Bernhard Baltes-Götz

Einführung in das Programmieren mit C# 4.0

2011 (Rev. 121022)

Herausgeber: Zentrum für Informations-, Medien- und Kommunikationstechnologie (ZIMK) an der Universität Trier
Universitätsring 15
D-54286 Trier

WWW: http://www.uni-trier.de/index.php?id=518

E-Mail: zimk@uni-trier.de

Tel.: (0651) 201-3417, Fax.: (0651) 3921

Autor: Bernhard Baltes-Götz (E-Mail: baltes@uni-trier.de)
Copyright © 2011; ZIMK

Vorwort

Dieses Manuskript entstand als Begleitlektüre zum C# - Einführungskurs, den das Zentrum für Informations-, Medien- und Kommunikationstechnologie (ZIMK) an der Universität Trier über zwei Semester (WiSe 2010/2011 und SoSe 2011) angeboten hat, sollte aber auch für das Selbststudium geeignet sein.

Lerninhalte und -ziele

C# ist eine von der Firma Microsoft für das .NET – Framework entwickelte und von der ***E**uropean **C**omputer **M**anufacturers **A**ssociation* (ECMA[1]) standardisierte Programmiersprache, die auf den Vorbildern C++ (siehe z.B. Baltes-Götz 2003) und Java (siehe z.B. Baltes-Götz 2010a) aufbaut, aber auch etliche Weiterentwicklungen bietet. Die .NET - Plattform etabliert sich allmählich als Standard bei der Softwareentwicklung unter Windows, und das Open Source - Projekt *Mono* hat die Plattform erfolgreich auf andere Betriebssysteme (Android, Linux, MacOS-X, UNIX) portiert. Ein Vorteil des .NET – Frameworks ist die freie Wahl zwischen verschiedenen Programmiersprachen, doch kann C# trotz der großen Konkurrenz als die bevorzugte .NET - Programmiersprache gelten. Schließlich wurde das Framework selbst überwiegend in C# entwickelt.

Wenngleich die Netzorientierung im Namen des neuen Software-Frameworks betont wird, eignet sich C# (wie jede andere .NET – Programmiersprache) für beliebige Anwendungen auf einem Arbeitsplatzrechner, Server oder auch Smartphone. Das Manuskript führt in die .NET –Plattform ein, behandelt neben elementaren Sprachelementen (z.B. Variablen, Datentypen, Operatoren, Anweisungen) die wesentlichen Konzepte und Methoden der objektorientierten Softwareentwicklung (z.B. Klassen, Vererbung, Polymorphie, Schnittstellen) und berücksichtigt viele Standardthemen der Programmierpraxis (z.B. grafische Benutzeroberflächen, Ausnahmebehandlung, Dateizugriff, Multithreading, Netzwerkverbindungen, Vektor- und Bitmap-Grafik, Datenbankprogrammierung).

Bei der Gestaltung von Benutzeroberflächen, Druck- und Grafikausgabe etc. wird die von Microsoft in den letzten Jahren mit großem Aufwand entwickelte und nachdrücklich empfohlene *Windows Presentation Foundation* verwendet.

Voraussetzungen bei den Leser(innen)

- **EDV-Allgemeinbildung**
  Dass die Teilnehmer wenigstens durchschnittliche Erfahrungen bei der *Anwendung* von Computerprogrammen haben sollten, versteht sich von selbst.
- **Programmierkenntnisse werden *nicht* vorausgesetzt.**
  Teilnehmer *mit* Programmiererfahrung werden sich bei den ersten Abschnitten eventuell etwas langweilen. Für diesen Personenkreis enthält das Manuskript einige Vertiefungen (mit entsprechender Kennzeichnung in der Abschnittsüberschrift), die von Einsteigern gefahrlos übersprungen werden können.
- **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 Spaß erleben.

[1] Auf der EMCA-Homepage http://www.ecma-international.org/ taucht die Bedeutung ***E**uropean **C**omputer **M**anufacturers **A**ssociation* neuerdings *nicht* mehr auf.

Software zum Üben

Für die unverzichtbaren Übungen sollte ein Rechner mit einer aktuellen C# - Entwicklungsumgebung zur Verfügung stehen, z.B. mit der *Microsoft Visual Studio 2010 Express Edition*. Das ebenfalls erforderliche *.NET – Framework* wird von der Visual Studio Express Edition bei Bedarf automatisch in der aktuellen Version installiert, ist aber mittlerweile auf praktisch jedem Rechner unter Windows XP, Vista oder 7 als Bestandteil des Betriebssystems oder als Basis zahlloser Anwendungen bereits vorhanden. Die erwähnte Software ist kostenlos verfügbar, und mit der *Visual Studio 2010 Express Edition* entwickelte Programme dürfen sogar frei vertrieben werden.

Dateien zum Manuskript

Die aktuelle Version dieses Manuskripts ist zusammen mit den behandelten Beispielen und Lösungsvorschläge zu vielen Übungsaufgaben auf dem Webserver der Universität Trier von der Startseite (http://www.uni-trier.de/) ausgehend folgendermaßen zu finden:

Rechenzentrum > Studierende > EDV-Dokumentationen >
Programmierung > Einführung in das Programmieren mit C#

Leider blieb zu wenig Zeit für eine sorgfältige Kontrolle des Textes, so dass einige Fehler und Mängel verblieben sein dürften. Entsprechende Hinweise an die E-Mail-Adresse

baltes@uni-trier.de

werden dankbar entgegen genommen.

Trier, im Oktober 2011 Bernhard Baltes-Götz

Inhaltsverzeichnis

1 EINLEITUNG 1

1.1 Beispiel für die objektorientierte Softwareentwicklung mit C# 1
1.1.1 Objektorientierte Analyse und Modellierung 1
1.1.2 Objektorientierte Programmierung 6
1.1.3 Algorithmen 8
1.1.4 Startklasse und Main()-Methode 9
1.1.5 Ausblick auf Anwendungen mit graphischer Benutzerschnittstelle 11
1.1.6 Zusammenfassung zu Abschnitt 1.1 11

1.2 Das .NET – Framework 12
1.2.1 Überblick 12
1.2.2 Installation 14
1.2.3 C#-Compiler und MSIL 15
1.2.4 Common Language Specification 17
1.2.5 Assemblies und Metadaten 17
1.2.5.1 Typ-Metadaten 18
1.2.5.2 Das Manifest eines Assemblies 19
1.2.5.3 Multidatei-Assemblies 19
1.2.5.4 Private und allgemeine Assemblies 19
1.2.5.5 Plattformspezifische Assemblies 20
1.2.5.6 Vergleich mit der COM-Technologie 22
1.2.6 CLR und JIT-Compiler 23
1.2.7 Namensräume und FCL 24
1.2.8 Zusammenfassung zu Abschnitt 1.2 27

1.3 Übungsaufgaben zu Kapitel 1 28

2 WERKZEUGE ZUM ENTWICKELN VON C# - PROGRAMMEN 29

2.1 C# - Entwicklung mit Texteditor und Kommandozeilen-Compiler 29
2.1.1 Editieren 29
2.1.2 Übersetzen in MSIL 31
2.1.3 Ausführen 34
2.1.4 Programmfehler beheben 34

2.2 Microsoft Visual Studio 2010 36
2.2.1 Microsoft Visual Studio 2010 Ultimate 36
2.2.1.1 Terminal-Server – Umgebung 36
2.2.1.2 Konfiguration beim ersten Start 39
2.2.1.3 Eine erste GUI-Anwendung 41
2.2.2 Microsoft Visual C# 2010 Express Edition 51
2.2.2.1 Installation 51
2.2.2.2 Registrierung 54
2.2.2.3 Ein erstes Konsolen-Projekt 55
2.2.2.4 FCL-Dokumentation und andere Hilfeinhalte 59
2.2.3 Debug- vs. Release-Konfiguration 61
2.2.3.1 Einfache Erstellungskonfiguration 62
2.2.3.2 Erweiterte Erstellungskonfiguration 62
2.2.4 Compiler-Optionen in der Entwicklungsumgebung setzen 64
2.2.4.1 Referenzen 65
2.2.4.2 Ausgabetyp 67
2.2.4.3 Zielplattform 68

2.3 Übungsaufgaben zu Kapitel 2 70

3 ELEMENTARE SPRACHELEMENTE 71

3.1 Einstieg 71
3.1.1 Aufbau von einfachen C# - Programmen 71
3.1.2 Syntaxdiagramme 72
3.1.2.1 Klassendefinition 72
3.1.2.2 Methodendefinition 74
3.1.2.3 Eigenschaftsdefinition 75
3.1.3 Hinweise zur Gestaltung des Quellcodes 75
3.1.4 Kommentare 76
3.1.5 Namen 77
3.1.6 Übungsaufgaben zu Abschnitt 3.1 79

3.2 Ausgabe bei Konsolenanwendungen 79
3.2.1 Ausgabe einer (zusammengesetzten) Zeichenfolge 80
3.2.2 Formatierte Ausgabe 80
3.2.3 Übungsaufgaben zu Abschnitt 3.2 82

3.3 Variablen und Datentypen 82
3.3.1 Strenge Compiler-Überwachung bei C# - Variablen 83
3.3.2 Wert- und Referenztypen 84
3.3.3 Klassifikation der Variablen nach Zuordnung 86
3.3.4 Elementare Datentypen 87
3.3.5 Vertiefung: Darstellung von Gleitkommazahlen im Arbeitsspeicher des Computers 88
3.3.5.1 Binäre Gleitkommadarstellung 89
3.3.5.2 Dezimale Gleitkommadarstellung 91
3.3.6 Variablendeklaration, Initialisierung und Wertzuweisung 93
3.3.7 Blöcke und Deklarationsbereiche für lokale Variablen 94
3.3.8 Konstanten 96
3.3.9 Literale 97
3.3.9.1 Ganzzahlliterale 97
3.3.9.2 Gleitkommaliterale 99
3.3.9.3 bool-Literale 100
3.3.9.4 char-Literale 100
3.3.9.5 Zeichenkettenliterale 101
3.3.10 Übungsaufgaben zu Abschnitt 3.3 102

3.4 Einfache Techniken für Benutzereingaben 103
3.4.1 Via Konsole 103
3.4.2 Via InputBox 104

3.5 Operatoren und Ausdrücke 106
3.5.1 Arithmetische Operatoren 107
3.5.2 Methodenaufrufe 109
3.5.3 Vergleichsoperatoren 111
3.5.4 Vertiefung: Gleitkommawerte vergleichen 112
3.5.5 Logische Operatoren 114
3.5.6 Vertiefung: Bitorientierte Operatoren 115
3.5.7 Typumwandlung (Casting) bei elementaren Datentypen 117
3.5.8 Zuweisungsoperatoren 119
3.5.9 Konditionaloperator 121
3.5.10 Auswertungsreihenfolge 121
3.5.11 Übungsaufgaben zu Abschnitt 3.5 123

3.6 Über- und Unterlauf bei numerischen Variablen 124
3.6.1 Überlauf bei Ganzzahltypen 124
3.6.2 Unendliche und undefinierte Werte bei den Typen float und double 127
3.6.3 Überlauf beim Typ decimal 129
3.6.4 Unterlauf bei den Gleitkommatypen 130

3.7 Anweisungen (zur Ablaufsteuerung) 130
3.7.1 Überblick 131
3.7.2 Bedingte Anweisung und Verzweigung 132
3.7.2.1 if-Anweisung 132
3.7.2.2 if-else - Anweisung 133
3.7.2.3 switch-Anweisung 137
3.7.3 Wiederholungsanweisungen 141
3.7.3.1 Zählergesteuerte Schleife (for) 142
3.7.3.2 Iterieren über die Elemente einer Kollektion (foreach) 144
3.7.3.3 Bedingungsabhängige Schleifen 145
3.7.3.4 Endlosschleifen 147
3.7.3.5 Schleifen(durchgänge) vorzeitig beenden 147
3.7.4 Übungsaufgaben zu Abschnitt 3.7 148

4 KLASSEN UND OBJEKTE 151

4.1 Überblick, historische Wurzeln, Beispiel 151
4.1.1 Einige Kernideen und Vorzüge der OOP 151
4.1.1.1 Datenkapselung und Modularisierung 151
4.1.1.2 Vererbung 153
4.1.1.3 Realitätsnahe Modellierung 155
4.1.2 Strukturierte Programmierung und OOP 155
4.1.3 Auf-Bruch zu echter Klasse 156

4.2 Instanzvariablen 159
4.2.1 Sichtbarkeitsbereich, Existenz und Ablage im Hauptspeicher 160
4.2.2 Deklaration mit Wahl der Schutzstufe 161
4.2.3 Initialisierung 162
4.2.4 Zugriff in klasseneigenen und fremden Methoden 163
4.2.5 Konstante und schreibgeschützte Felder 163

4.3 Instanzmethoden 164
4.3.1 Methodendefinition 165
4.3.1.1 Modifikatoren 166
4.3.1.2 Rückgabewerte und return-Anweisung 167
4.3.1.3 Formalparameter 168
4.3.1.4 Methodenrumpf 172
4.3.2 Methodenaufruf und Aktualparameter 172
4.3.3 Debug-Einsichten zu (verschachtelten) Methodenaufrufen 173
4.3.4 Methoden überladen 177

4.4 Objekte 178
4.4.1 Referenzvariablen deklarieren 178
4.4.2 Objekte erzeugen 179
4.4.3 Objekte initialisieren über Konstruktoren 181
4.4.4 Abräumen überflüssiger Objekte durch den Garbage Collector 184
4.4.5 Objektreferenzen verwenden 185
4.4.5.1 Objektreferenzen als Wertparameter 185
4.4.5.2 Rückgabewerte mit Referenztyp 187
4.4.5.3 this als Referenz auf das aktuelle Objekt 187

4.5 Eigenschaften 188
4.5.1 Syntaktisch elegante Zugriffsmethoden 188
4.5.2 Automatisch implementierte Eigenschaften 189
4.5.3 Zeitaufwand bei Eigenschafts- und Feldzugriffen 190

4.6 Statische Member und Klassen 191
4.6.1 Statische Felder und Eigenschaften 191
4.6.2 Wiederholung zur Kategorisierung von Variablen 192
4.6.3 Statische Methoden 193
4.6.4 Statische Konstruktoren 194
4.6.5 Statische Klassen 195

4.7 Vertiefungen zum Thema Methoden 195
4.7.1 Rekursive Methoden 195
4.7.2 Operatoren überladen 197

4.8 Aggregieren und innere Klassen 198
4.8.1 Aggregation 198
4.8.2 Innere (geschachtelte) Klassen 201

4.9 Verfügbarkeit von Klassen und Klassenbestandteilen 203

4.10 Bruchrechnungsprogramm mit WPF-Bedienoberfläche 204
4.10.1 Projekt anlegen mit Vorlage *WPF - Anwendung* 204
4.10.2 Deklaration von WPF-Klassen per XAML 205
4.10.3 Steuerelemente aus der Toolbox übernehmen 207
4.10.4 Positionen und Größen der Steuerelemente gestalten 208
4.10.5 Eigenschaften der Steuerelemente ändern 211
4.10.6 Automatisch erstellter und gepflegter Quellcode 214
4.10.7 Assembly mit der Bruch-Klasse einbinden 215
4.10.8 Ereignisbehandlungsmethoden anlegen 216

4.11 Übungsaufgaben zu Kapitel 4 217

5 WEITERE .NETTE TYPEN 223

5.1 Strukturen 223
5.1.1 Vergleich von Klassen und Strukturen 224
5.1.2 Zur Eignung von Strukturen im Bruchrechnungs-Projekt 227
5.1.3 Strukturen im Common Type System der .NET – Plattform 229

5.2 Boxing und Unboxing 230

5.3 Arrays 232
5.3.1 Array-Referenzvariablen deklarieren 233
5.3.2 Array-Objekte erzeugen 233
5.3.3 Arrays benutzen 234
5.3.4 Beispiel: Beurteilung des .NET - Pseudozufallszahlengenerators 235
5.3.5 Initialisierungslisten 237
5.3.6 Objekte als Array-Elemente 238
5.3.7 Mehrdimensionale Arrays 238
5.3.7.1 Rechteckige Arrays 238
5.3.7.2 Mehrdimensionale Arrays mit unterschiedlich großen Elementen 239
5.3.8 Die Kollektionsklasse ArrayList 240

5.4 Klassen für Zeichenketten 242
5.4.1 Die Klasse String für konstante Zeichenketten 242
5.4.1.1 String als WORM - Klasse 242
5.4.1.2 Methoden für String-Objekte 243
5.4.1.3 Interner String-Pool 245
5.4.2 Die Klasse StringBuilder für veränderliche Zeichenketten 247

5.5 Enumerationen 249

5.6 Indexzugriff bei eigenen Typen zur Verwaltung von Elementen 251

5.7 Übungsaufgaben zu Kapitel 5 254

6 VERERBUNG UND POLYMORPHIE 257

6.1 Das Common Type System (CTS) des .NET – Frameworks 258

6.2 Definition einer abgeleiteten Klasse 259

6.3 base-Konstruktoren und Initialisierungs-Sequenzen 260

6.4 Der Zugriffsmodifikator protected 262

6.5 Erbstücke durch spezialisierte Varianten verdecken 263
6.5.1 Geerbte Methoden, Eigenschaften und Indexern verdecken 263
6.5.2 Geerbte Felder verdecken 265

6.6 Verwaltung von Objekten über Basisklassenreferenzen 266

6.7 Polymorphie (Methoden überschreiben) 268

6.8 Abstrakte Methoden und Klassen 271

6.9 Das Liskovsche Substitutionsprinzip (LSP) 273

6.10 Versiegelte Methoden und Klassen 273

6.11 Übungsaufgaben zu Kapitel 6 274

7 TYPGENERIZITÄT UND KOLLEKTIONEN 277

7.1 Motive für die Einführung generischer Klassen in .NET 2.0 277

7.2 Generische Klassen 279
7.2.1 Definition 279
7.2.2 Restringierte Typformalparameter 280
7.2.3 Generische Klassen und Vererbung 282

7.3 Nullable<T> als Beispiel für generische Strukturen 283

7.4 Generische Methoden 284

7.5 Weitere Klassen aus dem Namensraum System.Collections.Generic 285
7.5.1 Verwaltung einer Menge mit der Klasse HashSet<T> 285
7.5.2 Verwaltung einer Menge von Schlüssel-Wert - Paaren mit der Klasse Dictionary<K, V> 286

7.6 Übungsaufgaben zu Kapitel 7 287

8 INTERFACES 289

8.1 Interfaces definieren 291

8.2 Interfaces implementieren 292

8.3 Interfaces als Referenzdatentypen 296

8.4 Explizite Schnittstellenimplementierung 296

8.5 Übungsaufgaben zu Kapitel 8 298

9 EINSTIEG IN DIE GUI-PROGRAMMIERUNG MIT WPF-TECHNIK 299

9.1 Einstimmung und Orientierung 299
9.1.1 Vergleich zwischen GUI- und Konsolenanwendungen 299
9.1.2 GUI-Technologien für die .NET -Plattform 300

9.2 Elementare Bausteine einer WPF-Anwendung 301
9.2.1 Eine minimale WPF-Anwendung (ohne XAML) 301
9.2.2 Anwendungsfenster und die Klasse Window 303
9.2.3 Windows-Nachrichten und die Klasse Application 306

9.3 Delegaten und CLR-Ereignisse 308
9.3.1 Delegaten 308
9.3.1.1 Delegatentypen definieren 309
9.3.1.2 Delegatenobjekte erzeugen und aufrufen 310
9.3.1.3 Delegatenobjekte kombinieren 311
9.3.1.4 Anonyme Methoden 312
9.3.1.5 Generische Delegaten 313
9.3.2 CLR-Ereignisse 314
9.3.2.1 Innenarchitektur von CLR-Ereignissen 314
9.3.2.2 Behandlungsmethoden registrieren 316
9.3.2.3 CLR-Ereignisse anbieten 319

9.4 Die Extended Application Markup Language (XAML) 322
9.4.1 Elementare Regeln zum Aufbau einer XML-Datei 322
9.4.2 XAML-Kurzbeschreibung 323
9.4.2.1 Wurzelelement 324
9.4.2.2 Instanzelemente 325
9.4.2.3 Eigenschaftswerte per Attribut oder Eigenschaftselement zuweisen 325
9.4.2.4 Vereinfachte Wertzuweisung bei der Inhaltseigenschaft 326
9.4.2.5 Eine Zeichenfolge als Wert für die Inhaltseigenschaft 327
9.4.2.6 Attributsyntax für Ereignisse 328
9.4.2.7 Attributsyntax für Markuperweiterungen 328
9.4.3 Code-Behind -Dateien 329
9.4.4 XAML-Verarbeitung beim Erstellen und Starten einer WPF-Anwendung 331

9.5 Spaß- und Motivationszufuhr 333
9.5.1 Projekt anlegen mit Vorlage *WPF - Anwendung* 334
9.5.2 Steuerelemente aus der Toolbox übernehmen 338
9.5.3 Positionen, Größen und Verankerungspunkte der Steuerelemente 338
9.5.4 Eigenschaften der Steuerelemente ändern 342
9.5.5 Automatisch erstellter Quellcode 343
9.5.6 Click-Ereignisbehandlung zum Befehlsschalter (Teil 1) 344
9.5.7 Formatierung der Listenelemente per DataTemplate-Objekt 347
9.5.8 Klick-Ereignisbehandlung zum Befehlsschalter (Teil 2) 348
9.5.9 Doppelklick-Ereignisbehandlung zum ListBox-Steuerelement 349
9.5.10 Symbole für die Anwendung und das Hauptfenster 349
9.5.11 Selbstkritik, Ausblick 352

9.6 Routingereignisse 352
9.6.1 Ein kurzer Blick hinter die Kulissen des WPF-Ereignissystems 353
9.6.2 Routingstrategien 353
9.6.3 Eine Beobachtungsstudie 354
9.6.4 Ereignisbehandlung durch statische Methoden 357

9.7 Layoutcontainer 359
9.7.1 Grid 360
9.7.1.1 Zeilen und Spalten definieren 360
9.7.1.2 Platzaufteilung 362
9.7.1.3 Platzanweisung 363
9.7.1.4 Angefügte Eigenschaften 363
9.7.1.5 Mehrzellige Elemente 365
9.7.2 DockPanel 366
9.7.3 StackPanel 367
9.7.4 WrapPanel 368
9.7.5 UniformGrid 368
9.7.6 Canvas 369
9.7.7 Geschachtelte Layoutcontainer 369

9.8 Basiswissen über Steuerelemente 370
9.8.1 Abstammungsverhältnisse 371
9.8.2 Einsatz von Steuerelementen in WPF-Anwendungen 372
9.8.2.1 Steuerelement als Memberobjekt in eine Fensterklasse aufnehmen 372
9.8.2.2 Steuerelement positionieren 373
9.8.2.3 Steuerelement konfigurieren 374
9.8.2.4 Ereignisbehandlungsmethoden 374
9.8.2.5 Ereignisse und On-Methoden 375
9.8.3 Standardkomponenten 378
9.8.3.1 Befehlsschalter 378
9.8.3.2 Kontrollkästchen und Optionsfelder 383
9.8.3.3 Texteingabefelder 386
9.8.3.4 Listen- und Kombinationsfelder 388
9.8.3.5 ToolTip 393

9.9 Zusammenfassung zu Kapitel 9 393

9.10 Übungsaufgaben zu Kapitel 9 395

10 AUSNAHMEBEHANDLUNG 397

10.1 Unbehandelte Ausnahmen 398

10.2 Ausnahmen abfangen 399
10.2.1 Die try-catch-finally - Anweisung 399
10.2.1.1 Ausnahmebehandlung per catch-Block 400
10.2.1.2 finally 402
10.2.2 Programmablauf bei der Ausnahmebehandlung 404
10.2.2.1 Beispiel 404
10.2.2.2 Komplexe Fälle 406

10.3 Ausnahmeobjekte im Vergleich zur traditionellen Fehlerbehandlung 406

10.4 Ausnahme-Klassen im .NET - Framework 408

10.5 Ausnahmen werfen (throw) 410

10.6 Ausnahmen definieren 411

10.7 Übungsaufgaben zu Kapitel 10 414

11 ATTRIBUTE 417

11.1 Attribute vergeben 418

11.2 Attribute per Reflexion auswerten 420

11.3 Attribute definieren 422

11.4 Attribute für Assemblies und Module 423

11.5 Eine Auswahl nützlicher FCL-Attribute 425
11.5.1 Bitfelder per FlagsAttribute 425
11.5.2 Unions per StructLayoutAttribute und FieldOffsetAttribute 426

11.6 Übungsaufgaben zu Kapitel 11 427

12 EIN- UND AUSGABE ÜBER DATENSTRÖME 429

12.1 Datenströme aus Bytes 429
12.1.1 Das Grundprinzip 429
12.1.2 Beispiel 429
12.1.3 Wichtige Methoden und Eigenschaften der Basisklasse Stream 430
12.1.4 Schließen von Datenströmen 431
12.1.5 Ausnahmen behandeln 433
12.1.6 FileStream 434
12.1.6.1 Öffnungsmodus 435
12.1.6.2 Zugriffsmöglichkeiten für das erstellte FileStream-Objekt 435
12.1.6.3 Zugriffsmöglichkeiten für andere Interessenten 436

12.2 Verarbeitung von Daten mit höherem Typ 436
12.2.1 Schreiben und Lesen im Binärformat 437
12.2.2 Schreiben und Lesen im Textformat 440
12.2.3 Serialisieren von Objekten 444

12.3 Verwaltung von Dateien und Verzeichnissen 448
12.3.1 Dateiverwaltung 448
12.3.2 Ordnerverwaltung 450
12.3.3 Überwachung von Ordnern 451

12.4 Übungsaufgaben zu Kapitel 12 452

13 MULTITHREADING 453

13.1 Threads erzeugen 454
13.1.1 Die Klasse Thread 454
13.1.2 Produzent - Konsument - Beispiel 455
13.1.2.1 Die Klasse Lager 455
13.1.2.2 Die Klassen `Produzent` und `Konsument` 456
13.1.3 Sekundäre Threads starten 457
13.1.4 Klassen aus dem Anwendungsbereich und aus der Informatik 458

13.2 Threads synchronisieren 459
13.2.1 Die lock-Anweisung 460
13.2.2 Die Klasse Monitor 462
13.2.3 Die Klasse Mutex 462
13.2.4 Koordination per Wait() und Pulse() 463
13.2.5 Andere Verfahren zur Thread-Koordination 464
13.2.5.1 Ein Schläfchen in Ehren 464
13.2.5.2 Weck mich, wenn Du fertig bist 465
13.2.5.3 Signalisierungsobjekte 465
13.2.5.4 Die Klasse ReaderWriterLock 467

13.3 Threads stoppen 469

13.4 Thread-Lebensläufe 470
13.4.1 Scheduling und Prioritäten 470
13.4.2 Zustände von Threads 471

13.5 Deadlock 472

13.6 Treadpool 473

13.7 Asynchronous Programming Model 475
13.7.1 Die Delegatenmethoden BeginInvoke() und EndInvoke() 475
13.7.2 Asynchrone Schreib- und Leseoperationen bei Dateien 478

13.8 Timer 479
13.8.1 Regelmäßige Hintergrundaktivitäten 480
13.8.2 Regelmäßige Aktivitäten im GUI-Thread 482

13.9 Task Parallel Library (TPL) 483
13.9.1 Implizite Task-Erstellung über statische Methoden der Klasse Parallel 483
13.9.2 Explizite Task-Objekte 485
13.9.2.1 Aufgaben ohne bzw. mit Rückgabe erstellen 485
13.9.2.2 Parameterabhängige Aufgaben 486
13.9.2.3 Fortsetzungsaufgaben 487
13.9.2.4 Warten (auf Ausnahmen) 487
13.9.2.5 Aufgaben abbrechen 489

13.10 Übungsaufgaben zu Kapitel 13 491

14 NETZWERKPROGRAMMIERUNG 493

14.1 Wichtige Konzepte der Netzwerktechnologie 493
14.1.1 Das OSI-Modell 494
14.1.2 Zur Funktionsweise von Protokollstapeln 497
14.1.3 Optionen zur Netzwerkprogrammierung in C# 498

14.2 Internet - Ressourcen per Request/Response – Modell nutzen 498
14.2.1 Statische Webinhalte anfordern 498
14.2.2 Datei-Download per HTTP - Protokoll 501
14.2.3 Dynamische erstellte Webseiten per GET oder POST anfordern 503
14.2.3.1 Überblick 503
14.2.3.2 Arbeitsablauf 504
14.2.3.3 GET 506
14.2.3.4 POST 508

14.3 IP-Adressen bzw. Host-Namen ermitteln 509

14.4 Socket-Programmierung 510
14.4.1 TCP-Server 510
14.4.2 TCP-Klient 514
14.4.3 Simultane Bedienung mehrerer Klienten 515

14.5 Übungsaufgaben zu Kapitel 14 517

15 DATENBINDUNG 519

15.1 Quelle und Ziel einer Datenbindung 519

15.2 Datenkontext und Bindung 520
15.2.1 Die Eigenschaft DataContext 520
15.2.2 Die Klasse Binding 521
15.2.3 Unterstützung durch die Entwicklungsumgebung 522
15.2.4 Binding-Objekte ohne Pfadangabe 523

15.3 Formatierung per DataTemplate-Objekt 523

15.4 Bindungsmodus 524

15.5 View-Objekt zu einer Kollektion 525
15.5.1 Sortieren 525
15.5.2 Filtern 526

15.6 Listenelemente ergänzen oder entfernen 527

15.7 Validierung von Eingabedaten 528

15.8 Datenbindung zwischen zwei Steuerelementen 529

15.9 Übungsaufgaben zu Kapitel 15 531

16 FENSTER UND DIALOGE 533

16.1 Fenster erstellen und anzeigen 533
16.1.1 Unterstützung durch das Visual Studio 533
16.1.2 Modaler und nichtmodaler Auftritt 535
16.1.3 Besitzverhältnisse 536
16.1.4 Fenstergröße 536
16.1.5 Zustand eines Fensters 538

16.2 Modale Dialogfenster 538
16.2.1 Definition einer Klasse für ein modales Dialogfenster 538
16.2.2 Datenaustausch mit dem aufrufenden Fenster 540
16.2.3 Auftritt und Abgang 540
16.2.4 Übernehmen Sie keinen Aberglauben 541

16.3 Nicht-modale Dialogfenster 542
16.3.1 Konfiguration 542
16.3.2 Auftritt und Abgang 544
16.3.3 Kommunikation mit dem aufrufenden Fenster 544

17 ANWENDUNGS- UND BENUTZEREINSTELLUNGEN 547

17.1 Konfigurationsdateien im XML-Format 548
17.1.1 Aufbau einer .NET - Anwendungskonfigurationsdatei 548
17.1.2 Ablage im Dateisystem 551
17.1.3 Die Klasse ApplicationSettingsBase 553
17.1.4 Unterstützung durch das Visual Studio 555
17.1.5 Lesen und Speichen von Einstellungen 557
17.1.6 Validierung von Einstellungen 559
17.1.7 Einstellungen per WPF-Datenbindung übernehmen 559

17.2 Lese- und Schreibzugriffe auf die Windows-Registrierungsdatenbank 560
17.2.1 Elementare Eigenschaften der Registry 560
17.2.2 Registry-Bearbeitung in .NET – Programmen 561
17.2.2.1 In die Registry schreiben 562
17.2.2.2 Aus der Registry lesen 563

17.3 Übungsaufgaben zu Kapitel 17 564

18 2D-GRAFIK 565

18.1 Vektorgrafik mit Objekten aus der Shape-Klassenhierarchie 566
18.1.1 Canvas-Layoutcontainer und das metrische WPF-Koordinatensystem 568
18.1.2 Automatische Fensterrestauration 568
18.1.3 Spezielle Eigenschaften von Shape-Objekten 570
18.1.4 Konkrete Shape-Ableitungen 573
18.1.4.1 Line 573
18.1.4.2 Rectangle und Ellipse 573
18.1.4.3 Polyline und Polygon 574
18.1.4.4 Path 575
18.1.5 Pinsel 580
18.1.5.1 SolidColorBrush 580
18.1.5.2 LinearGradientBrush 580
18.1.5.3 RadialGradientBrush 582

18.2 Transformationen 582
18.2.1 RenderTransform vs. LayoutTransform 583
18.2.2 TranslateTransform 584
18.2.3 ScaleTransform 585
18.2.4 RotateTransform 586

18.3 Rastergrafiken 588
18.3.1 Die Klasse Image 588
18.3.2 Rastergrafiken zum Tapezieren 590

19 TEXT DARSTELLEN UND DRUCKEN 593

19.1 Die Klasse TextBlock für kurze Fließtexte 593

19.2 Schrifteigenschaften 594

19.3 Die Klasse FlowDocument für Fließtext mit Block-Elementen 595

19.4 XPS-basierte Druckausgabe 597
19.4.1 Aufbau eines XPS-Dokuments und zugehörige WPF-Klassen 598
19.4.2 FixedDocument und FixedPage 600
19.4.3 XpsDocumentWriter 601
19.4.4 Automatische Paginierung bei Fließtext 605
19.4.5 Asynchrones Drucken 606
19.4.6 Mehr Kontrolle über die Druckkonfiguration 607
19.4.6.1 PrintQueue 607
19.4.6.2 PrintServer 607
19.4.6.3 PrintDialog 607

20 DATENBANKPROGRAMMIERUNG MIT ADO.NET 609

20.1 Datenbankmanagementsysteme 609

20.2 Relationale Datenbanken 611
20.2.1 Tabellen 611
20.2.2 Beziehungen zwischen Tabellen 613

20.3 SQL 614
20.3.1 Überblick 614
20.3.2 Spalten abrufen 615
20.3.3 Fälle auswählen über die WHERE-Klausel 615
20.3.4 Daten aus mehreren Tabellen zusammenführen 616
20.3.5 Abfrageergebnis sortieren 617
20.3.6 Auswertungsfunktionen 617
20.3.7 Daten gruppieren 617

20.4 Microsoft SQL Server 2008 R2 Express Edition 617
20.4.1 Eigenschaften 617
20.4.2 Installation 618
20.4.2.1 SQL Server 2008 R2 Express with Tools 618
20.4.2.2 Beispieldatenbank Northwind 624
20.4.3 Konfiguration 628
20.4.3.1 Rechteverwaltung 628
20.4.3.2 Netzwerkzugriff via TCP/IP erlauben 633

20.5 ADO.NET 635
20.5.1 Überblick 635
20.5.1.1 Verbindungsloses versus verbindungsorientiertes Arbeiten 635
20.5.1.2 Provider 635
20.5.1.3 ADO.NET - Namensräume 636

20.5.2 Die Connection-Klassen 637
20.5.2.1 Verbindungszeichenfolgen für Microsofts SQL-Server 637
20.5.2.2 Eigenschaften, Ereignisse und Methoden 641
20.5.3 Die Command-Klassen 641
20.5.3.1 Eigenschaften und Methoden 641
20.5.3.2 Parametrisierte Abfragen 642
20.5.4 Die DataAdapter-Klassen 643
20.5.4.1 Zuständigkeiten 643
20.5.4.2 Datentransfer von der Datenbank zum DataSet-Objekt 644
20.5.4.3 Datentransfer vom DataSet-Objekt zur Datenbank 645
20.5.4.4 Schematransfer von der Datenbank zum DataSet-Objekt 645
20.5.5 DataSet und andere Provider-unabhängige Klassen 646
20.5.5.1 Abfrageergebnisse und Schema-Informationen aus einer Datenbank übernehmen 647
20.5.5.2 DataTable-Objekte modifizieren 648
20.5.5.3 Beziehungen zwischen Tabellen vereinbaren 654
20.5.6 Datenbank-Update 656
20.5.7 Zusammenspiel der ADO.NET - Klassen beim verbindungslosen Arbeiten 657
20.5.8 Einsatz des DataReaders 659

20.6 Typisierte DataSets 660
20.6.1 Zu einer Datenbankabfrage automatisch definierte Klassen 661
20.6.2 Anwendungsbeispiel 663
20.6.2.1 Datenquelle einrichten 663
20.6.2.2 Abfragen modifizieren 668
20.6.2.3 Datenverbundene Bedienelemente erstellen 669

20.7 Datenbanken mit der Entwicklungsumgebung bearbeiten 672
20.7.1 Tabellen anzeigen und bearbeiten 673
20.7.2 Datenbankdiagramm erstellen 674
20.7.3 Beziehungen definieren 675

21 LINQ 677

21.1 Erweiterungen der Programmiersprache C# 677
21.1.1 Implizite Typzuweisung bei lokalen Variablen 677
21.1.2 Instanz- und Listeninitialisierer 677
21.1.3 Anonyme Klassen 678
21.1.4 Lambda-Ausdrücke 679
21.1.5 Erweiterungsmethoden 680

21.2 LINQ to Objects 680
21.2.1 Die from-in - Klausel zur Definition einer Datenquelle 681
21.2.2 Projektion des Elementtyps der Quelle auf den Elementtyp der Abfrage 682
21.2.3 Filtern 683

22 ADO.NET ENTITY FRAMEWORK 685

22.1 Entity Data Model 686
22.1.1 EDM aus einer vorhandenen Datenbank generieren lassen 687
22.1.2 Entitäten mit (Navigations-)Eigenschaften und Assoziationen 691
22.1.3 CSDL-Elemente 693

22.2 Der Provider EntityClient 695

22.3 Object Services 696
22.3.1 Klassen und Objekte zu einem EDM 696
22.3.2 Objekte per LINQ to Entities erstellen und verwenden 700
22.3.3 Lazy Loading 702

22.4 Entity Framework im Vergleich zu traditionellen ADO.NET - Techniken 702

23 ANHANG 705

23.1 Operatorentabelle 705

23.2 Lösungsvorschläge zu den Übungsaufgaben 706
Kapitel 1 (Einleitung) 706
Kapitel 2 (Werkzeuge zum Entwickeln von C# - Programmen) 708
Kapitel 3 (Elementare Sprachelemente) 708
Abschnitt 3.1 (Einstieg) 708
Abschnitt 3.2 (Ausgabe bei Konsolenanwendungen) 709
Abschnitt 3.3 (Variablen und Datentypen) 709
Abschnitt 3.5 (Operatoren und Ausdrücke) 710
Abschnitt 3.7 (Anweisungen) 711
Kapitel 4 (Klassen und Objekte) 713
Kapitel 5 (Weitere .NETte Typen) 714
Kapitel 6 (Vererbung und Polymorphie) 715
Kapitel 7 (Typgenerizität und Kollektionen) 716
Kapitel 8 (Interfaces) 716
Kapitel 9 (Einstieg in die GUI-Programmierung mit WPF-Technik) 717
Kapitel 10 (Ausnahmebehandlung) 717
Kapitel 11 (Attribute) 718
Kapitel 12 (Ein- und Ausgabe über Datenströme) 718
Kapitel 13 (Multithreading) 718
Kapitel 14 (Netzwerkprogrammierung) 719
Kapitel 15 (Datenbindung) 719
Kapitel 17 (Anwendungs- und Benutzereinstellungen) 719

LITERATUR 721

STICHWORTREGISTER 723

1 Einleitung

Im ersten Kapitel geht es zunächst um die Denk- und Arbeitsweise (leicht übertrieben: die *Weltanschauung*) der objektorientierten Programmierung. Danach werden die .NET - Plattform und ihre wohl wichtigste Programmiersprache C# vorgestellt.

1.1 Beispiel für die objektorientierte Softwareentwicklung mit C#

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

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

- den betroffenen Gegenstandsbereich *modellieren*
  Beispiel: In einem Programm zur Verwaltung einer Spedition sind z.B. Fahrer, Fahrzeuge, Servicestationen, Zielpunkte etc. und die kommunikativen Prozesse zu repräsentieren.
- *Algorithmen* realisieren, die in endlich vielen Schritten und unter Verwendung von endlich vielen Betriebsmitteln (z.B. CPU-Leistung, Arbeitsspeicher) bestimmte Ausgangszustände in akzeptable Zielzustände überführen.
  Beispiel: Im Speditionsprogramm muss u.a. für jede Fahrt zu den meist mehreren (Ent-)ladestationen eine optimale Route ermittelt werden (hinsichtlich Entfernung, Fahrtzeit, Mautkosten etc.).

Wir wollen präzisere und komplettere Definitionen zum komplexen Begriff eines Computerprogramms den Lehrbüchern überlassen (siehe z.B. Goll et al. 2000) 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 für das Programmieren mit C# nicht besonders repräsentativ, z.B. ist von der Objektorientierung außer einem gewissen Formalismus eigentlich nichts vorhanden.
- Repräsentative C# - Programme eignen sich in der Regel wegen ihrer Länge und Komplexität (aus der Sicht des Anfängers) nicht für eine Detailanalyse. Insbesondere können wir das eben erfolgreich zur Illustration einer realen Aufgabenstellung verwendete, aber potentiell sehr aufwändige, Speditionsverwaltungsprogramm jetzt nicht im Detail vorstellen.

Wir analysieren 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.

1.1.1 Objektorientierte Analyse und Modellierung

Einer objektorientierten Programmentwicklung geht die **objektorientierte Analyse** der Aufgabenstellung voran mit dem Ziel einer Modellierung durch kooperierende **Klassen**. Man identifiziert per **Abstraktion** die beteiligten **Objektsorten** und definiert für sie jeweils eine **Klasse**. Eine solche Klasse ist gekennzeichnet durch:

- **Merkmale** (**Instanz- bzw. Klassenvariablen, Felder**)
  Viele Merkmal gehören zu den *Objekten* bzw. *Instanzen* der Klasse (z.B. Zähler und Nenner eines Bruchs), manche gehören zur Klasse selbst (z.B. Anzahl der bereits erzeugten Brüche). Im letztlich entstehenden Programm landet jede Merkmalsausprägung in einer so genannten

Variablen. Dies ist ein benannter Speicherplatz, der Werte eines bestimmten Typs (z.B. Zahlen, Zeichen) 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. das Kürzen bei Brüchen) oder der Klasse selbst (z.B. das Erstellen neuer Objekte). Im letztlich entstehenden Programm sind die Handlungskompetenzen durch so genannte *Methoden* repräsentiert. Diese ausführbaren Programmbestandteile enthalten die oben angesprochenen Algorithmen. Die Kommunikation zwischen Klassen bzw. 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 aber andererseits 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 nun 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 Instantiieren (Erzeugen von Objekten) gedacht ist sondern als Akteur.

In unserem Bruchrechnungsbeispiel können wir uns bei der objektorientierten Analyse vorläufig wohl auf die Klasse der *Brüche* beschränken. Beim möglichen Ausbau des Programms zu einem Bruchrechnungstrainer kommen jedoch sicher weitere Klassen hinzu (z.B. Aufgabe, Übungsaufgabe, Testaufgabe).

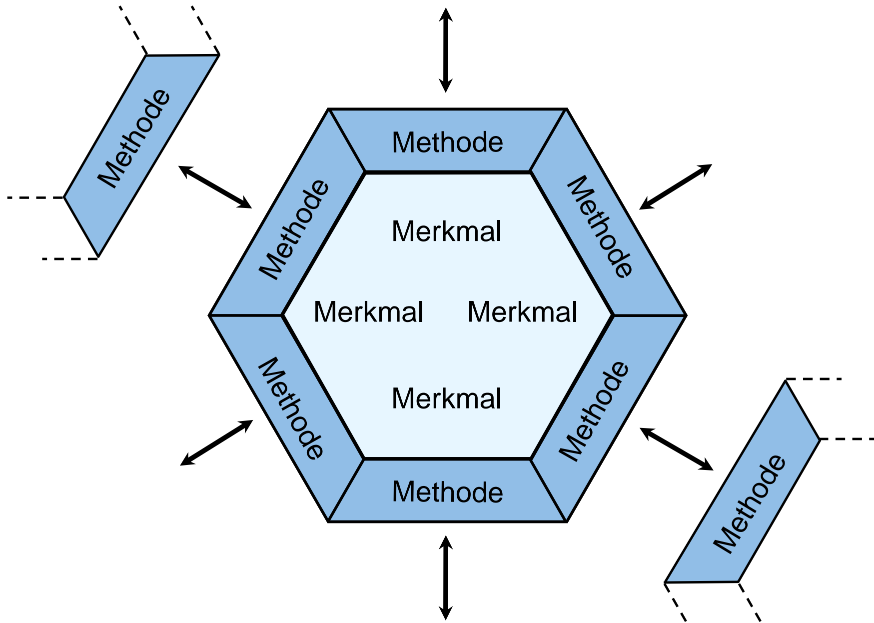
Dass Zähler und Nenner die zentralen **Merkmale** eines Bruchs sind, bedarf keiner Begründung. Sie werden in der Klassendefinition durch ganzzahlige Felder (C# - Datentyp **int**) repräsentiert, die folgende Namen erhalten sollen:

- `zaehler`
- `nenner`

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 direktem Zugriff durch fremde Klassen geschützt sein. So kann sichergestellt werden, dass nur sinnvolle Änderungen der Merkmalsausprägungen möglich sind.

Wie die folgende, an Goll et al. (2000) angelehnte, Abbildung zeigt, bilden die in aufrufbaren **Methoden** realisierten **Handlungskompetenzen** einer Klasse demgegenüber ihre öffentlich zugängliche **Schnittstelle** zur Kommunikation mit anderen Klassen:



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 Instanzmethode 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). Das objektorientierte Modellieren eines Gegenstandbereiches ist nicht unbedingt eine direkte Abbildung, sondern eine *Rekonstruktion*. Einerseits soll der Gegenstandsbereich im Modell gut repräsentiert sein, andererseits soll eine möglichst stabile, gut erweiterbare und wieder verwendbare Software entstehen.

Um fremden Klassen trotz Datenkapselung die Veränderung einer Merkmalsausprägung zu erlauben, müssen entsprechende Methoden (mit geeigneten Kontrollmechanismen) angeboten werden. Unsere `Bruch`-Klasse sollte wohl über Methoden zum Verändern von Zähler und Nenner verfügen. Bei einem geschützten Merkmal ist auch der direkte *Lese*zugriff ausgeschlossen, so dass im `Bruch`-Beispiel auch noch Methoden zum Ermitteln von Zähler und Nenner ratsam 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 erhebliche Vorteile realisiert:

- **Stabilität**
  Die Merkmalsausprägungen sind vor unsinnigen und gefährlichen Zugriffen geschützt, wenn Veränderungen nur über die vom Klassendesigner entworfenen Methoden möglich sind. Treten doch Fehler auf, sind diese leichter zu identifizieren, weil nur wenige Methoden verantwortlich sein können.
- **Produktivität**
  Durch Datenkapselung wird die **Modularisierung** unterstützt, so dass bei der Entwicklung großer Softwaresysteme zahlreiche Programmierer 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 (***F**ramework **C**lass **L**ibrary*, FCL) (siehe Abschnitt 1.2.7), von denen alle C# - Programmierer regen Gebrauch machen. Auch die Klasse `Bruch` aus dem Beispielprojekt besitzt einiges Potential zur Wiederverwendung. Wir werden unser Beispiel noch um die Klasse `BruchAddition` erweitern, welche `Bruch`-Objekte benutzt, um ein Programm zur Addition von Brüchen zu realisieren.

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

```
b1.Zaehler = 4;
```

Während der Entwickler Zugriffs*methoden* bereitzustellen hat (siehe unten), sieht der Nutzer ein öffentliches *Merkmal* der Klasse. 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 Klasse durch die C# - Eigenschaften unscharf zu werden scheint. Letztlich erspart eine C# - Eigenschaft wie `Zaehler` dem Nutzer lediglich die Verwendung von Zugriffs*methoden*, z.B.

```
b1.SetzeZaehler(4);
```

Damit kann man die C# - Eigenschaften in die Kategorie *syntactic sugar* (Mössenböck 2003, S. 3) einordnen, was aber keine Abwertung bedeuten soll. Wegen ihrer intensiven Nutzung in C# - Programmen ist ein Auftritt im ersten Kursbeispiel wohl trotz den angesprochenen didaktischen Probleme gerechtfertigt.

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

- `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. Sie ist aber der Einheitlichkeit und damit der Einfachheit halber angebracht und hält die Möglichkeit offen, das Merkmal `zaehler` einmal anders zu realisieren.
- `Kuerze()`
  Das Objekt wird beauftragt, `zaehler` und `nenner` zu kürzen. Welcher Algorithmus dazu benutzt wird, bleibt dem Klassendesigner überlassen.
- `Addiere(Bruch b)`
  Das Objekt wird beauftragt, den als Parameter (siehe unten) übergebenen Bruch zum eigenen Wert zu addieren.

- `Frage()`
  Das Objekt wird beauftragt, `zaehler` und `nenner` beim Anwender via Konsole (Eingabeaufforderung) zu erfragen.
- `Zeige()`
  Das Objekt wird 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 `Bruch`-Klasse enthält folgende Member:

- Felder
  `zaehler`, `nenner`
- Eigenschaften (Hier handelt es sich letztlich um Paare von Methoden.)
  `Zaehler`, `Nenner`
- Methoden
  `Kuerze()`, `Addiere()`, `Frage()` und `Zeige()`

Durch die (in C# keinesfalls beliebige) Groß-/Kleinschreibung der Namen wird die Unterscheidung von privaten Merkmalen (per Konvention mit kleinem Anfangsbuchstaben) und öffentlichen Eigenschaften bzw. Methoden (per Konvention mit großem Anfangsbuchstaben) erleichtert. 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!? 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 Gegenstandsbereichs standardisiert und übersichtlich zu beschreiben, wurde die **Unified Modeling Language** (UML) entwickelt. 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) Merkmale angesprochen werden. Es hängt vom Adressaten eines Klassendiagramms (z.B. Entwickler-Teamkollege oder Anwender der Klasse) ab, ob man die Felder, die Eigenschaften oder beides angibt.
- 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 der gewünschten Ausführungsart bzw. mit Details der gesendeten Nachricht) den Datentyp an.

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

Bruch
Zaehler: int Nenner: int
Kuerze() Addiere(Bruch b) Frage() Zeige()

Sind bei einer Anwendung *mehrere* Klassen beteiligt, dann sind auch die *Beziehungen* zwischen den Klassen wesentliche Bestandteile des UML-Modells.

Nach der sorgfältigen Modellierung per UML muss übrigens die Kodierung eines Softwaresystems nicht am Punkt Null beginnen, weil professionelle UML-Werkzeuge (z.B. in der Ultimate-Version von Microsofts Entwicklungsumgebung *Visual Studio* enthalten) Teile des Quellcodes automatisch aus dem Modell erzeugen können.

Das relativ einfache Einstiegsbeispiel sollte Sie nicht dazu verleiten, den Begriff *Objekt* auf *Gegenstände* zu beschränken. Auch *Ereignisse* wie z.B. die Fehler eines Schülers in einem entsprechend ausgebauten Bruchrechnungsprogramm kommen als *Objekte* in Frage.

