bearbeiten, z. B. nach dem Markieren einer Spalte:



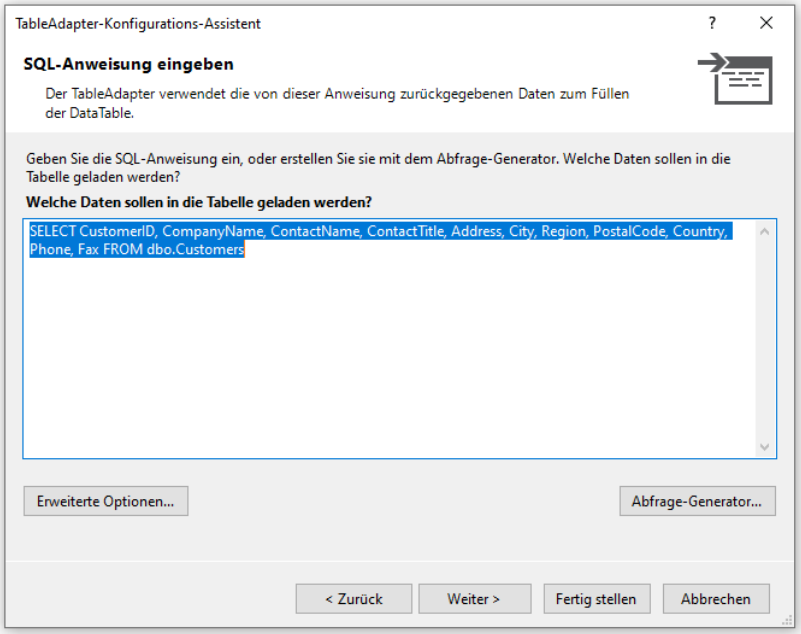
Es gibt aber kaum einen Grund, die automatisch aus der Datenbank übernommenen Schemainformationen zu ändern. Wie ein Blick auf das Datenbankdiagramm zeigt, ist auch die Master-Details - Beziehung aus der Datenbank übernommen worden.

18.7.2.3 Abfragen modifizieren

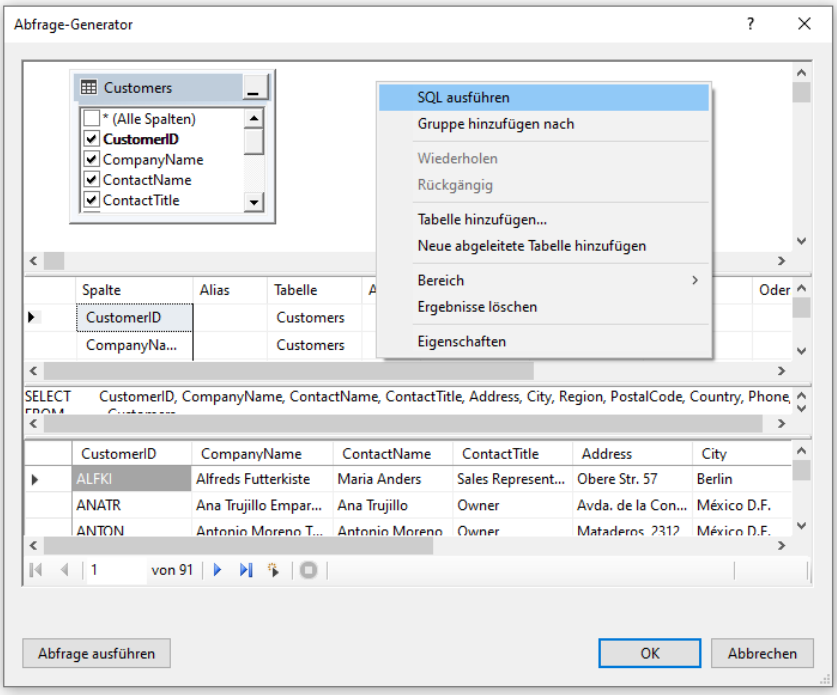
Natürlich sind auch beim Einsatz eines typisierten DataSets **SELECT**-Abfragen an die Datenbank im Spiel, wobei sich das Visual Studio bemüht, die Erstellung der **SELECT**-Kommandos durch Assistenten zu automatisieren bzw. zu vereinfachen. Um eine automatisch erstellte Abfrage zu modifizieren, startet man den **TableAdapter-Konfigurations-Assistenten** im DataSet-Designer über das Item **Konfigurieren** aus dem Kontextmenü zu einem **TableAdapter**, z. B.:



Hier ist das **SELECT**-Kommando zur automatisch erstellten Auswahlabfrage zu sehen und zu modifizieren:



Statt das **SELECT**-Kommando direkt zu verändern, kann man den **Abfrage-Generator** bemühen:



Die am unteren Rand angezeigte Vorschau auf das Abfrageergebnis erhält man über die Kontextmenü-Option **SQL ausführen**.

18.7.2.4 DataGrid

Wir befördern per WPF-Designer aus der **Toolbox**

- ein Steuerelement aus der Klasse **DataGrid**
- und zwei Befehlsschalter aus der Klasse **Button**

auf das Anwendungsfenster. Zur Anordnung und Platzverteilung verwenden wir den folgenden XAML-Code mit einem Wurzel-Layoutcontainer vom Typ **DockPanel** (siehe Abschnitt 12.6.2):

```
<DockPanel>
  <StackPanel DockPanel.Dock="Bottom" Orientation="Horizontal" HorizontalAlignment="Center">
    <Button Content="Änderungen in die Datenbank schreiben" Margin="0,0,20,5" Width="250"
     Click="BtnSaveClick"/>
    <Button Content="Änderungen verwerfen, Daten neu laden" Margin="20,0,0,5" Width="250"
     Click="BtnCancelClick"/>
  </StackPanel>
  <DataGrid DockPanel.Dock="Top" ItemsSource="{Binding}" Margin="5" />
</DockPanel>
```

Ein **DataGrid**-Steuerelement erlaubt es den Benutzern, am *Inhalt* der angebundenen Tabelle Änderungen vorzunehmen, die schlussendlich an die Datenbank übertragen werden können:

- Zellinhalte können editiert werden.
- In die leere Zeile am Tabellenende können neue Fälle eingetragen werden.
- Eine markierte Zeile kann per **Entf**-Taste gelöscht werden.

Die Befehlsschalter dienen dazu, die per **DataGrid** am Abfrageergebnis vorgenommenen Änderungen an die Datenbank zu übertragen oder zu verwerfen.

Außerdem lässt sich per **DataGrid**-Steuerelement die *Ansicht* der Tabelle ändern, was ohne Einfluss auf die Datenbank bleibt:

- Die Zeilen können nach einer frei wählbaren Spalte auf- oder absteigend sortiert werden.
- Die Spalten können per Drag&Drop verschoben werden.

18.7.2.5 Datentabelle und DataGrid verbinden

Damit die simple Datenbindungskonfiguration im **DataGrid**-Element des XAML-Codes

```
<DataGrid DockPanel.Dock="Top" ItemsSource="{Binding}" Margin="5" />
```

tatsächlich zum Erfolg führt, sind nur wenige Anweisungen in der Definition der Anwendungsfensterklasse `MainWindow` erforderlich:

```
using Typisiertes_DataSet.CustOrdTableAdapters;
 . . .
namespace Typisiertes_DataSet {
   public partial class MainWindow : Window {
       CustOrd dataset;
       CustomersTableAdapter adapter;

       public MainWindow() {
           InitializeComponent();
           dataset = new CustOrd();
           adapter = new CustomersTableAdapter();
           try {
               adapter.Fill(dataset.Customers);
           } catch (Exception e) {
               MessageBox.Show(e.Message);
               throw;
           }
           DataContext = dataset.Customers;
       }
   }
}
```

Von den zum typisierten DataSet gehörenden, automatisch erstellten Klassen verwenden wir:

- `CustOrd`
- `CustomersTableAdapter`

Der in **CustOrd.Designer.cs** für die **DbDataAdapter**-Klassen definierte Namensraum wird per **using**-Direktive importiert:

```
using Typisiertes_DataSet.CustOrdTableAdapters;
```

Im Konstruktor der Anwendungsfensterklasse wird ...

- aus den Klassen `CustOrd` und `CustomersTableAdapter` jeweils ein Objekt erzeugt,
- das `CustomersTableAdapter`-Objekt per **Fill()** - Aufruf gebeten, die Daten der `Customers`-Tabelle aus der Datenbank in die `Customers`-Tabelle des `CustOrd`-Objekts (genauer: in das von der `Customers`-Eigenschaft referenzierte `CustomersDataTable`-Objekt) zu befördern,
- der `MainWindow`-Eigenschaft **DataContext** das von der `Customers`-Eigenschaft des `CustOrd`-Objekts referenzierte `CustomersDataTable`-Objekt als Wert zugewiesen.

18.7.2.6 Ereignisbehandlung für die Befehlsschalter

Schließlich müssen die im XAML-Code angekündigten **Click**-Behandlungsmethoden zu den beiden Befehlsschaltern noch als Instanzmethoden der Klasse `MainWindow` implementiert werden.

Die Änderungen per **DataGrid**-Steuerelement betreffen das typisierte DataSet im Hauptspeicher und haben zunächst keinen Effekt auf die Datenbank. In der **Click**-Behandlungsmethode zum linken Schalter wird die **Update()** - Methode des `CustomersTableAdapter`-Objekts dazu benutzt, um die Änderungen in die Datenbank zu schreiben (vgl. Abschnitt 18.6.9):

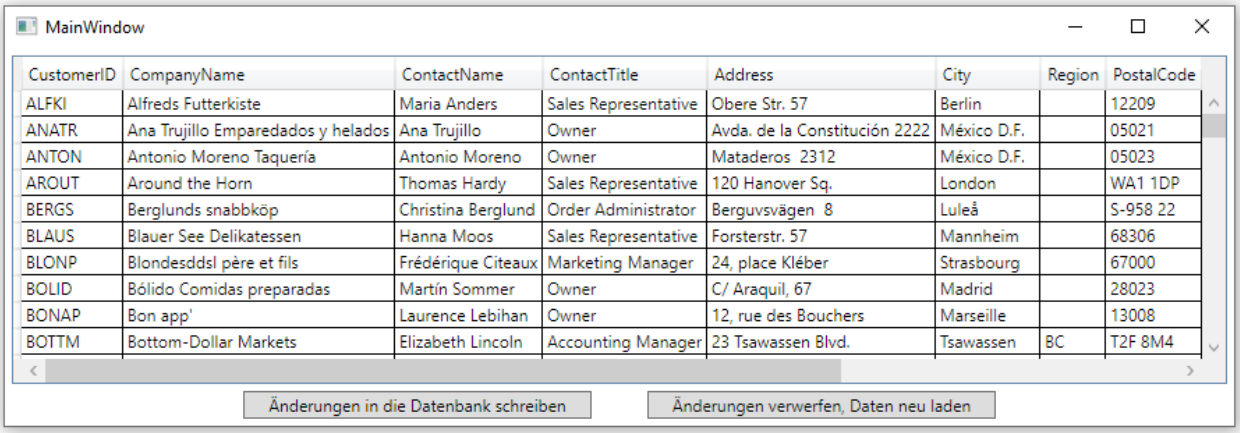
```
private void BtnSaveClick(object sender, RoutedEventArgs e) {
    try {
        adapter.Update(dataset.Customers);
        MessageBox.Show("Änderungen in die Datenbank geschrieben");
    } catch (Exception ex) {
        MessageBox.Show(ex.ToString());
    }
}
```

Aufgrund der lokal installierten Datenbank ist mit einem sehr kleinen Zeitaufwand zu rechnen, sodass die synchrone Ausführung der Methode im UI-Thread akzeptabel ist.

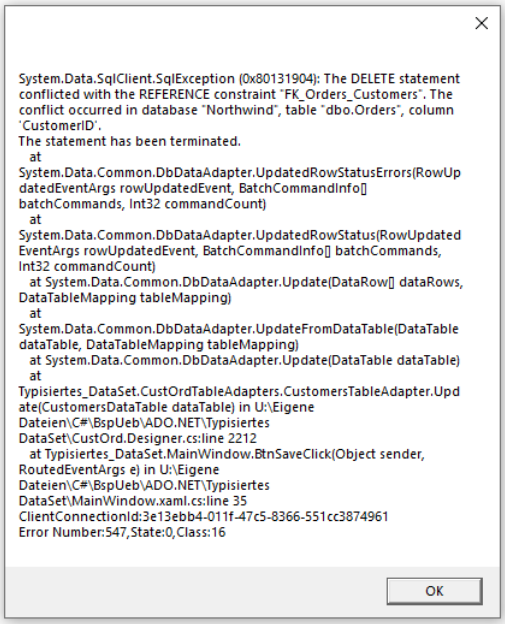
Diese Aufwandseinschätzung gilt auch für den **Fill()** - Aufruf an den Adapter, mit dem der Benutzer den Zustand im **DataGrid**-Steuerelement auf den Zustand in der Datenbank zurücksetzen kann:

```
private void BtnCancelClick(object sender, RoutedEventArgs e) {
    try {
        adapter.Fill(dataset.Customers);
        MessageBox.Show("Änderungen verworfen, Daten neu geladen");
    } catch (Exception ex) {
        MessageBox.Show(ex.ToString());
    }
}
```

Damit ist das Programm verwendbar, wenn auch noch nicht ausgereift:



Wird nach dem Löschen einer `Customers`-Zeile aus dem typisierten DataSet das Speichern in die Datenbank angefordert, dann kommt es zu einem Ausnahmefehler, sofern Aufträge dieses Kunden in der `Orders`-Tabelle vorhanden sind:



Das vollständige Projekt befindet sich im Ordner

…\BspUeb\ADO.NET\Typisiertes DataSet

18.8 Microsoft SQL Server 2019 Express

Der auf benutzerunabhängig und permanent ausgeführten Dienstprogrammen basierende SQL Server Express besitzt im Vergleich zu der bei Bedarf zur Unterstützung eines einzelnen Benutzers gestarteten LocalDB-Variante zusätzliche Kompetenzen, die entsprechende Konfigurationen erfordern:

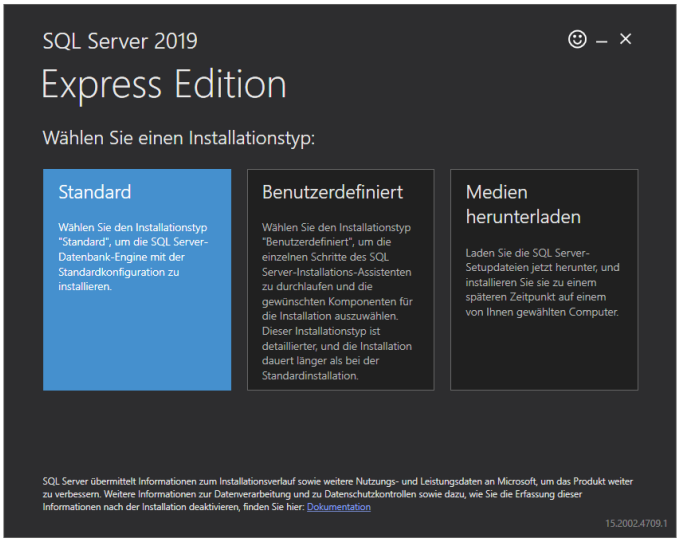
- Das DBMS kann simultan von mehreren Klienten benutzt werden. Die Zugriffsrechte werden rollenbasiert konfiguriert (siehe Abschnitt 18.8.4.1).
- Es ist eine Nutzung durch Klienten auf anderen Rechnern über Netzwerkverbindungen unter Verwendung des TCP/IP - Protokolls möglich, sodass die Konnektivität administriert werden muss (siehe Abschnitt 18.8.4.2).[1]

18.8.1 Installation

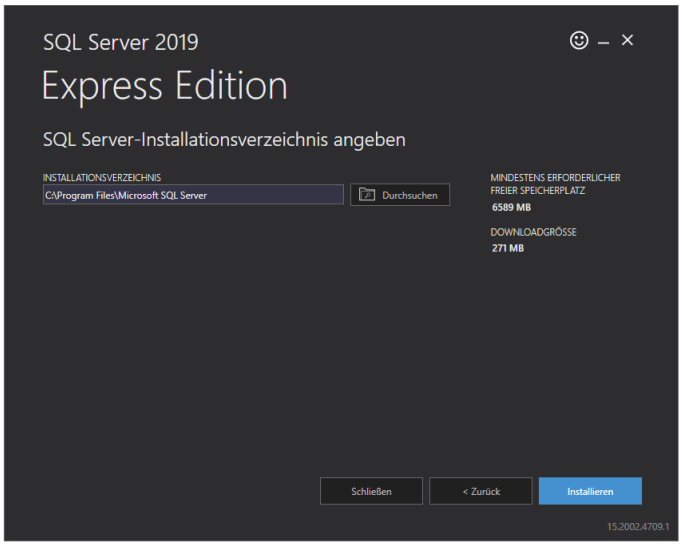
Das Installationsprogramm zu Microsofts SQL Server 2019 Express wird hier angeboten:

https://www.microsoft.com/de-de/sql-server/sql-server-downloads

Nach einem Doppelklick auf die heruntergeladene Datei (z. B. **SQL2019-SSEI-Expr.exe**) wählen wir die **Standard**-Installation,



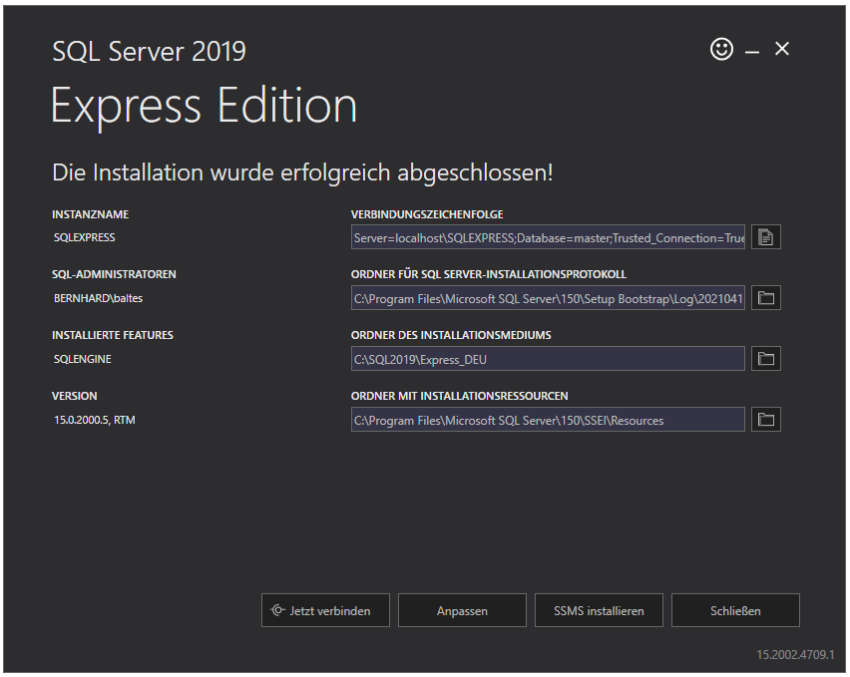
akzeptieren die Lizenzbedingungen und starten mit dem **Installieren** unter Verwendung des voreingestellten Installationsordners:



Nach einigen Minuten sollte die Erfolgsmeldung mit Angaben zu den verwendeten Installationsordnern und zur Version des SQL Servers erscheinen:

---

[1] Mit den im Abschnitt 18.8 gelegentlich ohne große Erläuterung auftauchenden Netzwerkbegriffe (z.B. TCP/IP, UDP, Port) werden wir uns im Kapitel 21 ausführlich beschäftigen.



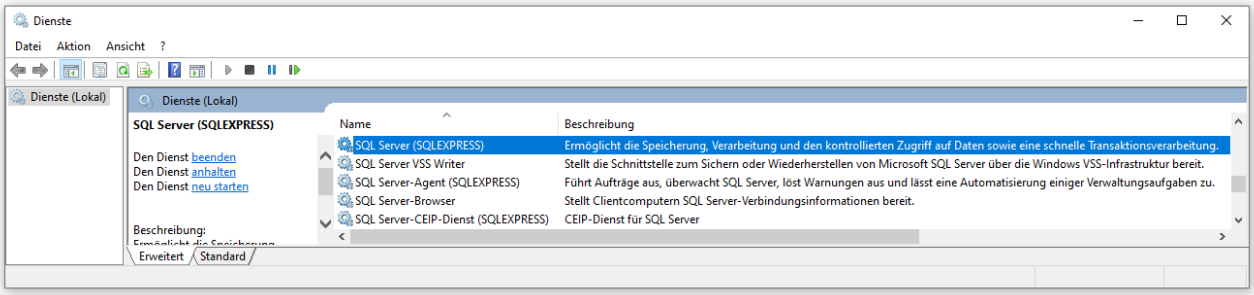
Anschließend befindet sich der SQL-Server im Ordner,

C:\Programme\Microsoft SQL Server

und auf Ihrem Rechner läuft der automatisch gestartete Windows-Dienst

SQL Server (SQLEXPRESS)

zusammen mit vier weiteren, zum SQL - Server gehörenden Diensten:



Folglich ist der SQL-Server auch ohne einen benutzerinitiierten Start dienstbereit, zunächst aber nur für lokal angemeldete Benutzer, weil per Voreinstellung keine Netzzugriffe erlaubt sind (siehe Abschnitt 18.8.4.2).

Als freundlichen Service liefert der Abschluss-Dialog des SQL Server - Installationsprogramms für die vom **SqlConnection**-Objekt des ADO.NET - Providers benötigte Verbindungszeichenfolge ein Beispiel (vgl. 18.8.3):

```
Server=localhost\SQLEXPRESS;Database=master;Trusted_Connection=True;
```

Außerdem befindet sich im Abschluss-Dialog der Schalter **SSMS installieren**, der die folgende Webseite zum **SQL Server Management Studio** öffnet:

https://docs.microsoft.com/de-de/sql/ssms/download-sql-server-management-studio-ssms?redirectedfrom=MSDN&view=sql-server-ver15

Wir haben dieses nützliche Programm schon im Abschnitt 18.4 installiert.

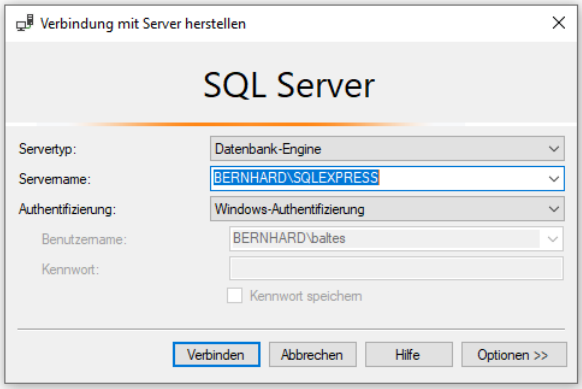
18.8.2 Kommunikation mit dem SQL Server 2019 Express

Für die Kommunikation mit dem SQL Server 2019 Express (z.B. zur Administration des Servers oder zur Inspektion von Datenbanken) stehen uns aufgrund von vorangegangenen Installationen zwei Optionen zur Verfügung:

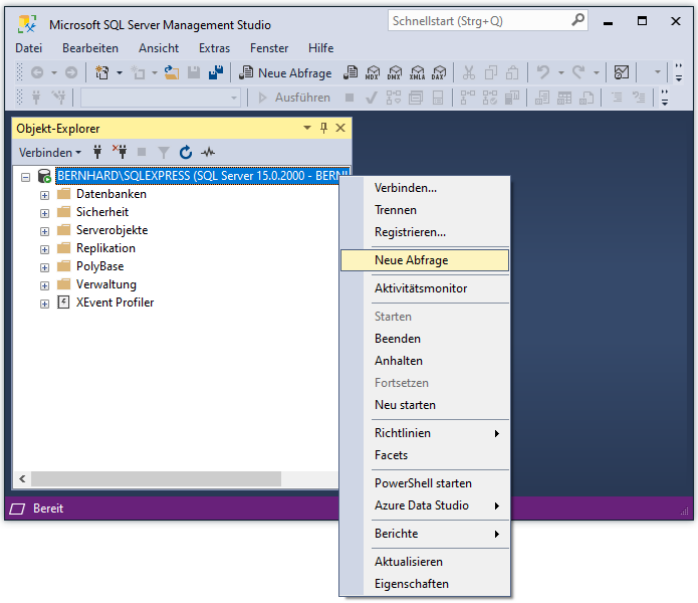
- Das SQL Server Management Studio (siehe Abschnitt 18.8.2.1)
- Der SQL Server-Objekt-Explorer im Visual Studio (siehe Abschnitt 18.8.2.2)

18.8.2.1 SQL Server Management Studio

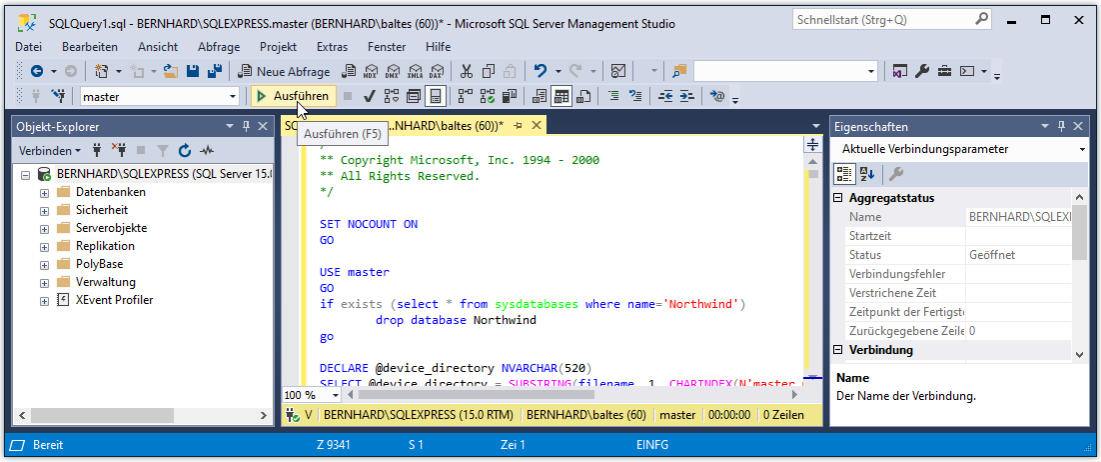
Das SQL Server Management Studio erlaubt nach dem Start im folgenden Dialog die Aufnahme der Verbindung zu einer SQL Server Express - Instanz:



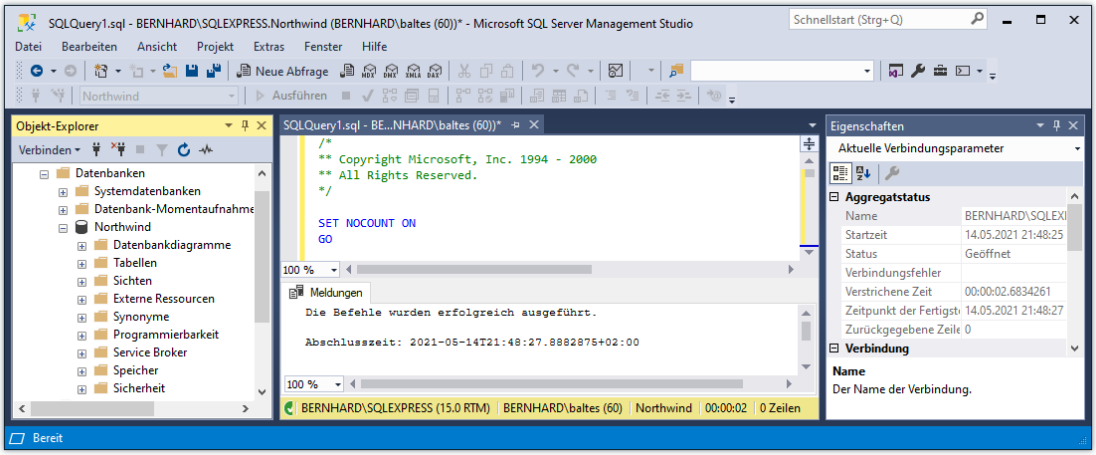
Um die **Northwind**-Datenbank für den SQL Server Express mit Hilfe der T-SQL - Skriptdatei **instnwnd.sql** (vgl. Abschnitt 18.3.2) zu erstellen, wählen wir im **Objekt-Explorer** aus dem Kontextmenü zur Server-Instanz das Item **Neue Abfrage**:



Dann fügen wir das Skript in das Editor-Fenster ein



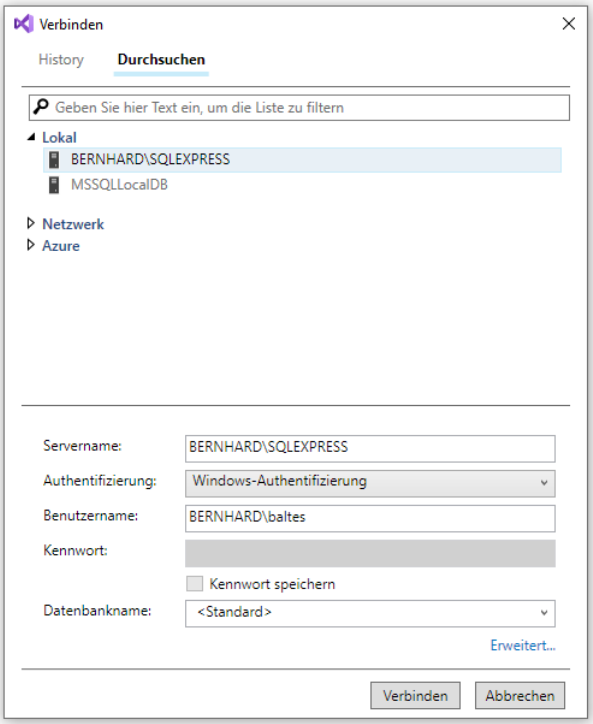
und betätigen den Schalter **Ausführen**. Wenige Sekunden später präsentiert das DBMS die **Northwind**-Datenbank:



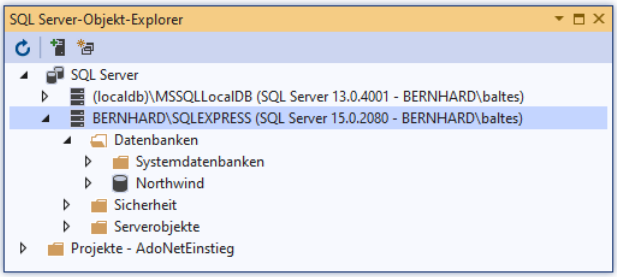
18.8.2.2 SQL Server-Objekt-Explorer im Visual Studio

Weil wir die SQL Server Data Tools installiert haben (siehe Abschnitt 3.3.2.3), steht im Visual Studio der (bei Bedarf per **Ansicht**-Menü zu startende) **SQL Server-Objekt-Explorer** bereit, den wir bereits zur Kooperation mit der LocalDB-Variante des SQL Servers verwendet haben (siehe Abschnitt 18.3.1):

Nach einem Klick auf den Schalter **SQL Server hinzufügen** kann mit dem folgenden Dialog



die Verbindung zum SQL Server Express hergestellt werden:



18.8.3 Verbindungszeichenfolge

Für die Verbindung zu einer bestimmten Datenbank benötigt ein **DbConnection**-Objekt etliche Informationen, die als sogenannte *Verbindungszeichenfolge* dem Konstruktor übergeben oder der Eigenschaft **ConnectionString** zugewiesen werden können, z. B.:

```
string cs = @"Server=tcp:bernhard\SQLEXPRESS; " +
            "Initial Catalog=Northwind; Integrated Security=true";
using SqlConnection dbConnection = new(cs);
```

Die Verbindungszeichenfolge besteht aus „Parameter=Wert“ - Zuweisungen, die jeweils durch ein Semikolon getrennt werden. In Abhängigkeit vom Provider und von sonstigen Bedingungen kommen zahlreiche Varianten in Frage.[1]

Bislang haben wir ein Verbindungsobjekt aus der von **DbConnection** abstammenden Klasse **SqlConnection** des ADO.NET - Providers **SqlClient** für den Zugriff auf einen LocalDB-Server verwendet, wobei eine einfache Verbindungszeichenfolge genügte. Wir verwenden weiterhin die Klasse **SqlConnection**, arbeiten aber nun mit einem mehrbenutzer- und netzwerktauglichen SQL-

[1] Es sind sogar spezialisierte Internet-Angebote mit Vorschlägen zur Bildung von korrekten Verbindungszeichenfolgen entstanden, z. B. http://www.connectionstrings.com/

Server. Anschließend werden häufig benötigte Parameter der Verbindungszeichenfolge beschrieben. Über weitere Parameter informiert z. B. die BCL-Dokumentation zur **ConnectionString**-Eigenschaft der Klasse **SqlConnection**.[1]

Im Abschnitt 18.6.13 wird erläutert, wie man eine Verbindungszeichenfolge aus der Anwendungskonfigurationsdatei **appsettings.json** lesen kann, damit die Anwendung bei einer Änderung der Verbindungszeichenfolge nicht neu übersetzt werden muss.

18.8.3.1 SQL-Server

Zur Bezeichnung dieses Parameters kann man zwischen den folgenden synonymen Schlüsselwörtern wählen:

Data Source, **Server**, **Address**, **Addr**, **Network Address**

Im Wert können u. a. auftreten:

Name eines Rechners, IP-Adresse, Name einer SQL Server - Instanz auf einem Rechner, Protokollbezeichnung, Portnummer

Wir beschränken uns auf zwei wichtige Spezialfälle:

- **Der SQL-Server läuft auf demselben Rechner wie die Klientenanwendung**
  In diesem Fall ist für den Wert die folgende Syntax zu verwenden:

  Data Source=*Rechnername\Instanzname*

  Den lokalen Rechner spricht man mit einem Punkt, mit **localhost** oder mit **(local)** an, wobei die BCL-Dokumentation zur letztgenannten Variante rät. Über den Instanznamen kann man mehrere Installationen des SQL-Servers auf demselben Rechner unterscheiden. Die SQL Server 2019 Express Edition wird per Voreinstellung mit dem Instanznamen SQLEXPRESS installiert.

  Beispiel:

  ```
  Data Source=(local)\SQLEXPRESS
  ```

- **Der SQL-Server wird über ein Netzwerk per TCP/IP - Protokoll angesprochen**
  Die Erreichbarkeit eines SQL-Servers über das Netzwerk unter Verwendung des TCP/IP - Protokolls (an einem bestimmten Port) muss explizit freigegeben werden (siehe Abschnitt 18.8.4.2). Ist diese Voraussetzung erfüllt, dann kann der SQL-Server in der Verbindungszeichenfolge folgendermaßen angesprochen werden:

  Server=tcp:*Rechnername\Instanzname*[, *portnumber*]

  Die Portnummer darf fehlen, wenn auf dem Server der Dienst **SQL Server-Browser** läuft und den UDP-Port 1434 abhört (siehe Abschnitt 18.8.4.2.2).

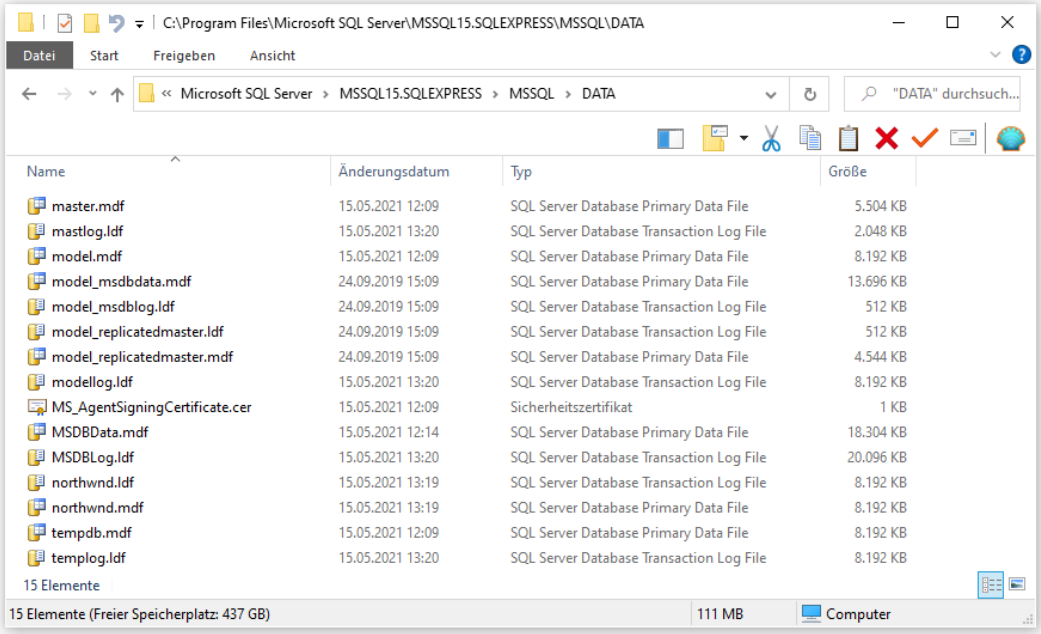
  Beispiel:

  ```
  Server=tcp:bernhard\SQLEXPRESS
  ```

[1] https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlconnection.connectionstring

18.8.3.2 Datenbank

Eine im voreingestellten Datenbankordner des SQL-Servers befindliche Datenbank



kann über ihren logischen Namen angesprochen werden, z. B.:

```
Database=Northwind
```

Das Schlüsselwort **Database** kann ersetzt werden durch **Initial Catalog**:

```
Initial Catalog=Northwind
```

18.8.3.3 Benutzer-Authentifizierung

Microsofts SQL Server unterstützen die Windows - und die SQL Server - Authentifizierung (siehe Abschnitt 18.8.4.1). Um die von Microsoft nachdrücklich empfohlene Windows - Authentifizierung zu nutzen, setzt man den Parameter **Integrated Security** auf den Wert **true** oder **sspi**, z. B.:

```
Integrated Security=true
```

Für den Parameter **Integrated Security** kann auch die Bezeichnung **Trusted_Connection** verwendet werden.

Bei der SQL Server - Authentifizierung sind die Parameter **User** (alias **User ID**) und **Password** zu verwenden, z. B.:

```
User ID=Otto;Password=123456
```

Ein Passwort sollte allerdings nicht in den Quellcode geschrieben, sondern z. B. beim Benutzer erfragt werden.

18.8.3.4 Kommunikations-Parameter

Über den Parameter **Connect Timeout** legt man fest, wie lange auf eine erfolgreiche Verbindung zum SQL Server gewartet werden soll (Voreinstellung: 15 Sekunden), z. B.:

```
Connection Timeout=3
```

Beim Datenbankzugriff über ein Netzwerk kann es sich bei einem großen Transportvolumen lohnen, über den Parameter **Packet Size** die Netzwerkpakete zu vergrößern (Voreinstellung: 8192 Bytes), z. B.:

```
Packet Size= 32768
```

18.8.4 Konfiguration

Ein SQL Server ist ein mächtiges System mit sehr vielen Einstellschrauben. Wir beschränken und auf die Benutzerrechte und die Netzwerkkonfiguration.

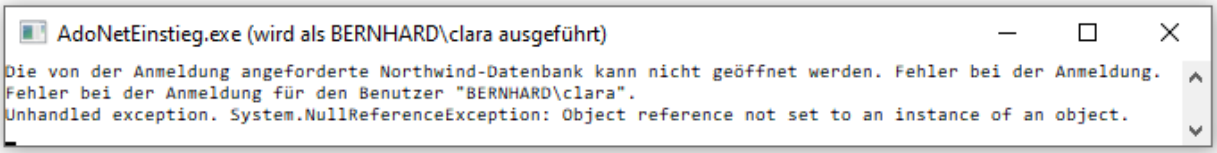
18.8.4.1 Rechteverwaltung

Bei der Verwaltung von Rechten geht es um:

- Authentifizierung
  Welche Konten (Benutzer, Gruppen) dürfen sich beim SQL Server anmelden? Eine solche Anmeldung ist z. B. dann erforderlich, wenn ein Programm im Auftrag und mit den Rechten des Benutzers mit einem SQL Server kooperieren möchte. Als Methode zur Benutzer-Authentifizierung empfiehlt Microsoft die Nutzung der Windows-Konten. Jedoch unterstützt der SQL-Server auch interne Konten.
- Autorisierung
  Welche Rechte haben die einzelnen Konten?

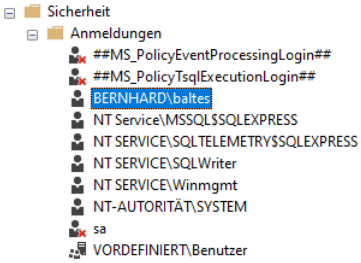
18.8.4.1.1 Anmeldung beim SQL-Server für ein Windows-Konto erlauben

Besitzt ein Windows-Konto keine ausreichenden Rechte für die Anmeldung beim SQL Server Express bzw. für den Zugriff auf eine Datenbank, dann scheitert die Verwendung von Programmen, die solche Rechte benötigen, z. B.:

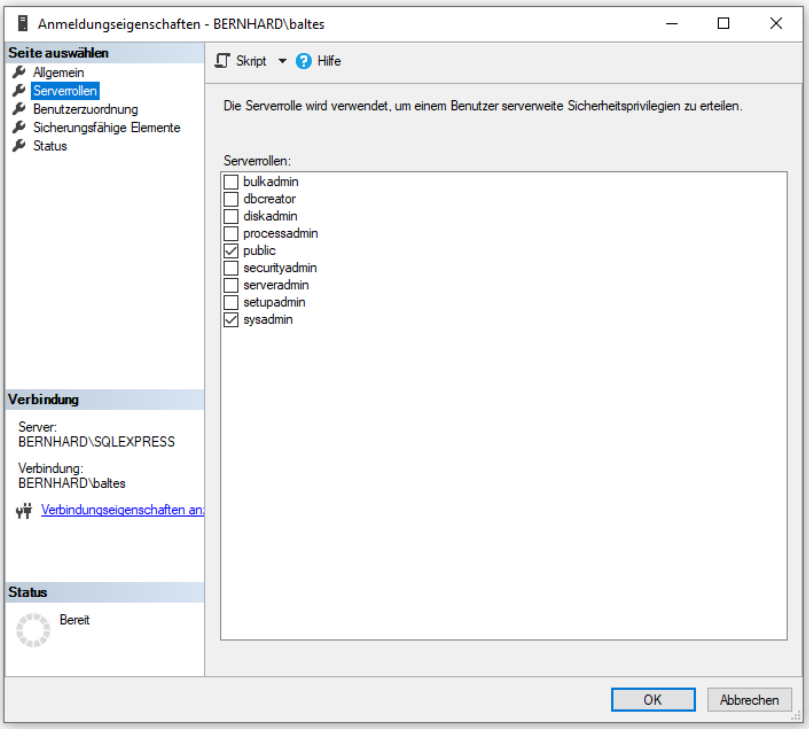


Gehen Sie folgendermaßen vor, um einem Windows-Konto den Zugriff auf den SQL Server Express zu erlauben:

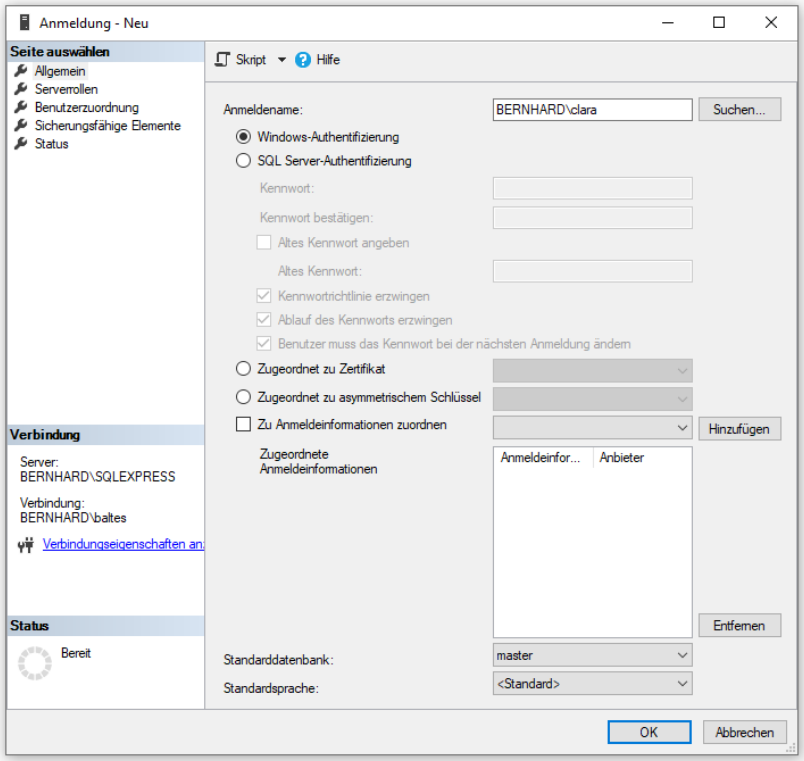
- Starten Sie das **SQL Server Management Studio**, und stellen Sie die Verbindung zum SQL Server Express mit dem Konto eines SQL-Server-Administrators her.
- Ob Ihr Windows-Konto tatsächlich die Serverrolle **sysadmin** besitzt, lässt sich über den **Objekt-Explorer** feststellen. Das Konto befindet sich dann in der Liste zum Knoten **Sicherheit > Anmeldungen**,

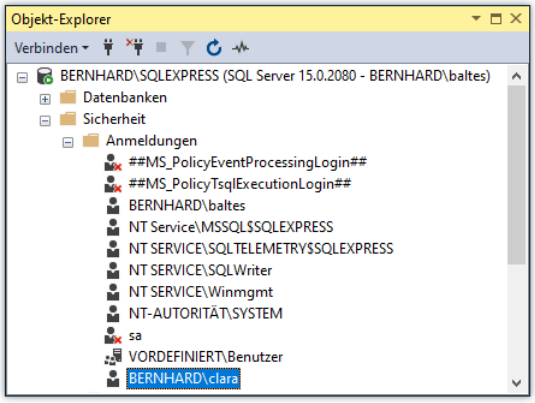
  und im **Eigenschafen**-Fenster zum Konto ist unter den **Serverrollen** auch **sysadmin** markiert:



- Wählen Sie aus dem Kontextmenü zum Knoten **Sicherheit > Anmeldungen** das Item **Neue Anmeldung**. Es erscheint die Dialogbox **Anmeldung - Neu** (siehe unten).
- Behalten Sie die voreingestellte **Windows-Authentifizierung** bei. Tragen Sie einen **Anmeldenamen** ein, oder klicken Sie auf **Suchen**, um einen Namen ohne Tippfehlerrisiko auszuwählen. Nach **Suchen** müssen Sie in der Dialogbox **Benutzer oder Gruppe wählen** auf **Erweitert** und anschließend auf **Jetzt suchen** klicken, um schließlich eine Liste mit den auf Ihrem System bekannten Benutzern und Gruppen zu erhalten.
- Nach Aufnahme eines neuen Benutzers sieht die Dialogbox **Anmeldung - Neu** so aus:



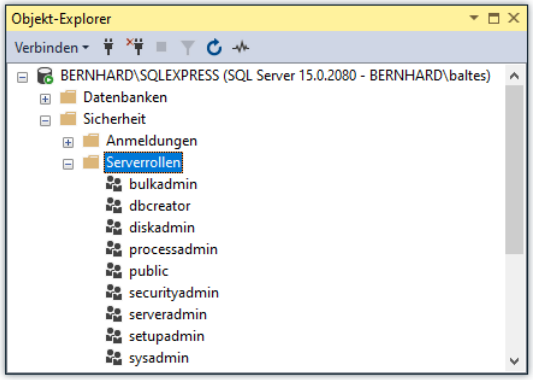
Der neue Benutzer wird im Knoten **Sicherheit > Anmeldungen** des Objekt-Explorers aufgelistet:



18.8.4.1.2 Autorisierung durch Rollen

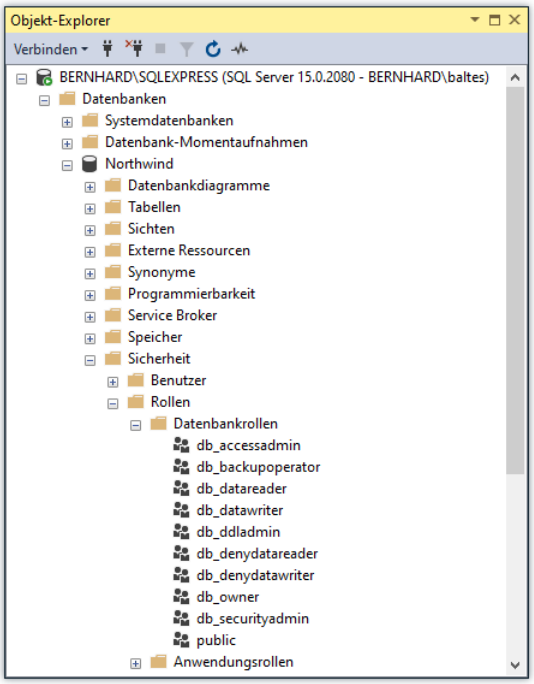
Zur Verwaltung von Benutzerrechten verwendet der SQL Server **Rollen**. Dabei sind zu unterscheiden:

- **Serverrollen**
  Diese sind vordefiniert und beziehen sich auf den gesamten SQL Server. Im SQL Server Management Studio zeigt der **Objekt-Explorer** diese Rollen unter **Sicherheit > Serverrollen**.



  Wer die Serverrolle **sysadmin** besitzt, der darf den SQL Server verwalten.

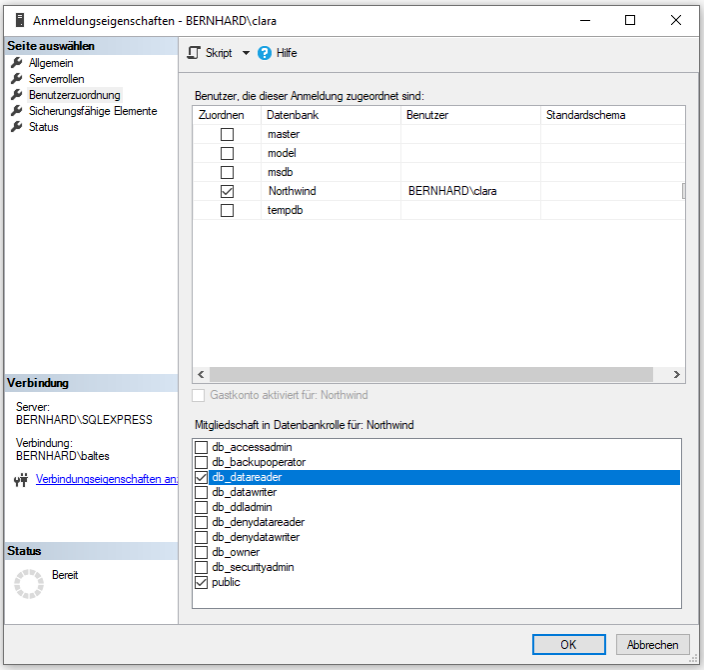
- **Datenbankrollen**
  Diese werden für jede Datenbank separat definiert und folglich vom **Objekt-Explorer** des Management Studios im Sicherheitsordner einer Datenbank aufgelistet, z. B.:



Die vordefinierten Datenbankrollen können durch Eigenkreationen ergänzt werden.

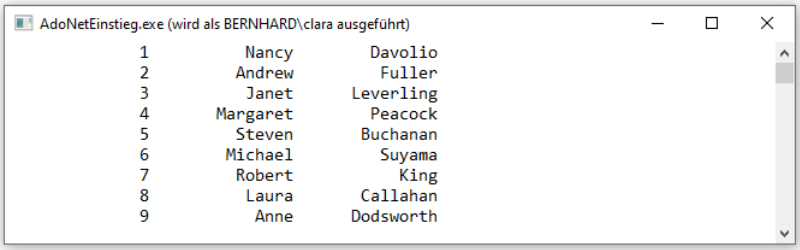
- **Anwendungsrollen**
  Zu einer Datenbank können auch Anwendungsrollen definiert werden, um Rechte zu vergeben, die nicht vom Benutzer abhängen, der am Klientenrechner angemeldet ist, sondern von der eingesetzten Klientenanmeldung. Weil dabei ein Passwort im Spiel und notwendigerweise auf dem Klientenrechner gespeichert ist, gelten solche Lösungen mittlerweile als unsicher.

Um den im Abschnitt 18.8.4.1.1 eingetragenen Benutzer zu autorisieren, öffnen wir im **Objekt-Explorer** des Management Studios über **Sicherheit > Anmeldungen** das Kontextmenü zu seinem Namen und starten mit dem Item **Eigenschaften** den Dialog mit den **Anmeldungseigenschaften**. Wir verzichten auf die Vergabe von **Serverrollen**, öffnen die Seite **Benutzerzuordnung**, und vergeben für die Datenbank **Northwind** die Rolle **db_datareader**:



Damit kann der Benutzer nur lesend auf die Datenbank **Northwind** zugreifen. Mit der Rolle **public**, die einem Benutzer nicht entzogen werden kann, sind keine nennenswerten Rechte verbunden.

Das ohne Autorisierung mit einem Ausnahmefehler gescheiterte Programm (siehe Abschnitt 18.8.4.1.1) kann vom nun berechtigten Konto erfolgreich ausgeführt werden:

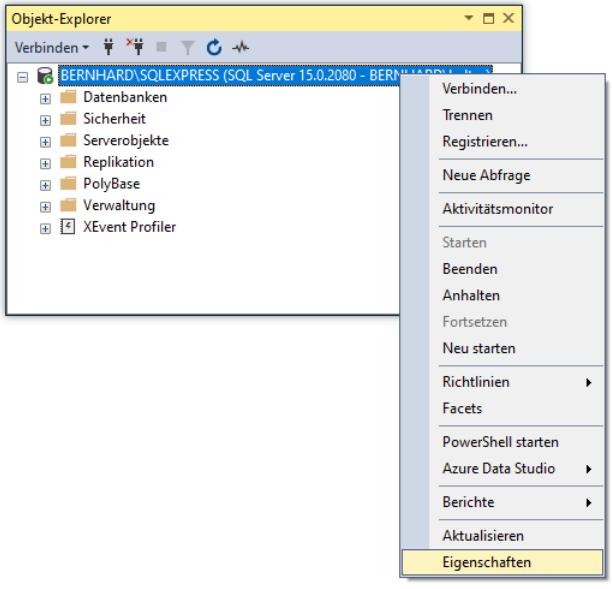


18.8.4.1.3 SQL Server - Authentifizierung erlauben

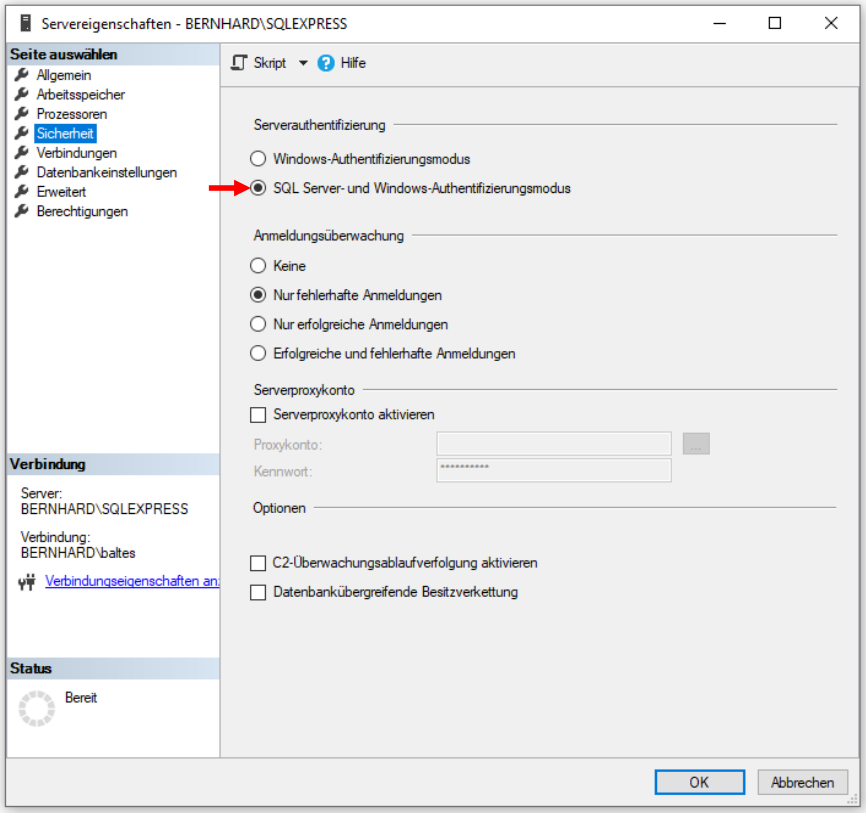
In einer Netzwerkumgebung *ohne* Windows - Domänencontroller ist man bei der Windows - Authentifizierung auf lokale Benutzerkonten (definiert auf dem Rechner mit dem SQL Server) beschränkt. Damit ein Benutzer von einem anderen Rechner aus via Netzwerk auf den SQL Server zugreifen kann, muss auf dem Klienten-PC ein gleichnamiges Windows-Konto mit identischem Passwort eingerichtet werden. In dieser Situation kann es sinnvoller sein, für den SQL Server zusätzlich zur Windows - Authentifizierung auch die SQL Server - Authentifizierung zu erlauben.

Nach einer Standardinstallation unterstützt der SQL Server 2019 Express nur die Windows - Authentifizierung. Im Management Studio lässt sich jedoch die SQL Server - Authentifizierung aktivieren:

- Wählen Sie im **Objekt-Explorer** aus dem Kontextmenü zur SQL-Serverinstanz das Item **Eigenschaften**, z. B.:

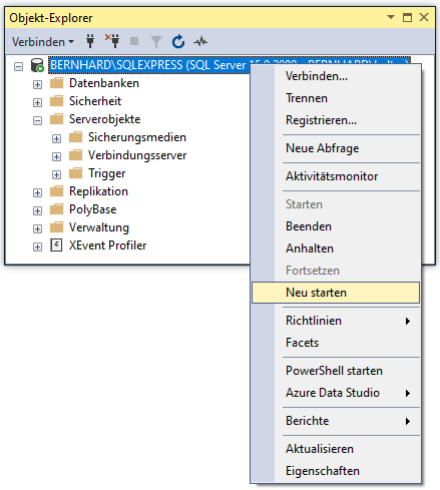


- Öffnen Sie in der Dialogbox **Servereigenschaften** die Seite **Sicherheit**.
- Wählen Sie die Option **SQL Server- und Windows-Authentifizierungsmodus**,



und quittieren Sie mit **OK**.