1.1.2 Objektorientierte Programmierung

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

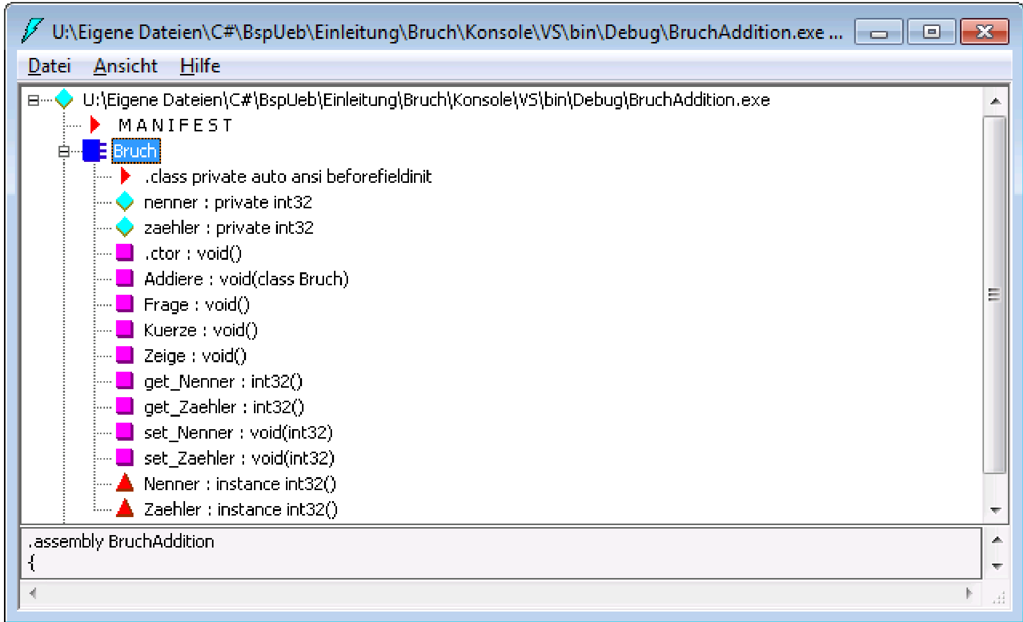
Zwar sind Ihnen die meisten Details der folgenden Klassendefinition selbstverständlich jetzt noch fremd, doch sind die Variablendeklarationen sowie die Eigenschafts- und Methodenimplementationen als zentrale Bestandteile leicht zu erkennen:

```
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 ggt = 0;
            int az = Math.Abs(zaehler);
            int an = Math.Abs(nenner);
            do {
               if (az == an)
                  ggt = az;
               else
                  if (az > an)
                     az = az - an;
                  else
                     an = an - az;
            } while (ggt == 0);

            zaehler /= ggt;
            nenner  /= ggt;
         }
      }

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

      public void Frage() {
         Console.Write("Zaehler: ");
         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 explizite Verwendung des Modifikators **public** für die Verwendung in klassenfremden Methoden frei gegeben. 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 C# 2010 Express Edition* (siehe Abschnitt 2.2.2) installierte Hilfsprogramm **ILDasm** liefert die folgende Beschreibung der Klasse `Bruch`:



Offenbar werden hier Felder durch eine türkise Raute ( ), Methoden durch ein magentafarbiges Quadrat ( ) und Eigenschaften durch ein rotes, mit der Spitze nach oben zeigendes Dreieck ( ) dargestellt.

Hier bestätigt sich übrigens die Andeutung von Abschnitt 1.1.1, 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).

1.1.3 Algorithmen

Am Anfang von Abschnitt 1.1 wurden mit der *Modellierung des Gegenstandbereichs* und der *Realisierung von Algorithmen* zwei wichtige Aufgaben der Softwareentwicklung 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 Voraussetzungen, 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.

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 u und v ($u > v > 0$) 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:

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

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 (u, $u - v$) dieselben Mengen gemeinsamer Teiler besitzen, sind auch die größten gemeinsamen Teiler identisch. Weil die Zahl Eins als trivialer Teiler zugelassen ist, existiert übrigens zu zwei natürlichen Zahlen (1, 2, 3, …) immer ein größter gemeinsamer Teiler, der eventuell gleich Eins ist.

Dieses Ergebnis wird 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 verkleinerten 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 zu Abschnitt 3.7 werden Sie eine erheblich effizientere Variante implementieren.

1.1.4 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 `Bruch`-Klasse in einer Konsolenanwendung zur Addition von zwei Brüchen. Dabei bringen wir einen Akteur ins Spiel, der in einem einfachen sequentiellen Handlungsplan `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 besondere Klasse geben muss, die eine Methode mit dem Namen **Main()** in ihren klassenbezogenen Handlungsrepertoire besitzt. Beim Start eines Programms wird seine Startklasse ausfindig gemacht und aufgefordert, ihre **Main()**-Methode auszuführen.

Es bietet sich an, die oben angedachte Handlungssequenz des Bruchadditionsprogramms in der obligatorischen Startmethode unterzubringen.

Obwohl prinzipiell möglich, erscheint es nicht sinnvoll, die auf Wiederverwendbarkeit hin konzipierte `Bruch`-Klasse 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 hat. Ihr Handlungsrepertoire kann sich auf die *Klassenmethode* **Main()** zur Ablaufsteuerung im Bruchadditionsprogramm beschränken. Indem wir eine andere Klasse zum Starten verwenden, wird u.a. gleich demonstriert, wie leicht das Hauptergebnis unserer Arbeit (die `Bruch`-Klasse) für thematisch verwandte Projekte genutzt werden kann.

In der `BruchAddition`–Methode **Main()** werden zwei Objekte (Instanzen) aus der Klasse `Bruch` erzeugt und mit der Ausführung verschiedener Methoden beauftragt:

Quellcode in **BruchAddition.cs**

```
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();
   }
}
```

Ein- und Ausgabe

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


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


Summe
    4
  -----
    7
```

Zum Ausprobieren startet man aus dem Ordner (zu finden an der im Vorwort vereinbarten Stelle)

...\BspUeb\Einleitung\Bruch\Konsole\VS\bin\Debug

das Programm **BruchAddition.exe**, z.B.:

```
U:\Eigene Dateien\C#\BspUeb\Einleitung\Bruch\Konsole\VS\bin\Debug\BruchAddition.exe

1. Bruch
Zaehler: 20
Nenner : 84
    5
  -----
    21


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


Summe
    4
  -----
    7
```

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

Wir haben zur Lösung der Aufgabe, ein Programm für die Addition von 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 Zugang zu kritischen Merkmalen haben (Datenkapselung). Sollten doch Fehler auftreten, sind die Ursachen in der Regel schnell identifiziert.

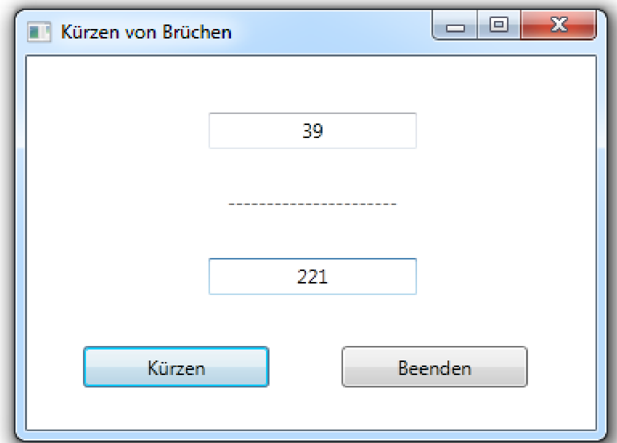
  Wir müssen bei der Definition der Klasse `Bruch` ihre allgemeine Verfügbarkeit explizit mit dem Zugriffsmodifikator **public** genehmigen. Per Voreinstellung ist eine Klasse nur intern (in der eigenen Übersetzungseinheit) verfügbar.

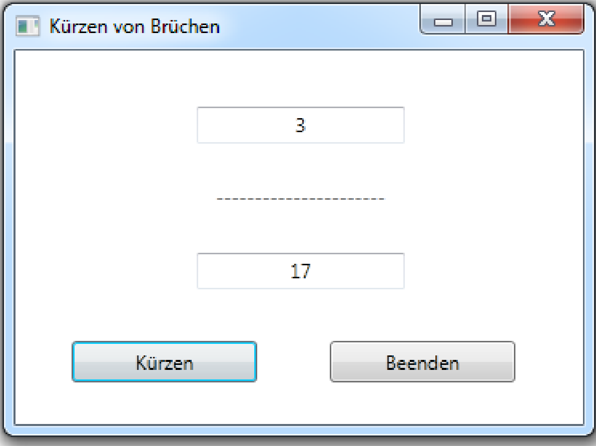
- Die Klasse `BruchAddition` dient *nicht* als Bauplan für Objekte, sondern enthält eine Klassenmethode **Main**(), die beim Programmstart automatisch aufgerufen wird und dann für einen speziellen Einsatz von `Bruch`-Objekten sorgt. Mit einer Wiederverwendung des `BruchAddition`-Quellcodes in anderen 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**. Es sind aber Quellcodedateien mit mehreren Klassen und einem beliebigen Namen erlaubt.

1.1.5 Ausblick auf Anwendungen mit graphischer Benutzerschnittstelle

Das obige Beispielprogramm arbeitet der Einfachheit halber mit einer Konsolen-orientierten Ein- und Ausgabe. Nachdem wir in dieser übersichtlichen Umgebung grundlegende Sprachelemente kennen gelernt haben, werden wir uns selbstverständlich auch mit der Programmierung von graphischen Benutzerschnittstellen beschäftigen. In folgendem 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:





Mit dem Quellcode zur Gestaltung der graphischen Oberfläche könnten Sie im Moment noch nicht allzu viel anfangen. Am Ende des Kurses werden Sie derartige Anwendungen aber mit Leichtigkeit erstellen.

Zum Ausprobieren startet man aus dem Ordner

...\BspUeb\Einleitung\Bruch\GUI\bin\Debug

das Programm **BruchKürzenGui.exe**.

1.1.6 Zusammenfassung zu Abschnitt 1.1

Im Abschnitt 1.1 sollten Sie einen ersten Eindruck von der Softwareentwicklung mit C# gewinnen. Alle dabei erwähnten Konzepte der objektorientierter Programmierung und 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 von Abschnitt 1.1 zu wiederholen:

- Vor der Programmentwicklung findet die objektorientierte Analyse der Aufgabenstellung statt. Dabei werden per Abstraktion die beteiligten Klassen identifiziert.
- Ein Programm besteht aus Klassen
- Eine Klasse ist charakterisiert durch Merkmale und Methoden.
- Eine Klasse dient als Bauplan für Objekte, kann aber auch selbst aktiv werden (Methoden ausführen und aufrufen).
- Ein Merkmal bzw. eine Methode wird entweder den Objekten einer Klasse oder der Klasse selbst zugeordnet.
- In den Methodendefinitionen werden Algorithmen realisiert, in der Regel unter Verwendung von zahlreichen vordefinierten Klassen aus diversen Bibliotheken.
- 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“.

1.2 Das .NET – Framework

Eben haben Sie C# als eine Programmiersprache kennen gelernt, die Ausdrucksmittel zur Modellierung von Anwendungsbereichen und zur Formulierung von Algorithmen bereitstellt. Unter einem *Programm* wurde dabei der vom Entwickler zu formulierende *Quellcode* verstanden. Während *Sie* derartige Texte bald ohne 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 wird z.B. der Wert Vier 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, heutzutage immerhin viele Milliarden Befehle pro Sekunde (*Instructions Per Second*, IPS).

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.

1.2.1 Überblick

Beim .NET – Framework handelt es sich um eine von der Firma Microsoft entwickelte Plattform (Sammlung von Technologien) zur Modernisierung und Vereinfachung der Anwendungsentwicklung unter Windows. Als Orientierung für die nächsten Abschnitte kann die folgende, stark vereinfachte Darstellung des .NET - Frameworks dienen:

.NET - Anwendung in einer CLS-Programmiersprache, z.B. C# (*Common Language Specification*)
übersetzte .NET - Anwendung in MSIL (*Microsoft Intermediate Language*)
Basisklassenbibliothek (FCL, *Framework Class Library*)
Laufzeitumgebung (CLR, *Common Language Runtime*)
Betriebssystem (Microsoft-System ab Windows 98, dank Mono aber auch Linux, MacOS-X oder UNIX)
Hardware

Seit Windows *Vista* ist das .NET – Framework fester Systembestandteil. Bei älteren Versionen (ab Windows 98) kann ein bei Microsoft kostenlos verfügbares .NET Framework - Paket nachinstalliert werden (zur Beschaffung und Installation siehe Abschnitt 1.2.2). Dieses Paket darf auch von Entwicklern zusammen mit eigenen .NET - Anwendungen vertrieben werden. Über Installationspakete aus dem Open Source - Projekt *Mono*[1] ist das .NET - Framework auch für Linux, MacOS-X und UNIX (genauer: Solaris) verfügbar.

Vom .NET – Framework sind bisher folgende Versionen erschienen:

Version	Erscheinungsjahr
1.0	2002
1.1	2003
2.0	2005
3.0	2006
3.5	2007
4.0	2010

Wir werden im Kurs lange Zeit mit dem Funktionsumfang der Version 2.0 auskommen, aber im weiteren Verlauf auch Bestandteile der neuen Versionen einsetzen:

- Das mit .NET 3.0 unter dem Namen *Windows Presentations Foundation* (WPF) eingeführte Framework für multimedial ambitionierte Bedienoberflächen, das die ältere WinForms-Bibliothek ersetzen soll, hat inzwischen Marktreife erlangt, so dass man bei neuen Projekten die WPF-Technik trotz der höheren Hardware-Anforderungen verwenden sollte. *Benutzen* werden wir das neue Framework von Anfang an, weil Microsoft mit dem Visual Studio 2010, also der im Kurs bevorzugten Entwicklungsumgebung, erstmals eine große Anwendung auf WPF umgestellt hat.
- Bei der Datenbankprogrammierung nutzen wir Optionen von .NET 3.5 (*__L__anguage __I__ntegrated __Q__uery*, LINQ).

[1] Webadresse: http://www.mono-project.com/Main_Page

Die .NET – Version 4.0 bringt vor allem eine „Kantenglättung“ (Schwichtenberg 2010, S. 139) und Detailverbesserungen (z.B. bei der Unterstützung von Mehrkernprozessoren).

Zwar besitzen das .NET-Framework und die Programmiersprache C# jeweils eine eigenständige Versionierung, doch sind die beiden Zeitreihen mit den Versionsständen stark korreliert, wie die folgende Tabelle aus Schwichtenberg (2009a) zeigt:

.NET - Framework	C#
1.0	1.0
1.1	1.1
2.0	2.0
3.0	2.0
3.5	3.0
4.0	4.0

1.2.2 Installation

Das zum Ausführen von .NET - Programmen erforderliche Framework (mit der CLR, der Klassenbibliothek FCL und den Compilern für C#, VB.NET etc.) ist in Windows Vista (in der Version 3.0) und in Windows 7 (in der Version 3.5) bereits enthalten und kann für andere Windows-Versionen nachinstalliert werden. Bei der Installation von Microsofts *Visual C# 2010 Express Edition* (siehe Abschnitt 2.2.2) landet das .NET - Framework (in der Version 4.0) automatisch auf der Festplatte. Wer diese Entwicklungsumgebung verwendet, muss das .NET – Framework also nicht separat installieren. Auch auf den Rechnern Ihrer Kunden ist in der Regel bereits ein .NET – Framework vorhanden. Wenn das Framework (in der benötigten Version) fehlt, kann man es kostenlos via Internet von Microsoft beziehen und auf Wunsch mit eigenen Anwendungen ausliefern.[1] Bei den Paketnamen ist zu beachten, dass der lange übliche Namensbestandteil *Redistributable* seit der Version 3.5 entfallen ist.

Je nach Version ist sowohl ein Online-Installer verfügbar, der die benötigten Installationsbestandteile während der Installation ermittelt und von einem Server lädt als auch ein (deutlich größerer) Offline-Installer, der kompatible Systeme ohne Internetzugriff versorgen kann.

Als Installationsordner werden (ohne Änderungsmöglichkeit) verwendet:

.NET 2.0	x86: %SystemRoot%\Microsoft.NET\Framework\v2.0.50727 x64: %SystemRoot%\Microsoft.NET\Framework64\v2.0.50727
.NET 3.0	x86: %SystemRoot%\Microsoft.NET\Framework\v3.0 x64: %SystemRoot%\Microsoft.NET\Framework64\v3.0
.NET 3.5	x86: %SystemRoot%\Microsoft.NET\Framework\v3.5 x64: %SystemRoot%\Microsoft.NET\Framework64\v3.5
.NET 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 1.2.5.5) laufen, werden dort zwei .NET – Laufzeitumgebungen (mit 32 bzw. 64 Bit) installiert.

Neuere Framework-Pakete installieren freundlicherweise ältere Versionen gleich mit. Dies ist sinnvoll, weil ältere .NET - Programme zur Vermeidung von Versionskonflikten stets von einer CLR auf ihrem eigenen Versionsstand ausgeführt werden wollen.

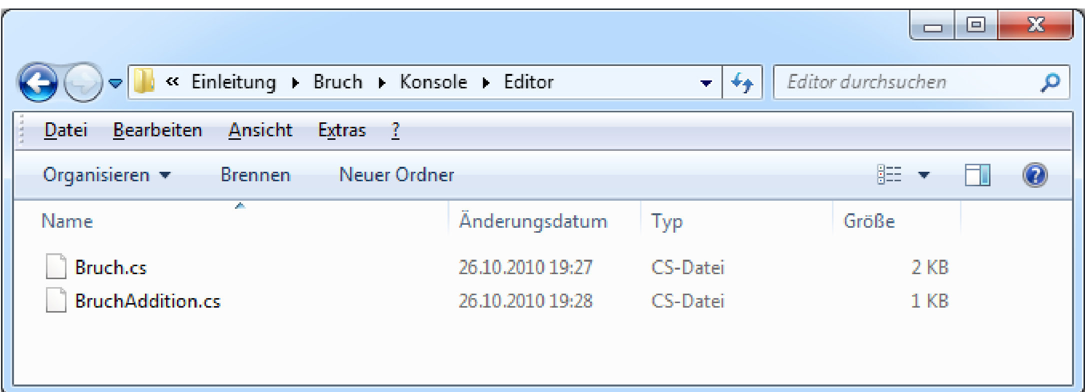
---

[1] Eine Quelle zum Herunterladen via Internet ist (z.B. per Suchmaschine) leicht zu finden, so dass auf die Angabe von länglichen, versionsabhängigen und oft nicht sehr zeitstabilen Internet-Adressen verzichtet wird.

1.2.3 C#-Compiler und MSIL

Den (z.B. mit einem beliebigen Texteditor verfassten) C# - Quellcode übersetzt der C# - **Compiler** in die **Microsoft Intermediate Language** (**MSIL**), oft kurz als **IL** (**Intermediate Language**) bezeichnet. Wenngleich dieser Zwischencode von den heute üblichen Prozessoren noch *nicht* direkt ausgeführt werden kann, hat er doch bereits viele Verarbeitungsschritte auf dem Weg vom Quell- zum Maschinencode durchlaufen. Weil kompakter als Maschinencode, eignet sind der MSIL-Code gut für die Übertragung über Netzwerke. Die Übersetzung des Zwischencodes in die Maschinensprache einer konkreten CPU geschieht *just-in-time* bei der Ausführung des Programms durch die CLR (siehe Abschnitt 1.2.6).[1]

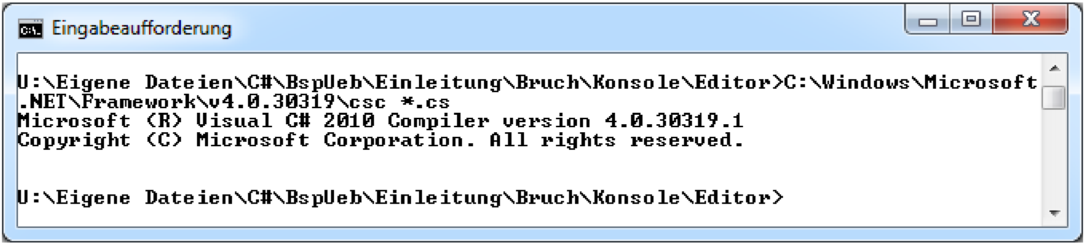
Befinden sich die beiden Quellcodedateien **Bruch.cs** und **BruchAddition.cs** im aktuellen Verzeichnis



eines Konsolenfensters, dann kann ihre Übersetzung durch den C# - Compiler **csc.exe** der 32-bittigen .NET-Version 4.0 mit dem folgenden Kommando

```
%SystemRoot%\Microsoft.NET\Framework\v4.0.30319\csc *.cs
```

veranlasst werden:



Weil sich der Ordner

%SystemRoot%\Microsoft.NET\Framework\v4.0.30319

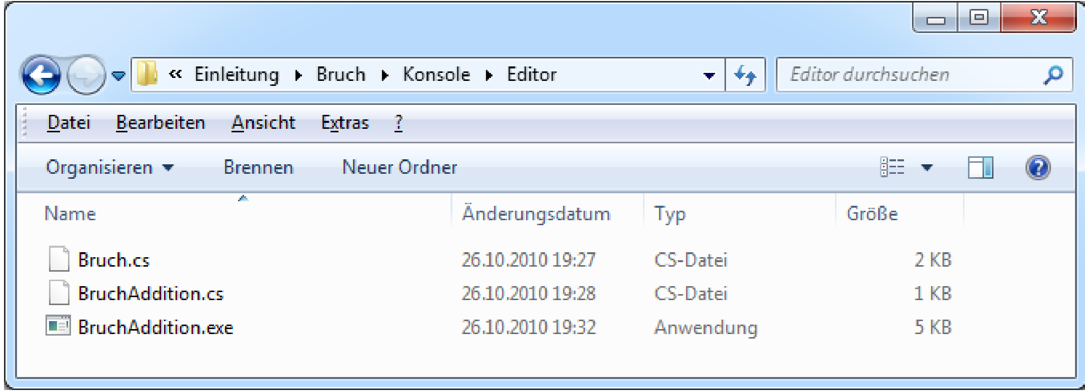
mit dem C# - Compiler per Voreinstellung *nicht* im Suchpfad für ausführbare Programme befindet, muss er im Aufruf angegeben werden. Der eigentliche Auftrag an den Compiler, sämtliche Dateien im aktuellen Verzeichnis mit der Namenserweiterung **.cs** zu übersetzen, ist sehr übersichtlich:

```
csc *.cs
```

Wir werden uns in Abschnitt 2.1.2 noch mit einigen Details des Compiler-Aufrufs beschäftigen.

Im Übersetzungsergebnis **BruchAddition.exe**

[1] Die Übersetzung vom MSIL-Code in Maschinencode sollte auf jeder Plattform gelingen, für die eine CLR verfügbar ist. Die auf einem Rechner mit 32-bittiger Windows-Version erstellten .NET-Programme sollten sich also problemlos in einer 64-Bit – Umgebung nutzen lassen (ohne Kompatibilitätsmodus). Grundsätzlich ist diese Portabilität tatsächlich gegeben, jedoch wird in Abschnitt 1.2.5.5 von Ausnahmen zu berichten sein.



ist u.a. der MSIL-Code der beiden Klassen `Bruch` und `BruchAddition` enthalten. Besonders kurz sind die implizit zu einer C# - Eigenschaft (vgl. Abschnitt 1.1.1) vom Compiler erstellten **get-** und **set**-Methoden, z.B.:[1]

```
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 sieben Anweisungen besteht.

Die konzeptionelle Verwandtschaft der MSIL mit dem Bytecode der Java-Plattform ist unverkennbar. Von den Unterschieden zwischen beiden Technologien ist vor allem die **Sprachunabhängigkeit** der Microsoft-Lösung zu erwähnen. Alle .NET – Quellprogramme werden unabhängig von der verwendeten Programmiersprache in die MSIL übersetzt, die daher auch als **CIL** (*Common Intermediate Language*) bezeichnet wird. 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.

[1] Diese Ausgabe liefert das schon in Abschnitt 1.1.2 angesprochene Werkzeug **ILDasm** nach einem Doppelklick auf die interessierende Methode (siehe Seite 7).

1.2.4 Common Language Specification

Mittlerweile sind für viele Programmiersprachen MSIL-Compiler verfügbar, etliche werden sogar mit dem .NET - Framework - Paket ausgeliefert (z.B. **csc.exe** für C#, **vbc.exe** für Visual Basic .NET). Allerdings unterstützen die .NET-Compiler in der Regel nicht den gesamten MSIL-Sprachumfang, so dass sich mit den Compilern zu verschiedenen .NET-Sprachen durchaus Klassen produzieren lassen, die *nicht* zusammenarbeiten können (siehe Richter, 2006, S.50). 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.[1] Beschränkt man sich bei der Klassendefinition auf diesen kleinsten gemeinsamen Nenner, ist die Interoperabilität mit anderen CLS-kompatiblen Klassen sichergestellt. Man kann einen .NET – Compiler auffordern, die CLS-Kompatibilität zu überwachen, so dass z.B. der C# - Compiler im eben beschriebenen Beispiel warnt:

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

1.2.5 Assemblies und Metadaten

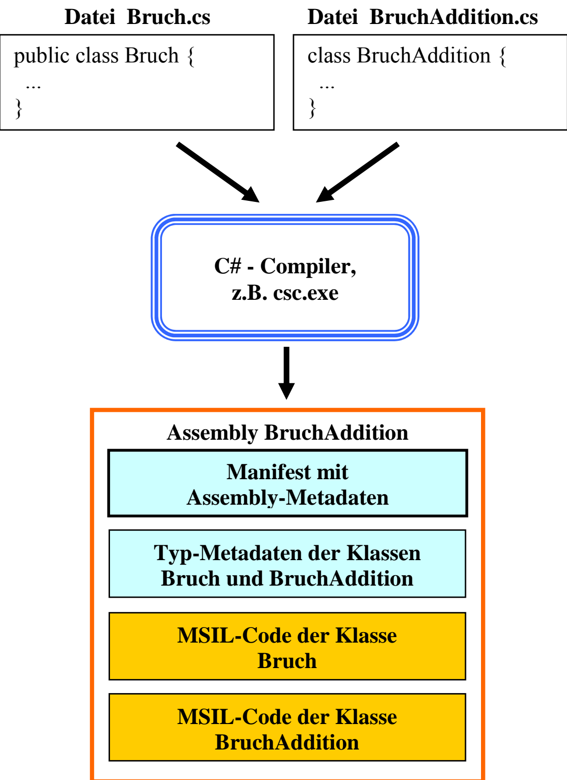
Die von einem .NET - Compiler erzeugten Binärdateien (mit MSIL-Code) werden als **Assemblies** bezeichnet und haben die Namenserweiterung:

- **.exe** (bei .NET – Anwendungen) oder
- **.dll** (bei .NET – Klassenbibliotheken)

Man übergibt dem Compiler im Allgemeinen *mehrere* Quellcodedateien mit jeweils einer Klassendefinition (siehe Beispiel in Abschnitt 1.2.3) 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 2003, S. 6) fasst wesentliche Informationen über Quellcode, C#-Compiler, MSIL-Code und Assemblies anhand des Bruchadditionsbeispiels zusammen und zeigt mit den anschließend zu beschriebenen Metadaten weitere wichtige Bestandteile eines .NET - Assemblies:

[1] Siehe z.B. http://msdn.microsoft.com/de-de/library/12a7a7h3.aspx



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

1.2.5.1 *Typ-Metadaten*

Neben dem MSIL-Code enthält ein Assembly **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 Methoden und Merkmale vorhanden.

Im Bruchadditions-Assembly sind z.B. über die `set_Zaehler` - Methode zur `Zaehler`-Eigenschaft der Klasse `Bruch` folgende Metadaten verfügbar: [1]

```
MetaInfo
Suchen   Weitersuchen
      Method #2 (06000002)
      -------------------------------------------------------
            MethodName: set_Zaehler (06000002)
            Flags     : [Public] [HideBySig] [ReuseSlot] [SpecialName]  (00000886)
            RVA       : 0x00002068
            ImplFlags : [IL] [Managed]  (00000000)
            CallCnvntn: [DEFAULT]
            hasThis
            ReturnType: Void
            1 Arguments
                  Argument #1:  I4
            1 Parameters
                  (1) ParamToken : (08000001) Name : value flags: [none] (00000000)
```

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

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.

1.2.5.2 Das Manifest eines Assemblies

Außerdem gehört zu einem Assembly das so genannte **Manifest** mit den **Assembly-Metadaten**. Dazu gehören:

- Name und Version des Assemblies
  Durch eine systematische Versionsverwaltung 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*) abgelegt werden sollen (siehe Abschnitt 1.2.5.4).
- Informationen über die Abhängigkeit von anderen Assemblies (z.B. aus der FCL)
  Auch hier sorgen exakte Versionsangaben für die Vermeidung von Versionsunverträglichkeiten.

Besteht ein Assembly aus *mehreren* Dateien (siehe unten), dann ist das Manifest nur in *einer* Datei vorhanden und enthält die Namen der weiteren zum Assembly gehörenden Dateien. In diesem Fall kann das Manifest auch in einer eigenen Datei untergebracht werden.

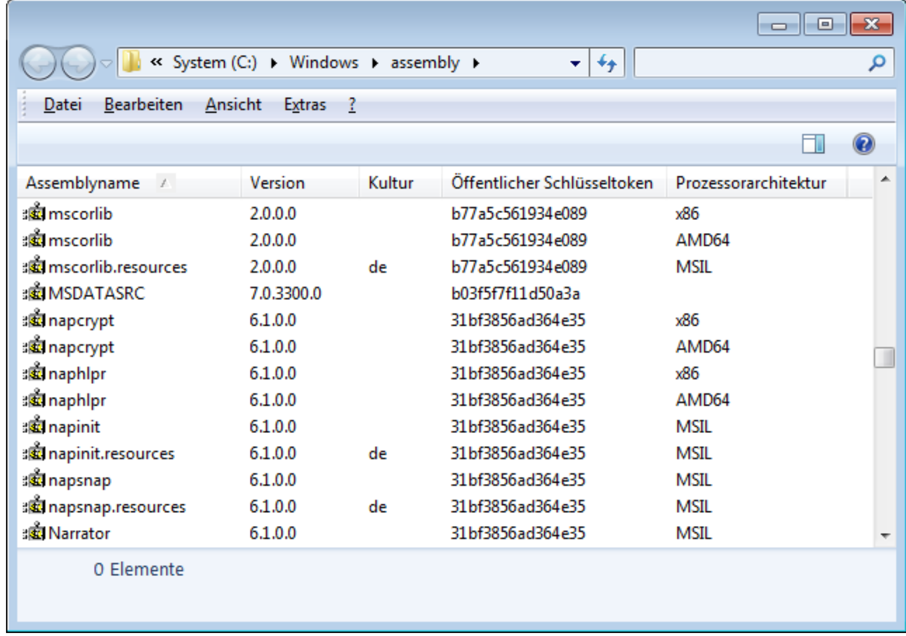
1.2.5.3 Multidatei-Assemblies

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.

1.2.5.4 Private und allgemeine Assemblies

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 (später zu behandelnden) Anwendungskonfigurationsdatei auch andere Ordner verwendet werden.

Für systemweit verfügbare Assemblies haben die .NET - Architekten den **Global Assembly Cache** (GAC) vorgesehen, der sich unterhalb des Windows-Ordners befindet, z.B.:



Assemblyname	Version	Kultur	Öffentlicher Schlüsseltoken	Prozessorarchitektur
mscorlib	2.0.0.0		b77a5c561934e089	x86
mscorlib	2.0.0.0		b77a5c561934e089	AMD64
mscorlib.resources	2.0.0.0	de	b77a5c561934e089	MSIL
MSDATASRC	7.0.3300.0		b03f5f7f11d50a3a	
napcrypt	6.1.0.0		31bf3856ad364e35	x86
napcrypt	6.1.0.0		31bf3856ad364e35	AMD64
naphlpr	6.1.0.0		31bf3856ad364e35	x86
naphlpr	6.1.0.0		31bf3856ad364e35	AMD64
napinit	6.1.0.0		31bf3856ad364e35	MSIL
napinit.resources	6.1.0.0	de	31bf3856ad364e35	MSIL
napsnap	6.1.0.0		31bf3856ad364e35	MSIL
napsnap.resources	6.1.0.0	de	31bf3856ad364e35	MSIL
Narrator	6.1.0.0		31bf3856ad364e35	MSIL

Eine Besonderheit bei GAC-Assemblies ist die erforderliche Signatur (siehe Spalte **Öffentlicher Schlüsseltoken**), die für eine global eindeutige Identifikation sorgt, so dass man von der Verpflichtung zu *starken Namen* spricht.

Aufgrund der obigen Ausführungen über den Plattform- und CPU-unabhängigen MSIL-Code in .NET – Assemblies wundern Sie sich vermutlich über die Spalte **Prozessorarchitektur**, die im Ordnerfenster zum GAC zu sehen ist und neben MSIL noch andere Einträge enthält. Offenbar sind manche Assemblies von einer Prozessor-Architektur (z.B. x86) abhängig, was im folgenden Abschnitt näher beleuchtet werden soll.

1.2.5.5 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 Software erstellt wurde. Ein besonders wichtiges Beispiel ist das Bibliotheks-Assembly **mscorlib.dll**, das elementare und oft benötigte FCL-Klassen implementiert und dabei Dienste des Betriebssystems nutzt. Auf einem Rechner mit 64-bittiger Windowsinstallation werden daher zwei Versionen dieses Assemblies benötigt (für die Plattformen x86 und x64, siehe Bildschirmphoto im letzten Abschnitt). Den Rest dieses Abschnitts sollte nur lesen, wer sich (jetzt schon) für weitere Details beim Einsatz von .NET –Software in einer 64-Bit - Umgebung interessiert.

Plattformabhängigkeiten bei Assemblies fallen derzeit besonders auf, weil sich die Windows-Welt auf dem Weg von der 32-Bit-Ära in die 64-bittige befindet. Eine 64-bittige Windows-Version enthält ein 32-Bit – Subsystem namens **WOW64** (*Windows on Windows 64*), so dass 32-Bit-Software in der Regel problemlos in der gewohnten Umgebung ablaufen kann. Dabei herrscht eine strenge Trennung, so dass 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, was in der Regel keine relevante Beschränkung darstellt. Die Missachtung dieser Regel wird mit einem Laufzeitfehler bestraft.

Mit .NET 2.0 hat Microsoft für Assemblies das Attribut **ProcessorArchitecture** mit den folgenden möglichen Werten eingeführt (in Klammern die Bezeichnung für den C# - Compiler):

- MSIL (anycpu)

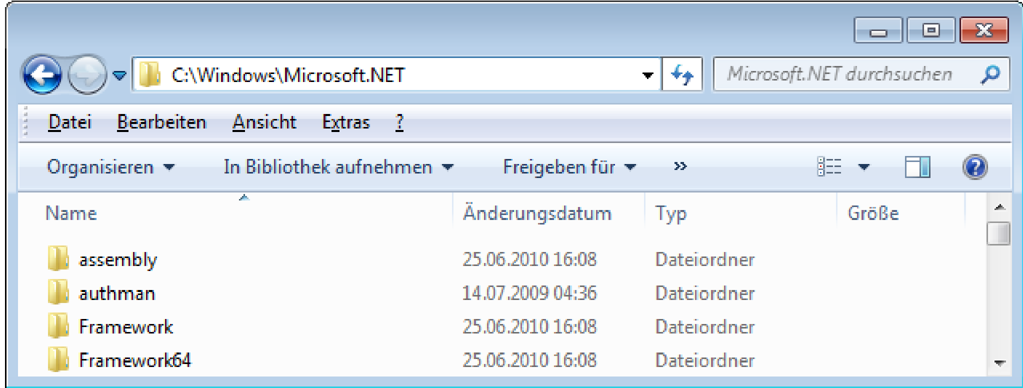
- X86 (x86)
- AMD64 (x64)
- IA64 (Itanium)

Beim Übersetzen wird das Attribut über eine Compiler-Option gesetzt, die beim C# - Compiler **platform** heißt und die Voreinstellung **MSIL** (**anycpu**) hat, wie der folgende Auszug aus der Hilfe-Ausgabe zeigt:

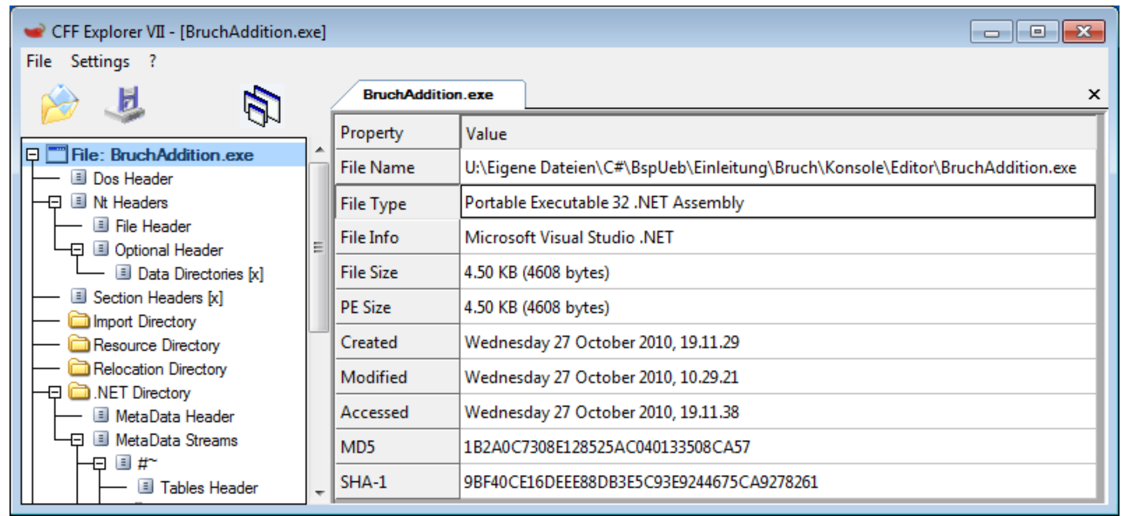
```
                    Visual C# 2010 Compiler Options

/platform:<string>           Limit which platforms this code can run on: x86,
                             Itanium, x64, or anycpu. The default is anycpu.
```

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

Damit auf einem PC mit 64-bittiger Windows-Installation auch x86 – abhängige Assemblies laufen, werden dort zwei .NET – Laufzeitumgebungen installiert (mit 32 bzw. 64 Bit), wie ein Blick in den Ordner **C:\Windows\Microsoft.NET** bestätigt:

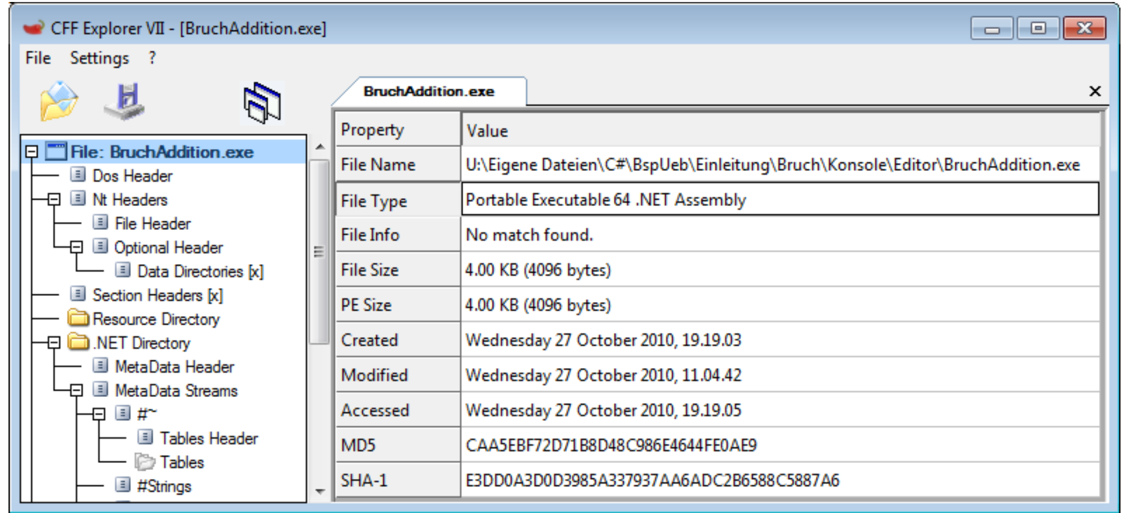


Unter einer Windows-Version mit 64 Bit produziert der C# - Compiler bei der voreingestellten Prozessorarchitektur MSIL ein **Portable Executable 32 .NET Assembly**, wie der kostenlos verfügbare **CFF-Explorer** von Daniel Pistelli zeigt:[1]



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

[1] Verfügbar über die Webseite: http://www.ntcore.com/exsuite.php



Beim Starten eines Assemblies auf einer 64-Bit – Maschine entscheidet die Laufzeitumgebung darüber, ob ein 32- oder ein 64-Bit – Prozess entsteht, wobei natürlich das Attribut **ProcessorArchitecture** eine wichtige Rolle spielt. Ein unter .NET 1.x erstelltes Assembly wird sicherheitshalber im 32-Bit – Modus ausgeführt, weil der damalige Compiler das Attribut **ProcessorArchitecture** noch nicht kannte.

In Abschnitt 2.2 wird sich zeigen, dass die im Kurs bevorzugte Entwicklungsumgebung für .NET – Programme *Visual Studio 2010* unter Windows 64 bei neuen EXE - Projekten in C# die Compiler-Voreinstellung MSIL bzgl. der Zielplattform bzw. Prozessor-Architektur *nicht* beibehält, sondern stattdessen die Zielplattform x86 einstellt. In einem Blog-Beitrag vom 9. Juni 2009 nennt der Microsoft-Mitarbeiter Rick Byers aus dem Visual Studio – Entwicklungsteam u.a. folgende Argumente:[1]

- Soll eine Anwendung sowohl in einem 32-Bit –Prozess als auch in einem 64-Bit – Prozess ausführbar sein, ist ein doppelter Testaufwand erforderlich, weil z.B. in der 32-bittigen und der 64-bittigen Laufzeitumgebung (CLR) unterschiedliche Fehler mit Relevanz für die Anwendung stecken könnten.
- Nur die wenigsten Anwendungen benötigen tatsächlich mehr Arbeitsspeicher als die unter Windows 32 angebotenen 2 GB. Ferner hat eine 64-Bit – Anwendung keine Geschwindigkeitsvorteile gegenüber ihrem 32-bittigen Gegenstück.

Bei DLL-Projekten verwendet Visual Studio 2010 hingegen die Voreinstellung MSIL (anycpu), damit die resultierenden Bibliotheks-Assemblies in einen 32-Bit – und in einer 64-Bit – Prozess geladen werden können.

1.2.5.6 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 soll, die in den 90er Jahren des letzten Jahrhunderts geschaffen wurde, um sprachübergreifende Interoperabilität von Software-Komponenten zu ermöglichen. Hier werden ebenfalls Metadaten bereitgestellt, welche alle 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 leicht zu Problemen kommen, weil die Metadaten zu einem COM-Server nicht auffindbar oder fehlerhaft sind. Die Metadaten eines .NET – Assemblies

[1] http://blogs.msdn.com/b/rmbyers/archive/2009/06/08/anycpu-exes-are-usually-more-trouble-then-they-re-worth.aspx

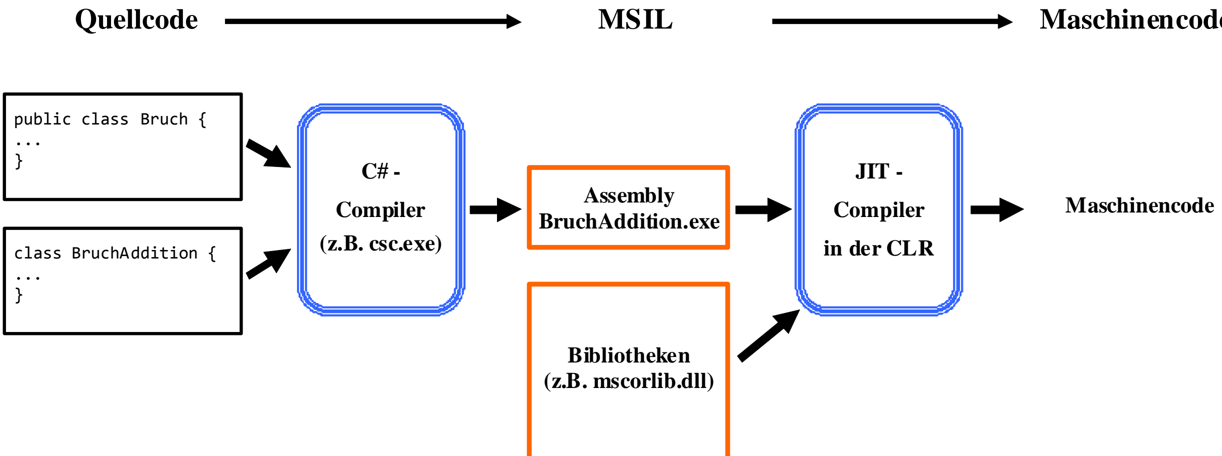
sind demgegenüber stets vorhanden und aktuell. Sie bieten außerdem wichtige Informationen (z.B. Version, Sprache, Abhängigkeiten), die in einer COM-Typbibliothek nur spärlich oder gar nicht vorhanden sind. Außerdem ist bei .NET – Metadaten kein Registry-Eintrag erforderlich.

1.2.6 CLR und JIT-Compiler

Bislang haben Sie erfahren, dass aus dem C# - Quellcode durch einen Compiler (z.B. **csc.exe**) ein Assembly mit Zwischencode (MSIL) 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 (***Just In Time***), der Leistungseinbußen aufgrund der zweistufigen Übersetzungsprozedur minimiert (z.B. durch das Speichern von mehrfach benötigtem Maschinencode).

.NET – Programme können auf jedem Windows-Rechner mit passendem Framework auf übliche Weise gestartet werden, z.B. per Doppelklick auf den Namen der Programmdatei.

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 ausnutzen kann als ein Maschinencode-Compiler, der ...

- älter ist als viele moderne 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** kann aus einem Assembly Maschinencode erzeugen und abspeichern, so dass bei der späteren Programmausführung kein JIT-Compiler mehr benötigt wird. Aus verschiedenen Gründen (siehe Richter 2006, S. 43ff) führt dies aber *nicht* unbedingt zu einer beschleunigten Programmausführung.

Die CLR hat bei der Verwaltung von .NET – Anwendungen neben der Übersetzungstätigkeit 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 später 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 orientiert an Merkmalen des jeweiligen Assemblies 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 .NET 4.0 haben Assemblies dieselben Rechte wie native Windows-Programme. Beschränkungen nach dem Sandkasten-Prinzip gelten jedoch für *hosted code*, der via Internet auf einen Rechner gelangt.[1]

1.2.7 Namensräume und FCL

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

Statt von der *Framework Class Library* wird oft von der ***B**ase **C**lass **L**ibrary* (BCL) gesprochen, gelegentlich synonym, oft mit einem Vorschlag zur Definition von zwei unterschiedlich breiten Bibliotheken. Wir plagen uns nicht mit Begriffsstreitereien oder überflüssigen Differenzierungen, sondern verwenden die Bezeichnung FCL für die Bibliothek mit dem in jeder .NET - Installation enthaltenen Klassen.

Die Klassen und sonstigen Datentypen der .NET – Standardbibliothek sind nach funktionaler Verwandtschaft in so genannte **Namensräume** eingeteilt. Dieses Organisationsprinzip dient in erster Linie dazu, Namenskollisionen in einem globalen Namensraum zu vermeiden. So dürfen z.B. zwei Klassen denselben Namen tragen, sofern sie sich in verschiedenen Namensräumen befinden. Ihre **voll qualifizierten Namen** sind dann verschieden.

Namensräume sind keinesfalls auf die FCL beschränkt, und die von C# - Entwicklungsumgebungen angebotenen Vorlagen für neue Projekte (siehe unten) definieren meist einen eigenen Namensraum (**namespace**) für jedes Projekt, z.B.:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

[1] Siehe http://msdn.microsoft.com/en-us/magazine/ee677170.aspx

```
namespace BruchAddition
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Bei kleinen Beispielprogrammen sind Namensräume jedoch überflüssig, und wir werden meist der Übersichtlichkeit halber darauf verzichten.

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

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

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

Durch diese **using**-Direktive wird dafür gesorgt, dass der Compiler dem Namen jeder Klasse, die nicht im projekteigenen Quellcode definiert ist, das Präfix **System** voranstellt und dann die referenzierten Assemblies (siehe unten) nach dem vervollständigten Namen durchsucht.

Bei Namenskollisionen gewinnt generell der lokalste Bezeichner. Im folgenden Beispiel werden der Namensraumbezeichner **System** und der Klassenname **Console** (im Namensraum **System**) durch lokale Bezeichner 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. 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 FCL sicher vermeiden. Wenn komplexe Softwaresysteme unter Beteiligung vieler Programmierer entstehen, sind Namenskollisionen aber nicht auszuschließen. Der von Entwicklungsumgebungen automatisch erstellte Quellcode (siehe unten) 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.

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

System.Windows.Media.Imaging

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

Einen ersten Eindruck vom Leistungsvermögen der FCL 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.Seccurity	... 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 (z.B. Befehlschalter, 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 in Abschnitt 1.1.2 geschilderten Vorteile der objektorientierten Programmierung wesentlich an der Funktionalität eines Programms beteiligt ist.

1.2.8 Zusammenfassung zu Abschnitt 1.2

Als Vorteile der .NET – Technologie für die Softwareentwicklung 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 1.2.4).
- **Portabilität**
  Es ist möglich, die .NET-Plattform auf andere Betriebssysteme zu portieren. Das von der Firma Novell unterstützte und von der Firma Microsoft mit Wohlwollen begleitete Open Source - Projekt **Mono** ist auf diesem Weg schon weit voran gekommen.

  Grundsätzlich enthalten ausführbare Programme und Klassenbibliotheken den portablen MSIL-Code (*Microsoft Intermediate Language*). Allerdings hat der Zwang zur Integration in historisch gewachsene Software-Landschaften dazu geführt, dass manche Assemblies von einer bestimmten Prozessor-Architektur (z.B. x86, x64) abhängig sind (vgl. Abschnitt 1.2.5.5).
- **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 Abschnitt 1.2 u.a. folgende Begriffe kennen gelernt:

- **MSIL**
  .NET – Compiler (z.B. **csc.exe** bei C#) übersetzen den Quellcode in die *Microsoft Intermediate Language*.
- **Common Language Specification (CLS)**
  Den von allen .NET – Programmiersprachen zu erfüllenden Sprachumfang bezeichnet man *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 Quellen) entsteht ein Assembly. Dies ist kleinste Einheit von .NET – Software bei der …
  - Weitergabe
  - Versionierung
  - Zuweisung von Sicherheitsbeweisen

  Ein Assembly kann beliebig viele Klassen implementieren. In einem ausführbaren Programm (Namenserweiterung **.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 MSIL-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.
- **CLR mit JIT-Compiler**

Die Ausführungsumgebung für MSIL-Code, der auch als *managed code* bezeichnet wird, besitzt einen Just-In-Time – Compiler zur Übersetzung von MSIL-Code in nativen Maschinencode. Außerdem kümmert sich die CLR um Stabilität (Code-Verifikation), Überwachung von Code mit beschränkten Rechten (*hosted code*, via Internet bezogen) und Speicherverwaltung (Garbage Collection).

- **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.
- **FCL (Framework Class Library)**
  In der voluminösen und universellen Standardbibliothek der .NET – Plattform wird von Namensräumen reichlich Gebrauch gemacht.

1.3 Übungsaufgaben zu Kapitel 1

1) Warum steigt die Produktivität der Softwareentwicklung durch objektorientiertes Programmieren?

2) Welche von den folgenden Aussagen sind richtig?
1. .NET - Programme sind nur unter Windows einsetzbar.
2. Das .NET - Framework für Windows wurde in C# programmiert.
3. Unter den .NET – Programmiersprachen zeichnet sich C# durch eine besonders leistungsfähige Standardbibliothek aus.
4. Die Klassen in einem mit C# erstellten DLL-Assembly können auch in anderen .NET – Programmiersprachen (z.B. VB.NET) genutzt werden.

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

4) Welche Fakten sprechen dafür, dass eine .NET – Anwendung trotz zweistufiger Übersetzung und JIT-Compiler zur Laufzeit keine nennenswerten Performanzdefizite im Vergleich zu einem nativen Windows-Programm hat?

5) In welcher Beziehung stehen Assemblies und Namensräumen?

6) Was bedeuten die Abkürzungen MSIL, FCL, CLS, COM?

2 Werkzeuge zum Entwickeln von C# - Programmen

In diesem Abschnitt werden kostenlos verfügbare Werkzeuge zum Entwickeln von .NET - Applikationen 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 des (im .NET – Framework enthaltenen) Compilers **csc.exe**, 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. Im weiteren Verlauf des Kurses kommt dann aber mit Microsofts Visual Studio 2010 eine bequeme und leistungsfähige Entwicklungsumgebung zum Einsatz.

2.1 C# - Entwicklung mit Texteditor und Kommandozeilen-Compiler

2.1.1 Editieren

Grundsätzlich kann man zum Erstellen der Quellcode-Datei einen beliebigen Texteditor verwenden, z.B. das im Windows-Zubehör enthaltene Programm **Notepad** (alias **Editor**). Um das Erstellen, Compilieren 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 Allerseits!");
   }
}
```

In unserem Einleitungsbeispiel (siehe Abschnitt 1.1) 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 der 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 Abschnitten 2 und 3 werden wir solche pseudo-objektorientierten (POO-) Programme benutzen, um elementare Sprachelemente in möglichst einfacher Umgebung kennen zu lernen. Aus den letzten Ausführungen ergibt sich, dass C# zwar eine objektorientierte Programmierweise nahe legen 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 müssen. Alle Themen werden aber später noch einmal systematischer und ausführlicher behandelt:

- In der ersten Zeile wird der Namensraum **System** importiert, damit die dort angesiedelte Klasse **Console** im Programm ohne Namensraum-Präfix angesprochen werden kann (vgl. Abschnitt 1.2.7). 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 frei wählbare Klassenname[1]. Hier ist wie bei allen Bezeichnern zu beachten, dass C# streng zwischen Groß- und Kleinbuchstaben unterscheidet.
- Dem Kopf der Klassendefinition folgt der mit geschweiften Klammern eingerahmte Rumpf.

---

[1] Ein paar Restriktionen gibt es beim Klasssennamen schon. Z.B. sind die reservierten Wörter der Programmiersprache C# verboten. Nähere Informationen folgen in Abschnitt 0.

- Weil die `Hallo`-Klasse startfähig sein soll, muss sie eine Methode namens **Main()** besitzen. Diese wird beim Programmstart ausgeführt und dient bei „echten“ OOP-Programmen oft dazu, Objekte zu erzeugen.
- 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 Objekt-Kreation ausgeführt werden (vgl. Abschnitt 1.1.1). 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 insbesondere die Aufgabe, die ersten Objekte zu erzeugen (siehe unsere Klasse `BruchAddition` auf Seite 9).
    - **void**
      Im Beispiel erhält die Methode **Main()** den Typ **void**, weil sie keinen Rückgabewert liefert. Mit Rückgabewerten von Methoden werden wir uns noch gründlich beschäftigen.

---

[1] Die folgende Mega-Fußnote sollte nur lesen, wer im Hallo-Beispielprogramm (z.B. aufgrund von Erfahrungen mit anderen C# - Beschreibungen) den Modifikator **public** 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 frei zu geben. 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 2001, Louis et al. 2002, Troelsen 2002) mit dem Modifikator **public** im Kopf der **Main()**-Definition, z.B.:

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

Allerdings wird **Main()** grundsätzlich *nur* von der CLR aufgerufen, die eben *nicht* wie eine fremde Klasse eingestuft wird. Laut C# - Sprachdefinition (ECMA 2006) ist für die **Main()** – Methode nur der Modifikator **static** vorgeschrieben, und demgemäß erweist sich der Modifikator **public** in der Praxis auch als überflüssig.
In den Hallo-Beispielprogrammen einiger Autoren (z.B. Eller & Kofler 2005) ist 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 Allerseits!");
    }
}
```

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 von mir (aber z.B. auch von Drayton et al. 2003 und Mössenböck 2003) bevorzugte Variante mit dem kompletten Verzicht auf **public**-Modifikatoren für das Hallo-Beispiel und vergleichbare Programme kann folgendermaßen begründet werden:

- Sie hält sich an die ECMA-Sprachbeschreibung (siehe ECMA 2006).
- Es werden keine überflüssigen, unzureichend begründeten Forderungen aufgestellt.