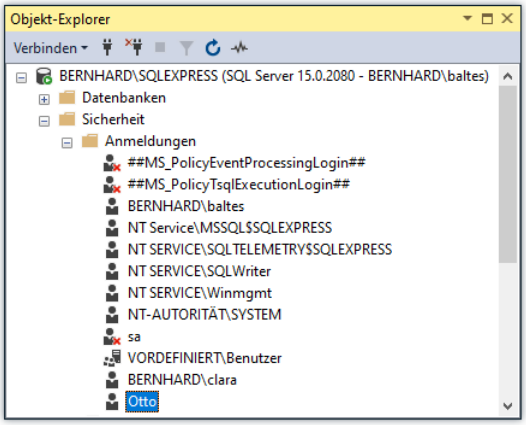
- Anschließend muss der SQL-Server neu gestartet werden, was per **Objekt-Explorer** möglich ist:



Wir legen nun per Management Studio ein SQL Server - internes Konto an:

- Öffnen Sie im **Objekt-Explorer** aus dem Kontextmenü zum Knoten **Sicherheit > Anmeldungen** das Item **Neue Anmeldung**. Es erscheint die Dialogbox **Anmeldung - Neu**.
- Vergeben Sie einen **Anmeldenamen**, wählen Sie die **SQL Server - Authentifizierung**, tragen Sie ein Kennwort ein, und entfernen Sie die Markierung des Kontrollkästchens **Benutzer muss das Kennwort bei der nächsten Anmeldung ändern**.
- Vergeben Sie auf der Seite **Benutzerzuordnung** für die Datenbank **Northwind** die Rollen **db_datareader** und **db_datawriter** (siehe Abschnitt 18.8.4.1.2).

Der neue Benutzer erscheint im Ordner **Sicherheit > Anmeldungen** des **Objekt-Explorers**, z. B.:



Bei der Autorisierung per Rollenvergabe gibt es keine Unterschiede zwischen SQL Server - und Windows - Konten.

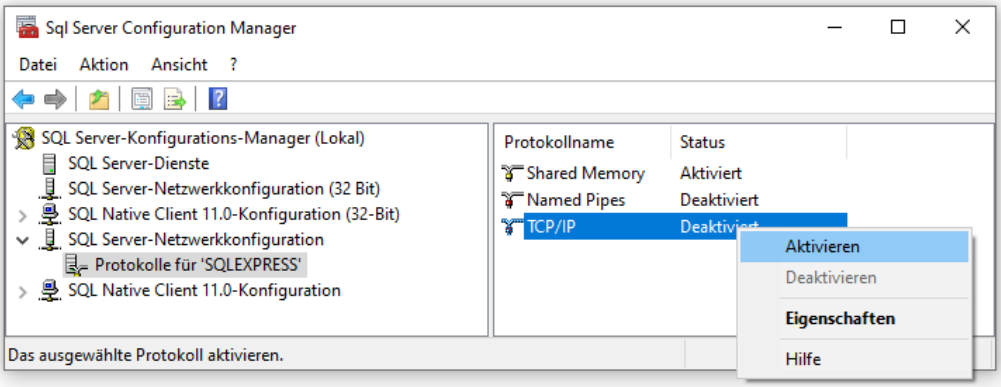
18.8.4.2 Netzwerkzugriff via TCP/IP erlauben

Per Voreinstellung steht der SQL-Server nur lokal angemeldeten Benutzern zur Verfügung, was für die Beispielprogramme im Kapitel 18 meist genügt. Trotzdem soll an dieser Stelle erläutert werden, wie man den Netzwerkzugriff über das TCP/IP - Protokoll auf den SQL Server erlaubt (zur erforderlichen Verbindungszeichenfolge siehe Abschnitt 18.8.3.1). Dazu sind drei Maßnahmen erforderlich:

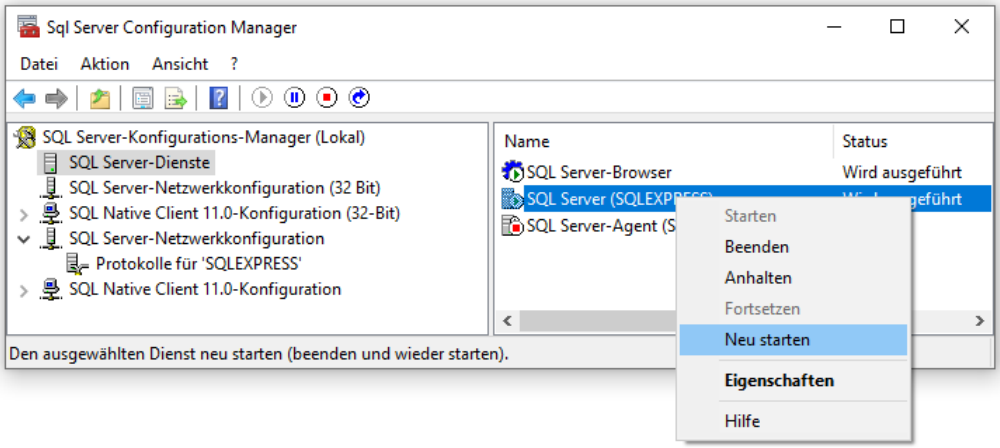
18.8.4.2.1 Netzwerkprotokoll TCP/IP aktivieren

Arbeitsschritte:

- Starten Sie den **SQL Server 2019 Konfigurations-Manager** über seinen Link im Unterordner **Konfigurationstools** der Programmgruppe zum **Microsoft SQL Server 2019**.
- Wählen Sie in der Baumansicht im linken Segment des Dialogs den Knoten
  SQL Server-Netzwerkkonfiguration > Protokolle für 'SQLEXPRESS'.
- Aktivieren Sie im rechten Segment des Dialogs das **TCP/IP** - Protokoll per Kontextmenü:



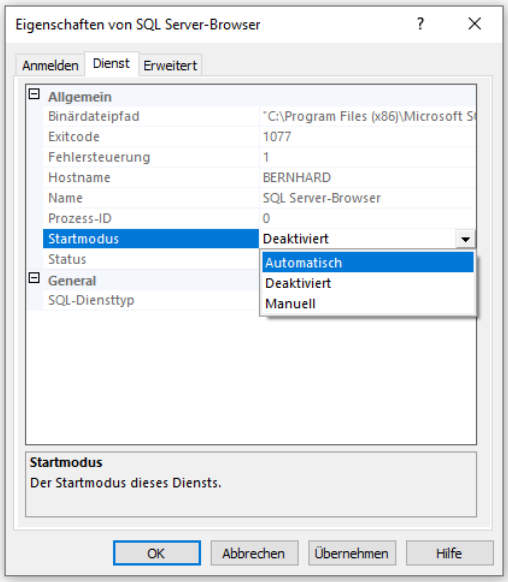
- Wählen Sie in der Baumansicht auf der linken Seite den Knoten **SQL Server-Dienste**, und starten Sie per Kontextmenü den Windows-Dienst **SQL Server (SQLEXPRESS)** neu:



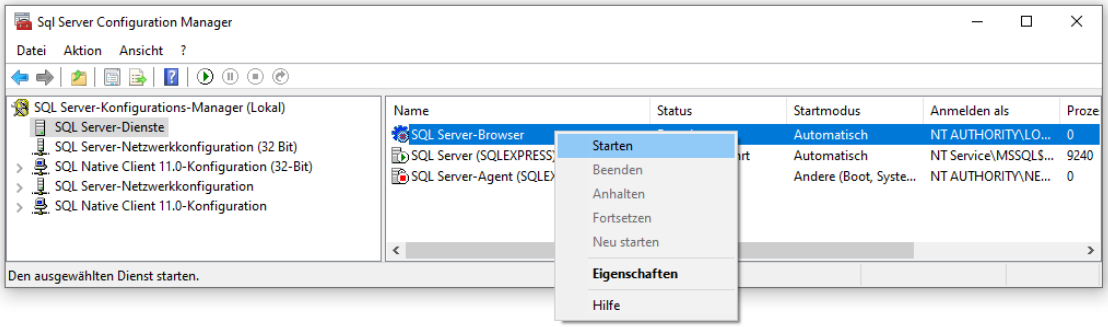
18.8.4.2.2 Dienst SQL Server-Browser starten

Wenn der SQL Server ohne Angabe einer Portnummer in der Verbindungszeichenfolge (vgl. Abschnitt 18.6.3) über Netz ansprechbar sein soll, dann muss der **SQL Server Browser Dienst** laufen. Sorgen Sie nötigenfalls für den automatischen Start:

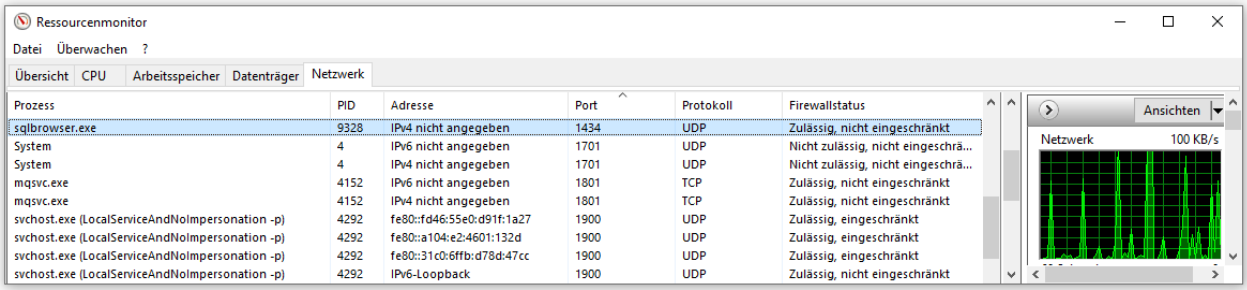
- Starten Sie den **SQL Server Configuration Manager** über seinen Link im Unterordner **Konfigurationstools** der Programmgruppe zum **Microsoft SQL-Server 2019**.
- Markieren Sie in der Baumansicht im linken Segment des Dialogs den Knoten **SQL Server-Dienste**, und wählen Sie per Kontextmenü den Eigenschaftsdialog zum Dienst **SQL Server-Browser**.
- Sorgen Sie auf der Registerkarte **Dienst** für den **automatischen** Start:



- Der **SQL Server-Browser** kann über sein Kontextmenü gestartet werden:



Wie man unter Windows z. B. mit dem **Ressourcenmonitor** verifizieren kann, lauscht der Dienst **SQL Server-Browser** am UDP-Port 1434:



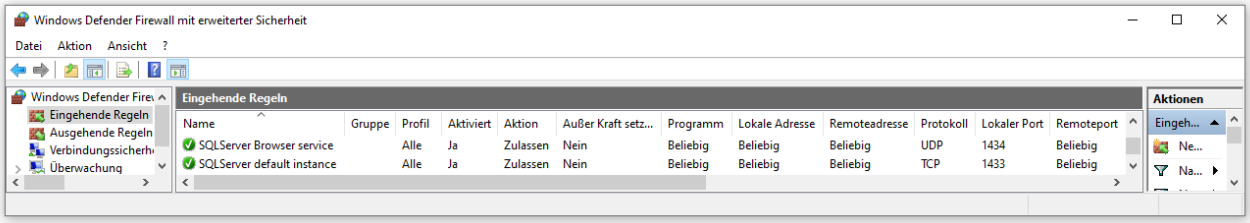
Er wird von der Datei **sqlbrowser.exe** realisiert.

18.8.4.2.3 Firewall-Ausnahmen definieren

Für die Dienste **SQL Server** und **SQL Server-Browser** muss jeweils eine Firewall-Ausnahme eingetragen werden, was z. B. unter Windows 10 nach einer Empfehlung von Microsoft[1] in einer mit administrativen Rechten ausgeführten PowerShell-Konsole durch die folgenden Kommandos geschehen kann:

```
New-NetFirewallRule -DisplayName "SQLServer default instance" -Direction Inbound -
LocalPort 1433 -Protocol TCP -Action Allow
New-NetFirewallRule -DisplayName "SQLServer Browser service" -Direction Inbound -
LocalPort 1434 -Protocol UDP -Action Allow
```

In der grafischen Bedienoberfläche der **Defender Firewall** werden die neuen Regeln so beschrieben:



Zum Starten der grafischen Bedienoberfläche kann man so vorgehen:

- **Firewall** in das Windows-Suchfeld eintippen
- Wahl des Treffers **Windows Defender Firewall mit erweiterter Sicherheit**

[1] https://docs.microsoft.com/de-de/sql/sql-server/install/configure-the-windows-firewall-to-allow-sql-server-access?view=sql-server-ver15

19 LINQ-to-Objects

Seit C# 3.0 ist eine einheitliche, für diverse Datenquellen geeignete, an SQL angelehnte Abfragetechnik namens *LINQ* (*Language Integrated Query*) verfügbar. In diesem Kapitel werden wir LINQ-Methoden für die standardisierte Verarbeitung von Kollektions-Objekten oder Arrays kennenlernen. Wir werden also sogenannte *LINQ-to-Objects - Abfragen* oder *lokale Abfragen* durchführen.

Eine zentrale Rolle spielen dabei Erweiterungsmethoden, welche die statische Klasse **Enumerable** im Namensraum **System.Linq** für alle Typen realisiert, welche die Schnittstelle **IEnumerable<T>** implementieren (zu Erweiterungsmethoden siehe Abschnitt 7.13). Solche Typen (z. B. Arrays) sollen im Folgenden als *Sequenzen* bezeichnet werden. Die ca. 40 Erweiterungsmethoden der Klasse **Enumerable** (auch *Standard-Abfrage-Operatoren* genannt) akzeptieren als Parameter eine Eingabesequenz und liefern als Rückgabe meist eine Ausgabesequenz, die im nächsten Schritt weiterverarbeitet werden kann. Manche LINQ-Methoden liefern aber auch ein spezielles Element der Eingabesequenz oder einen Wert (z. B. eine Summe). Neben diesen beiden Arten von Operatoren

- Sequenz → Sequenz
- Sequenz → Einzelelement oder Wert

existieren noch LINQ-Operatoren zum Erstellen von Sequenzen (z. B. die **Enumerable**-Erweiterungsmethode **Range()**).

Damit in einer Quellcodedatei die **Enumerable**-Erweiterungsfunktionen auf **IEnumerable<T>** - Instanzen wie eigene Instanzmethoden angewendet werden können, ist das Importieren des Namensraums **System.Linq** per **using**-Direktive empfehlenswert.

Im Kapitel 20 über das Entity Framework Core werden LINQ-Abfragen dazu verwendet, Informationen aus SQL-Datenbanken zu beziehen. Bei diesen sogenannten *LINQ-to-Entities* - Abfragen spielen Erweiterungsmethoden, welche die statische Klasse **Queryable** im Namensraum **System.Linq** für alle Typen realisiert, welche die von **IEnumerable<T>** abgeleitete Schnittstelle **IQueryable<T>** implementieren, eine wesentliche Rolle.

Auch die LINQ-Abfragetechnik kann im Manuskript nicht komplett dargestellt werden. Weiterführende Informationen finden sich z. B. in der Microsoft-Online-Dokumentation[1] sowie in Albahari & Johannsen (2020, S. 369ff).

19.1 Erweiterungsmethoden

Wenngleich zur Unterstützung der LINQ-Anwendung spezielle Abfrageausdrücke in die Programmiersprache C# eingeführt worden sind (siehe Abschnitt 19.2), übersetzt der Compiler letztlich alle LINQ-to-Objects - Datenzugriffe in Aufrufe von Erweiterungsmethoden der Klasse **Enumerable**, und viele LINQ-Optionen sind ausschließlich über die direkte Verwendung dieser Methodenaufrufe verfügbar. Das Manuskript behandelt daher zunächst die Verwendung von LINQ-to-Objects über Erweiterungsmethoden, beschreibt später aber auch die Abfrageausdrücke.

[1] https://docs.microsoft.com/en-us/dotnet/csharp/linq/

19.1.1 Technische Realisation

Auf den ersten Blick wirken C# - Anweisungen mit LINQ-to-Objects - Abfragemethoden ungewohnt, z. B.:

```
var custMex = customers
              .Where(c => c.Country == "Mexico")
              .Select(c => new {c.CustomerID, c.Country});
```

Durch die anschließenden Erläuterungen der technischen Realisation soll diesem Fremdheitseindruck entgegengewirkt werden.

19.1.1.1 Funktionalitäts-Injektion per Lambda-Notation

Wer in die objektorientierte Programmierung eingedacht ist und die Lambda-Notation kennt, der wird mit dem folgenden Beispiel keine Probleme haben. In der letzten Anweisung werden die **Enumerable**-Erweiterungsmethoden **Where()** und **Select()** sequentiell auf einen Array vom Typ `Customer[]` angewendet:

```
using System;
using System.Linq;

class Customer {
   public string CustomerId { get; set; }
   public string CompanyName { get; set; }
   public string Country { get; set; }
}
. . .
Customer[] customers = new Customer[7];
 . . .
var custMex = customers
              .Where(c => c.Country == "Mexico")
              .Select(c => new {c.CustomerID, c.Country});
```

Dass der Compiler im Beispiel auf Erweiterungsmethoden in der Klasse **Enumerable** zurückgreift und z. B. nicht etwa auf Erweiterungsmethoden in der Klasse **Queryable**, liegt daran, dass die Variable `customers` vom Typ `Customer[]` ist, der die Schnittstelle **IEnumerable**<`Customer`> erfüllt, die abgeleitete Schnittstelle **IQueryable**<`Customer`> aber nicht.

Die generische Erweiterungsmethode **Where()** ist folgendermaßen definiert:

public static IEnumerable<TSource> Where<TSource>(
this IEnumerable<TSource> *source*, **Func<TSource, bool>** *predicate*)

Durch den ersten Parameter mit der für Erweiterungsmethoden vorgeschriebenen Syntax (siehe Abschnitt 7.13) wird der erweiterte Datentyp festgelegt, der ...

- die Schnittstelle **IEnumerable<TSource>** erfüllen,
- also eine Methode namens **GetEnumerator()** besitzen muss
- mit einem Enumerator als Rückgabe,
- der sukzessive Instanzen vom Typ **TSource** liefert.

Der erweiterte Typ kann also die Rolle einer Datenquelle spielen. Als zweiter Parameter wird von **Where()** eine Methode erwartet, die einen **TSource**-Parameter entgegennimmt und einen Rückgabewert vom Typ **bool** liefert, die also den folgenden Delegatentyp erfüllt:

public delegate bool Func<in TSource, out bool>(TSource *arg*)

Als **Where()** - Rückgabe erhält man ein Objekt vom Typ **IEnumerable<TSource>**, im Beispiel also vom Typ **IEnumerable**<`Customer`>, sodass die nächste LINQ-Methode mit der Verarbeitung fortfahren kann.

Die generische Erweiterungsmethode **Select()** ist folgendermaßen definiert:

public static IEnumerable<TResult> Select<TSource, TResult>(
this IEnumerable<TSource> *source*, **Func<TSource, TResult>** *selector*)

Als zweiten Parameter erwartet **Select()** eine Methode, die einen **TSource**-Parameter entgegennimmt und eine Rückgabe vom Typ **TResult** liefert, die also den folgenden Delegatentyp erfüllt:

public delegate TResult Func<in TSource, out TResult>(TSource *arg*)

Als **Select()** - Rückgabe erhält man ein Objekt, das die generische Schnittstelle **IEnumerable<TResult>** mit Elementen vom Typ **TResult** erfüllt (zu Details siehe Abschnitt 19.1.3.3). Im Beispiel wird an **Select()** eine per Lambda-Ausdruck realisierte Methode übergeben, die ein Objekt einer anonymen Klasse liefert, das eine Auswahl der `Customer`-Eigenschaften besitzt.

Die „anonyme“ Klasse hat durchaus einen (vom Compiler vergebenen) Namen, der jedoch im Quellcode nicht verwendet werden darf. Bei der Deklaration der Variablen `custMex` mit dem Ergebnis der Abfrage ist daher das Schlüsselwort **var** nicht zu vermeiden.

Offenbar realisiert **Where()** einen *Filter*, während **Select()** die verbliebenen Elemente in ein neues Format überführt (*projiziert*).

Während im Beispiel die **Enumerable**-Erweiterungsmethoden (die Abfrage-Operatoren) jeweils eine komplette Eingabesequenz entgegennehmen und eine neu erstellte Ausgabesequenz liefern, verarbeiten die per Lambda-Notation realisierten Delegatenobjekte ein Element der Eingabesequenz und liefern ein Element der Ausgabesequenz. Die Erweiterungsmethoden erhalten jeweils per Delegatenobjekt eine Funktionalitäts-Injektion. Durch die per Lambda-Notation realisierten Hilfsmethoden lässt sich die Erweiterungsmethodensyntax sehr kompakt formulieren.

Im Beispiel wird für die Parameter der beiden per Lambda-Notation realisierten Delegatenmethoden derselbe Bezeichner verwendet. Das ist problemlos möglich, weil es sich um lokale Variablen der Methoden handelt.

Die im zweiten Parameter der Erweiterungsmethoden erwartete Funktionalitäts-Injektion muss nicht per Lambda-Notation vorgenommen werden. Man kann auch eine konventionell realisierte Methode verwenden, sofern diese den geforderten Delegatentyp realisiert, z. B.:

```
bool fromMex(Customer c) {
   return c.Country == "Mexico";
}

var custMex = customers
               .Where(fromMex)
               .Select(c => new {c.CustomerID, c.Country});
```

Diese Vorgehensweise kommt z. B. dann in Frage, wenn eine komplexe Delegatenmethode mehrfach benötigt wird.

Im folgenden Beispielprogramm taucht die Klasse `Customer` auf, die nicht zufällig an die Tabelle `Customers` aus der Beispieldatenbank **Northwind** erinnert (siehe Kapitel 18):

```
using System;
using System.Linq;

class Customer {
   public string CustomerId { get; set; }
   public string CompanyName { get; set; }
   public string Country { get; set; }
}
```

```
class LinqToObjectsEinstieg {
   static void Main() {
      Customer[] customers = new Customer[7];
      customers[0] = new Customer { CustomerId = "ALFKI",
            CompanyName = "Alfreds Futterkiste", Country = "Germany" };
      customers[1] = new Customer { CustomerId = "ANATR",
            CompanyName = "Ana Trujillo Emparedados y helados", Country = "Mexico" };
       . . .
      customers[6] = new Customer { CustomerId = "BLONP",
            CompanyName = "Blondesddsl père et fils", Country = "France" };

      var custMex = customers
                    .Where(c => c.Country == "Mexico")
                    .Select(c => new { c.CustomerId, c.CompanyName });

      foreach (var c in custMex)
         Console.WriteLine(c.CustomerId + "   " + c.CompanyName);
   }
}
```

Das Programm liefert die folgende Ausgabe:

```
ANATR  Ana Trujillo Emparedados y helados
ANTON  Antonio Moreno Taquería
```

19.1.1.2 Fluent-API

Weil viele LINQ-to-Objects - Erweiterungsmethoden eine Rückgabe vom Typ **IEnumerable<T>** liefern, lassen sich die Aufrufe hintereinanderschalten, wobei man von einem *Fluent-API* spricht. Ein Abfrage-Operator modifiziert (dem Paradigma der funktionalen Programmierung folgend) seine Eingabesequenz *nicht*, sondern er liefert eine *neu erstellte* Ausgabesequenz.

Die folgende Anweisung nimmt für den im Abschnitt 19.1.1.1 vorgestellten Array `customers` vom Typ `Customer[]` eine Filterung durch die Erweiterungsmethode **Where()**, eine absteigende Sortierung durch die Erweiterungsmethode **OrderByDescending()** und eine Projektion durch die Erweiterungsmethode **Select()** vor:

```
var compName = customers
                  .Where(c => c.Country == "Germany")
                  .OrderByDescending(c => c.CustomerId)
                  .Select(c => c.CompanyName.ToUpper());
```

Die Erweiterungsmethoden **Where()** und **Select()** wurden schon im Abschnitt 19.1.1.1 beschrieben. Es folgt die Beschreibung der im Beispiel zum Sortieren in absteigender Ordnung verwendeten **OrderByDescending()** - Überladung:

public static IOrderedEnumerable<TSource> OrderByDescending<TSource, TKey>(
this IEnumerable<TSource> *source*, **Func<TSource, TKey>** *key*)

Zum Sortieren in *aufsteigender* Ordnung verwendet man die **Enumerable**-Erweiterungsmethode **OrderBy()**, die eine analoge Syntax besitzt.

Das Parameterobjekt vom Delegatentyp **Func<TSource, TKey>** liefert für jedes Element der Eingabesequenz eine die Sortierposition definierende Rückgabe vom Typ **TKey**, der entweder die Schnittstelle **IComparable<TKey>** oder die Schnittstelle **IComparable** implementieren muss. Im Beispiel wird eine Eigenschaft des Elementtyps `Customer` benannt. In der folgenden Variante werden die Elemente der Eingabesequenz nach der Länge der `CompanyName`-Eigenschaftsausprägung aufsteigend sortiert:

```
var compNames = customers
                  .Where(c => c.Country == "Germany")
                  .OrderBy(c => c.CompanyName.Length)
                  .Select(c => c.CompanyName.ToUpper());
```

Besitzt die Konkretisierung des Typformalparameters **TKey** keine Anordnung, dann tritt kein Übersetzungsfehler auf, sondern ein Laufzeitfehler. Das passiert erst dann, wenn die **OrderBy()** - Eingabesequenz tatsächlich durchlaufen werden muss (z. B. im Rahmen einer **for**-Schleife), z. B.:

```
Unhandled exception. System.InvalidOperationException: Failed to compare two elements
in the array.
 ---> System.ArgumentException: At least one object must implement IComparable.
```

Hier zeigt sich die für LINQ-Abfrageoperatoren typische verzögerte Ausführung (engl.: *lazy execution* oder *deferred execution*, siehe Abschnitt 19.1.2).

Im Beispiel besitzt die Variable `compNames` mit dem Abfrageergebnis den Typ **IEnumerable<String>**, weil vom **Select()** - Operator ein Objekt dieses Typs aus der **IEnumerable<Customer>** - Eingabesequenz erstellt wird. Die bequeme Verwendung des Schlüsselworts **var** statt einer Ergebnistypdeklaration kann durch die praktisch immer vorhandene Möglichkeit zur Nutzung der QuickInfo-Anzeige der Entwicklungsumgebung gerechtfertigt werden:

```
var compNames = customers
                  .Where(c => c.Country == "Germany")
                  .OrderBy(c => c.CompanyName.Length)
                  .Select(c => c.CompanyName.ToUpper());
```

(Lokale Variable) IEnumerable<string> compNames

Manchmal verbessert die explizite Angabe des Ergebnistyps aber doch die Lesbarkeit des Quellcodes:

```
IEnumerable<string> compNames = customers
                                  .Where(c => c.Country == "Germany")
                                  .OrderBy(c => c.CompanyName.Length)
                                  .Select(c => c.CompanyName.ToUpper());
```

Wenn die Elemente der Ergebnissequenz einen *anonymen* Typ haben, dann ist das Schlüsselwort **var** allerdings nicht zu vermeiden, weil ein solcher Typ im Quellcode nicht notiert werden kann. Es überrascht kaum, dass anonyme Klassen (siehe Abschnitt 6.5) oft als Ergebnistyp einer LINQ-Projektion auftauchen, denn die anonymen Klassen sind zur Unterstützung der LINQ-Abfragen eingeführt worden.

Selbstverständlich eignet sich die Erweiterungsmethodensyntax auch zur schrittweisen Verarbeitung bzw. zur Erstellung von Zwischenergebnissen, z. B.:

```
IEnumerable<Customer> custGerm = customers
                                   .Where(c => c.Country == "Germany");
bool up = true;
 . . .
IOrderedEnumerable<Customer> result;
if (up)
   result = custGerm.OrderBy(c => c.CompanyName.Length);
else
   result = custGerm.OrderByDescending(c => c.CompanyName.Length);
```

19.1.2 Verzögerte Ausführung

Nachdem z. B. die folgende Anweisung ausgeführt worden ist,

```
var compNames = customers
                  .Where(c => c.Country == "Germany")
                  .OrderBy(c => c.CompanyName)
                  .Select(c => c.CompanyName);
```

befindet sich in der Ergebnisvariablen `compNames` vom Typ **IEnumerable<String>** *keine* Kollektion, deren Inhalt wiederholt identisch ausgelesen werden könnte. Stattdessen wird ...

- eine Folge von Verarbeitungsschritten (inklusive Lambda-Ausdrücke)
- zusammen mit einer Referenz auf die Quelle

gespeichert. Diese Verarbeitungsschritte werden erst dann ausgeführt, wenn aufgrund einer Anforderung der Ergebnissequenz ein Iterieren durch die Eingabesequenz erforderlich ist, z. B. im Rahmen einer **foreach**-Schleife:

Quellcodesegment	Ausgabe
`foreach (var c in compNames)` `  Console.WriteLine(c);`	`Alfreds Futterkiste` `Blauer See Delikatessen`

Wird das Abfrageergebnis erneut angefordert, laufen alle Verarbeitungsschritte erneut ab, sodass z. B. eine zwischenzeitliche Änderung der Quelle das Ergebnis beeinflussen kann:

Quellcodesegment	Ausgabe

`customers[0].CompanyName = "Alfredos Futterkiste";`
`foreach (var c in compNames)`
`  Console.WriteLine(c);`
	`Alfredos Futterkiste`
	`Blauer See Delikatessen`

Diese verzögerte Ausführung (engl.: *lazy execution* oder *deferred execution*) hat offensichtliche Vorteile:

- Ein Aufwand wird erst dann betrieben, wenn er wirklich erforderlich ist.
- Bei einer wiederholten Abfrage wird der jeweils aktuelle Zustand der Quelle berücksichtigt.

Bei einer *unveränderten* Quelle führt die wiederholte Ausführung von eventuell aufwändigen Arbeitsschritten (Filterung, Sortierung, Projektion) allerdings zu einem nutzlosen Aufwand. Um diesen Aufwand zu vermeiden, kann ein Abfrageergebnis mit der **Enumerable**-Methode **ToArray()** in einen Array oder mit der **Enumerable**-Methode **ToList()** in eine Liste überführt werden, z. B.:

Quellcodesegment	Ausgabe

`var compNames = customers`
`  .Where(c => c.Country == "Germany")`
`  .OrderBy(c => c.CompanyName)`
`  .Select(c => c.CompanyName);`
`var compNamesList = compNames.ToList();`
`customers[0].CompanyName = "Alfredos Futterkiste";`
`Console.WriteLine("Iteration über die Abfrage:");`
`foreach (var c in compNames)`
`  Console.WriteLine(c);`
`Console.WriteLine("\nIteration über die Liste:");`
`foreach (var c in compNamesList)`
`  Console.WriteLine(c);`
	`Iteration über die Abfrage:`
	`Alfredos Futterkiste`
	`Blauer See Delikatessen`
	
	`Iteration über die Liste:`
	`Alfreds Futterkiste`
	`Blauer See Delikatessen`

Ein **ToArray()** - oder **ToList()** - Aufruf führt dazu, dass die im LINQ- Abfrageergebnis vom Typ **IEnumerable<T>** enthaltenen Verarbeitungsschritte *sofort* ausgeführt werden. So kann bei man-

chen Algorithmen Rechenzeit eingespart werden, indem z. B. ein wiederholtes Sortieren unterbleibt. Allerdings muss ein erhöhter Speicherbedarf in Kauf genommen werden.

Auch andere LINQ-Methoden werden *sofort* ausgeführt (z. B. die sogenannten *Elementextraktoren* **Min()** und **Max()**, siehe Abschnitt 19.1.3.7.2). Auf der folgenden Webseite dokumentiert Microsoft, welche LINQ-Methoden verzögert bzw. sofort ausgeführt werden:

https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/concepts/linq/classification-of-standard-query-operators-by-manner-of-execution

19.1.3 Abfrageoperatoren

Wenn in der Literatur oder Dokumentation etwas nebulös von einem *LINQ-Provider* gesprochen wird, dann ist eine Klasse mit Erweiterungsmethoden für die Schnittstelle **IEnumerable<TSource>** (bei LINQ-to-Objects) bzw. für die Schnittstelle **IQueryable<TSource>** (bei LINQ-to-Entities) gemeint. Dort definierte Erweiterungsmethoden werden als *LINQ-Operatoren* bezeichnet. Diese Bezeichnung ist nicht sonderlich glücklich, weil *kein* Bezug zu den C# - Operatoren im Sinn von Abschnitt 4.5 besteht.

Der aktuelle Abschnitt behandelt Erweiterungsmethoden bzw. LINQ-Operatoren, die in der Klasse **Enumerable** (im Namensraum **System.Linq**) für die Schnittstelle **IEnumerable<TSource>** definiert sind. Weil diese Schnittstelle von fast allen Kollektionen implementiert wird (siehe Kapitel 11), können die LINQ-to-Objects - Operatoren auf praktisch alle Kollektionen angewendet werden. Außerdem fallen auch Arrays in den LINQ-to-Objects - Anwendungsbereich, wobei die Implementation der Schnittstelle **IEnumerable<TSource>** hier nicht so offensichtlich ist.[1]

19.1.3.1 Where()

Mit der **Enumerable**-Erweiterungsmethode **Where<TSource>()**

public static IEnumerable<TSource> Where<TSource>(
this IEnumerable<TSource> *quelle*, **Func<TSource, bool>** *bedingung*)

wird eine Teilmenge der Quellelemente ausgewählt. Neben dem **this**-Parameter vom zu erweiternden Typ erwartet die generische Methode **Where<TSource >()** einen Parameter vom Delegatentyp **Func<TSource, bool>**. Ein Objekt dieses Typs zeigt auf eine Methode mit einem Parameter vom Typ **TSource** und einem booleschen Rückgabewert, welche für jedes Element der Quelle über die Aufnahme in die Ergebnissequenz entscheidet.

19.1.3.2 OrderBy()

Bei LINQ-to-Objects bestimmt ein Enumerator die Reihenfolge der gelieferten Elemente, wobei aber oft eine alternative Reihenfolge benötigt wird.[2] Über die Erweiterungsmethoden **OrderBy()** oder **OrderByDescending()** lässt sich die gewünschte Reihenfolge der Elemente in der Ergebnissequenz herstellen, z. B.:

```
var cust = customers
             .OrderBy(c => c.CompanyName);
```

Als Ergebnistyp resultiert bei LINQ-to-Objects die von **IEnumerable<TSource>** abstammende Schnittstelle **IOrderedEnumerable<TSource>**.

---

[1] https://stackoverflow.com/questions/2773740/why-do-arrays-in-net-only-implement-ienumerable-and-not-ienumerablet

[2] Bei LINQ-to-Entities (siehe Abschnitt 20.5) ist keine voreingestellte Reihenfolge vorhanden, weil das involvierte DBMS bei einer **SELECT**-Abfrage (mit **WHERE**-Klausel) interne Optimierungen vornimmt, sodass für die gelieferten Elemente in der Regel keine Reihenfolge definiert ist (Griffiths 2020).

Ist ein hierarchisches Sortieren nach *mehreren* Kriterien erforderlich, dann ist ein Aufruf der Methode **OrderBy()** bzw. **OrderByDescending()** durch einen Aufruf oder durch mehrere Aufrufe der Methoden **ThenBy()** bzw. **ThenByDescending()** zu ergänzen. z. B.:

Quellcode	Ausgabe

`using System;`
`using System.Linq;`
 	
`class Qtype {`
`  public char Category { get; set; }`
`  public int Level { get; set; }`
`  public Qtype(char cat, int lev) {`
`  Category = cat; Level = lev;`
`  }`
`}`
 	
`class ThenBy {`
`  static void Main() {`
`  var prods = new Qtype[5] { new('A', 3), new('B', 1), new('B', 2),`
`  new('C', 2), new('C', 1) };`
`  var sortedProds = prods`
`  .LevelBy(p => p.Category)`
`  .ThenByDescending(p => p.Level);`
`  foreach (var p in sortedProds)`
`  Console.WriteLine(p.Category + " " + p.Level);`
`  }`
`}`
	`A 3`
	`B 2`
	`B 1`
	`C 2`
	`C 1`

Von der Eingabesequenz für die Erweiterungsmethoden **ThenBy()** und **ThenByDescending()** wird naheliegender weise verlangt, die von **IEnumerable<TSource>** abgeleitete Schnittstelle **IOrderedEnumerable<TSource>** zu implementieren, z. B:

public static IOrderedEnumerable<TSource> ThenBy<TSource, TKey>(
this IOrderedEnumerable<TSource> *source*, **Func<TSource, TKey>** *keySelector***)**

Viele LINQ-Operatoren sind bemüht, ...

- *so spät wie möglich* tätig zu werden (siehe Abschnitt 19.1.2 über die verzögerte Ausführung)
- und dabei *möglichst wenige* Elemente der Eingabesequenz zu verarbeiten.

Eine Sortierung ändert nichts an der verzögerten Ausführung. Ist diese aber nicht mehr aufzuschieben, dann müssen natürlich *alle* Elemente der Eingabesequenz verarbeitet werden.

19.1.3.3 Select()

Mit der **Enumerable**-Erweiterungsmethode **Select<TSource, TResult>()**

public static IEnumerable<TResult> Select<TSource, TResult>(
this IEnumerable<TSource> *quelle*, **Func<TSource, TResult>** *projektor***)**

wird der Elementtyp der Abfrage festgelegt, wobei man von einer *Projektion* des Quelltyps auf den Ausgabetyp spricht. Neben dem **this**-Parameter vom zu erweiternden Typ **IEnumerable<TSource>** erwartet **Select<TSource, TResult>()** einen Parameter vom Delegatentyp **Func<TSource, TResult>**:

public delegate TResult Func<in TSource, out TResult>(TSource *quelle***)**

Ein Objekt dieses Typs zeigt auf eine Methode mit einem Parameter vom Typ **TSource**, die eine Rückgabe vom Typ **TResult** liefert.

Im folgenden Beispiel

```
var custIds = customers.Select(c => c.CustomerId);
```

sind die Elemente der Datenquelle vom Typ `Customer`

```
class Customer {
   public string CustomerId { get; set; }
   public string CompanyName { get; set; }
   public string Country { get; set; }
}
```

und die Elemente der Abfrage vom Typ **String**. Das projizierende Delegatenobjekt wird per Lambda-Notation erstellt.

Weil die verzögerte Ausführung von LINQ-Abfrageoperatoren (vgl. Abschnitt 19.1.2) im Unterschied zu den Erweiterungsmethoden und der Lambda-Notation wirklich neu und ungewohnt ist, sollen die Konsequenzen noch einmal demonstriert werden. Die Methode **Select()** liefert ein Objekt, das alle zur Durchführung der Abfrage erforderlichen Informationen enthält, diese aber erst beim Datenzugriff verwendet. So wird ein stets aktuelles, auf Quelländerungen reagierendes Ergebnis sichergestellt. Im folgenden Beispiel wird der Wert eines Quellelements *nach* dem **Select()** - Aufruf, aber *vor* dem **ElementAt()** - Aufruf geändert, sodass in der verzögert ausgeführten Abfrage der aktuelle Zustand enthalten ist:

Quellcodesegment	Ausgabe

`Console.WriteLine(customers[6].CustomerId);`
`var custIds = customers.Select(c => c.CustomerId);`
`customers[6].CustomerId = "SEVEN";`
`Console.WriteLine(custIds.ElementAt(6));`
`customers[6].CustomerId = "SIEBEN";`
`Console.WriteLine(custIds.ElementAt(6));`
	`BLONP`
	`SEVEN`
	`SIEBEN`

In den obigen Beispielen liefert der Projektionsoperator **Select()** ein einzelnes Feld des Quellelementtyps, was zu einer besonders einfachen Syntax führt. Wird ein zusammengesetzter Abfrageelementtyp benötigt, dann verwendet man meist eine anonyme Klasse. Der folgende Erweiterungsmethodenaufruf

```
var cust = customers.Select(c => new {c.CustomerId, c.Country});
```

liefert Abfrageelemente mit den beiden Eigenschaften `CustomerId` und `Country`.

Die „anonyme“ Klasse hat durchaus einen (vom Compiler vergebenen) Namen, doch darf dieser Name im Quellcode nicht verwendet werden. Daher ist das Schlüsselwort **var** erforderlich, um eine Variable mit dem Ergebnistyp der Abfrage deklarieren zu können.

Eine explizit definierte Klasse statt einer anonymen Klasse als Ausgabetyp zu verwenden, hat z. B. den Vorteil, dass dort auch Methoden definiert werden können.

Statt einer Klasse sind als Elementtyp für die Ergebnissequenz auch zulässig:

- ein elementarer Typ, z. B.:
  ```
  var custNameLength = customers.Select(c => c.CompanyName.Length);
  ```
- eine Struktur, z. B.:
  ```
  struct CustStruct {
      public char FirstLetter { get; set; }
      public int NameLength { get; set; }
  }
   . . .
  var custStruct = customers.Select(c => new CustStruct {
         FirstLetter = c.CompanyName[0], NameLength = c.CompanyName.Length});
  ```

- ein Tupeltyp (seit C# 7.0, vgl. Abschnitt 6.6), z. B.:
```
var custTupel = customers.Select(c => (c.CustomerId, c.CompanyName));
```
- ein Record-Typ (seit C# 9.0, vgl. Abschnitt 6.7)
```
record CustRec(string CustomerId, string CompanyName);
 . . .
var custRec = customers.Select(c => new CustRec(c.CustomerId, c.CompanyName));
```

19.1.3.4 GroupBy()

Wenn die Elemente einer Eingabesequenz in Gruppen eingeteilt werden können, dann lässt sich mit Hilfe der **Enumerable**-Erweiterungsmethode **GroupBy()** eine Ausgabesequenz erzeugen, deren Elemente aus gruppenspezifischen Teilsequenzen bestehen. Der folgenden Überladung

public static IEnumerable<IGrouping<TKey, TSource>> GroupBy<TSource,TKey>(
this IEnumerable<TSource> *source*, **Func<TSource, TKey> keySelector)**

ist beim Aufruf als einziger Parameter ein Delegat vom Typ **Func<TSource, TKey>** zu übergeben, der zu einem Element der Eingabesequenz den Schlüsselwert liefert, der die Gruppenzugehörigkeit festlegt.

Wir betrachten ein Beispiel mit Eingabesequenz-Elementen vom Typ `Product`:

```
class Product {
   public char Supplier {get; set;}
   public int Quality {get; set;}
   public Product(char supp, int qual) {
      Supplier = supp; Quality = qual;
   }
}
```

Die `Supplier`-Eigenschaft eines Produkts (Typ **char**) enthält den einstelligen Buchstabencode des Lieferanten, und die `Quality`-Eigenschaft (Typ **int**) informiert über die Bewertung.

Für eine Liste mit Produkten

```
var prods = new List<Product> { new('A', 1), new('A', 2), new('A', 1), new('A', 3),
   new('B', 4), new('B', 2), new('B', 3), new('B', 5), new('B', 4),
   new('C', 5), new('C', 6), new('C', 4), new('C', 8), new('C', 5) };
```

wird durch die folgende Anweisung

```
var groups = prods.GroupBy(p => p.Supplier);
```

eine Ausgabesequenz mit dem Namen `groups` erstellt, die als Elemente gruppenspezifische Teilsequenzen der Produkte enthält:

- Für jeden Lieferanten existiert ein Element der äußeren Sequenz.
- Dieses Element der äußeren Sequenz ist wiederum eine Sequenz und enthält die von einem Lieferanten stammenden Produkte.
- Ein Element der äußeren Sequenz erfüllt nicht nur das Interface **IEnumerable**<`Product`>, sondern auch das daraus abgeleitete Interface **IGrouping<char,** `Product`>. Es ist eine Sequenz von Produkten, die außerdem die Eigenschaft **Key** vom Typ **char** besitzt, die den Wert der Gruppierungsvariablen enthält.

Damit besitzt `groups` den Typ **IEnumerable<IGrouping<char, Products>>**. Generell hat das Ergebnis einer Gruppierung den Typ **IEnumerable<IGrouping<TKey, TSource>>**.

Zur Präsentation eines Berichts über eine durch den Abfrageoperator **GroupBy()** erstellte Sequenz von Sequenzen eignet sich eine Verschachtelung von **foreach**-Schleifen:

```
foreach (var supp in groups) {
   Console.WriteLine("\nLieferant: " + supp.Key);
   Console.WriteLine($" Mittlere Qualität: {supp.Select(p => p.Quality).Average()}");
   foreach (var prod in supp)
      Console.Write(" " + prod.Quality);
   Console.WriteLine();
}
```

Die Laufvariable `supp` der äußeren Schleife ist vom Typ **IGrouping<char, `Product`>** und hat folglich eine Eigenschaft namens **Key**, die über den Lieferanten informiert. Weil `supp` das Interface **IEnumerable**<`Product`> erfüllt, können Erweiterungsmethoden auf diese Variable angewendet werden. Im Beispiel wird ...

- per **Select()** eine Projektion der `Product`-Elemente auf **int**-Werte mit dem Qualitätsurteil vorgenommen,
- für die von **Select()** gelieferte Sequenz von **int**-Werten über die Erweiterungsmethode **Average()** das arithmetische Mittel bestimmt (siehe Abschnitt 19.1.3.7.3),
- und der resultierende **double**-Wert ausgegeben.

Die Laufvariable `prod` der inneren Schleife ist vom Typ `Product` und besitzt folglich die Eigenschaft `Quality`, sodass die Qualitätseinschätzungen zu den einzelnen Produkten des aktuellen Lieferanten dokumentiert werden können.

Das Programm liefert die folgende Ausgabe:

```
Lieferant: A
 Mittlere Qualität: 1,75
 1 2 1 3

Lieferant: B
 Mittlere Qualität: 3,6
 4 2 3 5 4

Lieferant: C
 Mittlere Qualität: 5,6
 5 6 4 8 5
```

Zur Gruppeneinteilung lässt sich jede aus den Elementen der Ausgangssequenz berechenbare Funktion verwenden. In der folgenden Variante des Beispiels kommt eine Methode mit **bool**-Rückgabe zum Einsatz, wobei die Lieferanten mit einem Buchstabencode kleiner oder gleich 'B' die erste Gruppe bilden, und alle anderen Lieferanten in die Restgruppe fallen:

```
var groups = prods.GroupBy(p => p.Supplier <= 'B');
```

Das ansonsten unveränderte Programm liefert die Ausgabe:

```
Lieferant: True
 Mittlere Qualität: 2,7777777777777777
 1 2 1 3 4 2 3 5 4

Lieferant: False
 Mittlere Qualität: 5,6
 5 6 4 8 5
```

Eine weitere, auf der Dokumentations-Webseite von Microsoft[1] beschriebene, Option ist die Gruppierung mit Hilfe eines anonymen Typs mit mehreren Eigenschaften. Im (allmählich etwas überstrapazierten Beispiel) soll die Aufteilung nach dem Lieferanten-Buchstabencode ausdifferenziert werden durch eine dichotome Qualitätseinteilung:

```
var groups = prods.GroupBy(p => new { p.Supplier, HighQual = p.Quality > 5 });
```

---

[1] https://docs.microsoft.com/de-de/dotnet/csharp/linq/group-query-results

Der Ergebnisbericht des ansonsten unveränderten Programms sieht so aus:

```
Lieferant: { Supplier = A, HighQual = False }
 Mittlere Qualität: 1,75
 1 2 1 3

Lieferant: { Supplier = B, HighQual = False }
 Mittlere Qualität: 3,6
 4 2 3 5 4

Lieferant: { Supplier = C, HighQual = False }
 Mittlere Qualität: 4,666666666666667
 5 4 5

Lieferant: { Supplier = C, HighQual = True }
 Mittlere Qualität: 7
 6 8
```

Nun hat `supp.Key` die Eigenschaften `Supplier` und `HighQual`.

19.1.3.5 Join()

Liegen zwei Sequenzen (Quellen) vor, die in einer Master-Details - Beziehung zueinander stehen (vgl. Abschnitt 18.2.2), dann ist oft die Kombination zu einer neuen Sequenz gewünscht, deren Elemente Informationen aus beiden Quellen enthalten. Bei der Verbindung von zwei Quellelementen zu einem Ergebniselement wird meist das Inner Join - Prinzip verwendet:[1]

- In beiden Sequenzen wird ein Schlüssel festgelegt.
- Zwei Quellelemente bilden genau dann ein Ergebniselement, wenn ihre Schlüsselwerte übereinstimmen.

Die Verknüpfung von zwei Informationsquellen nach dem Inner Join - Prinzip stammt offensichtlich aus der Datenbanktechnik, hat aber auch als LINQ-to-Objects - Operator ein großes Anwendungspotential.

Bei der Inner Join - Kombination von zwei Sequenzen wird die Erweiterungsmethode **Join()** verwendet. Die folgende Überladung verwendet die **Equals()** - Methode des Typs **TKey** zur Identitätsprüfung von Schlüsselwerten:

public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult> (
this IEnumerable<TOuter> *outer***,**
IEnumerable<TInner> *inner***,**
Func<TOuter, TKey> *outerKeySelector***,**
Func<TInner, TKey> *innerKeySelector***,**
Func<TOuter, TInner, TResult> *resultSelector***)**

Der angesprochenen ersten Sequenz vom Typ **IEnumerable<TOuter>** werden im **Join()** - Aufruf als Parameter übergeben:

- die zweite Sequenz (Datentyp **IEnumerable<TInner>**)
- ein Delegat, der zu einem Element der ersten Sequenz den Schlüsselwert liefert
- ein Delegat, der zu einem Element der zweiten Sequenz den Schlüsselwert liefert
- ein Delegat, der aus einem Paar von Elementen aus der ersten und der zweiten Sequenz ein Element vom Ergebnistyp **TResult** erstellt

---

[1] Die englische Bezeichnung für diese spezielle Zusammenfassung von zwei Sequenzen wird im deutschen Sprachraum so übereinstimmend verwendet, dass eine Übersetzung nicht nützlich wäre.

Wir verwenden aus dem Beispiel im Abschnitt 19.1.3.4 die Klasse `Product` und die Liste bzw. Sequenz `prods` mit Elementen dieses Typs:

```
class Product {
   public char Supplier {get; set;}
   public int Quality {get; set;}
   public Product(char supp, int qual) {
      Supplier = supp; Quality = qual;
   }
}
 . . .
var prods = new List<Product> { new('A', 1), new('A', 2), new('A', 1), new('A', 3),
   new('B', 4), new('B', 2), new('B', 3), new('B', 5), new('B', 4),
   new('C', 5), new('C', 6), new('C', 4), new('C', 8), new('C', 5) };
```

Ein Element dieser Details-Sequenz enthält eine Produktbewertung (Typ **int**) und einen Buchstabencode für den Lieferanten (Typ **char**).

Nun wird ein zweiter Typ namens `Supplier` definiert, der zu jedem Lieferantencode den zugehörigen Namen kennt:

```
class Supplier {
   public char Id { get; set; }
   public string Name { get; set; }
   public Supplier(char id, string name) {
      Id = id; Name = name;
   }
}
```

Eine Sequenz bzw. Kollektion namens `supps` vom Typ **List**<`Supplier`> wird als Master-Informationsquelle betrachtet:

```
var supps = new List<Supplier> { new('A', "Your Source"),
                                 new('B', "Professional Parts"),
                                 new('C', "Buy, buy") };
```

Durch den folgenden, an die Details-Sequenz (mit den durch Master-Daten zu erweiternden Elementen) gerichteten Aufruf der Erweiterungsmethode **Join()** wird eine Sequenz namens `prodsAugm` erstellt, die zu jedem Produkt den Lieferantencode, den Lieferentennamen und die Qualitätseinschätzung enthält:

```
var prodsAugm = prods.Join(supps, p => p.Supplier, s => s.Id,
                           (p, s) => new { p.Supplier, s.Name, p.Quality });
```

Als Datentyp der kombinierten Sequenz wird eine anonyme Klasse verwendet.

Die Elemente der Ergebnissequenz werden von der folgenden **foreach**-Schleife

```
foreach (var p in prodsAugm)
   Console.WriteLine($"Lieferant: {p.Supplier} {p.Name,-20} Qualität: {p.Quality}");
```

ausgegeben:

```
Lieferant: A Your Source          Qualität: 1
Lieferant: A Your Source          Qualität: 2
 . . .
Lieferant: C Buy, buy             Qualität: 8
Lieferant: C Buy, buy             Qualität: 5
```

19.1.3.6 SelectMany()

Ein Inner Join - Ergebnis (eine Sequenz auf Details-Ebene mit Elementen, die durch Master-Daten ergänzt sind) ist auch mit dem LINQ-Operator **SelectMany()** zu erzielen:

public static IEnumerable<TResult> SelectMany<TSource, TResult>(
this IEnumerable<TSource> *source*,
Func<TSource, IEnumerable<TResult>> *selector*)

Der im Vergleich zu **Join()** einfachere Methodenkopf verlangt zur Lösung der Inner Join - Aufgabe mehr Kreativität und bietet dabei mehr Flexibilität. Die Quellsequenz vom Typ **IEnumerable<TSource>** erhält einen Aufruf der Erweiterungsmethode **SelectMany()**, wobei die an **SelectMany()** per Parameterobjekt übergebene Hilfsmethode mit dem Delegatentyp **Func<TSource, IEnumerable<TResult>>** die Hauptarbeit verrichtet. Sie produziert für jedes **TSource**-Element der Quelle eine Sequenz vom Typ **IEnumerable<TResult>**.

Im folgenden Beispiel

```
var prodsAugmSM = supps.SelectMany(
                          s => prods
                                .Where(p => p.Supplier == s.Id)
                                .Select(p => new { p.Supplier, s.Name, p.Quality }));
```

beauftragt die an **SelectMany()** übergebene, in Lambda-Notation realisierte Hilfsmethode zur Produktion der Teilsequenz zu einem **TSource**-Element die Sequenz `prods`, nacheinander die Erweiterungsmethoden **Where()** und **Select()** auszuführen. Die an **Where()** übergebene Hilfsmethode wählt die Produkte eines Lieferanten aus, und die in **Select()** injizierte Hilfsmethode erstellt für diese Produkte jeweils ein Objekt einer anonymen Klasse, das zwei `prods`-Eigenschaften und eine `supps`-Eigenschaft enthält. Von Interesse ist, dass die inneren Hilfsmethoden auf den Parameter der äußeren Hilfsmethode (also auf das **TSource**-Element) zugreifen können. Aus den Teilsequenzen zu den einzelnen **TSource**-Elementen erstellt **SelectMany()** die Gesamtsequenz `prodsAugmSM`, die identisch ist mit der im Abschnitt 19.1.3.5 erstellten Ergebnissequenz `prodsAugm`.

In einem weiteren **SelectMany()** - Beispiel soll das *kartesische Produkt* bzw. der *Cross Join* aus zwei Sequenzen erstellt werden.[1] Dabei wird jedes Element der ersten Sequenz mit jedem Element der zweiten Sequenz kombiniert. Wir stellen uns in einer Variation des Beispiels vor, dass alle Lieferanten in der Kollektion `supps` an allen Orten in der Kollektion `places` eine Niederlassung besitzen.

```
var places = new List<string> { "York Town", "Tock Village", "Bingten" };
```

Wir richten einen **SelectMany()** - Aufruf an die Sequenz `supps` und übergeben als Parameter eine Hilfsmethode, die zu jedem Lieferanten eine Sequenz bestehend aus den Kombinationen dieses Lieferanten mit allen Orten liefert. Zur Produktion dieser Sequenz richtet die Hilfsmethode einen Aufruf der Erweiterungsmethode **Select()** an die Sequenz `places`, und die an **Select()** übergebene Hilfsmethode liefert für jeden Ort ein Objekt aus einer anonymen Klasse mit einer `places`- und einer `supps`-Eigenschaft:

```
var placesWithSupps = supps.SelectMany(s => places.Select(
                          p => new {Place = p , Supplier = s.Name }));
foreach (var p in placesWithSupps)
   Console.WriteLine($"Lieferant: {p.Supplier,-20} in: {p.Place}");
```

Im Beispiel resultiert die folgende Ausgabe:

---

[1] Diese Terminologie stammt aus der Theorie der relationalen Datenbanken, wobei vom *Cross Join* zu zwei Tabellen gesprochen wird.

```
Lieferant: Your Source          in: York Town
Lieferant: Your Source          in: Tock Village
Lieferant: Your Source          in: Bingten
Lieferant: Professional Parts   in: York Town
Lieferant: Professional Parts   in: Tock Village
Lieferant: Professional Parts   in: Bingten
Lieferant: Buy, buy             in: York Town
Lieferant: Buy, buy             in: Tock Village
Lieferant: Buy, buy             in: Bingten
```

Mit den (unveränderlichen!) Objekten der anonymen Klasse kann man nicht sehr viel anfangen. Wie im Abschnitt 19.1.3.3 erläutert wurde, ist der Projektionsoperator **Select()** aber keinesfalls auf Ergebnissequenzen mit einer anonymen Klasse als Elementtyp eingeschränkt.

19.1.3.7 Weitere Operatoren

Neben **Where()**, **OrderBy()**, **Select()**, **GroupBy()**, **Join()** und **SelectMany()** sind in der Klasse **Enumerable** noch viele andere Erweiterungsmethoden vorhanden, von denen nun einige vorgestellt werden. Wie in vergleichbaren Fällen will und kann das Manuskript keine vollständige API-Beschreibung liefern, d. h.:

- Bei den meisten Methoden werden nicht alle Überladungen vorgestellt.
- Die Reaktionen der Methoden auf spezielle Arbeitsbedingungen (z. B. Berechnung des Mittelwerts zu einer Eingabesequenz ohne Elemente) werden nicht vollständig beschrieben.
- Ebenso werden die von einer Methode zu erwartenden Ausnahmen nur in besonders relevanten Fällen erwähnt.

Die BCL-Dokumentation von Microsoft informiert zuverlässig über die Erweiterungsmethoden.

19.1.3.7.1 Extraktion eines Start- oder Endsegments

In diesem Abschnitt werden Erweiterungsmethoden beschrieben, die von der Anordnung der Elemente in der Eingabesequenz abhängig sind:

- **Take()**, **TakeLast()**
  Von den ersten bzw. letzten Elementen der Eingabesequenz landet eine gewünschte Anzahl in der Ausgabesequenz:

  public static IEnumerable<TSource> Take<TSource>(this
  IEnumerable<TSource> *source*, **int** *count*)
  public static IEnumerable<TSource> TakeLast<TSource>(this
  IEnumerable<TSource> *source*, **int** *count*)

  Im folgenden Quellcodesegment wird die Methode **Take()** zusammen mit einer gleich noch mehrfach verwendeten **int[]** - Eingabesequenz vorgeführt:

Quellcodesegment	Ausgabe

`int[] ints = { 1, 3, 5, 7, 9, 12 };`
`var take3 = ints.Take(3);`
`foreach (var e in take3) Console.Write(e + " ");`
`var takeLast3 = ints.TakeLast(3);`
`foreach (var e in takeLast3) Console.Write(e + " ");`
	`1 3 5`
	`7 9 12`

- **Skip()**
  Diese Methode liefert den Rest der Eingabesequenz ohne die gewünschte Anzahl von übersprungenen Startelementen:
  public static IEnumerable<TSource> Skip<TSource>(this
  IEnumerable<TSource> *source***, int** *count***)**
  Beispiel:

Quellcodesegment	Ausgabe
`var rest = ints.Skip(2);` `foreach (var e in rest) Console.Write(e + " ");`	`5 7 9 12`

19.1.3.7.2 Elementextraktoren

Mit den im aktuellen Abschnitt beschriebenen Methoden soll ein einzelnes Element aus der Eingabesequenz extrahiert werden:

- **First()**, **Last()**, **ElementAt()**
  Diese Methoden liefern das erste bzw. letzte Element der Eingabesequenz oder das Element an einer bestimmten (0-basierten) Position:
  public static TSource First<TSource>(this IEnumerable<TSource> *source***)**
  public static TSource Last<TSource>(this IEnumerable<TSource> *source***)**
  public static TSource ElementAt<TSource>(
  this IEnumerable<TSource> *source***, int** *pos***)**
  Beispiele:

Quellcodesegment	Ausgabe

`int[] ints = { 1, 3, 5, 7, 9, 12 };`
`var firstElem = ints.First(); var lastElem = ints.Last();`
`var secondElem = ints.ElementAt(1);`
`Console.Write(firstElem + ", " + secondElem + ", " + lastElem);`
	`1, 3, 12`

  Ist die Eingabesequenz leer, dann werfen die Methoden **First()** und **Last()** eine Ausnahme. Die Methode **ElementAt()** wirft eine Ausnahme, wenn kein Element mit der angeforderten Position existiert.

- Die Methode **Single()** liefert das einzige Element bzw. das einzige Element, das eine Bedingung erfüllt:
  public static TSource Single<TSource>(this IEnumerable<TSource> *source***)**
  public static TSource Single<TSource>(this IEnumerable<TSource> *source***,**
  Func<TSource, bool> *predicate***)**
  Beispiel:

Quellcodesegment	Ausgabe
`Console.WriteLine(ints.Where(i => i > 10).Single());` `Console.WriteLine(ints.Single(i => i < 3));`	`12` `1`

  Wenn ...
  - entweder *kein* Element vorhanden ist, oder *mehrere* Elemente vorhanden sind bzw.
  - die Bedingung entweder von *keinem* Element oder aber von *mehreren* Elementen erfüllt wird,

  dann wirft **Single()** eine Ausnahme.

Die im aktuellen Abschnitt beschriebenen Methoden haben die folgenden Gemeinsamkeiten:

- Man erhält bei erfolgreicher Ausführung eine Rückgabe vom Typ **TSource**.
- Die LINQ-Operation wird *sofort* ausgeführt.
- Ist das gewünschte Element nicht vorhanden, dann wird eine Ausnahme geworfen.

Zu den Elementextraktionsmethoden existieren Überladungen, die bei Misserfolg einen typspezifischen Nullwert (siehe Abschnitt 8.6) liefern, statt eine Ausnahme zu werfen:

- **FirstOrDefault()**, **LastOrDefault()**, **ElementAtOrDefault()**
- **SingleOrDefault()**
  Während der typspezifische Nullwert geliefert wird, wenn *kein* Element vorhanden ist, führt die Existenz von *mehreren* Elementen zu einem Ausnahmefehler. Analog zu **Single()** sind zwei Überladungen vorhanden (ohne bzw. mit Prädikat).

19.1.3.7.3 Auswertungsmethoden

Die in diesem Abschnitt vorgestellten Auswertungsmethoden erinnern an analoge Funktionen der Abfragesprache SQL (vgl. Abschnitt 18.5.2.5):

- **Count()**
  Je nach Überladung erhält man als Rückgabe vom Typ **int** ...
  - die Anzahl der Elemente in der Eingabesequenz
  - die Anzahl der Elemente in der Eingabesequenz, die eine bestimmte Bedingung erfüllen.

  public static int Count(this IEnumerable<int> *source***)**
  public static int Count(this IEnumerable<int> *source***,**
  Func<TSource, Boolean> *predicate***)**

  Beispiel:

Quellcodesegment	Ausgabe
`int[] ints = { 1, 3, 5, 7, 9, 12 };` `Console.Write(ints.Count());`	`6`

  Wenn bei der Rückgabe ein **int**-Überlauf zu befürchten ist, dann verwendet man die Methode **LongCount()**.
- **Sum()**, **Average()**
  Diese in zahlreichen Überladungen für unterschiedliche numerische Elementtypen in der Eingabesequenz vorhandenen Methoden berechnen die Summe bzw. den Mittelwert der Elemente in der Eingabesequenz, z. B. für eine Sequenz vom konkretisierten generischen Typ **IEnumerable<int>**:

  public static int Sum (this IEnumerable<int> *source***)**

  Beispiel:

Quellcodesegment	Ausgabe
`Console.Write(ints.Sum()+ ", " + ints.Average());`	`37, 6,166666666666667`

- **Min()**, **Max()**
  Auch die Methoden zur Bestimmung des minimalen bzw. maximalen Elementwerts in der Eingabesequenz sind in zahlreichen Überladungen vorhanden, z. B. für eine Sequenz vom konkretisierten generischen Typ **IEnumerable<int>**:

  public static int Min (this IEnumerable<int> *source***)**
  public static int Max (this IEnumerable<int> *source***)**

  Beispiel:

Quellcodesegment	Ausgabe
`Console.Write(ints.Min() + ", " + ints.Max());`	`1, 12`

19.1.3.7.4 Existenz und Bedingungen prüfen

Im aktuellen Abschnitt befinden sich Erweiterungsmethoden, welche die Existenz von Elementen oder das Vorliegen von Bedingungen prüfen. Diese Methoden liefern keine Ausgabesequenz und auch keine Instanz mit dem Elementtyp der Eingabesequenz, sondern eine Rückgabe vom Typ **bool**. Sie werden außerdem sofort ausgeführt.

- **Contains()**
  Es wird überprüft, ob ein bestimmtes Element in der Eingabesequenz vorhanden ist:
  public static bool Contains<TSource>(this IEnumerable<TSource> *source***,**
  TSource *value***)**
  Beispiel:

Quellcodesegment	Ausgabe
`int[] ints = { 1, 3, 5, 7, 9, 12 };` `Console.Write(ints.Contains(5));`	`True`

  Statt bei der Identitätserkennung auf die **Equals()** - Methode des Elementtyps zu setzen, kann man eine **Contains()** - Überladung verwenden, die als zusätzlichen Parameter ein die Schnittstelle **IEqualityComparer<TSource>** implementierendes Objekt akzeptiert.

- **All()**, **Any()**
  Es wird überprüft, ob eine Bedingung von *allen* oder von *irgendeinem* Element der Eingabesequenz erfüllt wird:
  public static bool All<TSource>(this IEnumerable<TSource> *source***,**
  Func<TSource, bool> *predicate***)**
  public static bool Any<TSource>(this IEnumerable<TSource> *source***,**
  Func<TSource, bool> *predicate***)**
  Beispiel:

Quellcodesegment	Ausgabe
`Console.Write(ints.All(n => n % 2 == 0) + ", " +` `ints.Any(n => n % 2 == 0));`	`False, True`

  Die Erweiterungsmethode **All()** liefert die Rückgabe **true**, wenn *kein* Element existiert, das die Bedingung *nicht* erfüllt. Das ist auch bei einer *leeren* Sequenz der Fall, z. B.:

Quellcodesegment	Ausgabe

`var empty = ints.Where(i => i < -1);`
`Console.WriteLine("\nAll() in empty sequence:\n" +`
`empty.All(i => i > 77));`
	`All() in empty sequence:`
	`True`

  In dieser Lage rechnet nicht unbedingt jeder Programmierer mit der Rückgabe **true**.

19.1.3.7.5 Mengentheoretische Operationen

Es folgen Erweiterungsmethoden zur Realisation von mengentheoretischen Operationen:

- **Union()**
  Zwei Eingabesequenzen werden im mengentheoretischen Sinn vereinigt:
  public static IEnumerable<TSource> Union<TSource>(
  this IEnumerable<TSource> *first***, IEnumerable<TSource>** *second***)**

Beispiel:

Quellcodesegment	Ausgabe

`int[] first = { 1, 2, 3 };`
`int[] second = { 2, 3, 4 };`
`result = first.Union(second);`
`foreach (var e in result) Console.Write(e + " ");`
	`1 2 3 4`

- **Intersect()**
  Zwei Eingabesequenzen werden im mengentheoretischen Sinn geschnitten:
  public static IEnumerable<TSource> Intersect<TSource>(
  this IEnumerable<TSource> *first*, **IEnumerable<TSource>** *second*)
  Beispiel:

Quellcodesegment	Ausgabe
`result = first.Intersect(second);` `foreach (var e in result) Console.Write(e + " ");`	`2 3`

- **Except()**
  Es wird die Differenz von zwei Sequenzen gebildet, d. h. aus der ersten Sequenz werden alle Elemente entnommen, die sich auch in der zweiten Sequenz befinden:
  public static IEnumerable<TSource> Except<TSource>(
  this IEnumerable<TSource> *first*, **IEnumerable<TSource>** *second*)
  Beispiel:

Quellcodesegment	Ausgabe
`result = first.Except(second);` `foreach (var e in result) Console.Write(e + " ");`	`1`

- **Distinct()**
  Bei Objekten vom Typ **IEnumerable<TSource>** ist i. A. die für Mengenverwaltungskollektionen typische Abwesenheit von Dubletten *nicht* garantiert (vgl. Abschnitt 11.4). Die Erweiterungsmethode **Distinct()** liefert zu einer Eingabesequenz eine von Dubletten befreite Ausgabesequenz:
  public static IEnumerable<TSource> Distinct<TSource>(
  this IEnumerable<TSource> *source*)
  Beispiel:

Quellcodesegment	Ausgabe

`int[] intsd = { 1, 3, 3, 7, 7, 12 };`
`result = intsd.Distinct();`
`foreach (var e in result) Console.Write(e + " ");`
	`1 3 7 12`

19.1.3.7.6 Sequenzen verketten

Die Methode **Concat()** verkettet zwei Sequenzen:

public static IEnumerable<TSource> Concat<TSource>(
this IEnumerable<TSource> *first*, **IEnumerable<TSource>** *second*)

Beispiel:

Quellcodesegment	Ausgabe

`int[] first = { 1, 2, 3 };`
`int[] second = { 2, 3, 4 };`
`var result = first.Concat(second);`
`foreach (var e in result) Console.Write(e + " ");`
	`1 2 3 2 3 4`

19.1.3.7.7 Reihenfolge der Elemente umkehren

Die Ausgabesequenz der Methode **Reverse()** enthält die Elemente der Eingabesequenz in umgekehrter Reihenfolge:

public static IEnumerable<TSource> Reverse<TSource>(this IEnumerable<TSource> *source***)**

Beispiel:

Quellcodesegment	Ausgabe

`int[] ints = { 1, 3, 5, 7, 9, 12 };`
`var umgekehrt = ints.Reverse();`
`foreach (var e in umgekehrt) Console.Write(e + " ");`
	`12 9 7 5 3 1`

19.2 Abfrageausdrücke

Alternativ zur Erweiterungsmethodensyntax kann für LINQ-to-Objects - Abfragen oft eine SQL-ähnliche Syntax verwendet werden, die auf den ersten Blick angenehm wirkt, durch neue Schlüsselwörter und Syntaxregeln aber auch Lernaufwand erfordert, z. B.:

```
var compNames = from c in customers
                where c.Country == "Germany"
                orderby c.CompanyName.Length
                select c.CompanyName.ToUpper();
```

Dieser Abfrageausdruck führt exakt zum selben Ergebnis wie die folgende, im Abschnitt 19.1.1.2 über das Fluent-API verwendete Serienschaltung von Erweiterungsmethoden:

```
var compNames = customers
                .Where(c => c.Country == "Germany")
                .OrderBy(c => c.CompanyName.Length)
                .Select(c => c.CompanyName.ToUpper());
```

Der Compiler übersetzt Abfrageausdrücke generell in das Fluent-API und verwendet dabei die Lambda-Notation. Ein Abfrageausdruck kann also immer in die Erweiterungsmethodensyntax übersetzt werden. Umgekehrt lassen sich aber manche LINQ-Erweiterungsmethoden(überladungen) *nicht* per Abfragesyntax realisieren. Es ist also möglich, in eigenem Code auf Abfrageausdrücke zu verzichten. Man sollte sich nicht für eine Syntaxvariante entscheiden, sondern flexibel die im konkreten Anwendungsfall einfachere Variante wählen.

19.2.1 Regeln

Ein Abfrageausdruck muss eine **select**- oder **group**-Klausel enthalten, sodass der Compiler die folgende Anweisung reklamiert:

```
var compNames = from c in customers
                where c.Country == "Germany"
                orderby c.CompanyName.Length;
```

CS0742: Auf einen Abfragetext muss eine Select-Klausel oder Group-Klausel folgen.

Die analoge Erweiterungsmethodensyntax wird hingegen akzeptiert und liefert eine verwertbare Sequenz:

```
var compNames = customers
                .Where(c => c.Country == "Germany")
                .OrderBy(c => c.CompanyName.Length);
```

Um die Kritik des Compilers am Abfrageausdruck zu vermeiden, genügt eine Projektion mit identischer Abbildung:

```
var compNames = from c in customers
                where c.Country == "Germany"
                orderby c.CompanyName.Length
                select c;
```

Bei der Übersetzung dieses Abfrageausdrucks in die Erweiterungsmethodensyntax wird die **select**-Klausel schlicht ignoriert.

Ein Abfrageausdruck ist mit der obligatorischen **select**- oder **group**-Klausel beendet, sofern keine **into**-Klausel die Fortführung mit einer neuen Abfrage einleitet (siehe Abschnitt 19.2.9).

Man darf die Erweiterungsmethodensyntax mit der Abfragesyntax mixen, sofern die beteiligten Abfrageausdrücke vollständig sind, z. B.:

```
int[] ints1 = { 1, 3, 4, 8, 9, 10 };
int[] ints2 = { 10, 14, 18 };
var result = (from i in ints1 where i % 2 == 0 select i).Union(ints2);
```

19.2.2 from-in

Ein LINQ-Abfrageausdruck beginnt mit einer **from-in** - Klausel mit einer Angabe der Datenquelle und unterscheidet sich damit auf den ersten Blick von einer **SELECT**-Abfrage in SQL (vgl. Abschnitt 18.5). Indem zunächst die Datenquelle ausgewählt wird, von der die nachfolgenden Operationen ausgehen, kann die Entwicklungsumgebung mit ihrer IntelliSense-Technik das Erstellen von Abfragesyntax ebenso gut unterstützen wie das Erstellen von Erweiterungsmethodensyntax.

Im folgenden Beispiel ist die Datenquelle ein Array mit dem Elementtyp `Customer`:

```
Customer[] customers = new Customer[7];
 . . .
var custIds = from c in customers select c.CustomerId;
```

Die nach dem Schlüsselwort **in** angegebene Kollektion muss die Schnittstelle **IEnumerable<TSource>** implementieren, und die hinter dem Schlüsselwort **from** genannte Variable hat implizit den Typ **TSource**. Im Beispiel hat die Datenquelle `customers` den Array-Typ `Customer[]`, und `c` ist folglich vom Typ `Customer`.

Analog zur Iterationsvariablen einer **foreach**-Schleife (vgl. Abschnitt 4.7.3.2) repräsentiert `c` sukzessiv als sogenannte *Bereichsvariable* die Elemente der Eingabesequenz. Die Gültigkeit der Bereichsvariablen ist auf den Abfrageausdruck beschränkt und wird ggf. durch eine **into**-Klausel im Ausdruck beendet (siehe Abschnitt 19.2.9). In der Erweiterungsmethodensyntax, die der Compiler aus der Abfragensyntax erstellt, wird die Bereichsvariable zum Parameter einer per Lambda-Notation realisierten Hilfsmethode, z. B.:

```
IEnumerable<string> custIds = customers.Select(c => c.CustomerId);
```

Wenn eine Kollektion nur die nicht-generische Schnittstelle **IEnumerable** implementiert, dann muss vor der Bereichsvariablendeklaration der Elementtyp explizit angegeben werden, z. B.:

```
ArrayList al = new ArrayList();
al.Add("Eins"); al.Add(13);
var alab = from object c in al select c;
```

Bei der Erweiterungsmethodensyntax wird keine **from-in** - Entsprechung benötigt. Stattdessen wird an die Eingabesequenz vom Typ **IEnumerable<TSource>** der erste Methodenaufruf gerichtet, z. B.:

```
var custIds = customers.Select(c => c.CustomerId);
```

19.2.3 select

Im folgenden Beispiel

```
Customer[] customers = new Customer[7];
 . . .
var custIds = from c in customers select c.CustomerId;
```

findet eine Abbildung von einer Eingabesequenz auf eine Ausgabesequenz statt, die als Projektion (im Sinn von Abschnitt 19.1.3.3) durch eine **select**-Klausel realisiert wird.

Die **select**-Klausel erledigt offenbar den Job der Erweiterungsmethode **Select()**. Allerdings kann von den beiden **Select()** - Überladungen nur die einfachere durch eine **select**-Klausel ersetzt werden:

public static IEnumerable<TResult> Select<TSource, TResult>(
this IEnumerable<TSource> *source*, **Func<TSource, TResult>** *selector*)

Die alternative **Select()** - Überladung verlangt als zweiten Parameter eine Methode mit *zwei* Argumenten und einer **TResult**-Rückgabe:

public static IEnumerable<TResult> Select<TSource, TResult>(
this IEnumerable<TSource> *source*, **Func<TSource, int, TResult>** *selector*)

Als zweites Argument wird die Indexposition des Quellelements übergeben, die somit Einfluss auf die Produktion nehmen kann. Diese vermutlich seltene Konstellation macht die Verwendung der Erweiterungsmethodensyntax erforderlich.

19.2.4 where

Die **where**-Klausel übernimmt offenbar in einem Abfrageausdruck die Rolle der Erweiterungsmethode **Where()**. Allerdings kann von den beiden **Where()** - Überladungen nur die einfachere durch eine **where**-Klausel ersetzt werden:

public static IEnumerable<TSource> Where<TSource>(
this IEnumerable<TSource> *source*, **Func<TSource, bool>** *predicate*)

Das Prädikat im zweiten Parameter beurteilt das aktuelle Element.

Die alternative **Where()** - Überladung verlangt als zweiten Parameter eine Methode mit *zwei* Argumenten und einer **bool**-Rückgabe:

public static IEnumerable<TSource> Where<TSource>(
this IEnumerable<TSource> *source*, **Func<TSource, int, bool>** *predicate*)

Semantisch wird eine Methode erwartet, die bei der Bewertung eines Elements auch seine Indexposition in der Quelle berücksichtigt, sodass z. B. nur Elemente passieren (in die Ausgabesequenz gelangen), die eine geradzahlige Indexposition besitzen und außerdem weitere Bedingungen erfüllen. Wird diese Überladung in einer Abfrage benötigt, dann kommt man um die Erweiterungsmethodensyntax nicht herum. Allzu oft wird das allerdings nicht passieren.

19.2.5 oderby

In einem Abfrageausdruck lässt sich die Anordnung der Elemente in der Ausgabesequenz über eine **orderby**-Klausel steuern, die der Compiler in einen Aufruf der Erweiterungsmethode **OrderBy()** übersetzt. Ist eine absteigende Anordnung erwünscht, dann erweitert man die **orderby**-Klausel um das Schlüsselwert **descending**, sodass ein Aufruf der Erweiterungsmethode **OrderByDescending()** resultiert, z. B.:

```
var cust = from c in customers
           where c.Country == "Germany"
           orderby c.CompanyName descending
           select c;
```

Von den beiden Überladungen der Erweiterungsmethoden **OrderBy()** bzw. **OrderByDescending()** ist jeweils nur *eine* per **orderby**-Klausel nutzbar. Wer die kanonische Sortierung des Elementtyps **TSource** durch eine **IComparer<TSource>** - Implementierung ersetzen möchte, der muss die Erweiterungsmethodensyntax verwenden.

Ist ein hierarchisches Sortieren nach mehreren Kriterien erforderlich, dann werden in *einer* **orderby**-Klausel die Sortierkriterien hintereinander angegeben:

- durch Kommata getrennt
- und bei Bedarf durch das Schüsselwort **descending** ergänzt.

Das folgende Beispiel wurde im Wesentlichen schon im Abschnitt 19.1.3.2 zur Erläuterung der korrespondierenden Erweiterungsmethodensyntax verwendet:

Quellcodesegment	Ausgabe

`class Qtype {`
`  public char Category;`
`  public int Level;`
`  public Qtype(char cat, int lev) {`
`  Category = cat; Level = lev;`
`  }`
`}`
`. . .`
`var prods = new Qtype[5] { new('A', 3), new('B', 1), new('B', 2),`
`  new('C', 2), new('C', 1) };`
`var sortedProds = from p in prods`
`  orderby p.Category, p.Level descending`
`  select p;`
`foreach (var p in sortedProds)`
`  Console.WriteLine(p.Category + " " + p.Level);`
	`A 3`
	`B 2`
	`B 1`
	`C 2`
	`C 1`

19.2.6 group-by

Statt mit der Erweiterungsmethode **GroupBy()** kann man auch durch einen Abfrageausdruck mit **group-by** - Klausel eine Ausgabesequenz erzeugen, deren Elemente aus gruppenspezifischen Teilsequenzen bestehen. Im Beispiel aus dem Abschnitt 19.1.3.4 sind lediglich die Anweisungen mit einem **GroupBy()** - Aufruf zu ersetzen:

- Zur Gruppierung mit Hilfe der Elementeigenschaft `Supplier` ersetzt man die Erweiterungsmethodensyntax
  ```
  var groups = prods.GroupBy(p => p.Supplier);
  ```
  durch den Abfrageausdruck:
  ```
  var groups = from prod in prods group prod by prod.Supplier;
  ```
- Zur Gruppierung über einen logischen Ausdruck ersetzt man die Erweiterungsmethodensyntax
  ```
  var groups = prods.GroupBy(p => p.Supplier <= 'B');
  ```
  durch den Abfrageausdruck:
  ```
  var groups = from prod in prods group prod by prod.Supplier <= 'B';
  ```

- Zur Gruppierung über Objekte einer anonymen Klasse ersetzt man die Erweiterungsmethodensyntax

```
var groups = prods.GroupBy(p => new { p.Supplier, HighQual = p.Quality > 5 });
```

durch den Abfrageausdruck:

```
var groups = from prod in prods
             group prod by new {prod.Supplier, HighQual = prod.Quality > 5 };
```

Einige **GroupBy()** - Überladungen sind *nicht* per Abfrageausdruck darstellbar.

19.2.7 join

Statt mit der Erweiterungsmethode **Join()** kann man auch durch einen Abfrageausdruck mit **join-on-equals** - Klausel aus zwei Eingabesequenzen eine Ausgabesequenz nach dem Inner Join - Prinzip erstellen (vgl. Abschnitt 19.1.3.5):

```
var prodsAugm = from p in prods
                join s in supps
                on p.Supplier equals s.Id
                select new { p.Supplier, s.Name, p.Quality };
```

19.2.8 Zwei from-Klauseln (SelectMany() per Abfrageausdruck)

Zur Realisation des Erweiterungsmethode **SelectMany()** per Abfrageausdruck wird für die beiden Quellen (Eingabesequenzen) jeweils eine **from**-Klausel mit einer eigenen Bereichsvariablen verwendet. Um die im Abschnitt 19.1.3.6 vorgestellte **SelectMany()** - Lösung der Inner Join - Aufgabe als Abfrageausdruck zu formulieren, werden zusätzlich zu den beiden **from**-Klauseln noch eine **where**- und eine **select**-Klausel benötigt:

```
var prodsAugmSMA = from   s in supps
                   from   p in prods
                   where  s.Id == p.Supplier
                   select new { p.Supplier, s.Name, p.Quality };
```

Die im Abschnitt 19.1.3.5 vorgestellte **SelectMany()** - Lösung der Cross Join - Aufgabe ist als Abfrageausdruck elegant zu formulieren:

```
var placesWithSuppsA = from s in supps
                       from p in places
                       select new { Place = p, Supplier = s.Name };
```

19.2.9 Sequenzen von Abfragen (into)

Mit Hilfe des **into**-Schlüsselworts kann ein Abfrageausdruck nach einer **select**- oder **group**-Klausel fortgesetzt werden. Man startet vom Zwischenergebnis ausgehend eine neue Abfrage, z. B:

```
var aNames = from c in customers
             select c.CompanyName.ToUpper()
             into compName
             where compName.StartsWith("A")
             orderby compName.Length
             select compName;
```

Dieselbe Sequenz von Abfragen erzielt man auch durch eine Verschachtelung, z. B.:

```
var aNames = from compName in (
                  from c in customers
                  select c.CompanyName.ToUpper()
                )
                where compName.StartsWith("A")
                orderby compName.Length
                select compName;
```

Das **into**-Schlüsselwort aus dem vorherigen Beispiel ist durch ein zweites **from**-Schlüsselwort ersetzt worden. Beide Formulierungen werden vom Compiler in dieselbe Erweiterungsmethodensyntax übersetzt.

19.2.10 Mehrere Bereichsvariablen (let)

Mit dem Schlüsselwort **let** lassen sich in Abfrageausdrücken zusätzliche Bereichsvariablen deklarieren, die als Funktionen der primären Bereichsvariablen aufgefasst werden können. Diese zusätzlichen Bereichsvariablen lassen sich in **where**-, **orderby**- und **select**-Klauseln (wiederholt) verwenden, was eine Vereinfachung der Syntax ermöglicht, z. B.:

```
var result = from c in customers
                let len = c.CompanyName.Length
                where len < 20
                orderby len
                select new { c.CompanyName, len };
```

In einer **let**-Deklaration dürfen vorhandene **let**-Variablen verwendet werden, z. B.:

```
let len2 = len*len
```

Eine **into**-Klausel beendet den Gültigkeitsbereich der vorher deklarierten **let**-Variablen.

Die Möglichkeit zur Verwendung von *mehreren* Bereichsvariablen per **let**-Klausel ist ein Grund dafür, dass manche Aufgaben mit einem Abfrageausdruck leichter zu lösen sind als mit der Erweiterungsmethodensyntax (Albahari & Johannsen 2020, S. 381).

19.3 Unterabfragen

Eine als Parameter einer **Enumerable**-Erweiterungsmethode verwendete Delegatenmethode (z. B. in Lambda-Notation) darf selbstverständlich die LINQ-Abfragetechnik verwenden, sodass eine Unterabfrage entsteht. Das gilt natürlich auch bei Verwendung der Abfragesyntax.

Unterabfragen (engl.: *subqueries*) werden vor allem in folgenden Situationen benötigt:

- Per **GroupBy()** (oder **group-by**) ist eine gruppierte Sequenz entstanden, und es soll eine Unterabfrage für jede einzelne Gruppe durchgeführt werden. Für diese Situation wird gleich ein Beispiel vorgestellt.
- Der Elementtyp einer Sequenz enthält ein Memberobjekt mit Kollektionstyp.
- In die Erweiterungsmethode **SelectMany()** wird eine Hilfsmethode injiziert, die ein **IEnumerable<TResult>** - Ergebnis (also eine Ausgabesequenz) zu liefern hat, wobei sich oft die Anwendung einer Unterabfrage auf eine Eingabesequenz anbietet (siehe Abschnitt 19.1.3.6).

Der im Abschnitt 19.1.3.4 beschriebene LINQ-Operator **GroupBy()** bzw. seine Abfragesyntaxentsprechung **group-by** erstellt aus der Eingabesequenz eine Ausgabesequenz bestehend aus gruppenspezifischen Teilsequenzen. Im Beispiel von Abschnitt 19.1.3.4 wurde die mittlere Qualität der Produkte eines Lieferanten durch eine *nachgeschaltete* LINQ-Operation ermittelt. Nun wird die in jeder einzelnen Gruppe vorzunehmende Qualitätsermittlung im Rahmen einer Projektion als Unterabfrage in die Hauptabfrage integriert:

```
using System;
using System.Collections.Generic;
using System.Linq;

class Product {
   public char Supplier {get; set;}
   public int Quality {get; set;}
   public Product(char supp, int qual) {
      Supplier = supp; Quality = qual;
   }
}

class Unterabfrage {
   static void Main() {
      var prods = new List<Product> { new('A',1), new('A',2), new('A',1), new('A',3),
                        new('B',4), new('B',2), new('B',3), new('B',5), new('B',4),
                        new('C',5), new('C',6), new('C',4), new('C',8), new('C',5) };

      var groups = from prod in prods
                   group prod by prod.Supplier into supp
                   select new { supp.Key, AvgQual =
                                      (from p in supp select p.Quality).Average() };

      foreach (var supp in groups) {
         Console.WriteLine("Lieferant: " + supp.Key + " Qualität: " + supp.AvgQual);
      }
   }
}
```

Das Programm liefert die folgende Ausgabe:

```
Lieferant: A Qualität: 1,75
Lieferant: B Qualität: 3,6
Lieferant: C Qualität: 5,6
```

20 Entity Framework Core

Die Datenorganisation in den Tabellen einer relationalen Datenbank unterscheidet sich deutlich vom hierarchischen Objekt-Graphen einer objektorientierten Software. Zur Überbrückung der Kluft sind etliche ORM-Bibliotheken (*Object Relational Mapping*) entwickelt worden. Microsoft hat eine Lösung unter dem Namen *Entity Framework* beigesteuert, die seit dem Erscheinungsjahr 2008 viele Verbesserungen erfahren hat. Die modernste Entwicklungslinie trägt den Namen *Entity Framework Core* (kurz: *EF Core*) und hat im August 2021 den im Manuskript behandelten Versionsstand 5.0.

Wie die Namenserweiterung um den Zusatz *Core* zeigt, konzentriert sich die Weiterentwicklung von Microsofts ORM-Lösung auf .NET Core bzw. auf dessen Nachfolger .NET 5.0, 6.0 etc. Wer noch das auf Windows beschränkte .NET Framework 4.x verwendet, kann ...

- entweder EF Core bis zur Version 3.1 einsetzen
- oder mit Entity Framework 6.x arbeiten.

Das EF Core benötigt einen erweiterten ADO.NET - Provider, der mittlerweile für viele Datenbankmanagementsysteme verfügbar ist.[1] Für ältere Datenbankmanagementsysteme ist jedoch keine EF Core - Unterstützung vorhanden, z. B. für die Microsoft-Techniken SQL Server Compact und Jet Engine.

Bei der Entscheidung für oder gegen eine ORM-Zugriffsebene bei der Datenbankprogrammierung spielt die Performanz eine wichtige Rolle. Komplette Objekte aus einer Datenbank zu laden, führt oft zum aufwändigen Transport überflüssiger Daten. Wenn dann anonyme Klassen zum Einsatz kommen, um die übertragene Datenmenge auf das notwendige Maß zu beschränken, wird das ORM-Prinzip zumindest teilweise aufgegeben. Allerdings bietet das EF Core auch bei dieser Arbeitsweise noch Vorteile gegenüber einer Beschränkung auf die traditionellen ADO.NET - Klassen.

Bei der traditionellen ADO.NET - Arbeitsweise mit untypisierten DataSets werden Tabellen und Spalten durch Indexer-Argumente vom Zeichenfolgentyp angesprochen, und auf die Werte einer Tabellenzeile greift man über die **DataRow**-Eigenschaft **ItemArray** mit dem Datentyp **Object[]** zu, was lästige und fehleranfällige Typumwandlungen erfordert. Bei der Arbeit mit dem EF Core spricht man typsicher die Eigenschaften von Objekten an, was zu einer höheren Effektivität und einer geringeren Fehlerquote führt.

Die Probleme des Datenzugriffs mit schwacher Typisierung werden auch durch die im Abschnitt 18.7 vorgestellten typisierten DataSets überwunden. Allerdings bleibt bei diesem Ansatz der Paradigmenbruch zwischen der objektorientierten Programmierung einerseits und der Bearbeitung von Tabellenzeilen andererseits bestehen.

Das aktuelle Kapitel basiert im Wesentlichen auf Microsofts Dokumentation zum EF Core, die als PDF-Export stolze 1300 Seiten umfasst.[2]

---

1 https://docs.microsoft.com/de-de/ef/core/providers/

2 https://docs.microsoft.com/en-us/ef/core/

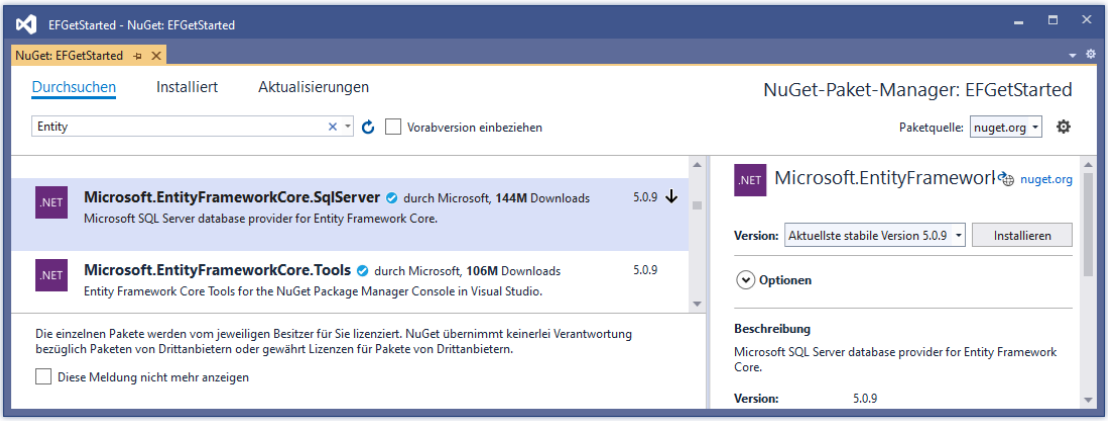
20.1 Erforderliche NuGet-Pakete

Um mit .NET 5.0 und dem Entity Framework Core 5.0 unter Verwendung von Microsofts SQL Server arbeiten zu können, werden über das (z. B. gemäß Abschnitt 3.1 installierte) .NET 5.0 SDK hinaus noch NuGet-Pakete benötigt, die mit einem von den folgenden Verfahren installiert werden können:[1]

- Über das Kommando **dotnet add** in einem Konsolenfenster des Betriebssystems
- Über das Kommando **Install-Package** in der Paket-Manager-Konsole der Entwicklungsumgebung Visual Studio (zu öffnen mit **Extras > NuGet-Paket-Manager > Paket-Manager-Konsole**)
- Über das **NuGet-Paket-Manager**-Fenster der Entwicklungsumgebung, das z. B. über das Item **NuGet-Pakete verwalten** aus dem Kontextmenü zum Knoten **Abhängigkeiten** in einem Projekt für das Zielframework .NET 5.0 geöffnet werden kann

Wir wählen das dritte Verfahren, um die beiden erforderlichen NuGet-Pakete für ein Projekt zu installieren:[2]

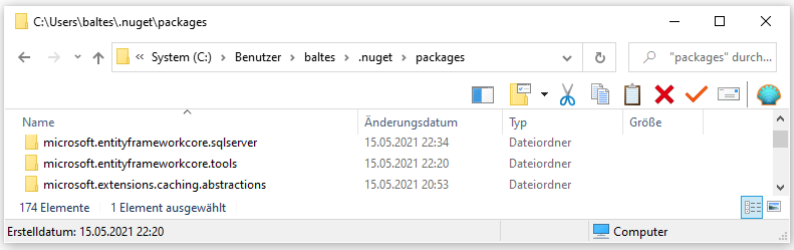
- **Microsoft.EntityFrameworkCore.SqlServer**
  Dieses Paket installiert den EF Core - Provider für Microsofts SQL-Server:



---

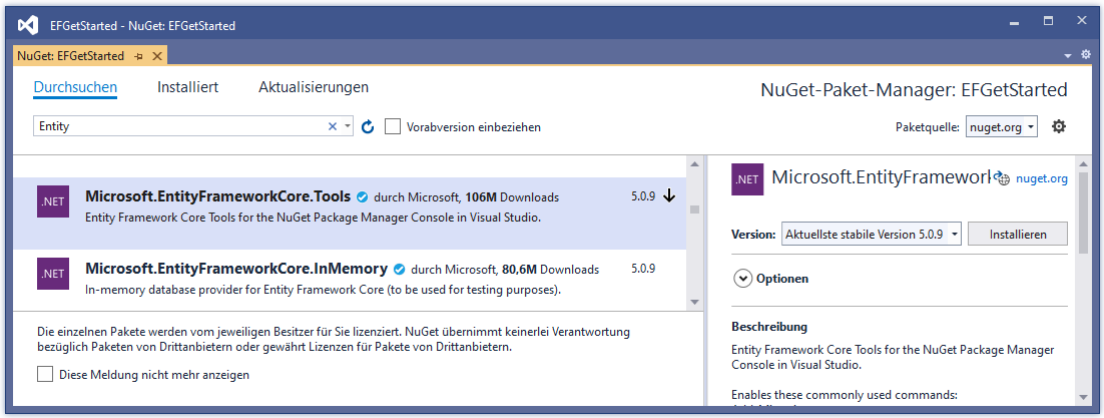
1 https://docs.microsoft.com/en-us/ef/core/get-started/overview/install

2 NuGet-Pakete werden unter Windows im benutzereigenen Ordner **...\.nuget\packages** abgelegt, z. B.:



Die Pakete müssen also nicht für jedes Projekt neu heruntergeladen werden.

- **Microsoft.EntityFrameworkCore.Tools**

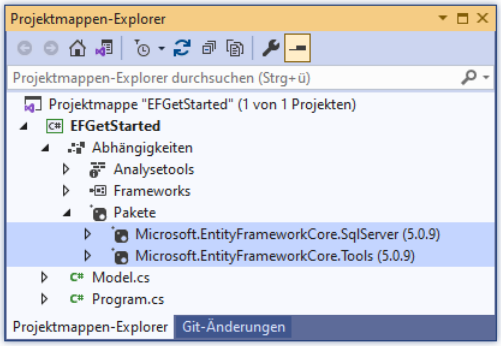


  Die EF Core - Werkzeuge werden u. a. dazu benötigt, ...
  - um aus einem EF Core - Modell eine Datenbank zu erstellen (Migration)
  - um aus einer vorhandenen Datenbank ein EF Core - Modell zu erstellen (Reverse Engineering)

  Nachdem das NuGet-Paket installiert worden ist, sind die Tools in zwei Varianten verfügbar:
  - Als Werkzeuge zur Verwendung in der **Paket-Manager-Konsole** im Visual Studio (engl. Bezeichnung *PMC Tools*)
  - Als Werkzeuge zur Verwendung über das Kommando **dotnet ef** in einem Konsolenfenster des Betriebssystems (Linux, macOS oder Windows). In der engl. Bezeichnung *CLI Tools* steht *CLI* für *Command Line Interface*.

  Wir entscheiden uns für die erste Variante wegen der besseren Integration in das Visual Studio.

So zeigt der **Projektmappen-Explorer** die installierten Pakete an:



20.2 EF Core - Modell

Eine Anwendung mit Datenbank-Persistenzlösung per EF Core besitzt ein Modell mit ...

- Klassen zur Modellierung der sogenannten *Entitäten* aus dem Aufgabenbereich
  Eine Entitätsklasse ist in der Regel mit einer Tabelle einer relationalen Datenbank assoziiert. Im Beispiel von Abschnitt 20.3.1 gehört zum EF Core - Modell u. a. die folgendermaßen definierte Entitätsklasse `Blog`:

```
public class Blog {
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; } = new List<Post>();
}
```

- einer von **DbContext** abgeleiteten Klasse zur ORM-Vermittlung zwischen den Entitäten und der Datenbank
  Ein Objekt der Kontextklasse repräsentiert eine Datenbanksitzung (eine Arbeitseinheit, siehe Abschnitt 20.2.1). Es ist u. a. dafür zuständig, ...
  - Tabellenzeilen der Datenbank abzufragen,
  - daraus Objekte einer Entitätsklasse zu erstellen,
  - Änderungen der Entitätsobjekte zu verfolgen,
  - modifizierte Entitätsobjekte in die Datenbank zu sichern.

  Im Beispiel von Abschnitt 20.3.1 kommt die folgende **DbContext**-Ableitung zum Einsatz:

```
public class BloggingContext : DbContext {
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder options) {
        options.UseSqlServer("Server=tcp:bernhard\\SQLEXPRESS; " +
        "Initial Catalog=EFGetStarted; Integrated Security=true");
    }
}
```

Man kann das EF Core - Modell ...

- im Quellcodeeditor definieren und dann mit PCM-Tools - Kommandos eine dazu passende Datenbank erstellen lassen, wobei die Migrations-Funktionalität im EF Core genutzt wird (**Code-First** - Technik, siehe Abschnitt 20.3), z. B.:

```
Add-Migration InitialCreate
Update-Database
```

  Durch die Beachtung von **Konventionen** erhält man bei minimalem Konfigurationsaufwand ein gutes Datenbankdesign. Z. B. wird eine Eigenschaft mit dem Namen
  - **Id** oder
  - *Klassenname***Id** (z. B. `BlogId`)

  zum Primärschlüssel der zu einer Entitätsklasse gehörenden Tabelle. Aus den öffentlichen Eigenschaften einer Entitätsklasse resultieren Spalten der Tabelle. Mit Hilfe der Migrations-Funktionalität im EF Core kann die Datenbank auch an ein verändertes Modell angepasst werden.
- per Reverse Engineering über das PCM-Tools - Kommando `Scaffold-DbContext` aus einer vorhandenen Datenbank erstellen lassen, z. B.:

```
Scaffold-DbContext "Server=tcp:bernhard\SQLEXPRESS; Initial Catalog=Northwind;
Integrated Security=true" Microsoft.EntityFrameworkCore.SqlServer -OutputDir
Models/Northwind
```

  Bei diesem EF Core - Einsatz nach dem **Database-First** - Prinzip werden die Entitätsklassen sowie die **DbContext**-Ableitung automatisch erstellt (siehe Abschnitt 20.4). Wird ein so initialisiertes Modell später geändert, kann zur Aktualisierung der Datenbank die Migrations-Funktionalität im EF Core verwendet werden.

20.2.1 Die Klasse DbContext

Kommt in einer objektorientierten Anwendung mit einer Datenbank als Persistenzlösung das EF Core als ORM-Lösung zum Einsatz, dann übernimmt eine von **DbContext** abgeleitete Klasse die Vermittlung zwischen den Entitäten und der Datenbank.

20.2.1.1 Aufgaben

Ein Objekt aus der Klasse **DbContext** (genauer: aus einer anwendungsspezifischen **DbContext**-Ableitung) erfüllt u. a. die folgenden Aufgaben:

- Verwaltung der Datenbankverbindung
  In der **DbContext**-Methode **OnConfiguring()** erfährt das Kontextobjekt, welcher EF Core - Provider zu verwenden ist (siehe Beispiel zu Beginn von Abschnitt 20.2).
- Konfiguration des EF Core Modells und des Datenbankschemas
  Sind bei der Ableitung des Datenbankschemas aus den Entitätsklassen Abweichungen von den EF Core - Konventionen erforderlich, dann erlaubt die **DbContext**-Methode **OnModelCreating()** die größte Flexibilität (siehe Abschnitt 20.2.2.1).
- Entitäten aus der Datenbank abfragen und nach einer Änderung wieder dorthin sichern
- Eine wichtige Leistung des **DbContext**-Objekts für die Objekte von Entitätsklassen ist die Änderungsnachverfolgung (siehe Abschnitt 20.6). Es wird ein Schnappschuss vom Ausgangszustand erstellt, der später mit dem aktuellen Zustand verglichen werden kann, um Änderungen festzustellen und in die Datenbank zu sichern. In die Änderungsverfolgung werden automatisch einbezogen:
  - Aus der Datenbank geladene Entitäten
  - Entitäten, die aufgrund einer Master-Details - Beziehung (aufgrund einer sogenannten *Navigationseigenschaft*, siehe Abschnitt 20.2.2.9) erreichbar sind

  Eine explizite Aufnahme kommt z. B. in Frage bei:
  - Neu erstellten Entitäten
  - Entitäten, die von anderen **DbContext**-Objekten geladen wurden

  Details zur Änderungsnachverfolgung werden im Abschnitt 20.6 behandelt.
- Von der Datenbank abgefragte, aber nicht durch Variablen referenzierte Entitäten werden vom **DbContext**-Objekt automatisch in einem Cache aufbewahrt. Wird ein dort vorhandenes Objekt erneut abgefragt, wird es ohne Datenbankzugriff aus dem Cache bezogen.

In der Regel dient ein **DbContext**-Objekt nicht zur Versorgung einer kompletten Anwendung, sondern es verwaltet eine Arbeitseinheit mit Datenbankbeteiligung, die eine kurze zeitliche Ausdehnung besitzt und typischerweise die folgenden Schritte umfasst:[1]

- Es wird ein **DbContext**-Objekt erstellt.
- Es werden Entitäten aus der Datenbank abgefragt oder neu erstellt.
- An den Entitäten werden von der Geschäftslogik diktierte Änderungen vorgenommen, die das **DbContext**-Objekt nachverfolgt.
- Durch die **DbContext**-Methoden **SaveChanges()** oder **SaveChangesAsync()** werden registrierte Änderungen an Entitätsobjekten in die Datenbank geschrieben.
- Das **DbContext**-Objekt erhält einen **Dispose()** - Aufruf.

Weil die Klasse **DbContext** das Interface **IDisposable** erfüllt, sollte am Ende einer Arbeitseinheit für das verwendete **DbContext**-Objekt die **Dispose()** - Methode aufgerufen werden, um die belegten Ressourcen (speziell die Datenbankverbindung) freizugeben. Nach Albahari & Johannsen (2020, S. 409f) ist der **Dispose()** - Aufruf jedoch meist überflüssig und manchmal sogar schädlich:

- Das EF Core schließt eine nicht mehr benötigte Datenbankverbindung automatisch.
- Wegen der verzögerten Ausführung von LINQ-Abfragen (siehe Abschnitt 19.1.2) kann es zu voreiligen **Dispose()** - Aufrufen kommen. Beim Zugriff auf ein **IQueryable<TEntity>** - Objekt ist das zuständige **DbContext**-Objekt eventuell schon entsorgt und kein Datenbankkontakt mehr möglich.

[1] https://docs.microsoft.com/de-de/ef/core/dbcontext-configuration/

Albahari & Johannsen (2020) deklarieren in Ihren Beispielen die **DbContext**-Objekte aber regelmäßig im Rahmen von **using**-Anweisungen, sodass ein **Dispose()** - Aufruf sichergestellt ist. Diese Praxis wird auch im Manuskript verwendet.

Während über die Notwendigkeit und Sinnhaftigkeit eines **Dispose()** - Aufrufs an ein **DbContext**-Objekt diskutiert wird, sind die folgenden Empfehlungen unstrittig:

- Die Klasse **DbContext** ist *nicht* Thread-sicher, sodass ein Objekt dieser Klasse nicht in verschiedenen Threads verwendet werden darf. Es ist also z. B. dringend davon abzuraten, eine asynchrone Methode wie **SaveChangesAsync()** aufzurufen und dann parallel im aufrufenden Thread auf das beteiligte **DbContext**-Objekt zuzugreifen.
- Wenn eine EF Core - Methode eine **InvalidOperationException** wirft, dann kann sich das **DbContext**-Objekt in einem defekten Zustand befinden, sodass es nicht mehr verwendet werden darf.

20.2.1.2 Konfiguration

Bei der Arbeit mit dem EF Core definiert man eine Ableitung aus der Klasse **DbContext** (im Namensraum **Microsoft.EntityFrameworkCore**), und zur Konfiguration der nach diesem Bauplan per **new**-Operator erstellten Objekte überschreibt man die von **DbContext** geerbte virtuelle Methode **OnConfiguring()**, die vom EF Core bei jeder Erstellung eines neuen Kontextobjekts aufgerufen wird:

protected internal virtual void OnConfiguring(DbContextOptionsBuilder *optionsBuilder***)**

Die folgende **DbContext**-Ableitung stammt aus dem Einführungsbeispiel der Microsoft-Dokumentation zu EF Core, das im Abschnitt 20.3.1 näher vorgestellt wird:[1]

```
using Microsoft.EntityFrameworkCore;
 . . .
public class BloggingContext : DbContext {
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder options) {
        options.UseSqlServer("Server=tcp:bernhard\\SQLEXPRESS; " +
            "Initial Catalog=EFGetStarted; Integrated Security=true");
    }
}
```

In der **DbContext**-Ableitung ist zu jeder Entitätsklasse (Konkretisierung des Typformalparameters **TEntity**) des EF Core - Modells eine Eigenschaft vom Typ **DbSet<TEntity>** vorhanden (siehe Abschnitt 20.2.2). Die Klasse **DbSet<TEntity>** implementiert die Schnittstelle **IQueryable<TEntity>**, sodass die in der statischen Klasse **Queryable** im Namensraum **System.Linq** definierten Erweiterungsmethoden genutzt werden können, um per LINQ-to-Entities Entitäten aus Datenbanken abzufragen bzw. dorthin zu sichern (siehe Abschnitt 20.5.2).

Die virtuelle **DbContext**-Methode **OnConfiguring()** besitzt einen Parameter vom Typ **DbContextOptionsBuilder**. Für diese Klasse enthalten die meisten EF Core - Provider eine Erweiterungsmethode zur Durchführung der Konfiguration. Beim EF Core - Provider zu Microsofts SQL Server befindet sich in der Klasse **SqlServerDbContextOptionsExtensions** (im Namensraum **Microsoft.EntityFrameworkCore**) die Erweiterungsmethode **UseSqlServer()**:

---

[1] https://docs.microsoft.com/de-de/ef/core/dbcontext-configuration/

public static DbContextOptionsBuilder UseSqlServer(
this DbContextOptionsBuilder *optionsBuilder***,**
String *connectionString***,**
Action<SqlServerDbContextOptionsBuilder> *sqlServerOptionsAction* **= default)**

Diese Methode informiert das **DbContext**-Objekt darüber, ...

- dass es mit Microsofts SQL Server kooperieren soll,
- welche Datenbank es verwenden soll.
  Die SQL Server - Instanz und die dort stationierte Datenbank werden durch die Verbindungszeichenfolge (vgl. Abschnitte 18.6.3.1 und 18.8.3) im zweiten **UseSqlServer()** - Parameter beschrieben.

Ohne eine Installation des NuGet-Pakets mit dem EF Core - Provider zum SQL Server ist die Erweiterungsmethode **UseSqlServer()** nicht verfügbar.

Im obigen Quellcode taucht nur der zweite **UseSqlServer()** - Parameter auf, weil ...

- der erste Parameter dazu dient, die Verbindung der Erweiterungsmethode zum unterstützten Datentyp herzustellen (vgl. Abschnitt 7.13),
- der dritte Parameter einen Voreinstellungswert besitzt und daher weggelassen werden darf (siehe Abschnitt 5.3.3 zu optionalen Parametern).

NuGet-Pakete mit einer EF Core - Unterstützung sind auch für andere Datenbankverwaltungssysteme vorhanden (z. B. für Oracle DB, MySQL, PostgreSQL, DB2, SQLite).[1] Um einen alternativen Provider für ein **DbContext**-Objekt zu verwenden, ruft man in der **DbContext**-Methode **OnConfiguring()** statt **UseSqlServer()** eine vom Provider implementierte Methode mit dem Namensanfang **Use** auf (z.B. **UseMySQL()**, **UseSqlite()**).

Weitere **DbContext**-Konfigurationen sind der BCL-Dokumentation zu entnehmen.[2]

Im Abschnitt 18.6.13 wird erläutert, wie man die Verbindungszeichenfolge aus der Anwendungskonfigurationsdatei **appsettings.json** lesen kann, damit die Anwendung bei einer Änderung der Verbindungszeichenfolge nicht neu übersetzt werden muss:

```
protected override void OnConfiguring(DbContextOptionsBuilder options) {
    IConfigurationRoot config = new ConfigurationBuilder()
        .AddJsonFile("appsettings.json")
        .Build();
    options.UseSqlServer(config.GetConnectionString("EFGetStartedDb"));
}
```

20.2.2 Entitätsklassen

Eine Entitätsklasse repräsentiert im Sinn der objektorientierten Analyse (vgl. Abschnitt 1.2) eine Sorte von Objekten aus dem Aufgabenbereich des Programms. Weil eine Datenbank als Persistenzlösung verwendet wird, ist eine Entitätsklasse mit einer Datenbanktabelle assoziiert, und ein Objekt einer Entitätsklasse enthält Informationen aus einer Tabellenzeile.

Das empfohlene Verfahren zur Aufnahme einer Entitätsklasse (als Konkretisierung des Typformalparameters) **TEntity** in ein EF Core - Modell besteht darin, in der **DbContext** - Ableitung eine automatisch implementierte Eigenschaft vom Typ der generischen Klasse **DbSet<TEntity>** zu deklarieren (siehe Beispiel im Abschnitt 20.2.1.2). Per Voreinstellung wird vom EF Core jede **DbSet<TEntity>** - Klasse auf eine Datenbanktabelle abgebildet.

---

1 https://docs.microsoft.com/de-de/ef/core/providers/

2 https://docs.microsoft.com/en-us/ef/core/dbcontext-configuration/

Ist eine Klasse Bestandteil einer vollständig definierten Beziehung (siehe Abschnitt 20.2.2.9), dann wird sie auch *ohne* **DbSet<TEntity>** - Eigenschaft in der **DbContext** - Ableitung als Entitätsklasse erkannt, und die zugehörige Datenbanktabelle erhält den Namen der Klasse (Brind 2021).

Mit der Annotation **NotMapped** kann eine entitäts-taugliche Klasse aus dem EF Core - Modell ausgeschlossen werden, z. B.:

```
[NotMapped]
public class Post {
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public Blog Blog { get; set; }
}
```

Entitätsklassen tauchen oft in CRUD-Anwendungen auf (*Create, Read, Update, Delete*), die sich auf die Verwaltung von Datenbeständen konzentrieren, doch dürfen diese Klassen selbstverständlich auch mit Handlungskompetenzen (Methoden) ausgestattet werden.