- 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()**) eben eine leere.
- Dem Kopf einer Methodendefinition folgt der mit geschweiften Klammern eingerahmte Rumpf mit Variablendeklarationen und sonstigen Anweisungen.
- 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 *ohne* Parameterliste auskommt, benötigt der im Methodenrumpf enthaltene **Aufruf** der Methode **Console.WriteLine()** einen Aktualparameter, damit der gewünschte Effekt auftritt. Wir geben eine durch doppelte Anführungszeichen begrenzte Zeichenfolge an.
- Bei einem Methodenaufruf handelt 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\Hallo.cs

Im Unterschied zur Programmiersprache Java müssen in C# Klassen- und Dateiname *nicht* übereinstimmen, zwecks Übersichtlichkeit sollten sie es aber in der Regel doch tun. Bei GAC-Assemblies (vgl. Abschnitt 1.2.5.4) ist die Namensübereinstimmung vorgeschrieben.

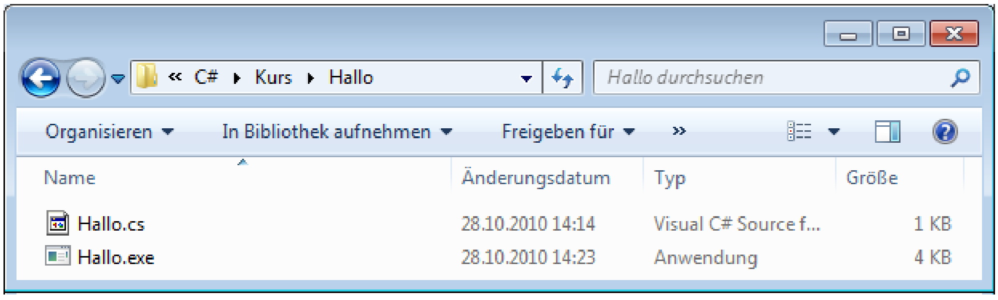
2.1.2 Übersetzen in MSIL

Öffnen Sie ein Konsolenfenster, und wechseln Sie in das Verzeichnis mit der neu erstellten Quellcodedatei **Hallo.cs**. Lassen Sie das Programm vom C#-Compiler **csc.exe** aus dem .NET Framework übersetzen, z.B. (unter Windows 7 mit 64 Bit und .NET-Framework 4.0):

```
%SystemRoot%\Microsoft.NET\Framework64\v4.0.30319\csc Hallo.cs
```

Auf Dauer ist ein derart umständliches Kommando nicht sinnvoll. Wer längerfristig per Texteditor programmieren möchte, wird den Framework-Ordner in die Definition der Umgebungsvariablen PATH aufnehmen. Beim Einsatz einer integrierten Entwicklungsumgebung (siehe Abschnitt 2.2) wird im Hintergrund derselbe Compiler verwendet, dabei jedoch automatisch mit passender Pfadangabe aufgerufen, so dass kein PATH-Eintrag erforderlich ist.

Falls keine Probleme auftreten (siehe Abschnitt 2.1.4), 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.:



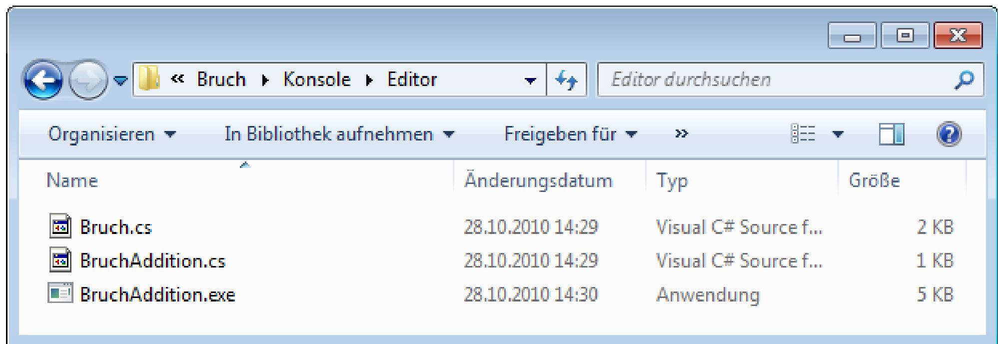
Sind *mehrere* Quellcodedateien in ein Assembly zu übersetzen, gibt man sie beim **csc**-Aufruf hintereinander an, z.B.

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

Dabei sind auch Jokerzeichen erlaubt, z.B.:

```
csc *.cs
```

Das entstehende Assembly erbt seinen Namen von der Startklasse (mit der **Main**()-Methode):



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

```
csc  /out:ba.exe  Bruch.cs  BruchAddition.cs
```

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.
  Beispiel:
  ```
  csc  /target:winexe  Bruch.cs  BruchGUI.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-Assemblys (vgl. Abschnitt 1.2.3)

Über die Befehlszeilenoption **reference** werden dem Compiler die Assemblies bekannt gemacht, welche die im zu übersetzenden Quellcode verwendeten Klassen implementieren, z.B.:[1]

```
csc  /reference:WPF\PresentationFramework.dll  Prog.cs
```

Zusätzlich wird generell das Bibliotheks-Assembly **mscorlib.dll** durchsucht, das elementare und oft benötigte FCL-Klassen implementiert.

Man kann den Compiler auch per **Response-Datei** mit Referenzen und sonstigen Befehlszeilenoptionen versorgen. Die Datei **csc.rsp** im Framework-Ordner (also z.B. in **%SystemRoot%\Microsoft.NET\Framework64\v4.0.30319**) ist Teil der Framework-Installation

[1] Das Assembly **PresentationFramework.dll** liegt im GAC-Unterordner **WPF**, der explizit angegeben werden muss, damit der Compiler das Assembly findet.

und wird automatisch ausgewertet, wenn keine Compiler-Option dagegen spricht (siehe unten). Beim NET-Framework 4.0 unter Windows 64 enthält sie folgende Referenzen:

```
# 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.

Die integrierten Entwicklungsumgebungen (siehe Abschnitt 2.2) rufen den C# - Compiler mit der Befehlszeilenoption

```
/noconfig
```

und verhindern damit die Auswertung der voreingestellten Antwortdatei. Folglich wird (zeitspa-rend!) nur noch das zentrale Bibliotheks-Assembly **mscorlib.dll** automatisch durchsucht. Zur Ver-waltung der in einem Projekt zusätzlich benötigten Referenzen bieten die Entwicklungsumgebun-gen bequeme Bedienelemente (siehe unten).

Die Compiler-Option **platform**, mit der die Abhängigkeit eines Assemblies von einer Prozessor-Architektur deklariert werden kann, wurde schon in Abschnitt 1.2.5.5 erwähnt. Durch Verzicht auf diese Option verwendet man die voreingestellte Platform **MSIL** (bzw. **anycpu**). Unter Windows 64

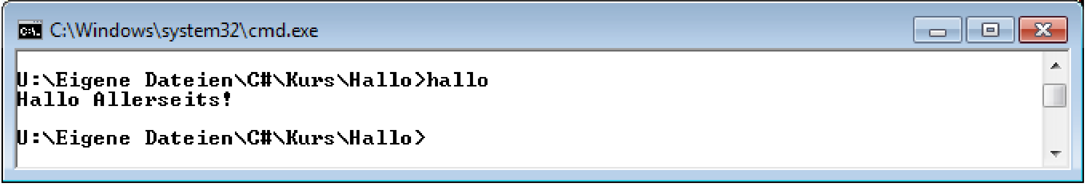
produziert der C# - Compiler daraufhin ein **Portable Executable 32 .NET Assembly**. Es wird unter Windows 64 in einem 64-Bit – Prozess ausgeführt und läuft selbstverständlich auch unter Windows 32.

Über weitere Befehlszeilenoptionen informiert der Compiler beim folgenden Aufruf

```
csc /?
```

2.1.3 Ausführen

.NET – 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.

2.1.4 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 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").

Die C# - Designer haben dafür gesorgt, dass möglichst viele Fehler vom Compiler aufgedeckt werden können (z.B. durch strenge Typisierung, Beschränkung der impliziten Typanpassung).[1]

Wir wollen am Beispiel eines provozierten Syntaxfehlers überprüfen, ob der Framework-Compiler 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 Allerseits!");
    }
}
```

meldet der Compiler:

[1] In C# 4.0 taucht mit dem Datentyp **dynamic** eine praktischen Zwängen (z.B. bei der Kooperation mit typfreien Skriptsprachen) geschuldete Ausnahme auf. An Stelle des Compilers ist hier die CLR für die Typprüfung verantwortlich, was leicht zu Laufzeitfehlern führen kann.

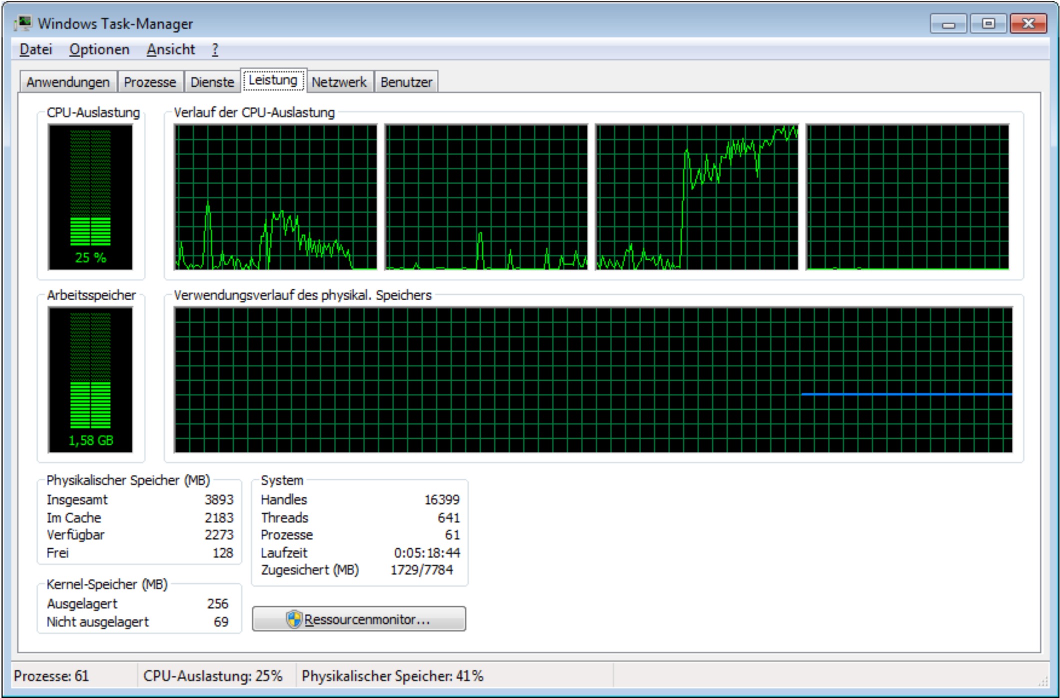
```
Hallo.cs(4,9): error CS0103: Der Name console ist im aktuellen Kontext nicht vorhan-
den.
```

Der Compiler hat die fehlerhafte Stelle sehr gut lokalisiert: Datei **Hallo.cs**, Zeile 4, Spalte 9 (vor dem kleinen *c* stehen acht Leerzeichen). 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.

Im äußerst simplen `Hallo`-Beispiel einen *semantischen* Fehler unterzubringen, den der Compiler nicht bemerkt, ist sehr schwer, vielleicht sogar unmöglich. Im Bruchadditionsbeispiel aus Abschnitt 1.1 stehen unsere Chancen weit besser, Fehler am Compiler vorbei zu schmuggeln. 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:

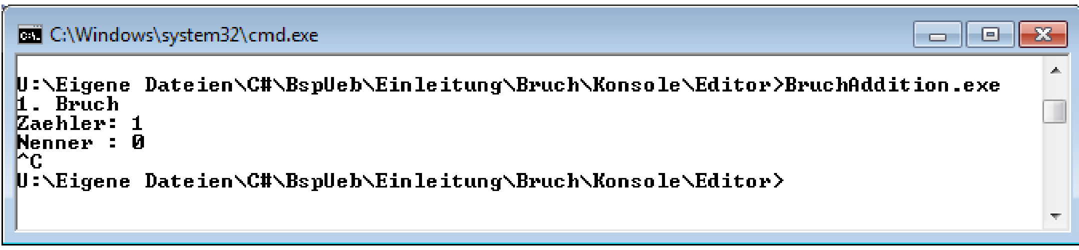
```
public int Nenner {
     get {
          return nenner;
     }
     set {
          if (value == 0)     // semantischer Fehler!
               nenner = value;
     }
}
```

Bei Eingabe kritischer „Brüche“ (wie z.B. $\frac{1}{0}$) zeigt das Programm `BruchAddition` aber ein unerwünschtes Verhalten: Die Methode `Kuerze()` (vgl. Abschnitt 1.1.2) gerät in eine Endlosschleife, und das Programm und 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 belegt einen logischer Kern, also 25% der CPU-Zeit.

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



2.2 Microsoft Visual Studio 2010

Auf die Dauer ist das Hantieren mit Notepad und Kommandozeile beim Entwickeln von .NET - Software keine ernsthafte Option. Im weiteren Verlauf des Kurses kommt mit Microsofts Visual Studio 2010 eine bequeme und leistungsfähige Entwicklungsumgebung zum Einsatz. Dieses Produkt hat seit vielen Jahren einen sehr großen Marktanteil bei der Softwareentwicklung für Windows und wird von Fremdfirmen sehr gut durch Erweiterungen für diverse Zwecke unterstützt.

Wir werden im Kurs zwei Varianten verwenden:

- **Microsoft Visual Studio 2010 Ultimate**
  Auf den Pool-PCs steht die Ultimate-Version zur Verfügung.
- **Microsoft Visual C# 2010 Express Edition**
  Dieser kostenlose Visual Studio - Ableger ist vermutlich auf Ihrem Privat-PC eine sehr gute Wahl. Im Manuskript werden die meisten Projekte mit der Express Edition erstellt.

Die Entwicklungsumgebungen sind bei der Projektverwaltung kompatibel, und bei den für uns relevanten Aufgaben gibt es auch keine wesentlichen Bedienungsunterschiede. Beide Programme bieten u.a. folgende Leistungen:

- Intelligenter Editor (z.B. mit kontextsensitiver Auflistung von Optionen zur Syntaxvervollständigung)
- Graphischer Designer für die Bedienoberfläche eines Programms
- Verschiedene Assistenten, z.B. zur Datenbankanbindung

2.2.1 Microsoft Visual Studio 2010 Ultimate

2.2.1.1 Terminal-Server – Umgebung

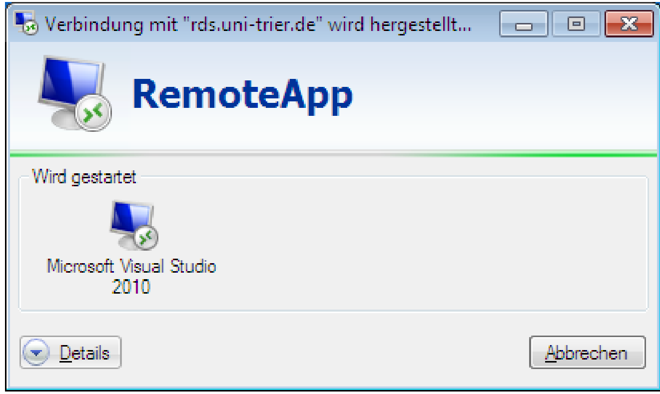
Auf den Pool-PCs an der Universität Trier wird das Visual Studio 2010 Ultimate über einen Terminalserver angeboten, wobei allerdings von der speziellen Installationsart kaum etwas zu spüren ist. Über den Link

Visual Studio 2010 Ultimate

in der Gruppe

Start > Alle Programme > Programmentwicklung > Microsoft Visual Studio

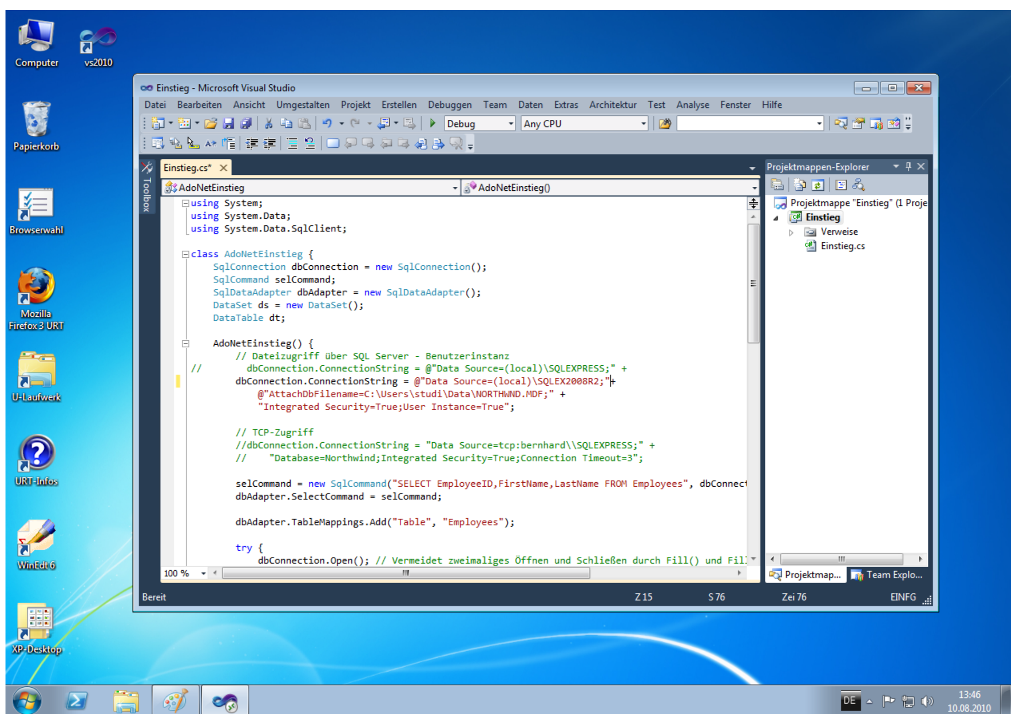
werden Sie mit dem Terminalserver verbunden,



wobei unter Windows XP (jedoch nicht unter Windows 7) die erneute Eingabe Ihres Kennworts erforderlich ist:

Wenn Sie unter Windows XP auch den Benutzernamen angeben müssen, verwenden Sie bitte den Präfix **URT** (siehe Beispiel).

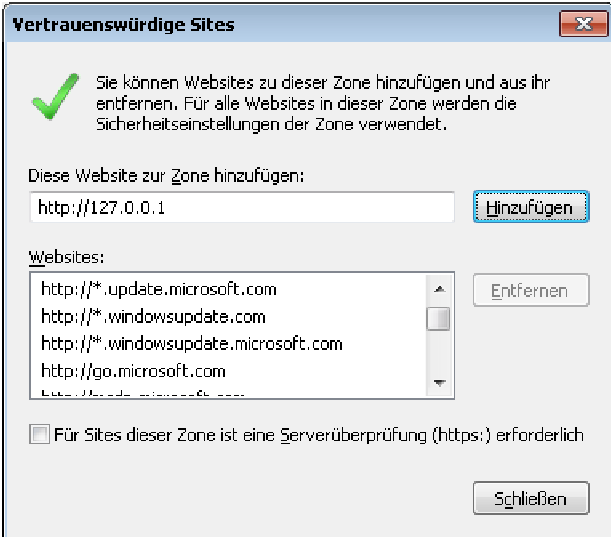
Das Visual Studio verhält sich anschließend wie ein lokales Programm und erlaubt z.B. einen Datenaustausch mit lokalen Pogrammen via Zwischenablage:



In der Regel werden Sie mit dem Visual Studio Dateien verwenden, die sich auf Ihrem Laufwerk U: befinden und für Dateiverwaltungsarbeiten wie gewohnt den Explorer der lokalen Windows-Sitzung verwenden. Sollte ein Zugriff auf das Dateisystem des Terminalservers erforderlich sein, empfiehlt sich die direkte Anmeldung beim Terminalserver (siehe unten).

Im Visual Studio 2010 werden (auch die lokal abgelegten) Hilfeinhalte per Browser angezeigt. Ist der **Internet Explorer** als Ihr Standard-Browser eingestellt, sollte der lokale Rechner in die Liste der vertrauenswürdigen Webangebote aufgenommen werden, weil ansonsten keine vollständige, ungestörte Anzeige möglich ist. Dies kann aus dem Visual Studio heraus so geschehen:

- Öffnen Sie über **Extras > Optionen > Umgebung > Webbrowser > Internet Explorer - Optionen** den Dialog mit den Internet-Optionen.
- Markieren Sie auf der Registerkarte **Sicherheit** die Zone **Vertrauenswürdige Sites**.
- Kicken Sie auf den Schalter Sites.
- Fügen Sie den Eintrag **http://127.0.0.1** hinzu, z.B.:



Gelegentlich ist eine **direkte Anmeldung beim Terminalserver** sinnvoll, weil dabei z.B. die kompletten Startmenügruppen zum Visual Studio 2010 Ultimate und zum SQL-Server 2008 R2 zur Verfügung stehen. Starten Sie dazu unter Windows 7 über

Start > Alle Programme > Zubehör > Remotedesktopverbindung

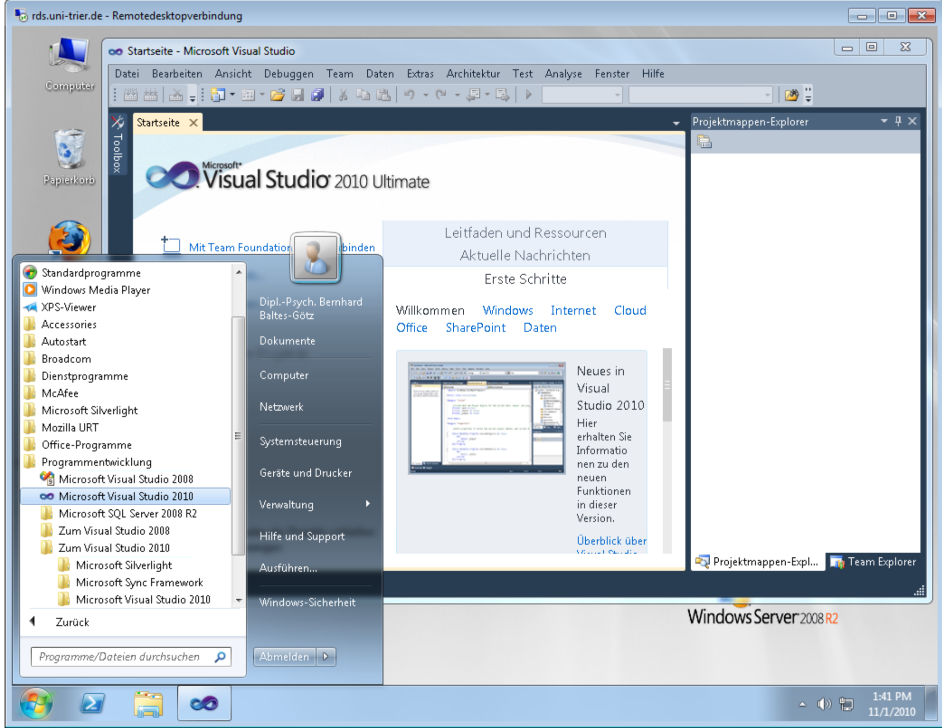
den RDP-Klienten (***R**emote **D**esktop **P**rotocol*):

Wählen Sie **rds.uni-trier.de** als **Computer**, und klicken Sie auf **Verbinden**. Unter Windows XP ist analog vorzugehen. Schlussendlich befinden Sie sich in einem Windows-Dialog mit dem Terminalserver, der das Betriebsystems *Windows Server 2008 R2* verwendet, das zur selben Gene-

ration gehört wie das Desktop-Betriebsystem *Windows 7*. Hier kann unsere Entwicklungsumgebung über den Menübefehl

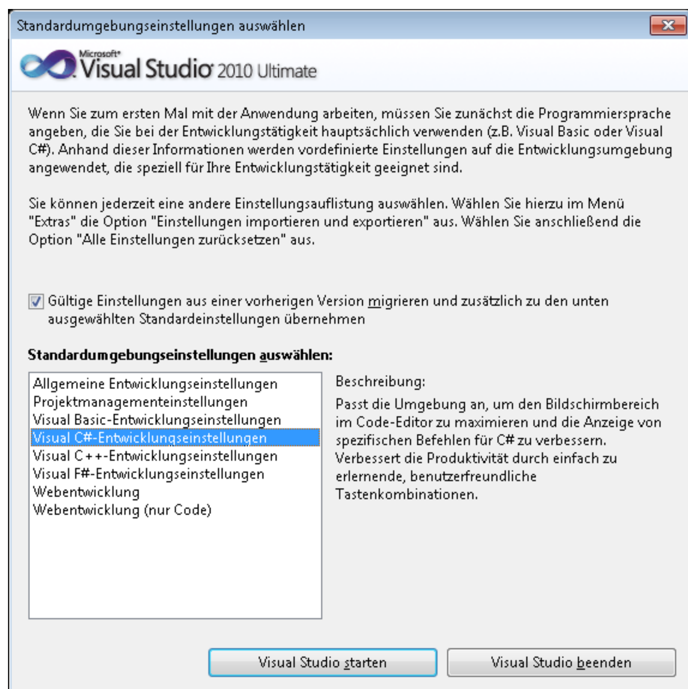
Alle Programme > Programmentwicklung > Microsoft Visual Studio 2010

gestartet werden:

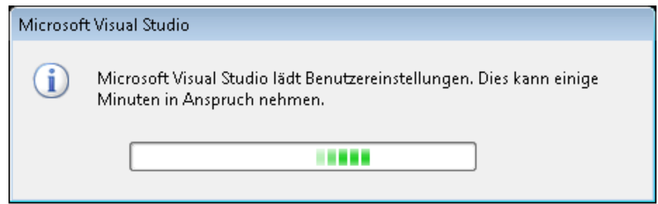


2.2.1.2 *Konfiguration beim ersten Start*

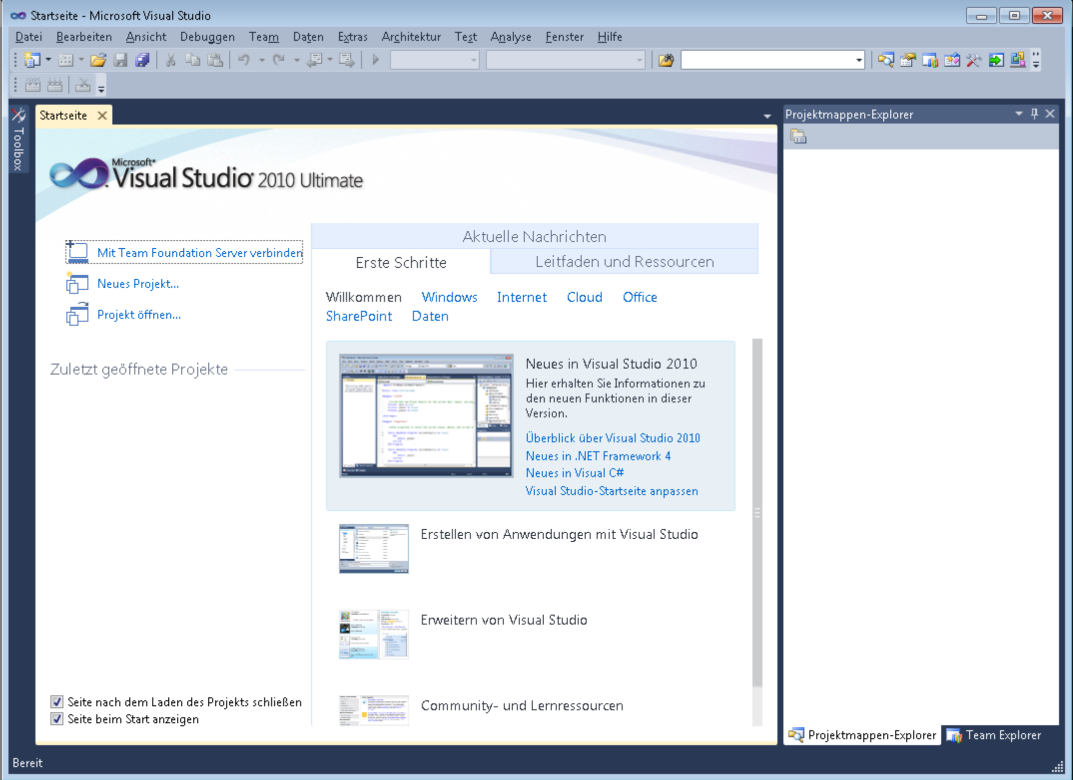
Beim ersten Start der Entwicklungsumgebung sind einige Einstellungen vorzunehmen. Wählen Sie zunächst in der folgenden Dialogbox die **Visual C# - Entwicklungseinstellungen**:



Nach einem Mausklick auf **Visual Studio starten** legt das Visual Studio etliche Ordner und Registrierungsdatenbank-Schlüssel an:



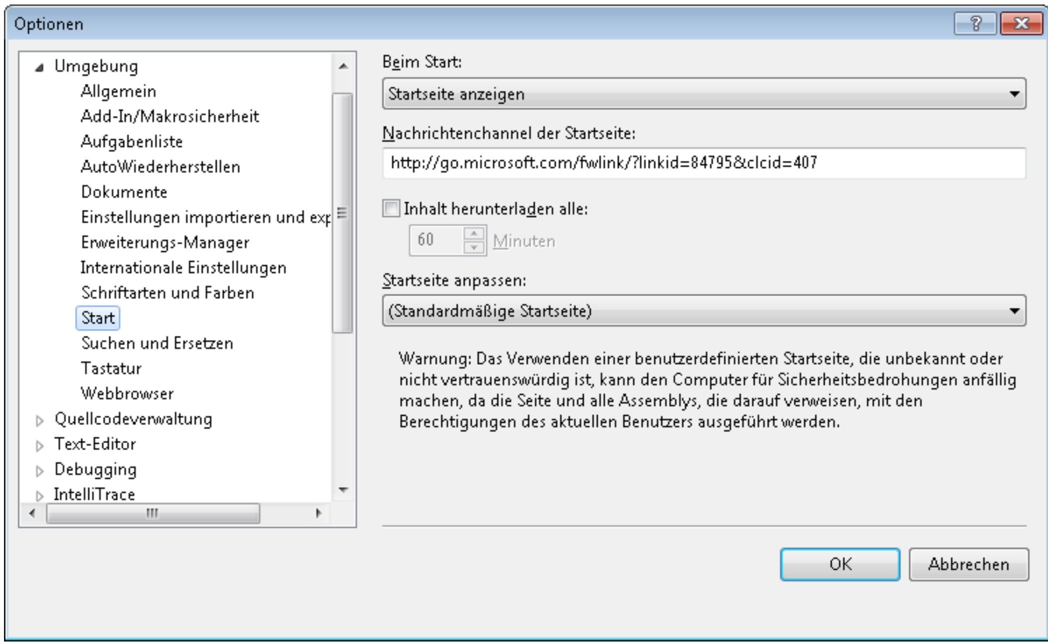
Auf der Startseite präsentiert das Visual Studio u.a. eine Liste der zuletzt geöffneten Projekte, aktuelle Nachrichten und Starthilfen:



Das Startverhalten der Entwicklungsumgebung lässt sich jederzeit nach

Extras > Optionen > Umgebung

im folgenden Dialog beeinflussen:



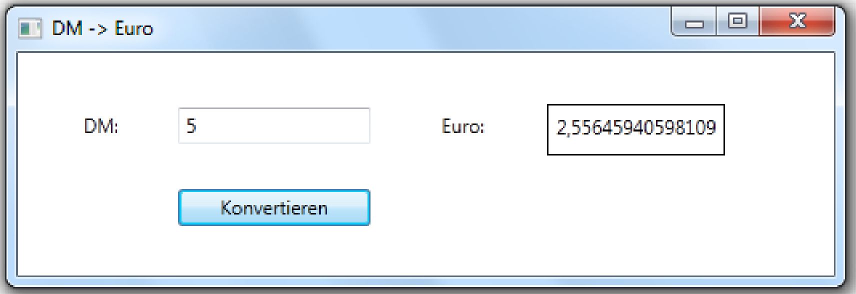
2.2.1.3 Eine erste GUI-Anwendung

Wir legen nun das erste Projekt mit dem Visual Studio an und lernen dabei neben Werkzeugen zur Erstellung von Programmen auch Bedienelemente, Ordner und Dateien zur Projektverwaltung kennen. Auf die (eher seltenen) Unterschiede zwischen den verschiedenen Versionen der Entwicklungsumgebung wird speziell hingewiesen. Die meisten Beschreibungen gelten also für alle Versionen.

Außerdem bietet dieser Abschnitt zwecks Steigerung Ihrer Motivation einen Vorausblick auf die spätere Praxis der rationellen Erstellung von Programmen mit graphischer Benutzeroberfläche (*Graphical User Interface*, GUI). Dabei werden größere Quellcode-Passagen von Assistenten der Entwicklungsumgebung erstellt, also von Programmen, die Programme schreiben. Dieser Quellcode ist zwar leicht zu erstellen, aber durch seine (im Einzelfall oft überflüssige) Komplexität schwer zu verstehen. Sobald wir das notwendige Grundwissen mit Hilfe von einfachen, komplett selbst verfassten Programmen erworben haben, spricht nichts mehr dagegen, Assistentenhilfe beim Programmieren in Anspruch zu nehmen.

2.2.1.3.1 Projekt anlegen

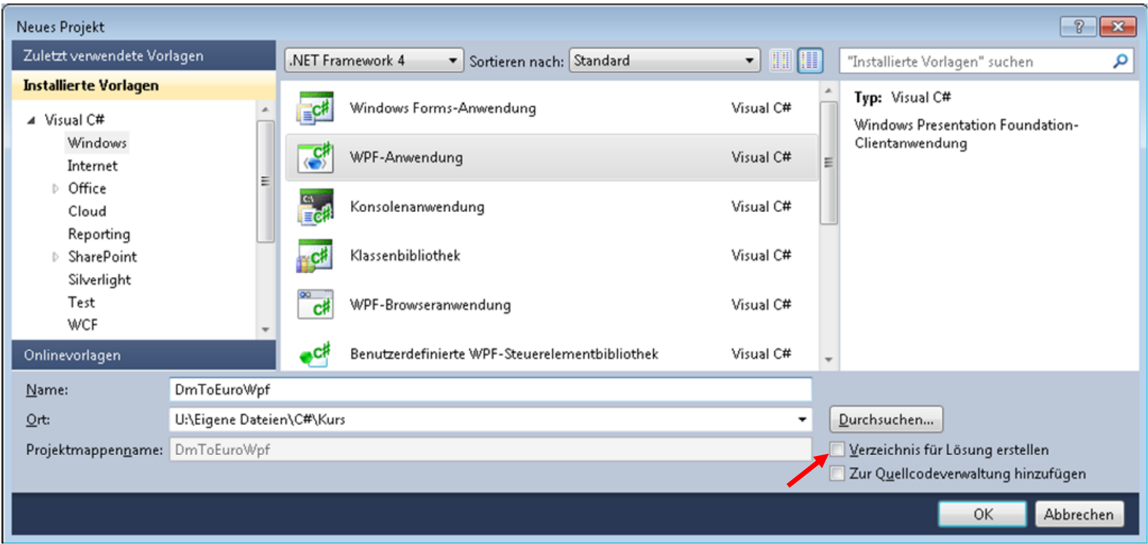
Nun machen wir uns an die Erstellung eines Währungs-Konverters mit graphischer Bedienoberfläche, der DM-Beträge in Euro-Beträge wandeln kann:[1]



Wir fordern über

Datei > Neu > Projekt

den folgenden Dialog an:



[1] Benötigt wird ein solches Programm z.B. noch bei der Auszahlung von Erbschaftsbeträgen zu Testamenten, welche in der DM-Ära verfasst wurden.

Hier wählen wir die Projektvorlage **WPF-Anwendung**, so dass unsere Anwendung die mit .NET 3.0 eingeführte GUI-Bibliothek *Windows Presentation Foundation* (WPF) verwenden wird. Dies kommt auch im Projektnamen `DmToEuroWpf` zum Ausdruck. Als **Ort** ist das übergeordnete Verzeichnis zum neu anzulegenden Projektordner anzugeben. Analog zu Abschnitt 2.1 können Sie z.B.

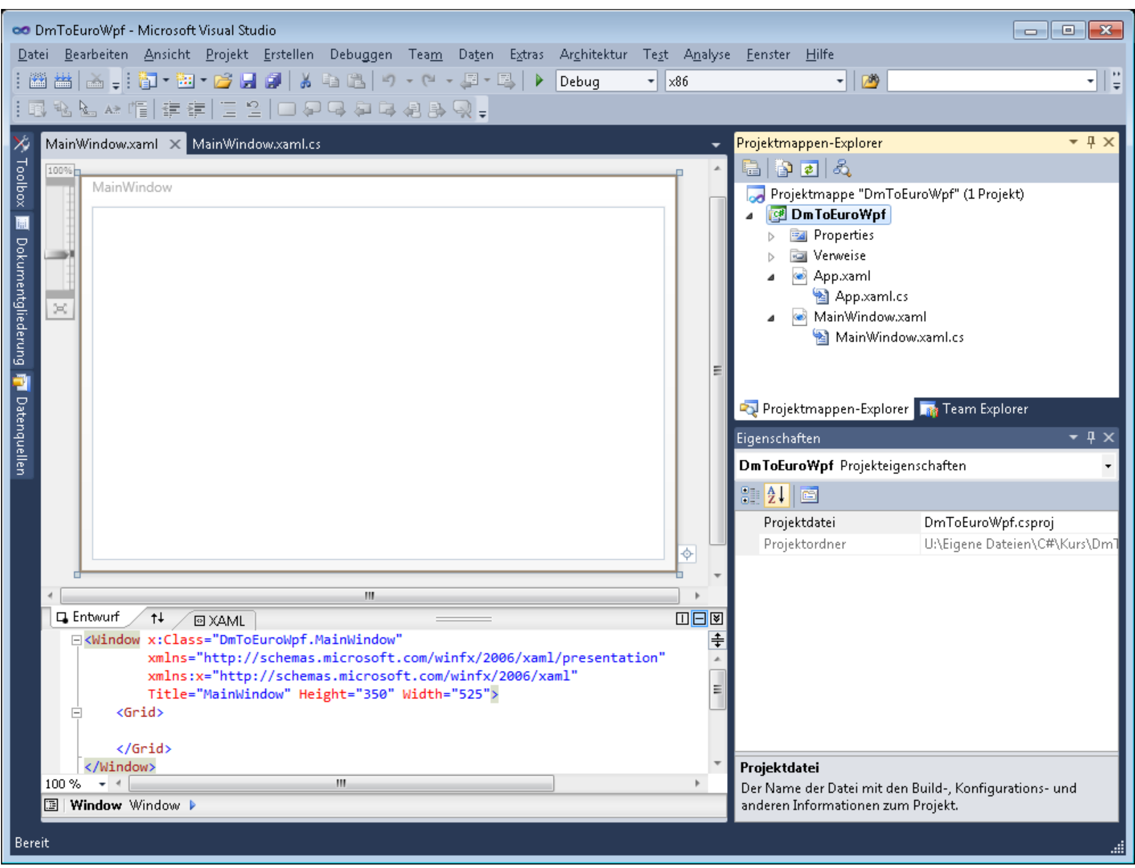
U:\Eigene Dateien\C#\Kurs

verwenden.

Jedes Projekt gehört zu einer **Projektmappe**, die eine *Familie* von zusammengehörigen Projekten (z.B. Klient- 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 entfernen wir im Dialog **Neues Projekt** die Markierung beim Kontrollkästchen **Verzeichnis für Lösung erstellen** und wählen damit eine flachere Projektdateiverwaltung.

Nach einem Mausklick auf **OK** 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 2.2.4.1) aufgelistet. Wir werden die Bestandteile nach Bedarf und bei passender Gelegenheit behandeln.

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

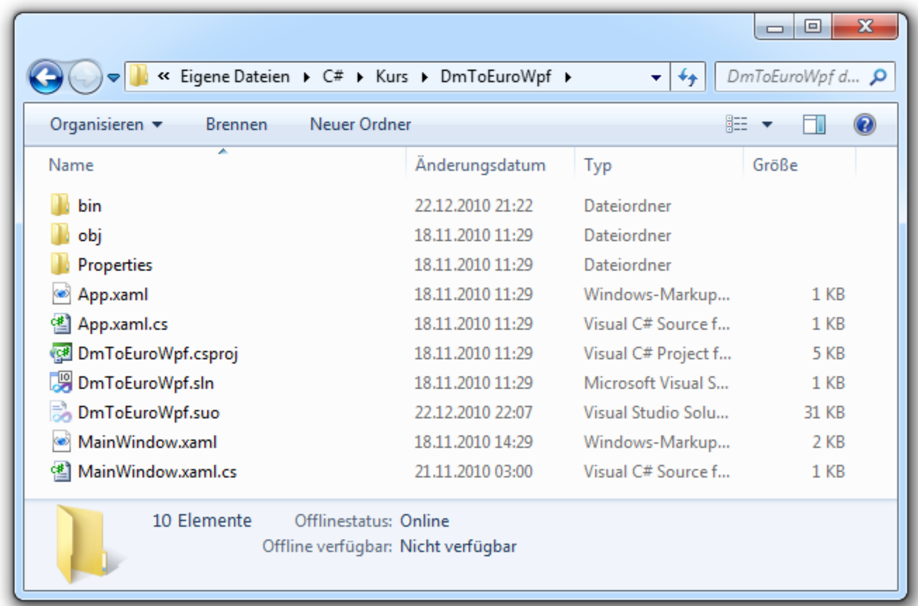


Falls Sie unter dem **Projektmappen-Explorer** kein **Eigenschaften**-Fenster sehen sollen, schalten Sie es bitte mit dem Menübefehl

Ansicht > Eigenschaftenfenster

oder mit der Funktionstaste **F4** ein. Im Augenblick sind hier die Eigenschaften des Projekts zu sehen (z.B. die Projektdatei).

Weil wir uns gegen Projektordner im Projektmappenordner entschieden haben, resultieren im Beispiel folgende Ordner und Dateien, die später nach Bedarf beschrieben werden:



Bei Verwendung der Programmiersprache C# besitzt eine Projektdatei die Namenserweiterung **csproj**, im Beispiel:

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

Bei einer Projektmappendatei wird die Namenserweiterung **sln** verwendet, im Beispiel:

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

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

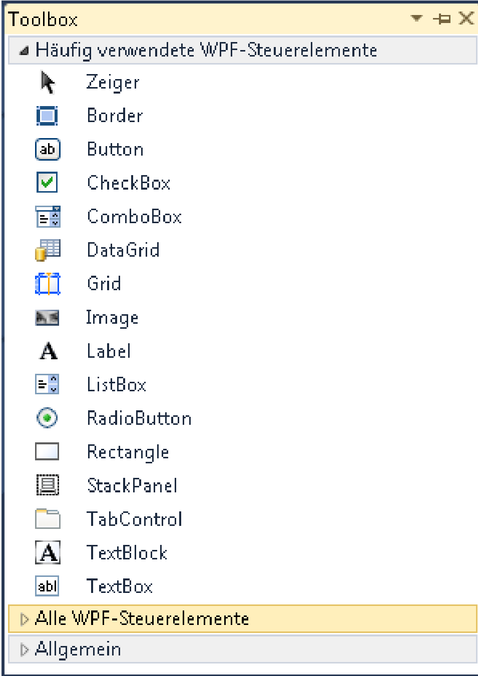
2.2.1.3.2 Bedienoberfläche entwerfen

Wir machen uns nun daran, das Hauptfenster unseres Programms mit den benötigten Bedienelementen (Steuerelementen, Controls) auszustatten. Während wir mit dem WPF-Designer fast wie mit einem Graphikprogramm arbeiten, erstellt und pflegt dieser Assistent den XAML-Code, der bei einem WPF-Projekt zur Deklaration der Benutzeroberfläche dient. Mit zunehmendem Wissen über die XAML-Beschreibungsprache werden wir später unsere Abhängigkeit vom Assistenten reduzieren. In der aktuellen Lernphase ändern wir den XAML-Code nur indirekt mit Hilfe des WPF-Designers.

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

Ansicht > Toolbox

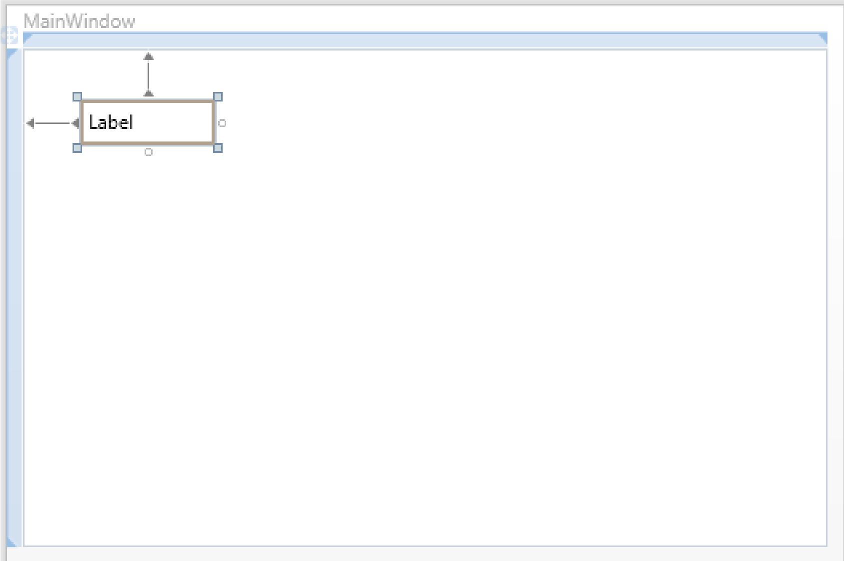
oder durch kurzes Verharren des Mauszeigers über der **Toolbox**-Schaltfläche am linken Fensterrand, und erweitern Sie nötigenfalls die Liste mit den **Häufig verwendeten WPF-Steuerelementen**:



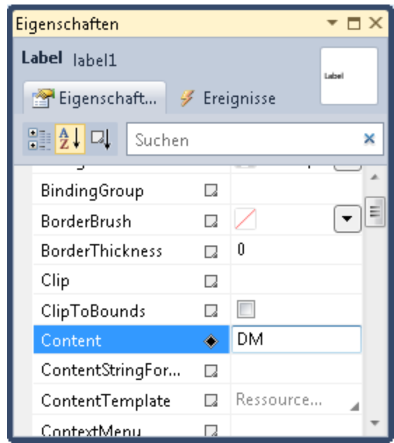
Erstellen Sie ein **Label**-Objekt (die Bezeichnung *Objekt* ist durchaus im Sinn von Abschnitt 1.1 gemeint) auf dem Formular, ...

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

Das Ergebnis sollte ungefähr so aussehen:



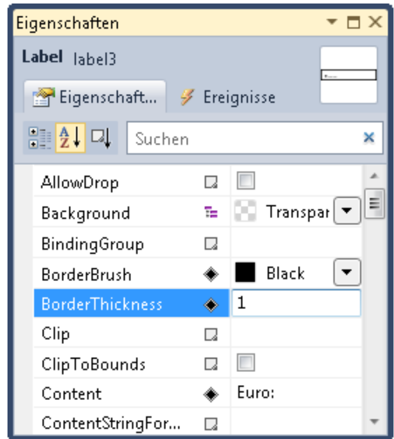
Ändern Sie die Beschriftung des **Label**-Objekts über einen passenden Wert für die Eigenschaft **Content**:



Setzen Sie ein Objekt der Klasse **TextBox** neben das **Label**-Objekt, wobei die Entwicklungsumgebung bei der Anpassung von Position und Größe mit diversen Hilfslinien unterstützt. In das **TextBox**-Steuerelement sollen die Benutzer unseres Programms den zu konvertierenden DM-Betrag eingeben:

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

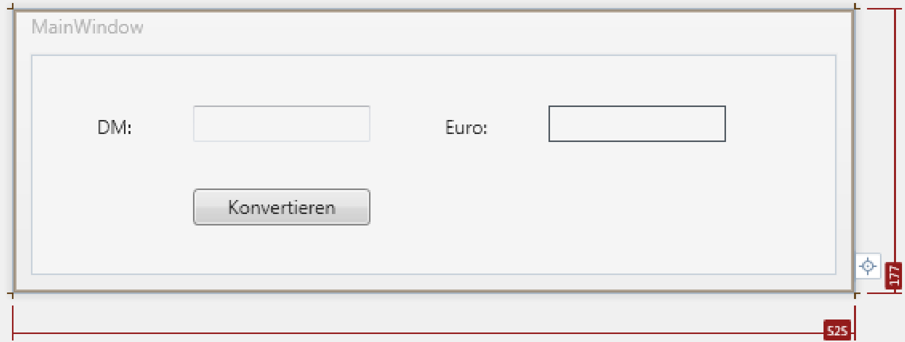
Während das erste Label wie das DM-Pendant zur Beschriftung dient, soll das zweite den Ergebnisbetrag anzeigen, also beim Programmstart leer sein. Passen sie also die **Content**-Eigenschaftsausprägungen der Objekte passend an. Um dem zweiten Label auch initial einen optischen Auftritt zu verschaffen, sollten Sie einen Rand anzeigen lassen. Dazu ist über die Eigenschaft **BorderBrush** eine Randfarbe und über die Eigenschaft **BorderThickness** eine Randstärke geeignet festzulegen, z.B.:



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

Auch beim **Button**-Objekt sorgt man über die **Content**-Eigenschaft für eine passende Beschriftung.

Mittlerweile hat sich gezeigt, dass die vorgegebene Fensterfläche in vertikaler Richtung überdimensioniert ist. Packen Sie mit der Maus den unteren Fensterrand, und wählen Sie eine neue Höhe:

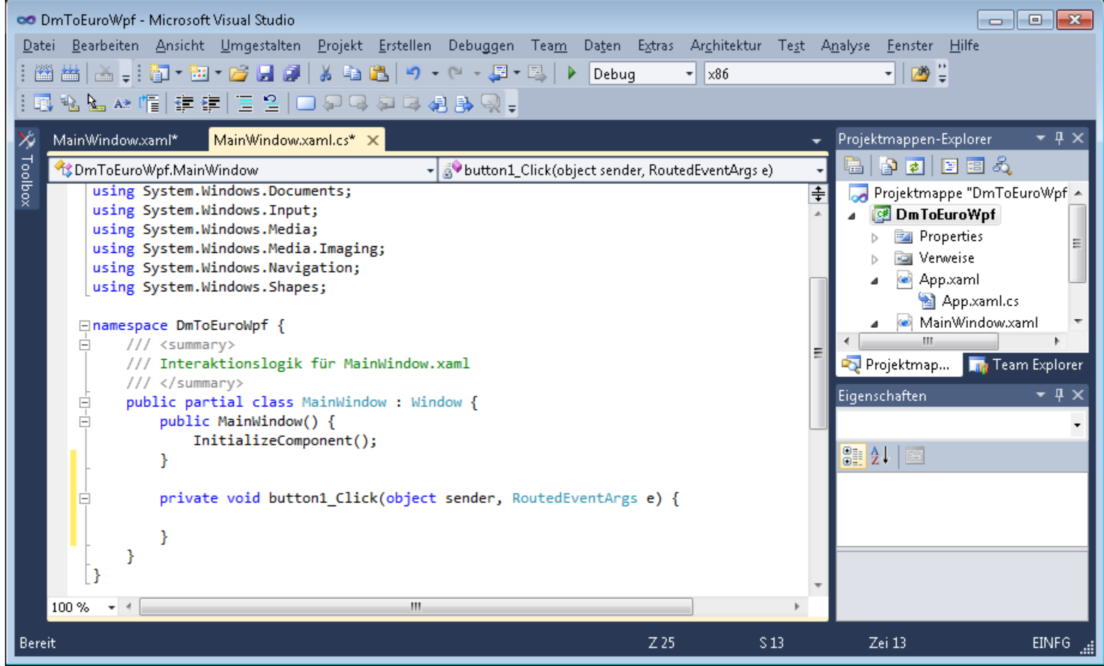


2.2.1.3.3 Behandlungsmethode zum Click-Ereignis des Befehlsschalters erstellen

Nun ist die Bedienoberfläche akzeptabel, und wir können uns um die Funktionalität des Programms kümmern. Dazu ist die Methode zu erstellen, die bei einem Mausklick auf den Befehlsschalter ausgeführt werden soll. Aufgaben der Methode:

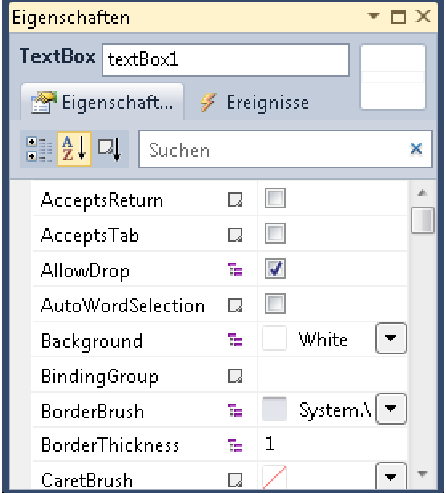
- beim **TextBox**–Objekt die Zeichenfolge erfragen
- diese Zeichenfolge nach Möglichkeit in eine Zahl wandeln
- das Ergebnis durch den DM-Euro – Umrechnungsfaktor 1,95583 dividieren
- den Euro-Betrag in eine Zeichenfolge wandeln und das umrahmte **Label**-Objekt bitten, diese Zeichenfolge anzuzeigen.

Sobald Sie einen Doppelklick auf das **Button**-Objekt setzen, öffnet das Visual Studio den Quellcode-Editor und fügt dort eine Methode namens `button1_Click()` ein, die im fertigen Programm nach jedem Mausklick auf den Schalter ausgeführt wird:

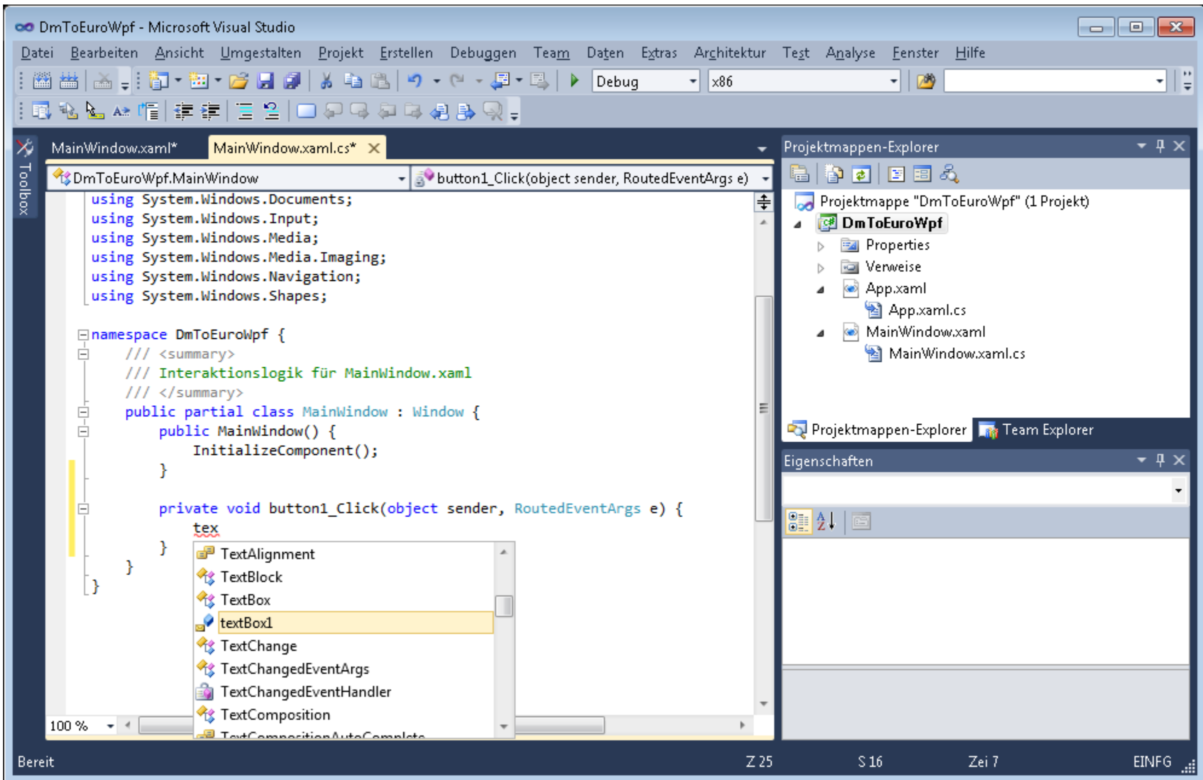


Nun wird auch erkennbar, dass wir gerade dabei sind, mit Assistentenhilfen eine Klasse namens `MainWindow` zu definieren. Für das Projekt hat die Entwicklungsumgebung den Namensraum `DmToEuroWpf` definiert, wobei die Bezeichnung mit dem Projektnamen übereinstimmt.

Das **TextBox**-Memberobjekt der Klasse `MainWindow` hat beim Fenster-Design automatisch den Namen `textBox1` erhalten, den Sie per Eigenschaften-Fenster ändern könnten:



Sobald Sie damit beginnen, sich im Methodenrumpf beim **TextBox**-Objekt über dessen **Text**-Eigenschaft nach der vom Benutzer eingetippten Zeichenfolge zu erkundigen, erahnt das Visual Studio Ihre Absicht und bietet mögliche Fortsetzungen Ihrer Anweisung an:



Akzeptieren Sie den Vorschlag `textBox1` 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 Anweisung C# - Anweisung bewährt sich wiederum die IntelliSense-Technik unserer Entwicklungsumgebung:

```
private void button1_Click(object sender, RoutedEventArgs e) {
    Convert.ToDtextBox1.Text
                ToDateTime
}               ToDecimal
                ToDouble
```

Wir müssen lediglich die runden Klammern um den Parameter ergänzen. Der Quellcode-Editor unserer Entwicklungsumgebung bietet außerdem …

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

  ```
  private void button1_Click(object sender, RoutedEventArgs e) {
      Convert.ToDouble(textBox1.Text)
  }
  ```

  Wir haben mittlerweile einen Methodenaufruf erstellt, der aber keine vollständige Anweisung ist, was unsere Entwicklungsumgebung durch rotes Unterschlängeln der mutmaßlichen Fehlerstelle reklamiert.

Um bei den weiteren Verarbeitungsschritten einen überlangen Ausdruck zu vermeiden, benutzen wir zur lokalen Zwischenspeicherung in der Methode `button1_Click()` eine lokale Variable namens `betrag` vom Typ **double** (siehe Abschnitt 3.3.4):

```
private void button1_Click(object sender, RoutedEventArgs e) {
    double betrag = Convert.ToDouble(textBox1.Text)

}
```

Am Ende der Variablendeklaration mit Wertzuweisung fehlt noch ein Semikolon, woran die Syntaxprüfung der Entwicklungsumgebung erinnert.

Die von **ToDouble**() als Rückgabe gelieferte und mittlerweile in der Variablen `betrag` gespeicherte Zahl wird durch 1,95583 dividiert. Diesen Euro-Betrag lassen wir durch die Methode **ToSring**() der Klasse **Convert** in eine Zeichenfolge (ein Objekt der Klasse **String**) wandeln:

```
private void button1_Click(object sender, RoutedEventArgs e) {
    double betrag = Convert.ToDouble(textBox1.Text);
    Convert.ToS(betrag / 1.95583);
}               ToSByte
                ToSingle
                ToString
```

Das Endergebnis muss dem **Label**-Objekt `label3` als neuer Wert der Eigenschaft **Content** übergegeben werden:

```
private void button1_Click(object sender, RoutedEventArgs e) {
    double betrag = Convert.ToDouble(textBox1.Text);
    label3.ConConvert.ToString(betrag / 1.95583);
}             Content
              ContentStringFormat
              ContentTemplate
              ContentTemplateSelector
              ContextMenu
              ContextMenuClosing
              ContextMenuOpening
              DataContext
              DataContextChanged
```

Durch ein abschließendes Semikolon hinter der zweiten Anweisung wird die Methode `button1_Click` fertig gestellt:

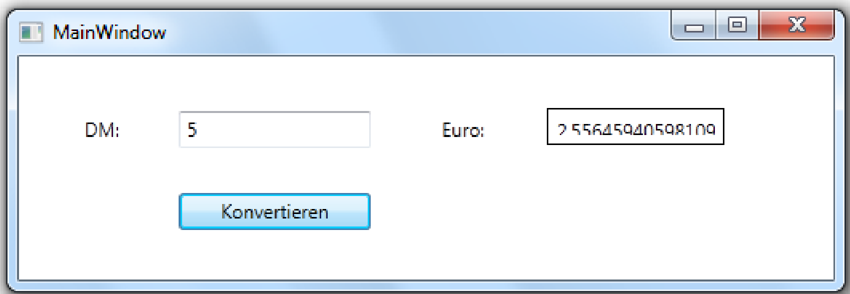
```
private void button1_Click(object sender, RoutedEventArgs e) {
   double betrag = Convert.ToDouble(textBox1.Text);
   label3.Content = Convert.ToString(betrag / 1.95583);
}
```

2.2.1.3.4 Testen und verbessern

Über den Schalter ▶, die Funktionstaste **F5** oder den Menübefehl

Debuggen > Debugging starten

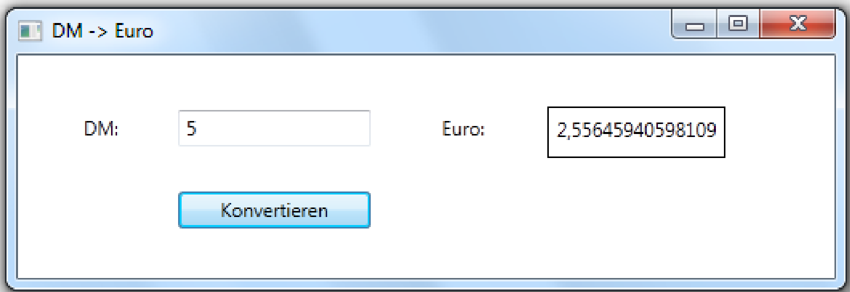
veranlasst man das Übersetzen und die Ausführung der Anwendung:



Beim ersten Einsatz es Programms zeigt sich noch Verbesserungsbedarf:

- Das **Label**-Objekt zur Ergebnisanzeige ist zu flach.
  Wechseln Sie zum Fensterdesigner per Mausklick auf sein Registerblatt, und ändern Sie die **Height**-Eigenschaft des betroffenen **Label**-Steuerelements. Dies kann per Maus geschehen oder über das Eigenschaftenfenster.
- Der Fenstertitel sollte an den Zweck des Programms erinnern.
  Markieren Sie bei aktivem Fensterdesigner das Hauptfenster und ändern Sie per Eigenschaftenfenster seine **Title**-Eigenschaft.

Nun erhalten wir ein zufriedenstellendes Ergebnis:



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

Soll eine Anwendung nur erstellt (aber nicht ausgeführt) werden, wählt man den Schalter (auf der Symbolleiste **Erstellen**), die Funktionstaste **F6** oder den Menübefehl

Erstellen > Projektmappe erstellen

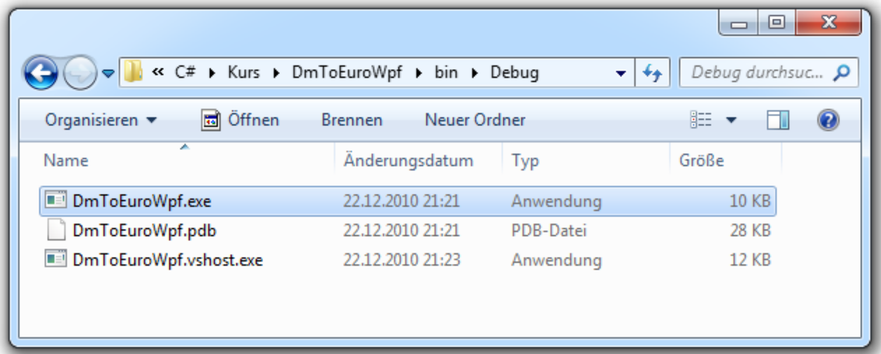
Bei der *einfachen* Erstellungskonfiguration (siehe Abschnitt 2.2.3) führen die Funktionstasten **F5** und **F6** bzw. die zugehörigen Menübefehle zu verschiedenen Übersetzungsergebnissen:

- Bei der mit **F5** angestoßenen Erstellung wird die Debug-Konfiguration verwendet, so dass der Compiler ein gut testbares, aber nicht optimiertes Assembly erzeugt und im Projekt-Unterordner **…\bin\Debug** ablegt.
- Bei der mit **F6** angestoßenen Erstellung wird die Release-Konfiguration verwendet, so dass der Compiler ein optimiertes, aber weniger gut testbares Assembly erzeugt und im Projekt-Unterordner **…\bin\Release** ablegt.

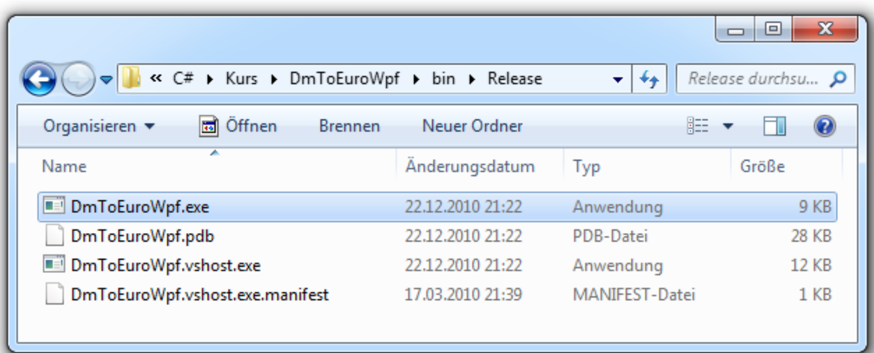
Momentan verwenden wir die Ultimate-Version der Entwicklungsumgebung, wobei ...

- die *erweiterte* Erstellungskonfiguration eingestellt
- und die Debug-Konfiguration aktiv ist.
  Zur Änderung dieser Wahl steht in der Symbolleiste **Standard** ein Bedienelement bereit:
  Debug

Das vom Compiler erzeugte Assembly **DmToEuroWpf.exe** landet also im Projekt-Unterordner **…\bin\Debug**:



Solange die Debug-Konfiguration aktiv bleibt, führt **F6** zum selben Übersetzungsergebnis wie **F5**, sodass die nach **F6** unterbleibende Ausführung den einzigen Unterschied darstellt. Bei aktiver Release-Konfiguration führen beide Erstellungsbefehle zu einem optimierten, aber weniger gut testbaren Assembly im Projekt-Unterordner **…\bin\Release**:



In jedem Fall ist das Assembly **DmToEuroWpf.exe** einsatzfähig und kann auf jedem Windows-Rechner mit .NET – Framework 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 transportiert werden.

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

2.2.2 Microsoft Visual C# 2010 Express Edition

Die Firma Microsoft stellt mit der Visual C# 2010 Express Edition eine kostenlose Einstiegsversion seiner Entwicklungsumgebung Visual Studio 2010 zur Verfügung, die für unsere Kurszwecke sehr gut geeignet ist.[1] Im Unterschied zu früheren Express-Versionen besteht die Firma Microsoft bei der Version 2010 auf einer Registrierung. Bei Bedienung und Projektverwaltung gibt es für C# - Entwickler wenig Unterschiede zu den kommerziellen Versionen, so dass beim eventuellen Umstieg keine Schwierigkeiten zu erwarten sind.

Während die kostenpflichtigen Visual Studio - Varianten *mehrere* Programmiersprachen unterstützen, gibt es für C#, VB.NET, C++ und die Webentwicklung jeweils eine eigene Express-Edition. Microsoft hat gegen die Installation von *mehreren* Express-Editionen auf *einem* Rechner nichts einzuwenden, doch wir benötigen im Kurs lediglich die C# -Variante.

2.2.2.1 Installation

Voraussetzungen:

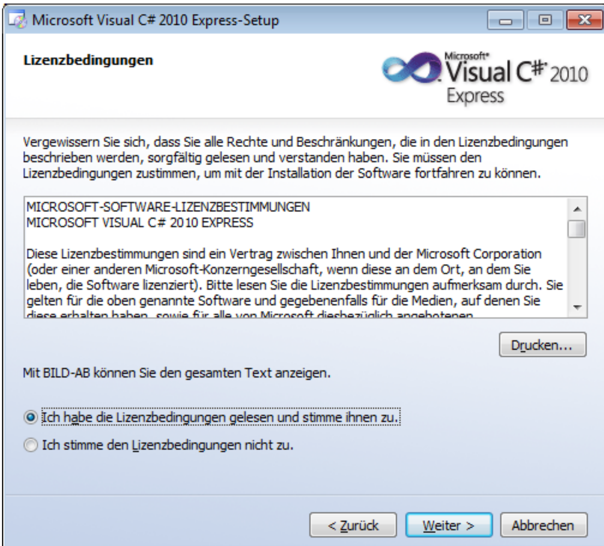
- Windows XP (x86) mit SP 3, Windows Vista mit SP 2 (x86 und x64), Windows 7 (x86 und x64), Windows Server 2003 mit SP 2 (x86 und x64), Windows Server 2003 R2 (x86 und x64), Windows Server 2008 mit SP 2 (x86 und x64), Windows Server 2008 R2 (x64)
- Prozessor mit 1,6 GHz
- 1 GB RAM
- 3 GB Festplattenspeicher

Die Installation der *Visual C# 2010 Express Edition* verläuft einfach und zuverlässig:

- Starten Sie das Installationsprogramm **setup.exe**.
- Bestätigen Sie der Windows-Benutzerkonensteuerung Ihr Vertrauen gegenüber dem Installationsprogramm.
- Entscheiden Sie selbst, ob Ihre Installationserfahrungen an Microsoft übermittelt werden sollen:

- Stimmen Sie der Lizenzvereinbarung zu:

[1] Die Visual C# 2010 Express Edition kann über Web-Adresse kostenlos bezogen werden:
http://www.microsoft.com/germany/Express/download/



- Wählen Sie die gewünschten Zusatzoptionen:

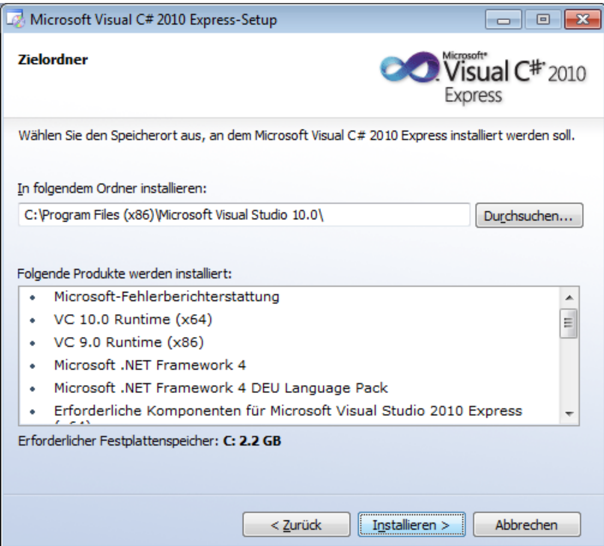
Silverlight ist Microsofts Alternative zu Adobes RAI-Technik (***R**ich **I**nternet **A**pplication*) Flash. Es arbeitet als Browser-Plugin und kann dabei .NET - Software mit WPF-Bedienoberfläche ausführen.

Die (relativ voluminöse) *SQL-Server 2008 Express Edition* benötigt man nur bei der Entwicklung von größeren Datenbankprojekten (z.B. zur Versorgung von mehreren, simultan zugreifenden Benutzern). Der für kleinere Datenbankaufgaben durchaus geeignete *SQL Server Compact 3.5* wird mit der *Visual C# 2010 Express Edition* auf jeden Fall installiert. Wir werden bei der Behandlung der Datenbankprogrammierung beide SQL - Server verwenden. Wird die *SQL Server 2008 Express Edition* (durch Wahl in obigem Dialog) gemeinsam mit der *Visual C# 2010 Express Edition* installiert, ist leider die nützliche graphische Bedienoberfläche *SQL Server Management Studio* nicht mit einbezogen. Bei einer separaten Installation des (ebenfalls kostenlos bei Microsoft verfügbaren) Pakets *SQL Server 2008 R2 Express with Tools* landet hingegen auch das Management Studio auf der Platte.

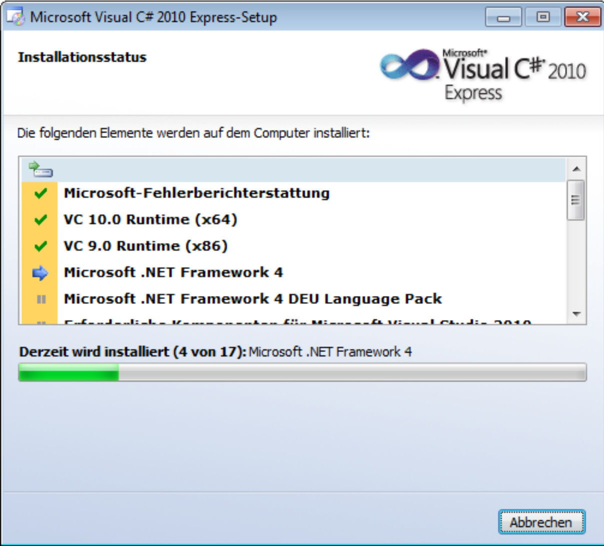
Somit wird insgesamt empfohlen, die *SQL Server 2008 Express Edition* nicht gemeinsam im der *Visual C# 2010 Express Edition* zu installieren, sondern bei Bedarf später das Paket *SQL Server 2008 R2 Express with Tools* zu installieren.

Mit der *Visual C# 2010 Express Edition* werden generell auch das .NET -Framework 4.0 sowie das Windows-SDK 7.0A (mit dem sehr nützlichen Hilfsprogramm **ILDasm** installiert.

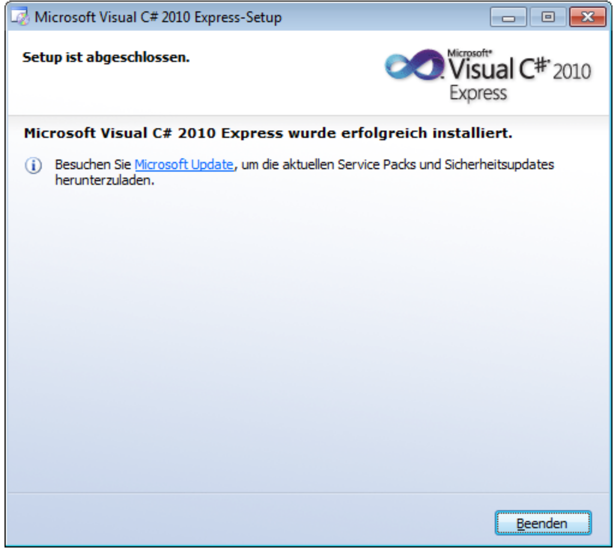
- Eventuell wollen Sie den Zielordner ändern:



- Nach einem Mausklick auf **Installieren** geht es los:



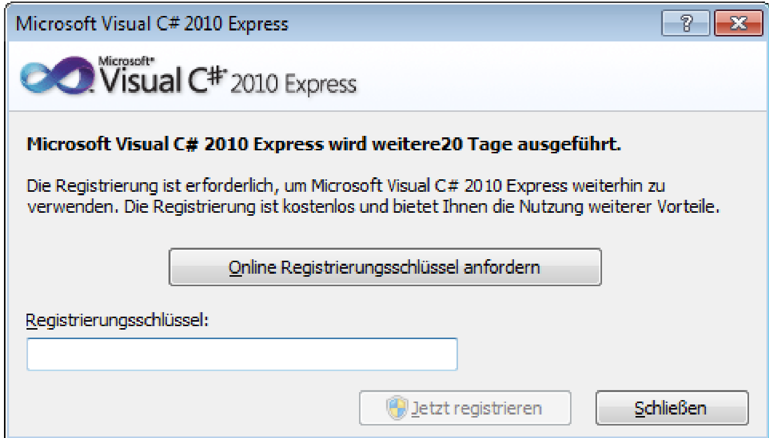
- Bis zur Fertigstellung vergeht einige Zeit:



Auf meinem PC mit der Intel-CPU Core i3 550 dauerte es 7 Minuten.

2.2.2.2 Registrierung

Abweichend von früheren Express-Versionen seiner Entwicklungsumgebung verlangt Microsoft bei der Generation 2010 eine (allerdings kostenlose) Registrierung. In den ersten Tagen nach der Installation bleibt man noch von dieser Pflicht verschont, danach erinnert Visual C# 2010 beim Starten gelegentlich an die ausstehende Registrierung:

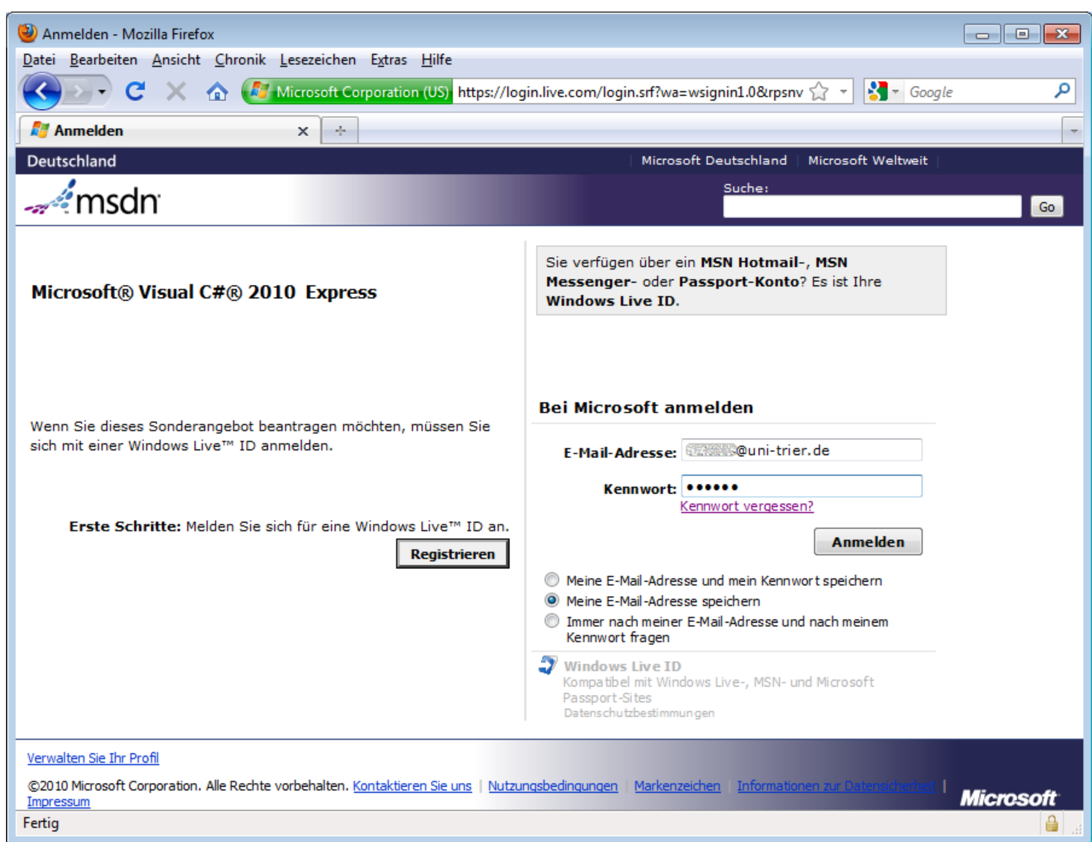


Dieser Dialog lässt sich auch über den folgenden Menübefehl hervorlocken:

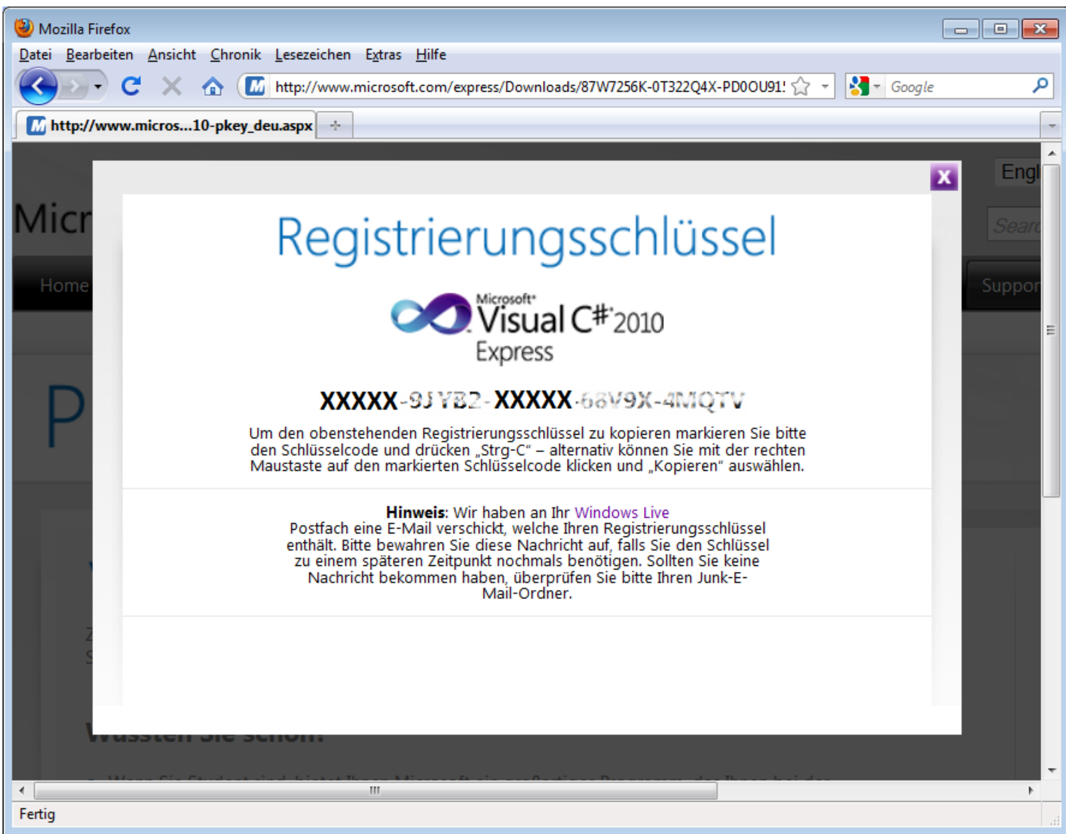
Hilfe > Produkt registrieren

Durch **Schließen** des Dialogs kann man die restliche registrierungsfreie Zeit nutzen und Visual C# 2010 Express starten.

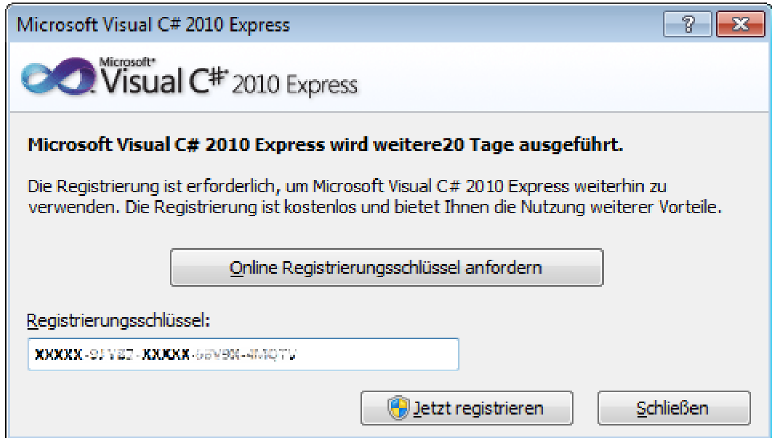
Über den Schalter **Online Registrierungsschlüssel anfordern** gelangt man auf die folgende Webseite, die zum Registrieren der Entwicklungsumgebung eine **Windows Life ID** verlangt, die nötigenfalls zunächst über den Schalter **Registrieren** erwerben werden muss:



Nach dem **Anmelden** wird im nächsten Dialog nach persönlichen Daten und Interessen gefragt, und als Gegenleistung für die bereitwilligen Angaben erhalten Sie schließlich den Registrierungsschlüssel:



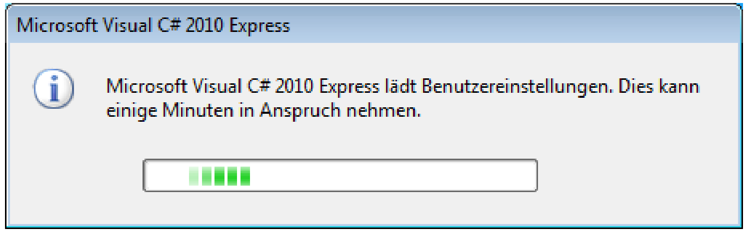
Tragen Sie den Schlüssel in den obigen Registrierungsdialog ein, und quittieren Sie mit **Jetzt registrieren**:



Je nach Betriebssystem und Benutzerrechten müssen Sie noch gegenüber der UAC (*User Account Control*) Ihr Einverständnis erklären und sind endlich fertig.

2.2.2.3 *Ein erstes Konsolen-Projekt*

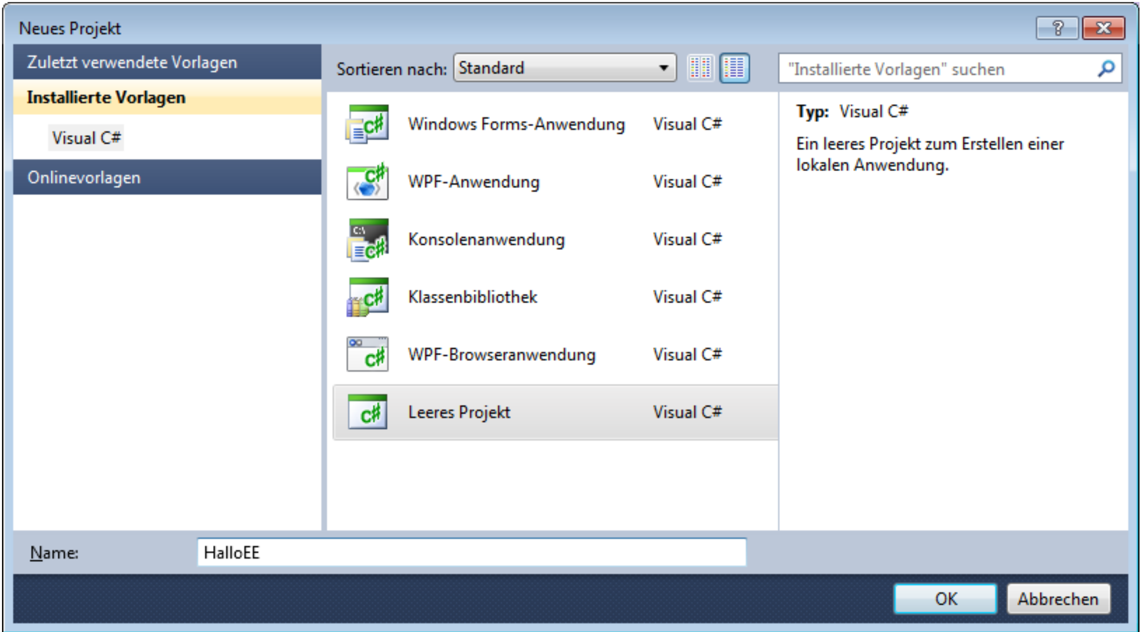
Beim ersten Start benötigt Visual C# 2010 Express einige Zeit zum Aufbau seiner Arbeitsumgebung



Von der Startseite ausgehend, die bei allen Visual Studio 2010 - Versionen praktisch identisch ist, öffnen wir mit dem Schalter oder dem Menübefehl

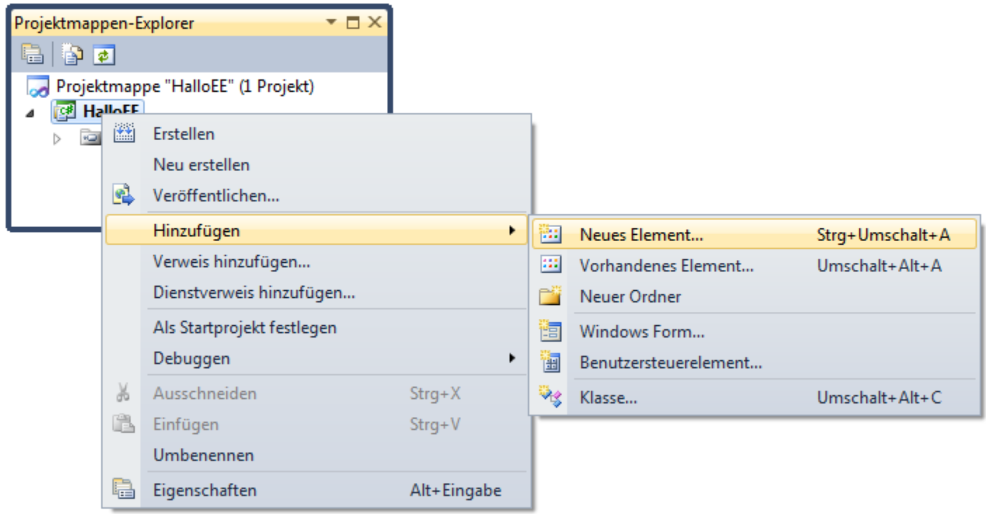
Datei > Neu > Projekt

den Dialog für neue Projekte:

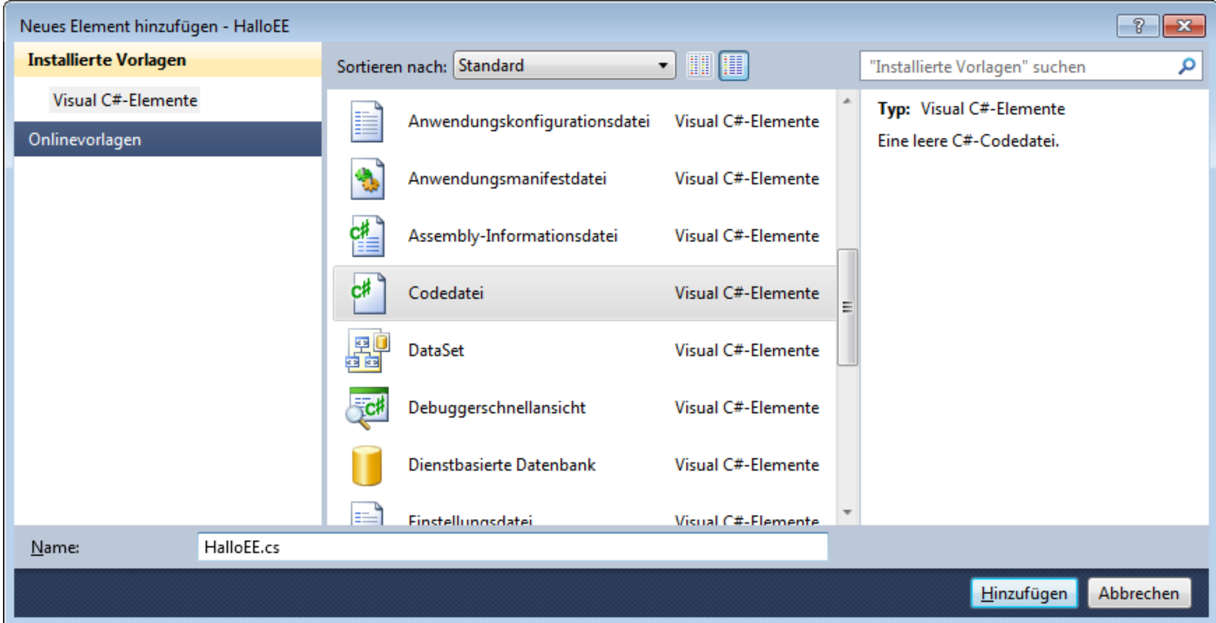


Wählen Sie die Vorlage **Leeres Projekt** sowie einen **Namen**. Nach einem Mausklick auf **OK** präsentiert die Entwicklungsumgebung ein neues Projekt ohne eine einzige Zeile Assistenten-Quellcode. Nach den guten Erfahrungen bei der flotten Entwicklung eines Programms mit Windows-Oberfläche dank Assistenten-Unterstützung (siehe Abschnitt 2.2.1.3) sind Sie vielleicht enttäuscht, jetzt beim Punkt Null beginnen zu müssen. Die Grundlagen einer Programmiersprache lassen sich meiner Meinung nach aber am besten in einer möglichst einfachen Umgebung erlernen. Genau eine solche Umgebung richten wir uns nun ein zur Verwendung in den Kapiteln 3 bis 8.

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

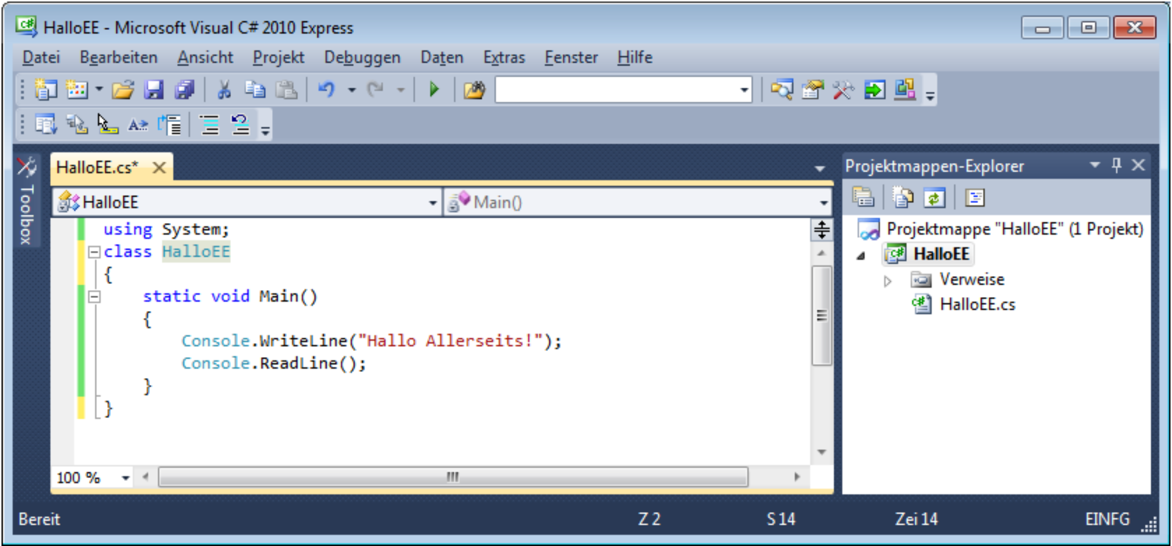


Entscheiden Sie sich für eine **Codedatei** mit geeignetem Namen:



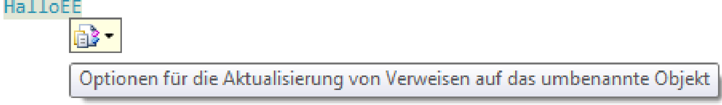
Nach dem Hinzufügen ist die Datei im Quellcode-Editor der Entwicklungsumgebung geöffnet.

Wir übernehmen den Quellcode vom `Hallo`-Beispielprogramm aus Abschnitt 2.1.1 und bitten am Ende der **Main**()-Methodendefinition die Klasse **Console**, ihre Methode **ReadLine**() auszuführen:

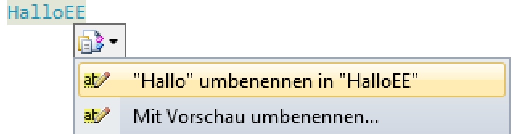


Diese Methode wartet auf die **Enter**-Taste und verhindert so, dass die im Rahmen der Entwicklungsumgebung mit dem Schalter ▶ oder der Funktionstaste **F5** gestartete Konsolenanwendung nach ihrer Bildschirmausgabe sofort verschwindet.

Außerdem passen wir den Namen der Startklasse an den Namen der Quellcodedatei an. Das Umbenennen einer Klasse, Methode oder Variablen kann in einem komplexen Projekt diverse Quellodeänderungen 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, macht ein Symbol



Optionen für die Aktualisierung von Verweisen auf das umbenannte Objekt zugänglich. Sobald die Maus auf das Symbol zeigt, ist per Klappmenü die benötigte Aktualisierung wählbar:

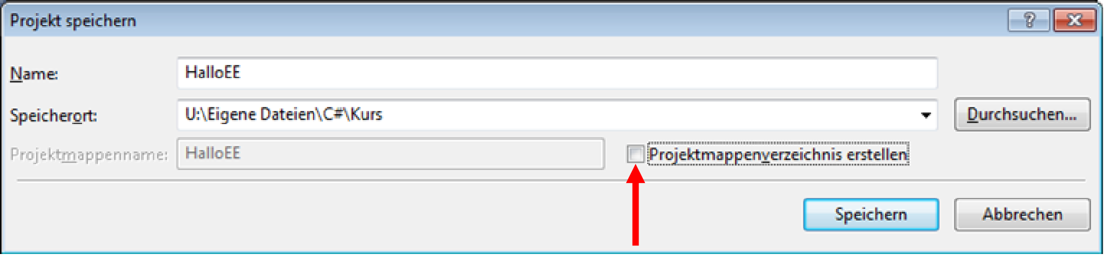


Bei unserem extrem primitiven Programm war kein weiteres Auftreten des Klassennamens zu aktualisieren. Sie werden aber bald die Möglichkeit schätzen lernen, in einem komplexen Programm einen Bezeichner zu ändern, ohne über die damit erzwungenen Aktualisierungen nachdenken zu müssen.

Fordern Sie über den Schalter oder den Menübefehl

Datei > Alle Speichern

den folgenden Dialog zum Speichern des Projekts an:



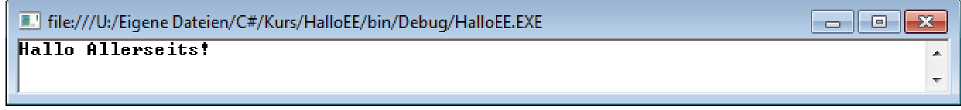
Sicher wird aus dem Miniaturprojekt nie eine Familie von Projekten entstehen, die in einem gemeinsamen Ordner versammelt werden sollten. Entfernen Sie daher nötigenfalls die Markierung beim Kontrollkästchen **Projektmappenverzeichnis erstellen**. Wir wählen damit eine flachere Projektdateiverwaltung, die für praktisch alle im Kurs entstehende Projekte angemessen ist. Im Projekt(mappen)ordner

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

entstehen die folgenden Dateien:

- **HalloEE.sln** zur Projektmappe
- **HalloEE.csproj** zum Projekt

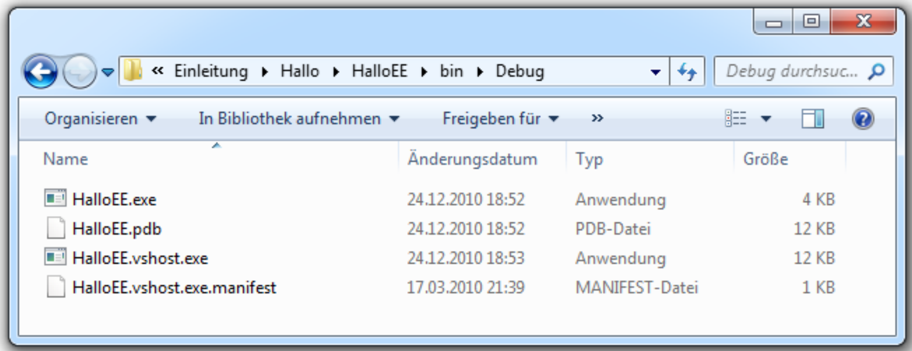
Über den Schalter oder die Funktionstaste **F5** veranlasst man das Übersetzen und das anschließende Starten der Anwendung:



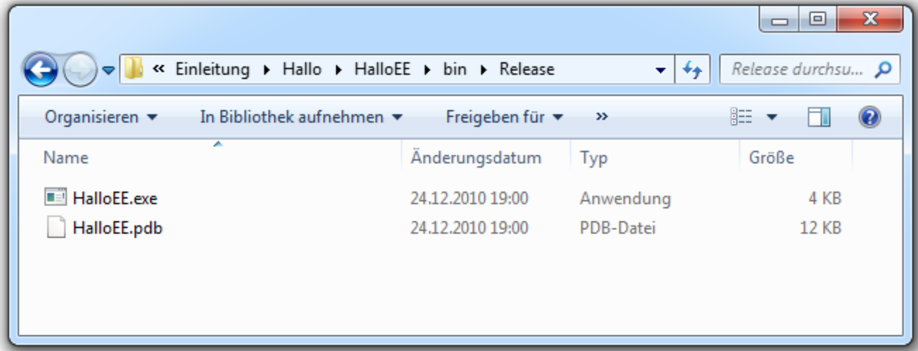
Sobald Sie die **Enter**-Taste drücken, endet die von der Klasse **Console** ausgeführte **ReadLine()**-Methode. Damit ist auch die **Main()**-Methode unserer Klasse fertig. Das Programm ist somit beendet, und das Konsolenfenster verschwindet.

Bei der Visual C# 2010 Express Edition ist die *einfache* Erstellungskonfiguration (siehe Abschnitt 2.2.3) voreingestellt, so dass die Funktionstasten **F5** und **F6** zu verschiedenen Übersetzungsergebnissen führen:

- Bei der mit **F5** (oder mit dem Menübefehl **Debuggen > Debugging starten**) angestoßenen Erstellung wird die Debug-Konfiguration verwendet, so dass der Compiler ein gut testbares, aber nicht optimiertes Assembly erzeugt und im Projekt-Unterordner **...\bin\Debug** ablegt, z.B.:



- Bei der mit **F6** (oder mit dem Menübefehl **Erstellen > Projektmappe erstellen**) angestoßenen Erstellung wird die Release-Konfiguration verwendet, so dass der Compiler ein optimiertes, aber weniger gut testbares Assembly erzeugt und im Projekt-Unterordner **…\bin\Release** ablegt, z.B.:



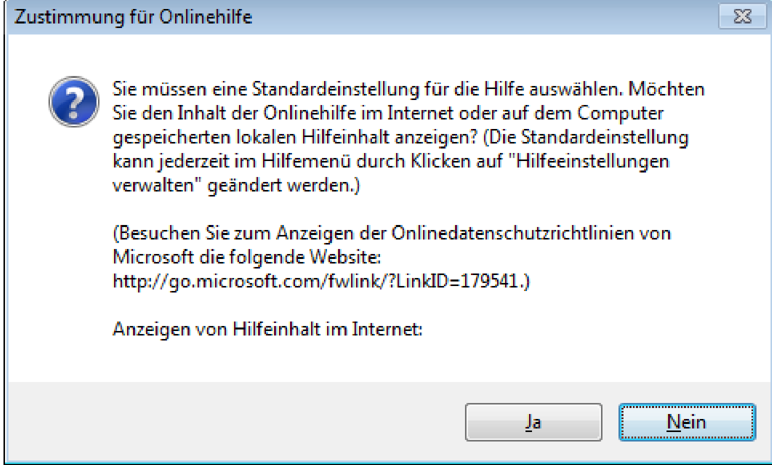
In jedem Fall ist das Assembly **HalloEE.exe** einsatzfähig und kann auf jedem Windows-Rechner mit .NET – Framework z.B. per Doppelklick auf die EXE-Datei gestartet werden.

2.2.2.4 FCL-Dokumentation und andere Hilfeinhalte

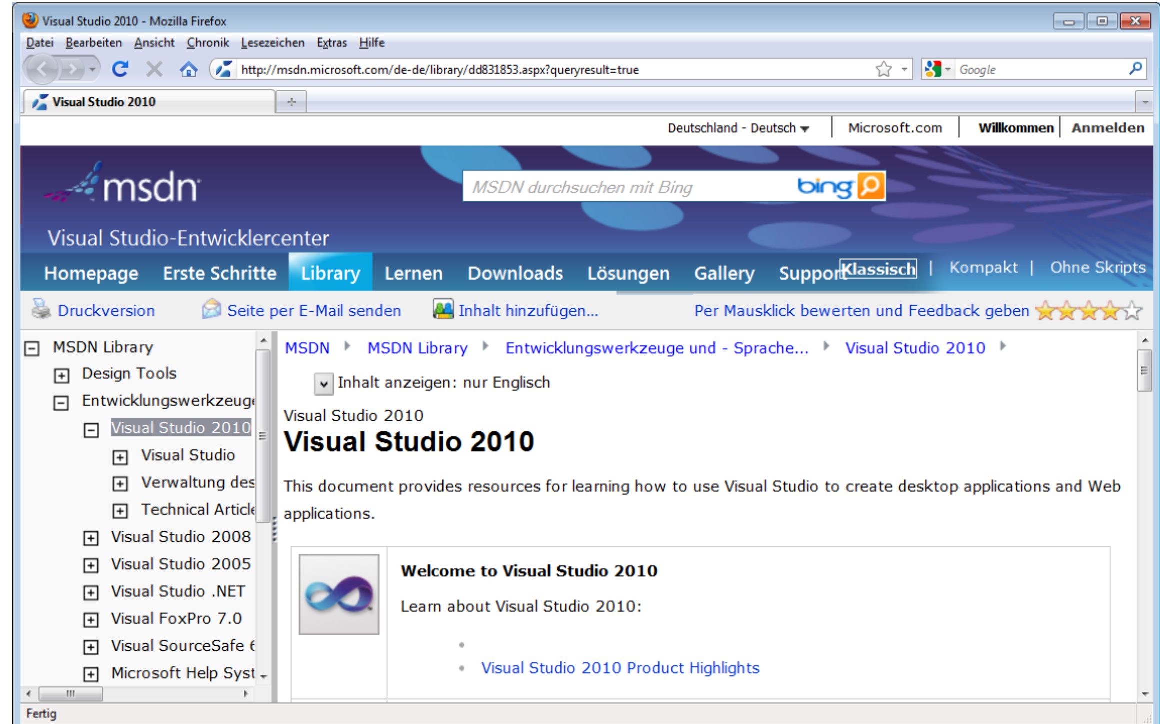
Beim ersten Aufruf der Hilfefunktion über den Menübefehl

Hilfe > Hilfe anzeigen

ist zu entscheiden, ob das Internet oder die lokale Festplatte als Informationsquelle bevorzugt wird:



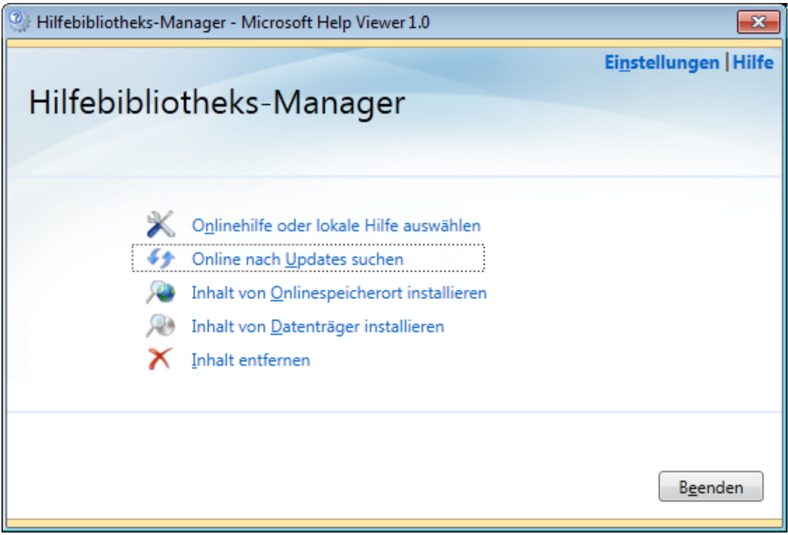
Wer über eine flotte Internet-Verbindung verfügt, kann sich eine lokale Informationsablage sparen und mit dem Schalter **Ja** für das Internet als Informationsquelle votieren. Unabhängig von der Informationsquelle dient zur Anzeige der bevorzugte Web Browser, z.B.:



Um die Entscheidung für eine Informationsquelle später zu revidieren, ruft man über den Menübefehl

Hilfe > Hilfeeinstellungen verwalten

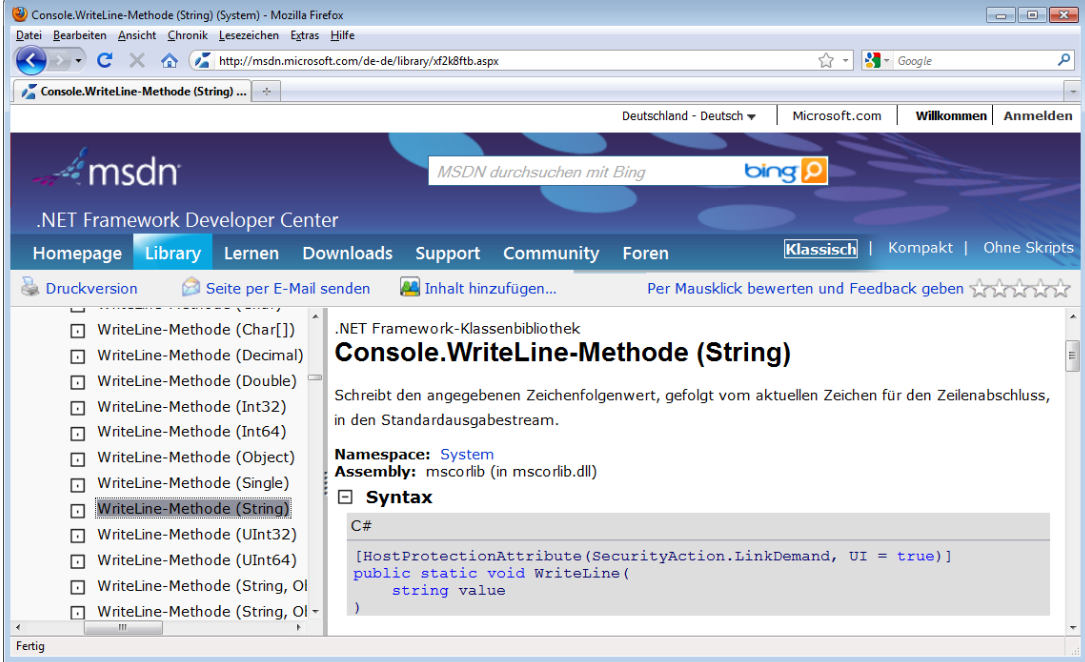
den **Hilfebibliotheksmanager** auf:



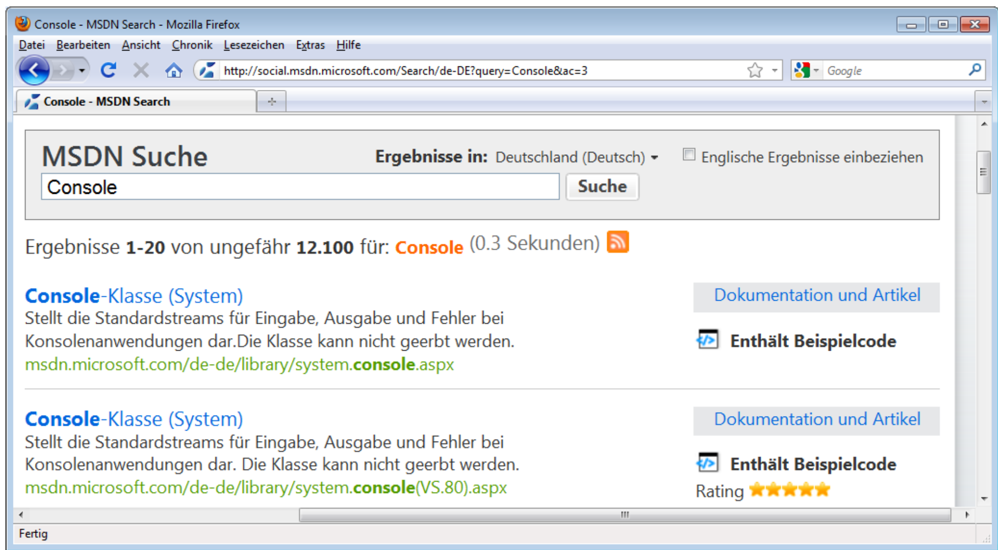
Wir benötigen die Hilfefunktion vor allem zur Präsentation der FCL-Dokumentation. Wer sich z.B. über die in unseren Beispielen oft benutzte Methode **WriteLine()** der Klasse **Console** aus dem Namensraum **System** informieren möchte, wählt in der Navigationszone den Ausgangspunkt **.NET Entwicklung** und bewegt sich weiter mit den Stationen:

.NET – Framework 4 > .NET – Framework-Klassenbibliothek > System-Namespace > Console-Klasse > Console Methoden > WriteLine-Methode

Nun ist noch zwischen vielen Spezialisierungen (Überladungen, siehe unten) der Methode **WriteLine()** zu wählen, damit im Inhaltsbereich passende Informationen samt Beispiel angezeigt werden:



Meist führt die Suchfunktion schneller zum Ziel, z.B.:



Zu einem markierten (die Einfügemarke enthaltenden) C# - Schlüsselwort oder FCL-Bezeichner im Quellcode-Editor der Entwicklungsumgebungen erreicht man die zugehörige Dokumentation besonders bequem über die Funktionstaste **F1**.

2.2.3 Debug- vs. Release-Konfiguration

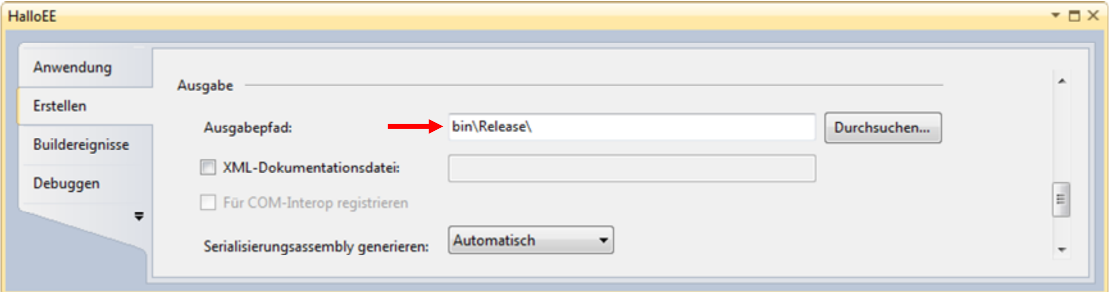
Wie bereits mehrfach erwähnt, unterscheiden Microsofts Entwicklungsumgebungen beim Übersetzen (bzw. Erstellen) die Konfigurationen **Debug** und **Release** (und ggf. noch weiteren benutzerdefinierten Konfigurationen). Eine Konfiguration ist ein Paket von Einstellungen für die Erstellung und legt auch den Ausgabeordner für die Übersetzung fest (**…\bin\Debug** bzw. **…\bin\Release**). Bei der Debug-Konfiguration werden zusätzliche Ausgaben erzeugt, die eine Quellcode-basierte Fehlersuche erlauben. Außerdem wird auf eine Optimierung des Codes verzichtet, weil diese die Fehlersuche erschwert. Bei der Release-Konfiguration wird umgekehrt auf Debug-Informationen verzichtet und eine Optimierung durchgeführt.

Für die Auswahl einer Konfiguration bieten unsere Entwicklungsumgebungen ein einfaches und eine erweitertes Verfahren.

2.2.3.1 Einfache Erstellungskonfiguration

Bei der *einfachen* Erstellungskonfiguration, die in der Visual C# 2010 Express Edition voreingestellt und bei den kommerziellen Visual Studio – Varianten einstellbar ist (siehe unten), kann man bequem zwischen der Debug- und der Release-Konfiguration wählen:

- Bei der mit **F5** (oder mit dem Menübefehl **Debuggen > Debugging starten**) angestoßenen Erstellung wird die Debug-Konfiguration und der Ausgabeordner **...\bin\Debug** verwendet.
- Bei der mit **F6** (oder mit dem Menübefehl **Erstellen > Projektmappe erstellen**) angestoßenen Erstellung kommt die Release-Konfiguration zum Einsatz. Den voreingestellten Ausgabeordner **...\bin\Release** kann man auf dem Registerblatt Erstellen des Formulars mit den Projekteigenschaften (z.B. erreichbar über den Menübefehl **Projekt > Eigenschaften**) ändern:



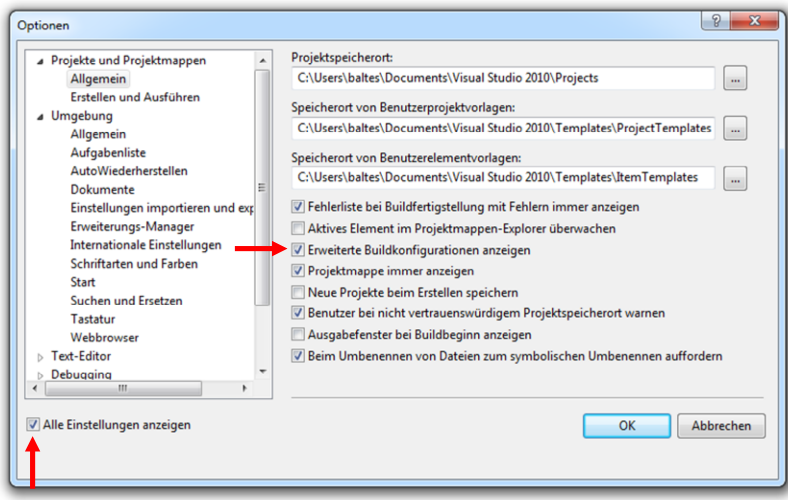
Als Zielplattform (im Sinn von Abschnitt 1.2.5.5) ist stets **x86** eingestellt.

2.2.3.2 Erweiterte Erstellungskonfiguration

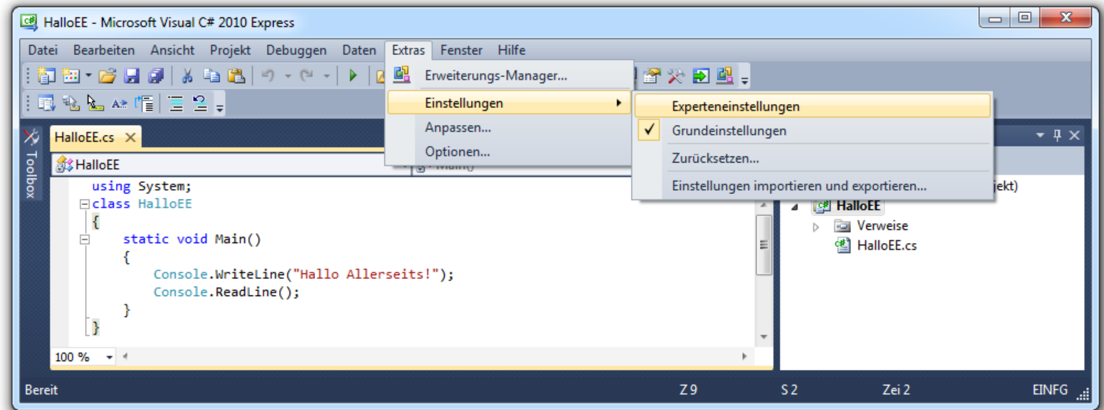
Bei der *erweiterten* Erstellungskonfiguration wählt man eine Projektmappenkonfiguration, die wiederum für jedes Projekt in der Mappe eine Projektkonfiguration festlegt. Per Voreinstellung impliziert z.B. die Debug-Mappenkonfiguration auch für jedes einzelne Projekt den Debug-Modus. Die gewählte Konfiguration wird unabhängig von der Erstellungsanforderung per **F5** (Menübefehl **Debuggen > Debugging starten**) oder **F6** (Menübefehl **Erstellen > Projektmappe erstellen**) verwendet. Ist also z.B. die Release-Konfiguration eingestellt, dann wird diese auch beim kombinierten Erstellen und Starten per **F5** genutzt.

Während die erweiterte Erstellungskonfiguration in den kommerziellen Visual Studio – Versionen voreingestellt ist, muss sie in der Express Edition bei Bedarf folgendermaßen eingestellt werden:

- Öffnen Sie über den Menübefehl **Extras > Optionen** den Dialog zur Konfiguration der Entwicklungsumgebung:

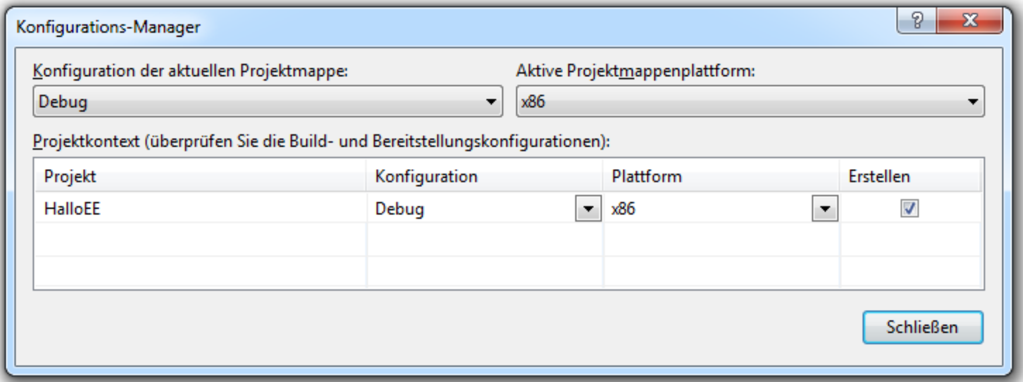


- Markieren Sie das Kontrollkästchen **Alle Einstellungen anzeigen** (links unten).
- Wählen Sie in der Navigationszone den Abschnitt **Projekte und Einstellungen > Allgemein**.
- Markieren Sie das Kontrollkästchen **Erweitete Buildkonfigurationen anzeigen**.
  Der Wechsel zur einfachen Erstellungskonfiguration durch Deaktivieren des Kontrollkästchens ist selbstverständlich auch in den kommerziellen Visual Studio – Versionen möglich. Dort fehlt allerdings das Kontrollkästchen **Alle Einstellungen anzeigen**.
- Bei der Express Edition muss man außerdem nach **Extras > Einstellungen** von der Voreinstellung **Grundeinstellungen** zu den **Experteneinstellungen** wechseln,



damit das Drop-Down – Menü zur Konfigurationswahl in der Symbolleiste **Standard** auftaucht:

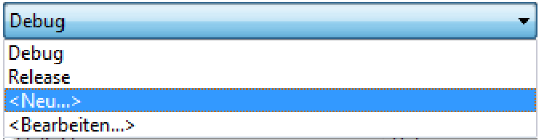
Welche Projektkonfigurationen (und Plattformen) man mit einer Projektmappenkonfiguration wählt, lässt sich per Konfigurations-Manager ermitteln und ändern. Man startet sein Fenster



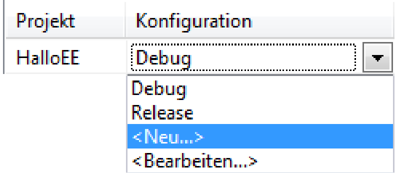
über das Drop-Down-Menü zur Konfigurationswahl in der Symbolleiste **Standard** (siehe oben) oder mit dem Menübefehl

Erstellen > Konfigurations-Manager

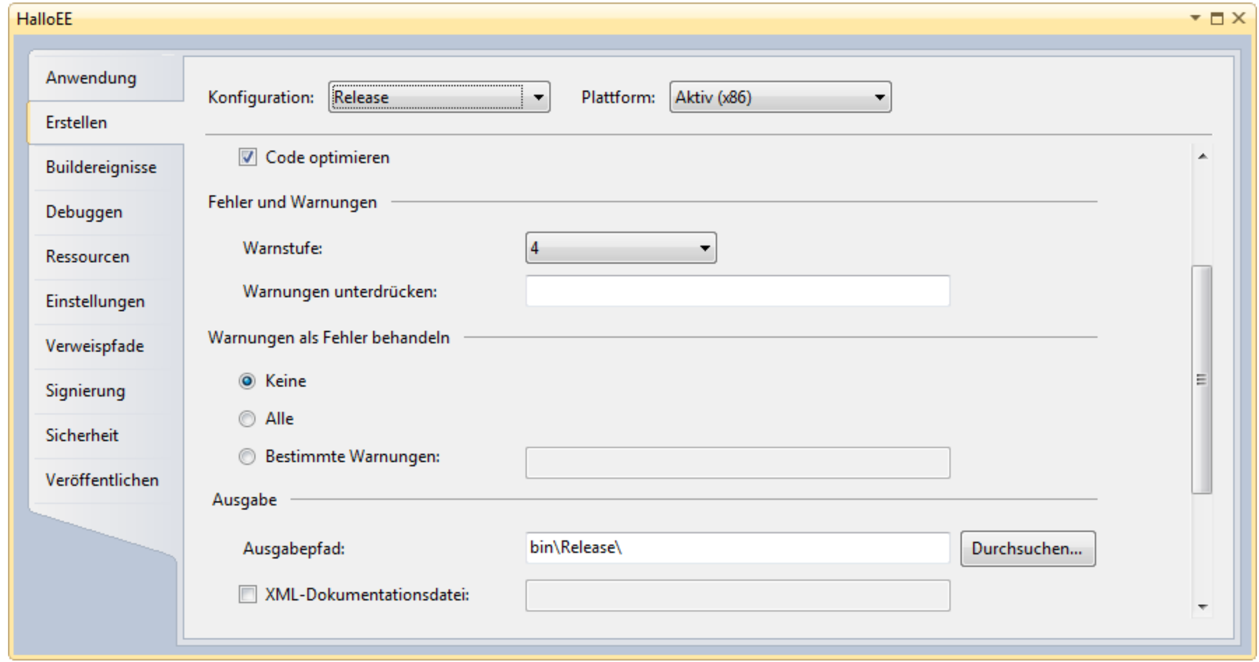
Für eine Mappenkonfiguration lässt sich nun zu jedem Projekt der Mappe eine Projektkonfiguration und eine Zielplattform wählen (siehe unten). Von den weiteren, meist selten benötigten Optionen dieses Dialogs soll hier noch die Kreation von eigenen Konfigurationen erwähnt werden. Eine neue Mappenkonfiguration legt man über das Drop-Down-Menü in der oberen linken Ecke an:



Eine neue Projektkonfiguration erstellt man über das Drop-Down-Menü zur Konfigurationszelle eines Projekts:



Die Flexibilität der erweiterten Erstellungskonfiguration wird deutlich bei einem Blick auf die Registerkarte **Erstellen** im Fenster mit den Projekteigenschaften (z.B. erreichbar über den Menübefehl **Projekt > Eigenschaften**).



Hier kann man für jede Kombination aus einer **Konfiguration** und einer **Plattform** diverse Einstellungen vornehmen (z.B. Code-Optimierung, Ausgabepfad). Es ist zu beachten, dass solche Einstellungen nur dann wirksam werden, wenn dem Projekt in der aktiven Mappenkonfiguration die passende Konfigurations-Plattform – Kombination zugewiesen ist (zu verifizieren per Konfigurations-Manager).

2.2.4 Compiler-Optionen in der Entwicklungsumgebung setzen

In Abschnitt 2.1.2 zur Übersetzung von Quellcode in die Microsoft Intermediate Language (MSIL) wurden wichtige Optionen des Compiler-Aufrufs behandelt, u.a.:

- Ausgabetyp des resultierenden Assemblies
- Referenzen auf Assemblies, die der Compiler nach Klassen durchsuchen soll
- Zielplattform (MSIL, x86, x64, Itanium)

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

2.2.4.1 Referenzen

Um dies üben zu können, erstellen wir nun mit der Visual C# 2010 Express Edition ein Hallo-Projekt gemäß Abschnitt 2.2.2.3, ersetzen aber die Textausgabe durch eine Messagebox:

Während die generelle GUI-Programmierung (*Graphical User Interface*) relativ anspruchsvoll ist und aus didaktischen Gründen vorläufig vermieden wird, gelingt die Präsentation einer Messagebox 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

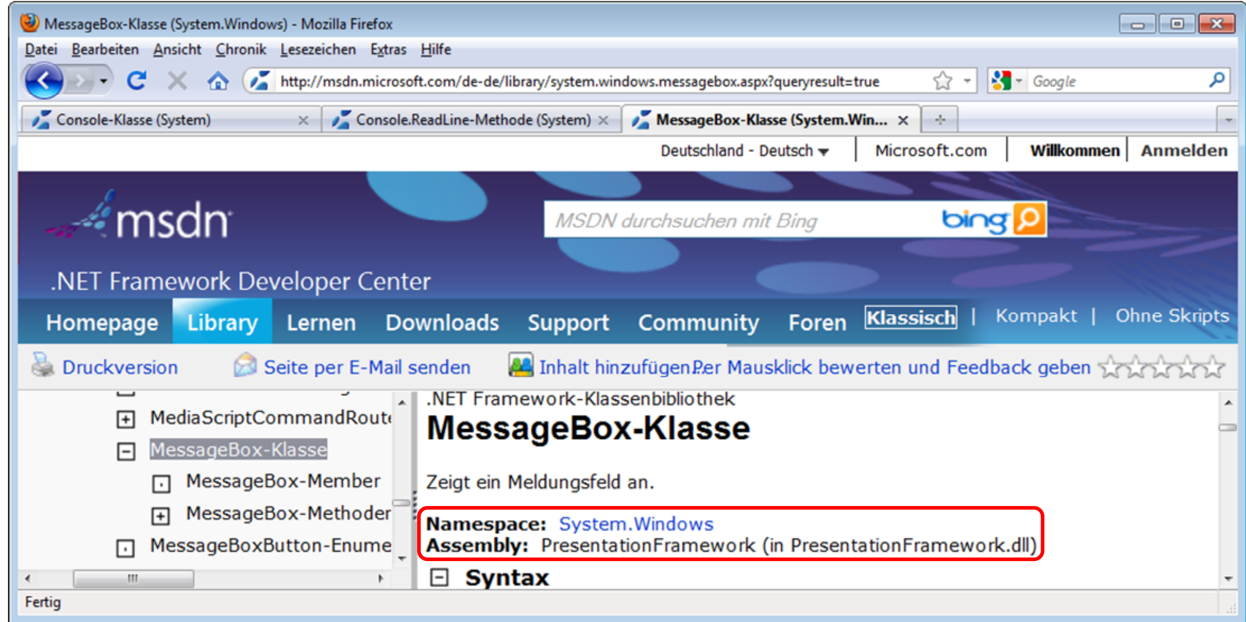
```
Console.ReadLine();
```

entfallen, weil kein Konsolenfenster offen gehalten werden muss (vgl. Abschnitt 2.2.4.2).

Ohne weitere Ergänzungen des Quellcodes kann die Übersetzung allerdings nicht gelingen, wie die Entwicklungsumgebung vorausblickend mit genauer Fehlerdiagnose mitteilt:

```
HalloMB.cs*
HalloEE                                   Main()
using System;
class HalloEE
{
    static void Main()
    {
        MessageBox.Show("Hallo Allerseits!");
                 Der Name "MessageBox" ist im aktuellen Kontext nicht vorhanden.
    }
}
100 %
```

Mit Hilfe der FCL-Dokumentation ermittelt man leicht, welcher Namensraum anzugeben ist (als Präfix zum Klassennamen oder in einer **using**-Direktive), um dem C# - Compiler die Klasse **MessageBox** bekannt zu machen. Um die folgende Information



aus dem Quellcode-Editor von Visual C# zu erhalten, platziert man die Einfügemarke in das kritisierte Wort **MessageBox** und drückt die Funktionstaste **F1**.

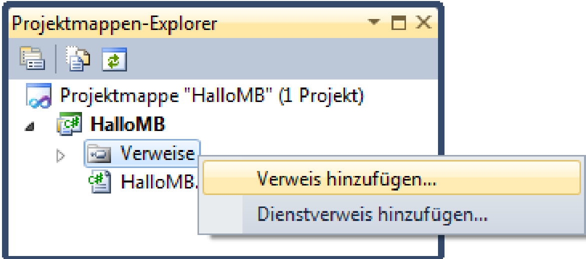
Mit dem folgenden funktionstüchtigen (!) Quellcode wird das Problem scheinbar nicht gelöst, sondern nur verlagert:

HalloMB.cs*

```
using System;
class HalloEE
{
    static void Main()
    {
        System.Windows.MessageBox.Show("Hallo Allerseits!");
        Console
    }
}
```

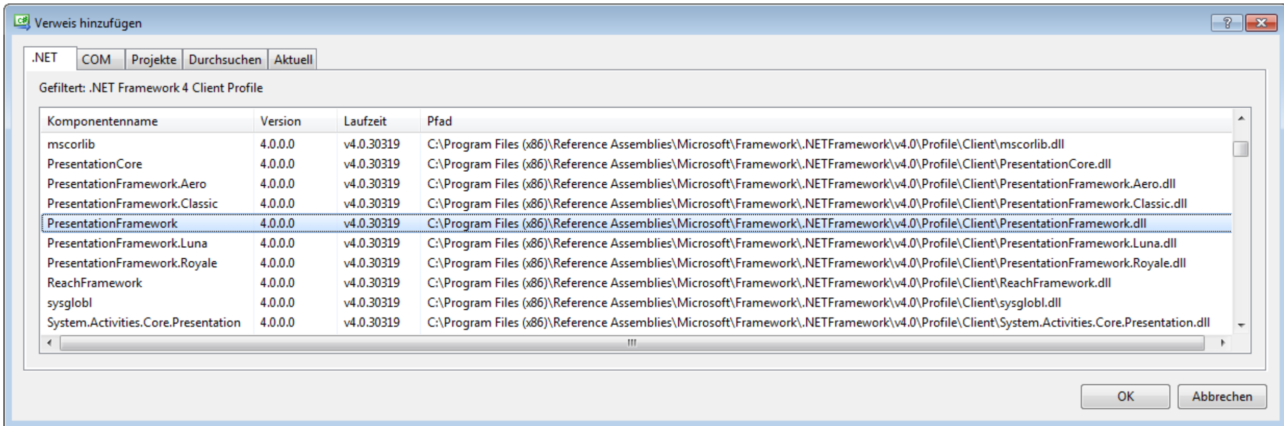
Der Typ- oder Namespacename "Windows" ist im Namespace "System" nicht vorhanden. (Fehlt ein Assemblyverweis?)

Visual C# gibt aber mit der Frage nach einem eventuell fehlenden Assembly-Verweis einen guten Hinweis auf das Restproblem: Weil sich das die Klasse **MessageBox** implementierende Assembly **PresentationFramework.dll** nicht in der beim Übersetzen zu berücksichtigenden Referenzliste 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 Eintrag **Verweise** die Option **Verweis hinzufügen**:

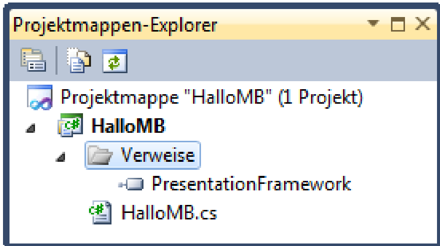


[1] Bei Verwendung der Projektvorlage **WPF-Anwendung** legt die Entwicklungsumgebung automatisch eine Referenz auf das Assembly **PresentationFramework.dll** an, nicht jedoch bei der Vorlage **Leeres Projekt**, die wir aus Gründen der Einfachheit und Transparenz derzeit bevorzugen.

Nun kann in folgender Dialogbox auf der Registerkarte **.NET** das gesuchte Assembly lokalisiert und im markierten Zustand mit **OK** in die Verweisliste aufgenommen werden:[1]



Anschließend taucht der Verweis im Projektmappen-Explorer auf,



und die Reklamation im Quellcode-Editor verschwindet. Wenn Sie das Programm mit dem Schalter oder der Funktionstaste **F5** übersetzen und starten, erscheint die gewünschte Messagebox.

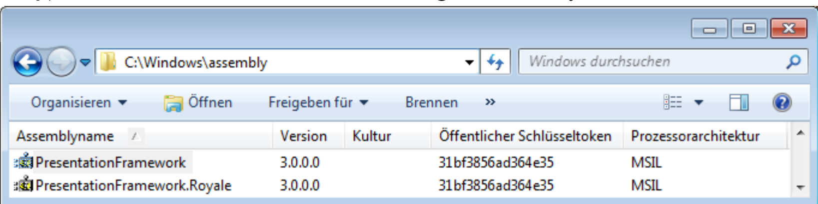
2.2.4.2 Ausgabetyp

Außerdem erscheint aber auch ein leeres Konsolenfenster. Um seinen Auftritt zu verhindern, ersetzt man per Compiler-Option den voreingestellten Ausgabetyp **exe** durch die Alternative **winexe** (vgl. Abschnitt 2.1.2). Bei einer integrierten Entwicklungsumgebung ändert man dazu eine Projektoption, z.B. bei der Visual C# 2010 Express Edition nach:

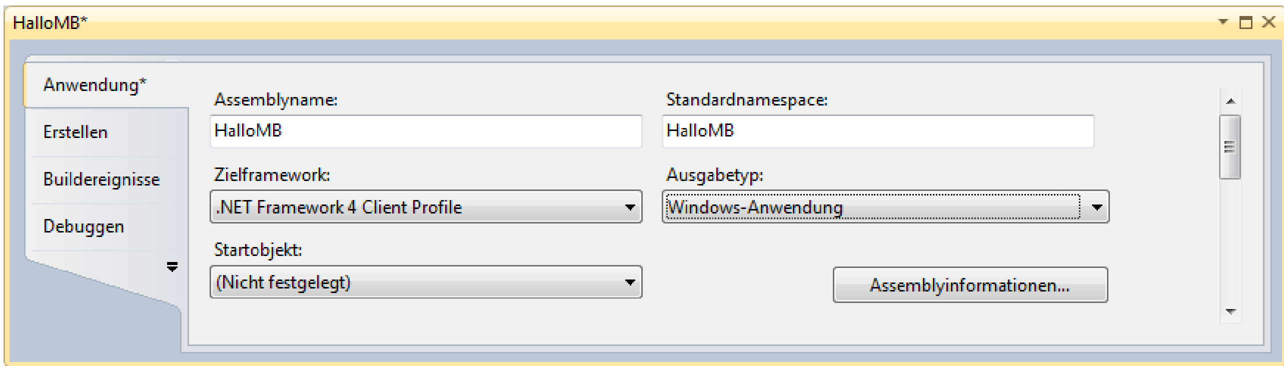
Projekt > Eigenschaften > Anwendung

in folgendem Dialogfenster:

[1] Vermutlich erinnern Sie sich an den im Abschnitt 1.2.5.4 genannten Installationsort für GAC-Assemblies (**C:\Windows\assembly**). Dort befindet sich auch das benötigte Assembly **PresentationFramework.dll**:



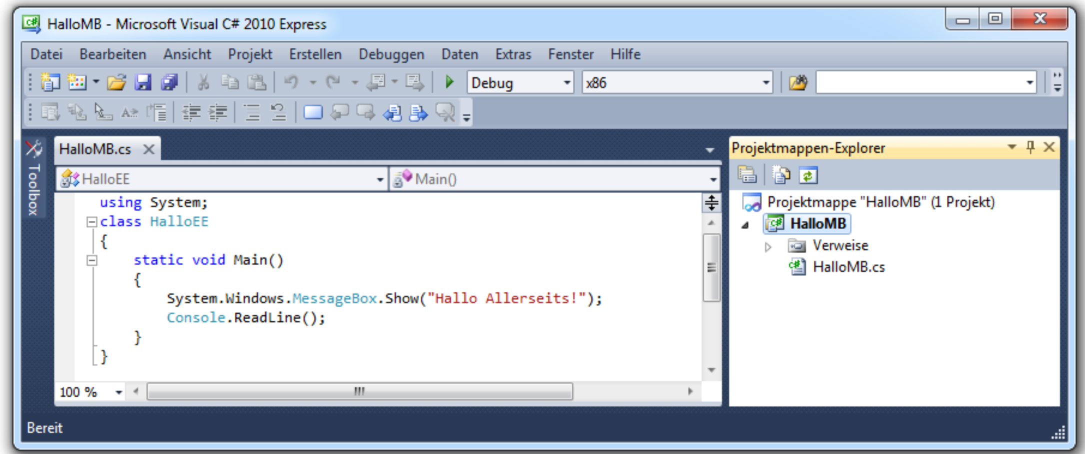
Von der **CLR** werden tatsächlich die Bibliotheks-Assemblies in **C:\Windows\assembly** verwendet. In der Entwicklungsphase werden jedoch seit .NET 4.0 die Referenzen auf Dateien im Ordner **C:\Program Files (x86)\Reference Assemblies** gesetzt. Damit sollen die teilweise abweichenden Bedürfnisse der Laufzeit- und der Entwicklungsphase jeweils optimal unterstützt werden.



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

2.2.4.3 Zielplattform

Wer das Visual Studio 2010 (Ultimate oder Express) unter Windows **64** einsetzt und mit der erweiterten Erstellungskonfiguration arbeitet (siehe Abschnitt 2.2.3.2), wird feststellen, dass bei neuen **EXE**-Projekten in C# die Zielplattform **x86** voreingestellt ist, z.B.:



Dies überrascht, weil nach Abschnitt 1.2.5.5 der C# - Compiler generell die voreingestellte Zielplattform **MSIL** verwendet. Während das Visual Studio **2008** die Voreinstellung des Compilers beibehalten hat, zeigt die Version 2010 ein anderes Verhalten:

- Bei DLL-Projekten wird per Voreinstellung die Zielplattform MSIL verwendet.
- Bei EXE-Projekten ist die Plattform x86 voreingestellt.

Der Microsoft-Mitarbeiter Rick Byers begründet die Entscheidung des Visual Studio – Entwicklungsteams in einem Blog-Beitrag vom 6. Juni 2009 mit dem Titel: [1]

AnyCPU Exes are usually more trouble than they're worth.

Zentrale Argumente aus dem Blog-Beitrag wurden schon im Abschnitt 1.2.5.5 über plattformspezifische Assemblies wiedergegeben.

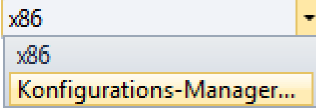
Wird für ein EXE-Projekt eine alternative Zielplattform benötigt, etwa um unter Windows 64 mehr als 4 GB Adressraum[2] nutzen zu können, muss man die erweiterte Erstellungskonfiguration aktivie-

---

[1] http://blogs.msdn.com/b/rmbyers/archive/2009/06/08/anycpu-exes-are-usually-more-trouble-then-they-re-worth.aspx

[2] Unter einem 32-bittigen Windows muss sich 32-Bit - Prozess die maximal 4 GB Arbeitsspeicher mit dem Betriebsystem und residenten Treibern teilen, so dass der Prozess in der Regel nur 2 GB zur Verfügung hat. Unter einem 64-bittigen Windows kann ein 32-Bit - Prozess hingegen volle 4 GB Arbeitsspeicher nutzen.

ren (siehe Abschnitt 2.2.3.2) und den Konfigurations-Manager bemühen, was über die Plattformliste in der Symbolleiste **Standard**

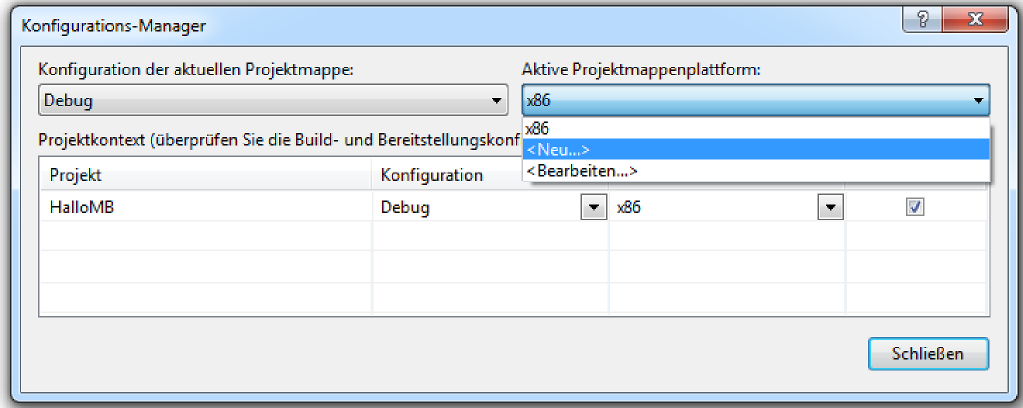


oder den Menübefehl

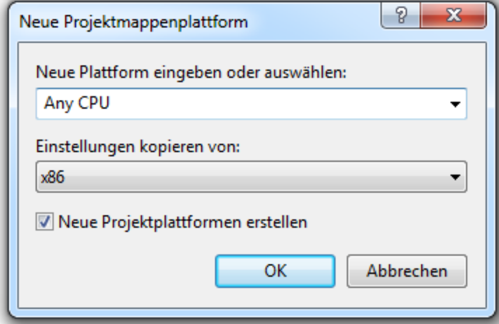
Erstellen > Konfigurations-Manager

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

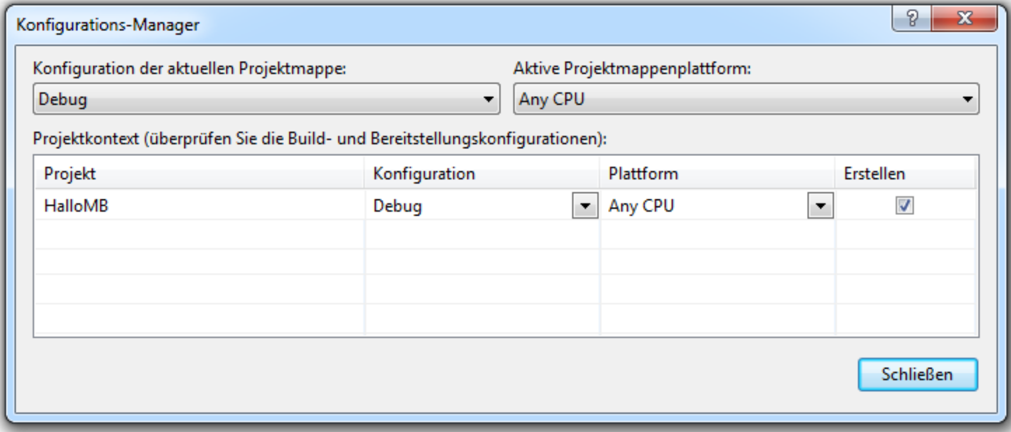
- Eine neue **Projektmappenplattform** anlegen:



- Im folgenden Dialog die gewünschte Plattform wählen, z.B. **Any CPU** (entspricht **MSIL**):



- Die Markierung beim Kontrollkästchen **Neue Projektplattformen erstellen** belassen, so dass nach dem Quittieren mit **OK** bei allen Projekten in der Mappe die neue Plattform eingestellt ist:



Beim nächsten Erstellen mit der modifizierten Projektmappenkonfiguration (z.B. per **F5** oder **F6** angefordert) entsteht ein Assembly mit der gewünschten Zielplattform.

Bei den Beispielprojekten im Kurs werden wir wohl kaum einen Grund finden, die vom Visual Studio 2010 vorgeschlagene Zielplattform zu ändern.

2.3 Übungsaufgaben zu Kapitel 2

1) Installieren Sie nach Möglichkeit auf Ihrem privaten PC die Visual C# 2010 Express Edition (gemäß Abschnitt 2.2.2.1).

2) Experimentieren Sie mit dem `Hallo`-Beispielprogramm in Abschnitt 2.1.1:

- Ergänzen Sie weitere Ausgabeanweisungen.
- Erstellen Sie eine Variante ohne **using**-Direktive (vgl. Abschnitt 1.2.7).

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

```
Using System;
class Hallo {
  static void Main() {
    Console.WriteLn("Hallo Allerseits!);
  }
```

3 Elementare Sprachelemente

In Kapitel 1 wurde anhand eines halbwegs realistischen Beispiels versucht, einen ersten Eindruck von der objektorientierten Softwareentwicklung 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 bei C# nicht wesentlich anders als bei älteren, *nicht* objektorientierten Sprachen (z.B. C) aus.

3.1 Einstieg

3.1.1 Aufbau von einfachen C# - Programmen

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

- Ein C# - Programm besteht aus **Klassen**. Unser Bruchadditionsbeispiel in Abschnitt 1.1 besteht aus den beiden Klassen `Bruch` und `BruchAddition`.
- In einer **Klassendefinition** werden **Felder** deklariert sowie **Eigenschaften** und **Methoden** definiert. Meist verwendet man für den Quellcode einer Klasse jeweils eine eigene Datei mit demselben Namen wie die Klasse und **.cs** als Namenserweiterung.
- 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 und von der Klasse selbst ausgeführt (Modifikator **static**). Wenn die **Main()** – Methode ihrem Aufrufer (also der CLR) eine ganze Zahl als Information über den (Miss)erfolg ihrer Tätigkeit liefert (z.B. 0: alles gut gegangen, 1: Fehler), ist in der Methodendefinition der Rückgabetyp **int** anzugeben. Fehlt eine solche Rückgabe, ist der Pseudorückgabetyp **void** anzugeben. Beim Bruchadditions-Beispiel in Abschnitt 1.1 ist die Klasse `BruchAddition` startfähig. Ihre **Main()**-Methode verwendet den Pseudorückgabetyp **void**.
- Die zu einem Programm gehörigen Quellcodedateien werden gemeinsam vom **Compiler** in die Microsoft Intermediate Language (MSIL) übersetzt, z.B.:

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

  Das resultierende Assembly **BruchAddition.exe** übernimmt seinen Namen von der Startklasse und enthält neben dem **MSIL-Code** auch Typ- und Assembly-**Metadaten**.
- Die ersten Zeilen einer C# - Quellcodedatei enthalten meist **using**-Direktiven zum Import von **Namensräumen**, damit die dortigen Klassen später ohne Namensraum-Präfix vor den Klassennamen angesprochen werden können.
- Die Definition einer Klasse, Eigenschaft oder Methode besteht aus:
  - **Kopf**
    Bei einer Klassendefinition folgt auf das Schlüsselwort **class** der relativ frei wählbare Klassenname.
  - **Rumpf**
    Im Rumpf einer Klassendefinition werden Felder deklariert sowie Eigenschaften und Methoden definiert. Im Rumpf einer Methode finden sich die Anweisungen zur Bewältigung des Auftrags.

  Details folgen gleich in Abschnitt 3.1.2.
- Eine **Anweisung** ist die kleinste ausführbare Einheit eines Programms. In C# sind bis auf wenige Ausnahmen alle Anweisungen mit einem **Semikolon** abzuschließen.

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

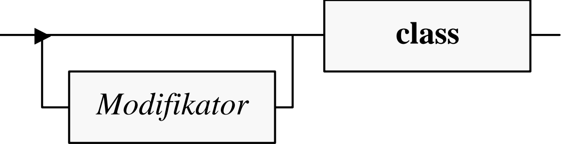
```
using System;
class Prog {
   static void Main() {
      // Platz zum Üben elementarer Sprachelemente
   }
}
```

Damit die wenig objektorientierten Beispiele Ihrem Programmierstil nicht prägen, wurde zu Beginn des Kurses (in Abschnitt 1.1) eine Anwendung vorgestellt, die bereits etliche OOP-Prinzipien realisiert.

3.1.2 Syntaxdiagramme

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

- Man bewegt sich vorwärts in Pfeilrichtung durch das Syntaxdiagramm und gelangt dabei zu Rechtecken, welche die an der jeweiligen Stelle zulässigen Sprachbestandteile angeben. z.B.:



- Bei Abzweigungen kann man sich für eine Richtung entscheiden, wenn nicht durch Pfeilspitzen eine Bewegungsrichtung vorgeschrieben ist.
- 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 durch *kursive* Schrift gekennzeichnet.

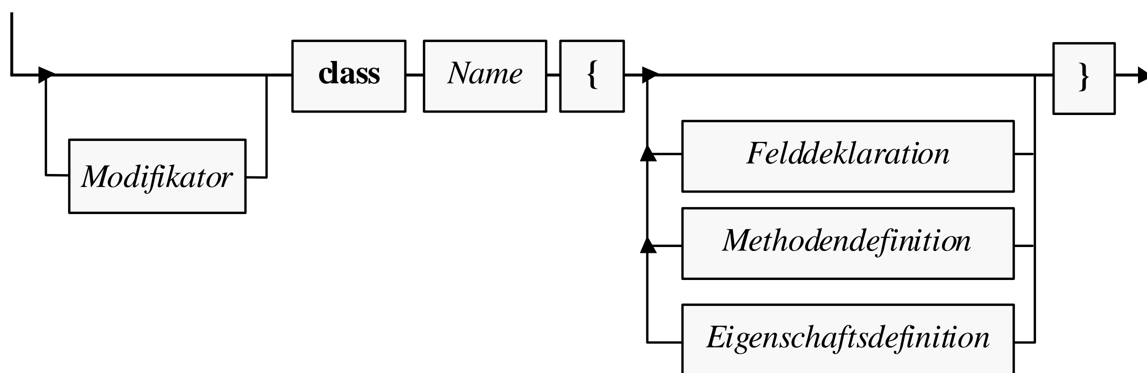
Als Beispiele betrachten wir die Syntaxdiagramme zur Definition von Klassen, Methoden- und Eigenschaften. Aus didaktischen Gründen zeigen die Diagramme nur solche Sprachbestandteile, die bisher in einem Beispiel verwendet oder im Text beschrieben wurden, so dass sie langfristig keinesfalls als Referenz taugen. Trotz der Vereinfachung sind die Syntaxdiagramme aber 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 3.1.2 geht es nicht nur darum, Syntaxdiagramme als metasprachliche Hilfsmittel einzuführen, sondern die vorgestellten Beispiele tragen hoffentlich trotz der gerade angesprochenen Kompromisse auch zur allmählichen Festigung der wichtigen Begriffe *Klasse*, *Methode* und *Eigenschaft* bei.

3.1.2.1 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 (eventuell an anderer Stelle erläuterte) Weise ersetzt, sollte eine syntaktisch korrekte Klassendefinition entstehen.

Als Beispiel betrachten wir die im Abschnitt 1.1 vorgestellte Klasse `Bruch`:

Modifikator ↓ Name ↓

Feld-deklarationen: `int zaehler, nenner = 1;`

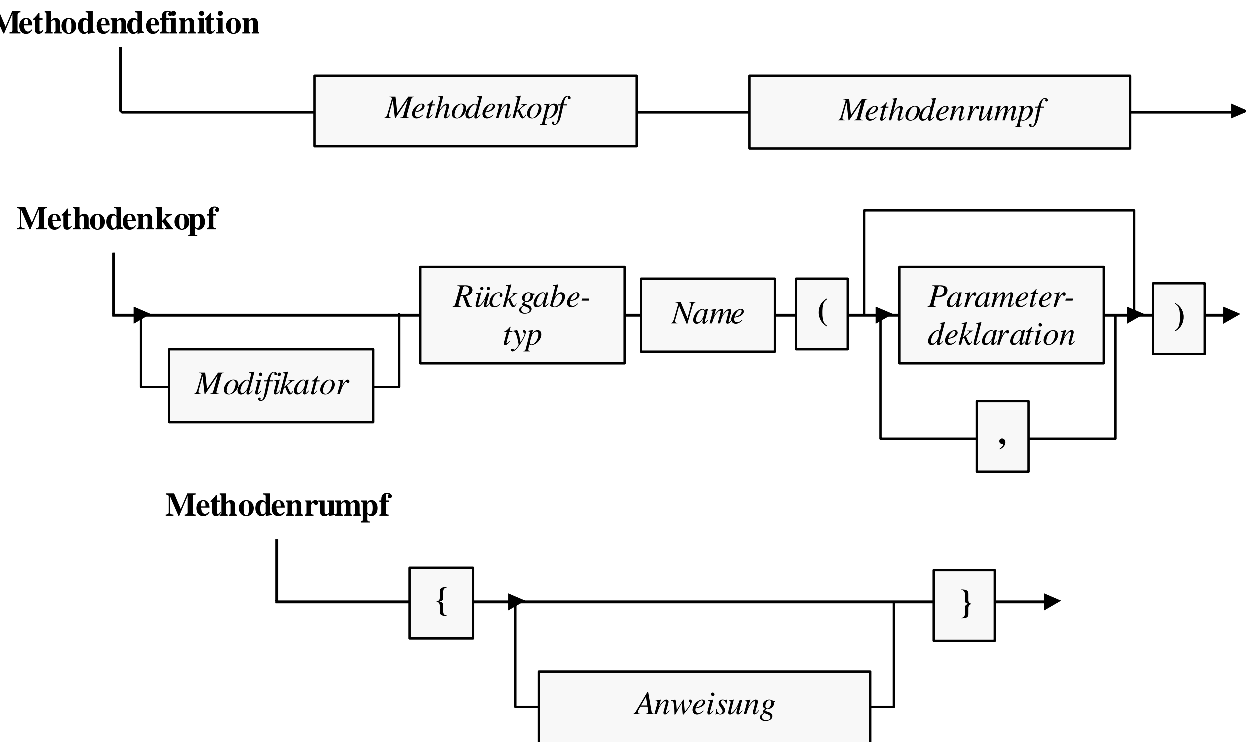
Eigenschafts-definitionen: `Zaehler`, `Nenner`

Methoden-definitionen: `Zeige()`, `Kuerze()`, `Addiere(Bruch b)`, `Frage()`

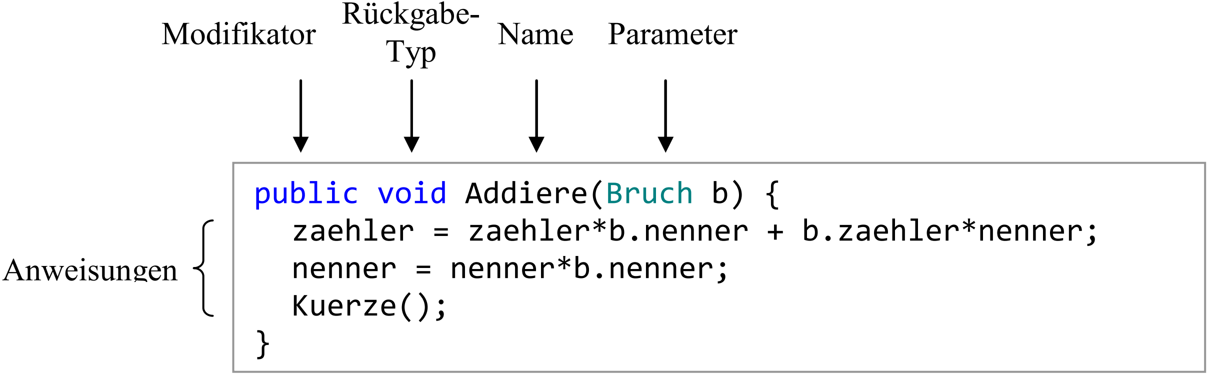
```
public class Bruch {

   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() {
      . . .
   }

}
```

3.1.2.2 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()`:



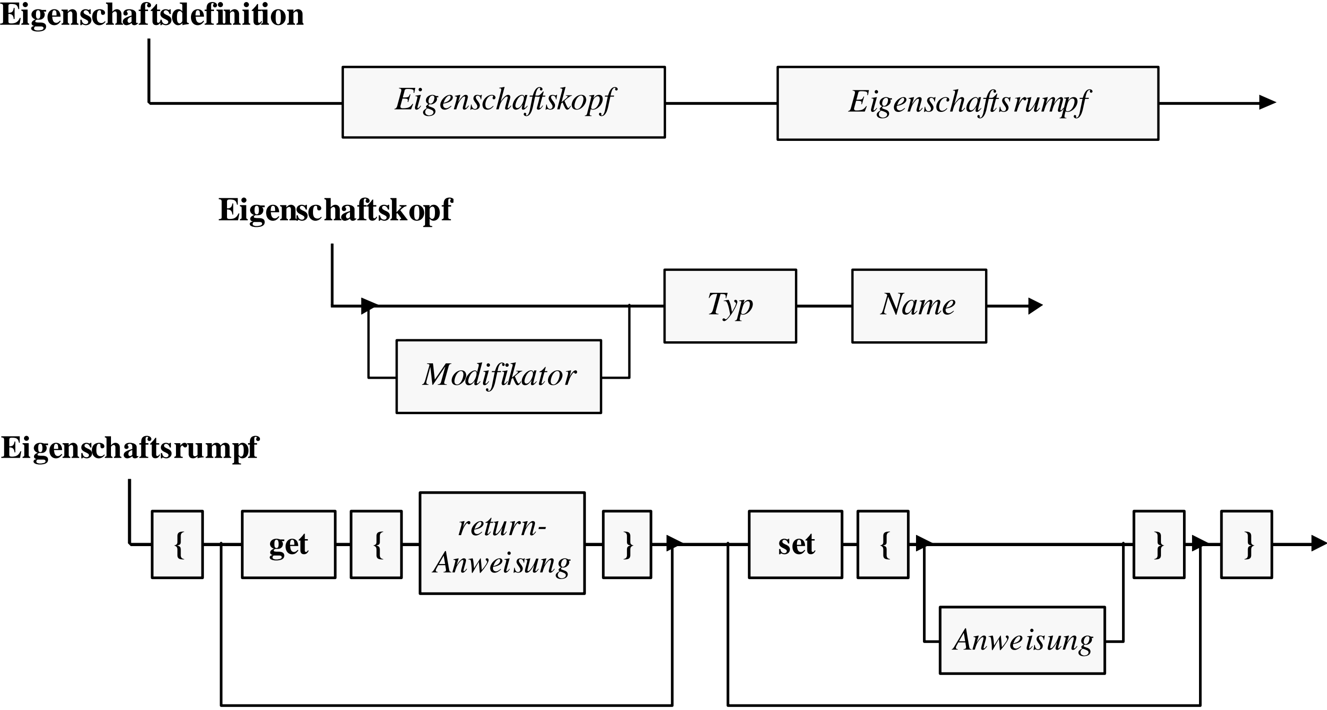
In vielen Methoden werden so genannte *lokale Variablen* (siehe unten) deklariert, z.B. in der `Bruch`-Methode `Kuerze()`:

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

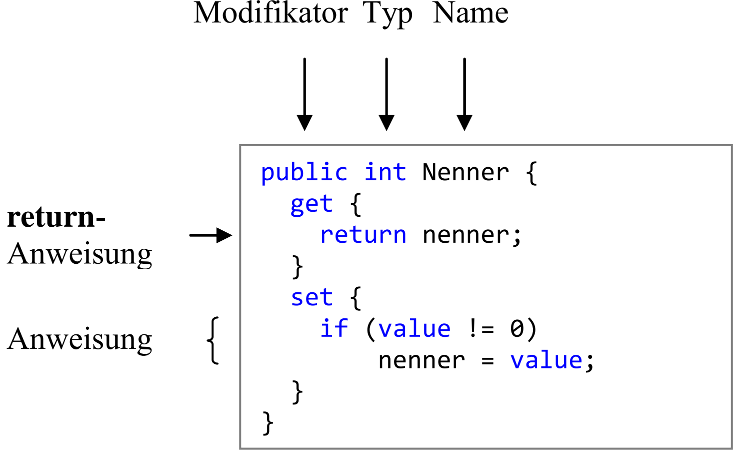
Weil wir bald u.a. von einer Variablendeklarations*anweisung* sprechen werden, benötigt das Syntaxdiagramm zum Methodenrumpf jedoch (im Unterschied zum Klassendefinitionsdiagramm) *kein* separates Rechteck für die Variablendeklaration.

3.1.2.3 Eigenschaftsdefinition

Auch beim Syntaxdiagramm für den Eigenschaftsbegriff gehen wir schrittweise vor:



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



3.1.3 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 Visual C# 2010 Express Edition, die via Windows-Zwischenablage in den Text übernommen wurden.

Ob man beim Rumpf einer Klassen- oder Methodendefinition die öffnende geschweifte Klammer an das Ende der Kopfzeile setzt oder an den Anfang der Folgezeile, ist Geschmacksache, z.B.:

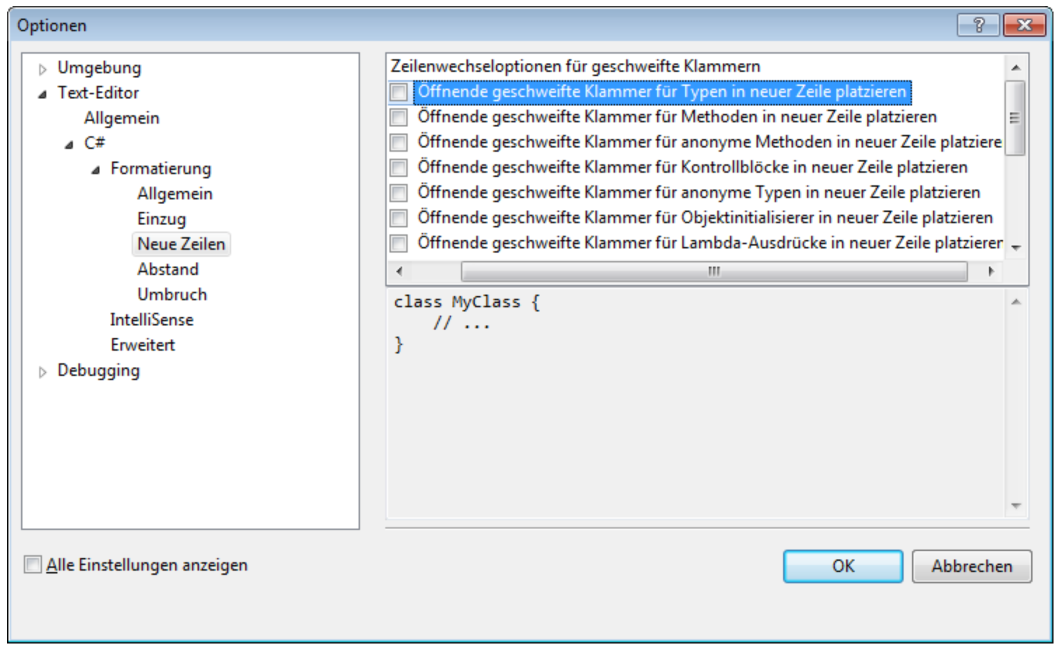
```
class Hallo {
  static void Main() {
    System.Console.WriteLine("Hallo");
  }
}
```

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

Die Visual C# 2010 Express Edition bevorzugt die rechte Variante, kann aber nach

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

umgestimmt werden:



Weitere Hinweise zur übersichtlichen Gestaltung des C# - Quellcodes finden sich z.B. bei Krüger (2002).

3.1.4 Kommentare

C# bietet folgende Möglichkeiten, den Quelltext zu kommentieren:

- **Zeilenrestkommentar**
  Alle Zeichen von `//` bis zum Ende der Zeile gelten als Kommentar, wobei kein Kommentar-Terminierungszeichen erforderlich ist, z.B.:

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

  Hier wird eine Variablendeklarationsanweisung in derselben Zeile kommentiert.

- **Mehrzeilenkommentar**
  Zwischen einer Einleitung durch `/*` und einer Terminierung durch `*/` kann sich ein ausführlicher Kommentar auch über mehrere Zeilen erstrecken, 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 u.a. auch dazu, ein Programmsegment (vorübergehend) zu deaktivieren, ohne ihn löschen zu müssen.

  Weil der Mehrzeilenkommentar ohne farbliche Hervorhebung der auskommentierten Passage unübersichtlich ist, wird er selten verwendet. Wenn man z.B. im Editor unserer Entwicklungsumgebung das Auskommentieren eines markierten Blocks mit dem Menübefehl

  Bearbeiten > Erweitert >Auswahl kommentieren

  bzw. mit der Tastenkombination **Strg+K**, **Strg+C** veranlasst, werden doppelte Schrägstriche vor jede Zeile gesetzt. 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, werden die Kommentarschrägstriche entfernt.

- **Dokumentationskommentar**
  Neben den Kommentaren, welche ausschließlich das Lesen des Quelltexts unterstützen sollen, kennt C# noch den Dokumentationskommentar in XML-Syntax. Er wird vom Compiler bei einem Aufruf mit der Option **doc** in eine separate XML-Dokumentationsdatei umgesetzt, z.B.:

```
csc  *.cs  /doc:Bruch.xml
```

  Daraus kann über meist kostenlos verfügbare Hilfsprogramme (z.B. *Sandcastle*) eine HTML-Dokumentation erstellt werden kann. In Microsofts Entwicklungsumgebungen fehlt leider eine entsprechende Funktionalität.

  Ein Dokumentationskommentar 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>
/// Ein Bruch-Objekt verhindert, dass sein Nenner auf Null
/// gesetzt wird und hat daher stets einen sinnvollen Wert.
/// </summary>
public int Nenner {
   . . .
}
```

  Durch `/**` eingeleitete und durch `*/` terminierte *mehrzeilige* Dokumentationskommentare erfordern die Beachtung spezieller Regeln und sind wegen des damit verbundenen Fehlerrisikos nicht empfehlenswert.

3.1.5 Namen

Für Klassen, Eigenschaften, Methoden, Felder, Parameter und sonstige Elemente eines C# - Programms benötigen wir Namen, wobei folgende Regeln gelten:

- Die Länge eines Namens ist nicht begrenzt.
- 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 vielen 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.

  `Anz   anz   ANZ`

  grundverschiedene Namen.
- Die folgenden **reservierten Schlüsselwörter** dürfen nicht als Namen verwendet werden:

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	volatile	void	while				

- Daneben gibt es **kontextbezogen-reservierte Schlüsselwörter**, die nur in bestimmten Kontexten als Namen verboten sind:

add	ascending	by	descending	dynamic	equals	from	get	global
group	into	join	let	on	orderby	partial	remove	select
set	value	var	where	yield				

  Vorsichtshalber sollte man sie aber generell als Namen vermeiden.
- Namen müssen in ihrem Deklarationsbereich (siehe unten) eindeutig sein.

Während Sie obige Regeln einhalten *müssen*, ist die Beachtung der folgenden Konventionen freiwillig, aber empfehlenswert:

- Die Namen von *lokalen* (methodeninternen) Variablen (siehe Abschnitt 3.3.3), Methodenparametern und *privaten* (gekapselten, nur klassenintern ansprechbaren) Feldern werden *klein* geschrieben, z.B.:

  `ggt` 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 Anfangsbuchstaben, z.B.:

  `Bruch` Name einer Klasse
  `Kuerze()` Name einer Methode
  `Nenner` Name einer Eigenschaft

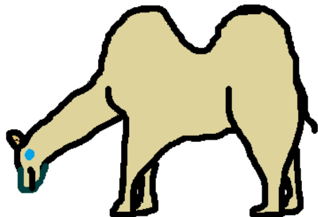
- Bei zusammengesetzten Namen beginnt jedes Wort mit einem Großbuchstaben (*Pascal Casing*), z.B.:

  `WriteLine()`

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:



Alternativ kann man bei zusammengesetzter Namen auch den Unterstrich zur Verbesserung der Lesbarkeit verwenden, was der folgende Methodenname demonstriert, das Visual Studio 2010 im `HalloWpf`-Projekt erstellt hat (siehe Abschnitt 2.2.1.3):

```
private void button1_Click(object sender, RoutedEventArgs e)
```

3.1.6 Übungsaufgaben zu Abschnitt 3.1

1) Welche **Main()**-Varianten 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_____
```

3.2 Ausgabe bei Konsolenanwendungen

Wie Sie bereits an einigen Beispielen beobachten konnten, lässt sich eine formatierte Konsolenausgabe in C# recht bequem über die Methode **Console.WriteLine()** erzeugen, z.B.:

```
using System;
  . . .
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.:

- Der Namensraum **System** muss am Beginn der Quelle per **using**-Direktive importiert werden, um die Klasse **Console** ohne Namensraum-Präfix 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):

- In obigem 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 in Abschnitt 3.2.2 näher ein.
- Oft reicht die im nächsten Abschnitt behandelte Ausgabe einer zusammengesetzten Zeichenfolge.

3.2.1 Ausgabe einer (zusammengesetzten) Zeichenfolge

Im `Hallo`-Beispiel haben wir der **WriteLine()**-Methode als einzigen Parameter eine Zeichenkette zur Ausgabe auf dem Bildschirm übergeben:

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

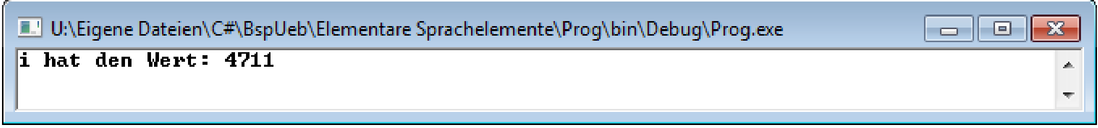
Übergebene Argumente anderen Typs werden vor der Ausgabe automatisch in eine Zeichenfolge konvertiert, z.B. der Wert einer ganzzahligen Variablen (siehe unten):

```
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 dann mit ihr zusammen ausgegeben wird:



Durch die bequeme Zeichenfolgenverkettung mit dem „+" – Operator und die automatische Konvertierung von beliebigen Datentypen in einer Zeichenfolge ist die **WriteLine()**-Variante mit unformatierter Ausgabe schon recht flexibel. Außerdem erlauben die folgenden **Escape-Sequenzen** (vgl. Abschnitt 3.3.9.4), die wie gewöhnliche Zeichen in die Ausgabezeichenfolge geschrieben werden, eine Gestaltung der Ausgabe:

\n	Zeilenwechsel (*new line*)
\t	Horizontaler Tabulator

Beispiel:

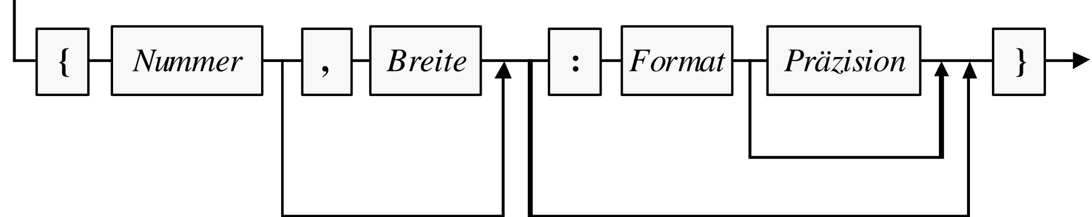
<table>
<tr><th>Quellcode-Fragment</th><th>Ausgabe</th></tr>
<tr><td><pre>int i = 47, j = 1771;
Console.WriteLine("Ausgabe:\n\t" + i + "\n\t" + j);</pre></td><td><pre>Ausgabe:
        47
        1771</pre></td></tr>
</table>

Noch mehr Gestaltungsmöglichkeiten bietet die im nächsten Abschnitt behandelte formatierte Ausgabe.

3.2.2 Formatierte Ausgabe

Bei der formatierten Ausgabe per **WriteLine()** wird als erster Parameter eine Zeichenfolge übergeben, die Platzhalter mit optionalen Formatierungsangaben für die restlichen, auf der Konsole auszugebenden Parameter enthält. Für einen Platzhalter ist folgende Syntax vorgeschrieben:

Platzhalter für die formatierte Ausgabe



Darin bedeuten:

Nummer	Fortlaufende Nummer des auszugebenden Arguments, bei Null 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)

Es werden u.a. folgende Formate unterstützt:

Format	**Beschreibung**	**Beispiele mit den Variablen** `int i = 41483; float f = 21.415926f;`	
		WriteLine-Parameterliste	Ausgabe
d, D	**Ganze Dezimalzahl**	`("{0,7:d}", i)`	`  41483`
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)`	`     41483` `  21,41593`

In der Formatierungszeichenfolge sind auch auszugebende gewöhnliche Zeichen und Escape-Sequenzen (vgl. Abschnitt 3.3.9.4) erlaubt:

\n	Zeilenwechsel (*new line*)
\t	Horizontaler Tabulator

Beispiel:

Quellcode-Fragment	Ausgabe

`int i = 47, j = 1771;`
`Console.WriteLine("Feste Breite:\n{0,8}\n{1,8}"+`
`    "\nTabulatoren:\n\t{2}\n\t{3}",i,j,i,j);`
	`Feste Breite:`
	`      47`
	`    1771`
	`Tabulatoren:`
	`        47`
	`        1771`

Auf eine Formatierungszeichenfolge mit *k* Platzhaltern müssen entsprechend viele Ausdrücke (z.B. Variablen) mit einem zum jeweiligen Platzhalterformat passenden Datentyp folgen.

Die beschriebenen Formatierungstechniken sind nicht nur bei Konsolenausgaben zu gebrauchen. Analog erstellte Zeichenfolgen kann man auch in eine Datei schreiben oder im Rahmen einer graphischen Benutzerschnittstelle präsentieren (siehe unten).

3.2.3 Übungsaufgaben zu Abschnitt 3.2

1) Wie ist das fehlerhafte „Rechenergebnis" in folgendem Programm zu erklären?

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>using System;
class Prog {
    static void Main() {
        Console.WriteLine("3,3 + 2 = " + 3.3 + 2);
    }
}</pre></td><td><pre>3,3 + 2 = 3,32</pre></td></tr>
</table>

Sorgen Sie mit einem Paar runder Klammern dafür, dass die folgende Ausgabe erscheint.

```
3,3 + 2 = 5,3
```

Verbringen Sie nicht zuviel Zeit mit der Aufgabe, weil wir die genauen technischen Hintergründe erst in Abschnitt 3.5.10 behandeln.

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)
```

3.3 Variablen und Datentypen

Während ein Programm läuft, müssen zahlreiche Informationen mehr oder weniger lange im Arbeitsspeicher des Rechners aufbewahrt und natürlich auch modifiziert werden, z.B.:

- Die Merkmalsausprägungen eines Objekts werden gespeichert, solange das Objekt existiert.
- Die zur Ausführung einer Methode benötigten Daten werden bis zum Ende des Methodenaufrufs aufbewahrt.

Zum Speichern eines Werts (z.B. einer Zahl) wird eine **Variable** verwendet, worunter Sie sich einen **benannten Speicherplatz von bestimmtem Datentyp** (z.B. Ganzzahl) vorstellen können.

Eine Variable erlaubt 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
   }
}
```

3.3.1 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:

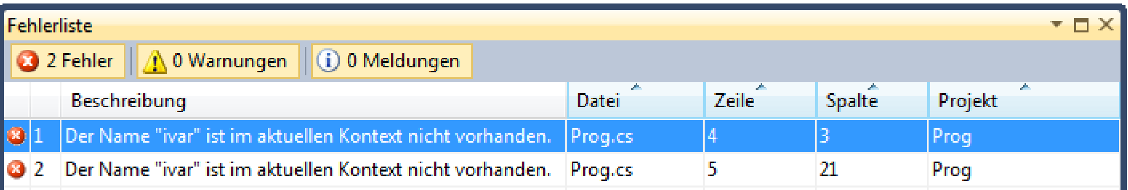
- Variablen müssen **explizit deklariert** werden.
  In obigem Beispiel wird die Variable `ivar` vom Typ **int** (für ganze Zahlen) deklariert. Wenn Sie versuchen, eine nicht deklarierte Variable zu verwenden, beschwert sich der Compiler, z.B.:

  ```
  Prog.cs(5,3): error CS0103: Der Name "ivar" ist im aktuellen Kontext
  nicht vorhanden.
  ```

  In der Visual C# 2010 Express Edition informiert schon der Quellcode-Editor über das Problem, z.B.:

  ```
  using System;
  class Prog {
      static void Main() {
          ivar = 4711;
                    Der Name "ivar" ist im aktuellen Kontext nicht vorhanden.
      }
  }
  ```

  Versucht man trotzdem eine Übersetzung, dann erscheint die Compiler-Reklamation in der **Fehlerliste**:



  Durch den Deklarationszwang werden z.B. Programmfehler wegen falsch geschriebener Variablennamen verhindert. Auch in VB.NET (Visual Basic .NET) besteht per Voreinstellung Deklarationszwang. Hier lässt er sich jedoch mit der (nicht empfehlenswerten) Compiler-Option **Option Explicit Off** aufheben, um das Verhalten vieler Skriptsprachen demonstrieren zu können, z.B.:

VB.NET - Quellcode	Ausgabe

`Option Explicit Off`
`Module Module1`
`  Sub Main()`
`    ii = 12       ' Verwendung ohne Deklaration`
`    ij = ii + 1   ' Tippfehler fällt nicht auf`
`    Console.WriteLine(ii)`
`  End Sub`
`End Module`
	12

- C# ist **streng und statisch typisiert**
  Für jede Variable ist bei der Deklaration ein fester (später nicht mehr änderbarer) **Datentyp** anzugeben. Er legt fest, …
  - welche Informationen (z.B. ganze Zahlen, rationale Zahlen, Zeichen) in der Variablen gespeichert werden können,
  - welche Operationen auf die Variable angewendet werden dürfen.

  Bereits der Compiler überwacht die korrekte Verwendung der Datentypen, so dass Fehler frühzeitig aufgedeckt werden (Typsicherheit) . Außerdem kann auf (zeitaufwändige) Typprüfungen zur Laufzeit verzichtet werden. In obigem C# - Programm wird die Variable `ivar` vom Typ **int** deklariert, der sich für ganze Zahlen im Bereich von -2147483648 bis 2147483647 eignet.
  ```
  int ivar;
  ```
  Anschließend erhält die Variable den **Initialisierungswert** 4711:
  ```
  ivar = 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 3.3.6).

  In C# 4.0 taucht mit dem Datentyp **dynamic** eine durch praktische Zwänge (z.B. bei der Kooperation mit typfreien Skriptsprachen) veranlasste Ausnahme von der strengen Typisierung auf. An Stelle des Compilers ist hier die CLR für die Typprüfung verantwortlich, was leicht zu Laufzeitfehlern führen kann. Dieser Datentyp sollte nur in begründeten Ausnahmefällen verwendet werden und kommt in unserem Kurs voraussichtlich nicht zum Einsatz.

Als wichtige Eigenschaften einer C# - Variablen halten wir fest:

- **Name**
  Es sind beliebige Bezeichner gemäß Abschnitt 3.1.5 erlaubt.
- **Datentyp**
  Damit sind festgelegt: Wertebereich, Speicherplatzbedarf und 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 Speicherregionen unterscheiden (*Stack* und *Heap*), weil Performanzgründe die gezielte Auswahl einer bestimmten Region nahelegen können.

3.3.2 Wert- und Referenztypen

Bei der objektorientierten Programmierung werden neben den traditionellen Variablen zur Aufbewahrung von Zahlen, Zeichen oder Wahrheitswerten auch Variablen benötigt, welche die Adresse eines Objekts aufnehmen und so die Kommunikation mit dem Objekt unterstützen. Wir unterscheiden also bei den Datentypen von Variablen zwei übergeordnete Kategorien:

- **Werttypen**
  Die traditionellen Datentypen werden in C# als *Werttypen* (*Value Types*) bezeichnet. Variablen mit einem Werttyp sind auch in C# unverzichtbar (z.B. als Felder von Klassen oder lokale Variablen). In der `Bruch`-Klassendefinition (siehe Abschnitt 1.1) 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 `nenner` auch noch einen Initialisierungswert erhält:
  ```
  int zaehler, nenner = 1;
  ```

Beim Feld `zaehler` wird auf die explizite Initialisierung verzichtet, so dass die automatische Null-Initialisierung von **int**-Feldern greift. In der `Bruch`-Methode `Kuerze()` tritt u.a. die lokale Variable `ggt` auf, die ebenfalls den Werttyp **int** besitzt:

```
int ggt = 0;
```

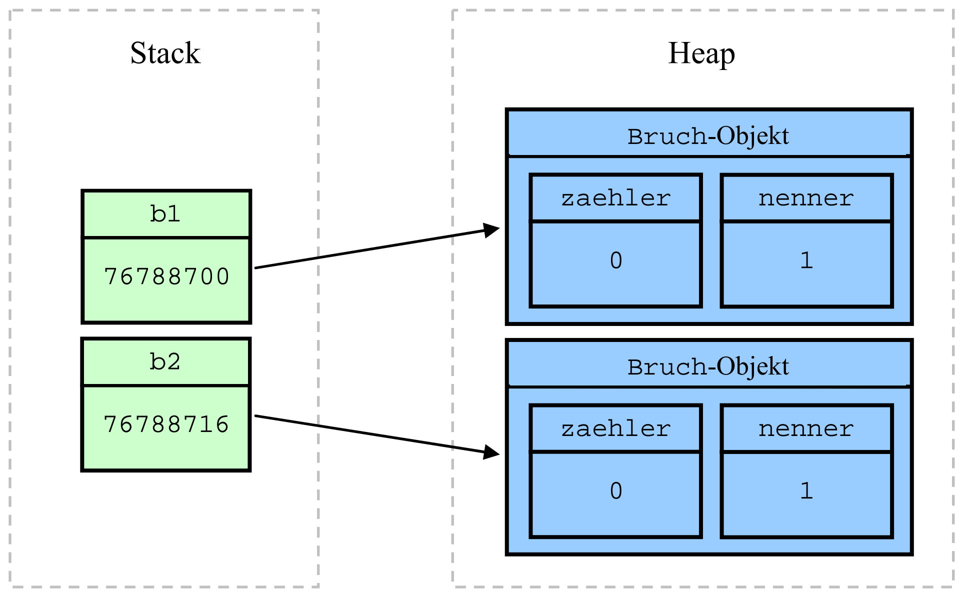
Wie Sie bereits wissen, ist bei *lokalen* (innerhalb einer Methode oder Eigenschaft deklarierten) Variablen vor dem ersten Lesezugriff eine Initialisierung erforderlich. Im Beispiel findet diese gleich bei der Deklaration statt (siehe Abschnitt 3.3.6).

- **Referenztypen**
  Eine Variable mit Referenztyp dient dazu, die **Speicheradresse eines Objekts** aus einer bestimmten Klasse aufnehmen. 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 Kommunikation mit Objekten.

  Man kann jede Klasse (aus der FCL übernommen oder selbst definiert) als Datentyp verwenden, also Referenzvariablen dieses Typs deklarieren. In der **Main**()-Methode der 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 unten) neu erzeugtes `Bruch`-Objekt. Daraus resultiert im programmeigenen Speicher folgende Situation:



  Das von `b1` referenzierte `Bruch`-Objekt wurde bei einem konkreten Programmlauf von der CLR an der Speicheradresse 76788700 (Hexadezimal: 0x0493b3dc) untergebracht. Wir plagen uns nicht mit solchen Adressen, sondern sprechen die dort abgelegten Objekte über Referenzvariablen an, wie 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 Instanzvariablen (Felder) 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 eine Instanzvariable mit Referenztyp erhalten.
- Eine Referenzvariable dient zur Aufnahme einer Objektadresse.
  So kann z.B. eine Variable vom Datentyp `Bruch` die Adresse eines `Bruch`-Objekts aufnehmen und die Kommunikation mit diesem Objekt unterstützen. 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, wenn im gesamten Programm keine einzige Referenz (Kommunikationsmöglichkeit) mehr vorhanden ist.

3.3.3 Klassifikation der Variablen nach Zuordnung

Nach der Zuordnung zu einer *Methode*, 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. auf einen Block innerhalb der Methode (siehe Abschnitt 3.3.7).

  Solange eine Methode ausgeführt wird, befinden sich ihre Variablen in einem Speicherbereich, den man als *Stack* (dt.: *Stapel*) bezeichnet. Die obige Abbildung 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)**
  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`. Vielleicht wäre es terminlogisch klarer, von *Objektvariablen* zu sprechen. Wir bleiben aber bei der dominanten Bezeichnung, die z.B. auch von Microsoft (2010, S. 93) in der *C# 4.0 Language Specification* verwendet wird.

  Solange ein Objekt existiert, befinden es sich mit all seine Instanzvariablen in einem Speicherbereich, den man als *Heap* (dt.: *Haufen*) bezeichnet.
- **Klassenvariablen (statische Felder)**
  Diese Variablen beziehen sich auf eine Klasse, nicht auf einzelne Instanzen. Z.B. hält man oft in einer Klassenvariablen fest, wie viele Objekte der Klasse 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.

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.

In Abschnitt 3 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, so genannte *globale* Variablen außerhalb von Klassen zu deklarieren.

3.3.4 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 Wertebereich und Speicherplatzbedarf 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. 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;`	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. Beispiel: `float p = 1252.61f;` **float**-Literale (siehe unten) benötigen den Suffix **f** (oder **F**).	Minimum: $-3.4028235 \cdot 10^{38}$ Maximum: $3.4028235 \cdot 10^{38}$ Kleinster Betrag > 0: $1.4 \cdot 10^{-45}$	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 15-16 signifikanten Dezimalstellen. Beispiel: `double p=113445.626535891;`	Minimum: $-1{,}7976931348623157 \cdot 10^{308}$ Maximum: $1{,}7976931348623157 \cdot 10^{308}$ Kleinster Betrag > 0: $4{,}9 \cdot 10^{-324}$	64 1 für das Vorz., 11 für den Expon., 52 für die Mantisse
decimal	Variablen vom Typ **decimal** speichern Gleitkommazahlen mit 28 bis 29 Dezimalstellen exakt und eignen sich besonders für die **Finanzmathematik**, wo Rundungsfehler zu vermeiden sind. Beispiel: `decimal p = 2344.2554634m;` **decimal**-Literale (siehe unten) benötigen den 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 Betrag > 0: 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.

Typ	Beschreibung	Werte	Bits
char	Variablen vom Typ **char** speichern ein Unicode-Zeichen. Im Speicher landet aber nicht die Gestalt eines Zeichens, sondern seine Nummer im Zeichensatz. Daher zählt **char** zu den ganzzahligen (integralen) Datentypen. Beispiel: `char zeichen = 'j';` **char** – Literale (s.u.) sind mit *einfachen* Anführungszeichen einzurahmen.	Unicode-Zeichen Tabellen mit allen Unicode-Zeichen sind z.B. auf der folgenden Webseite http://www.unicode.org/charts/ des Unicode-Konsortiums zu finden.	16
bool	Variablen vom Typ **bool** speichern Wahrheitswerte. Beispiel: `bool cond = false;`	**true**, **false**	1

Als *Gleitkommazahl* (synonym: *Gleitpunkt*- oder *Fließkommazahl*, engl.: *floating point number* bezeichnet man ein Tripel aus

(Vorzeichen, Mantisse, Exponent)

zur approximativen Darstellung einer reellen Zahl in der EDV. Das genaue Übersetzungsverfahren muss durch eine Norm geregelt werden (z.B. IEEE 754). Die drei Komponenten werden separat gespeichert und ergeben nach folgender Formel den dargestellten Wert, wobei ***b*** für die Basis eines Zahlensystems steht (meist verwendet: 2 oder 10):

$$\textbf{Wert} = \textbf{Vorzeichen} \cdot \textbf{Mantisse} \cdot \boldsymbol{b}^{\textbf{Exponent}}$$

Bei dieser von Konrad Zuse entwickelten Darstellungstechnik[1] resultiert im Vergleich zur Festkommadarstellung bei gleichem Speicherplatzbedarf ein erheblich größerer Wertebereich. Während die Mantisse für die Genauigkeit sorgt, speichert der Exponentialfaktor die Größenordnung, z.B.:[2]

$$-0{,}0000001252612 = (-1) \cdot 1{,}252612 \cdot 10^{-7}$$
$$1252612000000000 = (1) \cdot 1{,}252612 \cdot 10^{15}$$

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 Eindeutigkeit gesorgt.

Weil der verfügbare Speicher für Mantisse und Exponent begrenzt ist (siehe obige Tabelle), bilden die Gleitkommazahlen nur eine endliche (aber für praktische Zwecke ausreichende) Teilmenge der reellen Zahlen. Zur Verarbeitung von Gleitkommazahlen wurde die *Gleitkommaarithmetik* entwickelt, normiert und zur Verbesserung der Verarbeitungsgeschwindigkeit teilweise sogar in Computer-Hardware realisiert. Nähere Informationen über die Darstellung von Gleitkommazahlen im Arbeitsspeicher eines Computers folgen für speziell interessierte Leser im Abschnitt 3.3.5.

3.3.5 Vertiefung: Darstellung von Gleitkommazahlen im Arbeitsspeicher des Computers

Die als *Vertiefung* bezeichneten Abschnitte können beim ersten Lesen des Manuskripts gefahrlos übersprungen werden. Sie enthalten interessante Details, über die man sich irgendwann im Verlauf der Programmierkarriere informieren sollte. Im Kurskontext dienen sie auch als Zeitvertreib für

---

1 Quelle: http://www.calsky.com/lexikon/de/txt/g/gl/gleitkommazahl.php

2 Diese Beispiele orientieren sich der Einfachheit halber nur sinngemäß an der Norm IEEE 754. Dort wird beim Exponenten die Basis 2 verwendet (siehe Abschnitt 3.3.5.1).

Teilnehmer mit Vorkenntnissen, die sich eventuell (z.B. im Abschnitt über elementare Sprachelemente) etwas langweilen.

3.3.5.1 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

<pre>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);
      Console.ReadLine();
   }
}</pre>
	<pre> 1,2999999523162800
	  1,2500000000000000
	
	  1,3000000000000000</pre>

Die Zahl 1,3 kann im **float**-Format (sieben signifikante Dezimalstellen) *nicht* exakt gespeichert werden. Um dies zu demonstrieren, wird ein **float**-Wert (erzwungen per Literal-Suffix **f**) in einer **double**-Variablen abgelegt und dann mit 16 Dezimalstellen ausgegeben. Demgegenüber wird die Zahl 1,25 im **float**-Format fehlerfrei gespeichert. Mit einer **float**-Variablen lässt sich das Problem nicht demonstrieren, weil die Ungenauigkeit bei der Ausgabe von der CLR weggerundet wird.

Diese Ergebnisse sind durch das Speichern der Zahlen im **binären Gleitkommaformat** nach der Norm **IEEE-754** zu erklären, wobei jede Zahl als Produkt aus drei getrennt zu speichernden Faktoren dargestellt wird:

$$\textbf{Vorzeichen} \cdot \textbf{Mantisse} \cdot \textbf{2}^{\textbf{Exponent}}$$

Im ersten Bit einer **float**- und **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 (siehe Abschnitt 3.6.2). 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 Null, landet der Wert

$$01111111_2 = 127$$

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. Weil die führende Eins der normalisierten Mantisse *nicht* abgespeichert wird (*hidden bit*), 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. Das

i-te Mantissen-Bit (von links nach rechts mit *Eins* beginnend nummeriert) hat die Wertigkeit 2^{-i}, so dass sich der *dezimale* Mantissenwert folgendermaßen ergibt:

$$1+m \quad \text{mit} \quad m = \sum_{i=1}^{23 \text{ bzw. } 52} b_i 2^{-i}, \; b_i \in \{0,1\}$$

Eine **float**- bzw. **double**-Variable mit dem Vorzeichen *v* (Null oder Eins), 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 in Abschnitt 3.3.4 z.B.

$$1{,}4 \cdot 10^{-45}$$

als betragsmäßig kleinsten **float**-Wert nennt, obwohl der minimale Exponent nach obigen Überlegungen -126 beträgt, was zum (gerundeten) dezimalen Exponentialfaktor

$$1{,}2 \cdot 10^{-38}$$

führt. Dahinter steckt die *denormalisierte* Gleitkommadarstellung, die als Ergänzung zur bisher beschriebenen normalisierten Darstellung eingeführt wurde, um eine bessere Annäherung an die Zahl Null zu erreichen. Alle Exponenten-Bits sind auf Null gesetzt, und dem Exponentialfaktor wird der feste Wert 2^{-126} (**float**) bzw. 2^{-1022} (**double**) zugeordnet. Die Mantissen-Bits haben dieselbe 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* (Null oder Eins), mit komplett auf Null 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:

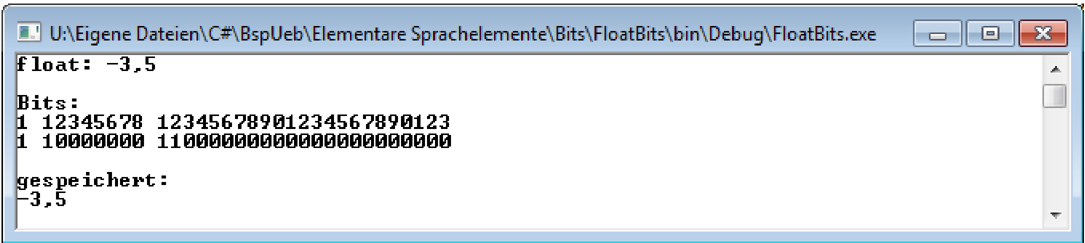
Wert	float-Darstellung (denormalisiert)		
	Vorz.	Exponent	Mantisse
$0{,}0 = (-1)^0 \cdot 2^{-126} \cdot 0$	0	00000000	00000000000000000000000
$-5{,}877472 \cdot 10^{-39} = (-1)^1 \cdot 2^{-126} \cdot 2^{-1}$	1	00000000	10000000000000000000000
$1{,}401298 \cdot 10^{-45} = (-1)^0 \cdot 2^{-126} \cdot 2^{-23}$	0	00000000	00000000000000000000001

Weil die Mantissen-Bits auch zur Darstellung der Größenordnung verwendet werden, schwindet die relative Genauigkeit mit der Annäherung an die Null.

Visual C# - 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 im Text nicht beschrieben, weil die erforderlichen Techniken (speziell die Nachbildung von C++ - Unions in C# mit Hilfe von Attributen, siehe Abschnitt 11) in .NET - Anwendungen ansonsten keine Rolle spielen. Eine Beispielausgabe des Programms `FloatBits`:



3.3.5.2 *Dezimale Gleitkommadarstellung*

Neben den eben beschriebenen *binären* Gleitkommatypen (mit der Basis Zwei in Mantisse und Exponent) bietet C# auch den *dezimalen* Gleitkommatyp **decimal**, dessen Speicherorganisation die Basis Zehn 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 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:

$$(-1)^v \cdot m \cdot 10^{-e}$$

Durch die 96-Manitissen-Bits einer **decimal**-Variablen lassen sich alle natürlichen Zahlen mit max. 28 Stellen darstellen:

$$\underbrace{9999999999999999999999999999}_{28\ \text{Stellen}} < 2^{96}-1 < \underbrace{99999999999999999999999999999}_{29\ \text{Stellen}}$$

Folglich kann jede Dezimalzahl mit maximal 28 Stellen (Vorkommastellen + Nachkommastellen) exakt in einer **decimal**-Variablen gespeichert werden.[1]

1 Von den insgesamt belegten 128 Bit bleiben also einige ungenutzt.

Wie in Abschnitt 3.3.5.1 zu sehen war, wird z.B. die Zahl 1,3 im *binären* Gleitkommaformat durch eine prinzipiell unendliche Serie von Brüchen mit einer Zweierpotenz im Nenner dargestellt, so dass 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 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:

Quellcode	Ausgabe

`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`

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 (sogar 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,

  $$1300000000 \quad - \quad \sum_{i=1}^{1000000000} 1{,}3$$

  zu berechnen, ergaben sich für die Datentypen **double** und **decimal** folgende Genauigkeits- und Laufzeitunterschiede:[2]

  ```
  double:
    Abweichung:   -24,5162951946259
    Benöt. Zeit:  3797,8515 Millisek.

  decimal:
    Abweichung:   0,0
    Benöt. Zeit:  28424,8047 Millisek.
  ```

  Die gut bezahlten Verantwortlichen bei den deutschen Landesbanken und anderen Instituten, 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 Sielcasino) 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, so dass man bei finanzmathematischen Aufgaben trotz des erhöhten Zeitaufwands (im Beispiel: Faktor 10) den Datentyp **decimal** verwenden sollte.

---

[1] Die Beschränkung des Exponenten auf Werte von 0 bis 28 und die unvollständige Verwendung der 128 im Speicher belegten Bits scheint trotzdem nicht schlüssig, weil z.B. die Zahl 10^{-30} trotz der Beschränkung auf 96 Mantissenbits fehlerfrei dargestellt werden könnte. Damit wäre eine bessere Annäherung an die Null möglich.

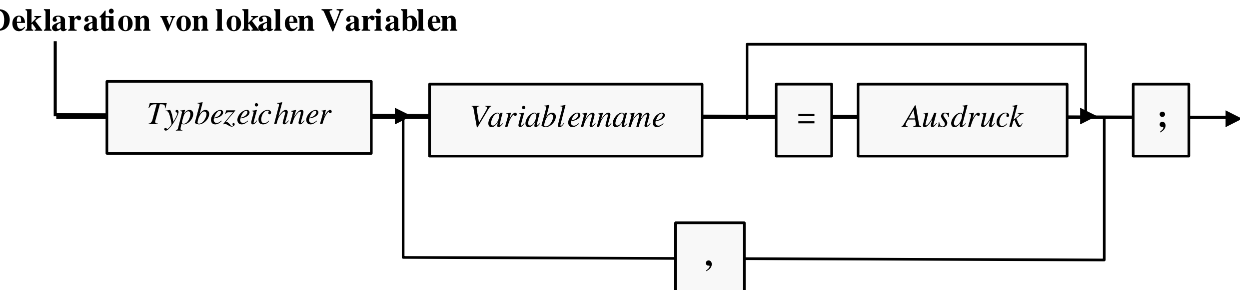
[2] Gemessen auf einem PC mit Intel-CPU Core i3 550 unter Windows 7-64

- Keine Unterstützung für Sonderfälle wie +/- Unendlich und NaN (***Not a Number***) (vgl. Abschnitt 3.6)

3.3.6 Variablendeklaration, Initialisierung und Wertzuweisung

In C#-Programmen muss jede Variable vor ihrer ersten Verwendung deklariert[1] werden. Dabei sind auf jeden Fall der Name und der Datentyp anzugeben, wie das Syntaxdiagramm zur **Variablendeklarationsanweisung** zeigt:

Deklaration von lokalen Variablen



Als Datentypen kommen in Frage (vgl. Abschnitt 3.3.2):

- Werttypen, z.B.
  ```
  int i;
  ```
- Referenztypen, also Klassen (aus der FCL oder selbst definiert), z.B.
  ```
  Bruch b1;
  ```

Wir betrachten vorläufig nur *lokale* Variablen, die innerhalb einer Methode oder Eigenschaft existieren. Ihre Deklaration darf im Methodenquellcode an beliebiger Stelle *vor* der ersten Verwendung erscheinen. Es ist üblich, ihre Namen mit einem Kleinbuchstaben beginnen zu lassen (vgl. Abschnitt 3.1.5).

Neu deklarierte Variablen kann man optional auch gleich **initialisieren**, also auf einen gewünschten Wert setzen, z.B.:

```
int i = 4711;
Bruch b1 = new Bruch();
```

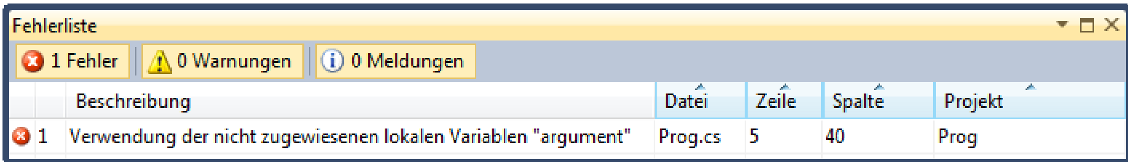
Im zweiten Beispiel wird per **new**-Operator ein `Bruch`-Objekt erzeugt und dessen Adresse in die neue Referenzvariable `b1` geschrieben. Mit der Objektkreation und auch mit der Konstruktion von gültigen *Ausdrücken*, die einen Wert von passendem Datentyp liefern müssen, werden wir uns noch ausführlich beschäftigen.

Weil *lokale* Variablen *nicht* automatisch initialisiert werden, muss man ihnen unbedingt vor dem ersten lesenden Zugriff einen Wert zuweisen. Auch im Umgang des Compilers mit uninitialisierten lokalen Variablen zeigt sich das Bemühen der C# - Designer um robuste Programme. Während C++ - Compiler in der Regel nur warnen, produziert der C# - Compiler eine Fehlermeldung und erstellt *keinen* Zwischencode. Dieses Verhalten wird durch folgendes Programm demonstriert:

[1] Während in der Programmiersprache C++ die beiden Begriffe *Deklaration* und *Definition* verschiedene Bedeutungen haben, werden sie im Zusammenhang mit den meisten anderen Programmiersprachen (so auch bei C#) synonym verwendet. In diesem Manuskript wird im Zusammenhang mit Variablen bevorzugt von *Deklarationen*, im Zusammenhang mit Klassen und Methoden stets von *Definitionen* gesprochen.

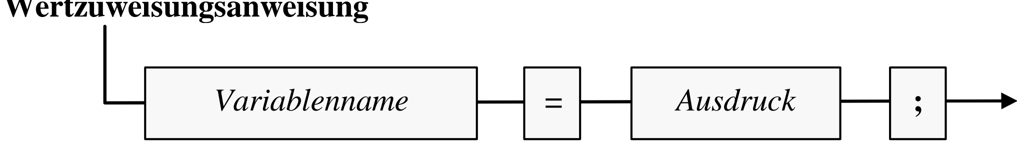
```
using System;
class Prog {
  static void Main() {
    int argument;
    Console.WriteLine("Argument =  {0}", argument);
  }
}
```

Der Compiler meint dazu:



Weil *Instanz- und Klassenvariablen* automatisch mit dem Standardwert ihres Typs 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 Pogrammablauf zu verändern, verwendet man eine **Wertzuweisung**, die zu den einfachsten und am häufigsten benötigten Anweisungen gehört:



Beispiel: `ggt = az;`
Durch diese Wertzuweisungsanweisung aus der `Kuerze()` - Methode unserer `Bruch`-Klasse (siehe Abschnitt 1.1) erhält die **int**-Variable `ggt` den Wert der **int**-Variablen `az`.

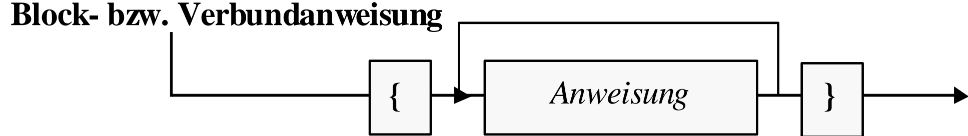
Es wird sich bald herausstellen, dass auch ein Ausdruck stets einen Datentyp hat. Bei der Wertzuweisung muss dieser Typ natürlich kompatibel zum Datentyp der Variablen sein.

U.a. haben Sie mittlerweile zwei Sorten von C# - Anweisungen kennen gelernt:

- Variablendeklaration
- Wertzuweisung

3.3.7 Blöcke und Deklarationsbereiche 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 Methodenrumpfes 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.

Unter den Anweisungen eines Blocks dürfen sich selbstverständlich auch wiederum Blockanweisungen befinden. Einfacher ausgedrückt: Blöcke dürfen geschachtelt werden.

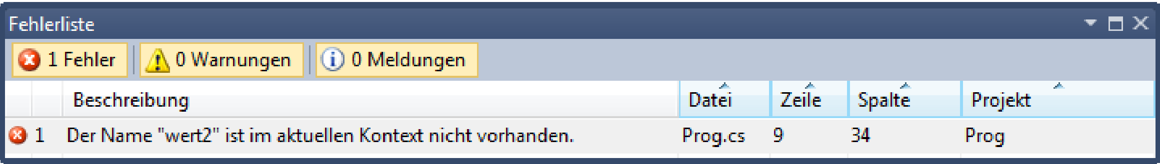
Oft treten Blöcke als Bestandteil von Bedingungen oder Schleifen (siehe Abschnitt 3.7) auf, z.B. in der Methode `Kuerze()` der Klasse `Bruch` (siehe Abschnitt 1.1):

```
public void Kuerze() {
   if (zaehler != 0) {
      int ggt = 0;
      int az = Math.Abs(zaehler);
      int an = Math.Abs(nenner);
         . . .
         . . .
      zaehler /= ggt;
      nenner  /= ggt;
   } else
      nenner = 1;
}
```

Anweisungsblöcke haben einen wichtigen Effekt auf die Gültigkeit der darin deklarierten Variablen: Eine lokale Variable ist verfügbar von der deklarierenden Zeile bis zur schließenden Klammer des innersten Blockes. Nur in diesem **Deklarations- bzw. Sichtbarkeitsbereich** kann sie über ihren Namen angesprochen werden, so dass der Compiler das Übersetzen des folgenden (weitgehend sinnfreien) Beispielprogramms

```
using System;
class Prog {
  static void Main(){
  int wert1 = 1;
  if (wert1 == 1) {
    int wert2 = 2;
    Console.WriteLine("Wert2 = " + wert2);
  }
  Console.WriteLine("Wert2 = " + wert2);
  }
}
```

mit einer Fehlermeldung ablehnt:



Bei hierarchisch geschachtelten Blöcken ist es in C# *nicht* erlaubt, auf mehreren Stufen Variablen mit identischem Namen zu deklarieren. Diese kaum sinnvolle Option ist in der Programmiersprache C++ vorhanden und erlaubt dort Fehler, die schwer aufzuspüren sind. In C# gehören die eingeschachtelten Blöcke zum Deklarationsbereich der umgebenden Blocks.