20.2.2.1 Datenbankschema per Konvention und/oder per Konfiguration

Bei der mit *Code First* bezeichneten bevorzugten EF Core - Einsatzart wird ein Modell definiert und später dazu passend eine Datenbank erstellt (vgl. Abschnitt 20.3). Dabei versucht das EF Core per Voreinstellung, das Datenbankschema durch Anwendung von **Konventionen** aus den Klassen im Modell zu erschließen. So wird z. B. angenommen, dass die einer Entitätsklasse zugeordnete Datenbanktabelle den Namen der **DbContext**-Eigenschaft vom Typ **DbSet<TEntity>** besitzt. Ein abweichender Datenbankname kann mit der Annotation **Table** deklariert werden, z. B.:

```
[Table("blogs")]
public class Blog {
    public int BlogId { get; set; }
    public string Url { get; set; }
    public List<Post> Posts { get; } = new List<Post>();
}
```

Alternativ lässt sich derselbe Zweck durch eine Überschreibung der **DbContext**-Methode **OnModelCreating()** erzielen, z. B.:

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Blog>().ToTable("blogs");
}
```

Auch andere Konfigurationen des Datenbankschemas können alternativ über Annotationen (Namensraum **System.ComponentModel.DataAnnotations** oder **System.ComponentModel.DataAnnotations.Schema**) oder mit Hilfe der **DbContext**-Methode **OnModelCreating()** vorgenommen werden. Die zuletzt genannte Technik ist wegen der etwas komplexen Syntax weniger intuitiv, hat aber einige Vorteile:

- Einige Einstellungen sind ausschließlich über die Methode **OnModelCreating()** realisierbar.
- Man kann alle Abweichungen von der Schemakonfiguration per Konvention an zentraler Stelle versammeln.
- Es ist möglich, Schemainformationen zur Laufzeit festzulegen (z. B. durch Verwendung einer Variablen als **ToTable()** - Parameter im obigen Beispiel).

Im Manuskript wird von den möglichen Konfigurationstechniken meist nur die (subjektiv) bequemere vorgeführt. Für eine vollständige Beschreibung der beiden alternativen Techniken wird auf die im Internet verfügbaren EF Core - Dokumentationen von Mike Brind[1] und Microsoft[2] verwiesen.

Damit im Code-First - Modus eine Schemakonfiguration in der Datenbank realisiert wird, muss eine sogenannte *Migration* erstellt und ausgeführt werden (siehe Abschnitt 20.3). Das kann bei der initialen Erstellung einer Datenbank geschehen oder bei einer späteren Änderung. In der Paket-Manager-Konsole sind die Kommandos **Add-Migration** und **Update-Database** auszuführen, z. B.:

```
PM> Add-Migration InitialCreate
PM> Update-Database
```

20.2.2.2 Abgebildete Eigenschaften

Per Voreinstellung werden aus einer Entitätsklasse alle öffentlichen Eigenschaften mit einem Setter und einem Getter in das EF Core - Modell aufgenommen und bei Verwendung einer relationalen Datenbank auf die Spalten der zugehörigen Tabelle abgebildet. Mit der Annotation **NotMapped** lässt sich eine Entitätseigenschaft aus dem EF Core - Modell ausschließen, z. B:

```
public class Blog {
    public int BlogId { get; set; }
    public string Url { get; set; }
    [NotMapped]
    public long GoogleHits { get; set; }
}
```

20.2.2.3 Spaltennamen

Per Voreinstellung geht das EF Core davon aus, dass die einer Entitätseigenschaft zugeordnete Tabellenspalte den Namen der Eigenschaft trägt. Ein abweichender Spaltenname kann mit der Annotation **Column** deklariert werden, z. B.:

```
public class Blog {
    [Column("Id")]
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

20.2.2.4 Datentypen der Spalten

Den Typ der Datenbankspalte wählt der EF Core - Provider passend zum .NET - Datentyp einer Entitätseigenschaft. Die folgende Tabelle zeigt für wichtige .NET - Datentypen den zugeordneten SQL Server - Datentyp:[3]

---

1 https://www.learnentityframeworkcore.com/configuration

2 https://docs.microsoft.com/en-us/ef/core/

3 https://docs.microsoft.com/de-de/dotnet/framework/data/adonet/sql-server-data-type-mappings
https://www.entityframeworktutorial.net/efcore/conventions-in-ef-core.aspx

.NET - Datentyp	Microsoft SQL Server - Datentyp
bool	bit
byte	tinyint
byte[]	varbinary(max)
short	smallint
int	int
long	bigint
float	real
double	float
decimal	decimal(18,2)
String	nvarchar(max)
DateTime	datetime2(7)

Über die Annotation **Column** kann ein alternativer Typ für die Datenbankspalte gewählt werden, z. B.:

```
[Column(TypeName = "nvarchar(200)")]
public string Url { get; set; }
```

Bei den Datentypen **String** und **byte[]** können für das Datenbankschema über die Annotationen **MaxLength** und **MinLength** Längenbeschränkungen angeordnet werden. Im folgenden Beispiel

```
[MaxLength(200), MinLength(5)]
public string Title { get; set; }
```

resultiert in einer per Migration erstellten Datenbank der SQL Server - Datentyp **nvarchar(200)**. Die Minimalanforderung wirkt sich also *nicht* auf das Datenbankschema aus.

Beim Datentyp **String** kann statt der Annotationen **MinLength** und **MaxLength** auch die Annotation **StringLength** verwendet werden, die einen obligatorischen Parameter für die maximale und einen optionalen Parameter für die minimale Länge besitzt, z. B.:

```
[StringLength(200, MinimumLength = 5)]
public string Title { get; set; }
```

Bei der Programmausführung gilt für alle Längenbeschränkungsattribute, ...

- dass eine Unterschreitung der Mindestlänge ohne Folgen bleibt,
- während eine Überschreitung der Maximallänge zu einem Ausnahmefehler führt.

20.2.2.5 Optionale und obligatorische Spalten

Für die Spalten in einer Datenbanktabelle sind per Voreinstellung fehlende Werte zulässig. Mit der NOT NULL - Beschränkung wird diese Voreinstellung aufgehoben. Das EF Core setzt per Voreinstellung für eine Datenbankspalte die NOT NULL - Beschränkung genau dann, wenn für die zugehörige Eigenschaft der Entitätsklasse *kein*e **null**-Zulässigkeit besteht, z. B. ...

- für die elementaren Datentypen **bool**, **byte**, **short**, **int**, **float**, **double**, **decimal**
- für Verweistypen ohne **null**-Zulässigkeit (möglich seit C# 8.0)

Mit der Annotation **Required** kann die NOT NULL - Beschränkung auch für Entitätseigenschaften mit einem **null**-fähigen Datentyp gesetzt werden, was z. B. bei Eigenschaften mit dem Typ **String** in Frage kommt:

```
[Required]
public string Url { get; set; }
```

20.2.2.6 Primärschlüssel

Per Voreinstellung wird eine Eigenschaft mit dem Namen **Id** oder *Typname***Id** als Primärschlüssel verwendet, z. B.:

```
public class Blog {
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; } = new List<Post>();
}
```

Über die Annotation **Key** kann eine Entitätseigenschaft mit einem alternativen Namen als Primärschlüssel deklariert werden, z. B.:

```
[Key]
public int Kennung { get; set; }
```

In der Regel wird für den Primärschlüssel einer Entitätsklasse bzw. Tabelle ein ganzzahliger Datentyp verwendet. Über alternative Datentypen sowie über zusammengesetzte (aus mehreren Eigenschaften gebildete) Primärschlüssel und weitere Details informiert Microsofts Dokumentation zum EF Core.[1]

Für einen Primärschlüssel mit ganzzahligem Datentyp erhalten neue Zeilen in der Regel einen vom DBMS berechneten Wert. Microsofts SQL Server vergibt per Voreinstellung mit 1 startende und um 1 wachsende Werte.

Über die Annotation **DatabaseGenerated** kann die automatische Vergabe der Primärschlüsselwerte durch das DBMS abgeschaltet werden:

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
public int BlogId { get; set; }
```

20.2.2.7 Generierte Werte

Für Primärschlüssel und einige andere Tabellenspalten (z.B. für eine zur Parallelitätsverwaltung dienende Spalte mit der Zeilenversion, siehe Abschnitt 20.7.4) berechnet ein DBMS die Werte per Voreinstellung automatisch. Im Abschnitt 20.2.2.6 wurde beschrieben, wie die automatische Wertberechnung für einen Primärschlüssel durch die Annotation **DatabaseGenerated** abgeschaltet werden kann.

Häufig ist es erforderlich, für die Entitäten einer Klasse bzw. für die zugehörigen Tabellenzeilen den Erstellungszeitpunkt festzuhalten. Dazu eignet sich eine Eigenschaft vom Typ **DateTime**, z. B.:

```
public DateTime Inserted { get; set;}
```

Es ist sinnvoll, den Erstellungszeitpunkt durch das DBMS ermitteln zu lassen, damit sich die Anwendung nicht darum kümmern muss. Dazu muss in der **DbContext**-Methode **OnModelCreating()** eine zur Berechnung des Erstellungszeitpunkts geeignete SQL-Funktion angegeben werden, z. B.:

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Blog>()
        .Property(b => b.Inserted)
        .HasComputedColumnSql("GETDATE()");
}
```

Wird statt der Methode **HasDefaultValueSql()** die Alternative **HasComputedColumnSql()** verwendet, dann setzt das DBMS den Wert der Spalte `Inserted` nicht nur beim Einfügen einer neuen

---

[1] https://docs.microsoft.com/en-us/ef/core/modeling/keys

Zeile, sondern auch bei jeder Aktualisierung der Zeile. So lässt sich das für eine Entität bzw. Datenbankzeile das Datum der letzten Änderung realisieren.

Hier begegnet uns der Unterschied zwischen einem ...

- Standardwert
  Er wird vom DBMS vergeben, wenn die Spalte im **INSERT**-Kommando für eine neue Zeile unversorgt bleibt. In der **DbContext**-Methode **OnModelCreating()** ist die Methode **HasDefaultValueSql()** zu verwenden.
- und einem berechneten Wert.
  Er wird vom DBMS bei jedem **INSERT**- und bei jedem **UPDATE**-Kommando neu gesetzt. In der **DbContext**-Methode **OnModelCreating()** ist die Methode **HasComputedColumnSql()** zu verwenden.

Im folgenden Beispiel wird mit Hilfe der Methode **HasDefaultValue()** ein Standardwert für eine Entitätseigenschaft bzw. Tabellenspalte mit dem Typ **int** vereinbart:

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Blog>()
        .Property(b => b.Rating)
        .HasDefaultValue(2);
}
```

20.2.2.8 Schatteneigenschaften

Das EF Core - Modell zu einer Datenbank kann zu einer Tabelle gehörende Eigenschaften enthalten, die nicht in der Entitätsklasse zur Tabelle definiert sind, sondern ausschließlich von der Änderungsnachverfolgung der **DbContext**-Klasse verwaltet werden.

Mögliche Motive für solche *Schatteneigenschaften* (engl.: *shadow properties*) können sein:[1]

- Es besteht kein Zugriff auf den Quellcode einer Entitätsklasse.
- Datenbanktechnische Eigenschaften wie z. B. der Zeitpunkt der letzten Änderung oder die zur Parallelitätskontrolle (vgl. Abschnitt 20.7.4) dienende Zeilenversion sollen aus der Entitätsklasse ferngehalten werden.

Im folgenden Beispiel wird zur Entitätsklasse `Blog` mit Hilfe der **DbContext**-Methode **OnModelCreating()** die Schatteneigenschaft `LastUpdated` vom Typ `DateTime` erstellt:

```
public class BloggingContext : DbContext {
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder options) {
        options.UseSqlServer("Server=tcp:bernhard\\SQLEXPRESS; " +
            "Initial Catalog=EFGetStarted; Integrated Security=true");
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {
        modelBuilder.Entity<Blog>()
                    .Property<DateTime>("LastUpdated")
                    .HasComputedColumnSql("GETDATE()");
    }
}
```

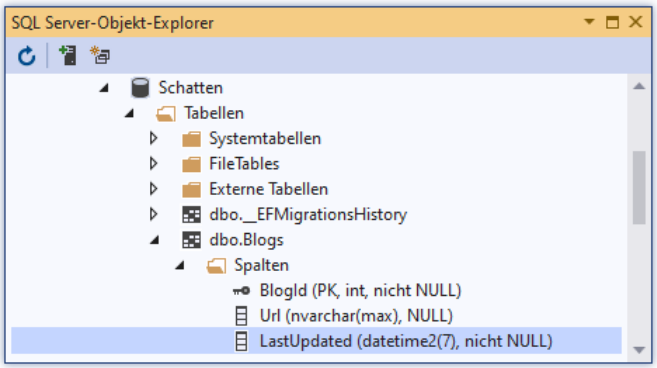
[1] https://www.learnentityframeworkcore.com/model/shadow-properties

```
public class Blog {
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; } = new List<Post>();
}
```

Wie das Beispiel zeigt, kann eine Schatteneigenschaft in der Methode **OnModelCreating()** genauso konfiguriert werden wie eine Entitätseigenschaft. Mit Hilfe der Methode **HasComputedColumnSql()** muss eine zur Berechnung des Erstellungszeitpunkts geeignete SQL-Funktion angegeben werden, weil die Schatteneigenschaft `LastUpdated` anderenfalls stets den Wert 01.01.0001 00:00:00 erhält.

Konsistent mit der Kreation in der Methode **OnModelCreating()** werden Schatteneigenschaften dem EF Core Modell zugerechnet, obwohl sie zu keiner Entitätsklasse gehören.

In der auf Basis des EF Core - Modells erstellten Datenbank enthält die Tabelle `Blogs` die Spalte `LastUpdated`:



Mit Hilfe der **DbContext**-Methode **Entry<TEntity>()**

public virtual EntityEntry<TEntity> Entry<TEntity>(TEntity *entity***) where TEntity : class**

sowie der **EntityEntry**-Methode **Property()**

public virtual PropertyEntry Property(String *propertyName***)**

und der **MemberEntry**-Eigenschaft **CurrentValue**

public virtual object CurrentValue { get; set; }

lassen sich die Werte von Schatteneigenschaften ändern, z. B.:

```
var blog = new Blog { Url = "https://devblogs.microsoft.com/aspnet/" };
context.Add(blog); // Entität in die Änderungsnachverfolgung aufnehmen
context.Entry(blog).Property("LastUpdated").CurrentValue = DateTime.UtcNow;
context.SaveChanges();
```

Fehlt in einer Details-Entitätsklasse der Fremdschlüssel, dann wird vom EF Core automatisch eine entsprechende Schatteneigenschaft eingefügt (vgl. Abschnitt 20.2.2.9).

Im Vorgriff auf Abschnitt 20.5 über die Abfrage von Daten per LINQ-to-Entities soll auf Anregung durch Brind (2021) demonstriert werden, dass Schatteneigenschaften mit Hilfe der generischen Methode **Property()** aus der Klasse **EF** (im Namensraum **Microsoft.EntityFrameworkCore**)

public static TProperty Property<TProperty>(Object *entity***, String** *propertyName***)**

auch in Abfragen verwendet werden können:[1]

[1] https://www.learnentityframeworkcore.com/model/shadow-properties

```
var blogs = context.Blogs.OrderBy(b => EF.Property<DateTime>(b, "LastUpdated"));
```

20.2.2.9 Master-Details - Beziehung

Bei den Beziehungen zwischen Entitätsklassen bzw. Datenbanktabellen beschränken wir uns auf den mit Abstand wichtigsten Fall der Master-Details - Beziehung (one-to-many, 1:n). Über die Behandlung anderer Beziehungstypen (one-to-one, many-to-many) im EF Core informiert z. B. die Online-Dokumentation.[1]

20.2.2.9.1 Begriffe und Beispiel

Bei einer Master-Details - Beziehung sind beteiligt:

- Die **Master-Entitätsklasse** (z. B. `Blog`)
  Sie besitzt ...
  - eine **Primärschlüssel-Eigenschaft**
  - eine sogenannte **Kollektions-Navigationseigenschaft**
    Diese Eigenschaft zeigt auf eine Kollektion mit den Objekten aus der Details-Klasse, die zu einem Master-Objekt gehören.

  Microsoft verwendet auch die Begriffe *Prinzipalentität* (engl.: *principal entity*) und *Auflistungs-Navigationseigenschaft*.[2]
- Die **Details-Entitätsklasse** (z. B. `Post`)
  Sie besitzt ...
  - eine **Primärschlüssel-Eigenschaft**
  - in der Regel eine **Fremdschlüssel-Eigenschaft**
    Diese hat den Datentyp und meist auch den Namen des Master-Primärschlüssels. Fehlt der Fremdschlüssel, wird er vom EF Core durch eine Schatteneigenschaft realisiert (siehe Abschnitt 20.2.2.8).
  - in der Regel eine sogenannte **Referenz-Navigationseigenschaft**
    Diese Eigenschaft zeigt auf das Master-Objekt, zu dem ein Details-Objekt gehört.

  Microsoft spricht auch von *abhängigen Entitäten* (engl.: *dependent entities*).

Im folgenden Beispiel

```
public class Blog {
    public int BlogId { get; set; } // Primärschlüssel
    public string Url { get; set; }

// Kollektions-Navigationseigenschaft
    public List<Post> Posts { get; } = new List<Post>();
}

public class Post {
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; } // Fremdschlüssel
    public Blog Blog { get; set; } // Referenz-Navigationseigenschaft
}
```

sind:

---

[1] https://docs.microsoft.com/en-us/ef/core/modeling/relationships#other-relationship-patterns
https://docs.microsoft.com/en-us/ef/core/change-tracking/relationship-changes

[2] https://docs.microsoft.com/de-de/ef/core/modeling/relationships

- `Blog` die Master-Entitätsklasse (assoziiert mit der Master-Tabelle `Blogs`)
  - mit dem Primärschlüssel `BlogId`
  - und der Kollektions-Navigationseigenschaft `Posts`
- `Post` die Details-Entitätsklasse (assoziiert mit der Details-Tabelle `Posts`)
  - mit dem Primärschlüssel `PostId`
  - mit dem Fremdschlüssel `BlogId`
  - und der Referenz-Navigationseigenschaft `Blog`

Die Primärschlüsseleigenschaften der beiden Entitätsklassen und die Fremdschlüsseleigenschaft der Details-Klasse sind auf Spalten in den jeweiligen Datenbanktabellen abgebildet. Demgegenüber sind von den Navigationseigenschaften in der Datenbank keine direkten Entsprechungen zu finden. Die zugehörigen Datentypen (im Beispiel: `List<Post>` und `Blog`) sind dem DBMS fremd.

Wenn bei bestehender Master-Details - Beziehung zwischen zwei Entitätsklassen ...

- eine Prinzipalentität gelöscht wird,
- und abhängige Entitäten vorhanden sind,

dann ist die referentielle Integrität der Datenbank tangiert. Dabei ist es von Bedeutung, ob die Master-Details - Beziehung als obligatorisch oder als optional definiert ist:

- Eine **obligatorische Beziehung** ist daran zu erkennen, dass für den Fremdschlüssel das **null**-Verbot besteht. Jede Details-Entität muss also auf eine Master-Entität verweisen. Beim Löschen einer Prinzipalentität ...
  - müssen entweder auch die abhängigen Entitäten gelöscht werden,
  - oder das Löschen der Prinzipalentität muss verhindert werden.
- Bei einer **optionalen Beziehung** sind für den Fremdschlüssel auch **null**-Werte erlaubt. Es sind also Details-Entitäten ohne zugehörige Master-Entität zulässig. Beim Löschen einer Prinzipalentität kann man ...
  - entweder die Fremdschlüsselwerte der abhängigen Entitäten auf **null** setzen
  - oder die abhängigen Entitäten ebenfalls löschen.

Mit dem Löschen von Entitäten bei bestehender Master-Details - Beziehung werden wir uns im Abschnitt 20.7.3 ausführlich beschäftigen.

20.2.2.9.2 Auf Konventionen basierende Vereinbarung

Eine vollständige, auf den EF Core - Konventionen basierende Definition einer Master-Details - Beziehung enthält:

- ein Paar korrespondierender Navigationseigenschaften:
  - vom Kollektionstyp in der Master-Entitätsklasse
  - vom Referenztyp in der Details-Entitätsklasse
- einen Fremdschlüssel mit einem eindeutig erkennbaren Namen nach einem Muster aus der folgenden Serie (mit absteigender Bevorzugung):
  - <Navigations-Eigenschaftsname><Master-Primärschlüsselname>
  - <Navigations-Eigenschaftsname>**Id**
  - <Master-Entitätsklassenname><Master-Primärschlüsselname>
  - <Master-Entitätsklassenname>**Id**

  Der Datentyp des Fremdschlüssels muss abgesehen von der **null** - (Un)zulässigkeit mit dem Datentyp des Primärschlüssels aus der Master-Entitätsklasse übereinstimmen.

In der Definition einer Master-Details - Beziehung sind nicht alle Bestimmungsstücke obligatorisch:

- Die Referenz-Navigationseigenschaft darf fehlen.
- Ein in der Details-Entitätsklasse fehlender Fremdschlüssel wird per Schatteneigenschaft *mit* erlaubtem **null**-Wert eingefügt (vgl. Abschnitt 20.2.2.8). Bei fehlender Fremdschlüssel-Definition entscheidet also das EF Core, dass eine *optionale* Master-Details - Beziehung entsteht.

Zur Begründung einer Master-Details - Beziehung genügt also eine Kollektions-Navigationseigenschaft in der Master-Entitätsklasse, wobei ein in der Details-Entitätsklasse fehlender Fremdschlüssel vom EF Core per Schatteneigenschaft realisiert wird.

20.2.2.9.3 Beziehung konfigurieren

Eine Master-Details - Beziehung wird im EF Core meist durch Beachtung der Konventionen vereinbart (siehe Abschnitt 20.2.2.9.2), wobei keine zusätzliche Konfiguration erforderlich ist (Brind 2021). Die im aktuellen Abschnitt beschriebenen Möglichkeiten, bei der Vereinbarung einer Master-Details - Beziehung von den EF Core - Konventionen abzuweichen, werden also nur selten benötigt.

In der Details-Entität kann man eine Eigenschaft mit einem „unkonventionellen“ Namen als Fremdschlüssel vereinbaren, indem man eine **ForeignKey**-Annotation mit der Referenz-Navigationseigenschaft als Konstruktorparameter anheftet, z. B.:

```
public class Post {
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    [ForeignKey("Blog")]
    public int BlogFk { get; set; }
    public Blog Blog { get; set; }
}
```

Besitzt eine Details-Entitätsklasse wie im folgenden Beispiel keine Fremdschlüssel-Eigenschaft,

```
public class Post {
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog { get; set; }
}
```

dann erstellt das EF Core eine Schatteneigenschaft mit dieser Funktion (vgl. Abschnitt 20.2.2.9.2). Für diese Schatteneigenschaft wird ein **null**-Wert erlaubt, sodass eine *optionale* Master-Details - Beziehung resultiert. In der folgenden Methode **OnModelCreating()** wird die Beziehung als obligatorisch definiert:

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Blog>()
        .HasMany(e => e.Posts)
        .WithOne(e => e.Blog)
        .IsRequired();
}
```

Daraus resultiert das kaskadierende Löschen abhängiger Details-Zeilen beim Löschen einer Master-Zeile (siehe Abschnitt 20.7.3).

20.2.2.9.4 Beziehung zwischen einem Details- und einem Master-Objekt herstellen

Um ein Details-Entitätsobjekt zu einem Master-Entitätsobjekt in Beziehung zu setzen, nimmt man es in die Details-Kollektion des Master-Objekts auf, z. B:

```
using (var context = new BloggingContext()) {
   var vsBlog = context.Blogs
                .Where(b => b.Url == "https://devblogs.microsoft.com/visualstudio/")
                .Single();
   var vsPost = new Post { Title = "Update 2019 16.11.1" };
   vsBlog.Posts.Add(vsPost);
   context.SaveChanges();
}
```

Nachdem der zum Kontext gehörende **ChangeTracker** (siehe Abschnitt 20.6 zur Änderungsnachverfolgung) über einen Aufruf der Methode **DetectChanges()** auf den aktuellen Stand gebracht worden ist, haben die Fremdschlüsseleigenschaft (im Beispiel mit dem Datentyp **int**) und die Referenz-Navigationseigenschaft (im Beispiel mit dem Datentyp `Blog`) des Details-Objekts vom EF Core konsistente Werte erhalten:

Quellcodesegment	Ausgabe

`context.ChangeTracker.DetectChanges();`
`Console.WriteLine(vsPost.BlogId);`
`Console.WriteLine(vsPost.Blog.Url);`
	`2`
	`https://devblogs.microsoft.com/visualstudio/`

Den Aufruf der Methode **DetectChanges()** nimmt das EF Core nötigenfalls automatisch vor (z.B. vor einem Aufruf der **DbContext**-Methode **SaveChanges()**).

Verschiebt man z. B. ein Details-Objekt in die Navigations-Kollektion eines anderen Master-Objekts, dann werden durch einen Aufruf der Methode **DetectChanges()** die Referenz-Navigationseigenschaft und die Fremdschlüssel-Eigenschaft des Details-Objekts angepasst, z. B.:

Quellcodesegment	Ausgabe

`var blogs = context.Blogs.ToArray();`
`var post = new Post { Title = "Xamarin.Forms 5.0" };`
`blogs[1].Posts.Add(post);`
`context.ChangeTracker.DetectChanges();`
`Console.WriteLine(post.BlogId);`
`blogs[1].Posts.Remove(post);`
`blogs[4].Posts.Add(post);`
`context.ChangeTracker.DetectChanges();`
`Console.WriteLine(post.BlogId);`
	`2`
	`5`

20.2.2.10 Vererbung

Eine ORM-Lösung muss auch die Vererbung berücksichtigen, die (neben der Datenkapselung und der Polymorphie) als zentrales Merkmal der objektorientierten Programmierung gilt und die Wiederverwendung von Code erleichtert (siehe Abschnitt 5.1.1). Das EF Core unterstützt Vererbungsbeziehungen zwischen Entitätsklassen, wobei per Voreinstellung das sogenannte *TPH-Muster* (*Table-Per-Hierarchy*) realisiert wird:[1]

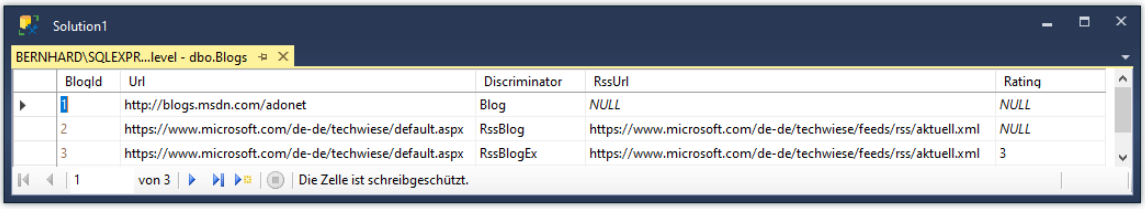
[1] https://docs.microsoft.com/en-us/ef/core/modeling/inheritance

- Für jede Vererbungshierarchie wird *eine* Datenbanktabelle verwendet, die per Voreinstellung ihren Namen von der Basisklasse übernimmt (genauer: von der zur Basisklasse gehörenden **DbSet<T>** - Eigenschaft in der **DbContext**-Ableitung).
- In dieser Tabelle gibt eine Diskriminatorspalte die Zugehörigkeit einer Zeile zu einer speziellen Entitätsklasse aus der Vererbungshierarchie an.

Wie das folgende Beispiel zeigt, ist auch eine Vererbungshierarchie mit mehr als zwei Ebenen möglich:

```
public class BloggingContext : DbContext {
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<RssBlog> RssBlogs { get; set; }
    public DbSet<RssBlogEx> RssExBlogs { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder options) {
        options.UseSqlServer("Server=tcp:bernhard\\SQLEXPRESS; " +
            "Initial Catalog=TphMultilevel; Integrated Security=true");
    }
}

public class Blog {
    public int BlogId { get; set; }
    public string Url { get; set; }
}

public class RssBlog : Blog {
    public string RssUrl { get; set; }
}

public class RssBlogEx : RssBlog {
    public int Rating { get; set; }
}
```

Im Beispiel resultiert in der Datenbank eine Tabelle namens `Blogs` mit der Spalte `Discriminator`, die als Schatteneigenschaft zur Basis-Entitätsklasse `Blog` realisiert ist:



Ist die Entitätsklasse `E2` eine Erweiterung der Klasse `E1`, dann erhalten die zu `E1`-Objekten gehörenden Datenbankzeilen für die in `E2` ergänzten Eigenschaften den Wert **NULL**.

Selbstverständlich ist auch eine *abstrakte* Entitäts-Basisklasse erlaubt.

In Abfragen (vgl. Abschnitt 20.5.1) können die Entitätsklassen gezielt angesprochen werden, z. B.:

```
var rbs = context.RssBlogs.ToList();
```

Seit der Version 5.0 unterstützt das EF Core auch das sogenannte *TPT-Muster* (*Table-Per-Type*), wobei jede Entitätsklasse auf eine eigene Datenbanktabelle abgebildet wird. Microsoft warnt allerdings vor einer im Vergleich zum TPH-Muster reduzierten Performanz.

20.2.2.11 Indizes

In einer Datenbank lassen sich Abfragen durch Indizes beschleunigen. Per Konvention erstellt das EF Core einen Index zu jeder Eigenschaft (bzw. Eigenschaftskombination), die als Fremdschlüssel dient. Über die Annotation **Index** oder mit Hilfe der Methode **OnModelCreating()** lassen sich zusätzliche Indizes definieren. Im folgenden Beispiel aus der EF Core - Dokumentation von Microsoft wird für die Tabelle `Blogs` zur Eigenschaft `Url` über eine Annotation

```
[Index(nameof(Url))]
public class Blog {
    public int BlogId { get; set; }
    public string Url { get; set; }
    public List<Post> Posts { get; } = new List<Post>();
}
```

bzw. über die Methode **OnModelCreating()** ein Index erstellt:[1]

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Blog>()
                .HasIndex(b => b.Url);
}
```

20.2.2.12 Weitere EF Core - Optionen

Einige EF Core - Optionen können aus Zeitgründen im Manuskript nicht behandelt werden, z. B.:[2]

- Tabellenaufteilung (engl.: *table splitting*)
  Diese Technik macht es möglich, mehrere Entitäten einer einzigen Tabellenzeile zuzuordnen.[3]
- Schlüssellose Entitätstypen (engl.: *keyless entity types*, frühere Bezeichnung: *Abfragetypen*)
  Durch diese Typen werden Abfragen ermöglicht für Datenbanktabellen ohne Primärschlüssel und für Datenbankansichten.[4]

20.3 Migrationen (Code First)

Das EF Core ist konzipiert für die als *Code First* bezeichnete Arbeitsweise, die auch bei den bisherigen, durch Beispiele unterstützten Erläuterungen implizit zugrunde lag. Dabei entsteht ein EF Core - Modell, das von der objektorientierten Analyse der Aufgabenstellung geprägt ist, aber auch (über Konventionen und Konfigurationen) die Verwendung einer Datenbank als Persistenzlösung berücksichtigt. Aus dem fertigen Modell wird die Datenbank erstellt, und nach einer Änderung des Modells muss das Datenbankschema aktualisiert werden, ohne vorhandene Daten zu gefährden.

Dabei kommt die Migrationstechnik im EF Core zum Einsatz:

- Mit Hilfe der EF Core Tools wird (ggf. durch Vergleich des aktuellen Modells mit einem Schnappschuss des vorherigen Modells) eine (neue) Migration erstellt. Diese besteht aus C# - Klassen und erscheint im **Projektmappen-Explorer** (siehe unten). Bei Bedarf können die Quellcodedateien einer Migration modifiziert werden.
- Durch die Ausführung der Migration wird die Datenbank erstellt oder modifiziert.

Bei der Transformation einer im produktiven Einsatz befindlichen Datenbank rät Microsoft nachdrücklich davon ab, dem automatisch erstellten Migrationscode blind zu vertrauen (siehe Abschnitt 20.3.3).

---

1 https://docs.microsoft.com/en-us/ef/core/modeling/indexes

2 https://docs.microsoft.com/en-us/ef/core/modeling/

3 https://docs.microsoft.com/en-us/ef/core/modeling/table-splitting

4 https://docs.microsoft.com/en-us/ef/core/modeling/keyless-entity-types

Die Migrationen werden durch Kommandos erstellt und ausgeführt, wobei das EF Core zwei Optionen anbietet (siehe auch Abschnitt 20.1):

- Im Visual Studio können Werkzeuge für die NuGet-Paket-Manager-Konsole verwendet werden, die in der englischen Literatur als *PMC-Tools* bezeichnet werden.
- Die alternativen Konsolen-Werkzeuge werden in der englischen Literatur als *CLI-Tools* (*Command Line Interface*) bezeichnet.

Da wir überwiegend das Visual Studio als Entwicklungsumgebung verwenden, arbeiten wir mit den hier integrierten PMC-Werkzeugen. Über die CLI-Werkzeuge informiert z. B. Brind (2021).[1]

20.3.1 Beispiel

Wir gehen in einem Konsolenprojekt für das Zielframework .NET 5.0 mit den im Abschnitt 20.1 beschriebenen NuGet-Paketinstallationen von einem fertig konfigurierten EF Modell bestehend aus den Klassen `BloggingContext`, `Blog` und `Post` aus:

```
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;

namespace EFGetStarted {
    public class BloggingContext : DbContext {
        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder options) {
            options.UseSqlServer("Server=tcp:bernhard\\SQLEXPRESS; " +
                "Initial Catalog=EFGetStarted; Integrated Security=true");
        }
    }

    public class Blog {
        public int BlogId { get; set; }
        public string Url { get; set; }

        public List<Post> Posts { get; } = new List<Post>();
    }

    public class Post {
        public int PostId { get; set; }
        public string Title { get; set; }
        public string Content { get; set; }

        public int BlogId { get; set; }
        public Blog Blog { get; set; }
    }
}
```

20.3.2 Datenbank durch eine initiale Migration anlegen

Wir arbeiten mit den Kommandos für die NuGet-Paket-Manager-Konsole und aktivieren bei Bedarf die Konsole mit dem folgenden Menübefehl:

Extras > NuGet-Paket-Manager > Paket-Manager-Konsole

Über das PMC-Kommando **Add-Migration** erstellen wir eine Migration mit dem Namen `InitialCreate`:

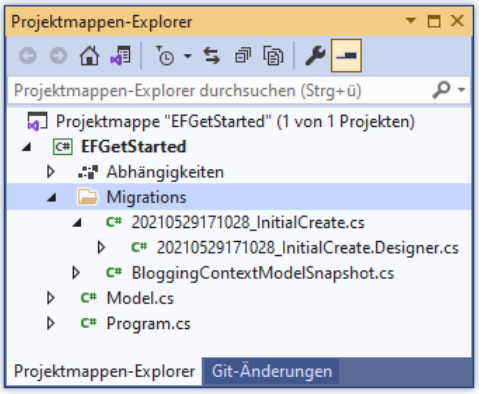
---

[1] https://www.learnentityframeworkcore.com/migrations

```
PM> Add-Migration InitialCreate
Build started...
Build succeeded.
```

Bei den PMC-Kommandos ist die Groß-/Kleinschreibung generell irrelevant.

Die Migration erscheint im **Projektmappen-Explorer**:



Die Quellcodedatei **20210704172729_InitialCreate.cs** übernimmt ihren Namen von der Migration, wobei noch ein Zeitstempel vorangestellt wird. Sie enthält eine Klasse mit dem Namen der Migration (im Beispiel: `InitialCreate`), die von der Klasse **Migration** im Namensraum **Microsoft.EntityFrameworkCore.Migrations** abstammt:

```
public partial class InitialCreate : Migration {
  protected override void Up(MigrationBuilder migrationBuilder) {
    migrationBuilder.CreateTable(
      name: "Blogs",
      columns: table => new
      {
        BlogId = table.Column<int>(type: "int", nullable: false)
          .Annotation("SqlServer:Identity", "1, 1"),
        Url = table.Column<string>(type: "nvarchar(max)", nullable: true)
      },
      constraints: table =>
      {
        table.PrimaryKey("PK_Blogs", x => x.BlogId);
      });

    migrationBuilder.CreateTable(
      name: "Posts",
      columns: table => new
      {
        PostId = table.Column<int>(type: "int", nullable: false)
          .Annotation("SqlServer:Identity", "1, 1"),
        Title = table.Column<string>(type: "nvarchar(max)", nullable: true),
        Content = table.Column<string>(type: "nvarchar(max)", nullable: true),
        BlogId = table.Column<int>(type: "int", nullable: false)
      },
      constraints: table =>
      {
        table.PrimaryKey("PK_Posts", x => x.PostId);
        table.ForeignKey(
          name: "FK_Posts_Blogs_BlogId",
          column: x => x.BlogId,
          principalTable: "Blogs",
          principalColumn: "BlogId",
          onDelete: ReferentialAction.Cascade);
      });

    migrationBuilder.CreateIndex(
      name: "IX_Posts_BlogId",
      table: "Posts",
      column: "BlogId");
  }
```

```
    protected override void Down(MigrationBuilder migrationBuilder) {
      migrationBuilder.DropTable(
        name: "Posts");
      migrationBuilder.DropTable(
        name: "Blogs");
    }
  }
```

In der Klasse `InitialCreate` werden die Methoden **Up()** und **Down()** definiert:

- **Up()** legt mit Hilfe der **MigrationBuilder**-Methode **CreateTable()** die Tabellen `Blogs` und `Posts` in der Datenbank an und definiert deren Schema (inklusive der Master-Details - Beziehung). Außerdem wird zur Fremdschlüsselspalte `BlogId` der Tabelle `Posts` mit Hilfe der **MigrationBuilder**-Methode **CreateIndex()** in der Datenbank ein Index erstellt (vgl. Abschnitt 20.2.2.11).
- **Down()** löscht die Tabellen `Blogs` und `Posts` aus der Datenbank.

In der Methode **Up()** hängt die Definition des Datenbankschemas leider teilweise vom DBMS ab. Für die Spalte `BlogId` in der Tabelle `Blogs` wird durch die folgende Methode(!) **Annotation()** dafür gesorgt, dass Microsofts SQL Server

```
BlogId = table.Column<int>(type: "int", nullable: false)
              .Annotation("SqlServer:Identity", "1, 1"),
```

eine sogenannte *Identitätsspalte* mit dem Startwert 1 und dem Inkrement 1 anlegt. Dazu ist vom SQL Server letztlich das folgende SQL-Kommando **CREATE TABLE** auszuführten:

```
CREATE TABLE [Blogs] (
    [BlogId] int NOT NULL IDENTITY,
    [Url] nvarchar(max) NULL,
    CONSTRAINT [PK_Blogs] PRIMARY KEY ([BlogId])
);
```

Für ein anderes DBMS setzt der zugehörige EF Core - Provider die zugrunde liegende Primärschlüsseldefinition in der Entitätsklasse `Blog`

```
public int BlogId { get; set; }
```

eventuell anders um.

Neben der Klasse `InitialCreate` hat das PMC-Kommando **Add-Migration** im Beispiel auch noch die Klasse `BloggingContextModelSnapshot` angelegt:

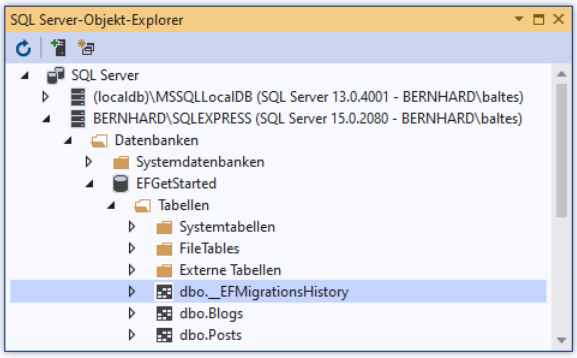
```
[DbContext(typeof(BloggingContext))]
partial class BloggingContextModelSnapshot : ModelSnapshot
{
    protected override void BuildModel(ModelBuilder modelBuilder)
    {
     . . .
    }
}
```

Sie enthält den aktuellen Zustand des EF Core - Modells, und dieser Schnappschuss (engl.: *Snapshot*) wird bei der nächsten Migrationserstellung als Ausgangszustand zugrunde gelegt (siehe Abschnitt 20.3.3).

Über das PMC-Kommando **Update-Database** beauftragen wir das EF Core, zur Erstellung der Datenbank die **Migration**-Methode **Up()** auszuführen:

```
PM> Update-Database
Build started...
Build succeeded.
Applying migration '20210529171028_InitialCreate'.
Done.
```

Ein Blick auf die neue Datenbank mit dem **SQL Server-Objekt-Explorer** im Visual Studio zeigt, dass vom EF Core auch eine Tabelle namens `_EFMigrationsHistory` angelegt wird, um die bereits angewendeten Migrationen verwalten zu können:



Soll eine Migration durch ein nicht direkt erreichbares DBMS ausgeführt werden, dann lässt man das zur Migration gehörende SQL-Skript durch das PMC-Kommando **Script-Migration** erstellen, z. B.:

```
PM> script-migration
Build started...
Build succeeded.
```

Im Beispiel resultiert das folgende SQL-Programm, das nun verteilt werden kann:

```
IF OBJECT_ID(N'[__EFMigrationsHistory]') IS NULL
BEGIN
    CREATE TABLE [__EFMigrationsHistory] (
        [MigrationId] nvarchar(150) NOT NULL,
        [ProductVersion] nvarchar(32) NOT NULL,
        CONSTRAINT [PK___EFMigrationsHistory] PRIMARY KEY ([MigrationId])
    );
END;
GO

BEGIN TRANSACTION;
GO

CREATE TABLE [Blogs] (
    [BlogId] int NOT NULL IDENTITY,
    [Url] nvarchar(max) NULL,
    CONSTRAINT [PK_Blogs] PRIMARY KEY ([BlogId])
);
GO

CREATE TABLE [Posts] (
    [PostId] int NOT NULL IDENTITY,
    [Title] nvarchar(max) NULL,
    [Content] nvarchar(max) NULL,
    [BlogId] int NOT NULL,
    CONSTRAINT [PK_Posts] PRIMARY KEY ([PostId]),
    CONSTRAINT [FK_Posts_Blogs_BlogId] FOREIGN KEY ([BlogId]) REFERENCES [Blogs] ([BlogId]) ON
DELETE CASCADE
);
GO

CREATE INDEX [IX_Posts_BlogId] ON [Posts] ([BlogId]);
GO

INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
VALUES (N'20210704172729_InitialCreate', N'5.0.6');
GO

COMMIT;
GO
```

Das Skript dokumentiert präzise die Effekte der Migration. Wer den SQL-Code ergänzen möchte, kann das über die **MigrationBuilder**-Methode **Sql()** in der **Up()** - und/oder **Down()** - Methode der Migration tun.

20.3.3 Datenbank durch aufbauende Migrationen verändern

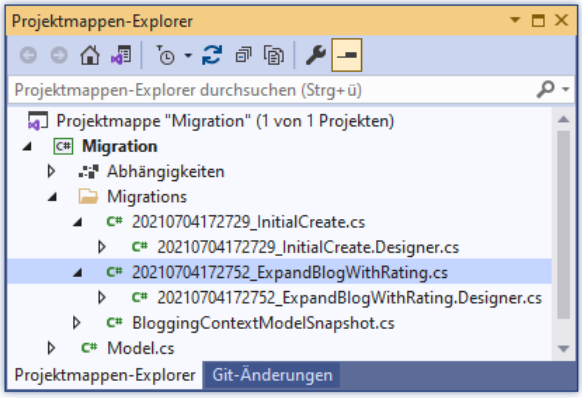
Wir erweitern die Entitätsklasse `Blog` um die Eigenschaft `Rating`:

```
public class Blog {
    public int BlogId { get; set; }
    public string Url { get; set; }
    public int Rating { get; set; }

    public List<Post> Posts { get; } = new List<Post>();
}
```

Um das Datenbankschema anzupassen, wird über das PMC-Kommando **Add-Migration** eine neue Migration erstellt, wobei das EF Core den vorhandenen Schnappschuss (vgl. Abschnitt 20.3.2) als Basis verwendet:

```
PM> Add-Migration ExpandBlogWithRating
Build started...
Build succeeded.
```

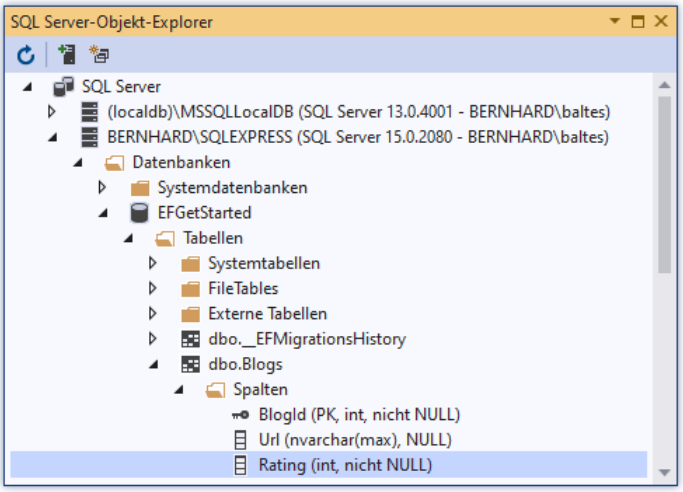
Im Projektmappen-Explorer erscheint die zusätzliche Migration:



Trifft das PMC-Kommando **Update-Database** auf eine vorhandene Datenbank mit dokumentiertem Migrationsstatus, wird nur die fehlende Migration ausgeführt:

```
PM> Update-Database
Build started...
Build succeeded.
Applying migration '20210529183542_ExpandBlogWithRating'.
Done.
```

Nach einer Aufforderung zum **Aktualisieren** zeigt der **SQL Server-Objekt-Explorer** die erweiterte Tabelle `Blogs`:



Fehlt die in der Verbindungszeichenfolge benannte Datenbank (siehe Methode **OnConfiguring()** der Klasse `BloggingContext` im Abschnitt 20.3.1), dann legt das PMC-Kommando **Update-Database** die Datenbank an und führt alle erforderlichen Migrationen aus, z. B.:

```
PM> Update-Database
Build started...
Build succeeded.
Applying migration '20210704172729_InitialCreate'.
Applying migration '20210704172752_ExpandBlogWithRating'.
Done.
```

Um die Datenbank auf den Zustand einer früheren Migration zurückzusetzen, führt man das PMC-Kommando **Update-Database** mit dem gewünschten Zielzustand als Kommandozeilenoption aus. Dabei können auch mehrere *spätere* Migrationen zurückgenommen werden, wobei die zugehörigen **Down()** - Methoden der Migrationsklassen (siehe Abschnitt 20.3.2) nacheinander ausgeführt werden, z. B.:

```
PM> Update-Database InitialCreate
Build started...
Build succeeded.
Reverting migration '20210904130302_AddTestCases'.
Reverting migration '20210704172752_ExpandBlogWithRating'.
```

Die aus der Datenbank entfernten Migrationen sind weiter im Projekt vorhanden und auch die Snapshot-Klasse ändert sich nicht.

Um eine Migration aus dem Projekt zu entfernen, sollte sie nicht einfach gelöscht werden, weil danach die verbliebenen Migrationen nicht mehr konsistent mit der Snapshot-Klasse sind. Zulässig ist hingegen die (wiederholte) Anwendung des PMC-Kommandos **Remove-Migration**, das die jeweils *letzte* Migration entfernt, z.B.:

```
PM> Remove-Migration
Build started...
Build succeeded.
Removing migration '20210904130302_AddTestCases'.
Reverting the model snapshot.
Done.
```

Dabei werden die Klassen im EF Core - Modell (im Beispiel `BloggingContext`, `Blog` und `Post`) *nicht* verändert.

Es kann sinnvoll oder sogar erforderlich sein, den Quellcode einer Migration zu kontrollieren und anzupassen. Bei einigen Datenbanktransformationen ist eine manuelle Veränderung des vom EF

Core erstellten Migrationscodes sogar unbedingt erforderlich, weil anderenfalls ein Datenverlust droht, z. B.:[1]

- Der Plan, eine Spalte umzubenennen, hat ein Löschen und ein Neuerstellen, also einen Datenverlust zur Folge, wenn der automatisch erstellte Migrationscode nicht verändert wird.
- Auch beim Versuch, zwei Spalten zusammenzuführen, bewirkt der automatisch erstellte Migrationscode einen Datenverlust.

Microsoft rät deutlich davon ab, eine im produktiven Einsatz befindliche Datenbank durch ungeprüften EF Core - Migrationscode zu transformieren:[2]

> While productive for local development and testing of migrations, this approach isn't ideal for managing production databases: The SQL commands are applied directly by the tool, without giving the developer a chance to inspect or modify them. This can be dangerous in a production environment.

Stattdessen wird geraten, Datenbanktransformationen per SQL-Skript zu realisieren. Wie am Ende von Abschnitt 20.3.2 zu sehen war, kann man über das PMC-Kommando **Script-Migration** ein SQL-Skript erstellen lassen. Es beinhaltet per Voreinstellung *alle* Migrationen des Projekts, kann aber auch auf einen Bereich von Migrationen eingeschränkt werden.[3]

Grundsätzlich kann die vom PMC-Kommando **Update-Database** veranlasste Datenbankaktualisierung auch vom Anwendungsprogramm in der Startphase vorgenommen werden. Das empfiehlt Microsoft allerdings nur für die lokale Entwicklung und Testung von Migrationen.

20.3.4 Kreation von Testfällen

In der Entwicklungsphase ist es oft von Nutzen, bei der Erstellung der Datenbank zu einem EF Core - Modell Testfälle zu erzeugen. Das lässt sich natürlich mit den regulären EF Core - Techniken zur Datenbankbearbeitung erledigen (siehe Abschnitt 20.7.1), erfordert dann aber einen separaten Datenbankzugriff, was bei häufiger Wiederholung lästig werden kann. Mit Hilfe der **DbContext**-Methode **OnModelCreating()** lässt sich die Bevölkerung einer Datenbank mit Testfällen bei der Erstellung oder Modifikation per Migration erledigen. Im folgenden Beispiel werden zwei `Blog`-Entitäten angelegt, wobei der Primärschlüsselwert anzugeben ist. Zur ersten `Blog`-Master-Entität entstehen auch gleich zwei abhängige `Post`-Details-Entitäten, wobei Werte für die Primär- und die Fremdschlüssel anzugeben sind:

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Blog>().HasData(
        new Blog {
            BlogId = 1,
            Url = "https://devblogs.microsoft.com/aspnet/",
            Rating = 3
        },
        new Blog {
            BlogId = 2,
            Url = "https://devblogs.microsoft.com/visualstudio/",
            Rating = 2
        }
    );
```

---

1 https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/managing

2 https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/applying

3 https://docs.microsoft.com/en-us/ef/core/cli/powershell

```
        modelBuilder.Entity<Post>().HasData(
            new Post {
                PostId = 1,
                BlogId = 2,
                Title= "New in 2019 16.11.2"
            },
            new Post {
                PostId = 2,
                BlogId = 2,
                Title = "VS 2022 Preview"
            }
        );
    }
```

In einer Migrationsklasse, die lediglich die obigen Testfälle ergänzt, sieht die **Up()** - Methode so aus:

```
    protected override void Up(MigrationBuilder migrationBuilder) {
      migrationBuilder.InsertData(
        table: "Blogs",
        columns: new[] { "BlogId", "Rating", "Url" },
        values: new object[] { 1, 3, "http://blogs.msdn.com/adonet/" });

      migrationBuilder.InsertData(
        table: "Blogs",
        columns: new[] { "BlogId", "Rating", "Url" },
        values: new object[] { 2, 2, "https://devblogs.microsoft.com/visualstudio/" });

      migrationBuilder.InsertData(
        table: "Posts",
        columns: new[] { "PostId", "BlogId", "Content", "Title" },
        values: new object[] { 1, 2, null, "New in 2019 16.11.2" });

      migrationBuilder.InsertData(
        table: "Posts",
        columns: new[] { "PostId", "BlogId", "Content", "Title" },
        values: new object[] { 2, 2, null, "VS 2022 Preview" });
    }
```

Weitere Informationen zum Erstellen von Testfällen finden sich bei Brind (2021).[1]

20.4 Reverse Engineering (Database First)

Für eine vorhandene Datenbank kann man nachträglich ein EF Core - Modell (mit einer **DbContext**-Ableitung und einer Entitätsklasse zu jeder relevanten Datenbanktabelle) erstellen lassen. Auch für diese Leistung sind die EF Core Tools zuständig, die in einer PMC (Package Manager Console) - und einer CLI (Command Line Interface) - Variante verfügbar sind (zur Installation der Werkzeuge siehe Abschnitt 20.1). Wir wählen (wie schon im Abschnitt 20.3 über die Migration) die ins Visual Studio integrierten PMC-Werkzeuge.

Nachdem aus einer vorhandenen Datenbank ein EF Core - Modell entstanden ist, kann nach einer Änderung des Modells die erforderliche Aktualisierung der Datenbank mit der Migrationstechnik vorgenommen werden (siehe Abschnitt 20.4.3).

Das Erstellen des EF Core - Modells zu einer vorhandenen Datenbank wird übereinstimmend als *reverse engineering* (dt.: *Zurückentwicklung*, *Nachkonstruktion*) bezeichnet. Bei der Verwendung der Bezeichnung *Database First* besteht weniger Eindeutigkeit, weil in früheren Entity Framework

[1] https://www.learnentityframeworkcore.com/migrations/seeding

- Versionen diese Bezeichnung für eine im EF Core nicht mehr unterstützte Technik verwendet wurde (mit einer zentralen Rolle für sogenannte *EDMX* -Dateien). Manche Autoren bezeichnen daher das im aktuellen Abschnitt beschriebene Reverse Engineering auch als *Code First - Ansatz für existierende Datenbanken* (z.B. Brind 2021).[1]

20.4.1 PMC-Kommando Scaffold-DbContext

Wir lassen das EF Core - Modell (mit einer **DbContext**-Ableitung und einer Entitätsklasse zu jeder relevanten Datenbanktabelle) zu einer vorhandenen Datenbank durch das PMC-Kommando **Scaffold-DbContext** erstellen (dt. Übersetzung von *scaffold*: *Gerüst*). Seine wichtigsten Kommandozeilenargumente sind:

- Als erstes Kommandozeilenargument ist eine Verbindungszeichenfolge anzugeben (vgl. Abschnitte 18.6.3.1 und 18.8.3), um das Datenbankschema beim DBMS abfragen zu können, z. B.:
  ```
  "Server=tcp:bernhard\SQLEXPRESS; Initial Catalog=Northwind; Integrated Security=true"
  ```
- Das zweite Kommandozeilenargument ist der Providername, z. B.:
  ```
  Microsoft.EntityFrameworkCore.SqlServer
  ```

Im gleich folgenden Beispiel werden noch die beiden folgenden Kommandozeilenargumente genutzt:

- Soll nur eine Auswahl der Datenbanktabellen in das EF Core - Modell aufgenommen werden, dann verwendet man das Kommandozeilenargument **-Tables**, z. B.:
  ```
  -Tables Customers, Orders
  ```
- Die per Reengineering automatisch erstellte Modell-Konfiguration wird per Voreinstellung über die **DbContext**-Methode **OnModelCreating()** realisiert. Mit dem Kommandozeilenargument
  ```
  -DataAnnotations
  ```
  wird angeordnet, dass die Konfiguration nach Möglichkeit über Annotationen erfolgen soll (vgl. Abschnitt 20.2.2.1).

20.4.2 Beispiel

Wir gehen von einem Konsolenprojekt für das Zielframework .NET 5.0 mit den im Abschnitt 20.1 beschriebenen NuGet-Paketinstallationen aus und arbeiten mit ...

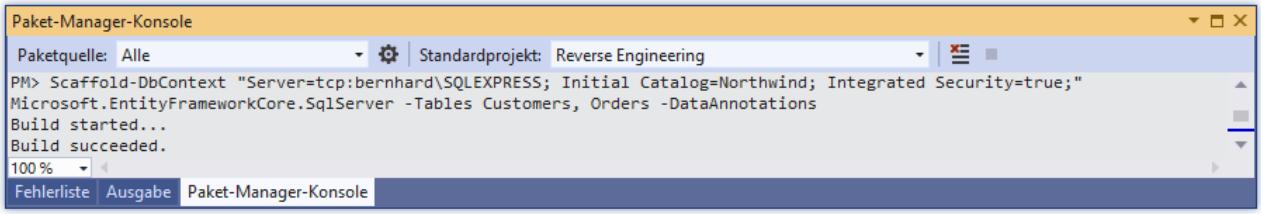
- dem Microsoft SQL Server Express (zur Installation des SQL-Servers siehe Abschnitt 18.8.1)
- und der Beispieldatenbank Northwind (zur Installation der Datenbank siehe Abschnitt 18.8.2.1).

Wir führen in der Paket-Manager-Konsole das folgende **Scaffold-DbContext** - Kommando aus:

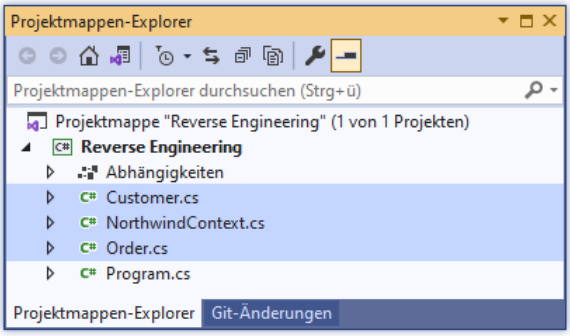
```
Scaffold-DbContext "Server=tcp:bernhard\SQLEXPRESS; Initial Catalog=Northwind;
Integrated Security=true" Microsoft.EntityFrameworkCore.SqlServer -Tables Customers,
Orders -DataAnnotations
```

Nach der erfolgreichen Ausführung

[1] https://www.learnentityframeworkcore.com/walkthroughs/existing-database



sind im **Projektmappen-Explorer** die **DbContext**-Ableitung `NorthwindContext` sowie die beiden Entitätsklassen `Customer` und `Order` vorhanden:



Das folgende Programm ermittelt per LINQ-to-Entities die deutschen Northwind-Kunden und listet sie mit ihren jeweiligen Aufträgen auf (zu LINQ siehe Kapitel 19 und Abschnitt 20.5):

```
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;

namespace Reverse_Engineering {
    class Program {
        static void Main() {
            using NorthwindContext context = new();

            var custColl = context.Customers
                           .Where(c => c.Country == "Germany")
                           .Include(c => c.Orders)
                           .ToList();

            foreach (var c in custColl) {
                Console.WriteLine(c.CustomerId + " " + c.CompanyName);
                foreach (var ord in c.Orders)
                    Console.WriteLine(" " + ord.OrderDate);
            }
        }
    }
}
```

Die Ausgabe startet mit:

```
ALFKI Alfreds Futterkiste
 25.08.1997 00:00:00
 03.10.1997 00:00:00
 13.10.1997 00:00:00
 15.01.1998 00:00:00
 16.03.1998 00:00:00
 09.04.1998 00:00:00
```

```
BLAUS Blauer See Delikatessen
 09.04.1997 00:00:00
 17.04.1997 00:00:00
 27.06.1997 00:00:00
 29.07.1997 00:00:00
 27.01.1998 00:00:00
 17.03.1998 00:00:00
 29.04.1998 00:00:00
```

20.4.3 Aktualisierungen des EF Core - Modells und der Datenbank

Auch in einer per Reverse Engineering erstellten Datenbankanwendung stehen irgendwann Änderungen beim EF Core - Modell bzw. beim Datenbankschema an. Man kann ...

- das Modell aktualisieren und die Migrationstechnik anwenden
- oder die Datenbank aktualisieren und das Reverse Engineering wiederholen.

Brind (2021) empfiehlt die Migrationstechnik.

20.4.3.1 Migrationstechnik

Nachdem das EF Core - Modell zu einer vorhandenen Datenbank durch das PMC-Kommando **Scaffold-DbContext** erstellt worden ist, kann man das Modell aktualisieren und per Migrationstechnik die Datenbank auf den aktuellen Stand bringen.

Zur Initialisierung der Migrationsverwaltung in der Datenbank sollte nach einer Empfehlung von Brind (2021) eine initiale Migration erstellt

```
PM> add-migration Base
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
```

und die **Up()** - Methode der **Migration**-Ableitung neutralisiert werden, z. B.:[1]

```
public partial class Base : Migration {
    protected override void Up(MigrationBuilder migrationBuilder) {
    }

    protected override void Down(MigrationBuilder migrationBuilder) {
        migrationBuilder.DropTable(
            name: "Orders");
        migrationBuilder.DropTable(
            name: "Customers");
    }
}
```

Diese Migration wird anschließend auf die Datenbank angewendet:

```
PM> update-database
Build started...
Build succeeded.
Applying migration '20210905130308_Base'.
Done.
```

Als Beispiel für eine Aktualisierung des Modells erweitern wir die Entität `Customer` um die Eigenschaft `CreditScore` (Kreditwürdigkeit):

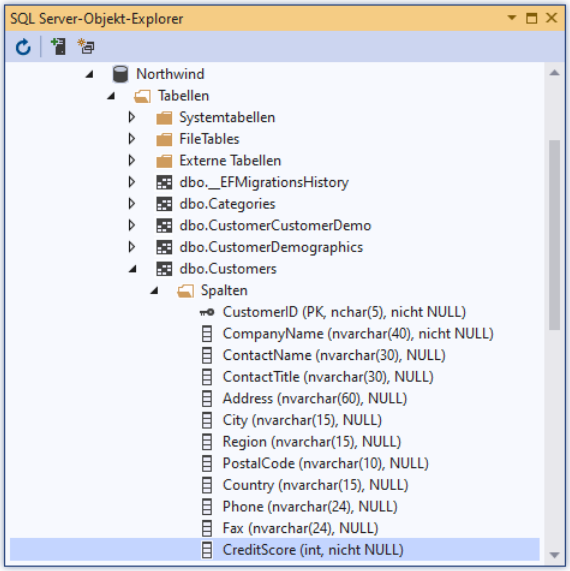
[1] https://www.learnentityframeworkcore.com/walkthroughs/existing-database

```
public partial class Customer {
 . . .
    public int CreditScore { get; set; }
. . .
}
```

Wir erstellen eine weitere Migration und wenden diese auf die Datenbank an:

```
PM> add-migration AddCreditScore
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM> update-database
Build started...
Build succeeded.
Applying migration '20210905130846_AddCreditScore'.
Done.
```

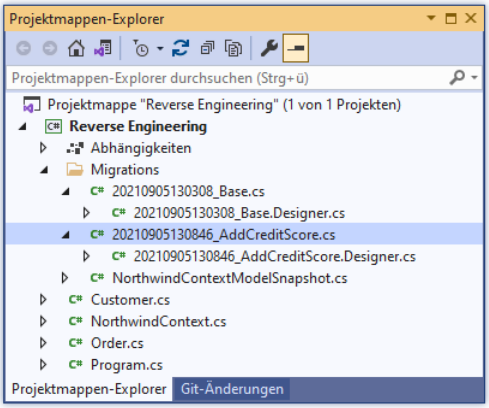
Mit dem **SQL Server -Objekt-Explorer** lässt sich anschließend verifizieren,



dass ...

- in der **Northwind**-Datenbank die zur Verwaltung der Migrationen erforderliche Tabelle **_EFMigrationsHistory** vorhanden ist,
- die `Customers`-Tabelle der Datenbank um die neue Spalte `CreditScore` erweitert wurde.

Wie der **Objektmappen-Explorer** zeigt, können im Projekt nach der Starthilfe durch das PMC-Kommando **Scaffold-DbContext** Aktualisierungen mit der Migrationstechnik genauso vorgenommen werden wie in einem durch den Code First - Ansatz erstellten Projekt:



20.4.3.2 Reverse Engineering wiederholen

Wird nach einer Änderung der Datenbank das PMC-Kommando **Scaffold-DbContext** erneut angewendet, dann werden alle EF Core - Klassen überschrieben. Durch geschickte Programmierung (z. B. mit Hilfe von partiellen Klassen) lassen sich die Verluste allerdings minimieren.

Die Datenbank Northwind besitzt im Vergleich zur Anwendung des **Scaffold-DbContext** - Kommandos im Abschnitt 20.4.1 in der Tabelle `Customers` mittlerweile die zusätzliche Spalte `CreditScore` (siehe Abschnitt 20.4.3.1). Um das EF Core Modell anzupassen, erweitern wir das **Scaffold-DbContext** - Kommando um die Option **-Force**

```
Scaffold-DbContext "Server=tcp:bernhard\SQLEXPRESS; Initial Catalog=Northwind;
Integrated Security=true" Microsoft.EntityFrameworkCore.SqlServer -Tables Customers,
Orders -DataAnnotations -Force
```

und führen es in der Paket-Manager-Konsole erneut aus.

20.5 Daten abfragen per LINQ-to-Entities

Das EF Core verwendet die LINQ-Abfragetechnik (siehe Kapitel 19) in der Spezialisierung LINQ-to-Entities, um Informationen aus einer Datenbank abzufragen. Der wesentliche Unterschied zu LINQ-to-Objects (siehe Kapitel 19) besteht darin, dass die LINQ-to-Entities - Abfragen vom EF Core - Provider in die Sprache der Datenbank (bei relationalen Datenbanken: in SQL) übersetzt werden, sodass wesentliche Teile der Abfragen vom DBMS ausgeführt werden können (siehe Abschnitt 20.5.3).

Implementiert eine Klasse die von **IEnumerable<TSource>** abgeleitete Schnittstelle **IQueryable<TSource>**, dann profitiert sie von LINQ-Erweiterungsmethoden in der statischen Klasse **Queryable** (im Namensraum **System.Linq**), z. B. von der filternden Methode **Where()**:

public static IQueryable<TSource> Where<TSource>(
this IQueryable<TSource> *quelle***,**
System.Linq.Expressions.Expression<Func<TSource, bool>> *bedingung***)**

Von der Eingabesequenz für die sortierenden Erweiterungsmethoden **OrderBy()**, **OrderByDescending**(), **ThenBy()** und **ThenByDescending()** wird verlangt, die von **IQueryable<TSource>** abgeleitete Schnittstelle **IOrderedQueryable<TSource>** zu implementieren, z. B:

public static IOrderedQueryable<TSource> OrderBy<TSource, TKey>(
this IOrderedQueryable<TSource> *source***,**
System.Linq.Expressions.Expression<Func<TSource, TKey>> *keySelector***)**

Im Vergleich zu den LINQ-to-Objects - Operatoren im Abschnitt 19.1.3 fällt vor allem der sperrige zweite Parameter vom Typ **Expression<Func<TSource, TKey>>** (aus dem Namensraum **System.Linq.Expression**) auf, mit dem wir uns im Abschnitt 20.5.2 beschäftigen werden.

Um die Entitätsklasse **TEntity** in ein EF Core - Modell aufzunehmen, wird in der **DbContext** - Ableitung eine automatisch implementierte Eigenschaft vom Typ der generischen Klasse **DbSet<TEntity>** deklariert (siehe Beispiel im Abschnitt 20.2.1.2). **DbSet<TEntity>** implementiert die Schnittstelle **IQueryable<TEntity>**, sodass die in der statischen Klasse **Queryable** definierten Erweiterungsmethoden genutzt werden können, um Entitäten aus den zum Kontext gehörenden Datenbanktabellen abzufragen. Die Quelle der Abfrage wird also über eine Eigenschaft der **DbContext** - Ableitung im EF Core - Modell angesprochen.

Weil **IQueryable<T>** aus **IEnumerable<T>** abgeleitet ist, implementiert **DbSet<TEntity>** auch die Schnittstelle **IEnumerable<T>**. Bei der Wahl der auszuführenden Erweiterungsmethode verfährt der Compiler nach dem Prinzip der größten Spezifität.

20.5.1 Einfache Abfragen

Vor der nicht ganz trivialen Beschäftigung mit LINQ-to-Entities vergewissern wir uns anhand von einfachen Beispielen, dass es sich um eine effektive und bequeme Datenzugriffstechnologie handelt.

20.5.1.1 Einzelne Entitäten abrufen

Im ersten Beispiel wird mit der **Queryable**-Erweiterungsmethode **FirstOrDefault()** und einem Prädikat als Parameter aus der `Blogs`-Tabelle der im Abschnitt 20.3.1 erstellten Datenbank die Entität mit dem Primärschlüsselwert 2 abgerufen:

```
var vsBlog = context.Blogs.FirstOrDefault(b => b.BlogId == 2);
if (vsBlog != null)
  Console.WriteLine(vsBlog.Url);
```

Diese Abfrage ist bequemer und potentiell effizienter mit der generischen **DbContext**-Methode **Find<TEntity>()**

public virtual TEntity Find<TEntity>(params Object[] *keyValues***) where TEntity : class**

zu realisieren:[1]

```
var vsBlog = context.Blogs.Find(2);
```

Während **FirstOrDefault()** auf jeden Fall die Datenbank befragt, überprüft **Find<TEntity>()**, ob die gesuchte Entität bereits im Rahmen der Änderungsnachverfolgung (siehe Abschnitt 20.6) zwischengespeichert worden ist und vermeidet eventuell einen Datenbankkontakt. Befindet sich die gesuchte Entität weder im Zwischenspeicher noch in der Datenbank, dann liefert **Find<TEntity>()** die Rückgabe **null**.

20.5.1.2 Mehrere Entitäten abrufen

In der folgenden Anweisung werden mit Hilfe der **Enumerable**-Erweiterungsmethode **ToList()** *alle* `Blogs`-Zeilen bzw. -Entitäten aus der im Abschnitt 20.3.1 erstellten Datenbank in eine Kollektion vom Typ **List<Blog>** befördert:

```
var blogs = context.Blogs.ToList();
foreach (var e in blogs)
    Console.WriteLine("Id: " + e.BlogId + ", URL: " + e.Url);
```

Im nächsten Beispiel werden die Treffer mit Hilfe der Methode **Where()** eingeschränkt:

```
var xBlogs = context.Blogs
    .Where(b => b.Url.Contains("xamarin"))
    .ToList();
foreach (var e in xBlogs)
    Console.WriteLine("Id: " + e.BlogId + ", URL: " + e.Url);
```

Die Projektion einer Entität auf eine eingeschränkte Instanz (z. B. auf ein Objekt einer definierten oder anonymen Klasse mit wenigen Eigenschaften) steht nicht ganz im Einklang mit dem ORM-Konzept. Daher taucht in den Einstiegsbeispielen der **Select()** - Abfrageoperator *nicht* auf (siehe Abschnitt 20.5.2).

Übrigens enthält auch die folgende Anweisung eine Abfrage:

```
var allBlogs = context.Blogs;
```

Es ist kein terminierender LINQ-Operator vorhanden, sodass erst beim Iterieren per **foreach**-Schleife ein Datenbankzugriff stattfindet:

---

[1] Beim Aufruf einer generischen Methode kann aufgrund der Fähigkeiten des Compilers zur Typinferenz die Konkretisierung des Typformalparameters meist weggelassen werden.

```
foreach (var e in allBlogs)
    Console.WriteLine("Id: " + e.BlogId + ", URL: " + e.Url);
```

20.5.2 Ausdrucksbäume

Als zweiter Parameter der zu Beginn von Abschnitt 20.5 vorgestellten Überladung der **Queryable**-Erweiterungsmethode **Where()**

public static System.Linq.IQueryable<TSource> Where<TSource>(
this System.Linq.IQueryable<TSource> *source*,
System.Linq.Expressions.Expression<Func<TSource, bool>> *predicate*)

ist ein Objekt der Klasse **Expression<TDelegate>** zu übergeben. Hier ist explizit ein **Lambda-Ausdruck** verlangt, der einen speziellen Delegatentyp zu erfüllen hat:

public sealed class Expression<TDelegate> : System.Linq.Expressions.LambdaExpression

Beim Aufruf der **Queryable**-Erweiterungsmethode **Where()** übergibt man einen Lambda-Ausdruck, den der Compiler in ein **Expression<Func<TSource, bool>>** - Objekt verpackt, z. B.:

```
var blog = context.Blogs
    .Where(b => b.Url.Contains("xamarin"))
    .OrderBy(b => b.BlogId)
    .First();
```

Ein **Expression<Func<TSource, bool>>** - Objekt ist *kein* ausführbares Delegatenobjekt, sondern der **Ausdrucksbaum** zum Lambda-Ausdruck, der Variablen, Operatoren, Methodenaufrufe etc. enthält und die Funktion des Lambda-Ausdrucks beschreibt. Man kann den Ausdrucksbaum durch einen Aufruf seiner **Compile()** - Methode in ein ausführbares Delegatenobjekt übersetzen, z. B.:

```
System.Linq.Expressions.Expression<Func<Blog, bool>> expr =
    b => b.Url.Contains("xamarin");
Func<Blog, bool> deleg = expr.Compile();
Console.WriteLine("URL enthält xamarin: " + deleg(blog));
```

Während ...

- aus einem **Func<TSource, bool>** - Parameter der **Enumerable**-Erweiterungsmethode **Where()** schlussendlich ausführbarer IL-Code resultiert,
- entsteht aus dem **Expression<Func<TSource, bool>>** - Parameter der **Queryable**-Erweiterungsmethode **Where()** der Ausdrucksbaum zum übergebenen Lambda-Ausdruck, der die Wirkungsweise des Lambda-Ausdrucks beschreibt.

Die beiden **Where()** - Aufrufe sehen identisch aus, doch erstellt der Compiler daraus komplett verschiedene Objekte.

Ein Ausdrucksbaum wird zur Laufzeit von einem EF Core - Provider nach Möglichkeit z. B. in SQL-Anweisungen übersetzt, die vom zugehörigen DBMS ausgeführt werden. Enthält ein Lambda-Ausdruck Methodenaufrufe, dann sind aber nicht unbedingt passende server-seitige Methoden zu finden.

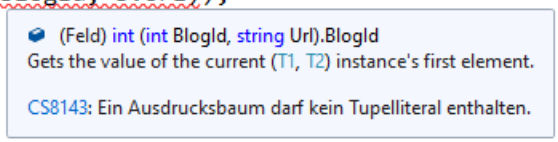
Mit dem LINQ-Operator **Select()** wird oft eine Projektion auf einen explizit definierten oder einen anonymen Typ mit einer sparsamen Eigenschaftsausstattung vorgenommen. Das klappt auch mit der **Queryable**-Variante von **Select()** und kann das Transportvolumen bei einer Datenbankabfrage reduzieren.[1]

---

[1] Die Projektion einer Entität auf eine eingeschränkte Instanz (z. B. auf ein Objekt einer definierten oder anonymen Klasse mit wenigen Eigenschaften) steht nicht ganz im Einklang mit dem ORM-Konzept.

Der syntaktisch bequeme, mit C# 7.0 eingeführte Tupeltyp (vgl. Abschnitt 6.6) wird von der **Queryable**-Methode **Select()** *nicht* unterstützt, z. B.:

```
var blogs = context.Blogs
    .Select(b => (b.BlogId, b.Url));
```



Der in C# 9.0 eingeführte Record-Typ (vgl. Abschnitt 6.7) ist hingegen erlaubt, z. B.:

```
record BlogRec(int BlogId, string Url);
 . . .
var blogRec = context.Blogs
    .Select(b => new BlogRec(b.BlogId, b.Url));
```

20.5.3 Server- vs. klientenseitige Auswertung

Das EF Core versucht, eine per LINQ-to-Entities formulierte Abfrage so weit wie möglich vom EF Core - Provider in SQL übersetzen zu lassen, damit das DBMS die Auswertung vornehmen kann. Um eine Vorstellung vom SQL-Code zu einem **IQueryable<T>** - Objekt zu erhalten, lässt man den Code von der Methode **ToQueryString()** in eine **String**-Variable schreiben, z. B.:

Quellcodesegment	Ausgabe

`using Microsoft.EntityFrameworkCore;`
`. . .`
`var blogs = context.Blogs`
`.OrderBy(b => b.BlogId)`
`.Select(b => new { b.BlogId, b.Url });`
`Console.WriteLine(blogs.ToQueryString());`
	`SELECT [b].[BlogId], [b].[Url]`
	`FROM [Blogs] AS [b]`
	`ORDER BY [b].[BlogId]`

Um die Erweiterungsmethode **ToQueryString()** der Klasse **EntityFrameworkQueryable-Extensions** nutzen zu können, muss der Namensraum **Microsoft.EntityFrameworkCore** importiert werden.

Das im Beispiel resultierende **SELECT**-Kommando ist mit den folgenden Hinweisen leicht zu verstehen:

- Den Spaltennamen (z. B. `BlogID`) wird in der **ToQueryString()** - Rückgabe generell der Tabellen-Aliasname vorangestellt.
- Bei der Vereinbarung eines Aliasnamens für die Tabelle wird das optionale Schlüsselwort **AS** verwendet.
- Alle Namen werden durch eckige Klammern begrenzt, obwohl das in SQL nur dann erforderlich ist, wenn ein Name Schlüsselwörter, Sonderzeichen oder Leerzeichen enthält.

Sind in der *finalen Projektion* (aufgrund eines **Select()** - Abfrageoperators) Verarbeitungsschritte erforderlich, die das DBMS *nicht* ausführen kann (z. B. wegen eines Methodenaufrufs), dann finden diese Verarbeitungsschritte *lokal* statt.

Wenn eine LINQ-to-Entities - Abfrage *vor* der finalen Projektion vom DBMS nicht realisierbare Verarbeitungsschritte enthält (z. B. aufgrund eines Methodenaufrufs im Rahmen eines **Where()** - Abfrageoperators), dann wirft das EF Core eine Ausnahme, z. B.:

```
Unhandled exception. System.InvalidOperationException: The LINQ expression
'DbSet<Blog>().Where(b => Program.StandardizeUrl(b.Url).Contains("xamarin"))' could
not be translated.
```

Neben Methodenaufrufen in Lambda-Ausdrücken können auch komplexe LINQ-Operatoren (z. B. **Join()**, **GroupJoin()**, **SelectMany()**, **GoupBy()**) dafür verantwortlich sein, dass es nicht (für jeden EF Core - Provider) möglich ist, eine Übersetzung in server-seitig ausführbare SQL-Anweisungen vorzunehmen, z. B.:

```
Unhandled exception. System.InvalidOperationException: The LINQ expression
'DbSet<Blog>()
    .GroupJoin(
        inner: DbSet<Post>(),
        outerKeySelector: b => b.BlogId,
        innerKeySelector: p => p.PostId,
        resultSelector: (b, grouping) => new {
            b = b,
            grouping = grouping
         })' could not be translated.
```

Zu den nicht in SQL übersetzbaren LINQ-Operatoren gehören auch die **Select()** - und **Where()** - Überladungen, die im Delegatentyp für den Ausdrucksbaum einen Parameter für die Indexposition vorschreiben, z. B.:

public static IQueryable<TResult> Select<TSource, TResult>(
this System.Linq.IQueryable<TSource> *source*,
Expression<Func<TSource, int, TResult>> *selector*)

public static IQueryable<TSource> Where<TSource>(
this System.Linq.IQueryable<TSource> *source*,
Expression<Func<TSource, int, bool>> predicate)

Während bei LINQ-to-Objects ein Enumerator die Reihenfolge der gelieferten Elemente bestimmt, nimmt ein DBMS bei einer **SELECT**-Abfrage interne Optimierungen vor, sodass für die gelieferten Elemente in der Regel keine Reihenfolge und damit auch keine Indexposition definiert ist (Griffiths 2020).

Eine vollständige Beschreibung der in LINQ-to-Entities (nicht) unterstützten Abfragemethoden bietet Microsoft hier:

https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/language-reference/supported-and-unsupported-linq-methods-linq-to-entities

Scheitert die Übersetzung einer LINQ-Anfrage in SQL an einem *Methodenaufruf*, dann kann prinzipiell dafür gesorgt werden, dass die server-seitige Ausführung der Abfrage doch möglich wird:[1]

- Mit Hilfe des Datenbankverwaltungssystems wird eine SQL-Funktion für die Datenbank definiert,
- und in der EF Core - Modellkonfiguration wird die SQL-Funktion auf eine C# - Methode abgebildet.

Das EF Core blockiert eine Abfrage, die nicht weitgehend (von der terminalen Projektion abgesehen) server-seitig ausgeführt werden kann, per Voreinstellung durch das Werfen einer Ausnahme, weil für eine klientenseitige Auswertung potentiell ein großes Datenvolumen von der Datenbank in den Hauptspeicher übertragen werden müsste. Die Ausnahme wird nicht bei der Deklaration der Abfrage geworfen, sondern erst bei der Ausführung, die z. B. durch eine Enumeration angefordert wird.

Um diese Voreinstellung (z. B. wegen eines erträglichen Datenvolumens) außer Kraft zu setzen, kann die Erweiterungsmethode **AsEnumerable()** eingesetzt werden, z. B.:[2]

---

[1] https://docs.microsoft.com/en-us/ef/core/querying/user-defined-function-mapping

[2] https://docs.microsoft.com/en-us/ef/core/querying/client-eval

```
var xBlogs = context.Blogs
    .AsEnumerable()
    .Where(b => StandardizeUrl(b.Url).Contains("xamarin"));
```

AsEnumerable() ändert lediglich den Typ der Abfrage in **IEnumerable<T>**, sodass die nachfolgenden LINQ-Operatoren in **Enumerable**-Erweiterungsmethoden aufgelöst, also lokal ausgeführt werden:[1]

```
public static IEnumerable<TSource> AsEnumerable<TSource>(
     this IEnumerable<TSource> source) => source;
```

Wenn eine aufgrund von lokal auszuführenden Operatoren aufwändige Abfrage eventuell wiederholt werden muss, dann sollte eine von den **Enumarable**-Erweiterungsmethoden **ToArray()** oder **ToList()** verwendet werden, um das Ergebnis zu speichern, z. B.:

```
var xBlogs = context.Blogs
    .AsEnumerable()
    .Where(b => StandardizeUrl(b.Url).Contains("xamarin"))
    .ToList();
```

20.5.4 Laden zugehöriger Entitäten

Das Laden der mit einer Entität über Navigationseigenschaften verknüpften Entitäten kann im EF Core auf unterschiedliche Weise realisiert werden:

- Vorsorgliches (eifriges) Laden durch die initiale Abfrage (engl.: *eager loading*)
- Explizites Laden
- Automatisches Laden beim Zugriff (engl.: *lazy loading*)

20.5.4.1 Laden durch die initiale Abfrage

Werden beim Laden einer Entität alle über Navigationseigenschaften assoziierten Entitäten ebenfalls geladen, dann steht anschließend ohne weitere Datenbankzugriffe ein ganzer Objekt-Graph (eine Menge assoziierter Objekte) zur Verfügung. Wenn dieser Graph umfangreich ist und viele schlussendlich nicht benötigte Daten enthält, dann leidet allerdings die Performanz.

Am einfachsten lässt sich das initiale Laden zugehöriger Entitäten (engl.: *eager loading*) durch die **EntityFrameworkQueryableExtensions**-Erweiterungsmethode **Include()** veranlassen. Im folgenden Beispiel werden die über eine Kollektions-Navigationseigenschaft erreichbaren Entitäten einbezogen:

```
using Microsoft.EntityFrameworkCore;
 . . .
var xFirst = context.Blogs
                        .Include(b => b.Posts)
                        .Where(b => b.Url.Contains("xamarin"))
                        .First();
```

Wird im Rahmen einer per **Select()** vorgenommenen Projektion auf eine anonyme Klasse eine Navigationseigenschaft explizit angesprochen, dann werden die zugehörigen Entitäten (*ohne* **Include()**) im Rahmen der initialen Abfrage geladen, z. B.:

```
var xFirstA = context.Blogs
                        .Where(b => b.Url.Contains("xamarin"))
                        .Select (b => new { b.Url, b.Posts })
                        .First();
```

---

[1] Der Quellcode zur Klasse **Enumerable** in der BCL zu .NET 5.0 kann über die folgende Webseite inspiziert werden: https://source.dot.net/
Zur Definition der Methode **AsEnumerable()** wird ein Lambda-Operator verwendet (siehe Abschnitt 5.7.3).

Ein ergänzter **Include()** - Aufruf wäre in dieser Situation ohne Einfluss auf die erzeugte SQL-Syntax und damit nutzlos (Brind 2021).

Außerdem kann per Projektion das übertragene Datenvolumen reduziert werden, z. B.:

```
var xFirstAR = context.Blogs
                          .Where(b => b.Url.Contains("xamarin"))
                          .Select (b => new {
                                                b.Url,
                                                Posts = b.Posts.Select(p => p.Title) })
                          .First();
```

Allerdings steht eine Projektion nicht im Einklang mit dem ORM-Grundprinzip.

Im folgenden Beispiel wird die zu einer Referenz-Navigationseigenschaft gehörige Entität geladen:

```
var xPost = context.Posts
                .Include(p => p.Blog)
                .Where(p => p.Title.Contains("xamarin"))
                .First();
```

Weitere Optionen für das initiale Laden zugehöriger Entitäten werden in der Online-Dokumentation zum EF Core behandelt, z. B.:[1]

- Berücksichtigung von mehreren Master-Details - Beziehungen durch eine Sequenz von **Include()** - Aufrufen
- Unterstützung von Beziehungen auf mehr als zwei Ebenen mit Hilfe der Erweiterungsmethode **ThenInclude()**

20.5.4.2 Explizites Laden

Das EF Core bietet die Möglichkeit, die mit einer Entität über Navigationseigenschaften assoziierten Entitäten explizit zu laden, wobei ein zusätzlicher Datenbankzugriff stattfindet. Zu einer Entität, die z. B. über die Erweiterungsmethode **Single()** aus der Datenbank abgerufen worden ist,

```
var blog = context.Blogs.First();
```

erhält man über die **DbContext**-Methode **Entry()** ein Objekt der Klasse **EntityEntry<TEntity>**. Über seine generische Methode **Collection<TProperty>()**, die als Parameter einen Ausdruck akzeptiert, der eine Kollektions-Navigationseigenschaft benennt,

public virtual CollectionEntry<TEntity,TProperty> Collection<TProperty>(
Expression<Func<TEntity, IEnumerable<TProperty>>> *propertyExpression*)
where TProperty : class

erhält man ein **CollectionEntry<TEntity, TProperty>** - Objekt (aus dem Namensraum **Microsoft.EntityFrameworkCore.ChangeTracking**). Dieses Objekt kann mit seiner Methode **Load()** beauftragt werden, die von der Navigationseigenschaft referenzierten Entitäten zu laden, z. B.:

```
context.Entry(blog).Collection(b => b.Posts).Load();
Console.WriteLine("Posts zum Blog: " + blog.Url);
foreach (var p in blog.Posts)
    Console.WriteLine(p.Title);
```

Das Laden der per Referenz-Navigationseigenschaft erreichbaren Entität verläuft analog:

```
var post = context.Posts.First();
context.Entry(post).Reference(p => p.Blog).Load();
Console.WriteLine($"\nDer Post \"{post.Title}\" gehört zum Blog {post.Blog.Url}");
```

---

[1] https://docs.microsoft.com/en-us/ef/core/querying/related-data/eager

20.5.4.3 Automatisches Laden beim Zugriff

Das automatische Laden zugehöriger Entitäten beim Zugriff (engl. *lazy loading*) ist wegen seiner Auswirkung auf die Performanz durch häufige Datenbankzugriffe in Misskredit geraten und wird vom EF Core 5.0 *nicht* per Voreinstellung unterstützt. Man kann das lazy loading aber aktivieren und hat dabei zwei Optionen:[1]

- Master-Entitäts - Definitionen, die dem EF Core zur Laufzeit die Definition von abgeleiteten Proxy-Klassen (dt.: *Stellvertreterklassen*) mit der ergänzten Kompetenz zum automatischen Laden erlauben
- Master-Entitäts - Definitionen mit einem privaten, per Konstruktor initialisierten Member-Objekt, das die Schnittstelle **ILazyLoader** oder den Delegatentyp **Action<Object, String>** erfüllt und die Kompetenz zum automatischen Laden besitzt

Wir beschränken uns anschließend auf die etwas einfachere Stellvertretertechnik. Um sie in einem Projekt verwenden zu können, muss das NuGet-Paket

Microsoft.EntityFrameworkCore.Proxies

installiert werden.

In der Konfigurationsmethode **OnConfiguring()** zur **DbContext**-Ableitung wird das lazy loading durch einen Aufruf der **DbContextOptionsBuilder**-Methode **UseLazyLoadingProxies()** aktiviert:

```
protected override void OnConfiguring(DbContextOptionsBuilder options) {
    options.UseSqlServer("Server=(localdb)\\MSSQLLocalDB; " +
    "Initial Catalog=DepLoad; Integrated Security=true");
    options.UseLazyLoadingProxies();
}
```

Für die beteiligten Entitäten müssen die folgenden Bedingungen erfüllt sein:

- Die Navigationseigenschaften in der Master- und in der Details-Klasse müssen als überschreibbar (**virtual**) deklariert sein.
- Die beiden Entitätsklassen dürfen nicht versiegelt sein.

Unter diesen Voraussetzungen kann das EF Core zur Laufzeit zu einem Entitäts-Objekt ein Proxy-Objekt erzeugen, das die Kompetenz besetzt, bei Bedarf die über Navigationseigenschaften verknüpften Entitäten abzufragen. Im folgenden Beispiel sind die beiden Voraussetzungen erfüllt:

```
public class Blog {
    public int BlogId { get; set; }
    public string Url { get; set; }

    public virtual List<Post> Posts { get; } = new List<Post>();
}

public class Post {
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public virtual Blog Blog { get; set; }
}
```

Zur Demonstration verwenden wir die Syntax aus dem Abschnitt 20.5.4.2 über das explizite Laden und lassen dabei die Anweisungen mit den **Load()** - Aufrufen weg:

---

[1] https://docs.microsoft.com/en-us/ef/core/querying/related-data/lazy
https://www.learnentityframeworkcore.com/lazy-loading

```
// Kollektions-Navigationseigenschaft
var blog = context.Blogs.First();
Console.WriteLine("Posts zum Blog: " + blog.Url);
foreach (var p in blog.Posts)
    Console.WriteLine(p.Title);

// Referenz-Navigationseigenschaft
var post = context.Posts.First();
Console.WriteLine($"\nDer Post \"{post.Title}\" gehört zum Blog {post.Blog.Url}");
```

20.5.5 Materialisieren vs. Iterieren

Das im Abschnitt 19.1.2 für LINQ-to-Objects beschriebene Prinzip der verzögerten Ausführung gilt auch für LINQ-to-Entities - Abfragen, sofern ein Ergebnis vom Typ **IQueryable<T>** geliefert wird. Einige Erweiterungsmethoden aus der Klasse **Queryable** bewirken aber die *sofortige* Ausführung:

- Erweiterungsmethoden, die einzelne Entitäten liefern: **First()**, **FirstOrDefault()**, **Single()**, **SingleOrDefault()**
- Erweiterungsmethoden, die eine Auswertung vornehmen: **Count()**, **Sum()** etc.

Weil **IQueryable<T>** eine Spezialisierung der Schnittstelle **IEnumerable<T>** ist, können auf ein **IQueryable<T>** - Objekt auch die von der Klasse **Enumerable** angebotenen Erweiterungsmethoden angewendet werden, die eine Kollektion bzw. einen Array mit Entitäten liefern und sofort ausgeführt werden: **ToList()**, **ToArray()**

In der Praxis wird bei LINQ-to-Entities - Abfragen eher selten mit einer Ergebnissequenz (also mit dem Ergebnistyp **IQueryable<T>**) gearbeitet, sodass ...

- im Sinne der verzögerten Ausführung erst beim Iterieren über die Ergebnissequenz (meist im Rahmen einer **foreach**-Schleife) ein Datenbankzugriff stattfindet,
- wobei pro Schleifendurchgang genau eine Entität angesprochen werden kann.

LINQ-to-Entities - Abfragen enden häufiger mit einer „materialisierenden“ Operation wie **ToList()** oder **SingleOrDefault()**, sodass eine Referenz auf eine Kollektion von Entitäten oder auf eine einzelne Entität zur Verfügung steht.

Microsoft betont die unterschiedlichen Effekte der beiden Abfragetechniken (bezeichnet als *streaming query* bzw. *buffering query*) auf den Speicherbedarf:[1]

> In principle, the memory requirements of a streaming query are fixed - they are the same whether the query returns 1 row or 1000; a buffering query, on the other hand, requires more memory the more rows are returned. For queries that result large resultsets, this can be an important performance factor.

Allerdings kann die Argumentation nicht überzeugen, weil zumindest bei einer Abfrage mit aktivierter Änderungsnachverfolgung (siehe Abschnitt 20.6.2) auch beim Iterieren über ein **IQueryable<T>** - Ergebnis zu jedem Element in der Ergebnissequenz eine Entität entsteht, die referenziert ist, also vom Garbage Collector nicht abgeräumt werden kann. Dass beim Iterieren die „Rohdaten“ nicht gleichzeitig, sondern sukzessive vom DBMS bezogen werden, sollte den gesamten Speicherbedarf nur dann wesentlich beeinflussen, wenn ...

- keine Änderungsnachverfolgung stattfindet,
- das Iterieren (z.B. **foreach**-Schleife) vorzeitig abgebrochen wird.

Eine Abfrage ohne Änderungsnachverfolgung resultiert dann (siehe Abschnitt 20.6.2), wenn

---

[1] https://docs.microsoft.com/en-us/ef/core/performance/efficient-querying
Mit *streaming* ist der sukzessive Datentransfer beim Iterieren gemeint.

- die Änderungsnachverfolgung explizit deaktiviert wird,
- aufgrund von Projektionen keine vollständigen Entitäten geladen werden.

Weil sich viele Datenbankzugriffe auf das Lesen beschränken, kann oft auf die Änderungsnachverfolgung verzichtet werden.

20.5.6 Sofortige asynchrone Ausführung

Für sofort auszuführende LINQ-Operatoren (z. B. **SingleOrDefault()**, **Count()**, **ToList()**) enthält die statische Klasse **EntityFrameworkQueryableExtensions** asynchron auszuführende Alternativen als Erweiterungsmethoden (z. B. **SingleOrDefaultAsync()**, **CountAsync()**, **ToListAsyc()**). Man erhält ein **Task<T>** - Objekt zur Verwendung im Rahmen der aufgabenbasierten asynchronen Programmierung (vgl. Abschnitte 17.4 und 17.5), z. B.:

public static Task<List<TSource>> ToListAsync<TSource>(
this IQueryable<TSource> *source*,
CancellationToken *cancellationToken* **= default)**

Zu der im Abschnitt 20.5.1.1vorgestellten Methode **Find<TEntity>()** enthält die Klasse **DbContext** auch asynchron arbeitende Alternativen, z.B. die folgende Überladung der Methode **FindAsync<TEntity>**:

public virtual System.Threading.Tasks.ValueTask<TEntity> FindAsync<TEntity> (
params object[] *keyValues***) where TEntity : class**

20.5.7 Sonstige Optionen

20.5.7.1 Globale Abfragefilter

Das EF Core erlaubt die Vereinbarung von globalen Abfragefiltern für Entitätsklassen, die bei jeder Abfrage automatisch berücksichtigt werden. Wichtige Anwendungen für diese Technik sind:

- Vorläufiges Löschen (engl.: *soft delete*)
  Um eine Tabellenzeile (eine Entität) als gelöscht markieren zu können, ohne sie endgültig aus der Datenbank zu entfernen, definiert man für eine Entitätsklasse eine Eigenschaft namens **IsDeleted**.[1]
- Mehrmandanten-Datenbank(tabellen)
  Damit eine Datenbank(tabelle) von mehreren Mandanten genutzt werden kann, definiert man für eine betroffene Entitätsklasse eine Eigenschaft namens **TenantId** zur Aufnahme der Mandantenkennung.

Mit Hilfe von globalen Abfragefiltern kann man dafür sorgen, dass keine Entitäten aus Tabellenzeilen entstehen, die als gelöscht markiert sind oder zu fremden Mandanten gehören. Die globalen Filter werden in der **DbContext**-Methode **OnModelCreating()** vereinbart, z. B.:[2]

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Post>().HasQueryFilter(p => !p.IsDeleted);
}
```

---

[1] Details zum vorläufigen Löschen im EF Core finden sich z. B. hier:
https://www.thereformedprogrammer.net/ef-core-in-depth-soft-deleting-data-with-global-query-filters/

[2] Das Beispiel stammt von der Webseite:
https://docs.microsoft.com/de-de/ef/core/querying/filters

20.5.7.2 Direkte Ausführung von SQL-Kommandos

Wenn sich eine Abfrage nicht in LINQ formulieren lässt, oder wenn sich der zu einer LINQ-Abfrage generierte SQL-Code als ineffektiv erweist, dann kommt bei einer relationalen Datenbank ein mit Hilfe der Erweiterungsmethode **FromSqlRaw()** aus der statischen Klasse **RelationalQueryableExtensions** (im Namensraum **Microsoft.EntityFrameworkCore**) an die Datenbank gesendetes SQL-Kommando in Frage, sofern der EF Core - Provider dies unterstützt.[1]

20.6 Änderungsnachverfolgung

Das EF Core besitzt eine per Voreinstellung aktive Änderungsnachverfolgung (engl.: einen *change tracker*) mit folgenden Leistungen:

- Nachverfolgung von Änderungen bei Entitäten
- Identitätsauflösung für Entitäten

Zuständig für die Änderungsnachverfolgung ist das über die **ChangeTracker**-Eigenschaft der aktuellen **DbContext**-Instanz referenzierte **DbChangeTracker**-Objekt.

20.6.1 Beginn und Terminierung der Änderungsnachverfolgung

Anlässe für die Aufnahme einer Entität in die Änderungsnachverfolgung der aktuellen **DbContext**-Instanz sind:

- Eine Datenbankabfrage wird ausgeführt.
  Die Aufnahme einer abgefragten Entität findet statt:
  - bei der direkten Ausführung einer Abfrage, die eine einzelne Entität oder eine Kollektion von Entitäten liefert (z. B. mit **First()** oder **ToList()** als Abfrageoperator).
  - bei der Iteration über die **IQueryable<TResult>** - Rückgabe einer Abfrage mit verzögerter Ausführung (meist im Rahmen einer **foreach**-Schleife)

  Ein per *Projektion* entstehendes Objekt wird *nicht* in die Änderungsnachverfolgung einbezogen.
- Eine neu erstellte, in der Datenbank noch nicht vorhandene, Entität wird über die **DbSet<TEntity>** - Methode **Add()** zusammen mit den per Navigationseigenschaft erreichbaren Entitäten in die Änderungsnachverfolgung aufgenommen (siehe Abschnitt 20.7).
- Eine in der Datenbank vorhandene Entität wird über die **DbSet<TEntity>** - Methode **Attach()** zusammen mit den per Navigationseigenschaft erreichbaren Entitäten in die Änderungsnachverfolgung aufgenommen (siehe Abschnitt 20.6.6). Das ist z. B. sinnvoll bei Entitäten, die von einem **DbContext**-Objekt abgefragt worden sind, das mittlerweile einen **Dispose()** - Aufruf erhalten hat.

Eine Entität wird auch dann in die Änderungsnachverfolgung einbezogen, wenn sie über eine Navigationseigenschaft einer unter Beobachtung stehenden Entität erreichbar ist.

Anlässe für die *Terminierung* der Änderungsnachverfolgung für eine Entität sind:

- Das **DbContext**-Objekt erhält einen **Dispose()** - Aufruf.
- Durch einen Aufruf der **ChangeTracker**-Methode **Clear()** wird die Nachverfolgung für alle Entitäten beendet.
- Der Status einer Entität (siehe Abschnitt 20.6.5) wird auf **Detached** gesetzt, z. B.:

```
context.Entry(dotNetBlog).State = EntityState.Detached;
```

[1] https://docs.microsoft.com/en-us/ef/core/querying/raw-sql

20.6.2 Nachverfolgung von Änderungen bei Entitäten

An einer abgefragten Entität vorgenommene Änderungen werden per Voreinstellung beim Aufruf der **DbContext**-Methode **SaveChanges()** in die Datenbank übertragen, z. B.:

```
xPost.Content = "Xamarin.Forms wird in MAUI überführt";
context.SaveChanges();
```

Dazu findet ein Vergleich mit dem zum Zeitpunkt der Abfrage konservierten Ausgangszustand statt.

Wenn eine Änderungsnachverfolgung überflüssig ist (z. B. bei einem ausschließlich lesenden Datenbankzugriff), dann sollte sie zur Vermeidung des damit verbundenen Aufwands abgeschaltet werden. Das kann mit Wirkung für eine konkrete Abfrage durch einen Aufruf der **Queryable**-Erweiterungsmethode **AsNoTracking()** geschehen, z. B.:

```
List<Blog> blogs = context.Blogs.AsNoTracking().ToList();
```

Um für die aktuelle **DbContext**-Instanz die Änderungsnachverfolgung generell abzuschalten, setzt man die Eigenschaft **QueryTrackingBehavior** des **ChangeTracker**-Objekts zum Kontext auf den Wert **NoTracking** der Enumeration **QueryTrackingBehavior**:

```
context.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;
```

20.6.3 Identitätsauflösung

Bei aktiver Änderungsnachverfolgung verhindert das EF Core, dass durch Datenbankabfragen im Rahmen einer Kontext-Instanz (innerhalb einer Arbeitseinheit) zu einer Tabellenzeile (also zu einem Primärschlüsselwert) mehr als ein in die Änderungsnachverfolgung einbezogenes Objekt der zugehörigen Entitätsklasse entsteht. Resultiert aus einer Abfrage eine Entität mit einem im aktuellen Kontext bereits vorhandenen Primärschlüsselwert, dann wird ...

- die Adresse der vorhandenen Entität geliefert
- die vorhandene Entität *nicht* verändert

Zwischenzeitlich in der Datenbank an der zugehörigen Tabellenzeile vorgenommene Änderungen wirken sich also *nicht* aus.

Eine einfache Möglichkeit, auf jeden Fall den aktuellen Zustand der Datenbank in einer Abfrage zu erhalten, besteht in der Verwendung einer neuen **DbContext**-Instanz. An dieser Stelle soll noch einmal daran erinnert werden, dass die **DbContext**-Instanzen im EF Core eher für eine zeitlich begrenzte Verwendung konzipiert sind.[1]

Im folgenden Beispiel liefern zwei Abfragen dasselbe Objekt:

```
var post = context.Posts.Single(p => p.PostId == 5);
var xPost = context.Posts.Where(p => p.Title.Contains("xamarin")).First();
Console.WriteLine(post == xPost); //true
```

Bei einer Abfrage mit deaktivierter Änderungsnachverfolgung (siehe Abschnitt 20.6.2) findet *keine* Identitätsauflösung statt, sodass zu einer Tabellenzeile mehrere Entitäten entstehen können. Seit der EF Core - Version 5.0 ist aber auch bei abgeschalteter Änderungsnachverfolgung eine Identitätsauflösung möglich, sodass aus einer Tabellenzeile nur *eine* Entität entstehen kann. Das wird für eine konkrete Abfrage durch einen Aufruf der **Queryable**-Erweiterungsmethode **AsNoTrackingWithIdentityResolution()** veranlasst, z. B.:

```
List<Blog> blogs = context.Blogs.AsNoTrackingWithIdentityResolution().ToList();
```

---

[1] Diese Webseite beschäftigt sich u. a. mit dem Thema *Overusing a single **DbContext** instance*: https://docs.microsoft.com/en-us/ef/core/change-tracking/identity-resolution

Um dieses Verhalten für den aktuellen Kontext generell einzustellen, setzt man die Eigenschaft **QueryTrackingBehavior** des **ChangeTracker**-Objekts zum Kontext auf den Wert **NoTrackingWithIdentityResolution** der Enumeration **QueryTrackingBehavior**:

```
context.ChangeTracker.QueryTrackingBehavior =
                    QueryTrackingBehavior.NoTrackingWithIdentityResolution;
```

20.6.4 Report der Änderungsnachverfolgung anfordern

Ein **ChangeTracker** bietet über seine Eigenschaft **DebugView** ein nützliches Protokoll seiner Tätigkeit. Im folgenden Beispiel

```
var adoPost = context.Posts.Single(p => p.Title.Contains("Adonet"));
var dotNetBlog = new Blog { Url =
              "https://devblogs.microsoft.com/dotnet/" };
adoPost.Blog = dotNetBlog;
context.ChangeTracker.DetectChanges();
Console.WriteLine(context.ChangeTracker.DebugView.LongView);
```

wird aus der mehrfach in Beispielen verwendeten Datenbank mit einer `Blogs`- und einer `Posts`-Tabelle ...

- eine `Post`-Entität mit einem bestimmten Titelbestandteil aus der Datenbank geladen,
- eine neue `Blog`-Entität erstellt,
- die `Blog`-Navigationseigenschaft der `Post`-Entität auf die neue `Blog`-Entität gesetzt.

Der **ChangeTracker** erstellt zu jeder Entität bei Aufnahme in die Änderungsnachverfolgung einen Schnappschuss. Über einen Aufruf seiner Methode **DetectChanges()** wird der **ChangeTracker** veranlasst, für jede Entität durch einen Vergleich des aktuellen Zustands mit dem Schnappschuss die Veränderungen festzustellen.

In der Ausgabe ist u. a. zu erfahren, dass eine `Blog`-Entität mit noch unbekanntem Primärschlüsselwert überwacht wird (`BlogId: -2147482647 PK Temporary`). Sobald die Datenbank einen Primärschlüsselwert für die `Blog`-Entität vergeben hat, wird dieser Wert auch als `BlogId`-Fremdschlüssel der `Post`-Entität eingetragen:

```
Blog {BlogId: -2147482647} Added
  BlogId: -2147482647 PK Temporary
  Url: 'https://devblogs.microsoft.com/dotnet/'
  Posts: [{PostId: 49}]
Post {PostId: 49} Modified
  PostId: 49 PK
  BlogId: -2147482647 FK Modified Temporary Originally 129
  Content: <null>
  Title: 'Adonet - Post 2'
  Blog: {BlogId: -2147482647}
```

20.6.5 Status und sonstige Informationen zu einer Entität

Wird eine Entität mit der **DbSet<TEntity>** - Methode **Add()**

public virtual EntityEntry<TEntity> Add(TEntity *entity*) where TEntity : class

in die Änderungsnachverfolgung durch das **DbContext**-Objekt aufgenommen, dann erhält man über die Rückgabe vom Typ **EntityEntry<TEntity>** diverse Informationen über die Entität, die z. B. für die Änderungsnachverfolgung relevant sind. Ein Objekt derselben Klasse liefert auch die generische **DbContext**-Methode **Entry<TEntity>()**:

public virtual EntityEntry<TEntity> Entry<TEntity>(TEntity *entity*) where TEntity : class

Welches SQL-Kommando beim Aufruf der **DbContext**-Methode **SaveChanges()** für eine Entität ausgeführt wird, hängt von ihrem Status ab, den man über die **EntityEntry<TEntity>** - Eigenschaft **State** mit Werten der Enumeration **EntityState** feststellen kann:

Detached	Die Entität ist nicht in die Änderungsnachverfolgung durch das **DbContext**-Objekt einbezogen.
Unchanged	Die Entität ist in die Änderungsnachverfolgung durch das **DbContext**-Objekt einbezogen. Sie existiert in der Datenbank, und die Kopie im Speicher ist im Vergleich zum Zustand in der Datenbank nicht verändert worden.
Deleted	Die Entität ist in die Änderungsnachverfolgung durch das **DbContext**-Objekt einbezogen. Sie existiert in der Datenbank und ist vorgemerkt für das Löschen aus der Datenbank.
Modified	Die Entität ist in die Änderungsnachverfolgung durch das **DbContext**-Objekt einbezogen. Sie existiert in der Datenbank, und die Kopie im Speicher ist im Vergleich zum Zustand in der Datenbank verändert worden.
Added	Die Entität ist per **Add()** in die Änderungsnachverfolgung durch das **DbContext**-Objekt aufgenommen worden. Sie existiert noch nicht in der Datenbank.

Über die **State**-Eigenschaft kann der Status einer Entität auch geändert werden. Im folgenden Beispiel wird eine neue Entität in die Änderungsnachverfolgung aufgenommen, indem ihr Status den Wert **Added** erhält:

Quellcodesegment

```
var blog = new Blog { Url = "https://devblogs.microsoft.com/aspnet/" };
context.Entry(blog).State = EntityState.Added;
foreach (var e in context.ChangeTracker.Entries())
    Console.WriteLine(" " + e.Entity + " " + e.State);
```

Ausgabe

```
DataChange.Blog Added
```

Es ist zu beachten, dass sich eine Entitäts-Statusänderung per Direktzugriff auf die **State**-Eigenschaft *nicht* auf den Status von abhängigen Entitäten auswirkt, z. B.:

Quellcodesegment

```
var aspBlog = new Blog { Url = "https://devblogs.microsoft.com/aspnet/" };
var dotNetPost = new Post { Title = "New in .NET 5.0" };
aspBlog.Posts.Add(dotNetPost);
Console.WriteLine("Blog: " + context.Entry(aspBlog).State);
Console.WriteLine("Post: " + context.Entry(dotNetPost).State);
context.Entry(aspBlog).State = EntityState.Added;
Console.WriteLine("Blog: " + context.Entry(aspBlog).State);
Console.WriteLine("Post: " + context.Entry(dotNetPost).State);
```

Ausgabe

```
Blog: Detached
Post: Detached
Blog: Added
Post: Detached
```

Resultiert eine Statusänderung hingegen aus der Aufnahme einer Entität in die Änderungsnachverfolgung über einen Aufruf der **DbSet**-Methode **Add()**, dann wird eine entsprechende Statusänderung automatisch auch für abhängige Entitäten vorgenommen:

Quellcodesegment

```
var aspBlog = new Blog { Url = "https://devblogs.microsoft.com/aspnet/" };
var dotNetPost = new Post { Title = "New in .NET 5.0" };
aspBlog.Posts.Add(dotNetPost);
context.Blogs.Add(aspBlog);
Console.WriteLine("Blog: " + context.Entry(aspBlog).State);
Console.WriteLine("Post: " + context.Entry(dotNetPost).State);
```

Ausgabe

```
Blog: Added
Post: Added
```

Ein **EntityEntry<TEntity>** - Objekt enthält noch weitere Informationen über eine Entität, z. B.:

- Wenn eine Entität aus der Datenbank geladen worden ist, dann sind ihre ursprünglichen Werte bekannt.
- Es ist zu erfahren, welche Eigenschaften verändert worden sind.

20.6.6 Vorhandene Entitäten in die Änderungsnachverfolgung aufnehmen

Eine in der Datenbank vorhandene Entität wird über die **DbSet<TEntity>** - Methode **Attach()**

public virtual EntityEntry<TEntity> Attach(TEntity *entity***)**

oder über die **DbContext**-Methode **Attach<TEntity>()**

public virtual EntityEntry<TEntity> Attach<TEntity>(TEntity *entity***)**
where TEntity : class

zusammen mit den per Navigationseigenschaft erreichbaren Entitäten in die Änderungsnachverfolgung aufgenommen. Das kann z. B. notwendig werden, weil die Entität von einer mittlerweile per **Dispose()** eliminierten **DbContext**-Instanz geladen wurde. Entitäten im Rahmen einer **NoTracking**-Abfrage zu laden und später in die Änderungsnachverfolgung aufzunehmen, ist *keine* sinnvolle Vorgehensweise.[1]

Zur Aufnahme von *mehreren* Entitäten ist in den Klassen **DbSet<TEntity>** und **DbContext** jeweils eine Methode namens **AttachRange()** vorhanden.

Eine Entität aus einer Klasse mit einem vom DBMS generierten Primärschlüssel (vgl. Abschnitt 20.2.2.6) besitzt nach der Aufnahme per **Attach()** oder **AttachRange()** den Status ...

- **Unchanged**, wenn der Primärschlüsselwert vorhanden ist,
- **Added,** wenn der Primärschlüsselwert fehlt.

Statt der Methode **Attach()** bzw. **AttachRange()** kann auch die Methode **Update()** bzw. **UpdateRange()** dazu verwendet werden, um eine Entität bzw. mehrere Entitäten zusammen mit den per Navigationseigenschaft erreichbaren Entitäten in die Änderungsnachverfolgung aufzunehmen. In diesem Fall hat eine Entität aus einer Klasse mit einem vom DBMS generierten Primärschlüssel nach der Aufnahme den Status ...

- **Modified**, wenn der Primärschlüsselwert vorhanden ist,
  Weil der **ChangeTracker** nicht weiß, welche Eigenschaften geändert worden sind, werden beim Aufruf von **SaveChanges()** *alle* Eigenschaften an die Datenbank übertragen.
- **Added,** wenn der Primärschlüsselwert fehlt.

Weil die Methoden **Attach()**, **AttachRange()**, **Update()** und **UpdateRange()** keine Datenbankzugriffe zur Folge haben, sind keine asynchronen Varianten vorhanden.

Wenn sich die Aufnahme von Entitäten in die Änderungsnachverfolgung *nicht* auf die per Navigationseigenschaft erreichbaren Entitäten auswirken soll, dann ändert man den Status der aufzunehmenden Entitäten (siehe Abschnitt 20.6.5).

20.7 Entitäten erstellen, modifizieren und sichern

Alle vom **ChangeTracker** der aktuellen **DbContext**-Instanz festgestellten Modifikationen werden mit der **DbContext**-Methode **SaveChanges()** an die Datenbank übertragen, wobei im Hintergrund die vom EF Core - Provider zur beteiligten Datenbank erstellten SQL-Kommandos **UPDATE**, **INSERT** und **DELETE** vom DBMS ausgeführt werden:

---

[1] https://docs.microsoft.com/en-us/ef/core/change-tracking/explicit-tracking

- Ergänzte Entitäten werden mit dem SQL-Kommando **INSERT** in die Datenbanktabelle eingefügt.
- Geänderte Entitäten erhalten mit dem SQL-Kommando **UPDATE** neue Werte.
- Aus der Datenbank geladene und dann gelöschte Entitäten werden über das SQL-Kommando **DELETE** aus der Datenbanktabelle entfernt.

Mit Hilfe der **DbContext**-Methode **SaveChangesAsync()** kann die Sicherung in einem Hintergrund-Thread erfolgen.

20.7.1 Entitäten erstellen, in die Änderungsnachverfolgung aufnehmen und sichern

Durch die folgenden Anweisungen wird im bereits mehrfach verwendeten Blogging-Beispiel eine Entität vom Typ `Blog` per **new**-Operator erstellt, mit der **DbSet<TEntity>** - Methode **Add()** in die Änderungsnachverfolgung durch das **DbContext**-Objekt aufgenommen (Status: **Added**) und über die **DbContext**-Methode **SaveChanges()** in die Datenbank gesichert:

```
using var context = new BloggingContext();
 . . .
var blog = new Blog { Url = "https://devblogs.microsoft.com/dotnet/" };
var ee = context.Blogs.Add(blog);
Console.WriteLine(ee.State); // Ausgabe: Added
context.SaveChanges();
```

Der EF Core - Provider sorgt beim Aufruf von **SaveChanges()** dafür, dass das zum Sichern erforderliche **INSERT**-Kommando im SQL-Dialekt der Datenbank erstellt und ausgeführt wird.

Beim **Add()** - Aufruf an ein **DbSet<TEntity>** - Objekt mit einer Entität als Parameter (siehe Abschnitt 20.6.1)

public virtual EntityEntry<TEntity> Add(TEntity *entity***)**

werden ggf. auch alle per Navigationseigenschaft erreichbaren Entitäten in die Änderungsnachverfolgung des **DbContext**-Objekts aufgenommen, wenn sie dort noch nicht registriert sind. Das geschieht im folgenden Beispiel mit zwei `Post`-Entitäten in der der `Posts`-Liste einer `Blog`-Entität:

```
var blog = new Blog { Url = "https://devblogs.microsoft.com/dotnet/" };
blog.Posts.Add(new Post { Title = ".NET - Post 1" });
blog.Posts.Add(new Post { Title = ".NET - Post 2" });
context.Blogs.Add(blog);
```

Vom **ChangeTracker** zum aktuellen **DbContext**-Objekt erfährt man über die Methode **Entries()** von den aktuell beobachteten Entitäten, z. B.:

Quellcodesegment	Ausgabe

`Console.WriteLine("\nTracked: ");`
`foreach (var e in context.ChangeTracker.Entries())`
`  Console.WriteLine(" " + e.Entity);`
	`Tracked:`
	` DataChange.Blog`
	` DataChange.Post`
	` DataChange.Post`

Im folgenden Beispiel wird ...

- eine `Post`-Entität aus der Datenbank geladen,
- eine neue `Blog`-Entität erstellt,
- die neue `Blog`-Entität zum Referenzziel der Navigationseigenschaft `Blog` der `Post`-Entität gemacht.

Dadurch wird die `Blog`-Entität in die Änderungsnachverfolgung aufgenommen, und der Fremdschlüssel `BlogId` der `Post`-Entität wird aktualisiert.

Quellcodesegment	Ausgabe

`var mauiPost = context.Posts`
`  .Single(p => p.Title.Contains("MAUI"));`
`var dotNetBlog = new Blog { Url =`
`  "https://devblogs.microsoft.com/dotnet/" };`
`mauiPost.Blog = dotNetBlog;`
`Console.WriteLine("\nTracked: ");`
`foreach (var e in context.ChangeTracker.Entries())`
`  Console.WriteLine($" {e.Entity} ({e.State})");`
`context.SaveChanges();`
	`Tracked:`
	` DataChange.Blog (Added)`
	` DataChange.Post (Modified)`

Beim Aufruf von **SaveChanges()** werden alle in die Änderungsnachverfolgung einbezogenen Entitäten mit dem Status **Added** oder **Modified** in die Datenbank gesichert.

Statt der **DbSet<TEntity>** - Methode **Add()** kann auch die generische **DbContext**-Methode **Add<TEntity>()** dazu verwendet werden, um eine Entität zusammen mit den per Navigationseigenschaft erreichbaren Entitäten in die Änderungsnachverfolgung durch das **DbContext**-Objekt aufzunehmen:

public virtual EntityEntry<TEntity> Add<TEntity>(TEntity *entity***) where TEntity : class**

Weitere Methoden zur Aufnahme neuer Entitäten in die Änderungsnachverfolgung sind:

- die **DbSet<TEntity>** - Methode **AddRange()** zur Aufnahme von mehreren Entitäten
- die asynchronen **DbSet<TEntity>** - Methoden **AddAsync()** und **AddRangeAsync()**
- die analogen **DbContext**-Methoden

Normalerweise haben die Methoden **Add()** und **AddRange()** keine Datenbankzugriffe zur Folge, sodass *keine* Notwendigkeit für die Verwendung der asynchronen Varianten besteht.[1]

20.7.2 Entitäten ändern und sichern

Durch die folgenden Anweisungen wird im bereits mehrfach verwendeten Blogging-Beispiel eine Entität vom Typ `Post` aus der Datenbank abgefragt, modifiziert und über die **DbContext**-Methode **SaveChanges()** in die Datenbank gesichert:

```
using (var context = new BloggingContext()) {
   . . .
  var post = context.Posts.Single(p => p.Title.Contains("MAUI"));
  post.Title = "MAUI (Update)";
  context.SaveChanges();
}
```

Der EF Core - Provider sorgt beim Aufruf von **SaveChanges()** dafür, dass das zum Sichern erforderliche **UPDATE**-Kommando im SQL-Dialekt der Datenbank erstellt und ausgeführt wird.

Das Ändern des Primärschlüsselwerts einer Prinzipalentität gefährdet die referentielle Integrität, wenn abhängige Entitäten vorhanden sind. Wir sparen uns die Behandlung dieses Problems, weil Primärschlüsselwerte in der Regel von der Datenbank berechnet werden und von einem Klienten nicht geändert werden dürfen. Das Löschen einer Prinzipalentität ist aber meist zulässig, und dabei tritt ein ähnliches Problem mit der referentiellen Integrität auf (siehe Abschnitt 20.7.3).

---

[1] https://docs.microsoft.com/en-us/ef/core/change-tracking/miscellaneous

20.7.3 Entitäten löschen

Eine aus der Datenbank geladene Entität (mit dem aktuellen Status **Unchanged** oder **Modified**) kann mit der **DbSet<TEntity>** - Methode **Remove()**

public virtual EntityEntry<TEntity> Remove(TEntity *entity***)**

in den Status **Deleted** versetzt, also zum Löschen aus der Datenbank vorgemerkt werden, z. B.:

Quellcodesegment	Ausgabe

`using (var context = new BloggingContext()) {`
`  var post = context.Posts.Single(`
`  p => p.Title.Contains("Adonet - Post 1"));`
`  Console.WriteLine("State of Post: " + context.Entry(post).State);`
`  context.Posts.Remove(post);`
`  Console.WriteLine("State of Post: " + context.Entry(post).State);`
`  context.SaveChanges();`
`}`
	`State of Post: Unchanged`
	`State of Post: Deleted`

Durch die folgenden Anweisungen wird eine Entität bzw. Tabellenzeile mit einem bekannten Primärschlüsselwert gelöscht:

```
var post = context.Posts.Single(p => p.PostId == 1);
context.Posts.Remove(post);
context.SaveChanges();
```

Dabei sind in der Regel *zwei* Datenbankkontakte erforderlich:

- Den ersten Kontakt verursacht die Abfrage per **Single()**, wenn sich noch keine `Posts`-Entität mit dem gesuchten Primärschlüsselwert in der Änderungsnachverfolgung befindet.
- Den zweiten Kontakt verursacht die Aktualisierung per **SaveChanges()**.

Durch eine von Brind (2021) vorgeschlagene Technik lässt sich der erste Datenbankkontakt einsparen:[1]

```
var adoPost = new Post { PostId = 1 };
context.Posts.Attach(adoPost);
context.Posts.Remove(adoPost);
context.SaveChanges();
```

Es wird eine Stellvertreter-Entität mit passendem Primärschlüsselwert erstellt, per **Attach()** in die Änderungsnachverfolgung aufgenommen, per **Remove()** in den Zustand **Deleted** versetzt und schließlich per **SaveChanges()** aus der Datenbank entfernt.

Durch die folgende Überladung der **DbSet<TEntity>** - Methode **RemoveRange()** werden die per Serienparameter benannten Entitäten in den Zustand **Deleted** versetzt, sodass sie beim nächsten Aufruf von **SaveChanges()** aus der Datenbank entfernt werden:

public virtual void RemoveRange(params TEntity[] *entities***)**

Zu den **DbSet<TEntity>** - Methoden **Remove()** und **RemoveRange()** existieren analoge Methoden in der Klasse **DbContext**. Asynchrone Varianten sind nicht vorhanden, weil die Methoden keine Datenbankzugriffe zur Folge haben.

Der EF Core - Provider sorgt beim Aufruf von **SaveChanges()** dafür, dass die zum Löschen erforderlichen **DELETE**-Kommandos im SQL-Dialekt der Datenbank erstellt und ausgeführt werden.

Die beiden Entitäten `Blog` und `Post` (bzw. die Tabellen `Blogs` und `Posts`) im Blogging-Beispiel stehen in einer Master-Details - Beziehung (vgl. Abschnitt 20.2.2.9):

---

[1] https://www.learnentityframeworkcore.com/dbcontext/deleting-data

- `Blog` ist die Master-Entität, und `Blogs` ist die Master-Tabelle.
- `Post` ist die Details-Entität, und `Posts` ist die Details-Tabelle.

Während das Löschen einer Details-Zeile (siehe oben) keine Relevanz für die referentielle Integrität der Datenbank hat, resultiert aus dem Löschen einer Master-Zeile mit zugehörigen Details-Zeilen ein Problem, weil Fremdschlüsselwerte ihre Gültigkeit verlieren.

Die möglichen Lösungen für das Problem hängen davon ab, ob es sich um eine optionale oder um eine obligatorische Beziehung handelt:

- Eine **obligatorische Beziehung** ist daran zu erkennen, dass für den Fremdschlüssel das **null**-Verbot besteht. Um die Verletzung der referentiellen Integrität zu beseitigen, müssen die abhängigen Zeilen bzw. Entitäten ebenfalls gelöscht werden. Man spricht hier vom *kaskadierenden Löschen*.
- Bei einer **optionalen Beziehung** sind für den Fremdschlüssel **null**-Werte erlaubt. Um die Verletzung der referentiellen Integrität zu beseitigen, kann man ...
  - die Fremdschlüsselwerte der abhängigen Entitäten löschen (auf **null** setzen),
  - oder die abhängigen Entitäten löschen.

  Das EF Core wählt per Voreinstellung die *erste* Lösung.

Die bislang verwendete Konfiguration der Blogging-Datenbank besitzt eine *obligatorische* Master-Details - Beziehung zwischen den Entitätsklassen `Blog` und Post, weil in der Entitätsklasse `Post` für den Fremdschlüssel `BlogId` ein Datentyp mit **null**-Verbot (nämlich **int**) verwendet wird:

```
public class Post {
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

Beim Löschen einer Master-Entität werden folglich auch die abhängigen Entitäten gelöscht:

```
using var context = new BloggingContext();
var vsBlog = context.Blogs.Single(b => b.Url.Contains("visualstudio"));
context.Blogs.Remove(vsBlog);
context.SaveChanges();
```

Dabei ist es nicht erforderlich, die Details-Entitäten in die Abfrage einzubeziehen.