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);
}
```

Bei den Quellcode-Editoren unsere Entwicklungsumgebungen 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:

Einfügemarke des Editors vor der Startklammer

```
do {
    if (az == an)
        ggt = az;
    else
        if (az > an)
            az = az - an;
        else
            an = an - az;
} while (ggt == 0);
```

hervorgehobene Endklammer

3.3.8 Konstanten

Für die in einem Programm benötigten festen Werte (z.B. Mehrwertsteuersatz) sollte man in der Regel jeweils eine *Konstante* definieren, 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.

Beispiel:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>using System;
class Prog {
  static void Main() {
    const double MWST = 1.19;
    double netto = 28.10, brutto;
    brutto = netto * MWST;
    Console.WriteLine("Brutto: {0:f2}", brutto);
  }
}</pre></td><td><pre>Brutto: 33,44</pre></td></tr>
</table>

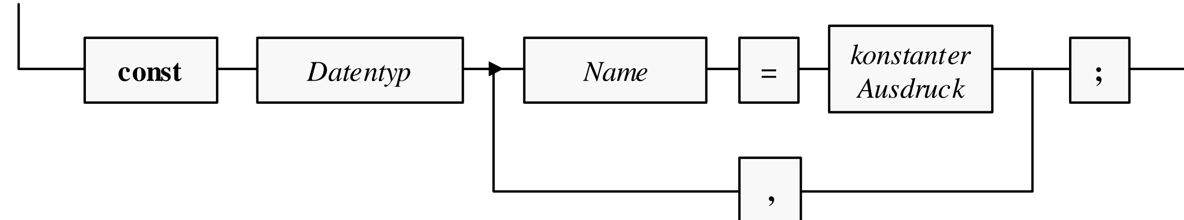
Im Vergleich zu einer Variablen weist eine Konstante folgende Besonderheiten auf:

- Ihre Deklaration beginnt mit dem Modifikator **const** und *muss* eine Initialisierung enthalten, wobei ein *konstanter* Ausdruck zu verwenden ist, der nur Literale (siehe Abschnitt 3.3.9) und Konstanten enthält.
- Ihr Wert kann im Programm nicht geändert werden.

Programmierer verwenden traditionell im Namen einer Konstanten ausschließlich Großbuchstaben und verbessern bei einem Mehrwortnamen die Lesbarkeit durch trennende Unterstriche (z.B.: `MAXIMALE_QUOTE`). Dies ist aber nicht vorgeschrieben, und in der FCL scheint Microsoft folgende Konvention anzuwenden:

- Namen von Konstanten werden komplett groß geschrieben, wenn sie aus maximal zwei Zeichen bestehen, z.B. **Math.PI** (Kreiszahl π, siehe unten).
- Bei längeren Namen wird der Pascal-Stil verwendet (Anfangsbuchstaben aller Wörter groß), z.B. **Double.MaxValue** (größter Wert des Typs **double**).

Das Syntaxdiagramm zur Deklaration einer konstanten lokalen Variablen:



Neben lokalen Variablen können auch Felder einer Klasse als konstant deklariert werden (siehe Abschnitt 4.2.5).

3.3.9 Literale

Die im Programmcode auftauchenden expliziten Werte bezeichnet man als *Literale*. Wie Sie aus dem Abschnitt 3.3.8 wissen, ist es oft sinnvoll, Literale innerhalb von Konstanten-Deklarationen zu verwenden, z.B.:

```
const double MWST = 1.19;
```

Aber auch andere Einsatzorte (z.B. Initialisierungen von Variablen, Parameterwerte) kommen in Frage.

Auch die Literale besitzen in C# stets einen **Datentyp**, wobei einige Regeln zu beachten sind.

In diesem Abschnitt haben manche Passagen Nachschlage-Charakter, so dass man beim ersten Lesen nicht jedes Detail aufnehmen muss bzw. kann.

3.3.9.1 Ganzzahlliterale

Ganzzahlliterale können im dezimalen oder im hexadezimalen Zahlensystem (mit der Basis 16 und den Ziffern 0, 1, …, 9, A, B, C, D, E, F) geschrieben werden, wobei der hexadezimale Fall durch das Präfix **0x** oder **0X** zu kennzeichnen ist. 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:

$$11_{\text{Hex}} = 1 \cdot 16^1 + 1 \cdot 16^0 = 1 \cdot 16 + 1 \cdot 1 = 17$$

Vermutlich fragen Sie sich, wozu man sich mit dem Hexadezimalsystem plagen sollte. Gelegentlich ist ein ganzzahliger Wert (z.B.als Methodenparameter) anzugeben, den man (z.B. aus einer Tabelle) nur in hexadezimler Darstellung kennt. In diesem Fall spart man sich durch Verwendung dieser Darstellung die Wandlung in das Dezimalsystem.

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 folgender Serie, der seinen Wert aufnehmen kann:

  int, **uint**, **long**, **ulong**

  Beispiele:

Literale	Typ
`2147483647, -21`	**int**
`2147483648`	**uint**
`-2147483649, 9223372036854775807`	**long**
`9223372036854775808`	**ulong**

- Ein Ganzzahlliteral mit Suffix **u** oder **U** (*unsigned*, ohne Vorzeichen) hat den ersten Typ aus folgender Serie, der seinen Wert aufnehmen kann:

  uint, **ulong**

  Beispiele:

Literale	Typ
`2147483647U`	**uint**
`9223372036854775808u`	**ulong**

- Ein Ganzzahlliteral mit Suffix **l** oder **L** (*Long*) hat den ersten Typ aus folgender 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. Aufgrund der in C# vorhandnen automatischen Typanpassungen wird das Suffix **L** sehr selten benötigt.

- 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 unsere Entwicklungsumgebung, z.B.

```
using System;
class Prog {
    static void Main() {
    Console.WriteLine(18446744073709551616);
  }
}
```

  Die integrale Konstante ist zu groß.

  Wird der Compiler trotzdem mit der Übersetzung beauftragt, produziert er eine Fehlermeldung:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`    Console.WriteLine(18446744073709551616);`
`  }`
`}`
	`Prog.cs(4,23): error CS1021: Die integrale Konstante ist zu groß`

Weil der Compiler ganze Zahlen gut kennt und einige Intelligenz bei der Verwendung von Ganzzahlliteralen zeigt, sind nach meinem Eindruck die eben erwähnten Literale überflüssig. Man kann z.B. einer vorzeichenlosen Ganzzahlvariablen ein Literal mit geeignetem Wert zuweisen, obwohl dieses Literal formal zu einem vorzeichenbehafteten Typ gehört:

```
uint i = 88;
```

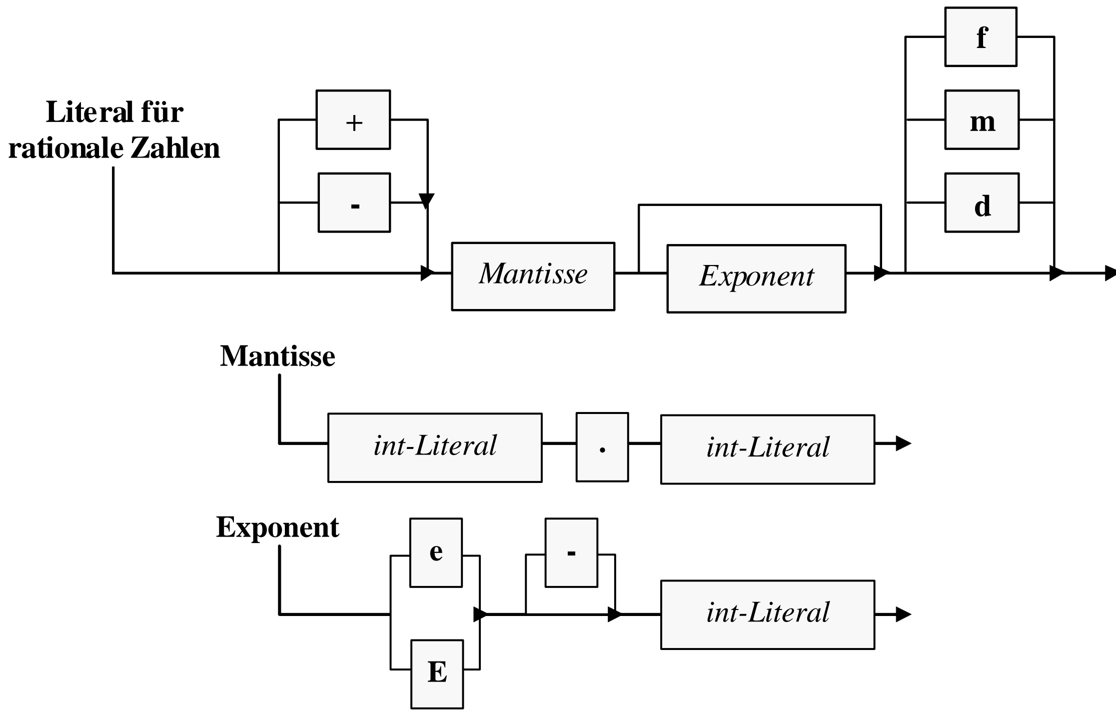
struct System.Int32
Stellt eine 32-Bit-Ganzzahl mit Vorzeichen dar.

3.3.9.2 *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 (kaum jemals erforderlichen) Suffix **D** oder **d** wird der Datentyp **double** „optisch betont“, z.B.:
  `1d`

Hinsichtlich der Schreibweise von Gleitkommaliteralen bietet C# etliche Möglichkeiten, von denen die wichtigsten in den folgenden Syntaxdiagrammen dargestellt werden:



Die in der Mantisse und im Exponenten auftretenden Ganzzahlliterale müssen das dezimale Zahlensystem verwenden und den Datentyp **int** besitzen, so dass die in Abschnitt 3.3.9.1 beschriebenen Präfixe (**0x**, **0X**) und Suffixe (z.B. **L**, **U**) verboten sind. Die Exponenten werden natürlich zur Basis Zehn verstanden.

Beispiele:

```
0.45875e-20
9.78f
2279800223423485.45m
```

Der Compiler achtet bei Wertzuweisungen streng auf die Typkompatibilität. Z.B. führt die folgende Deklarationsanweisung:

```
float p = 1.25;
```

zu der Fehlermeldung:

```
Prog.cs(4,19): error CS0664: Literale des Typs "Double" können nicht im-
plizit in den Typ "float" konvertiert werden. Verwenden Sie ein F-Suffix,
um ein Literal mit diesem Typ zu erstellen.
```

3.3.9.3 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);`
`  }`
`}`
	U:\Eigene Dateien\C#\BspUeb\...
	False

3.3.9.4 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 so genannte *Escape-Sequenz*:
- **Escape-Sequenzen**
  Hier dürfen einem einleitenden Rückwärts-Schrägstrich u.a. folgen:
  - Ein Steuerzeichen, z.B.:

Neue Zeile	`\n`
Horizontaler Tabulator	`\t`
Alarmton	`\a`

  - Einfaches oder doppeltes Hochkomma sowie der Rückwärts-Schrägstrich:
    `\'`
    `\"`
    `\\`

  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 Bit, also im Bereich von 0 bis 2^{16}-1 = 65535) in hexadezimaler, vierstelliger Schreibeweise (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 = '\u03b1';
```

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).

3.3.9.5 Zeichenkettenliterale

Zeichenkettenliterale 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";
```

Zeichenkettenliterale 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";
```

Aus einem ganz speziellen und einmaligen Grund ist ausnahmsweise die Groß-/Kleinschreibung irrelevant.

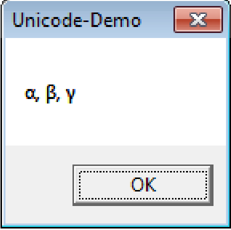
Während ein **char**-Literal stets *genau ein* Zeichen enthält, kann ein Zeichenkettenliteral aus beliebig vielen Zeichen bestehen oder auch leer sein, z.B.:

```
string name = "";
```

Das folgende Programm enthält einen Aufruf der statischen **Show**()-Methode der Klasse **MessageBox** aus dem Namensraum **System.Windows** zur Anzeige eines Zeichenkettenliterals, das drei Unicode-Escape-Sequenzen enthält:[1]

```
class Prog {
   static void Main() {
      System.Windows.MessageBox.Show("\u03b1, \u03b2, \u03b3", "Unicode-Demo");
   }
}
```

Beim Programmstart erscheint die folgende Dialogbox:



Um die besondere Bedeutung des Rückwärts-Schrägstrichs in einer Zeichenkette abzuschalten und den Verdopplungsaufwand zu sparen, stellt man das @-Zeichen voran, z.B.:

```
Console.WriteLine(@"Pfad: C:\Program Files\Mapp\bin");
```

---

[1] Beim Übersetzen des Programm sind Verweise auf die folgenden Assemblies erforderlich:
PresentationCore, **PresentationFramework**, **System**, **System.Xaml**, **WindowsBase**
Wie man im Visual Studio Verweise setzt, ist in Abschnitt 2.2.4 beschrieben.

3.3.10 Übungsaufgaben zu Abschnitt 3.3

1) Wieso klagt der Compiler über ein unbekanntes Symbol, obwohl die Variable `i` deklariert worden ist?

<table>
<tr><th>Quellcode</th><th>Fehlermeldung</th></tr>
<tr><td><pre>class Prog {
  static void Main() {
    {
        int i = 2;
    }
    System.Console.WriteLine(i);
  }
}</pre></td><td><pre>Prog.cs(6,28): error CS0103: Der Name i ist im
aktuellen Kontext nicht vorhanden.</pre></td></tr>
</table>

2) Beseitigen Sie bitte alle Fehler in folgendem 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 folgende Ausgabe produziert:

```
Dies ist ein Zeichenkettenliteral:
        "Hallo"
```

4) In folgendem 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:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class Prog {
  static void Main() {
    char zeichen = 'c';
    int nummer = zeichen;
    System.Console.WriteLine("zeichen = " + zeichen +
                             "\nnummer = " + nummer);
  }
}</pre></td><td><pre>zeichen = c
nummer = 99</pre></td></tr>
</table>

Warum hat der ansonsten sehr pingelige C# - Compiler nichts dagegen, einer **int**-Variablen den Wert einer **char**-Variaben zu übergeben? Wie kann man das Zeichen 'c' über eine Unicode-Escape-Sequenz ansprechen?

3.4 Einfache Techniken für Benutzereingaben

3.4.1 Via Konsole

In der `Frage()`-Methode der Klasse `Bruch` aus dem Einleitungsbeispiel (siehe Abschnitt 1.1) wird folgende Anweisung genutzt, um einen **int**-Wert von der Konsole entgegen zu nehmen:

```
zaehler = Convert.ToInt32(Console.ReadLine());
```

Mit 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 der statischen 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 hier eine Verschachtelung zweier Methodenaufrufe vor, die bei Programmierern der kompakten Schreibweise wegen sehr beliebt ist. Ein etwas umständliches, aber für Anfänger leichter verständliches Äquivalent zur obigen Anweisung könnte lauten:

```
string eingabe;
eingabe = Console.ReadLine();
zaehler = Convert.ToInt32(eingabe);
```

Die 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.:

<table>
<tr><th>Quellcode</th><th>Ein- und Ausgabe</th></tr>
<tr><td><pre>
using System;
class Prog {
  static void Main() {
    Console.Write("Ihre Lieblingszahl? ");
    int zahl = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Verstanden: " + zahl);
  }
}
</pre></td><td><pre>
Ihre Lieblingszahl? drei

Unbehandelte Ausnahme:
System.FormatException: Die
Eingabezeichenfolge hat das falsche
Format.
</pre></td></tr>
</table>

Geschieht dies nach einem Start aus Visual C# heraus mit der Taste **F5** oder mit dem Schalter (also im so genannten Debug-Modus), führt der Ausnahmefehler zu folgender Anzeige:

```
using System;
class Prog {
    static void Main() {
        Console.Write("Ihre Lieblingszahl? ");
        int zahl = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Verstanden: " + zahl);
    }
}
```

FormatException wurde nicht behandelt.

Die Eingabezeichenfolge hat das falsche Format.

Hinweise zur Fehlerbehebung:

Stellen Sie sicher, dass die Methodenargumente das richtige Format aufweisen.

Bearbeiten Sie beim Konvertieren einer Zeichenfolge in DateTime diese so, dass das Datum vor dem Einfügen aller Variablen in das DateTime-Objekt übernommen wird.

Für diese Ausnahme die allgemeine Hilfe aufrufen.

Weitere Hilfeinformationen online suchen...

Aktionen:

Details anzeigen...

Ausnahmedetail in die Zwischenablage kopieren

In dieser Situation lässt sich das havarierte und noch „baumelnde“ Programm mit dem Menübefehl

Debuggen > Debugging beenden

oder mit der Tastenkombination **Umschalt+F5** stoppen.

Um derartige Probleme zu verhindern, sind Programmiertechniken erforderlich, mit denen wir uns momentan noch nicht beschäftigen wollen, z.B. die folgende Ausnahmebehandlung:

Quellcode

```
using System;
class Prog {
  static void Main() {
    int zahl;
    try {
      Console.Write("Ihre Lieblingszahl? ");
      zahl = Convert.ToInt32(Console.ReadLine());
      Console.WriteLine("Verstanden: " + zahl);
    }
    catch {
      Console.WriteLine("Falsche Eingabe!");
    }
  }
}
```

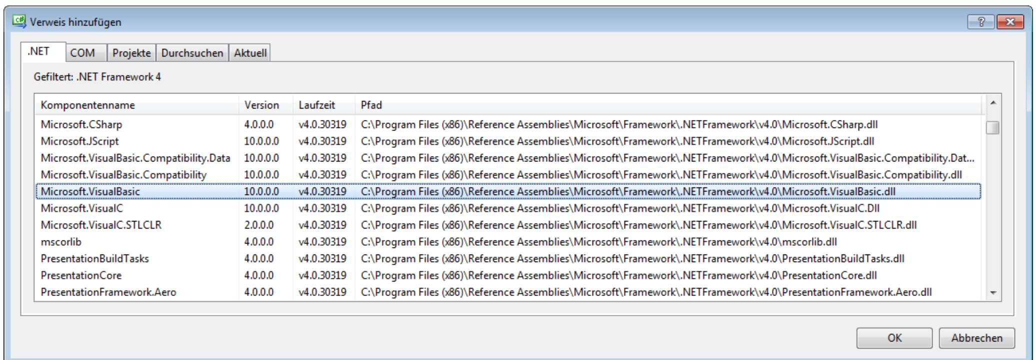
Ein- und Ausgabe

```
Ihre Lieblingszahl? drei
Falsche Eingabe!
```

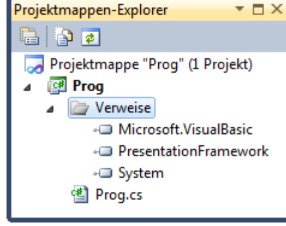
In den Übungs- bzw. Demoprogrammen verwenden wir der Einfachheit halber ungesicherte **ToInt32** – Aufrufe bzw. analoge Varianten für andere Datentypen.

3.4.2 Via InputBox

Wer Anwendungen mit grafikorientierter Benutzerinteraktion erstellen möchte, ohne die Komplexität einer WPF-Anwendung in Kauf nehmen zu müssen, kann statt der **Console**-Methode **ReadLine()** z.B. die statische Methode **InputBox()** der Klasse **Interaction** aus dem Namensraum **Microsoft.VisualBasic** benutzen. Die Klasse **Interaction** ist als Migrationshilfe für Visual Basic 6 - Programmierer gedacht, kann aber auch in C# - Programmen genutzt werden. Dazu muss dem Compiler das implementierende und im **GAC** (Global Assembly Cache) installierte Assembly **microsoft.visualbasic.dll** bekannt gemacht werden. Wie Sie aus Abschnitt 2.2.4 bereits wissen, kann man unseren Entwicklungsumgebungen sehr bequem erklären, welche Assembly-Referenzen beim Übersetzen eines Projekts erforderlich sind. Man wählt im **Projektmappen-Explorer** aus dem Kontextmenü zum Projektnamen die Option **Verweis hinzufügen** und kann dann in folgender Dialogbox das gesuchte Assembly lokalisieren und im markierten Zustand per **OK** in die Verweisliste des Projekts aufnehmen:



Anschließend wird die gewählte Referenz im Projektmappen-Explorer angezeigt:

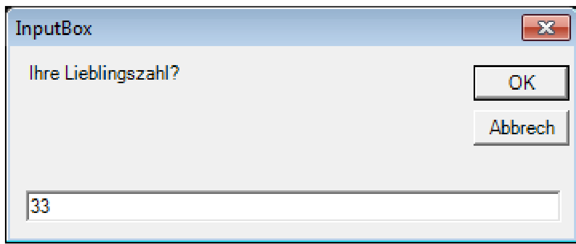


Für die folgende GUI-Alternative zum Beispielprogramm in Abschnitt 3.4.1 wird außerdem eine Referenz auf Assembly **PresentationFramework.dll** gesetzt, weil die Klasse **MessageBox** zum Einsatz kommt:

Aus dem entsteht folgende GUI-Variante:

```
using System;
using System.Windows;
using Microsoft.VisualBasic;

class InputBox {
   static void Main() {
      int zahl = Convert.ToInt32(
        Interaction.InputBox("Ihre Lieblingszahl?", "InputBox", "", -1, -1)
      );
      MessageBox.Show("Verstanden: " + zahl);
   }
}
```

Ein- und Ausgabe werden per Dialogbox erledigt:

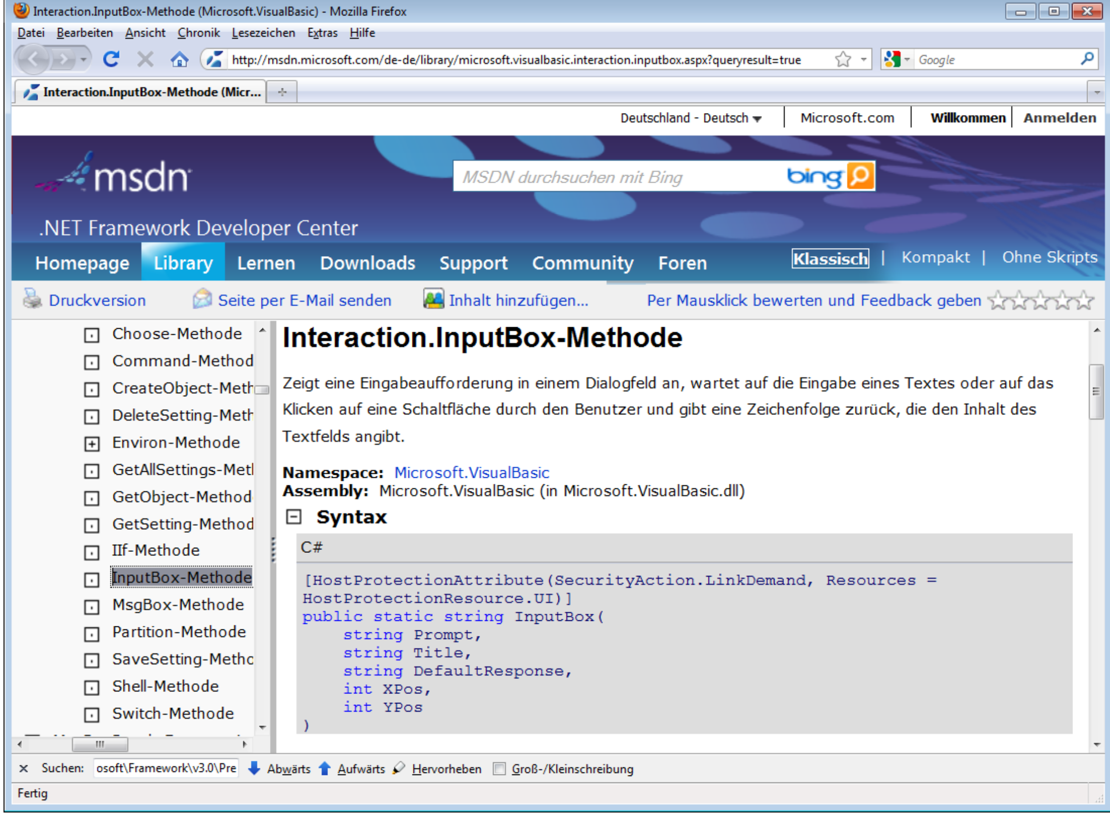


Der folgende Schönheitsfehler

Abbrech

ist wohl entstanden, weil der verantwortliche Programmierer nicht mit dem höheren Platzbedarf der deutschen Beschriftung im Vergleich zur englischen Variante gerechnet hat (vgl. Abschnitt 2.2.1.3).

Über die Parameter (Argumente) und den Rückgabewert des **Inputbox**()-Methodenaufrufs kann man sich z.B. über die Hilfefunktion der Visual C# 2010 Express Edition informieren. Bei markiertem Methodennamen ruft ein Druck auf die Taste **F1** die folgende Webseite herbei:



Zwar sieht die Ein- bzw. Ausgabe per GUI attraktiver aus, jedoch lohnt sich der erhöhte Aufwand bei Demo- bzw. Übungsprogrammen zu elementaren Sprachelementen kaum, so dass wir in der Regel darauf verzichten werden.

Die **Convert**-Methode **ToInt32** reagiert natürlich auch im optisch aufgewerteten Programm auf ungeschickte Benutzereingaben mit einer Ausnahme. Später werden wir das Problem mit einer professionellen Ausnahmebehandlung lösen.

3.5 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 3.5 werden wir Ausdrücke als wichtige Bestandteile von C# - Anweisungen recht detailliert betrachten. Dabei lernen Sie elementare Datenverarbeitungs-Möglichkeiten kennen, die von so genannten Operatoren mit ihren Argumenten veranstaltet werden, z.B. von den arithmetischen Operatoren (+, -, *, /) für die Grundrechenarten. Am Ende des Abschnitts kann immerhin schon das Programmieren eines Währungskonverters als Übungsaufgabe gestellt werden. Allzu große Begeisterung wird wohl trotzdem nicht aufkommen, doch ein sicherer Umgang mit Operatoren und Ausdrücken ist unabdingbare Voraussetzung für das erfolgreiche Implementieren von Methoden und Eigenschaften. Hier werden Algorithmen bzw. Handlungskompetenzen von Klassen oder Objekten realisiert.

Während die Variablen zur *Speicherung* von Werten dienen, geht es bei den **Operatoren** darum, aus vorhandenen Variableninhalten 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 folgender Wertzuweisung:

Operator

```
az = az - an;
```

Ausdruck

Durch diese Anweisung aus der `Kuerze()`-Methode unserer `Bruch`-Klasse (siehe Abschnitt 1.1) 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.

Man kann einen Ausdruck als eine *temporäre* Variable mit einem **Datentyp** und einem **Wert** auffassen.

Schon bei einem Literal, einer Variablen oder einem Methodenaufruf haben wir es mit einem Ausdruck zu tun.[1]

Beispiel: `1.5`
Dies ist ein Ausdruck mit dem Typ **double** und dem Wert 1,5.

Mit Hilfe diverser Operatoren entsteht ein komplexerer Ausdruck, wobei sein Typ und sein Wert von den Argumenten und den Operatoren abhängen.

Beispiele: `2 * 1.5`
Hier resultiert der **double**-Wert 3,0.
`2 > 1.5`

[1] Besteht ein Ausdruck aus einem Methodenaufruf mit dem Pseudorückgabetyp **void**, dann liegt allerdings *kein* Wert vor.

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.

Beispiel:
```
int i = 12;
int j = i++;
```
Im Beispiel hat der Ausdruck `i++` den Typ **int** und den Wert 12. Außerdem wird die Variable `i` beim Auswerten des Ausdrucks durch den Postinkrementoperator auf den neuen Wert 13 gesetzt.

Die meisten Operatoren verarbeiten *zwei* Operanden (Argumente) und heißen daher **zweistellig** bzw. **binär**.

Beispiel: `a + b`
Der Additionsoperator wird mit einem „+“ bezeichnet und erwartet zwei numerische Argumente.

Manche Operatoren begnügen sich mit *einem* Argument und heißen daher **einstellig** bzw. **unär**.

Beispiel: `!cond`
Der Negationsoperator wird mit einem „!“ bezeichnet und erwartet *ein* Argument mit dem Typ **bool**.

Wir werden auch noch einen *dreistelligen* Operator kennen lernen.

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.

3.5.1 Arithmetische Operatoren

Weil die arithmetischen Operatoren für die vertrauten Grundrechenarten der Schulmathematik 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 in der Regel von den Argumenten. Wenn man es ganz genau nimmt, was aus didaktischen Gründen bei einem Einsteiger nicht unbedingt sinnvoll ist, kommt es auf die vorhandenen Überladungen des (+)-Operators an. Gemäß Microsofts Sprachspezifikation (2010, S. 192f) kennt C# sieben verschiedene numerische (+)-Operatoren, wobei die beiden Argumente einen identischen Datentyp aus der folgenden Liste haben müssen und das Ergebnis vom selben Typ ist:

int, **uint**, **long**, **ulong**, **float**, **double**, **decimal**

Wenn bei einem Argument oder bei beiden Argumenten der Datentyp ungeeignet ist, findet nach Möglichkeit eine automatische (implizite) Typanpassung „nach oben“ statt (vgl. Abschnitt 3.5.7). Bevor z.B. ein **int**-Argument zu einem **double**-Wert addiert werden kann, muss es in den Typ **double** konvertiert werden. Ist keine automatische Typanpassung möglich, beschwert sich der Compiler, z.B.:

```
doub = 2.0 + dec;
```
Der Operator "+" kann nicht auf Operanden vom Typ "double" und "decimal" angewendet werden.

Sind z.B. beide Argumente einer (+)-Operation vom Typ **byte**, werden sie vor der Addition in den Typ **int** gewandelt, den folglich auch die Summe hat. Der Compiler lehnt es ab, diesen **int**-Wert in eine **byte**-Variable zu schreiben:

```
byte b1 = 1, b2 = 2;
b1 = b1 + b2;
```

(lokale Variable) byte b1

Fehler:
Der Typ "int" kann nicht implizit in "byte" konvertiert werden. Es ist bereits eine explizite Konvertierung vorhanden. (Möglicherweise fehlt eine Umwandlung.)

Es hängt vom Datentyp 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 Über- bzw. Unterlauf (siehe Abschnitt 3.6) werden Sie noch weitere Unterschiede zwischen Ganzzahl- und Gleitkommaarithmetik kennen lernen.

In der folgenden Tabelle mit allen arithmetischen Operatoren stehen *Num*, *Num1* und *Num2* für Ausdrücke mit numerischem Typ, *Var* vertritt eine numerische *Variable*:

Operator	Bedeutung	Beispiel	
		Quellcode-Fragment	**Ausgabe**
-*Var*	**Vorzeichenumkehr**	`int i = 2;` `Console.WriteLine(-i);`	 `-2`
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 % 5.4);`	`4` `-2,8`

++*Var*
--*Var*
	Präinkrement bzw. -dekrement
	Als Argument ist nur eine Variable erlaubt.
	++*Var* liefert *Var* + 1, erhöht *Var* um 1
	--*Var* liefert *Var* – 1, reduziert *Var* um 1
		`int i = 4;`
		`double a = 1.2;`
		`Console.WriteLine(++i + "\n" +`
		`                  --a);`
			
			`5`
			`0,2`

<table>
<tr><th rowspan="2">Operator</th><th rowspan="2">Bedeutung</th><th colspan="2">Beispiel</th></tr>
<tr><th>Quellcode-Fragment</th><th>Ausgabe</th></tr>
<tr><td>Var++
Var--</td><td>Postinkrement bzw. -dekrement
Als Argument ist nur eine Variable erlaubt.
Var++ liefert Var
erhöht Var um 1
Var-- liefert Var
reduziert Var um 1</td><td><code>int i = 4;</code>
<code>Console.WriteLine(i++ + "\n" +</code>
<code>i);</code></td><td>4
5</td></tr>
</table>

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.
- Der Wert des Arguments wird verändert.
  Wegen dieses **Nebeneffekts** sind Prä- und Postinkrement- bzw. -dekrementausdrücke im Unterschied zu sonstigen arithmetischen Ausdrücken bereits vollständige *Anweisungen* (vgl. Abschnitt 3.7.1), wenn man ein Semikolon dahinter setzt:

Quellcode	Ausgabe

<pre>using System;
class Prog {
  static void Main() {
    int i = 12;
    i++;
    Console.WriteLine(i);
  }
}</pre>
	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 Modulo-Operator gibt es viele sinnvolle Anwendungen, z.B.:

- Man kann für eine ganze Zahl bequem feststelen, ob sie gerade (durch Zwei teilbar) ist. Dazu prüft man, ob der Rest aus der Division durch Zwei gleich Null ist:

Quellcode-Fragment	Ausgabe
`int i = 19;` `Console.WriteLine( (i % 2 == 0) );`	False

- Man kann bei einer Gleitkommazahl den gebrochenen Anteil ermitteln bzw. abspalten:

Quellcode-Fragment	Ausgabe

`double a = 7.1248239;`
`a = a - a % 1.0;`
`Console.WriteLine("{0,10:f5}", a);`
	7,00000

3.5.2 Methodenaufrufe

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 in Abschnitt 3.4.1 vorgestellte Methode **Convert.ToInt32()** liefert z.B. einen **int**-Wert, sofern die als Parameter übergebene Zeichenfolge als ganze Zahl im **int**-Wertebereich (siehe Tabelle in Abschnitt 3.3.4) interpretierbar ist. Bei der Methodendefinition ist der Datentyp der Rückgabe anzugeben (siehe Syntaxdiagramm in Abschnitt 3.1.2.2).
- Liefert eine Methode dem Aufrufer *kein* Ergebnis, ist in der Definition der Pseudo-Rückgabetyp **void** anzugeben.
- Neben der Wertrückgabe hat ein Methodenaufruf oft weitere Effekte, z.B. auf die Merkmalsausprägungen des 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 Paar runder Klammern um die Parameterliste um einen Operator handelt. Microsoft spricht in der C# Referenz vom **() Operator**, ohne die Leser allzu sehr mit Formalismen und Abstraktion zu quälen.[1] Jedenfalls sollten Sie sich nicht darüber wundern, dass der Methodenaufruf in Tabellen mit den C# - Operatoren auftaucht und dort eine (eine ziemlich hohe) Auswertungspriorität erhält (vgl. Abschnitt 3.5.10).

Bei passendem Rückgabetyp darf ein Methodenaufruf auch als Argument für komplexere Ausdrücke oder für übergeordnete Methodenaufrufe verwendet werden (siehe Abschnitt 4.3.1.2). 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 3.7), was Sie z.B. bei den zahlreichen Einsätzen der Methode **Console.WriteLine()** in unseren Beispielprogrammen bereits 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 statischen Methoden der Klasse **Math** im Namensraum **System** zur Verfügung (siehe FCL-Dokumentation). Im folgenden Programm wird die Methode **Pow()** zur Berechnung der allgemeinen Potenzfunktion (b^e) genutzt:

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 Anfänger leichter nachvollziehbares Äquivalent zum obigen **WriteLine()**-Aufruf könnte z.B. so aussehen:

```
double d;
d = Math.Pow(2.0, 3.0);
Console.WriteLine(d);
```

[1] Siehe z.B. http://msdn.microsoft.com/en-us/library/0z4503sa.aspx

3.5.3 Vergleichsoperatoren

Durch Anwendung eines *Vergleichsoperators* auf zwei komparable (miteinander vergleichbare) Argumentausdrücke entsteht ein **Vergleich**. Dies ist ein einfacher **logischer Ausdruck** (vgl. Abschnitt 3.5.5), kann dementsprechend die booleschen Werte **true** (wahr) und **false** (falsch) annehmen und eignet sich dazu, eine *Bedingung* zu formulieren. 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 komparable Ausdrücke
  Hier ein Beispiel für *fehlende* Vergleichbarkeit:

  ```
  Console.WriteLine(2.4 == "123");
  ```
  Der Operator "==" kann nicht auf Operanden vom Typ "double" und "string" angewendet werden.

- *Num1* und *Num2* für numerische Ausdrücke

Operator	Bedeutung	Beispiel	
		Programmfragment	Ausgabe
Expr1 == *Expr2*	**Gleichheit**	`Console.WriteLine(2 == 3);`	`False`
Expr1 != *Expr2*	**Ungleichheit**	`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-Meldung leicht zu beseitigen ist, sondern es kann auch ein mehr oder weniger unangenehmer Semantikfehler resultieren, also ein irreguläres Verhalten des Programms (vgl. Abschnitt 2.1.4 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

<pre>using System;
class Prog{
  static void Main(){
    int i = 1;
    Console.WriteLine(i == 2);
    Console.WriteLine(i);
  }
}</pre>
	<pre>False
	1</pre>

Durch Weglassen eines Gleichheitszeichens wird aus dem Vergleich jedoch ein *Wertzuweisungsausdruck* (siehe Abschnitt 3.5.8) mit dem Typ **int** und dem Wert 2:

[1] Wir wissen schon aus Abschnitt 3.2.1, dass **WriteLine()** einen Ausdruck mit beliebigem Typ verarbeiten kann, wobei automatisch eine Zeichenfolgen-Repräsentation erstellt wird.

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 natürlich auch den Wert der Variablen `i`, was im weiteren Verlauf eines größeren Programms recht unangenehm werden kann.

3.5.4 Vertiefung: Gleitkommawerte vergleichen

Bei den *binären* Gleitkommatypen (**float** und **double**) muss man beim Identitätstest unbedingt technisch bedingte Abweichungen von der reinen Mathematik berücksichtigen, 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 naive Vergleich

```
10.0 - 9.9 == 0.1
```

führt trotz Datentyp **double** (mit mindestens 15 signifikanten Dezimalstellen) zum Ergebnis `false`. Wenn man die in Abschnitt 3.3.5.1 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 9,9) nicht exakt gespeichert werden können:[1]

Quellcode	Ausgabe

`using System;`
`class Prog {`
`   static void Main() {`
`      double d = 9.9f;`
`      Console.WriteLine("{0,20:f16}", d);`
`   }`
`}`
	`9,8999996185302700`

Beim Speichern der Zahlen 9,9 und 0,1 treten *verschiedene* Abweichungen von den korrekten Werten auf, so dass im Berechnungsergebnis 10,0 – 9,9 ein anderer Fehler steckt als im Speicherabbild

[1] Um dies zu demonstrieren, wird ein **float**-Wert (erzwungen per Literal-Suffix **f**) in einer **double**-Variablen abgelegt und dann mit 16 Dezimalstellen ausgegeben. Mit einer **float**-Variablen lässt sich das Problem nicht demonstrieren, weil die Ungenauigkeit bei der Ausgabe von der CLR weggerundet wird.

der Zahl 0,1. Weil die Vergleichspartner nicht Bit für Bit identisch sind, meldet der Identitätsoperator ein **false**.

Für Anwendungen im Bereich der Finanzmathematik wurde schon in Abschnitt 3.3.5 der dezimale Gleitkommadatentyp **decimal** vorgeschlagen. 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 ihres relativ kleinen Wertebereichs nicht für alle Anwendungen.

Sollen **double**-Werte mit möglichst großer Präzision verglichen werden, kann eine an der Rechen- bzw. Speichergenauigkeit orientierte **Unterschiedlichkeitsschwelle** verwendet werden. Nach diesem Vorschlag werden zwei *normalisierte* (also insbesondere von Null verschiedene) **double**-Werte d_1 und d_2 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 Wahl der Bezugsgröße d_1 oder d_2 für den Nenner ist beliebig. Um das Verfahren vollständig festzulegen, wird die Verwendung der betragsmäßig größeren Zahl vorgeschlagen.

Ein Begriff der numerischen Identität 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 häufig anzutreffenden Vorschlag, $|d_1 - d_2|$ 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 **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 $t\ 10^k$ enthalten:

$$d_1 = (t + e_1)\ 10^k \quad \text{und} \quad d_1 = (t + e_2)\ 10^k$$

Für den Betrag des technisch bedingten relativen Fehlers gilt bei normalisierten Werten (mit einer Mantisse im Intervall [1, 2)) mit der oberen Schranke ε für den absoluten Mantissenfehler einer einzelnen **double**-Zahl die folgende Abschätzung:

$$\left|\frac{d_1 - d_2}{d_1}\right| = \left|\frac{e_1 - e_2}{t + e_1}\right| \le \frac{|e_1| + |e_2|}{|t + e_1|} \le \frac{2 \cdot \varepsilon}{|t + e_1|} \le 2 \cdot \varepsilon \quad (\text{wegen } |t + e_1| \ge 1)$$

Bei normalisierten **double**-Werten (mit 52 Mantissen-Bits) ist aufgrund der begrenzten Speichergenauigkeit mit Fehlern im Bereich des Abstands zwischen zwei benachbarten Mantissenwerten zu rechnen:

$$2^{-52} \approx 2{,}2 \cdot 10^{-16}$$

Die vorgeschlagene Schwelle $1{,}0 \cdot 10^{-14}$ berücksichtigt über den Speicherfehler hinaus auch eingeflossene Rechnungsungenauigkeiten. Mit welcher Fehlerkumulation bzw. -verstärkung zu rechnen ist, hängt vom konkreten Algorithmus ab, so dass die Unterschiedlichkeitsschwelle eventuell angehoben werden muss. Immerhin hängt sie (anders als bei einem Kriterium auf Basis der einfachen Differenz $|d_1 - d_2|$) nicht von der Größenordnung der Zahlen ab.

An der vorgeschlagenen Identitätsprüfung mit Hilfe einer Schwelle für den relativen Abweichungsbetrag ist u.a. zu bemängeln, …

- dass noch Feinarbeit erforderlich ist, um eine Division durch Null zu verhindern,
- dass eine Verallgemeinerung für die mit geringerer Genauigkeit gespeicherten *denormalisierte* Werte (Betrag kleiner als 2^{-1022} beim Typ **double**, siehe Abschnitt 3.3.5.1) benötigt wird

Dass die definierte Indifferenzrelation nicht transitiv ist, muss hingenommen werden.

Die besprochenen Genauigkeitsprobleme sind auch bei den Grenzfällen von *einseitigen* 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 Unterschiedschwelle dürften die technischen Beschränkungen der Gleitkommatypen keine große Rolle mehr spielen. Präzisere Aussagen zur Computer-Arithmetik finden sich z.B. bei Müller (2004) oder Strey (2003).

3.5.5 Logische Operatoren

Durch Anwendung der logischen Operatoren 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**

<table>
<tr><th>Argument 1
La1</th><th>Argument 2
La2</th><th>Logisches UND
La1 && La2
La1 & La2</th><th>Logisches ODER
La1 || La2
La1 | La2</th><th>Exklusives ODER
La1 ^ La2</th></tr>
<tr><td>true</td><td>true</td><td>true</td><td>true</td><td>false</td></tr>
<tr><td>true</td><td>false</td><td>false</td><td>true</td><td>true</td></tr>
<tr><td>false</td><td>true</td><td>false</td><td>true</td><td>true</td></tr>
<tr><td>false</td><td>false</td><td>false</td><td>false</td><td>false</td></tr>
</table>

In der folgenden Tabelle gibt es noch wichtige Erläuterungen und Beispiele:

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`

Operator	Bedeutung	Beispiel	
		Programmfragment	Ausgabe

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 UND-Operatoren **&&** und **&** bzw. zwischen den beiden ODER-Operatoren || und | ist für Einsteiger vielleicht etwas unklar, 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.:

```
a == b & i++ > 3
```

Hier erhöht der Postinkrementoperator beim Auswerten des rechten UND-Arguments den Wert der Variablen `i`. Eine solche Auswertung wird jedoch in der folgenden Variante des Beispiels unterlassen, wenn bereits nach Auswertung des linken UND-Arguments das Gesamtergebnis **false** feststeht:

```
a == b && i++ > 3
```

Mit der Entscheidung, grundsätzlich die unbedingte Operatorvariante zu verwenden, nimmt man (mehr oder weniger relevante) Leistungseinbußen in Kauf. Eher empfehlenswert ist der Verzicht auf Nebeneffekt-Konstruktionen im Zusammenhang mit bedingt arbeitenden Operatoren.

Wie der Tabelle auf Seite 122 zu entnehmen ist, unterscheiden sich die beiden UND-Operatoren **&&** und **&** bzw. die beiden ODER-Operatoren || und | auch hinsichtlich der Auswertungspriorität.

Um die Verwirrung noch ein wenig zu steigern, werden die Zeichen **&** und | auch für *bitorientierte* Operatoren verwendet (siehe Abschnitt 3.5.6). Diese Operatoren erwarten zwei *integrale* Argumente (z.B. Datentyp **int**), während die logischen Operatoren den Datentyp **bool** voraussetzen. Folglich kann der Compiler mühelos erkennen, ob ein logischer oder ein bitorientierter Operator gemeint ist.

3.5.6 Vertiefung: Bitorientierte Operatoren

Über unseren momentanen Bedarf hinausgehend bietet C# einige Operatoren zur bitweisen Analyse und Manipulation von Variableninhalten. Statt einer systematischen Darstellung der verschiedenen Operatoren beschränken wir uns auf ein Beispielprogramm, das zudem nützliche Einblicke in die Speicherung von **char**-Daten im Computerspeicher vermitteln kann. 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-Kodierung zu einem vom Benutzer erfragten Zeichen. Dabei kommt die statische Methode **ToChar**() der Klasse **Convert** aus dem Namensraum **System** zum Einsatz. Außerdem kommt mit der **for**-Schleife eine Wiederholungsanweisung zum Einsatz, die erst in Abschnitt 3.7.3.1 offiziell vorgestellt wird. Im Beispiel startet die **int**-wertige 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`):

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>
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--) {
      if ((1 << i & cbit) != 0)
        Console.Write('1');
      else
        Console.Write('0');
    }
  }
}
</pre></td><td><pre>
Zeichen: x
Unicode: 0000000001111000
</pre></td></tr>
</table>

Der **Links-Shift-Operator** << im Ausdruck:

```
1 << i
```

verschiebt die Bits in der binären Repräsentation der Ganzzahl Eins um *i* Stellen nach links. Von den 32 Bit, die ein **int**-Wert insgesamt belegt (siehe Abschnitt 3.3.4), 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 leider in doppelter Bedeutung verwendet: Wenn beide Argumente vom Typ **bool** sind, wird **&** als *logischer* Operator interpretiert (siehe Abschnitt 3.5.5). 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 Eins enthält, wenn *beide* Argumentmuster an dieser Stelle eine Eins besitzen. Hat bei einem Programmablauf die **char**-Variable `cbit` vom Benutzer den Wert `'x'` erhalten, ist das Unicode-Bitmuster dieses Zeichens

```
0000000001111000
```

beteiligt, und `1 << i & cbit` liefert z.B. bei `i` = 6 das Muster:

```
0000000001000000
```

Das von `1 << i & cbit` erzeugte Bitmuster hat den Typ **int** und kann daher mit der Null verglichen werden:

```
(1 << i & cbit) != 0
```

Dieser logische Ausdruck wird bei einem Schleifendurchgang genau dann wahr, wenn das zum aktuellen im `i`-Wert korrespondierende Bit in der Binärdarstellung des untersuchten Zeichens den Wert Eins hat.

3.5.7 Typumwandlung (Casting) bei elementaren Datentypen

Beim Auswerten des Ausdrucks

```
2.3/7
```

trifft der Divisionsoperator auf ein **double**- und ein **int**-Argument, so dass sich der Compiler zwischen der Ganzzahl- und der Gleitkommaarithmetik entscheiden muss. Er wählt die zweite Alternative und nimmt für das **int**-Argument automatisch eine Wandlung in den Datentyp **double** vor.

In vergleichbaren Situationen (z.B. bei Wertzuweisungen) kommt es automatisch zu den folgenden **erweiternden Konvertierungen**:

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

```
using System;
class Prog {
   static void Main() {
      long i = 9223372036854775313;
      double d = i;
      Console.WriteLine(i);
      Console.WriteLine("{0,20:f2}", d);
   }
}
```

Ausgabe

```
9223372036854775313
9223372036854780000,00
```

Eine Abweichung von 4687 (z.B. Euro oder Meter) kann durchaus unerfreuliche Konsequenzen haben.

Weil eine **char**-Variable die Unicode-Nummer eines Zeichens speichert, macht die Konvertierung in numerische Typen kein Problem, z.B.:

Quellcode

```
class Prog {
  static void Main() {
    System.Console.WriteLine("x/2 \t= " + 'x' / 2);
    System.Console.WriteLine("x*0,27 \t= " + 'x' * 0.27);
  }
}
```

Ausgabe

```
x/2     = 60
x*0,27  = 32,4
```

Gelegentlich gibt es gute Gründe, über den **Casting-Operator** eine *explizite* Typumwandlung zu erzwingen. Im folgenden Programm wird z.B. mit

```
(int)'x'
```

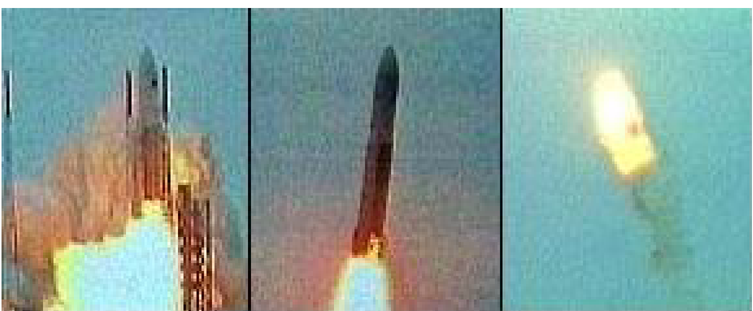
die **int**-erpretation des (aus Abschnitt 3.5.6 bekannten) Bitmusters zum kleinen „x“ ausgegeben, damit Sie nachvollziehen können, warum das letzte Programm beim „Halbieren“ dieses Zeichens auf den Wert 60 kam:

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, z.B. weil bei einem Methodenaufruf ein ganzzahliger Datentyp benötigt wird. Dabei werden die Nachkommastellen abgeschnitten. Soll stattdessen ein Runden stattfinden, addiert man vor der Typkonvertierung 0,5 zum Gleitkommawert.

Es ist auf jeden Fall zu beachten, dass eine **einschränkende Konvertierung** stattfindet, so dass die zu erwartenden Gleitkommazahlen im Wertebereich des Ganzzahltyps liegen müssen. 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 3.6.1). So soll die Explosion der europäischen Weltraumrakete Ariane-5 am 4. Juni 1996 (Schaden: ca. 500 Millionen Dollar)



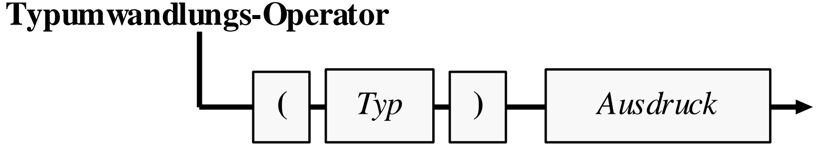
durch die Konvertierung eines **double**-Werts (mögliches Maximum: $1{,}7976931348623157 \cdot 10^{308}$) in einen **short**-Wert (mögliches Maximum: $2^{15}-1 = 32767$) verursacht worden sein.

Um die Nachkommastellen in einem Gleitkommawert zu entfernen oder zu extrahieren, kann man auch den Modulo-Operator verwenden (vgl. Abschnitt 3.5.1) und dabei die Wertebereichsproblematik vermeiden, z.B.:

Quellcode-Fragment	Ausgabe

`double a = 214748364852.13;`
`double b = a - a%1;`
`Console.WriteLine("{0}\n{1}",a,b);`
	214748364852,13
	214748364852

Die C# - Syntax zur expliziten Typumwandlung:



3.5.8 Zuweisungsoperatoren

Bei den ersten Erläuterungen zur Wertzuweisung (vgl. Abschnitt 3.3.6) blieb aus didaktischen Gründen unerwähnt, dass eine Wertzuweisung einen *Ausdruck* darstellt, dass wir es also mit dem binären (zweistelligen) Operator „=“ zu tun haben, für den folgende 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.

Wie beim Inkrement- bzw. Dekrementoperators 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.

In folgendem 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, *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 erfolgt eine Abarbeitung von **rechts nach links** (vgl. Tabelle in Abschnitt 3.5.10), so dass die Anweisung

```
i = j = j * i;
```

folgendermaßen ausgeführt wird:

- Weil der Multiplikationsoperator eine höhere Priorität besitzt als der Zuweisungsoperator, wird zuerst der Ausdruck `j * i` ausgewertet, was zum Zwischenergebnis 8 (mit Datentyp **int**) führt.

- Nun wird 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.

Ausdrücke der Art

```
i = j = k;
```

stammen übrigens 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 Abschnitt 3.3.6 wissen, stellt ein Zuweisungsausdruck bereits eine vollständige **Anweisung** dar, sobald man ein Semikolon dahinter setzt. Dies gilt auch für die die Prä- und Postinkrementausdrücke (vgl. Abschnitt 3.5.1) sowie für Methodenaufrufe, jedoch *nicht* für die anderen Ausdrücke, die in Abschnitt 3.5 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 für Schreibfaule, die gelegentlich auch als **Aktualisierungsoperatoren** bezeichnet 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 3.5.1), akzeptiert der Compiler den äquivalenten Ausdruck mit Aktualisierungsoperator:

```
b1 += b2;
```

3.5.9 Konditionaloperator

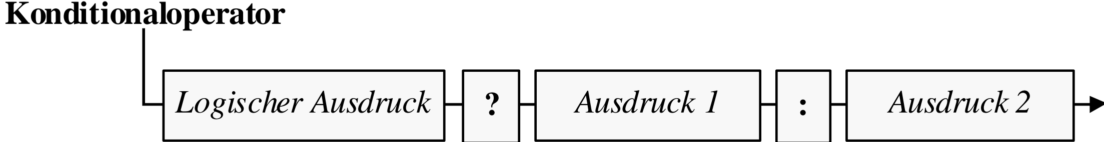
Der **Konditionaloperator** erlaubt eine sehr kompakte Schreibweise, wenn beim neuen Wert einer Zielvariablen bedingungsabhängig zwischen zwei Ausdrücken zu entscheiden ist, z.B.

$$i = \begin{cases} i + j & \text{falls } k > 0 \\ i - j & \text{sonst} \end{cases}$$

In C# ist für diese Zuweisung mit Fallunterscheidung nur eine einzige Zeile erforderlich:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>using System;
class Prog {
  static void Main() {
    int i = 2, j = 1, k = 7;
    i = (k>0) ? i+j : i-j;
    Console.WriteLine(i);
  }
}</pre></td><td>3</td></tr>
</table>

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*.