Nun soll demonstriert werden, dass das EF Core bei erlaubten **null**-Werten für die Fremdschlüsseleigenschaft (also bei einer *optionalen* Master-Details - Beziehung) beim Löschen einer Master-Zeile ...

- die zugehörigen Details-Zeilen *nicht* aus der Datenbanktabelle löscht
- sondern die Fremdschlüsselwerte dieser Zeilen auf **NULL** setzt.

Dazu erhält die Fremdschlüsselvariable `BlogId` in der Entitätsklasse `Post` einen Datentyp mit zulässigen **null**-Werten (vgl. Abschnitt 8.3), obwohl ein `Post` ohne `Blog` nicht unbedingt sinnvoll ist:

```
public int? BlogId { get; set; }
```

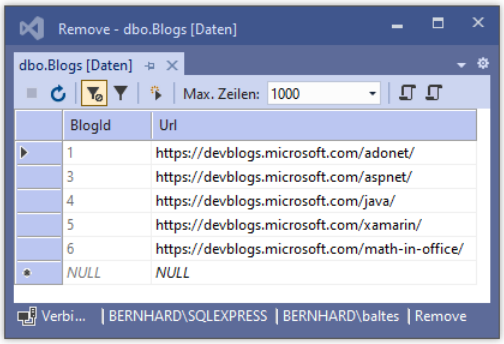
Weil beim Löschen einer Master-Zeile in den zugehörigen Details-Zeilen die Fremdschlüsselwerte zu ändern sind, wird in der Abfrage durch die Erweiterungsmethode **Include()** dafür gesorgt, dass auch die abhängigen Entitäten einbezogen werden (siehe Abschnitt 20.5.4.1). Im Beispiel wird eine Master-Zeile per **Remove()** - Aufruf in den Zustand **Deleted** versetzt, wobei die abhängige Details-Zeile in den Zustand **Modified** gelangt:

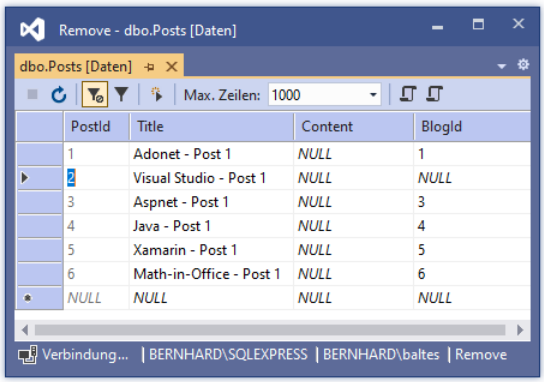
Quellcodesegment	Ausgabe

```
using (var context = new BloggingContext()) {
   var vsBlog = context.Blogs
                   .Include(b => b.Posts)
                   .Single(b => b.Url.Contains("visualstudio"));
   context.Blogs.Remove(vsBlog);
   foreach (var e in context.ChangeTracker.Entries())
       Console.WriteLine(" " + e.Entity + " " + e.State);
   context.SaveChanges();
}
```

```
DataChange.Post Modified
DataChange.Blog Deleted
```

Beim Datenbank-Update wird die Master-Zeile *gelöscht*, die Details-Zeile hingegen *geändert*:





Für eine *optionale* Master-Details - Beziehung kann in der Methode **OnModelCreating()** angeordnet werden, dass beim Löschen einer Master-Entität die abhängigen Entitäten kaskadierend gelöscht werden sollen:

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder
        .Entity<Blog>()
        .HasMany(e => e.Posts)
        .WithOne(e => e.Blog)
        .OnDelete(DeleteBehavior.ClientCascade);
}
```

Dann wird durch das obige Codesegment trotz der optionalen Beziehung mit der Master- auch die Details-Entität gelöscht:

Quellcodesegment	Ausgabe

```
using (var context = new BloggingContext()) {
   var vsBlog = context
                   .Blogs.Include(b => b.Posts)
                   .Single(b => b.Url.Contains("visualstudio"));
   context.Blogs.Remove(vsBlog);
   foreach (var e in context.ChangeTracker.Entries())
       Console.WriteLine(" " + e.Entity + " " + e.State);
   context.SaveChanges();
}
```

```
DataChange.Post Deleted
DataChange.Blog Deleted
```

20.7.4 Parallelitätsverwaltung

Parallele schreibende Zugriffe auf dieselbe Zeile in einer Datenbanktabelle durch mehrere Benutzer verursachen Konflikte, z. B.:

- Benutzer A ruft die Tabellenzeile ab und bearbeitet sie lokal.
- Noch während der laufenden Bearbeitung durch Benutzer A wird dieselbe Tabellenzeile durch Benutzer B geändert.
- Nach der Fertigstellung seiner Bearbeitung möchte Benutzer A das Ergebnis in die Datenbank schreiben.

20.7.4.1 Konfliktstrategien

Per Voreinstellung diagnostiziert das EF Core keine Zugriffskonflikte, sodass der letzte Schreibzugriff gewinnt. Beim Sichern einer Entität durch die **DbContext**-Methode **SaveChanges()** werden nach Möglichkeit aber nur die *geänderten* Eigenschaften zur Datenbank übertragen. Zwischenzeitliche Modifikationen anderer Eigenschaften gehen also nicht verloren. Dieses sparsame und schonende Sichern kann aber unerwünscht sein, z. B.:

- Die Verkaufsberater A und B eines Autohändlers rufen eine Datenbankzeile mit einem Fahrzeugangebot ab. Beide möchten das Angebot attraktiver machen.
- Verkaufsberater A gewährt eine 5-Jahres - Garantie bei konstantem Preis und sichert seine Änderung.
- Verkaufsberater B verzichtet auf eine Garantieverlängerung, reduziert aber den Preis um 1000 € und sichert seine Änderung.
- Es resultiert ein Fahrzeugangebot mit verlängerter Garantie *und* reduziertem Preis, das vermutlich beiden Verkaufsberatern missfällt.

Wenn der **ChangeTracker** keine Informationen über den Änderungsstatus der Eigenschaften besitzt, weil eine Entität z. B. von einem anderen Kontext abgefragt wurde (vgl. Abschnitt 20.6.6) dann werden beim Update *alle* Eigenschaftsausprägungen an die Datenbank übertragen.

Als Alternativen zu der im EF Core voreingestellten Aktualisierung ohne Kontrolle zwischenzeitlicher Datenbankveränderungen kommen in Frage:

- Pessimistische Parallelitätskontrolle (Sperrung)
  Die von einem Benutzer zum Aktualisieren geöffneten Tabellenzeilen werden für andere Benutzer gesperrt, damit ...
  - keine unkoordinierten Änderungen stattfinden,
  - keine potentiell ungültigen Daten gelesen werden.

  Weitere Details müssen nicht beschrieben werden, weil das EF Core die pessimistische Parallelitätskontrolle *nicht* unterstützt. Das Sperren wäre nicht bei jedem DBMS realisierbar und hätte Performanzprobleme bei Anwendungen mit vielen Benutzern.[1]
- Optimistische Parallelitätskontrolle
  Vor der Aktualisierung einer Entität wird kontrolliert, ob sich die zugehörige Tabellenzeile seit der Abfrage geändert hat. Ist dies der Fall, dann muss (z. B. aufgrund einer Benutzerpräferenz) zwischen den konkurrierenden Versionen entschieden werden.

[1] https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/concurrency

20.7.4.2 Erkennung paralleler Zugriffe

Zur Erkennung zwischenzeitlicher Veränderungen in der Datenbank vor einer Aktualisierung bietet das EF Core zwei Techniken an:

- Kontrolle aufgrund einzelner kritischer Eigenschaften
- Kontrolle aufgrund einer Zeilenversionseigenschaft

Wird bei einer anstehenden Aktualisierung ein Zugriffskonflikt entdeckt, dann wird man in der Regel den betroffenen Benutzer über die konkurrierenden Daten informieren und um eine Entscheidung bitten.

20.7.4.2.1 Einzelne kritische Eigenschaften

Vor der Aktualisierung einer Entität wird überprüft, ob bei einer Liste von kritischen Eigenschaften zwischenzeitlich in der Datenbank Wertveränderungen vorgenommen wurden. Die Auszeichnung der kritischen Eigenschaften kann über die Annotation **ConcurrencyCheck**

```
public class Blog {
    public int BlogId { get; set; }
    [ConcurrencyCheck]
    public string Url { get; set; }
    public int Rating { get; set; }
}
```

oder in der **DbContext**-Methode **OnModelCreating()** erfolgen:

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Blog>()
        .Property(a => a.Url).IsConcurrencyToken();
}
```

Bei der Aktualisierung einer Entität

```
var blog = context.Blogs
    .Where(b => b.Url == "https://devblogs.microsoft.com/adonet/")
    .Single();
blog.Rating = 3;
try {
    context.SaveChanges();
} catch (Exception e) {
    Console.WriteLine(e.Message);
}
```

überprüft das EF Core, ob in der Datenbanktabelle *genau eine* Zeile mit übereinstimmenden Werten für alle kritischen Eigenschaften vorhanden ist. Wird eine solche Zeile *nicht* gefunden, wirft **SaveChanges()** eine **DbUpdateConcurrencyException**, z. B.:

```
Database operation expected to affect 1 row(s) but actually affected 0 row(s). Data
may have been modified or deleted since entities were loaded.
```

Daher sollte **SaveChanges()** bei bestehender Parallelitätsüberwachung im Rahmen einer **try-catch** -Anweisung aufgerufen werden.

20.7.4.2.2 Zeilenversion

Wenn die Aktualisierung einer Entität bei *beliebigen* zwischenzeitlich in der Datenbank erfolgten Wertveränderungen verhindert werden soll, dann empfiehlt sich die Verwendung einer Eigenschaft als *Parallelitätstoken* (alias *Zeilenversion* oder *Zeitstempel*). Das kann mit der Annotation **Timestamp**

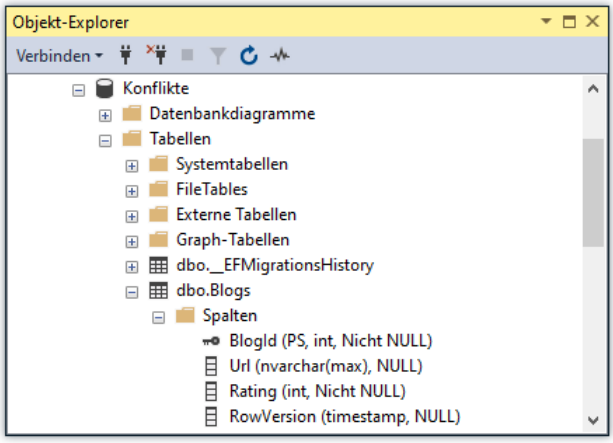
```
[Timestamp]
public byte[] RowVersion { get; set; }
```

oder in der **DbContext**-Methode **OnModelCreating()** geschehen, z. B.:

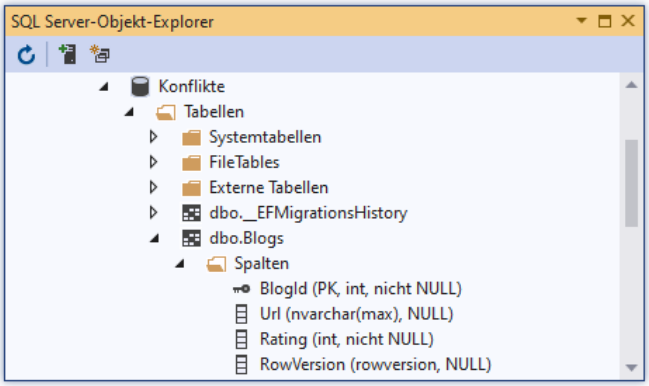
```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Blog>()
        .Property(p => p.RowVersion)
        .IsRowVersion();
}
```

Es hängt vom DBMS ab, welcher Datentyp für die Zeilenversionseigenschaft zu verwenden ist; bei Microsofts SQL Server eignet sich der Typ **byte[]** (siehe Beispiel).

Vom **Objekt-Explorer** im SQL Server Management Studio wird die `Blogs`-Tabelle der im Beispiel verwendeten SQL Server - Datenbank so dargestellt:



Betrachtet man denselben Datenbankstatus mit dem **SQL Server Objekt-Explorer** im Visual Studio, dann fällt die abweichende Datentypanzeige für die `Blogs`-Spalte `RowVersion` auf:



Die beiden Bezeichner **timestamp** und **rowversion** sind allerdings synonym, wobei Microsoft die Verwendung von **rowversion** in DDL-Anweisungen empfiehlt.[1]

Wird bei der Ausführung der **DbContext**-Methode **SaveChanges()**

[1] https://docs.microsoft.com/en-us/sql/t-sql/data-types/rowversion-transact-sql

```
var blog = context.Blogs
    .Where(b => b.Url == "https://devblogs.microsoft.com/adonet/")
    .Single();
blog.Rating = 3;
try {
    context.SaveChanges();
} catch (Exception e) {
    Console.WriteLine(e.Message);
}
```

festgestellt, dass sich betroffene Datenbankzeilen seit der Abfrage geändert haben, dann kommt es zu einer **DbUpdateConcurrencyException**:

```
Database operation expected to affect 1 row(s) but actually affected 0 row(s). Data
may have been modified or deleted since entities were loaded.
```

21 Netzwerkprogrammierung

Die BCL enthält zahlreiche Klassen zur Netzwerkprogrammierung, wobei man zwischen einem Zugriff auf Netzwerkressourcen über Standardprotokolle der Anwendungsebene (z. B. HTTP) und einer Programmierung auf elementaren Protokollebenen (z. B. TCP/IP) mit einer entsprechend weiterreichenden Kontrolle wählen kann.

In diesem Manuskript können wichtige Einsatzfelder für die *server-basierte* Netzwerkprogrammierung *nicht* behandelt werden, z. B.:

- **Webdienste**
  Ein Webdienst bietet ein API (*Application Programming Interface*) an und ist zur Benutzung durch andere Programme konzipiert, wobei die kommunizierenden Programme ...
  - meist auf unterschiedlichen Rechnern laufen,
  - oft durch unterschiedliche Techniken (z. B. Programmiersprachen) realisiert sind,
  - zum Datenaustausch meist das XML- oder das JSON-Format benutzen.

  Eine Anwendung auf einem Rechner in einem Reisebüro kann sich z. B. vom Webdienst einer Fluglinie Daten über Verbindungen, freie Plätze, Preise etc. beschaffen. Ein Webdienst hat keine Bedienoberfläche, kümmert sich also nicht um die Präsentation der Daten. Man kann von einer *M2M-Kommunikation* (*Machine-to-Machine*) sprechen. Im kommerziellen Bereich ist die Bezeichnung *B2B-Kommunikation* (*Business-to-Business*) üblich.
  Zur Realisation von Webdiensten wird meist eine von den folgenden Techniken eingesetzt:
  - Das SOAP-Protokoll (*Simple Object Access Protocol*) verwendet maschinenlesbare Verträge in der ***W**eb **S**ervices **D**escription **L**anguage* (WSDL).
  - Die REST-Prinzipien (*Representational State Transfer*) beschreiben Anforderungen an einen Webdienst. Als Protokoll wird meist HTTPS verwendet.

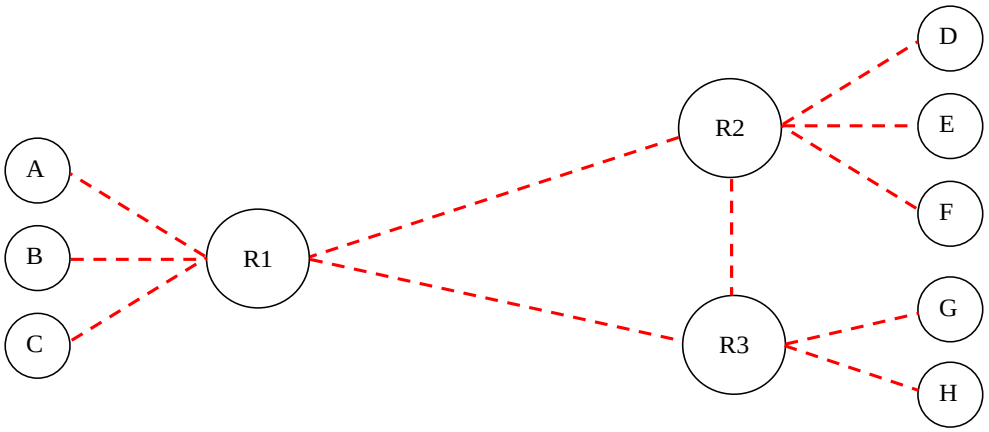
  Wenn die realisierenden Techniken SOAP bzw. REST bei der Begriffsbestimmung weggelassen werden, dann kann man jedes per Netz (auf einem bestimmten Rechner, an einem bestimmten Port, siehe unten) erreichbare und zur M2M-Kommunikation geeignete Programm (z.B. einen NTP-Server (*Network Time Protocol*), der die Uhrzeit liefert) als *Webdienst* bezeichnen.
- **Webanwendungen**
  Webanwendungen sind für die interaktive Nutzung mit Hilfe eines Web-Browsers konzipiert. Es werden (oft in Kooperation mit einer Datenbankanwendung) HTML-Seiten mit angeforderten Daten oder Berechnungen individuell erstellt und dann zum Browser gesendet. Diese HTML-Seiten enthalten Formulare, sodass eine Bedienoberfläche entsteht. Man kann von einer *H2M-Kommunikation* (*Human-to-Machine*) sprechen. Im kommerziellen Bereich ist die Bezeichnung *B2C-Kommunikation* (*Business-to-Customer*) üblich.
  Statt komplexe Webanwendungen zu erstellen, die unter Verwendung von entsprechend vielen Klassen alle Teilaufgaben erledigen, wird heutzutage eine Verlagerung von Kompetenzen in spezialisierte Webdienste angestrebt, die sich z. B. um die Benutzerauthentifizierung oder die Zahlungsabwicklung kümmern.
  Trotz H2M-Eignung lassen sich Webanwendungen auch durch Klienten*programme* ansprechen, was wir im Abschnitt 21.2.4.2 tun werden. Die so bezogenen HTML-Seiten können zwar per Software interpretiert werden, sind aber nicht ideal für den Datentransfer zwischen Programmen.
  Auf der Klientenseite sind mittlerweile im Browser ablaufende JavaScript-Lösungen stark vertreten. Als mächtige Programmiersprache kann JavaScript Webdienste direkt (ohne den Umweg über eine Webanwendung) ansprechen. Folglich existieren viele Optionen für das Design von internet-basierter Software, wobei sich die Klientenseite nicht unbedingt auf das Rendern von HTML-Code und die Datenerfassung über Formulare beschränkt.

21.1 Wichtige Konzepte der Netzwerktechnologie

Als **Netzwerk** bezeichnet man eine Anzahl von Systemen (z.B. Rechnern), die über ein **gemeinsames Medium** (z.B. Ethernet-Kabel, WLAN, Infrarotkanal) verbunden sind und über ein **gemeinsames Protokoll** (z.B. TCP/IP) Daten austauschen können.

Unter einem **Protokoll** ist eine Menge von **Regeln** zu verstehen, die für eine erfolgreiche Kommunikation von allen beteiligten Systemen eingehalten werden müssen.

Zwischen zwei Kommunikationspartnern jeweils eine reservierte Leitung (temporär) zu schalten und auch in „Funkpausen“ aufrecht zu erhalten, wäre unökonomisch. Bei den meisten aktuellen Netzwerkprotokollen werden **Datenpakete** mit **Adressierung** übertragen, was die gemeinsame Verwendung eines Verbindungswegs für mehrere, simultan ablaufende Kommunikationsprozesse ermöglicht. Dabei sind Vermittlungsstationen (Router) für die korrekte Weiterleitung der Pakete zuständig, z. B.:



Von der Anwendungsebene (z.B. Versand einer E-Mail über einen SMTP-Server (*Simple Mail Transfer Protocol*)) bis zur physischen Ebene (z.B. elektromagnetische Wellen auf einem Ethernet-Kabel) sind zahlreiche Übersetzungen vorzunehmen bzw. Aufgaben zu lösen, jeweils unter Beachtung der zugehörigen Regeln. Im nächsten Abschnitt werden die beteiligten Ebenen mit ihren jeweiligen Protokollen behandelt, wobei wir uns auf Themen mit Relevanz für die Anwendungsentwicklung konzentrieren.

21.1.1 Das OSI-Modell

Nach dem **OSI - Modell** (*Open System Interconnection*) der ISO (*International Standards Organisation*) werden bei der Kommunikation über Netzwerke sieben aufeinander aufbauende Schichten (engl.: *layers*) mit jeweiligen Zuständigkeiten und zugehörigen Protokollen unterschieden. Bei der anschließenden Beschreibung dieser Schichten sollen wichtige Begriffe und vor allem die heute üblichen Internet-Protokolle (z. B. IP, TCP, UDP, ICMP, HTTP) eingeordnet werden.

1. Physische Ebene (Bit-Übertragung, z. B. über Kupferdrahtleitungen)

Hier wird festgelegt, wie von der Netzwerk-Hardware Bits zwischen zwei *direkt verbundenen* Stationen zu übertragen sind. Im einfachen Beispiel einer seriellen Verbindung über Kupferkabel wird z. B. festgelegt, dass zur Übertragung einer 0 eine bestimmte Spannung während einer festgelegten Zeit angelegt wird, während eine 1 durch eine gleichlange Phase der Spannungsfreiheit ausgedrückt wird.

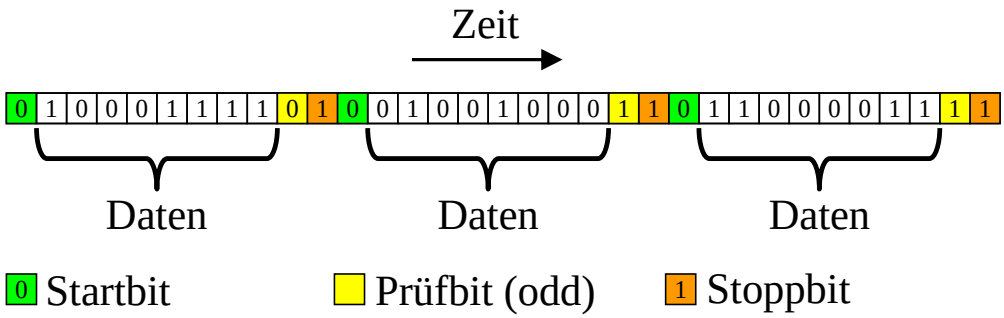
2. Link-Ebene (Gesicherte Übertragung von Datenpaketen, z. B. per Ethernet-Protokoll)

Hier wird vereinbart, wie zwischen zwei Stationen, die sich im selben Subnetz (einer Rundruf-Domäne) befinden oder direkt verbunden sind, ein **Datenpaket** zu übertragen ist, das aus einer Anzahl von Bits besteht und durch eine Prüfsumme gesichert ist. Die Datenpakete werden auch als *Frames* bezeichnet und sind von der Übertragungstechnik abhängig. In der Regel gehören zum Protokoll dieser Ebene auch Start- und Endmarkierungen, damit sich die beteiligten Geräte rechtzeitig auf eine Informationsübertragung einstellen können.

Im einfachen Beispiel einer seriellen Datenübertragung zwischen direkt verbundenen Stationen über das betagte RS-232 - Protokoll wird der folgende Frame-Aufbau verwendet:

Startbit:	0
8 Datenbits	0 oder 1
Prüfbit:	odd (siehe unten)
Stoppbit:	1

In der folgenden Abbildung sind drei RS-232 - Frames zu sehen, die nacheinander über eine serielle Leitung gesendet werden:



Das odd-Prüfbit wird so gesetzt, dass es die acht Datenbits zu einer ungeraden Summe ergänzt.

Bei einem Ethernet-Frame ist der Aufbau etwas komplizierter, weil die Kommunikation in einem Subnetz (in einer Rundruf-Domäne) unterstützt wird und folglich eine Adressierung erforderlich ist (siehe z. B. Spurgeon 2000, S. 40ff):

- Es ist ein Header vorhanden, der u. a. die MAC-Adressen (*Media Access Control*) von Sender und Empfänger enthält. Diese Level-2 - Adressen sind nur für die Subnetz-interne Kommunikation relevant.
- Es können Daten im Umfang von 46 bis 1500 Bytes transportiert werden.

3. Netzwerkebene (Übertragung von Informationspaketen, z. B. per IP-Protokoll)

Die Datenpakete (Frames) der eben behandelten zweiten Ebene hängen von der verwendeten Netzwerktechnik ab, sodass auf der Strecke vom Absender bis zum Empfänger in der Regel *mehrere* Frame-Architekturen beteiligt sind (z. B. bei der WLAN-Verbindung zum hausinternen Router eine andere als auf der Kupferkabelstrecke zum DSL-Provider). Auf der dritten Ebene kommen hingegen **Informationspakete** zum Einsatz, die auf der gesamten Strecke (im Intra- und/oder im Internet) unverändert bleiben und beim Wechsel der Netzwerktechnik in verschiedene Schicht 2 - Frames umgeladen werden.

Durch die Protokolle der Schicht 3 sind u. a. die folgenden Aufgaben zu erfüllen:

- **Adressierung (über Subnetzgrenzen hinweg)**
  Jedes Paket enthält eine Absender- und eine Zieladresse mit *globaler* Gültigkeit (über Subnetzgrenzen hinweg).
- **Routing**
  In komplexen (und ausfallsicheren) Netzen führen mehrere Wege vom Absender eines Pakets zum Ziel. Vermittlungsrechner (sog. *Router*) entscheiden darüber, welchen Weg ein Paket nehmen soll.
- **Datenflusskontrolle**
  Eine weitere Aufgabe der dritten Protokollebene besteht in der Datenflusskontrolle, wobei der Empfänger zur Vermeidung einer Überlastung dem Sender signalisiert, in welcher Geschwindigkeit er Pakete senden darf.

In der aktuellen Netzwerktechnik kommt auf der Ebene 3 überwiegend das **IP-Protokoll** zum Einsatz. Seine Pakete bezeichnet man auch als **IP-Datagramme**. In der älteren, immer noch verbreiteten IP-Version 4 (IPv4) besteht eine Adresse aus 32 Bits, die üblicherweise durch vier per Punkt getrennte Dezimalzahlen (aus dem Bereich von 0 bis 255) dargestellt werden, z. B.:

192.168.178.12

Bei der modernen IP-Version 6 (IPv6) besteht eine Adresse aus 128 Bits, welche durch acht per Doppelpunkt getrennte Blöcke mit jeweils vier Hexadezimalziffern dargestellt werden, z. B.:

2001:08c7:c79c:0000:0000:0000:88c7:0091

Innerhalb eines Blocks dürfen führende Nullen weggelassen werden, z. B.:

2001:8c7:c79c:0:0:0:88c7:91

Eine Gruppe aufeinanderfolgender Blöcke mit dem Wert 0000 bzw. 0 darf durch zwei Doppelpunkte ersetzt werden, z. B.:

2001:8c7:c79c::88c7:91

Nach dieser Regel kann die reservierte IPv6-Adresse des lokalen Rechners

0:0:0:0:0:0:0:1

sehr kurz geschrieben werden:

::1

Der OSI-Ebene 3 wird auch das **Internet Control Message Protocol** (ICMP) zugerechnet, das zur Übermittlung von Fehlermeldungen und verwandten Informationen dient. Wenn z. B. ein Router ein IP-Datagramm verwerfen muss, weil seine Maximalzahl von Weiterleitungen (*Time To Live*, TTL) erreicht wurde, dann schickt er in der Regel eine *Time Exceeded* - Meldung an den Absender. Auch die von **ping** - Anwendungen versendeten *Echo Requests* und die zugehörigen Antworten zählen zu den ICMP - Nachrichten.

4. Transportschicht (Gesicherte Paketübertragung, z. B. per TCP-Protokoll)

Zwar bemüht sich die Protokollebene 3 darum, Pakete auf möglichst schnellem Weg vom Absender zum Ziel zu befördern, sie kann jedoch nicht garantieren, dass *alle* Pakete in *korrekter Reihenfolge* ankommen. Dafür sind die Protokolle der Transportschicht zuständig, wobei momentan vor allem das **Transmission Control Protocol** (TCP) zum Einsatz kommt. Das TCP wiederholt z. B. die Übertragung von Paketen, wenn innerhalb einer festgelegten Zeit keine Bestätigung des Empfängers beim Absender eingetroffen ist.

5. Sitzungsebene (Übertragung von Byte-Strömen zwischen Anwendungen, z. B. per TCP)

Auf dieser Ebene sind Regeln angesiedelt, die den Datenaustausch zwischen zwei **Anwendungen** (meist auf verschiedenen Rechnern) ermöglichen. Auch solche Aufgaben werden in der heute üblichen Praxis vom Transmission Control Protocol (TCP) abgedeckt, das folglich für die OSI-Schichten 4 und 5 zuständig ist.

Damit eine Anwendung auf Rechner A mit einer Anwendung auf Rechner B kommunizieren kann, werden sogenannte **Ports** verwendet. Hierbei handelt es sich um Zahlen zwischen 0 und 65535 (2^{16} - 1), die eine kommunikationswillige bzw. -fähige Anwendung auf einem Rechner identifizieren. So wird es z. B. möglich, auf einem Rechner verschiedene Server-Programme zu installieren, die trotzdem von Klienten aufgrund ihrer verschiedenen Ports (z. B. 25 für einen SMTP-Server, 80 für einen WWW-Server) gezielt angesprochen werden können. Die Ports ...[1]

- von 0 bis 1023 sind für Standarddienste reserviert (*system ports*, *well-known ports*).
- von 1024 bis 49151 werden von der IANA (*Internet Assigned Numbers Authority*) verwaltet.
- von 49152 bis 65535 (im sogenannten *dynamischen* Port-Bereich) sind für lokale Zwecke und zur temporären Verwendung durch Klientenprogramme vorgesehen.

Eine TCP-Verbindung ist also bestimmt durch:

- Die IP-Adresse des Serverrechners und die Portnummer des Dienstes
- Die IP-Adresse des Klientenrechners und die dem Klientenprogramm vom Betriebssystem zugeteilte Portnummer

Das TCP-Protokoll stellt eine *virtuelle Verbindung* zwischen zwei Anwendungen her. Auf beiden Seiten steht eine als **Socket** (dt.: *Steckdose*) bezeichnete Programmierschnittstelle zur Verfügung. Die beiden Sockets kommunizieren über **Datenströme** miteinander. Aus der Sicht des Anwendungsprogrammierers werden per TCP keine Pakete übertragen, sondern Ströme von Bytes (zur Kommunikation über Datenströme siehe Kapitel 16). Auf einem Rechner können durchaus mehrere Programme per Socket-Technik mit Programmen auf anderen Rechnern (oder auch auf demselben Rechner) kommunizieren, wobei der Rechner trotzdem nur eine physische Netzwerksteckdose benötigt.

Von den Internet-Protokollen ist auch das **User Datagram Protocol** (UDP) auf der Ebene 5 einzuordnen. Es sorgt ebenfalls für eine Kommunikation zwischen Anwendungen und nutzt dazu Ports wie das TCP. Allerdings sind die Ports praktisch die einzige Erweiterung gegenüber der IP-Ebene. Es handelt sich also um einen ungesicherten Paketversand ohne Garantie für eine vollständige Auslieferung in korrekter Reihenfolge. Aufgrund der somit eingesparten Verwaltungskosten eignet sich das UDP zur Übertragung größerer Datenmengen, wenn dabei der Verlust einzelner Pakete zu verschmerzen ist (z. B. beim Multimedia - Streaming).

6. Präsentation

Hier geht es z. B. um die Verschlüsselung oder Komprimierung von Daten. Die TCP/IP - Protokollfamilie kümmert sich nicht darum, sondern überlässt derlei Arbeiten den Anwendungen.

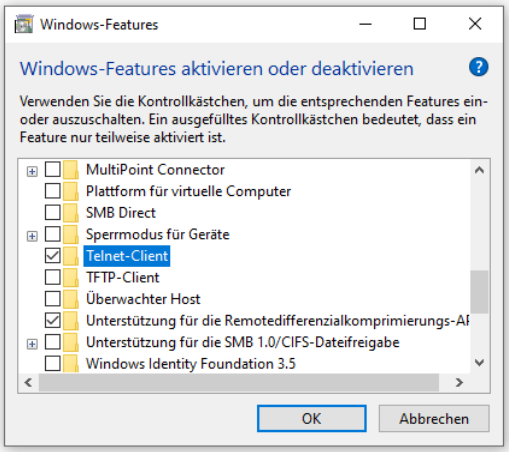
---

[1] https://de.wikipedia.org/wiki/Liste_der_standardisierten_Ports
https://de.wikipedia.org/wiki/Port_(Protokoll)

7. Anwendung (Protokolle für Endbenutzer-Dienstleistungen, z. B. HTTP(S))

Hier wird für verschiedene Dienste festgelegt, wie Anforderungen zu formulieren und Antworten auszuliefern sind. Als Beispiel betrachten wir die Kommunikation zwischen einem Klientenprogramm, das eine Mail versenden möchte, und einem SMTP-Server (*Simple Mail Transfer Protocol*), der an Port 25 lauscht. Zur Kommunikation mit einem SMTP-Server eignet sich z. B. ein Telnet-Klient, wobei unter Windows 10 eine Aktivierung erforderlich ist:

- Im Suchfeld *Systemsteuerung* eingeben und das gefundene Programm starten
- Aus der Liste mit den **kleinen Symbolen** die **Programme und Features** wählen
- Klick auf **Windows-Features aktivieren oder deaktivieren**
- **Telnet-Client** markieren und quittieren:



Um mit dem SMTP-Server **smtp.srv-dom.de** zu kommunizieren, startet man den Telnet-Klienten in einem Konsolenfenster über das folgende Kommando:

```
> telnet  smtp.srv-dom.de  25
```

Nach der Antwort des SMTP-Servers

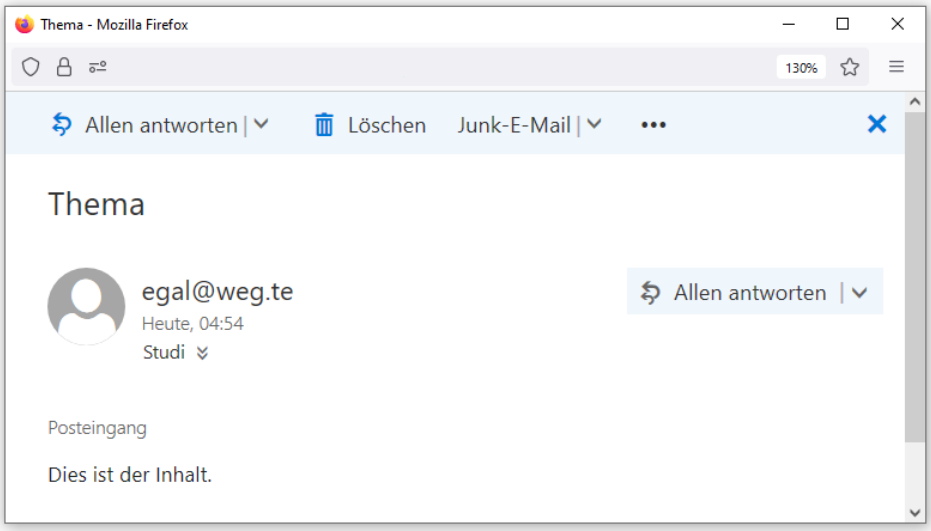
```
220 srv.srv-dom.de SMTP MAIL Service ready at Wed, 7 Jul 2021 04:23:41 +0200
```

kann man z. B. folgendermaßen eine Mail übergeben:

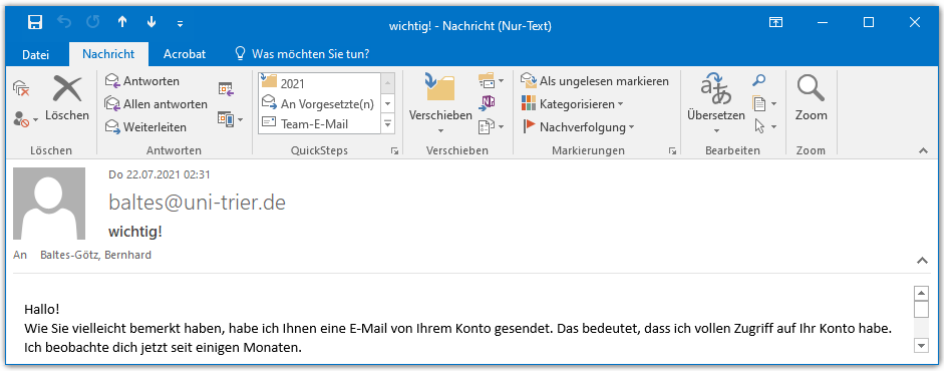
Klient	**Serverantwort**

`HELO mainpc.client-dom.de`
`MAIL FROM:egal@weg.te`
`RCPT TO:empf@srv-dom.de`
`DATA`
`From: egal@weg.te`
`To: empf@srv-dom.de`
 	
`Subject: Thema`
`Dies ist der Inhalt.`
`.`
`QUIT`
	`250 smtp.srv-dom.de Hello`
	`250 2.1.0 Sender Ok`
	`250 2.1.5 Recipient Ok`
	`354 Start mail input; end with <CRLF>.<CRLF>`
	
	`250 2.6.0  . . .  Queued mail for delivery`
	`221 Bye`

Der Mailempfänger lässt sich hoffentlich nicht durch die vorgegaukelte Absenderadresse täuschen:



Vor dem uralten „Trick“ mit einer gefälschten Mail-Absenderadresse müssen IT-Laien immer noch gewarnt werden, weil Kriminelle mit bescheidenen IT-Kennnissen, aber einer gewissen Geschicklichkeit beim sogenannten *Social Engineering* täglich versuchen, auf diese Weise an das Geld von Opfern heranzukommen, z. B.:



Noch häufiger als das SMTP-Protokoll kommt im Internet auf Anwendungsebene das HTTP-Protokoll (*Hypertext Transfer Protocol*) für den Austausch zwischen Webserver und -Browser zum Einsatz, heutzutage meist in der sicheren Variante HTTPS.

21.1.2 Optionen zur Netzwerkprogrammierung in C#

C# (bzw. die .NET - Plattform) unterstützt sowohl die Socket - orientierte TCP - Kommunikation (auf der Ebene 4/5 des OSI - Modells) als auch die Nutzung wichtiger Protokolle auf der Anwendungsebene. Unterhalb der Socket - Ebene ist keine Netzwerkprogrammierung mit verwaltetem Code (IL) möglich. Entsprechende Dienste aus dem Windows-API können jedoch über die *P/Invoke* - Technik genutzt werden.

Die im Kapitel 21 zu behandelnden BCL-Klassen zur Netzwerkprogrammierung befinden sich meist in den Namensräumen **System.Net** und **System.Net.Sockets**.[1]

[1] Im .NET Framework ist ein Verweis auf das GAC - Assembly **System** erforderlich.

21.1.3 Datenstromtechnik

Bei der Netzwerkprogrammierung können die im Kapitel 16 beschriebenen Stromverarbeitungsklassen intensiv genutzt werden, weil zwischen Dateien einerseits und Netzwerkverbindungen ab der OSI-Ebene 5 (bzw. bei einer etablierten TCP-Verbindung) andererseits hinsichtlich der Daten-ein- und Ausgabe viele Gemeinsamkeiten bestehen. Ein Objekt aus der von **Stream** abgeleiteten Klasse **NetworkStream** kann mit Hilfe der elementaren **Stream**-Methoden **Read()** und **Write()** (siehe Abschnitt 16.1) Daten austauschen. Auf einem **NetworkStream**-Objekt können Stromverarbeitungsklassen für höhere Datentypen aufsetzen (z. B. **BinaryReader** und **BinaryWriter**, siehe Abschnitt 16.3).

Der wesentliche Unterschied zwischen den beiden Anwendungen des generellen Datenstromkonzepts besteht darin, dass ein **NetworkStream**-Objekt im Unterschied zu einem **FileStream**-Objekt *keine Position* besitzt. Infolgedessen besitzt die Eigenschaft **CanSeek** bei einem **NetworkStream**-Objekt immer den Wert **false**. Weil die Methode **Seek()** keine Position ansteuern und die Eigenschaft **Position** keine Position liefern kann, werfen beide eine **NotSupportedException**.

21.2 Internet - Ressourcen per Request/Response nutzen

Beim Zugriff auf eine per URI bzw. URL (siehe Abschnitt 21.2.1) beschriebene Internet-Ressource unterstützt die .NET - Plattform das **Request/Response - Muster**. Teilweise bedingt durch den technologischen Wandel stehen in der BCL für den Zugriff auf Internet-Ressourcen unterschiedliche Klassen bereit:

- **WebRequest** und **WebResponse**
  Diese seit dem .NET Framework 1.1 vorhandenen, stromorientierten Klassen bieten viel Flexibilität beim lesenden und/oder schreibenden Zugriff auf per URI identifizierte Internet-Ressourcen (z. B. Verwendung von Header-Attributen, Cookies und Timeout-Werten). Es existieren protokollspezifische Ableitungen für URIs, die mit **http:**, **https:**, **ftp:** und **file:** beginnen.
- **WebClient**
  Diese ebenfalls seit dem .NET Framework 1.1 vorhandene Klasse wurde als Verpackung für die Klassen **WebRequest** und **WebResponse** entwickelt, um deren Verwendung zu vereinfachen. Microsoft empfiehlt, bei Neuentwicklungen statt der Klasse **WebClient** die mit dem .NET Framework 4.5 eingeführte Klasse **HttpClient** zu verwenden, die wir schon im Abschnitt 17.5.9 verwendet haben.[1]
- **HttpClient**
  Die relativ junge Klasse **HttpClient** ist bei modernen HTTP(S) - Anwendungen im Vorteil, z.B. beim Zugriff auf REST-basierte Dienste.

21.2.1 URI bzw. URL

Internet-Ressourcen werden über einen sogenannten *Uniform Resource Identifier* (*URI*) beschrieben. Statt *URI* wird oft die weitgehend synonyme Bezeichnung *URL* verwendet (*Uniform Resource Locator*). Ein URI wie z. B.

[1] https://docs.microsoft.com/en-us/dotnet/api/system.net.webclient

https://www.egal.de:81/cgi/beispiel/cgi.php?vorname=Kurt

ist folgendermaßen aufgebaut:

Syntax:	*Proto-koll*	**://**	*User:Pass@* (optional)	*Domänen-name*	**:***Port* (optional)	*Pfad*	**?***URL-Parameter* (optional)
Beispiel:	https	://		www.egal.de	:81	/cgi/beispiel/cgi.php	?vorname=Kurt

Ein **Domänenname** startet auf der rechten Seite mit dem Namen einer Top-Level - Domäne (im Beispiel: **de**). Von rechts nach links folgt mindestens ein Subdomänenname. Im Beispiel sind zwei Subdomänennamen vorhanden: **www.egal**. Zwischen den Namenssegmenten steht ein Punkt. Genau genommen sollte der Domänenname mit einem Punkt enden, der für die Wurzeldomäne steht. Dieser Punkt wird aber in der Regel weggelassen.

Wenn zu einem Domänennamen ein DNS-Eintrag (*Domain Name System*, siehe Abschnitt 21.4) vorhanden ist (also mindestens eine IP-Adresse existiert), dann liegt ein **Hostname** vor. In dieser Situation muss sich der einleitende Subdomänenname aber nicht unbedingt auf einen konkreten Rechner beziehen. Es ist z. B. sehr unwahrscheinlich, dass **www.google.com** der Name *eines* konkreten Rechners ist.[1]

Die URL- bzw. URI-Parameter dienen zur Anforderung von individuellen bzw. dynamisch erstellten Webseiten unter Verwendung der **GET**-Methode im HTTPS-Protokoll (siehe Abschnitt 21.2.4.2). Durch das Zeichen **&** getrennt dürfen auch *mehrere* Parameter (Name-Wert - Paare) angegeben werden, z. B.:

?vorname=Kurt&nachname=Schmidt

Bei vielen *statischen* Webseiten kann am Ende der Pfadangabe durch # eingeleitet noch ein seiteninternes Sprungziel genannt werden, z. B.:

https://de.wikipedia.org/wiki/Port_(Protokoll)#Dynamic_Ports

21.2.2 WebClient

Die Klasse **WebClient** reduziert durch das Verpacken der Klassen **WebRequest** und **WebResponse** den Programmieraufwand. Sie eignet sich zum Herunter- bzw. Hochladen von Dateien, Zeichenfolgen und Byte-Arrays. Obwohl die Klasse von Microsoft für Neuentwicklungen *nicht* mehr empfohlen wird, nutzen wir sie zum Herunterladen einer Datei:

---

[1] Zu den Begriffen *Domain*, *Subdomain* und *Host* siehe:

- https://datatracker.ietf.org/doc/html/rfc1034
- https://datatracker.ietf.org/doc/html/rfc1035

```
using System;
using System.Net;

class FileDownload {
   static void Main(string[] args) {
      if (args.Length < 2) {
         Console.WriteLine("Aufruf: FileDownload <URI> <Dateiname>");
         return;
      }
      try {
         WebClient client = new();
         Console.WriteLine("Download startet");
         client.DownloadFile(args[0], args[1]);
         Console.WriteLine("Fertig!");
      } catch(Exception e) {
         Console.WriteLine(e.Message);
      }
   }
}
```

Man verwendet die **WebClient**-Methode **DownloadFile()** und überlässt Routinearbeiten wie das Öffnen und Schließen von Dateien den BCL-Programmierern. Im ersten Parameter nennt man die Webadresse (den URI) und im zweiten Parameter den lokalen Dateinamen. Das Beispielprogramm erwartet diese Informationen in Befehlszeilenargumenten.

Hier ist ein erfolgreicher Einsatz zu sehen:

Die Klasse **WebClient** bietet zur Methode **DownloadFile()** auch zwei asynchron arbeitende Alternativen an:

- Die mit dem .NET Framework 4.5 eingeführte Methode **DownloadFileTaskAsync()** verwendet die zum asynchronen Programmieren zu bevorzugende TPL (*Task Parallel Library*).
- Die mit dem .NET Framework 2.0 eingeführte Methode **DownloadFileAsync()** verwendet das veraltete EAP (*Event-based asynchronous Pattern*, siehe Abschnitt 17.6.2).

Zwar implementiert **WebClient** die Schnittstelle **IDisposable**, doch geschieht dies laut Albahari & Johannsen (2020, S. 694) nur aufgrund der Basisklasse **Component**, die zur Aufnahme in die Steuerelemente-Toolbox im Visual Studio erforderlich ist. Die Autoren raten zum Verzicht auf die Methode **Dispose()**, weil diese keine nützlichen Aktivitäten entfalte. Dieser Zustand kann sich natürlich bei jedem .NET - Update ändern. Wenn die plausibel klingende Vermutung von Albahari & Johannsen stimmt, dann werden die Entwickler mit einer unbegründeten Forderung zum Aufruf der **Dispose()** - Methode belastet. Wenn das öfter passiert und publik wird, dann schwindet die Motivation zur Beachtung der **IDisposable** - Regel.

21.2.3 WebRequest und WebResponse

Die Klassen **WebRequest** und **WebResponse** bieten aufgrund ihrer Flexibilität (z.B. bei der Anzahl der unterstützten Protokolle) in vielen Situationen eine gute Lösung. Ihre Verwendung wird anhand von zwei Beispielen demonstriert.

21.2.3.1 HTML-Code abrufen

Beim Zugriff auf Webinhalte mit Hilfe der Klassen **WebRequest** und **WebResponse** ruft man zunächst die statische Methode **Create()** der Klasse **WebRequest** mit einem URI als Parameter auf, um ein Objekt aus einer zum Protokoll passenden **WebRequest** - Ableitung zu erhalten, z. B.:

```
WebRequest request = WebRequest.Create("https://www.uni-trier.de/");
```

Enthält der URI z. B. die Protokollbezeichnung *http* (*Hyper Text Transfer Protocol*) oder *https* (sicheres HTTP), dann liefert **Create()** ein Objekt der Klasse **HttpWebRequest**.

Ist eine protokollspezifische Konfiguration der Anforderung erforderlich, dann kann man nach einer expliziten Typumwandlung die entsprechenden Eigenschaften der zugehörigen **WebRequest** - Ableitung ansprechen. Aufgrund der folgenden **UserAgent** - Manipulation stellt sich ein C# - Programm beim Webserver als Firefox - Browser vor:

```
((HttpWebRequest)request).UserAgent =
  "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:89.0) " +
  "Gecko/20100101 Firefox/89.0";
```

Mit der **WebRequest**-Methode **GetResponse()** fordert man die Antwort des Servers an, z. B.

```
using (WebResponse response = request.GetResponse()) { ... }
```

Nach dem **GetResponse()** - Aufruf sind keine weiteren Anforderungen durch dasselbe **WebRequest**-Objekt möglich; ein Versuch führt zu einer **InvalidOperationException**:

```
Unhandled exception. System.InvalidOperationException: This operation cannot be
performed after the request has been submitted.
```

Das von **GetResponse()** gelieferte Objekt aus einer **WebResponse** - Unterklasse (im Beispiel: **HttpWebResponse**) bietet Eigenschaften mit Metainformationen zur Serverantwort, die teilweise protokollspezifisch und daher erst nach einer Typumwandlung zugänglich sind, z. B.:

```
Console.WriteLine("Letzte Änderung:\t"+
  ((HttpWebResponse)response).LastModified);
```

An den eigentlichen Inhalt kommt man über ein **Stream**-Objekt heran, das die **WebResponse** - Methode **GetResponseStream()** liefert, z. B.:

```
using (Stream responseStream = response.GetResponseStream()) {
   string cs = ((HttpWebResponse)response).CharacterSet.ToUpper();
   Encoding enc;
   switch (cs) {
      case "UTF-8":
      case "ISO-8859-1":
         enc = Encoding.GetEncoding(cs); break;
      default:
         enc = Encoding.Default; break;
   }
   using StreamReader reader = new(responseStream, enc);
   string s;
   int i = 0;
   while ((s = reader.ReadLine()) != null && i++ < 20)
      Console.WriteLine(s);
}
```

Zum Lesen einer Serverantwort mit einem bestimmten Zeichensatz eignet sich ein entsprechend konfigurierter **StreamReader** (vgl. Abschnitt 16.3.2). Allerdings setzen aktuell (Juli 2021) ca. 97% aller Webseiten den Zeichensatz UTF-8 ein, der vom **StreamReader**-Konstruktor als Voreinstellung verwendet wird, sodass die Berücksichtigung abweichender Zeichensätze kaum noch erforder-

lich ist.[1] Ein Aufruf der **StreamReader** - Methode **ReadLine()** liefert die nächste Zeile der Serverantwort.

Nach dem Lesen der Serverantwort sollte die **WebResponse**-Methode **Dispose()** oder **Close()** entweder explizit oder implizit im Rahmen einer **using**-Anweisung aufgerufen werden, um die Ressourcen der Verbindung freizugeben.

Das folgende Programm zeigt die Schritte im Zusammenhang und verzichtet der Einfachheit halber auf die bei Netzverbindungen empfehlenswerte Ausnahmebehandlung:

```
using System;
using System.IO;
using System.Net;
using System.Text;

class RequestResponse {
   static void Main() {
      // Request-Objekt zu einem URI erzeugen
      WebRequest request = WebRequest.Create("https://www.uni-trier.de/");

      // Protokollspezifische Request-Konfiguration
      ((HttpWebRequest)request).UserAgent =
        "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:89.0) " +
        "Gecko/20100101 Firefox/89.0";

      // Response anfordern
      using (WebResponse response = request.GetResponse()) {
         // Zugriff auf Metainformation
         Console.WriteLine("Letzte Änderung:\t" +
            ((HttpWebResponse)response).LastModified);
         Console.WriteLine("Zeichensatz:    \t" +
            ((HttpWebResponse)response).CharacterSet);
         Console.WriteLine("Mit Enter weiter zum Inhalt");
         Console.ReadLine();

         // Zugriff auf den Strom mit der Serverantwort
         using (Stream responseStream = response.GetResponseStream()) {
            // StreamReader mit passender Codierung erstellen
            string cs = ((HttpWebResponse)response).CharacterSet.ToUpper();
            Encoding enc;
            switch (cs) {
               case "UTF-8":
               case "ISO-8859-1":
                  enc = Encoding.GetEncoding(cs); break;
               default:
                  enc = Encoding.Default; break;
            }
            using StreamReader reader = new(responseStream, enc);
            // 20 Zeilen der Serverantwort ausgeben
            string s;
            int i = 0;
            while ((s = reader.ReadLine()) != null && i++ < 20)
               Console.WriteLine(s);
         }
      }
   }
}
```

Ein Programmlauf (am 08.07.2021) bringt das folgende Ergebnis:

---

[1] https://w3techs.com/technologies/overview/character_encoding

```
Letzte Änderung:        08.07.2021 05:10:29
Zeichensatz:            utf-8
Mit Enter weiter zum Inhalt

<!DOCTYPE html>
<html dir="ltr" lang="de-DE">
<head>

<meta charset="utf-8">
<!--
This website is powered by TYPO3 - inspiring people to share!
TYPO3 is a free open source Content Management Framework initially created by Kasper Skaarhoj and licensed under GNU/GPL.
TYPO3 is copyright 1998-2021 of Kasper Skaarhoj. Extensions are copyright of their respective owners.
Information and contribution at https://typo3.org/
-->


<link rel="shortcut icon" href="/typo3conf/ext/zimktheme_unitrier/Resources/Public/Icons/favicon_transparent.ico"
type="image/vnd.microsoft.icon">

<meta name="generator" content="TYPO3 CMS" />
<meta name="description" content="Studieren und forschen auf einem grünen Campus in der ältesten Stadt Deutschlands.
Schwerpunkte sind Geistes-, Sozial- und Umweltwissenschaften." />
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
<meta name="robots" content="robots.txt" />
<meta name="application-name" content="Universität Trier" />
  . . .
  . . .
```

21.2.3.2 Datei herunterladen

Die Klassen **WebRequest** und **WebResponse** verursachen beim Herunterladen einer Datei etwas mehr Schreibaufwand als die im Abschnitt 21.2.2 beschriebene Klasse **WebClient**, bieten aber auch mehr Flexibilität (z.B. Header-Daten für die Anforderung und für die Antwort). Das in diesem Abschnitt vorgestellte Download-Programm wertet drei Befehlszeilenargumente aus:

- Webadresse der Quelldatei
- lokaler Dateiname
- Datum der letzten Änderung im Format jjjj.mm.tt

Das Datum wird für die **HttpWebRequest**-Eigenschaft **IfModifiedSince** verwendet, sodass der angesprochene Webserver statt der angeforderten Datei eine Fehlermeldung liefert, wenn das Änderungsdatum der Datei weiter zurückliegt. Auf diese und andere Server-Fehlermeldungen reagiert die **HttpWebRequest** - Methode **GetResponse**() mit einer **WebException**:

```
using System;
using System.IO;
using System.Net;

class FileDownloadWR {
 static void Main(string[] args) {
  if (args.Length < 3) {
   Console.WriteLine("Aufruf: filedownload <URI> <Dateiname> <jjjj.mm.tt>");
   return;
  }

  try {
   DateTime date = new(Convert.ToInt32(args[2].Substring(0, 4)),
                       Convert.ToInt32(args[2].Substring(5, 2)),
                       Convert.ToInt32(args[2].Substring(8, 2)));
   WebRequest request = WebRequest.Create(args[0]);
   ((HttpWebRequest)request).IfModifiedSince = date;
```

```
      using (WebResponse response = request.GetResponse())
       using (Stream ws = response.GetResponseStream())
       using (FileStream fs = new(args[1], FileMode.Create)) {
          Console.WriteLine("Download beginnt ...");
          ws.CopyTo(fs);
          Console.WriteLine("Fertig: " + fs.Length + " Bytes übertragen");
       }
    } catch (FormatException) {
        Console.WriteLine("Korrektes Datumsformat: jjjj.mm.tt");
    } catch (WebException we)
        when ((we.Response as HttpWebResponse).StatusCode == HttpStatusCode.NotModified) {
     Console.WriteLine($"Datei nicht geändert seit: {args[2]}");
    } catch (Exception e) {
       Console.WriteLine(e.Message);
    }
  }
}
```

Beim folgenden Einsatz wurde die gesuchte Datei in einer neuen Version angetroffen:

Man muss nicht unbedingt eine eigene Fehlermeldung für den Fall einer unveränderten Datei formulieren, weil die **Message**-Eigenschaft der Klasse **WebException** in dieser Situation ausreichend informativ ist:

```
The remote server returned an error: (304) Not Modified.
```

Zur synchronen **Stream**-Methode **CopyTo()**,

public void CopyTo (Stream *destination***)**

die im Programm den von der **WebResponse**-Methode **GetResponseStream()** gelieferten Strom in ein **FileStream**-Objekt umleitet, gibt es eine asynchrone, mit TPL-Technik arbeitende Alternative:

public Task CopyToAsync(Stream destination, CancellationToken cancellationToken)

21.2.4 HttpClient

Die Klasse **HttpClient** bietet als relativ moderne und bequeme Verpackung von **HttpWebRequest** und **HttpWebResponse** einige Vorteile, z. B.:

- Es sind *mehrere* Anforderungen durch *dasselbe* Objekt möglich, während Objekte der **HttpWebRequest** und **WebClient** nur für *eine* Anforderung taugen.
- REST-basierte Webdienste werden gut unterstützt.

Die älteren Klassen **WebRequest**, **WebResponse** und **WebClient** haben allerdings gegenüber **HttpClient** den Vorteil, dass sie auch die Protokolle **ftp** und **file** unterstützen.

Ein **HttpClient**-Objekt ist explizit *nicht* für die *einmalige* Verwendung gedacht, sondern für die Versorgung einer kompletten Anwendung bei beliebig vielen Anforderungen:[1]

```
HttpClient is intended to be instantiated once per application, rather than per-use.
```

Weil wichtige Methoden der Klasse **HttpClient** thread-sicher sind, kann das solitäre **HttpClient**-Objekt einer Anwendung bedenkenlos simultan durch mehrere Threads verwendet werden.

---

[1] https://docs.microsoft.com/en-us/dotnet/api/system.net.http.httpclient

Wir erinnern im Abschnitt 21.2.4.1 an den bereits im Kapitel 17 vorgeführten Einsatz der Klasse **HttpClient** zum asynchronen Abruf einer Webseite und verwenden die Klasse im Abschnitt 21.2.4.2.1 dazu, um dynamisch erstellte Webseiten anzufordern.

Eine ausführliche Beschreibung der Klasse **HttpClient** findet sich z. B. bei Albahari & Johannsen (2020, S. 696ff).

21.2.4.1 HTML-Code abrufen

Im Abschnitt 17.5.9 haben wir in einer WPF-Anwendung die Methode `GetHtmlCodeAsync()` erstellt, um den per URI identifizierten HTML-Code asynchron herunterzuladen und in eine lokale Datei zu befördern, ohne dabei den UI-Thread zu blockieren. Bei dieser Lösung wird mit Hilfe der asynchronen **HttpClient**-Methode **GetStreamAsync()** der HTML-Code als Datenstrom bezogen, der anschließend durch die asynchrone **Stream**-Methode **CopyToAsync()** in das **FileStream**-Objekt umgeleitet, also in die angebundene Datei geschrieben wird.

```
private async Task GetHtmlCodeAsync(string uri) {
 try {
  using Stream stream = await client.GetStreamAsync(uri).ConfigureAwait(false);
  using var fileStream = new FileStream("demo.html", FileMode.Create,
                          FileAccess.Write, FileShare.None, 4096, useAsync: true);
  await stream.CopyToAsync(fileStream).ConfigureAwait(false);
 } catch (Exception ex) {
  MessageBox.Show(this, ex.ToString(), ex.Message);
 }
}
```

Für die **Task<Stream>** - Rückgabe von **GetStreamAsync()** sowie für die **Task**-Rückgabe von **CopyToAsync()** wird per `ConfigureAwait(false)` angeordnet, dass die zugehörige Fortsetzungsmethoden *nicht* in dem beim Methodenstart gültigen Synchronisierungskontext (im Beispiel: im UI-Thread), sondern in einem Pool-Thread ausgeführt werden sollen (siehe Abschnitt 17.5.5).

Fast alle Methoden der Klasse **HttpClient** arbeiten asynchron mit TPL-Technik.

21.2.4.2 Dynamisch erstellte Webseiten per GET oder POST anfordern

Im bisherigen Verlauf von Abschnitt 21.2 haben wir *statische* Webangebote (HTML-Seiten oder Dateien) in C# - Programmen angefordert. Nun beschäftigen wir uns mit dem Abruf von *dynamisch* erstellten Webseiten durch C# - Programme. Wie zu Beginn von Kapitel 21 angekündigt, ist es aus Zeitgründen leider nicht möglich, die Realisation von dynamischen Webangeboten durch serverseitige C# - Programme zu behandeln.

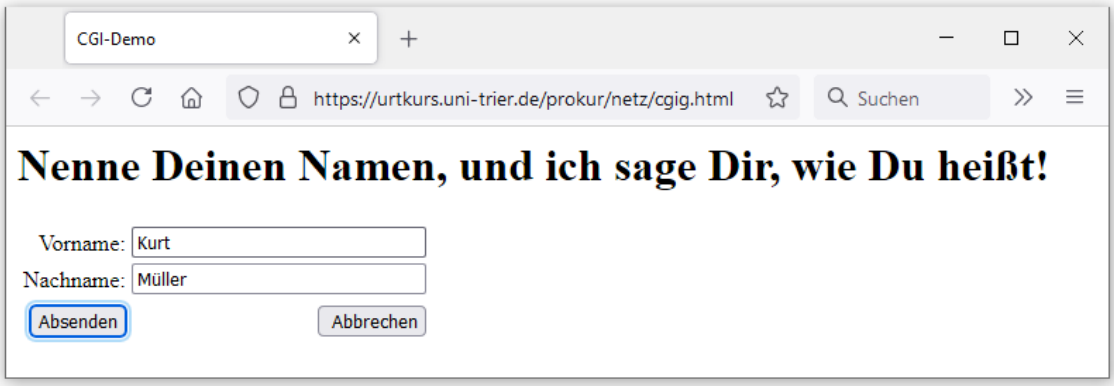
21.2.4.2.1 CGI-Muster und Beispiel

Web-Angebote beschränken sich in der Regel nicht darauf, statische HTML-Seiten und sonstige Dateien bereitzuhalten, sondern beherrschen auch verschiedene Technologien, um HTML-Seiten dynamisch nach Kundenwunsch zu erzeugen und an Klientenprogramme (meist WWW-Browser) auszuliefern (z. B. mit den Ergebnissen eines Suchauftrags oder mit einer individuellen Produktkonfiguration). Die Nutzer äußern ihre Wünsche, indem sie per Browser eine Formularseite (mit Eingabeelementen wie Textfeldern, Kontrollkästchen usw.) ausfüllen und ihre Daten zum Webserver übertragen. Dieses Programm (z. B. Apache HTTP Server, MS Internet Information Server) analysiert und beantwortet die Formulardaten aber nicht selbst, sondern überlässt diese Arbeit externen Anwendungen, die in unterschiedlichen Programmier- bzw. Skriptsprachen erstellt werden können (z. B. C#, Java, PHP oder Perl). Ursprünglich kooperierte ein Webserver mit einem Ergänzungsprogramm über das sogenannte *Common Gateway Interface* (CGI), wobei das Ergänzungsprogramm bei jeder Anforderung neu gestartet und nach dem Erstellen der HTML-Antwortseite wieder beendet wurde. Längst haben sich jedoch Lösungen etabliert, die stärker mit dem Webserver verzahnt sind, permanent im Speicher verbleiben und so eine bessere Leistung bieten (z. B. PHP als Apache -

Modul). So wird vermieden, dass bei jeder Anforderung ein Programm (z. B. der PHP-Interpreter) gestartet und eventuell auch noch eine Datenbankverbindung aufwändig hergestellt werden muss. Wir werden anschließend der Einfachheit halber alle Verfahren zur dynamischen Produktion individueller HTML-Seiten als *CGI - Lösungen* bezeichnen.

Der Browser zeigt eine vom Webserver erhaltene HTML-Seite mit Formularelementen an, über die ein Benutzer seine Wünsche artikulieren kann. Aus den Formulareinträgen erstellt der Browser eine CGI-Anfrage, die er unter Verwendung der Methoden **GET** oder **POST** aus dem HTTP(S) - Protokoll an den Webserver übermittelt. Zur Erläuterung technischer Details betrachten wir ein sehr einfaches Formular, das ein in PHP realisiertes CGI-Skript auf einem Webserver anspricht.

In diesem Browser-Fenster



ist die HTML-Seite zu sehen, die über den URI

https://urtkurs.uni-trier.de/prokur/netz/cgig.html

abrufbar ist und den folgenden HTML-Code mit einem Formular enthält:

```
<html>
<head>
<title>CGI-Demo</title>
</head>
<h1>Nenne Deinen Namen, und ich sage Dir, wie Du hei&szlig;t!</h1>
<form method="get" action="cgig.php">
<table border="0" cellpadding="0" cellspacing="4">
 <tr>
  <td align="right">Vorname:</td>
  <td><input name="vorname" type="text" size="30"></td>
 </tr><tr>
  <td align="right">Nachname:</td>
  <td><input name="nachname" type="text" size="30"></td>
 </tr>
 <tr> </tr>
 <tr>
  <td align="right"> <input type="submit" value=" Absenden "> </td>
  <td align="right">  <input type="reset" value=" Abbrechen"> </td>
  </td>
 </tr>
</table>
</form>
</html>
```

Bei der **GET**-Technik, die man im **form** - Tag einer HTML-Seite durch die Angabe

```
method="get"
```

wählt, schickt der Browser die Formulardaten als URL-Parameter an den Webserver (siehe Abschnitt 21.2.1). Die Formulardaten werden als Name-Wert - Paare am Ende der URI-Zeichenfolge

hinter einem Fragezeichen angehängt, wobei zwei Formularfelder jeweils durch ein &-Zeichen getrennt werden. Im Beispiel mit den Feldern bzw. Parametern `vorname` und `nachname` (siehe HTML-Quellcode) resultieren die folgenden URL-Parameter:

```
vorname=Kurt&nachname=Müller
```

Weil nach Eintreffen der Antwortseite die zugrundeliegende Anforderung in der Adresszeile des Browsers erscheint, kann die **GET**-Syntax dort inspiziert werden (siehe unten).

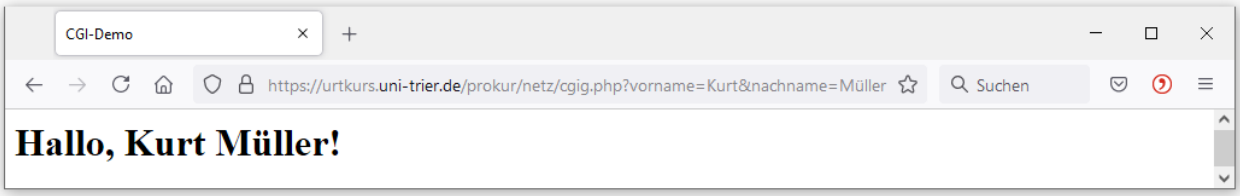
Der Webserver gibt die URL-Parameter an das im **action**-Attribut der Formulardefinition angegebene Programm weiter. Im Beispiel handelt es sich um ein **PHP**-Skript, das wenig kreativ aus den übergebenen Namen einen Gruß formuliert:[1]

```
<?php
 $vorname = $_GET["vorname"];
 $nachname = $_GET["nachname"];
 echo "<html>\n<head><title>CGI-Demo</title>\n</head>\n";
 echo "<body>\n<h1>Hallo, ".$vorname." ".$nachname."!</h1>\n</body>\n</html>";
?>
```

In diesem PHP-Skript wird die auszugebende HTML-Seite über **echo**-Kommandos an die Standardausgabe geschickt, und der Webserver befördert die PHP-Produktion über das HTTP(S) - Protokoll zum Browser, der den empfangenen HTML-Quellcode

```
<html>
<head>
   <title>CGI-Demo</title>
   <meta http-equiv="content-type" content="text/html; charset=UTF-8">
</head>
<body>
   <h1>Hallo, Kurt Müller!</h1></body>
</html>
```

anzeigt:



21.2.4.2.2 GET

Zum Versand von Formulardaten bzw. Request-Parametern an einen Webserver kennt das HTTP(S) - Protokoll die Methoden **GET** und **POST**, die nun in C# - Programmen realisiert werden sollen. Bei der bereits im Abschnitt 21.2.4.2.1 erläuterten **GET**-Technik schickt der Browser die Formulardaten als URL-Parameter an den Webserver. Daraus ergibt sich eine Längenbeschränkung, wobei die konkreten Maximalwerte vom Server und vom Browser abhängen. Man sollte vorsichtshalber eine URI-Gesamtlänge von 2048 Zeichen einhalten und ggf. das POST-Verfahren (siehe Abschnitt

---

[1] Obwohl die PHP-Implementierung des CGI-Beispiels für die von uns geplante Erstellung von klientenseitigen C# - Programmen zur Anforderung von dynamischen Webseiten keine Bedeutung hat, werden für Interessierte einige Details genannt: Der Webserver schreibt die URL-Parameter in eine Umgebungsvariable namens **QUERY_STRING** und stellt auf analoge Weise der CGI-Software noch weitere Informationen zur Verfügung, z. B.:

```
QUERY_STRING="vorname=Kurt&nachname=Müller"
REMOTE_PORT="56368"
REQUEST_METHOD="GET"
```

Das mit der **GET**-Technik arbeitende PHP-Skript greift auf die URL-Parameter in der Umgebungsvariablen **QUERY_STRING** über den superglobalen Array **$_GET** zu.

21.2.4.2.3), das keine praxisrelevante Längenbeschränkung kennt, zur Übergabe von Request-Parametern verwenden.

Um in C# eine CGI-Software anzusprechen, die per **GET**-Technik mit Request-Parametern versorgt werden möchte, müssen die oben für den Zugriff auf statische Webseiten erstellten Programme nicht wesentlich geändert werden.

Per **GET**-Technik eine dynamisch erstellte Webseite in einem C# - Programm anzusprechen, erfordert wenig Änderungen gegenüber dem Zugriff auf eine statische Webseite. Im folgenden Programm kommt ein Objekt der Klasse **HttpClient** zum Einsatz.[1] Es wird gemäß einer Empfehlung von Microsoft als *statisches* Feld definiert, weil ein **HttpClient**-Objekt nicht (wie ein **HttpWebRequest**- oder **WebClient**-Objekt) für die *einmalige* Verwendung gedacht ist, sondern für die Versorgung einer kompletten Anwendung bei beliebig vielen Anforderungen. Obwohl die Klasse **HttpClient** die Schnittstelle **IDisposable** implementiert, muss man die lästige Frage nach dem richtigen Zeitpunkt für einen **Dispose()** - Aufruf nicht sonderlich ernst nehmen. Wenn in einer Anwendung definitiv keine weitere Nutzung eines **HttpClient**-Objekts mehr stattfindet, spricht natürlich nichts gegen einen **Dispose()** - Aufruf.

```
using System;
using System.Net.Http;
using System.Threading.Tasks;
using System.Web;

class CgiGet {
 static HttpClient client = new();

 static async Task Main() {
  string cgiuri = "https://urtkurs.uni-trier.de/prokur/netz/cgig.php";

  Console.WriteLine("Geben Sie bitte Ihren Vornamen an");
  Console.Write("Vorname: ");
  string vorName = HttpUtility.UrlEncode(Console.ReadLine());

  Console.WriteLine("Geben Sie bitte Ihren Nachnamen an");
  Console.Write("Nachname: ");

  string nachName = HttpUtility.UrlEncode(Console.ReadLine());
  Console.WriteLine("\nDie Anforderung wird übertragen ...");

  try {
   var response = await client.GetAsync(cgiuri + "?vorname=" +
                                        vorName + "&nachname=" + nachName);
   if (response.IsSuccessStatusCode)
    Console.WriteLine("\nAntwort:\n" +
                      await response.Content.ReadAsStringAsync());
  } catch (Exception e) {
   Console.WriteLine(e.Message);
  }
 }
}
```

Mit der statischen **HttpUtility**-Methode **UrlEncode()** wird für die korrekte URL-Codierung der **GET**-Parameter gesorgt, z. B.:

---

[1] Im .NET Framework benötigt das Programm bzw. Projekt einen Verweis auf ...

- das GAC-Assembly **System.Net.Http**, weil die Klasse **HttpClient** (aus dem Namensraum **System.Net.Http**) dort implementiert ist.
- das GAC-Assembly **System.Web**, weil die Klasse **HttpUtility** (aus dem Namensraum **System.Web**) dort implementiert ist.

- Leerzeichen werden durch ein „+“ ersetzt.
- Für die mit einer speziellen Bedeutung belasteten Zeichen (also &, +, = und %) erscheint nach einem einleitenden Prozentzeichen ihr Hexadezimalwert im Zeichensatz der Webseite (Voreinstellung: UTF-8), z. B. %26 für &.

Die Server-Antwort wird mit der asynchronen **HttpClient**-Methode **GetAsync()** angefordert. Man erhält ein Objekt der Klasse **HttpResponseMessage**, von dem durch einen Aufruf der Methode **ToString()** die Header der Server-Antwort in Erfahrung zu bringen sind, z. B.:

```
StatusCode: 200, ReasonPhrase: 'OK', Version: 1.1, Content:
System.Net.Http.HttpConnectionResponseContent, Headers:
{
  Date: Mon, 12 Jul 2021 14:11:02 GMT
  Server: Apache
  Strict-Transport-Security: max-age=0
  Upgrade: h2
  Connection: Upgrade
  Vary: Accept-Encoding
  X-Content-Type-Options: nosniff
  Transfer-Encoding: chunked
  Content-Type: text/html; charset=utf-8
}
```

Besitzt die **HttpResponseMessage**-Eigenschaft **IsSuccessStatusCode** den Wert **true**, dann kann der Inhalt der Server-Antwort über die asynchrone Methode **ReadAsStringAsync()** angefordert werden:

```
if (response.IsSuccessStatusCode)
   Console.WriteLine(await response.Content.ReadAsStringAsync());
```

Weil das Programm die vom CGI-Skript gelieferte HTML-Seite nur als Text darstellt, ist sein Auftritt nicht berauschend:

```
Geben Sie bitte Ihren Vornamen an
Vorname: Kurt
Geben Sie bitte Ihren Nachnamen an
Nachname: Müller

Die Anforderung wird übertragen ...

Antwort:
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//DE">
<html>
<head>
        <title>CGI-Demo</title>
        <meta http-equiv="content-type" content="text/html; charset=UTF-8">
</head>
<body>
        <h1>Hallo, Kurt Müller!</h1></body>
</html>
```

21.2.4.2.3 POST

Bei der **POST**-Methode des HTTP(S) - Protokolls werden die Request-Parameter (wie bei der **GET**-Methode im Format von Name-Wert - Paaren) an den Server gesendet, ohne in der Adresszeile der Antwortseite aufzutauchen. In einem C# - Programm zur Nutzung einer mit POST arbeitenden CGI-Lösung kann man mit Hilfe der konkretisierten generischen Klasse **Dictionary<String, String>** ein Objekt der Klasse **FormUrlEncodedContent** mit den korrekt kodierten POST-Parametern (siehe Abschnitt 21.2.4.2.2)

```
var formData = new Dictionary<string, string> {
```

```
                                { "vorname", vorName }, { "nachname", nachName }
};
var formDataEnc = new FormUrlEncodedContent(formData);
```

erstellen und dieses Objekt zusammen mit dem URI in einem Aufruf der **HttpClient**-Methode **PostAsync()** verwenden:

```
var response = await client.PostAsync(cgiuri, formDataEnc);
```

Vom gelieferten Objekt der Klasse **HttpResponseMessage** sind durch einen Aufruf der Methode **ToString()** die Header der Server-Antwort in Erfahrung zu bringen (siehe Abschnitt 21.2.4.2.2).

Besitzt die **HttpResponseMessage**-Eigenschaft **IsSuccessStatusCode** den Wert **true**, kann der Inhalt der Server-Antwort über die asynchrone Methode **ReadAsStringAsync()** angefordert werden:

```
if (response.IsSuccessStatusCode)
   Console.WriteLine(await response.Content.ReadAsStringAsync());
```

Es folgt das vollständige Programm:[1]

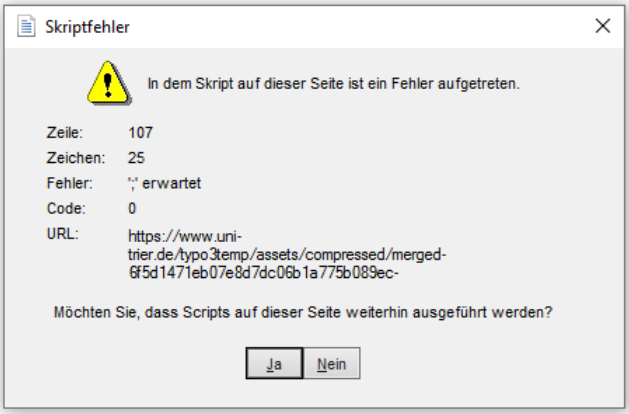
```
using System;
using System.Collections.Generic;
using System.Net.Http;
using System.Threading.Tasks;

class CgiPost {
   static HttpClient client = new();
   static async Task Main() {
      string cgiuri = "https://urtkurs.uni-trier.de/prokur/netz/cgip.php";

      Console.WriteLine("Geben Sie bitte Ihren Vornamen an");
      Console.Write("Vorname: ");
      string vorName = Console.ReadLine();

      Console.WriteLine("Geben Sie bitte Ihren Nachnamen an");
      Console.Write("Nachname: ");
      string nachName = Console.ReadLine();

      var formData = new Dictionary<string, string> {
         { "vorname", vorName }, { "nachname", nachName }
      };
      var formDataEnc = new FormUrlEncodedContent(formData);

      Console.WriteLine("\nDie Anforderung wird übertragen ...");
      try {
         var response = await client.PostAsync(cgiuri, formDataEnc);
         if (response.IsSuccessStatusCode)
            Console.WriteLine(await response.Content.ReadAsStringAsync());
      } catch (Exception e) {
         Console.WriteLine(e.Message);
      }
   }
}
```

21.3 Browser-Integration per WebView2-Steuerelement

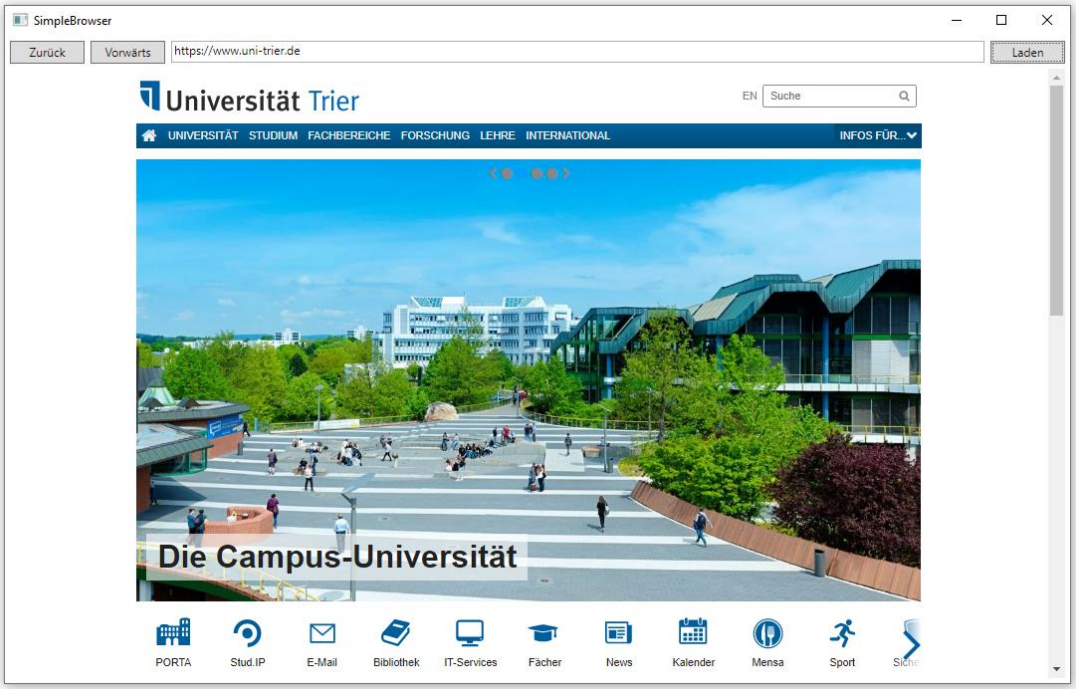
Im .NET - Namensraum **System.Windows.Controls** befindet sich die Steuerelementklasse **WebBrowser** mit der Fähigkeit, HTML-Code zu rendern. Leider scheitert das auf einer betagten Versi-

[1] Im .NET Framework benötigt das Programm bzw. Projekt einen Verweis auf das GAC-Assembly **System.Net.Http**, weil die Klasse **HttpClient** (aus dem Namensraum **System.Net.Http**) dort implementiert ist.

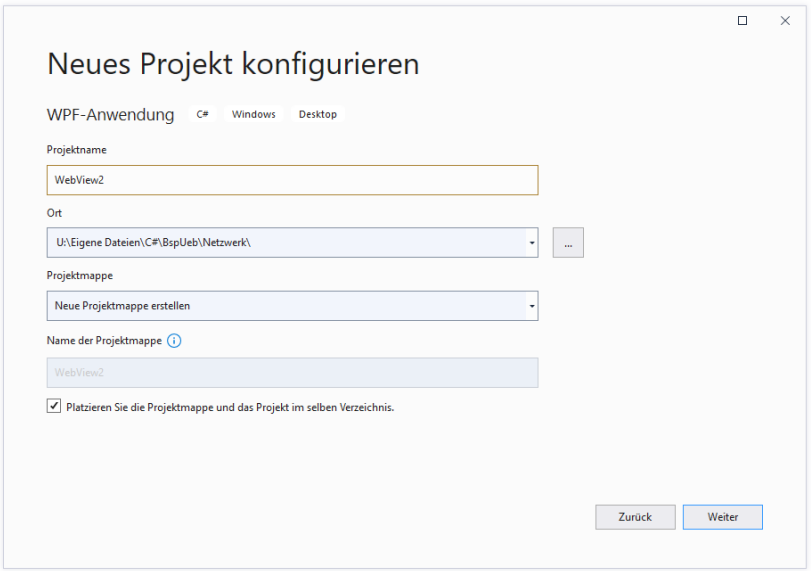
on von Microsofts Internet Explorer basierende Steuerelement oft an aktuellen Webseiten und meldet Skript-Fehler am laufenden Band, z. B.:

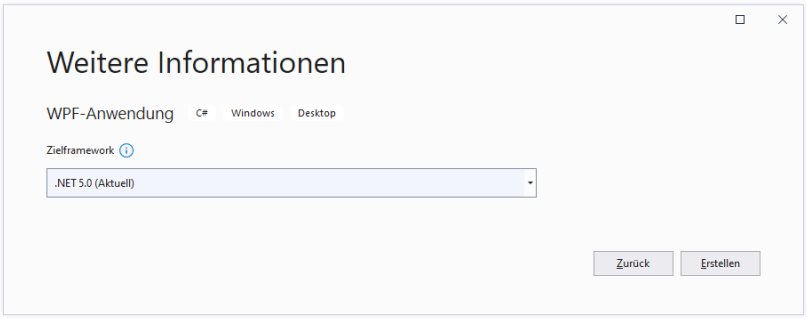


Die als NuGet-Paket verfügbare Komponente **WebView2** macht es besser, z. B.:

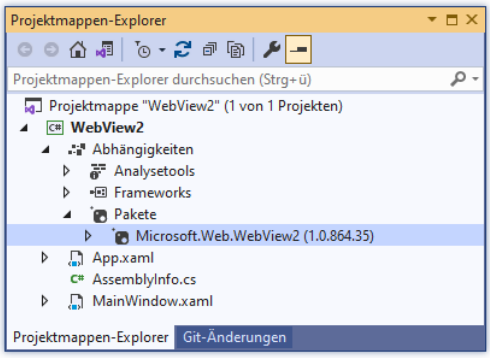


Um die Verwendung der Komponente zu üben, erstellen wir eine neue WPF-Anwendung für .NET 5.0:





Über den **NuGet-Paket-Manager** im Visual Studio, der z. B. mit dem Item **NuGet-Pakete verwalten** aus dem Kontextmenü zum Projektknoten **Abhängigkeiten** geöffnet werden kann, installieren wir das NuGet-Paket **Microsoft.Web.WebView2** mit dem folgenden Ergebnis im Projektmappen-Explorer:



Wir verwenden für das Anwendungsfenster (siehe obiges Bildschirmfoto) den folgenden XAML-Code, der als Steuerelemente drei **Button**-Objekte, ein **TextBox**-Objekt sowie die **WebView2**-Komponente enthält:

```
<Window x:Class="WebView2.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:wv2="clr-namespace:Microsoft.Web.WebView2.Wpf;assembly=Microsoft.Web.WebView2.Wpf"
    mc:Ignorable="d"
    Title="WebView2 - Browser"
    Height="536" Width="1050"
    MinWidth="600" MinHeight="400" >
```

```
    <Grid>
      <Button Content="Zurück" HorizontalAlignment="Left" Name="button1" VerticalAlignment="Top"
              Width="75" Margin="5,5,0,0" Click="button1_Click" />
      <Button Content="Vorwärts" HorizontalAlignment="Left" Margin="86,5,0,0" Name="button2"
              VerticalAlignment="Top" Width="75" Click="button2_Click" />
      <Button Content="Laden" HorizontalAlignment="Right" Margin="430,5,5,0" Name="button3"
              VerticalAlignment="Top" Width="75" Click="button3_Click" IsDefault="True" />
      <TextBox Margin="168,6,85,0" Name="textBox" VerticalAlignment="Top"
               VerticalContentAlignment="Center" />
      <wv2:WebView2 Margin="5,33,5,5" Name="webView" />
    </Grid>
  </Window>
```

Für die **WebView2**-Komponente wird ein XML-Namensraum mit dem Präfix **wv2** definiert:

```
xmlns:wv2="clr-namespace:Microsoft.Web.WebView2.Wpf;assembly=Microsoft.Web.WebView2.Wpf"
```

In der Quellcodedatei zur Hauptfensterklasse ist einiges zu tun. Zunächst wird der Namensraum importiert, in dem sich die Komponente **WebView2** befindet:

```
using Microsoft.Web.WebView2.Core;
```

Zur Initialisierung der **WebView2**-Komponente muss deren asynchrone Instanzmethode **EnsureCoreWebView2Async()** aufgerufen werden, die als Rückgabe ein **Task**-Objekt liefert. Das von einer **async**-Methode gelieferte **Task**-Objekt sollte per **await**-Ausdruck in den Programmablauf integriert werden (vgl. Abschnitt 17.5.8). Weil ein Konstruktor nicht als **async** deklariert werden darf, kann hier kein **await**-Ausdruck verwendet werden. Daher wird nach dem von Microsoft auf dieser Webseite

https://docs.microsoft.com/en-us/microsoft-edge/webview2/get-started/wpf

vorgeschlagenen Verfahren die folgende asynchrone Hilfsmethode `InitializeAsync()` mit dem Rückgabetyp **void** definiert:

```
async void InitializeAsync() {
   await webView.EnsureCoreWebView2Async(null);
}
```

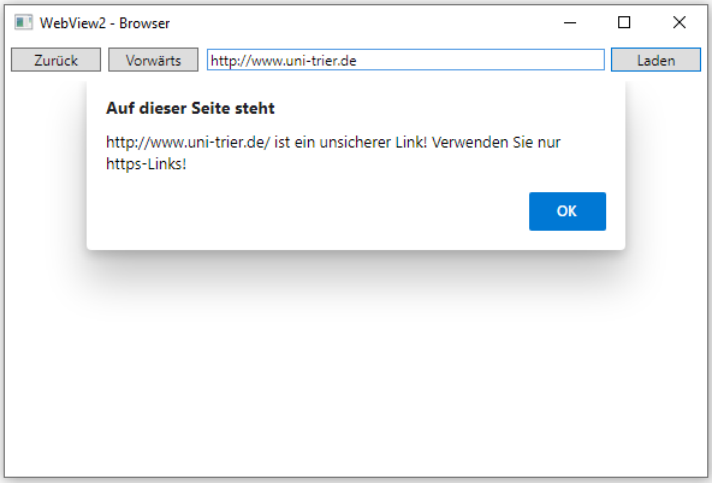
Hier ist ein **await**-Ausdruck mit dem Aufruf der Methode **EnsureCoreWebView2Async()** erlaubt. Andererseits kann `InitializeAsync()` im Hauptfenster-Konstruktor aufgerufen werden:

```
public MainWindow() {
   InitializeComponent();
   try {
      InitializeAsync();
   } catch (Exception ex) {
      MessageBox.Show(ex.ToString(), "Fehler bei der WebView2-Initialisierung");
   }
   webView.NavigationStarting += EnsureHttps;
}
```

Im Konstruktor wird außerdem die ebenfalls von der oben angegebenen Microsoft-Webseite stammende Methode `EnsureHttps()`

```
void EnsureHttps(object sender, CoreWebView2NavigationStartingEventArgs args) {
 string uri = args.Uri;
 if (!uri.StartsWith("https://")) {
  webView.CoreWebView2.ExecuteScriptAsync(
     $"alert('{uri} ist ein unsicherer Link! Verwenden Sie nur https-Links!')");
  args.Cancel = true;
 }
}
```

beim **WebView2**-Ereignis **NavigationStarting** registriert, um die Benutzer vor der Verwendung des unsicheren HTTP-Protokolls zu warnen, z. B:



Die **WebView2**-Eigenschaft **CoreWebView2** zeigt auf ein Objekt der gleichnamigen Klasse, das die Methode **Navigate()** zur Anzeige einer HTML-Seite beherrscht. In der Behandlungsmethode zum **Click**-Ereignis des `Laden`-Schalters wird diese Methode benutzt, um die Webseite mit der im **TextBox**-Steuerelement eingetragenen Adresse zu öffnen:

```
private void button3_Click(object sender, RoutedEventArgs e) {
   try {
      webView.CoreWebView2.Navigate(textBox.Text);
   } catch (Exception ex) {
      MessageBox.Show(ex.ToString(), "Fehler");
   }
}
```

In den Behandlungsmethoden zu den **Click**-Ereignissen der Schalter `Zurück` und `Vorwärts` werden die **WebView2**-Methoden **GoBack()** bzw. **GoForward()** aufgerufen:

```
private void button1_Click(object sender, RoutedEventArgs e) {
   webView.GoBack();
}

private void button2_Click(object sender, RoutedEventArgs e) {
   webView.GoForward();
}
```

Beim Programmstart im Debug-Modus der Entwicklungsumgebung zeigt eine Fehlermeldung, dass wir immer noch nicht ganz fertig sind:

Die WPF-Komponente **WebView2** stützt sich auf die **WebView2**-Runtime, wobei es sich im Wesentlichen um einen Browser mit der **Chromium**-Technik handelt.[1] Zur lokalen Installation der **WebView2**-Runtime bietet Microsoft zwei Optionen:[2]

---

1 Der Chromium-Browser ist die Open Source - Variante des Chrome-Browsers der Firma Google.

2 https://developer.microsoft.com/de-de/microsoft-edge/webview2/

- ein vollständiges Installationsprogramm
- einen Bootstrapper[1]
  Über einen Link kann man den Bootstrapper in ein eigenes Programm integrieren.

Nachdem z. B. das vollständige Installationsprogramm

MicrosoftEdgeWebView2RuntimeInstallerX64.exe

ausgeführt worden ist, klappt unser Programm mit **WebView2**-Komponente endlich.

In Windows 11 wird vermutlich die **WebView2**-Runtime enthalten sein, sodass zur Darstellung von Webinhalten in eigenen Programmen weniger Aufwand erforderlich ist (Schulz 2021).

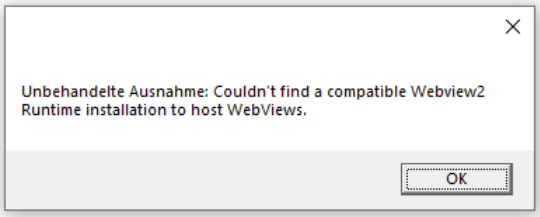
Fehlt die **WebView2**-Runtime, dann passiert aus der Sicht des Benutzers beim Programmstart *nichts*, weil das Programm aufgrund einer unbehandelten und unsichtbaren Ausnahme beendet wird. Wir erstellen daher eine Behandlungsmethode für das Ereignis **UnhandledException** der Klasse **AppDomain** (vgl. Abschnitt 17.1.8):

```
private void UnHandler(object sender, UnhandledExceptionEventArgs e) {
   MessageBox.Show($"Unbehandelte Ausnahme: {((Exception)e.ExceptionObject).Message}");
}
```

Diese Methode wird im Konstruktor beim Ereignis registriert:

```
AppDomain.CurrentDomain.UnhandledException += UnHandler;
```

Nun werden die Benutzer ggf. über die Ursache für einen gescheiterten Programmstart informiert:



Das vollständige Projekt befindet sich im Ordner

…\BspUeb\Netzwerk\WebView2

21.4 IP-Adressen bzw. Hostnamen ermitteln

Jeder an das Internet angeschlossene Rechner verfügt über (mindestens) eine **IP-Adresse** (32-bittig im IPv4, 128-bittig im IPv6) sowie über einen **Hostnamen**, wobei die Zuordnung vom **Domain Name System** (**DNS**) geleistet wird.

Die statische Methode **GetHostEntryAsync()** der BCL-Klasse **Dns** (im Namensraum **System.NET**) mit den beiden Überladungen

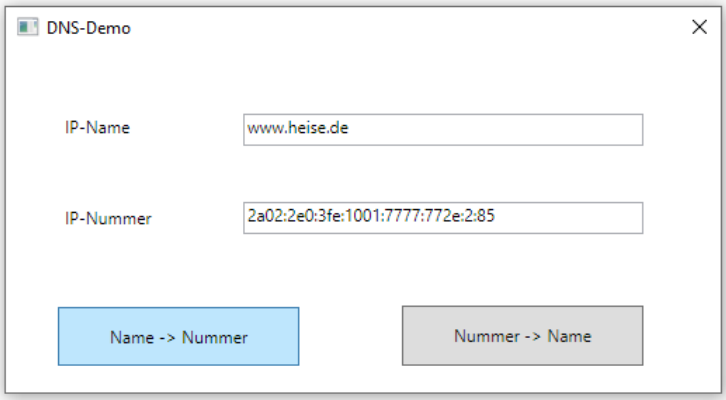
public static Task<IPHostEntry> GetHostEntryAsync(IPAddress *adresse***)**
public static Task<IPHostEntry> GetHostEntryAsync(String *nameOderAdresse***)**

nimmt eine Rechneridentifikation per **IPAddress**-Objekt oder Zeichenfolge entgegen und versucht, per DNS-Anfrage den jeweils fehlenden Bestandteil (die IP-Adresse(n) zu einem Hostnamen bzw. den Namen zu einer Adresse) zu ermitteln. Die Methode arbeitet asynchron mit TPL-Technik. Über die **Result**-Eigenschaft des **Task**-Objekts erhält man ein **IPHostEntry**-Objekt, das in seinen Eigenschaften **AddressList** bzw. **HostName** die gewünschten Daten bereit hält.

---

[1] Als *Bootstrapper* bezeichnet man ein kleines Programm, das dazu dient, das eigentliche Installationsprogramm herunterzuladen und zu starten.

Im folgenden Programm werden beide Konvertierungsrichtungen verwendet:



Vom Quellcode des Programms sind vor allem die Ereignisbehandlungsmethoden zu den beiden Schaltflächen von Interesse:

```
private async void cbToNumber_Click(object sender, RoutedEventArgs e) {
   if (tbName.Text.Length == 0)
      return;
   try {
      IPHostEntry host = await Dns.GetHostEntryAsync(tbName.Text);
      tbNumber.Text = host.AddressList[0].ToString();
   } catch (Exception ex) {
      MessageBox.Show(ex.ToString(), "Fehler");
   }
}

private async void cbToName_Click(object sender, RoutedEventArgs e) {
   if (tbNumber.Text.Length == 0)
      return;
   try {
      IPHostEntry host = await Dns.GetHostEntryAsync(tbNumber.Text);
      tbName.Text = host.HostName;
   } catch (Exception ex) {
      MessageBox.Show(ex.ToString(), "Fehler");
   }
}
```

Die Klasse **Dns** rechnet mit einer *Liste* von IP-Adressen, was bei der heutigen Vielfalt von Netzwerkadaptern in Standard-PCs (z. B. LAN, WLAN, VPN) und in Anbetracht der bei Server-PCs oft mehrfach vorhandenen Ethernet-Anschlüssen durchaus realistisch ist. Das obige Programm zeigt nur die erste IP-Adresse an:

```
tbNumber.Text = host.AddressList[0].ToString();
```

Das vollständige Projekt finden Sie im folgenden Ordner:

…\BspUeb\Netzwerk\DNS

21.5 Socket-Programmierung

Unsere bisherigen Beispielprogramme im Kapitel 21 haben meist Inhalte (HTML-Seiten, Dateien) von Webservern bezogen und dazu BCL-Klassen benutzt, die mit einem etablierten Anwendungsprotokoll (meist HTTPS) arbeiten. Im aktuellen Abschnitt gewinnen wir eine erweiterte Flexibilität durch den direkten Einsatz des TCP-Protokolls. Daraus ergibt sich z. B. die Möglichkeit, *eigene* Anwendungsprotokolle zu entwickeln. Das auf der Transport- bzw. Sitzungsebene des OSI-Modells (siehe Abschnitt 21.1.1) angesiedelte TCP-Protokoll schafft zwischen zwei (durch *Portnummern* identifizierten) Anwendungen, die sich meist auf verschiedenen (durch *IP-Adressen* identifizierten)

Rechnern befinden, eine virtuelle, *Datenstrom-orientierte* und *gesicherte* Verbindung. An beiden Enden der Verbindung steht das von praktisch allen aktuellen Programmiersprachen unterstützte *Socket-API* zur Verfügung, das in .NET - Implementationen durch Klassen im Namensraum **System.Net.Sockets** realisiert wird.

TCP-Programmierer müssen sich nicht um die Zustellung, die Integrität oder die korrekte Reihenfolge von IP-Paketen kümmern. Sie schreiben stattdessen nach den Regeln eines Protokolls der Anwendungsschicht Bytes in einen Ausgabestrom bzw. entnehmen Bytes aus einem Eingabestrom. Dabei ist der Ausgabestrom des Senders virtuell mit dem Eingabestrom des Empfängers verbunden. Außerdem sind von einer über TCP kommunizierenden Anwendung eventuell Aufgaben wie Authentifizierung, Verschlüsselung oder Komprimierung zu erledigen.

Wir beschäftigen uns in diesem Abschnitt mit der Erstellung von Klienten- *und* Serveranwendungen. Ein wesentlicher Unterschied zwischen den beiden Rollen besteht darin, dass ein Serverprogramm mehr oder weniger permanent läuft und an einem fest vereinbarten Port auf eingehende Verbindungswünsche wartet, während ein Klientenprogramm nur bei Bedarf aktiv wird und dabei einen dynamisch vom Betriebssystem zugewiesenen Port benutzt.

21.5.1 TCP-Server

Wir erstellen einen Server, der am TCP-Port 55555 lauscht und anfragenden Klienten die aktuelle Uhrzeit mitteilt. Im Konstruktoraufruf für das zentrale Objekt der Klasse **TcpListener** geben wir neben der Portnummer auch eine IP-Adresse an, z. B.

```
int svrPort = 55555;
IPAddress ip = Dns.GetHostEntry("localhost").AddressList[1];
TcpListener server = null;
 . . .
server = new TcpListener(ip, svrPort);
```

Vom veralteten Konstruktoraufruf *ohne* IP-Adresse und der damit erforderlichen automatischen Wahl einer IP-Adresse aus der Liste lokal verfügbarer Adressen hält der Compiler nichts mehr:

```
TcpListener.cs(17,12): warning CS0618:
        "System.Net.Sockets.TcpListener.TcpListener(int)" ist veraltet: "This
        method has been deprecated. Please use TcpListener(IPAddress localaddr,
        int port) instead. http://go.microsoft.com/fwlink/?linkid=14202"
```

Das vom statischen Methodenaufruf **Dns.GetHostEntry()** gelieferte **IPHostEntry**-Objekt hat auf dem Testrechner über das **AddressList**-Element mit dem Index 0 die IPv6-Adresse und über das Element mit dem Index 1 die IPv4-Adresse geliefert.

Nach dem Start des Servers

```
server.Start();
```

wird dieser durch Aufruf seiner Methode **AcceptTcpClient()** beauftragt, auf eine Verbindungsanfrage zu lauern:

```
using (TcpClient tcpClient = server.AcceptTcpClient()) { . . . }
```

Während der Wartezeit ist der aktuelle Thread durch den **AcceptTcpClient()** - Aufruf blockiert. Dieser endet erst beim Eintreffen einer Verbindungsanfrage und liefert dann ein (zum Senden und Empfangen geeignetes) **TcpClient**-Objekt zurück, das als Verpackung für die beiden folgenden Objekte dient:

- ein **Socket**-Objekt, ansprechbar über die **TcpClient** - Eigenschaft **Client**
- ein **NetworkStream**-Objekt, erreichbar über die **TcpClient**-Methode **GetStream()**

Im Beispielprogramm wird nach dem Eintreffen einer Klientenanfrage mit Hilfe der **DateTime**-Struktur eine Zeichenfolge mit der aktuellen Datum/Zeit - Kombination erstellt und unter Verwendung der ASCII-Codierung in einen Byte-Array gewandelt:

```
byte[] msg = Encoding.ASCII.GetBytes(DateTime.Now.ToString());
```

Diesen Array befördert dann die **NetworkStream**-Methode **Write()** zur Gegenstelle:

```
stream.Write(msg, 0, msg.Length);
```

Bei einer realen Server-Anwendung ist für den Informationsaustausch mit Klienten ein Anwendungsebenen-Protokoll zu verabreden, das weitaus komplexer ausfällt als die rudimentäre Variante des Beispielprogramms. U. a. ist für jede Situation zu regeln, wer gerade senden darf. Die Beschreibung einer SMTP-Kommunikation im Abschnitt 21.1.1 vermittelt einen Eindruck von einem realistischen Anwendungsebenen-Protokoll.

Am Ende einer Klientenbedienung richtet man einen **Dispose()** - Aufruf an das **TcpClient** - Objekt, der per **using**-Anweisung (siehe oben) automatisiert werden kann, um die folgenden Abschlussarbeiten auszulösen:

- Das interne **NetworkStream** - Objekt erhält einen **Dispose()** - Aufruf.
- Das interne **Socket** - Objekt wird aufgefordert, noch anstehende Übertragungen auszuführen.
- Das interne **Socket** - Objekt erhält einen **Close()** - Aufruf. Dies beendet die Verbindung zum Klienten und gibt die Netzwerk-Ressourcen (z. B. den belegten Port) an das Betriebssystem zurück. Die explizite Rückgabe von Ressourcen ist dann relevant, wenn ein Programm anschließend weiter aktiv bleibt. Bei Beendigung eines Programms werden die belegten Ressourcen automatisch freigegeben.

Das folgende Programm bedient in einer **while**-Schleife beliebig viele Klienten nacheinander:

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class TcpServer {
 static void Main() {
  int svrPort = 55555;
  IPAddress ip = Dns.GetHostEntry("localhost").AddressList[1];

  TcpListener server = null;
  NetworkStream stream = null;
  try {
   server = new TcpListener(ip, svrPort);
   server.Start();
   Console.WriteLine("Zeitserver lauscht seit " + DateTime.Now + " (IP: " +
    ip + ", Port: " + svrPort + ")");
   while (true) {
    using (TcpClient tcpClient = server.AcceptTcpClient()) {
     Console.WriteLine("\n" + DateTime.Now + " Anfrage von\n  IP-Nummer:\t" +
       (tcpClient.Client.RemoteEndPoint as IPEndPoint).Address + "\n  Port:   \t" +
       (tcpClient.Client.RemoteEndPoint as IPEndPoint).Port);
     stream = tcpClient.GetStream();
     byte[] msg = Encoding.ASCII.GetBytes(DateTime.Now.ToString());
     stream.Write(msg, 0, msg.Length);
    }
   }
  } catch (Exception e) {
   Console.WriteLine(e);
   Console.ReadLine();
  } finally {
   server.Stop();
  }
 }
}
```

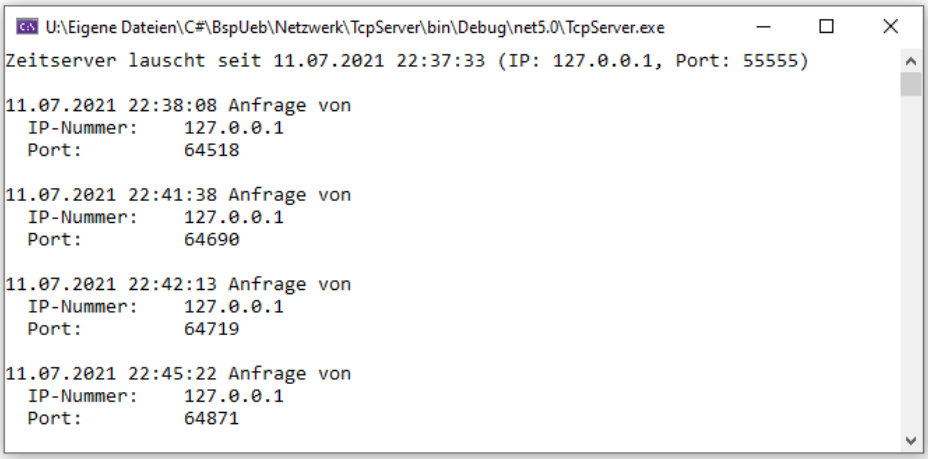
Der Kürze halber fehlt ein Verfahren zum regulären Beenden des Programms, sodass es rabiat abgebrochen werden muss.

Der Zeit-Server kann unter Windows z. B. über den Telnet-Klienten angesprochen werden, nachdem dieses Programm als Windows-Feature aktiviert worden ist (siehe Abschnitt 21.1.1). Wenn der Server auf dem lokalen Rechner läuft, kann die Zeitabfrage so vorgenommen werden:

```
>telnet localhost 55555
```

Der Server antwortet

und protokolliert seine Dienstleistungen:



So kommt das Programm an die IP-Adresse und den Port des verbundenen Klienten heran:

```
(tcpClient.Client.RemoteEndPoint as IPEndPoint).Address
(tcpClient.Client.RemoteEndPoint as IPEndPoint).Port
```

Die **RemoteEndPoint**-Eigenschaft des **Socket**-Objekts, das über die **TcpClient**-Eigenschaft **Client** angesprochen wird, zeigt auf ein Objekt vom deklarierten Datentyp **EndPoint**, das tatsächlich zur Klasse **IPEndPoint** gehört und daher die Eigenschaften **Address** und **Port** mit den gewünschten Informationen besitzt.

21.5.2 TCP-Klient

Als Gegenstück zum eben präsentierten Zeit-Server wird nun ein passendes Klienten-Programm entwickelt, wobei ein Objekt der schon bekannten Klasse **TcpClient** die zentrale Rolle spielt. Bei der gewählten Konstruktor-Überladung sind IP-Adresse und Portnummer des Servers anzugeben, z. B.:

```
string svr = "localhost";
int port = 55555;
 . . .
using (TcpClient tcpClient = new(svr, port)) { . . . }
```

Weil die Verbindung bereits im Konstruktor aufgebaut wird, setzt man seinen Aufruf am besten in einen **try**-Block.

Die empfangenen Daten stehen über ein Objekt der Klasse **NetworkStream** zur Verfügung, das von der **TcpClient**-Methode **GetStream()** geliefert wird:

```
NetworkStream stream = client.GetStream();
byte[] bZeit = new byte[48];
stream.ReadTimeout = 1000;
int nRead = stream.Read(bZeit, 0, bZeit.Length);
```

Per Voreinstellung kehrt der **Read()** - Aufruf erst dann zurück, wenn die erwarteten Daten im angesprochenen Netzwerkstrom ankommen. Über die **NetworkStream**-Eigenschaft **ReadTimeout** kann man eine Zeitspanne in Millisekunden festlegen, nach deren Ablauf der Leseversuch mit einer **IOExecption** abgebrochen werden soll. Für eine asynchrone Programmierung mit TPL-Technik steht die **Stream**-Methode **ReadAsync()** zur Verfügung (siehe Abschnitt 17.5.9).

Bei der Wandlung der erhaltenen Bytes in Unicode-Zeichen ist die vom Zeit-Server verwendete Codierung anzugeben, z. B.:

```
string sZeit = Encoding.ASCII.GetString(bZeit, 0, nRead);
```

Welche Abschlussarbeiten ein **TcpClient** - Objekt nach einem **Dispose()** - oder **Close()** - Aufruf erledigt, wurde schon im Abschnitt 21.5.1 erläutert.

Es folgt der komplette Quellcode des Klientenprogramms:

```
using System;
using System.Net.Sockets;
using System.Text;

class TcpClientDemo {
   static void Main() {
      string svr = "localhost";
      int port = 55555;
      try {
         using (TcpClient tcpClient = new(svr, port)) {
            NetworkStream stream = tcpClient.GetStream();
            byte[] bZeit = new byte[48];
            stream.ReadTimeout = 1000;
            int nRead = stream.Read(bZeit, 0, bZeit.Length);
            string sZeit = Encoding.ASCII.GetString(bZeit, 0, nRead);
            Console.WriteLine("Datum und Zeit von {0}: {1}", svr, sZeit);
         }
      } catch (Exception e) {
         Console.WriteLine(e);
      }
   }
}
```

Wenn der im Abschnitt 21.5.1 vorgestellte Server läuft, bringt das Klientenprogramm das Datum und die Uhrzeit in Erfahrung, z. B.:

21.5.3 Simultane Bedienung vieler Klienten

Bei einer ernsthaften Server-Programmierung ist eine Multithreading-Lösung erforderlich, damit mehrere Klienten simultan bedient werden können. Durch die Verwendung der *Task Parallel Library* (TPL) mit Hilfe der C# - Sprachunterstützung durch die Schlüsselwörter **async** und **await** (siehe Abschnitt 17.5) gelingt mit relativ wenig Aufwand eine leistungsfähige, zur simultanen Bedienung von vielen Klienten geeignete Server-Anwendung.

Wir transformieren den im Abschnitt 21.5.1 vorgestellten Zeit-Server und erfahren dabei, dass sich eine synchrone Lösung tatsächlich in wenigen Schritten in eine asynchrone Lösung übertragen lässt, wobei kein direkter Kontakt mit der Klasse **Thread** oder mit dem Threadpool erforderlich ist. Die Methode **Main()** erhält den Modifikator **async** und den Rückgabetyp **Task**, sodass der Operator **await** benutzt werden darf:

```
static async Task Main() {
  int svrPort = 55555;
  const int mxClients = 30;
  IPAddress ip = Dns.GetHostEntry("localhost").AddressList[1];

  TcpListener server = null;
  Task[] clientTasks = new Task[mxClients];
  try {
    server = new TcpListener(ip, svrPort);
    server.Start();
    Console.WriteLine("Zeitserver lauscht seit " + DateTime.Now + " (IP: " +
      ip + ", Port: " + svrPort + ")");
    int clientsAccepted = 0;
    while (++clientsAccepted <= mxClients) {
      Console.WriteLine($"\nReady for client {clientsAccepted}" +
                        $" (currently active: {noActiveClients})");
      clientTasks[clientsAccepted-1] = ServeClientAsync(await server.AcceptTcpClientAsync());
      ThreadUsage();
    }
    Task.WaitAll(clientTasks);
    Console.WriteLine($"\nMax. number of worker threads simultaneously used: {maxUsedWorker}");
    Console.WriteLine($"Max. number of IOCP-Threads simultaneously used:   {maxUsedIO}");
  } catch (Exception e) {
    Console.WriteLine(e);
  } finally {
    server.Stop();
  }
}
```

Die Bedienung der Klienten wird in die asynchrone Methode `ServeClientAsync()` verlagert, die einen Parameter vom Typ **TcpClient** erwartet:

```
static async Task ServeClientAsync(TcpClient client) {
  using (client) {
    int clientId;
    lock (lockObject) {
      noActiveClients++;
      clientId = ++noClients;
    }
    Console.WriteLine("\n" + clientId + " " + DateTime.Now + " Anfrage von\n  IP-Nummer:\t" +
             (client.Client.RemoteEndPoint as IPEndPoint).Address + "\n  Port:   \t" +
             (client.Client.RemoteEndPoint as IPEndPoint).Port);
    NetworkStream stream = client.GetStream();
    byte[] msg = Encoding.ASCII.GetBytes(DateTime.Now.ToString());
    await stream.WriteAsync(msg, 0, msg.Length);
    byte[] answer = new byte[1];
    await stream.ReadAsync(answer, 0, 1);
    Console.WriteLine("Client served: " + clientId+ ", Worker Thread Id: " +
                      Thread.CurrentThread.ManagedThreadId);
    lock (lockObject) {
      noActiveClients--;
    }
  }
}
```

Beim Aufruf von `ServeClientAsync()` in der zentralen **while**-Schleife des Zeit-Servers wird der erforderliche **TcpClient**-Aktualparameter von der asynchronen **TcpListener**-Methode **AcceptTcpClientAsync()** geliefert, die aufgrund des deklarierten Rückgabetyps **Task<TcpClient>** per **await**-Operator aufgerufen werden muss.

Die Methode `ServeClientAsync()` verwendet zum Senden bzw. Empfangen die asynchronen **Stream**-Methoden **WriteAsync()** bzw. **ReadAsync()**. Beide werden per **await**-Operator aufgerufen, sodass `ServeClientAsync()` während des Wartens die Kontrolle an **Main()** zurückgibt. Damit können neue Klienten bedient werden, bevor die Kommunikation mit alten Klienten abgeschlossen ist.

Im Rumpf der **while**-Schleife in **Main()** wird die Methode `ThreadUsage()` aufgerufen, um die maximale Anzahl der verwendeten Worker- bzw. IO-Threads aus dem Threadpool zu ermitteln:

```
static void ThreadUsage() {
   int usedWorker, usedIO;
   ThreadPool.GetMaxThreads(out int maxWorker, out int maxIO);
   ThreadPool.GetAvailableThreads(out int availableWorker, out int availableIO);
   usedWorker = maxWorker - availableWorker;
   if (usedWorker > maxUsedWorker)
      maxUsedWorker = usedWorker;
   usedIO = maxIO - availableIO;
   if (usedIO > maxUsedIO)
      maxUsedIO = usedIO;
}
```

Etliche Anweisungen in den Methoden **Main()** und `ServeClientAsync()` sowie die gesamte Methode `ThreadUsage()` dienen lediglich dazu, die Effektivität der Multithreading-Lösung zu untersuchen. Zur Lösung der Kernaufgabe würde ein deutlich schlankerer Quellcode ausreichen.

Um die Kommunikation mit einem Klienten künstlich in die Länge zu ziehen, wird in das Klientenprogramm eine Rückmeldung eingebaut, die mit einer Verzögerung von 5 Sekunden erfolgt:

```
Thread.Sleep(5000);
stream.Write(new byte[1] { 1 }, 0, 1);
Console.WriteLine("Client ready");
```

Beim anschließend protokollierten Einsatz fragen 30 Klienten nach der Zeit:

```
Zeitserver lauscht seit 27.07.2021 03:14:08 (IP: 127.0.0.1, Port: 55555)

Ready for client 1 (currently active: 0)

Client 1, 27.07.2021 03:14:16, Anfrage von
  IP-Nummer:    127.0.0.1
  Port:         56366

Ready for client 2 (currently active: 1)

Client 2, 27.07.2021 03:14:17, Anfrage von
  IP-Nummer:    127.0.0.1
  Port:         56368

Ready for client 3 (currently active: 2)

Client 3, 27.07.2021 03:14:18, Anfrage von
  IP-Nummer:    127.0.0.1
  Port:         56371

 . . .

Ready for client 29 (currently active: 7)
Client served: 22, Worker Thread Id: 5

Client 29, 27.07.2021 03:14:41, Anfrage von
  IP-Nummer:    127.0.0.1
  Port:         56443

Ready for client 30 (currently active: 7)
Client served: 23, Worker Thread Id: 5

Client 30, 27.07.2021 03:14:42, Anfrage von
  IP-Nummer:    127.0.0.1
  Port:         56445
Client served: 24, Worker Thread Id: 6
Client served: 25, Worker Thread Id: 6
Client served: 26, Worker Thread Id: 6
Client served: 27, Worker Thread Id: 6
Client served: 28, Worker Thread Id: 6
Client served: 29, Worker Thread Id: 6
Client served: 30, Worker Thread Id: 6

Max. number of worker threads simultaneously used: 1
Max. number of IOCP-Threads simultaneously used:   1
```

Insgesamt werden aus dem Threadpool ...

- ein Worker-Thread
- und ein I/O Completion Port - Thread (IOCP-Thread)

verwendet, weil ...

- ein Klient während seiner Denkpause keinen Thread blockiert
- ein IOCP-Thread nicht bei der vom Betriebssystem abgewickelten asynchronen Ein- bzw. Ausgabe beteiligt ist, sondern nur die Erfolgsmeldung verarbeiten muss (siehe Abschnitt 17.2).

Traditionell versorgt ein Server jeden Klienten in einem eigenen Thread, wobei selbstverständlich der Threadpool zum Einsatz kommt. Anschließend ist die **Main()** - Methode eines traditionell agierenden Zeit-Servers zur asynchronen Versorgung vieler Klienten zu sehen:

```
static void Main() {
  int svrPort = 55555;
  const int mxClients = 30;
  IPAddress ip = Dns.GetHostEntry("localhost").AddressList[1];

  TcpListener server = null;
  Task[] clientTasks = new Task[mxClients];
  try {
    server = new TcpListener(ip, svrPort);
    server.Start();
    Console.WriteLine("Zeitserver lauscht seit " + DateTime.Now + " (IP: " +
      ip + ", Port: " + svrPort + ")");
    int clientsAccepted = 0;
    while (++clientsAccepted <= mxClients) {
      Console.WriteLine($"\nReady for client {clientsAccepted}" +
                        $" (currently active: {noActiveClients})");
      TcpClient client = server.AcceptTcpClient();
      clientTasks[clientsAccepted - 1] = Task.Factory.StartNew(ServeClient, client);
      ThreadUsage();
    }
    Task.WaitAll(clientTasks);
    Console.WriteLine($"\nMax. number of worker threads simultaneously used: {maxUsedWorker}");
    Console.WriteLine($"Max. number of IOCP-Threads simultaneously used:   {maxUsedIO}");
  } catch (Exception e) {
    Console.WriteLine(e);
  } finally {
    server.Stop();
  }
}
```

Für jeden Klienten wird über die statische **TaskFactory**-Methode **StartNew()** ein Auftrag an den Threadpool übergeben (siehe Abschnitt 17.2.2).

Beim anschließend protokollierten Einsatz fragen wiederum 30 Klienten nach der Zeit. Dabei werden maximal zehn Worker-Threads simultan beschäftigt.[1]

[1] Dass keine IOCP-Threads festgestellt wurden, liegt vermutlich an der Diagnose-Technik.

```
Zeitserver lauscht seit 27.07.2021 03:45:26 (IP: 127.0.0.1, Port: 55555)

Ready for client 1 (currently active: 0)

Ready for client 2 (currently active: 0)

Client 1, 27.07.2021 03:45:33, Anfrage von
  IP-Nummer:    127.0.0.1
  Port:         64788

Ready for client 3 (currently active: 1)

Client 2, 27.07.2021 03:45:33, Anfrage von
  IP-Nummer:    127.0.0.1
  Port:         64790

 . . .

Ready for client 29 (currently active: 8)

Client 28, 27.07.2021 03:45:49, Anfrage von
  IP-Nummer:    127.0.0.1
  Port:         64856
Client served: 20, Worker Thread Id: 6

Ready for client 30 (currently active: 8)

Client 29, 27.07.2021 03:45:50, Anfrage von
  IP-Nummer:    127.0.0.1
  Port:         64857
Client served: 21, Worker Thread Id: 7

Client 30, 27.07.2021 03:45:50, Anfrage von
  IP-Nummer:    127.0.0.1
  Port:         64858
Client served: 22, Worker Thread Id: 8
Client served: 23, Worker Thread Id: 11
Client served: 24, Worker Thread Id: 9
Client served: 25, Worker Thread Id: 12
Client served: 26, Worker Thread Id: 13
Client served: 27, Worker Thread Id: 4
Client served: 28, Worker Thread Id: 10
Client served: 29, Worker Thread Id: 6
Client served: 30, Worker Thread Id: 7

Max. number of worker threads simultaneously used: 10
Max. number of IOCP-Threads simultaneously used:   0
```

Die klassische Multithreading-Technik zur Bedienung vieler Klienten schneidet im Vergleich zur TPL-Lösung umso schlechter ab, je mehr Zeit sich die Klienten für ihre Reaktion nehmen.

Die Visual Studio - Projekte zu den im aktuellen Abschnitt 21.5.3 beschriebenen Programmen sind hier zu finden:

…\BspUeb\Netzwerk\Socket\TcpServerTPL
…\BspUeb\Netzwerk\Socket\TcpClient mit verzögerter Rückmeldung
…\BspUeb\Netzwerk\Socket\TcpServerThreadpool

Anhang

A. Operatorentabelle

In der folgenden Tabelle sind alle im Kurs behandelten Operatoren in absteigender Bindungskraft (von oben nach unten) aufgelistet. Gruppen von Operatoren mit gleicher Bindungskraft (Priorität) sind durch horizontale Linien abgegrenzt.[1]

Operator	**Bedeutung**
x.*y*	Member-Zugriff
Methode(*Parameter*)	Methoden- oder Delegatenaufruf
[]	Index-Zugriff
x++, *x* --	Postinkrement bzw. -dekrement
new	Objekterzeugung
typeof(*Type*)	Typ eines Bezeichners ermitteln
nameof(*Var*)	Name eines Bezeichners ermitteln
checked, unchecked	Ganzzahl-Überlaufdiagnose
default(*Type*)	Standardwert eines Typs Wert ermitteln
?., ?[	Null-bedingter Operator
-*x*	Vorzeichenumkehr
!	Negation
~	Bitweise Negation
++*x*, --*x*	Präinkrement bzw. -dekrement
(*Type*)*x*	Typumwandlung
await	Auf die Beendigung einer Task warten
x..*y*	Bereich
switch	**switch**-Ausdruck
with	**with**-Ausdruck

[1] https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/operators/

Operator	Bedeutung
*, /	Multiplikation, Division
%	Modulo (Divisionsrest)
+, -	Addition, Subtraktion
+	String-Verkettung
<<, >>	Links- bzw. Rechts-Shift
>, <, >=, <=, is, as	Vergleichsoperatoren
==, !=	Gleichheit, Ungleichheit
&	Bitweises UND
&	Logisches UND (mit unbedingter Auswertung)
^	Exklusives bitweises ODER
^	Exklusives logisches ODER
|	Bitweises ODER
|	Logisches ODER (mit unbedingter Auswertung)
&&	Logisches UND (mit bedingter Auswertung)
||	Logisches ODER (mit bedingter Auswertung)
??	Null-Sammeloperator (Null-Koaleszenz)
? :	Konditionaloperator
=	Wertzuweisung
+=, -=, *=, /=, %=	Wertzuweisung mit Aktualisierung
??=	Null-Sammelzuweisungsoperator
=>	Lambda-Operator

Mit Ausnahme der Zuweisungsoperatoren sind alle binären Operatoren *links*-assoziativ. Die Zuweisungsoperatoren und der Konditionaloperator sind *rechts*-assoziativ.

Eine wertvolle Referenz zur Klärung von Zweifelsfällen bzgl. der Bindungskraft (Priorität) von Operatoren ist das Buch von Albahari & Johannsen (2020, S. 66ff).

B. Lösungsvorschläge zu den Übungsaufgaben

Kapitel 1 (Einstieg in die objektorientierte Software-Entwicklung mit C#)

Aufgabe 1

1. **Falsch**
2. **Richtig**
3. **Falsch**
   Nur die Startklasse eines C# - Programms besitzt eine statische Methode namens **Main()**. Mit dieser Methode, die von der Laufzeitumgebung aufgerufen wird, startet die Programmausführung.
4. **Falsch**
   Bei realisierter Datenkapselung ist den Methoden fremder Klassen der *direkte* Zugriff auf die Instanzvariablen eines Objekts verboten. Eine Klasse bietet jedoch oft öffentliche Methoden an, die kontrollierte Änderungen am Zustand eines Objekts ermöglichen.

Aufgabe 2

Das Prinzip der **Datenkapselung** reduziert die Fehlerquote und damit den Aufwand zur Fehlersuche und -bereinigung. Die perfektionierte **Modularisierung** durch die Koppelung von Merkmalen und zugehörigen Handlungskompetenzen in einer Klassendefinition erleichtert die …

- Kooperation von mehreren Programmierern bei großen Projekten,
- die Wiederverwendung von Software.

Kapitel 2 (Die .NET - Plattform)

Aufgabe 1

1. **Falsch**
   In C# werden Programme für die .NET - Plattform entwickelt, die grundsätzlich auf jedem Betriebssystem aufsetzen kann. Die Open Source - Projekte Mono und .NET Core sowie die Firma Xamarin (mittlerweile von Microsoft übernommen) haben dafür gesorgt, dass sich in C# auch Software für Linux, macOS, Android und iOS entwickeln lässt. Mit dem .NET Core - Nachfolger .NET 5.0 wird die Diversifizierung der .NET - Implementationen reduziert.
2. **Richtig**
3. **Richtig**
4. **Richtig**
   Allerdings müssen die Regeln der CLS (Common Language Specification) eingehalten werden, damit die Interoperabilität garantiert ist.

Aufgabe 2

Bisher wurden als Aufgaben der CLR erwähnt:

- Verifikation des IL-Codes beim Laden
  So werden technische Defekte abgefangen.
- Der JIT-Compiler in der CLR übersetzt den IL-Code der Assemblies in Maschinencode.
- Speicherverwaltung (Garbage Collection)
- Überwachung von Code mit beschränkten Rechten (via Internet bezogen)

Aufgabe 3

Namensräume und Assemblies sind zwei voneinander *unabhängige* Organisationsstrukturen:

- Klassen, die zum selben Namensraum gehören, können in *verschiedenen* Assemblies implementiert sein.
- In einem Assembly können Klassen aus verschiedenen Namensräumen implementiert werden, was aber üblicherweise nicht geschieht.

Z. B. befindet sich die Klasse **Uri** aus dem Namensraum **System**, die zur Modellierung von Netzwerk-Ressourcen dient, im DLL-Assembly **System.dll**. Die zum selben Namensraum gehörende Klasse **Console**, die in unseren Übungsprogrammen häufig zur Ein-/Ausgabe per Konsolen verwendet wird, steckt jedoch im DLL-Assembly **mscorlib.dll**, das auch ohne Referenz vom Compiler stets durchsucht wird.

Aufgabe 4

IL	Ein .NET - Compiler übersetzt den Quellcode nicht in Maschinensprache, sondern in die *Intermediate Language* (IL). Diesen Zwischencode übersetzt der JIT-Compiler der CLR in Maschinencode.
BCL	Die Standardklassenbibliothek des .NET - Frameworks wird als *Base Class Library* bezeichnet. Sie enthält ausgereifte Lösungen für praktisch alle Routineaufgaben der Programmierung (z. B. grafische Bedienoberflächen, Dateiverarbeitung, Netzwerkprogrammierung).
CLS	Microsoft hat unter dem Namen *Common Language Specification* einen Sprachumfang definiert, den *jede* .NET - Programmiersprache erfüllen muss. Beschränkt man sich bei der Klassendefinition auf diesen kleinsten gemeinsamen Nenner, ist die Interoperabilität mit anderen CLS-kompatiblen Klassen sichergestellt.
COM	Die traditionelle Windows-Komponententechnologie (*Component Object Model*) soll vom .NET - Framework abgelöst werden. Aktuell (2020) spielt COM-Software in der Windows-Welt aber noch eine große Rolle.

Kapitel 3 (Werkzeuge zum Entwickeln von C# - Programmen)

Aufgabe 2

Beim Hallo-Programm lohnt sich die **using**-Direktive für den Namensraum **System** ausnahmsweise *nicht*, weil das Namensraumpräfix im Quellcode nur *einmal* auftritt:

```
class Hallo {
   static void Main() {
      System.Console.WriteLine("Hallo Allerseits!");
   }
}
```

Aufgabe 3

- Das Schlüsselwort **using** wird klein geschrieben.
- Die Startmethode muss den Namen **Main()** haben.
- Der Methodenname **WriteLine()** ist falsch geschrieben.
- Die Zeichenfolge im Parameter des **WriteLine()** - Aufrufs muss mit dem "-Zeichen abgeschlossen werden.
- Die schließende Klammer zum Rumpf der Klassendefinition fehlt.

Kapitel 4 (Elementare Sprachelemente)

Abschnitt 4.1 (Einstieg)

Aufgabe 1

Der 1. Aufruf scheitert:	Der Methodenname **Main** muss groß geschrieben werden.
Der 2. Aufruf klappt:	Der Modifikator **public** ist allerdings nicht erforderlich.
Der 3. Aufruf klappt:	Statt **void** ist auch der Rückgabetyp **int** erlaubt. Dann muss **Main()** aber einen **int**-Wert an die CLR zurückliefern. Wie das per **return**-Anweisung bewerkstelligt wird, erfahren Sie später.
Der 4. Aufruf scheitert:	Der Rückgabetyp **double** ist verboten.
Der 5. Aufruf klappt:	Diese Variante haben wir bisher meistens benutzt.

Aufgabe 2

Unzulässig sind:

- `4you`
  Namen müssen mit einem Buchstaben oder mit einem Unterstrich beginnen.
- `else`
  Schlüsselwörter wie **else** sind als Namen verboten.

Abschnitt 4.2 (Ausgabe bei Konsolenanwendungen)

Aufgabe 1

Von den beiden im **WriteLine()** - Parameter auftretenden Plus-Operatoren

```
Console.WriteLine("3,3 + 2 = " + 3.3 + 2);
```

Bestandteil der Zeichenkette — Erster Plus-Op. — Zweiter Plus-Op.

wird der linke (unmittelbar auf die Zeichenkette folgende) zuerst ausgeführt und bewirkt eine Verkettung von Zeichenfolgen, wobei die Zahl 3,3 automatisch in eine Zeichenfolge konvertiert wird. Anschließend wirkt der zweite Plus-Operator analog und erweitert die Zeichenfolge um die Ziffer „2“.

Mit runden Klammern kann man dafür sorgen, dass der *zweite* Plus-Operator zuerst ausgeführt wird. Er steht zwischen zwei numerischen Argumenten und addiert diese. Das Ergebnis wird dann vom ersten Plus-Operator in eine Zeichenfolgenverkettung einbezogen:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  Console.WriteLine("3,3 + 2 = " + (3.3 + 2));`
`  }`
`}`
	`3,3 + 2 = 5,3`

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\FormAus

Abschnitt 4.3 (Variablen und Datentypen)

Aufgabe 1

Eine lokale Variable ist verfügbar von der deklarierenden Anweisung bis zur schließenden Klammer des Blocks, in dem sich die Deklaration befindet.

Aufgabe 2

So lässt sich das Programm übersetzen:

```
class Prog {
  static void Main() {
    float pi = 3.141593f;
    double radius = 2.0;
    System.Console.WriteLine("Der Flächeninhalt beträgt: {0:f3}",
        pi * radius * radius);
  }
}
```

Aufgabe 3

Lösungsvorschlag:

```
using System;
class Prog {
  static void Main() {
    Console.WriteLine("Dies ist ein Zeichenketten-Literal:\n\t\"Hallo\"");
  }
}
```

Aufgabe 4

char gehört zu den integralen (ganzzahligen) Datentypen. Jedes Zeichen wird über seine Nummer im Unicode-Zeichensatz gespeichert, das Zeichen 'c' offenbar durch die Zahl 99 (im Dezimalsystem). Der dezimalen Zahl 99 entspricht die hexadezimale Zahl 0x63 (= 6 · 16 + 3). In der folgenden Anweisung wird der **char**-Variablen `zeichen` die Unicode-Escape-Sequenz für das Zeichen 'c' zugewiesen:

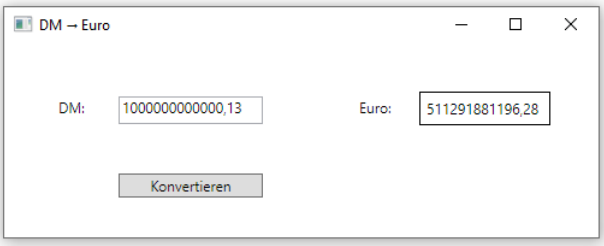
```
char zeichen = '\u0063';
```

Aufgabe 5

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\DmToEuroDec

Für die Eingabe 1000000000000,13 liefert die neue Programmversion das korrekte Ergebnis,



während bei Verwendung des Datentyps **double** nach dem Runden auf 2 Nachkommastellen ein Cent zu viel ausgegeben wird.

Das Auftreten eines Fehlers an der 15. Mantissenstelle steht übrigens *nicht* im Widerspruch zur Norm IEEE-754 (64 Bit), die beim Typ **double** eine *Speicherung* mit mind. 15 signifikanten Dezi-

malstellen in der Mantisse verspricht. Der Speicherfehler wurde durch die Gleitkommadivision vergrößert.

Abschnitt 4.5 (Operatoren und Ausdrücke)

Aufgabe 1

Die Ausdrücke haben folgende Typen und Werte:

Ausdruck	Typ	Wert	Anmerkungen
`6/4*2.0`	**double**	`2.0`	Abarbeitung mit Zwischenergebnissen: **`6/4`**`*2.0` **`1*2.0`**
`(int)6/4.0*3`	**double**	`4.5`	Der Typumwandlungsoperator hat die höchste Priorität und bezieht sich daher (ohne Wirkung) auf die Zahl 6. Abarbeitung mit Zwischenergebnissen: **`(int)6`**`/4.0*3` **`6/4.0`**`*3` **`1.5*3`**

`(int)(6/4.0*3)`	**int**	`4`	Abarbeitung mit Zwischenergebnissen:
			`(int)(`**`6/4.0`**`*3)`
			`(int)(`**`1.5*3`**`)`
			`(int)4.5`

`3*5+8/3%4*5`	**int**	`25`	Abarbeitung mit Zwischenergebnissen:
			`3*5``+8/3%4*5`
			`15 + `**`8/3`**`%4*5`
			`15 + `**`2%4`**`*5`
			`15 + `**`2*5`**
			`15 + 10`

Aufgabe 2

`erg1` erhält den Wert 2, denn:

- `(i++ == j ? 7 : 8)` hat den Wert 8, weil 2 ≠ 3 ist.
- 8 **%** 3 ergibt 2.

`erg2` erhält den Wert 0, denn:

- Der Präinkrementoperator trifft auf die bereits vom Postinkrementoperator in der vorangehenden Zeile auf den Wert 3 erhöhte Variable `i` und setzt sie auf den Wert 4.
- Dies ist auch der Wert des Ausdrucks `++i`, sodass die Bedingung im Konditionaloperator erneut den Wert **false** hat.
- `(++i == j ? 7 : 8)` hat also den Wert 8, und 8 **%** 2 ergibt 0.

Aufgabe 3

Die Vergleichsoperatoren (>, ==) haben eine höhere Bindungskraft als die logischen Operatoren und der Zuweisungsoperator, sodass z. B. in der folgenden Anweisung

```
la1 = 2 > 3 && 2 == 2 ^ 1 == 1;
```

auf runde Klammern verzichtet werden konnte. Besser lesbar ist aber wohl die äquivalente Variante:

```
la1 = (2 > 3) && (2 == 2) ^ (1 == 1);
```

`la1` erhält den Wert **false**, denn der Operator ^ wird aufgrund seiner höheren Bindungskraft vor dem Operator **&&** ausgeführt.

`la2` erhält den Wert **true**, weil die runden Klammern dafür sorgen, dass der Operator ^ zuletzt ausgeführt wird.

`la3` erhält den Wert **false**.

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\Exp

Aufgabe 5

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\DmToEuroSep

Abschnitt 4.7 (Anweisungen)

Aufgabe 1

Weil die **else**-Klausel der *zweiten* (nach oben nächstgelegenen) **if**-Anweisung zugeordnet wird, ergibt sich der folgende „Gewinnplan“:

`losNr`	Gewinn
durch 13 teilbar	0 €
nicht durch 13, aber durch 7 teilbar	1 €
weder durch 13, noch durch 7 teilbar	100 €

Aufgabe 2

Im logischen Ausdruck der **if**-Anweisung findet an Stelle eines Vergleichs eine *Zuweisung* statt. Der Zuweisungsausdruck `(b = false)` besitzt den bei einer **if**-Anweisung erforderlichen Typ **bool**, weil die Variable `b` und der zugewiesene Ausdruck (Literal **false**) diesen Typ besitzen.

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\PrimitivOBC

Die Lösung ohne **break** und **continue** ist komplizierter, also in der Regel nicht sinnvoll.

Aufgabe 4

Das Semikolon am Ende der Zeile

```
while (i < 100) ;
```

wird vom Compiler als die zur **while**-Schleife gehörige (leere) Anweisung interpretiert, sodass mangels `i`-Inkrementierung eine *Endlosschleife* vorliegt. Die restlichen Zeilen werden als selbständiger Anweisungsblock gedeutet, aber nie ausgeführt.

Aufgabe 5

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\FloP

Bei $i = 52$ erhält man letztmals das mathematisch korrekte Vergleichsergebnis. Dies ist gerade die Anzahl der Bits in der **double**-Mantisse.

Die Zahl 1,0 hat nach der Norm IEE 754 die folgende normalisierte **double**-Darstellung (vgl. Abschnitt 4.3.5.1)

$$(-1)^0 \cdot 2^{(1023-1023)} \cdot (1+0{,}0)$$

mit den Bits:

Vorz.	Exponent	Mantisse
0	01111111111	0000000000000000000000000000000000000000000000000000

Das Bitmuster zum Vergleichswert:

$$1{,}0 + 2^{-i} = (-1)^0 \cdot 2^{(1023-1023)} \cdot (1 + 2^{-i}),\ i = 1, \ldots, 52$$

hat eine zusätzliche Eins beim i-ten Mantissen-Bit. Bei $i = 52$ ist diese Eins am rechten Rand angekommen und letztmals vorhanden:

Vorz.	Exponent	Mantisse
0	01111111111	0000000000000000000000000000000000000000000000000001

Während bei binärer Gleitkommatechnik mit **double**-Präzision die Zahl

$$1{,}0 + 2^{-53} \approx 1{,}0 + 1{,}1102230246251565 \cdot 10^{-16} \approx 1{,}00000000000000011102230246251 57$$

nicht mehr von der Zahl 1,0 zu unterscheiden ist, kann die Zahl

$$2^{-53} \approx 1{,}1102230246251565 \cdot 10^{-16}$$

mit dem Bitmuster

Vorz.	Exponent	Mantisse
0	01111001010	0000000000000000000000000000000000000000000000000000

von der 0,0 mit dem Bitmuster

Vorz.	Exponent	Mantisse
0	00000000000	0000000000000000000000000000000000000000000000000000

unterschieden werden.

Erst ab 2^{-1023} ($\approx 1{,}1125369292536007 \cdot 10^{-308}$) wird erstmals die denormalisierte Darstellung benötigt

Vorz.	Exponent	Mantisse
0	00000000000	1000000000000000000000000000000000000000000000000000

und bei 2^{-1074} ($\approx 4{,}9406564584124654 \cdot 10^{-324}$)

Vorz.	Exponent	Mantisse
0	00000000000	0000000000000000000000000000000000000000000000000001

ist letztmals eine Unterscheidung von der Zahl 0,0 möglich.

Aufgabe 6

Lösungsvorschläge mit den beiden Algorithmus-Varianten befinden sich in den Ordnern:

...\BspUeb\Elementare Sprachelemente\GGT.Diff

...\BspUeb\Elementare Sprachelemente\GGT.Mod

Kapitel 5 (Klassen und Objekte)

Aufgabe 1

1. **Falsch**
2. **Stimmt**
   Die Methodenüberladung ist erlaubt und nützlich (siehe Abschnitt 5.3.5).
3. **Falsch**
   Es darf kein Rückgabetyp angegeben werden (siehe Abschnitt 5.4.3.1).
4. **Falsch**
   Mit der Datenkapselung wird verhindert, dass die *Methoden fremder Klassen* auf Instanzvariablen zugreifen. Es geht *nicht* darum, Objekte einer Klasse voreinander zu schützen. Die von einem Objekt ausgeführten Methoden haben stets vollen Zugriff auf die Instanzvariablen anderer Objekte derselben Klasse, sofern eine entsprechende Referenz vorhanden ist. Der Klassendesigner ist für das sinnvolle Verhalten der Methoden verantwortlich.
5. **Stimmt**
6. **Falsch**
   Der Rückgabetyp einer Methode ist irrelevant für ihre Signatur (siehe Abschnitt 5.3.5).

Aufgabe 2

Ein **readonly** - Feld kann nach der Initialisierung nicht mehr verändert werden, auch nicht von klasseneigenen Methoden. Eine *getonly* - Eigenschaft erlaubt nur den lesenden Zugriff. Ein zugrunde liegendes Feld kann jedoch von klasseneigenen Methoden jederzeit geändert werden.

Aufgabe 3

In der Anweisung

```
return Anzahl;
```

wurde `Anzahl` versehentlich groß geschrieben, sodass sich die **get**-Methode der Eigenschaft `Anzahl` selbst aufruft. Diese ungeplante Rekursion führt zu einem Stack-Überlauf (vgl. Abschnitt **5.7.1**).

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Klassen und Objekte\R2Vek

Aufgabe 5

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Klassen und Objekte\Rekursion

Aufgabe 6

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Einleitung\Bruch\GUI

Aufgabe 7

Die korrekte Lösung:

Begriff	Pos.
Definition einer Instanzmethode mit Referenzrückgabe	7
Deklaration lokale Variable	8
Definition einer Instanzmethode mit Wertparameter vom Typ einer Klasse	5
Deklaration von Instanzvariablen	1
Methodenaufruf	6
Deklaration einer statischen Eigenschaft	12

Begriff	Pos.
Konstruktordefinition	3
Deklaration einer Klassenvariablen	2
Objekterzeugung	11
Definition einer Klassenmethode	10
Definition einer Instanzeigenschaft	4
Operatorüberladung	9

Kapitel 6 (Weitere .NETte Typen)

Aufgabe 1

Dank Autoboxing kann man einer **object**-Variablen auch einen **int**-Wert zuweisen, wobei ein neues Objekt auf dem Heap entsteht, das den **int**-Wert als Kopie erhält. Im Ausdruck

```
o1 == o2
```

werden die Inhalte der beiden Referenzvariablen, also die Adressen der beiden referenzierten Objekte, verglichen, die im Beispielprogramm verschieden sind.

Beim Vergleich von zwei Referenzvariablen mit Datentyp **String** orientiert sich der Identitätsoperator allerdings *nicht* an den enthaltenen Adressen, sondern an den Inhalten der referenzierten **String**-Objekte (siehe Abschnitt 6.3.1.2.2).

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Weitere .NETte Typen\Arrays\Lotto

Man könnte das Problem, eine zufällige Teilmenge aus einem Array mit den Elementen 1 bis 49 zu ziehen, in den folgenden Schritten elegant lösen:

- Bringe die Elemente des Arrays in eine zufällige Reihenfolge, wähle also eine zufällige Permutation.
- Wähle aus der Permutation die ersten 6 Elemente.

Eine zufällige Anordnung von Array-Elementen lässt sich mit dem **Fisher-Yates - Algorithmus** herstellen.[1] Wird (wie im Beispiel) nur eine kleine Teilmenge aus der Permutation verwertet, ist allerdings der Aufwand höher als bei der im Lösungsvorschlag praktizierten Suche nach unverbrauchten Lottozahlen.

Aufgabe 3

Hier finden Sie den Lösungsvorschlag für eine Konsolen-Anwendung:

...\BspUeb\Weitere .NETte Typen\Arrays\Eratosthenes

Hier finden Sie den Lösungsvorschlag für eine WPF-Anwendung:

...\BspUeb\WPF\Eratosthenes

Die Behandlungsmethode zum **Click**-Ereignis des Befehlsschalters ist erfreulich einfach:

```
private void btnStart_Click(object sender, RoutedEventArgs e) {
    Cursor oldCursor = Cursor;
    Cursor = Cursors.Wait;
    try {
        listBox.ItemsSource = null;
        var items = new List<int>();

        // Kandidaten-Array erzeugen und vorbereiten
        int grenze = Convert.ToInt32(textBox.Text);
        if (grenze < 2 || grenze > 2146435070)
            throw new ArgumentException("Unzulässiges Argument");
        int maxBasis = (int)(Math.Sqrt(grenze) + 0.5);
        bool[] prim = new bool[grenze + 1];
        for (int i = 1; i <= grenze; i++)
            prim[i] = true;

        // Eratosthenes-Sieb
        for (int basis = 2; basis <= maxBasis; basis++)
            if (prim[basis])
                for (int i = 2; i * basis <= grenze; i++)
                    prim[i * basis] = false;

        // Ergebnis ausgeben
        for (int i = 2; i <= grenze; i++)
            if (prim[i])
                items.Add(i);
        listBox.ItemsSource = items;
    } catch (Exception ex) {
        listBox.ItemsSource = null;
        MessageBox.Show(this, ex.ToString(), ex.Message);
    } finally {
        Cursor = oldCursor;
    }
}
```

Vor dem Start der potentiell zeitaufwändigen Sieb-Operation wird der aktuelle Cursor (Mauszeiger) in einer Variablen vom Typ der Klasse **Cursor** (Namensraum **System.Windows.Input**) gesichert. Dann wird der Cursor mithilfe der Eigenschaft **Wait** der Klasse **Cursors** (Namensraum **System.Windows.Input**) ersetzt durch den Wait- bzw. Sanduhr-Cursor:

```
Cursor oldCursor = Cursor;
Cursor = Cursors.Wait;
```

[1] https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle

In der **finally**-Klausel der **try-catch-finally** - Anweisung zur Absicherung gegen Ausnahmefehler wird der Standard-Cursor reaktiviert. Damit findet die Reaktivierung auf jeden Fall statt, sowohl beim normalen Ablauf wie auch nach einem Ausnahmefehler.

Wenn die gewünschte Obergrenze des Suchbereichs das erlaubte Maximum für die Anzahl der Elemente eines Arrays übertrifft (siehe Abschnitt 6.2.4) oder < 2 ist, dann wirft die Methode eine Ausnahme, damit in der **catch**-Klausel alle Fehler durch denselben Aufruf der Methode **MessageBox.Show()** berichtet werden können:

```
if (grenze < 2 || grenze > 2146435070)
    throw new ArgumentException("Unzulässiges Argument");
```

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Weitere .NETte Typen\Arrays\FloatMatrix

Aufgabe 5

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Weitere .NETte Typen\Zeichenfolgen\PerZuf

Aufgabe 6

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Weitere .NETte Typen\Zeichenfolgen\StringUtil

Kapitel 7 (Vererbung und Polymorphie)

Aufgabe 1

In der Basisklasse fehlt ein parameterfreier Konstruktor. Weil die abgeleitete Klasse keinen expliziten Konstruktor besitzt, kommt dort der Standardkonstruktor zum Einsatz, der den parameterfreien Konstruktor der Basisklasse aufzurufen versucht (vgl. Abschnitt 7.3).

Aufgabe 2

In der Klasse `Figur` haben `xpos` und `ypos` den voreingestellten Zugriffsschutz **private**. Damit haben die Methoden der `Kreis`-Klasse keinen direkten Zugriff (auch die Konstruktoren nicht). Soll dieser Zugriff möglich sein, müssen `xpos` und `ypos` in der `Figur`-Definition die Schutzstufe **protected** (oder **public**) erhalten.

Aufgabe 3

Beim *Überladen* existieren in einer Klasse mehrere Methoden mit demselben Namen, aber verschiedenen Parameterlisten. Eventuell sind einige von den überladenen Methoden in der Klasse selbst definiert und andere geerbt.

Beim *Verdecken* und beim *Überschreiben* findet eine Ersetzung einer Basisklassenmethode durch eine Unterklassenmethode mit dem gleichen Namen und einer identischen Parameterliste statt. Der wesentliche Unterschied zwischen den beiden Ersetzungstechniken zeigt sich dann, wenn ein Unterklassenobjekt über eine Referenzvariable vom *Basisklassentyp* angesprochen wird:

- Bei verdeckenden Methoden kommt dann die *Basisklassenvariante* zum Einsatz (frühe Bindung).
- Bei überschreibenden Methoden wird jedoch die *Unterklassenvariante* benutzt (späte bzw. dynamische Bindung).

Bei klassenbezogenen Methoden kommt das späte Binden bzw. Überschreiben *nicht* in Frage.

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Klassen und Objekte\Bruch\b9 ToString-Überschreibung

Kapitel 8 (Typgenerisches Programmieren)

Aufgabe 1

1. **Richtig**
2. **Falsch** (siehe Abschnitt 8.5)
3. **Falsch**
   Wenn die Aussage korrekt wäre, hätte C# ein gravierendes Fehlerrisiko. Leider liegt genau diese Situation (die sogenannte *Kovarianz* des Elementdatentyps) bei Arrays vor (siehe Abschnitt 8.2.3).
4. **Richtig**

Aufgabe 2

Ein typisches Ergebnis (gemessen auf einem Rechner unter Windows 10 (64 Bit) mit Intel-CPU Core i3 550):

```
Zeit in Millisek. für List<int>: 13,9998
Zeit in Millisek. für ArrayList: 112,9942
```

Das zugehörige Programm finden Sie in:

...\BspUeb\Typgenerisches Programmieren\Listenwerte

Aufgabe 3

Lösungsvorschlag:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`static T Max<T>(params T[] ta) where T : IComparable<T> {`
`if (ta.Length == 0)`
`return default(T);`
`T max = ta[0];`
`foreach (T t in ta)`
`if (t.CompareTo(max) > 0)`
`max = t;`
`return max;`
`}`
 	
`public static void Main() {`
`Console.WriteLine("int-max:\t" + Max(4, 12, 13, 56));`
`Console.WriteLine("double-max:\t" + Max(2.16, 47.11, 34.2, 79.71));`
`Console.WriteLine("String-max:\t" + Max("abc", "def", "zeta"));`
`}`
`}`
	`int-max: 56`
	`double-max: 79,71`
	`String-max: zeta`

Kapitel 9 (Interfaces)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Interfaces\Bruch

Aufgabe 2

Leicht vereinfachend kann man die wesentlichen Unterschiede so beschreiben:

- **Bestandteile**
  Eine abstrakte Klasse enthält mindestens *eine* abstrakte Methode und ansonsten beliebige Klassen-Member. Demgegenüber enthält ein traditionelles Interface ausschließlich abstrakte Methoden, Eigenschaften, Indexer und Ereignisse, wobei das Schlüsselwort **abstract** im Kopf der Methodendefinitionen ebenso überflüssig wie verboten ist. Außerdem sind bei einem Interface Instanz-Konstruktoren und -Felder verboten. Seit C#8 sind in Schnittstellen auch konkrete (implementierte) Methoden erlaubt, die von einem implementierenden Typ verwendet oder durch eine eigene Implementation überschrieben werden können. Zur Verwendung einer konkreten Schnittstellenmethode ohne eigene Implementation ist jedoch eine Referenz vom Schnittstellentyp erforderlich. Der implementierende Typ erbt die konkreten Schnittstellenmethoden also nicht.
- **Abstammungsverhältnisse (Typkompatibilitäten)**
  Eine Klasse kann nur *eine* abstrakte Basisklasse besitzen, aber beliebig viele Interfaces implementieren.

Aufgabe 3

Weil Interfaces als Datentypen taugen und eine Klasse mehrere Interfaces implementieren darf, sind ihre Objekte zu mehreren Datentypen kompatibel. Eine Klasse erbt allerdings nichts von den Schnittstellen, sondern sie gibt Verpflichtungserklärungen ab und muss die entsprechenden Implementierungs-Leistungen erbringen.

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Interfaces\Iterator

Kapitel 10 (Delegaten und Ereignisse)

Aufgabe 1

1. **Stimmt**
2. **Falsch**
3. **Stimmt**
4. **Stimmt**

Aufgabe 2

Lösungsvorschlag:

```
using System;
using System.Collections.Generic;

class FindAll {
   static void Main() {
      var list = new List<String>();
      list.Add("Doro"); list.Add("Liselotte"); list.Add("Friedrich");
      list.Add("Theo"); list.Add("Walter");
      Console.WriteLine("Liste der Kurznamen mit max. 4 Zeichen:");
      var k4 = list.FindAll(s => s.Length <= 4);
      foreach (string s in k4)
         Console.WriteLine(s);
   }
}
```

Aufgabe 3

Im folgenden Lösungsvorschlag ist die Metamethode statisch realisiert:

```
using System;
using System.Collections.Generic;

class FindAll {
   static Predicate<String> ListLE(int k) {
      return (s => s.Length <= k);
   }

   static void Main() {
      var list = new List<String>();
      list.Add("Doro"); list.Add("Liselotte"); list.Add("Friedrich");
      list.Add("Theo"); list.Add("Walter");
      for (int i = 4; i < 10; i++) {
         Console.WriteLine($"\nListe der Kurznamen mit max. {i} Zeichen:");
         var k = list.FindAll(ListLE(i));
         foreach (string s in k)
            Console.WriteLine(s);
      }
   }
}
```

Aufgabe 4

1. **Falsch**

   Ein Ereignis ist ein (statisches) Feld mit einem Delegatentyp, weist aber Besonderheiten im Vergleich zu einer gewöhnlichen Delegatenvariablen auf.

2. **Falsch**

   Wie vorzugehen ist, wird im Abschnitt 10.2.3 erläutert.

3. **Falsch**

   Am Ende von Abschnitt 10.2.2 wird eine strikt zu vermeidende Speicherplatzverschwendung durch die unterlassene Aufhebung einer Registrierung beschrieben.

4. **Richtig**

Kapitel 11 (Kollektionen)

Aufgabe 1

1. **Falsch**
   Es wird die Einfügereihenfolge konserviert, aber keine Sortierung vorgenommen, z. B.:

Quellcode	Ausgabe

`using System;`
`using System.Collections.Generic;`
`class Prog {`
`  static void Main() {`
`  var liste = new List<String> {"Otto", "Thea", "Maria"};`
`  foreach (string s in liste) Console.WriteLine(s);`
`  }`
`}`
	`Otto`
	`Thea`
	`Maria`

2. **Richtig**
3. **Richtig**
4. **Richtig**
5. **Richtig**

Aufgabe 2

Sie finden einen Lösungsvorschlag in:

...\BspUeb\Kollektionen\PersonenListe

Über die Aufgabenstellung hinausgehend enthält der Lösungsvorschlag eine Erweiterung der Klasse `Person` um die Methode **CompareTo(**`Person p`**)** und die somit gerechtfertigte Zusicherung, das Interface **IComparable**<`Person`> zu erfüllen (siehe Kopf der Klassendefinition):

```
using System;
class Person : IComparable<Person> {
   public string Vorname;
   public string Name;
   public Person(string vorname, string nachname) {
      Vorname = vorname;
      Name = nachname;
   }
   public int CompareTo(Person p) {
      int vergl = (this.Name+this.Vorname).CompareTo(p.Name+p.Vorname);
      if (vergl < 0)
         return -1;
      else
         if (vergl == 0)
            return 0;
         else
            return 1;
   }
}
```

Infolgedessen können die Elemente der **List**<`Person`> - Kollektion sogar sortiert werden.

Kapitel 12 (Einstieg in die GUI-Programmierung mit WPF)

Aufgabe 1

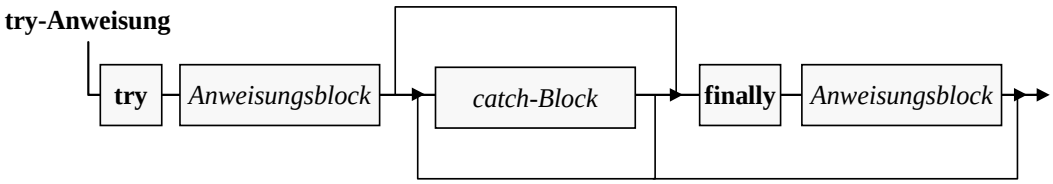
Sie finden einen Lösungsvorschlag im Verzeichnis:

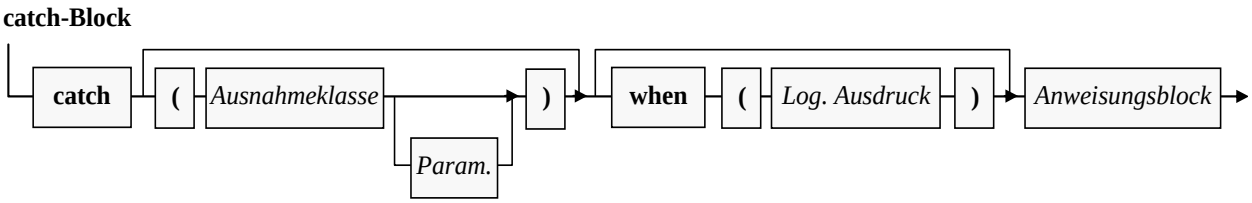
...\BspUeb\WPF\Eurokonverter

Kapitel 13 (Ausnahmebehandlung)

Aufgabe 1

Lösungsvorschlag:





Aufgabe 2

Die Klasse **OverflowException** stammt von der Klasse **ArithmeticException** ab. Weil die **Main()** - Methode der Klasse `Sequenzen` einen **ArithmeticException**-Handler besitzt, wird dort auch die **OverflowException** „behandelt“.

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Ausnahmebehandlung\DuaLog\ArgumentOutOfRangeException

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Ausnahmebehandlung\EinfachStapelEx

Kapitel 14 (Attribute)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Attribute\NonsenseAttribute

Kapitel 15 (Sonstige C# - Sprachbestandteile)

Aufgabe 1

Sie haben sich hoffentlich nicht wegen die Platzierung der Aufgabe im Kapitel 15 verpflichtet gefühlt, eine Lösung mit Hilfe von Mustervergleichen zu erstellen. Mit einem logischen Ausdruck ist die Aufgabe einfach und schnell zu lösen:

```
public bool Zugelassen() => Passagiere >= 3 || Elektromobil;
```

Wie zu Beginn von Abschnitt 15.1 erwähnt, haben es die neuen Optionen zum Mustervergleich oft schwer, ihre Überlegenheit gegenüber herkömmlichen Lösungen zu beweisen.

Aufgabe 2

Wie bei jedem Ausdruck mit binärem Operator wird im folgenden Beispiel

```
ass?[1]?.Length == ++res
```

zunächst der linke Operand des Identitätsoperators ausgewertet. Wenn die Variable `ass` ins Leere zeigt, oder das Element 1 des **String**-Arrays gleich **null** ist, dann kann die linke Seite dank der Null-bedingten Operatoren trotzdem fehlerfrei ausgewertet werden (resultierender Wert: **null**). Folglich wird anschließend auch die rechte Seite des Identitätsoperators ausgewertet. Obwohl `res` nicht den Typ **Nullable<int>** hat, kann die **int**-Variable doch mit **null** verglichen werden (siehe Abschnitt 8.3).

Kapitel 16 (Dateiverarbeitung)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Ein- und Ausgabe über Datenströme\Mittelwerte

Aufgabe 2

Das Einschalten der **AutoFlush**-Funktion ist bei einer Dateiausgabe in der Regel überflüssig und sehr zeitintensiv. Also sollte die folgende Zeile entfernt werden:

```
sw.AutoFlush = true;
```

Kapitel 17 (Multithreading)

Aufgabe 1

- Falsch
- Richtig
- Richtig
- Richtig

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Multithreading\Thread\Signalisierungsobjekte\CountdownEvent

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Multithreading\async await\DirSize

Literatur

Agafonov, E. & Koryavchenko, A. (2015). *Mastering C# Concurrency*. Birmingham: Packt Publishing.

Albahari, J. (2014). *Threading in C#*. Online-Dokument: http://www.albahari.com/threading/.

Albahari, J. & Johannsen, E. (2020). *C# 8.0 in a Nutshell*. Beijing: O'Reilly.

Baltes, S. & Diehl, S. (2014). *Sketches and Diagrams in Practice*. Paper presented at the International Symposium on the Foundations of Software Engineering (November 2014, Hong Kong).

Baltes-Götz, B. (2003). *Einführung in das Programmieren mit Visual C++ 6.0*. Online-Dokument: http://www.uni-trier.de/index.php?id=22785

Baltes-Götz, B. (2010). *Einführung in das Programmieren mit C# 3.0*. Online-Dokument: https://www.uni-trier.de/index.php?id=30454

Baltes-Götz, B. & Götz, J. (2020). *Einführung in das Programmieren mit Java 13*. Online-Dokument: http://www.uni-trier.de/index.php?id=22787

Balzert, H. (2011). *Lehrbuch der Objektmodellierung: Analyse und Entwurf mit der UML 2*. Heidelberg: Spektrum.

Bloch, J. (2018). *Effective Java* (3rd. ed.). Upper Saddle River, NJ: Addison-Wesley.

Booch, G. et al. (2007). *Object-Oriented Analysis and Design with Applications* (3rd ed.). Boston, MA: Addison-Wesley.

Brind, M. (2021). *Learn Entity Framework Core*. Online-Dokument: https://www.learnentityframeworkcore.com/

Cleary, S. (2014). *Concurrency in C#. Cookbook*. Sebastopol, CA: O'Reilly.

Conrod, P. & Tylee, L. (2019). *Visual C# and Databases - 2019 Edition*. Kidware Software

Ebner, M. (2000). *Delphi 5 Datenbankprogrammierung*. München: Addison-Wesley.

ECMA (2009). *Open XML Paper Specification*. Online-Dokument: http://www.ecma-international.org/publications/standards/Ecma-388.htm

ECMA (2017). *C# Language Specification* (5th ed.). Online-Dokument: https://www.ecma-international.org/publications/standards/Ecma-334.htm

Goll, J. & Heinisch, C. (2016). *Java als erste Programmiersprache* (8. Aufl.). Wiesbaden: Springer Vieweg

Griffiths, I. (2013). *Programming C# 5.0*. Beijing: O'Reilly.

Griffiths, I. (2020). *Programming C# 8.0*. Sebastopol, CA: O'Reilly.

Gunnerson, E. (2002). *C#* (2. Aufl.). Bonn: Galileo.

Hamilton, B. & MacDonald, M (2003). *ADO.NET in a Nutshell*. Beijing: O'Reilly.

Horstmann, C.S. & Cornell, G. (2002). *Core Java. Volume II – Advanved Features*. Palo Alto, CA: Sun Microsystems Press.

Kreft, K & Langer, A. (2014). *Java 8. Default-Methoden und statische Methoden in Interfaces*. Online-Dokument: http://www.angelikalanger.com/Articles/EffectiveJava/72.Java8.DefaultMethods/72.Java8.DefaultMethods.html

Lau, O. (2009). Faites vos jeux! Zufallszahlen erzeugen, erkennen und anwenden. *c't Magazin für Computertechnik*. 2009, Heft 2, 172-178.

Lahres, B. & Rayman, G. (2009). *Praxisbuch Objektorientierung. Professionelle Entwurfsverfahren* (2. Aufl.). Bonn: Galileo

Lerman, J. (2010). *Programming Entity Framework* (2nd ed). Sebastopol, CA: O'Reilly Media.

Liskov, B. H. & Wing, J. M. (1999). *Behavioral Subtyping Using Invariants and Constraints*. Online-Dokument: http://reports-archive.adm.cs.cmu.edu/anon/1999/CMU-CS-99-156.pdf

Louis, D. & Strasser, S. (2002). *C# in 21 Tagen*. München: Markt + Technik.

MacDonald, M. (2012). *Pro WPF 4.5 in C#* (4rd ed.). New York, NY: Apress.

Misner, S. (2007). *Microsoft SQL Server 2005 Express Edition. Start Now!* Redmond, WA: Microsoft Press.

Microsoft (2017). *C# Reference*. Online-Dokument https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/index

Morrison, V. (2005a). *Concurrency: What Every Dev Must Know About Multithreaded Apps*. MSDN Magazine. August 2005. Online-Dokument: http://msdn.microsoft.com/de-de/magazine/cc337899.aspx.

Morrison, V. (2005b). *Memory Models: Understand the Impact of Low-Lock Techniques in Multithreaded Apps*. MSDN Magazine. August 2005. Online-Dokument: http://msdn.microsoft.com/de-de/magazine/cc337899.aspx.

Mössenböck, H. (2019). *Kompaktkurs C# 7.0.* Heidelberg: dpunkt.

Nathan, A. (2010). *WPF 4. Unleashed*. Indianapolis, IN: SAMS.

Petre, M. (2013). UML in practice. In: *35th International Conference on Software Engineering (ICSE 2013)*, 18-26 May 2013, San Francisco, CA, USA (forthcoming), pp. 722–731.

Peterson, J. (2014). *Common Pitfalls with IDisposable and the Using Statement*. Online-Dokument: https://superdevelopment.com/2014/03/13/common-pitfalls-with-idisposable-and-the-using-statement/

Petzold, C. (2008). *Foundations: Vector Graphics and the WPF Shape Class*. MSDN-Magazine. März 2008. Online-Dokument: http://msdn.microsoft.com/de-de/magazine/cc337899.aspx.

Richter, J. (2006). *Microsoft .NET Framework Programmierung in C#* (2. Aufl.). Unterschleißheim: Microsoft Press.

Richter, J. (2012). *CLR via C#* (4. Aufl.). Redmond, WA: Microsoft Press.

Sceppa, D. (2003). *Microsoft ADO.NET - Das Entwicklerhandbuch*. Unterschleißheim: Microsoft Press.

Schacherl, R. (2014). *Windows-8-Apps für C#-Entwickler*. Frankfurt a.M.: entwickler.press.

Schulz, H. (2021). Aufgedreht. Windows 11: Neuer Store für Anwender und Entwickler. *c't Magazin für Computertechnik*. 2021, Heft 16, 24-26.

Spurgeon, C. E. (2000). *Ethernet. The Definitive Guide*. Sebastopol, CA: O'Reilly.

Strey, A. (2005). Computer-Arithmetik. Online-Dokument: http://www.informatik.uni-ulm.de/ni/Lehre/SS05/CompArith/

Wegener, J. (2013). *WPF 4.5 und XAML*. München: Hanser-Verlag.

Stichwortregister

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z