3.5.10 Auswertungsreihenfolge

Bisher haben wir Ausdrücke mit *mehreren* Operatoren und das damit verbundene Problem der Auswertungs*reihenfolge* nach Möglichkeit gemieden. Nun werden die Regeln vorgestellt, nach denen der C# - Compiler komplexe Ausdrücke mit mehreren Operatoren auswertet:

1) Klammern
Wenn aus den anschließend erläuterten Regeln (Priorität und Assoziativität) nicht die gewünschte Auswertungsfolge resultiert, greift man mit runden Klammern steuernd ein. Die Auswertung von (eventuell mehrstufig) eingeklammerten Teilausdrücken erfolgt von innen nach außen.

2) Priorität
Bei konkurrierenden Operatoren (auf derselben Klammerungsebene) entscheidet zunächst die Priorität (siehe Tabelle unten) darüber, in welcher Reihenfolge die Auswertung vorgenommen wird. Z.B. hält sich C# bei arithmetischen Ausdrücken an die mathematische Regel

Punktrechnung geht vor Strichrechnung

3) Assoziativität (Auswertungsrichtung)
Steht ein Argument zwischen zwei Operatoren mit gleicher Priorität, dann entscheidet die Assoziativität der Operatoren über die Reihenfolge der Auswertung (vgl. ECMA 2006, S. 151):

- Die meisten binären Operatoren sind links-assoziativ; sie werden also von links nach rechts ausgewertet. Z.B. wird

  ```
  x - y - z
  ```

  ausgewertet als

  ```
  (x - y) - z
  ```

- Die Zuweisungsoperatoren und der Konditionaloperator sind rechts-assoziativ; sie werden also von rechts nach links ausgewertet. Z.B. wird

```
a += b -= c = 3
```

ausgewertet als

```
a += (b -= (c = 3))
```

Es ist dafür gesorgt, dass Operatoren mit gleicher Priorität stets auch die gleiche Assoziativität besitzen, z.B. die im letzten Beispiel enthaltenen Operatoren +=, -= und =.

Für manche Operationen gilt das mathematische Assoziativitätsgesetz, so dass die Reihenfolge der Auswertung irrelevant ist, z.B.:

$(3 + 2) + 1 = 6 = 3 + (2 + 1)$

Anderen Operationen fehlt diese nette Eigenschaft, z.B.:

$(3 - 2) - 1 = 0 \neq 3 - (2 - 1) = 2$

In der folgenden Tabelle sind die bisher behandelten Operatoren in absteigender Priorität aufgelistet. Gruppen von Operatoren mit gleicher Priorität sind durch horizontale Linien voneinander abgegrenzt. 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**)
B Ausdruck mit beliebigem 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**
x++, *x*--	Postinkrement bzw. -dekrement	V_n
Methode(*Parameter*)	Methodenaufruf	
-*x*	Vorzeichenumkehr	*N*
!	Negation	*L*
++*x*, --*x*	Präinkrement bzw. -dekrement	V_n
(*Typ*)	Typumwandlung	*B*
*, /	Punktrechnung	*N, N*
%	Modulo (Divisionsrest)	*N, N*
+, -	Strichrechnung	*N, N*
+	Stringverkettung	*S, B* oder *B, S*
<<, >>	Links- bzw. Rechts-Shift	*I, I*

<table>
<tr><th>Operator</th><th>Bedeutung</th><th>Operanden</th></tr>
<tr><td>>, <, >=, <=</td><td>Vergleichsoperatoren</td><td>N, N</td></tr>
<tr><td>==, !=</td><td>Gleichheit, Ungleichheit</td><td>B, B</td></tr>
<tr><td>&
&</td><td>Bitweises UND
Logisches UND (mit unbedingter Auswertung)</td><td>I, I
L, L</td></tr>
<tr><td>^</td><td>Exklusives logisches ODER</td><td>L, L</td></tr>
<tr><td>|
|</td><td>Bitweises ODER
Logisches ODER (mit unbedingter Auswertung)</td><td>I, I
L, L</td></tr>
<tr><td>&&</td><td>Logisches UND (mit bedingter Auswertung)</td><td>L, L</td></tr>
<tr><td>||</td><td>Logisches ODER (mit bedingter Auswertung)</td><td>L, L</td></tr>
<tr><td>? :</td><td>Konditionaloperator</td><td>L, B, B</td></tr>
<tr><td>=
+=, -=, *=, /=, %=</td><td>Wertzuweisung
Wertzuweisung mit Aktualisierung</td><td>V, B
V_n, N</td></tr>
</table>

Im Anhang finden Sie eine erweiterte Version dieser Tabelle, die zusätzlich alle Operatoren enthält, die im weiteren Verlauf des Kurses noch behandelt werden.

3.5.11 Übungsaufgaben zu Abschnitt 3.5

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 Wahrheitsweite erhalten in folgendem Programm die booleschen Variablen `la1` bis `la3`?

```
using System;
class Prog {
  static void Main() {
    bool la1, la2, la3;
    int i = 3;
    char c = 'n';

    la1 = 3 > 2 && 2 == 2 ^ 1 == 1;
    Console.WriteLine(la1);

    la2 = ((2 > 3) && (2 == 2)) ^ (1 == 1);
    Console.WriteLine(la2);

    la3 = !(i > 0 || c == 'j');
    Console.WriteLine(la3);
  }
}
```

Tipp: Die Negation von zusammengesetzten Ausdrücken ist etwas unangenehm. Mit Hilfe der Regeln von **DeMorgan** 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 FCL-Dokumentation zur Klasse **Math** im Namensraum **System** eine passende Methode.
- Verwenden Sie zum Einlesen des Arguments eine Variante der in Abschnitt 3.4 beschriebenen Technik, wobei die **Convert**-Methode **ToInt32()** geeignet zu ersetzen ist.

5) Erstellen Sie ein Programm, das einen DM-Betrag entgegen nimmt und diesen in Euro konvertiert. In der Ausgabe sollen ganzzahlige, korrekt gerundete Werte für Euro und Cent erscheinen, z.B.:

Eingabe: `DM-Betrag: 321`
Ausgabe: `164 Euro und 12 Cent`

Umrechnungsfaktor: 1 Euro = 1,95583 DM

3.6 Über- und Unterlauf bei numerischen Variablen

Wie Sie inzwischen wissen, haben die numerischen Datentypen jeweils einen bestimmten Wertebereich (siehe Tabelle in Abschnitt 3.3.4). 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, so dass Wertebereichsprobleme unbedingt vermieden bzw. rechtzeitig diagnostiziert werden müssen.

3.6.1 Überlauf bei Ganzzahltypen

Ohne besondere Vorkehrungen stellt ein C# - Programm im Fall eines Ganzzahl-Überlaufs keinesfalls seine Tätigkeit ein (z.B. mit einem Ausnahmefehler), sondern arbeitet munter weiter. Dieses

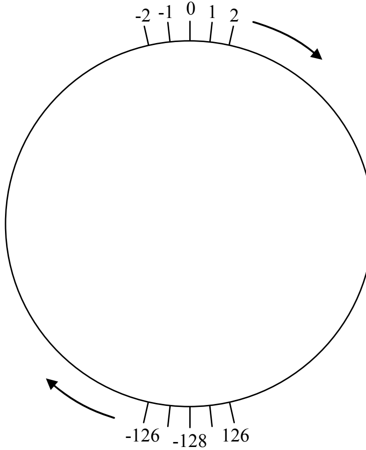
Verhalten ist beim Programmieren von Pseudozufallszahlgeneratoren willkommen, ansonsten aber eher bedenklich. Das folgende Programm

```
using System;
class Prog {
  static void Main() {
    int i = 2147483647, j = 5, k;
    k = i + j;    // Überlauf!
    Console.WriteLine(i + " + " + j + " = " + k);
  }
}
```

liefert ohne jede Warnung das fragwürdige Ergebnis:

```
2147483647 + 5 = -2147483644
```

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 3.5.7) ist also Vorsicht geboten, weil ansonsten das Kommando „Mr. Spock, please push the engine.“ [1] zum heftigen Rückwärtsschub führen könnte.

Oft kann ein Überlauf durch Wahl eines **geeigneten Datentyps** verhindert werden. Mit den Deklarationen

```
long i = 2147483647, j = 5, k;
```

erhält man das korrekte Ergebnis, weil neben `i`, `j` und `k` nun auch der Ausdruck `i+j` den Typ **long** hat:

```
2147483647 + 5 = 2147483652
```

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 = 2147483647, 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.

---

[1] Mr. Spock arbeitete jahrelang als erster Offizier auf dem Raumschiff Enterprise.

- Schließlich wird der **long**-Variablen k das falsche Ergebnis zugewiesen.

In C# steht im Unterschied zu vielen anderen Programmiersprachen 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 = 2147483647, j = 5, k;
    k = checked(i + j);
    Console.WriteLine(i + " + " + j + " = " + k);
  }
}
```

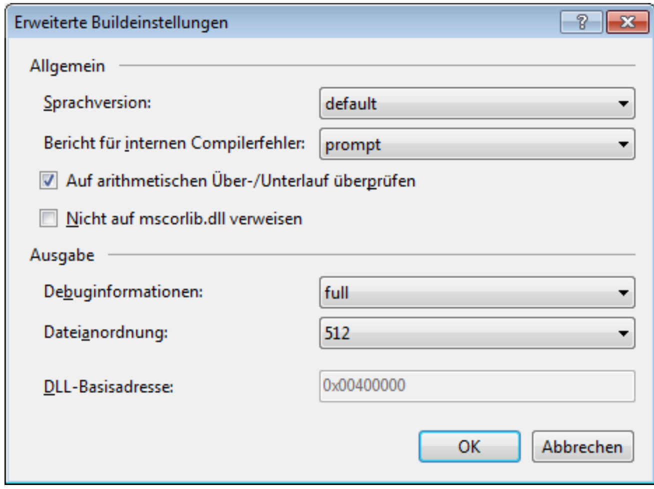
rechnet nach einem Überlauf nicht mit „Zufallszahlen" weiter, sondern bricht mit einem Ausnahmefehler ab:

```
U:\Eigene Dateien\C#\Kurs\Prog\bin\Debug>prog

Unbehandelte Ausnahme: System.OverflowException: Die arithmetische Operation hat
 einen Überlauf verursacht.
   bei Prog.Main() in U:\Eigene Dateien\C#\BspUeb\Elementare Sprachelemente\Prog
\Prog.cs:Zeile 5.
```

Im Abschnitt über Ausnahmebehandlung werden Sie lernen, solche Situationen programmintern zu beheben, so dass sie nicht mehr zum Abbruch des Programms führen.

An Stelle 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;
    . . .
  }
  ```
- **checked**-Compileroption
  Man kann die Überwachung für einen kompletten Compiler-Lauf einschalten.
  Beispiel:
  ```
  csc /checked Prog.cs
  ```
  Für ein Projekt der Visual C# 2010 Express Edition vereinbart man diese Compiler-Option folgendermaßen:
  - Menübefehl **Projekt > Eigenschaften**
  - Registerkarte **Erstellen**
  - Schalter **Erweitert**
  - Kontrollkästchen **Auf arithmetischen Über-/Unterlauf überprüfen**



Der Vollständigkeit halber soll an dieser Stelle noch der **unchecked**-Operator erwähnt werden, mit dem sich Kontrollfunktionen des Compilers abschalten lassen, was für sehr spezielle Zwecke (z.B. zum Programmieren von Pseudozufallszahlengeneratoren) vielleicht einmal sinnvoll sein kann. Auch ohne **checked**-Instruktion verhindert der Compiler z.B., dass einer Ganzzahlvariablen per Literal ein übergroßer Wert zugewiesen wird. Unsere Entwicklungsumgebung verrät, wie man diese Überwachung abschalten kann:

```
int i = (int)2147483652;
```

struct System.Int32
Stellt eine 32-Bit-Ganzzahl mit Vorzeichen dar.

Fehler:
Der Konstantenwert "2147483652" kann nicht in "int" konvertiert werden. (Verwenden Sie zum Überschreiben die unchecked-Syntax.)

Welches Ergebnis bei diesem Programm resultiert, wissen Sie schon:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`    int i = unchecked((int) 2147483652);`
`    Console.WriteLine(i);`
`  }`
`}`
	`-2147483644`

3.6.2 Unendliche und undefinierte Werte bei den Typen float und double

Auch bei den binären Gleitkommatypen **float** und **double** kann ein Überlauf auftreten, obwohl der unterstützte Wertebereich hier weit größer ist. 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 = Double.MaxValue * 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));
    Console.ReadLine();
  }
}
```

liefert die Ausgabe:

```
Double.MaxValue =        1,79769313486232E+308
Double.MaxValue * 10 =   +unendlich
Unendl. + 10 =           +unendlich
Unendl. * -13 =          -unendlich
13.0/0.0 =               +unendlich
```

Mit Hilfe der Unendlich-Werte „gelingt“ offenbar sogar die Division durch Null.

Bei diesen „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:

```
using System;
class Prog {
  static void Main() {
    double bigd;
    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 die Ausgabe:

```
Unendlich - Unendlich = n. def.
Unendlich / Unendlich = n. def.
Unendlich * 0.0 =       n. def.
0.0 / 0.0 =             n. def.
```

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.

Über die statischen **Double**-Methoden

- **public bool IsPositiveInfinity(double *d*)**
- **public bool IsNegativeInfinity(double *d*)**
- **public bool IsNaN(double *d*)**

mit einem Parameter vom Typ **double** und einer Rückgabe vom Typ **bool** lässt sich für eine **double**-Variable prüfen, ob sie einen unendlichen oder undefinierten Wert besitzt, 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**).

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 Unterschied zur Beschreibung von C#-Syntaxregeln) erstmals benutzt, die auch in der FCL-Dokumentation in ähnlicher Form Verwendung findet, z.B.:

```
C#
public static bool IsPositiveInfinity(
    double d
)
```

Dabei wird die Benutzung einer vorhandenen Methode erläutert durch Angabe von:

- Modifikatoren (z.B. für den Zugriffsschutz)
- Rückgabetyp
- Methodenname
- Parameterliste (mit Angabe der Parametertypen)

Analog werden wir auch Eigenschaften beschreiben.

Für besonders neugierige Leser sollen abschließend noch die **float**-Darstellungen der speziellen Gleitkommawerte angegeben werden (vgl. Abschnitt 3.3.5.1):

Wert	**float-Darstellung**		
	Vorz.	**Exponent**	**Mantisse**
+unendlich	0	11111111	00000000000000000000000
-unendlich	1	11111111	00000000000000000000000
NaN	0	11111111	10000000000000000000000

3.6.3 Überlauf beim Typ decimal

Beim Typ **decimal** wird im Fall eines Überlaufs *nicht* mit dem speziellen Wert Unendlich weiter gearbeitet, denn ...

- der Sprung vom maximalen Wert (ca. $7{,}9 \cdot 10^{28}$) ins Unendliche wäre viel zu groß (zum Vergleich das ungefähre Maximum bei **float** bzw. **double**: $3.4 \cdot 10^{38}$ bzw. $1{,}8 \cdot 10^{308}$).
- im Unterschied zu den binären Gleitkommatypen sind bei **decimal** spezielle Werte wie ±Unendlich nicht darstellbar.

Stattdessen wird ein Ausnahmefehler gemeldet, der unbehandelt zur Beendigung des Programms führt. Im Unterschied zu den Ganzzahltypen (vgl. Abschnitt 3.6.1) muss die Überlaufdiagnose *nicht* per **checked**-Operator, -Anweisung oder -Compileroption angeordnet werden. Das Beispielprogramm

```
using System;
class Prog {
  static void Main() {
    decimal d = Decimal.MaxValue;
    d = d * 10;
    Console.WriteLine(d);
  }
}
```

wird mit folgender Meldung abgebrochen:

```
U:\Eigene Dateien\C#\Kurs\Prog\bin\Debug>prog

Unbehandelte Ausnahme: System.OverflowException: Der Wert für eine Decimal war z
u groß oder zu klein.
   bei System.Decimal.FCallMultiply(Decimal& d1, Decimal& d2)
   bei System.Decimal.op_Multiply(Decimal d1, Decimal d2)
   bei Prog.Main() in U:\Eigene Dateien\C#\BspUeb\Elementare Sprachelemente\Prog
\Prog.cs:Zeile 5.
```

3.6.4 Unterlauf bei den Gleitkommatypen

Bei den Gleitkommatypen **float**, **double** und **decimal** ist auch ein **Unterlauf** möglich, wobei eine Zahl mit sehr kleinem 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

`using System;`
`class Prog {`
`  static void Main() {`
`    double smalld = Double.Epsilon;`
`    Console.WriteLine(smalld);`
`    smalld /= 10.0;`
`    Console.WriteLine(smalld);`
`  }`
`}`
	`4,94065645841247E-324`
	`0`

Das öffentliche Feld **Double.Epsilon** enthält den kleinsten Betrag, der in einer **double**-Variablen gespeichert werden kann (vgl. Abschnitt 3.3.5.1 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 Pogramm kommt bei der Rechnung

$$10^{-323} * 10^{-308} * 10^{16}$$

dem korrekten Ergebnis 10 recht nahe. Wird aber der Gesamtfaktor Eins unglücklich in den Rechenweg eingebaut, führt ein irreversibler Unterlauf zum falschen Ergebnis Null:

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`

3.7 Anweisungen (zur Ablaufsteuerung)

Wir haben uns in Abschnitt 3 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) kennen lernen.

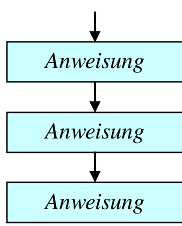
3.7.1 Überblick

Ausführbare Programmteile, die in C# nach unserem bisherigen Kenntnistand als Methoden oder Eigenschaften von Klassen zu realisieren sind, bestehen aus Anweisungen (engl. *statements*).

Am Ende von Abschnitt 3.7 werden Sie die folgenden Sorten von Anweisungen kennen:

- **Variablendeklarationsanweisung**
  Die Variablendeklarationsanweisung wurde schon in Abschnitt 3.3.6 eingeführt.
  Beispiel: `int i = 1, k;`
- **Ausdrucksanweisungen**
  Folgende Ausdrücke werden zu Anweisungen, sobald man ein Semikolon dahinter setzt:
    - **Wertzuweisung** (vgl. Abschnitte 3.3.6 und 3.5.8)
      Beispiel: `k = i + j;`
    - **Prä- bzw. Postinkrement- oder -dekrementoperation**
      Beispiel: `i++;`
      Hier ist nur der „Nebeneffekt“ des Ausdrucks `i++` von Bedeutung. Sein Wert bleibt ungenutzt.
    - **Methodenaufruf**
      Beispiel: `Console.WriteLine(cond);`
      Besitzt die aufgerufene Methode einen Rückgabewert (siehe unten), 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 die Syntax eine Anweisung verlangt, 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 in Abschnitt 3.3.7 im Zusammenhang mit dem Deklarationsbereich 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 ja sehr unpraktisch, dieselbe Bedingung für jede betroffene Anweisung wiederholen zu müssen.
- **Anweisungen zur Ablaufsteuerung**
  Die Methoden der bisherigen Beispielprogramme in Abschnitt 3 bestanden meist aus einer *Sequenz* von Anweisungen, die bei jedem Programmlauf komplett und linear durchlaufen wurde:



Oft möchte man jedoch z.B.

- die Ausführung einer Anweisung (eines Anweisungsblocks) von einer *Bedingung* abhängig machen
- oder eine Anweisung (einen Anweisungsblock) *wiederholt* ausführen lassen.

Für solche Zwecke stellt C# etliche Anweisungen zur Ablaufsteuerung zur Verfügung, 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.

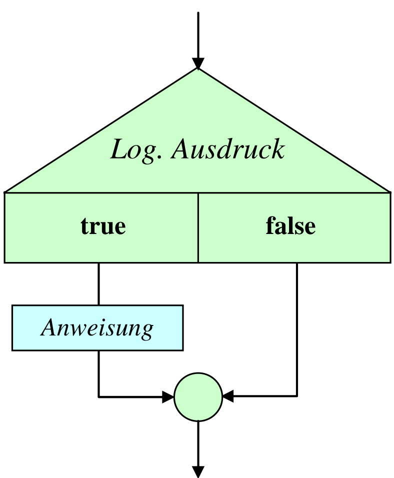
Anweisungen werden durch ein **Semikolon** abgeschlossen, sofern sie nicht mit einer schließenden Blockklammer enden.

3.7.2 Bedingte Anweisung und Verzweigung

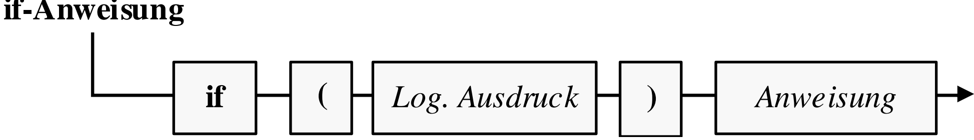
Oft ist es erforderlich, dass eine Anweisung nur unter einer bestimmten Bedingung ausgeführt wird. Etwas allgemeiner formuliert geht es darum, dass viele Algorithmen *Fallunterscheidungen* benötigen, also an bestimmten Stellen in Abhängigkeit vom Wert eines steuernden Ausdrucks in unterschiedliche Pfade verzweigen müssen.

3.7.2.1 if-Anweisung

Nach dem folgenden **Programmablaufplan** (**PAP**) bzw. **Flussdiagramm** soll eine (Block-)Anweisung nur dann ausgeführt werden, wenn ein logischer Ausdruck den Wert **true** besitzt:



Während der hier erstmals als Darstellungstechnik verwendete Programmablaufplan den Zweck (die Semantik) eines Sprachbestandteils erläutert, beschreibt das bereits vertraute Syntaxdiagramm recht präzise, wie zulässige Exemplare des Sprachbestandteils zu bilden sind. Das folgende Syntaxdiagramm beschreibt die zur Realisation einer bedingten Ausführung geeignete **if**-Anweisung:



Um genau zu sein, muss zu diesem Syntaxdiagramm noch angemerkt werden, dass als bedingt auszuführende Anweisung keine Variablendeklaration erlaubt ist. Es ist übrigens nicht vergessen worden, ein Semikolon ans Ende des **if**-Syntaxdiagramms zu setzen. Dort wird eine Anweisung verlangt, wobei konkrete Beispiele oft mit einem Semikolon enden.

Im folgenden Beispiel wird eine Meldung ausgegeben, wenn die Variable `anz` einen Wert kleiner oder gleich Null besitzt:

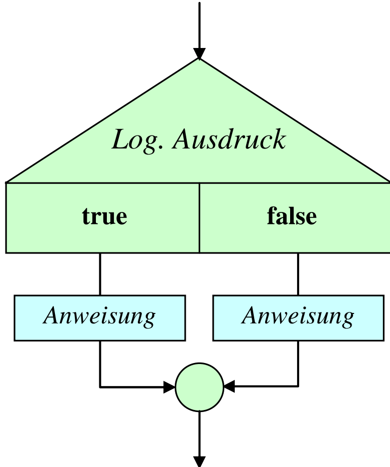
```
if (anz <= 0)
  Console.WriteLine("Die Anzahl muss > 0 sein!");
```

Der Zeilenumbruch zwischen dem logischen Ausdruck und der (Unter-)Anweisung dient nur der Übersichtlichkeit und ist für den Compiler irrelevant.

Selbstverständlich kommt als Anweisung auch ein *Block* in Frage.

3.7.2.2 if-else - Anweisung

Soll auch etwas passieren, wenn der steuernde logische Ausdruck den Wert **false** besitzt,



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 sehr übersichtlich ist, bietet das Syntaxdiagramm den Vorteil, bei komplizierter, variantenreicher Syntax alle zulässigen Formulierungen kompakt und präzise als Pfade durch das Diagramm zu beschreiben.

Wie schon bei der einfachen **if**-Anweisung gilt auch bei der **if-else**-Anweisung, dass Variablendeklarationen nicht als eingebettete Anweisungen erlaubt sind.

Im folgenden **if-else** - Beispiel wird der natürliche Logarithmus zu einer Zahl geliefert, falls diese positiv ist. Anderenfalls erscheint eine Fehlermeldung mit Alarmton (Escape-Sequenz **\a**). Das Argument wird vom Benutzer über eine **ToDouble()-ReadLine()** - Konstruktion erfragt (vgl. Abschnitt 3.4.1).

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

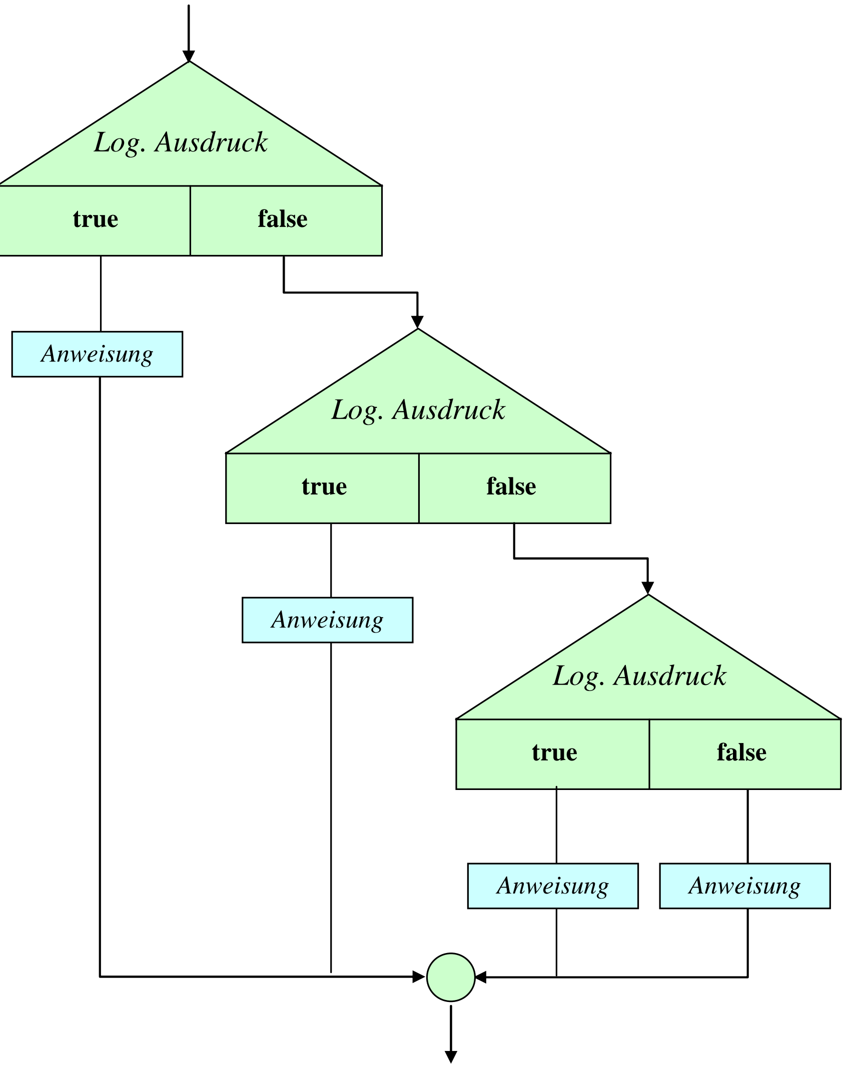
```
using System;
class Prog {
  static void Main() {
    Console.Write("Argument: ");
    double arg = Convert.ToDouble(Console.ReadLine());
    if (arg > 0)
      Console.WriteLine("ln(" + arg + ") = "
                        + Math.Log(arg));
    else
      Console.WriteLine("\aArgument <= 0!");
  }
}
```

</td><td>

```
Argument: 2
ln(2) = 0,693147180559945
```

</td></tr>
</table>

Eine bedingt auszuführende Anweisung darf durchaus wiederum vom **if**– bzw. **if-else** – Typ sein, so dass sich mehrere, *hierarchisch geschachtelte* Fälle unterscheiden lassen. Den folgenden Programmablauf mit „sukzessiver Restaufspaltung“



realisiert z.B. eine **if-else** – Konstruktion nach diesem Muster:

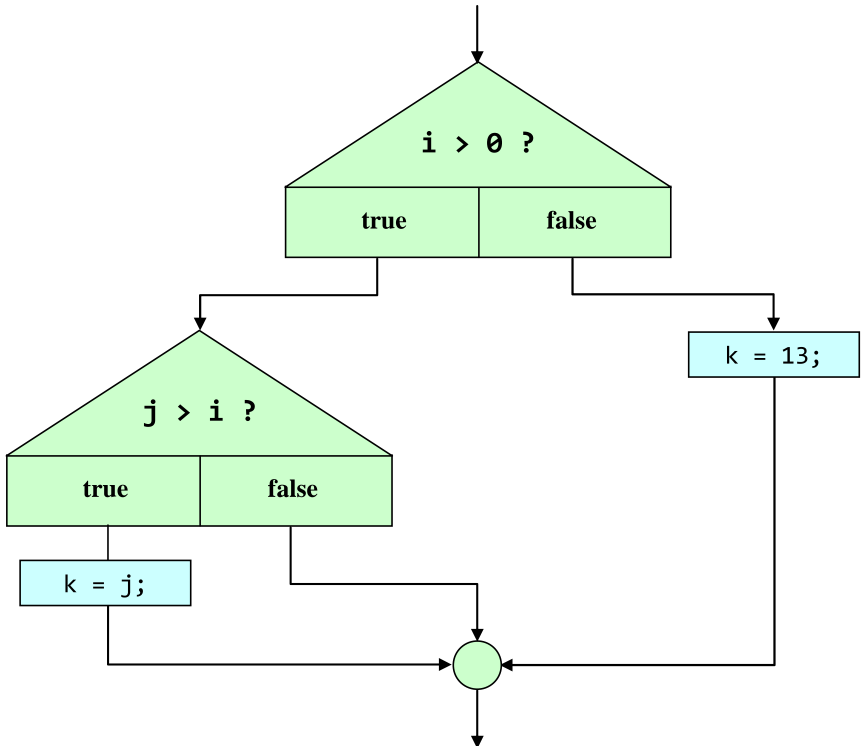
if (*Logischer Ausdruck 1*)
  Anweisung 1
else if (*Logischer Ausdruck 2*)
  Anweisung 2
  . . .
  . . .
else if (*Logischer Ausdruck k*)
  Anweisung k
else
  Default-Anweisung

Wenn alle logischen Ausdrücke den Wert **false** annehmen, wird die **else**-Klausel zur letzten **if**-Anweisung ausgeführt. Die Bezeichnung *Default-Anweisung* in der Syntaxdarstellung erfolgte im Hinblick auf die in Abschnitt 3.7.2.3 vorzustellende **switch**-Anweisung, die bei einer Mehrfallunterscheidung gegenüber einer verschachtelten **if-else** – Konstruktion zu bevorzugen ist, wenn die Fallzuordnung über die verschiedenen Werte *eines* Ausdrucks (z.B. vom Typ **int**) erfolgen kann.

Beim Schachteln von bedingten Anweisungen kann es zum so genannten **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;
```

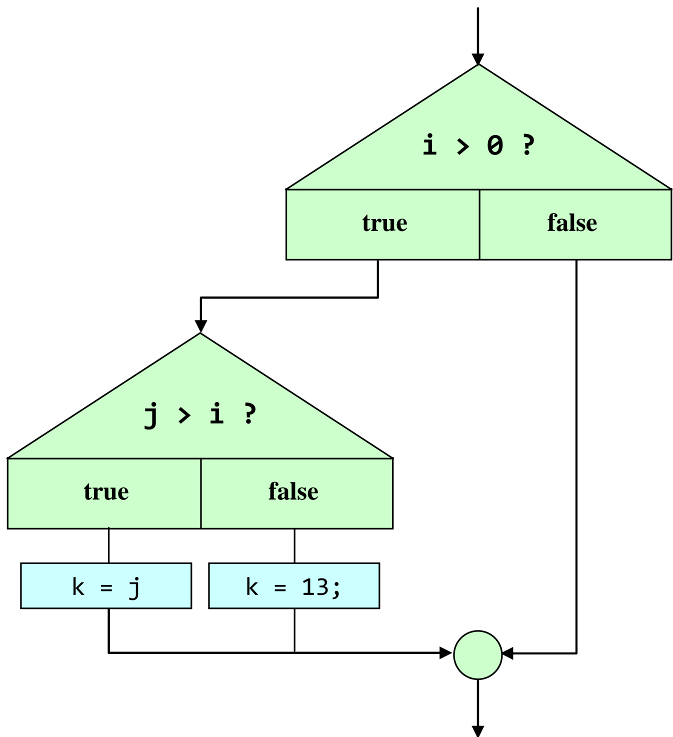
lassen die Einrücktiefen vermuten, dass der Programmierer die **else**-Klausel auf die *erste* **if**-Anweisung bezogen zu haben glaubt:



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 be-

[1] Deutsche Übersetzung von *dangling*: *baumelnd*.

zieht er die **else**-Klausel also auf die *zweite* **if**-Anweisung, so dass de facto folgender Programmablauf resultiert:



Mit Hilfe von Blockklammern kann die gewünschte Zuordnung erzwungen werden:

```
if (i > 0)
   {if (j > i)
      k = j;}
else
   k = 13;
```

Alternativ kann man auch 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;
```

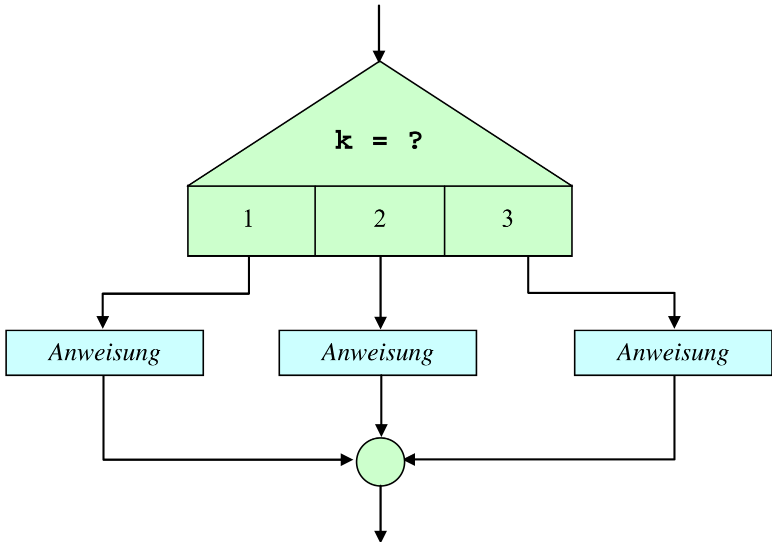
Gelegentlich kommt als Alternative zu einer simplen **if-else**-Anweisung, die zur Berechnung eines Wertes bedingungsabhängig zwei unterschiedliche Ausdrücke benutzt, der Konditionaloperator (vgl. Abschnitt 3.5.9) in Frage, z.B.:

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;`

3.7.2.3 *switch-Anweisung*

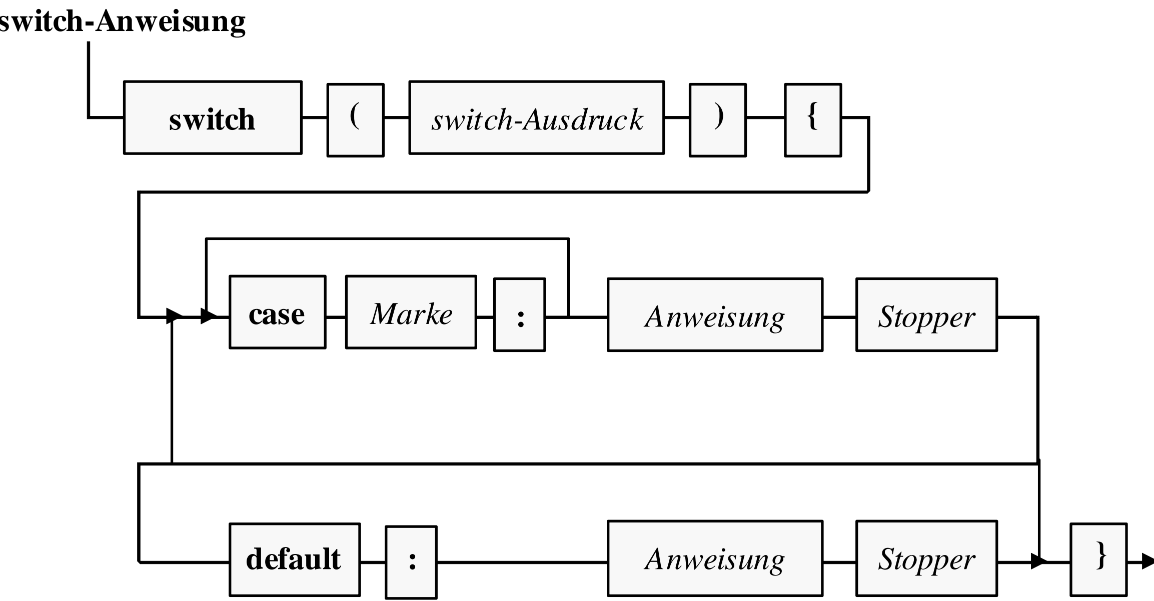
Wenn eine Fallunterscheidung mit mehr als zwei Alternativen in Abhängigkeit vom Wert *eines* Ausdrucks vorgenommen werden soll,



dann ist eine **switch**-Anweisung weitaus handlicher als eine verschachtelte **if-else** - Konstruktion. In Bezug auf den Datentyp des steuernden Ausdrucks ist C# sehr flexibel und erlaubt:

- numerische Datentypen
  Dazu gehört auch der *integrale* Datentyp **char** (vgl. Abschnitt 3.3.4).
- Aufzählungstypen (siehe unten)
- Zeichenfolgen (Datentyp **string**)

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.



Weil später noch ein praxisnahes (und damit auch etwas kompliziertes) Beispiel folgt, ist hier ein ebenso einfaches wie sinnfreies Exemplar zur Erläuterung der Syntax angemessen:

Quellcode	Ausgabe

<pre>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 Durchfall per goto)");
    goto case 3;
   case 3:
   case 4:
    Console.WriteLine("Fälle 3, 4 (mit break-Stopper)");
    break;
   default: Console.WriteLine("Restkategorie");
    break;
   }
 }
}</pre>
	<pre>Fall 2 (mit Durchfall per goto)
	Fälle 3, 4 (mit break-Stopper)</pre>

Als **case**-Marken sind *konstante* Ausdrücke erlaubt, 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 einer **case**-Marke überein, dann wird die zugehörige Anweisung ausgeführt, ansonsten (falls vorhanden) die **default**-Anweisung.

Soll für mehrere Werte des **switch**-Ausdrucks dieselbe Anweisung ausgeführt werden, setzt man die zugehörigen **case**-Marken hintereinander und lässt die Anweisung auf die letzte Marke folgen. Leider gibt es keine Möglichkeit, eine *Serie* von Fällen durch Angabe der Randwerte (z.B. von *a* bis *z*) festzulegen.

Jeder Fall *muss* mit einem **Stopper** abgeschlossen werden, sogar der Spezialfall **default**:

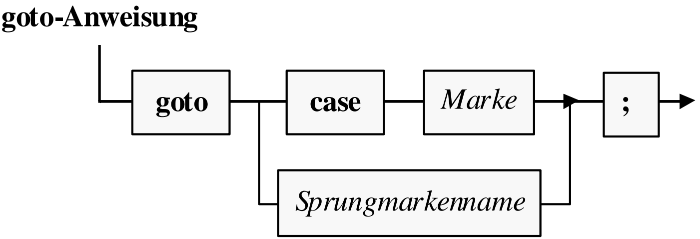
```
       default: Console.WriteLine("Restkategorie"); //break;
   }
```

Das Steuerelement kann nicht von einer case-Bezeichnung ("default:") zur nächsten fortfahren.

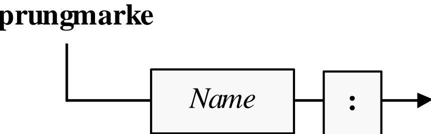
Der in anderen Programmiersprachen (z.B. C, C++, Java) mögliche „Durchfall“ zu den Anweisungen „tieferer“ Fälle ist verboten, womit einige elegante Formulierungen und viele Programmierfehler vermieden werden.

Meist stoppt man mit der **break**-Anweisung, wobei die Methode hinter der **switch**-Anweisung fortgesetzt wird. Mögliche Alternativen zur **break**-Anweisung sind:

- **return**-Anweisung
  Die Methode wird verlassen.
- **throw**-Anweisung
  Es wird eine Ausnahme ausgelöst (siehe unten).
- **goto**-Anweisung
  Per **goto**-Anweisung



kann eine **case**-Marke innerhalb der **switch**-Anweisung oder eine Sprungmarke



an anderer Stelle innerhalb der Methode angesteuert werden. Das gute (böse) alte **goto**, als Inbegriff rückständigen Programmierens 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.

Im folgenden Beispielprogramm wird die Persönlichkeit des Benutzers mit Hilfe seiner Farb- und Zahlpräferenzen 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;
        }
        char farbe = Char.ToLower(args[0][0]);
        int zahl = Convert.ToInt32(args[1]);
        switch (farbe) {
            case 'r': case 'R':
                Console.WriteLine("Sie sind ein emotionaler Typ.");
                break;
            case 'g': case 'G':
            case 'b': case 'B': {
                    Console.WriteLine("Sie scheinen ein sachlicher Typ zu sein");
                    if (zahl % 2 == 0)
                        Console.WriteLine("Sie haben einen geradlinigen Charakter.");
                    else
                        Console.WriteLine("Sie machen wohl gerne krumme Touren.");
                }
                break;
            case 's': case 'S':
                Console.WriteLine("Nehmen Sie nicht Alles so tragisch.");
                break;
            default:
                Console.WriteLine("Offenbar mangelt es Ihnen an Disziplin.");
                break;
        }
        Console.ReadLine();
    }
}
```

Das Programm `PerST` demonstriert nicht nur die **switch**-Anweisung, sondern auch die Verwendung von **Befehlszeilenargumenten**. Benutzer des Programms sollen beim Start ihre bevorzugte Farbe aus einer Palette mit den drei Grundfarben und Schwarz sowie ihre Lieblingszahl angeben, wobei die Farbe folgendermaßen durch einen Buchstaben zu kodieren ist:

r für Rot
g für Grün
b für Blau
s für Schwarz

Wer z.B. die Farbe Blau und die Zahl 17 bevorzugt, sollte das Programm also (bis auf die beliebige Groß-/Kleinschreibung beim Programmnamen) folgendermaßen starten:

```
PerST b 17
```

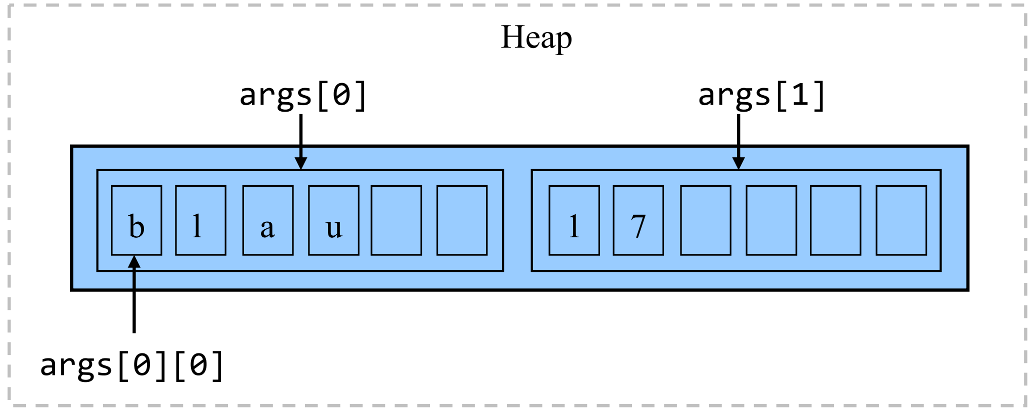
Im Quellcode wird jeweils nur *eine* Anweisung benötigt, um die (durch Leerzeichen getrennten) Befehlszeilenargumente auszuwerten und das Ergebnis in eine **char**- bzw. **int**-Variable zu befördern. Solche Anweisungen werden Sie mit Leichtigkeit selbst formulieren, sobald Methoden-Parameter sowie Arrays 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. Dies sind Objekte der Klasse **String**, die jeweils eine Zeichenkette repräsentieren. Literale mit diesem Datentyp sind uns schon öfter begegnet (z.B. `"Hallo"`).
- In der Parameterliste einer Methode kann die gewünschte Arbeitsweise näher spezifiziert werden.
- Besitzt die **Main**()-Methode einer Startklasse einen (ersten und einzigen) Parameter vom Datentyp **String[]** (Array mit **String**-Elementen), dann übergibt das .NET – Laufzeitsystem dieser Methode als **String**-Objekte 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 jedoch frei wählbar (im Beispiel: `args`). In der Methode **Main()** kann man auf den Array `args` lesend genauso zugreifen wie auf eine lokale Variable vom selben Typ.
- Das erste Befehlszeilenargument landet im ersten Element des Zeichenketten-Arrays `args` und wird mit `args[0]` angesprochen, weil Array-Elemente mit Null beginnend nummeriert sind. Das Array-Element `args[0]` kann mehrere Zeichen enthalten (z.B. `blau`), von denen im Beispiel aber nur das erste interessiert, welches mit `args[0][0]` (Zeichen Nummer Null vom Array-Element Nummer Null) angesprochen.
- Damit unser Programm auch groß geschriebene Farbnamen akzeptiert (z.B. `Blau`), wenden wir die statische Methode **ToLower**() der Klasse **Char** auf den extrahierten Anfangsbuchstaben des ersten Befehlszeilenarguments an, bevor er der **char**-Variablen `farbe` zugewiesen wird:

  ```
  char farbe = Char.ToLower(args[0][0]);
  ```

- Das zweite Element des Zeichenketten-Arrays `args` (mit der Nummer Eins) enthält das zweite Befehlszeilenargument. Zumindest bei kooperativen Benutzern des Beispielprogramms kann man aus dieser Zeichenfolge mit der Klassenmethode **ToInt32**() der Klasse **Convert** eine Zahl vom Datentyp **int** gewinnen und anschließend der Variablen `zahl` zuweisen.

Man kann sich den **String**-Array `args` ungefähr so vorstellen:



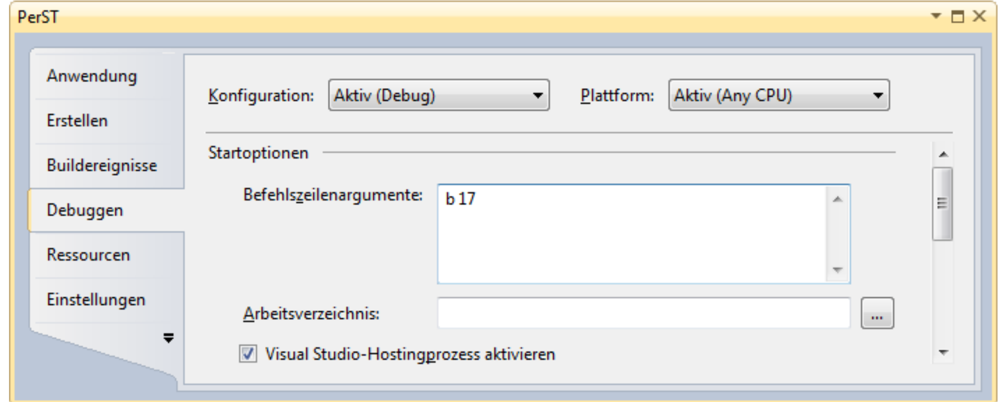
Ansonsten ist im Beispielprogramm noch die **return**-Anweisung von Interesse, welche die **Main**()-Methode und damit das Programm in Abhängigkeit von einer Bedingung sofort beendet:

```
return;
```

Sie ist uns oben schon als Stopper in der **switch**-Anweisung begegnet und wird im Zusammenhang mit der ausführlichen Behandlung der Methoden noch näher erläutert.

Für den Programmstart in der Visual C# 2010 - Umgebung kann man die Befehlszeilenargumente folgendermaßen vereinbaren:

- Menübefehl **Projekt > Eigenschaften**
- Registerkarte **Debuggen**
- **Befehlszeilenargumente** eintragen, z.B.:

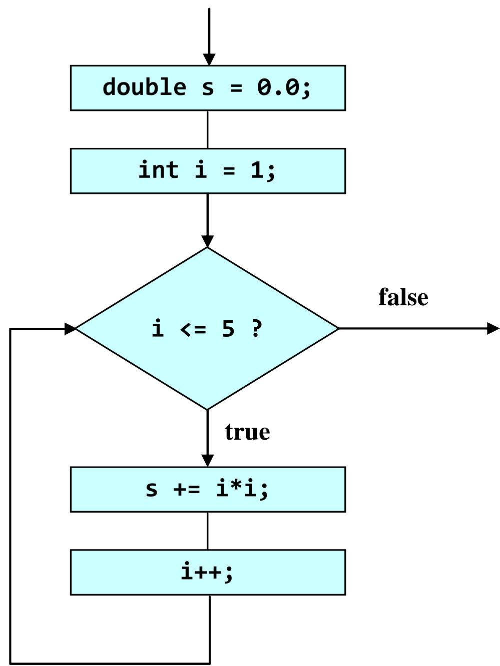


3.7.3 Wiederholungsanweisungen

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 Eins bis Fünf berechnet:[1]

[1] Das Verzweigungssymbol sieht aus darstellungstechnischen Gründen etwas anders aus als in Abschnitt 3.7.2, was aber keine Verwirrung stiften sollte.



C# bietet verschiedene Wiederholungsanweisungen, die sich bei der Ablaufsteuerung unterscheiden und gleich im Detail vorgestellt 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, bis die Zählvariable einen festgelegten Grenzwert erreicht hat (siehe obige Abbildung).
- **Iterieren über die Elemente einer Kollektion** (**foreach**)
  Mit der von Visual Basic übernommenen **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 wie *vor* dem 4. Schleifendurchgang, geht es bei der Entscheidung zwischen Kopf- und Fußsteuerung lediglich darum, ob auf jeden Fall ein *erster* Schleifendurchgang stattfinden soll oder nicht.

Die gesamte Konstruktion aus Schleifensteuerung und (Verbund-)anweisung stellt in syntaktischer Hinsicht *eine* zusammengesetzte Anweisung dar.

3.7.3.1 Zählergesteuerte Schleife (for)

Die Anweisung einer **for**-Schleife wird ausgeführt, solange eine Bedingung erfüllt ist, die normalerweise auf eine ganzzahlige Indexvariable Bezug nimmt. Anschließend wird die Methode hinter der **for**-Schleife fortgesetzt.

Auf das Schlüsselwort **for** folgt die von runden Klammern umgebene Schleifensteuerung, wo die Vorbereitung der Indexvariablen (nötigenfalls samt Deklaration), die Fortsetzungsbedingung und die Aktualisierungsvorschrift untergebracht werden können. Am Ende steht die wiederholt auszuführende (Block-)Anweisung:

for (*Vorbereitung*; *Bedingung*; *Aktualisierung*)   *Anweisung*

Zu den drei Bestandteilen der Schleifensteuerung sind einige Erläuterungen erforderlich, wobei anschließend etliche weniger typische bzw. sinnvolle Möglichkeiten weggelassen werden:

- ***Vorbereitung***
  In der Regel wird man sich auf *eine* Indexvariable beschränken und dabei einen Ganzzahl-Typ wählen. Somit kommen im Vorbereitungsteil der **for**-Schleifensteuerung in Frage:
  - eine Wertzuweisung, z.B.:
    `i = 1`
  - eine Variablendeklaration mit Initialisierung, z.B.
    `int i = 1`

  Im folgenden Programm findet sich die zweite Variante:

Quellcode	Ausgabe

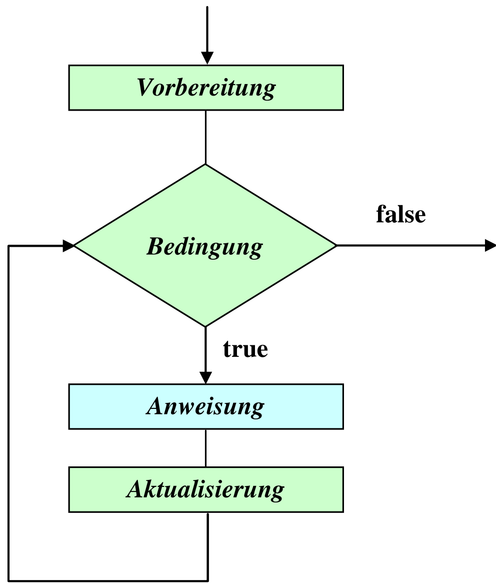
<pre>using System;
class Prog {
  static void Main() {
    double s = 0.0;
    for (int i = 1; i <= 5; i++)
      s += i*i;
    Console.WriteLine("Quadratsumme = " + s);
    }
}</pre>
	<pre>Quadratsumme = 55</pre>

  Der Vorbereitungsteil wird *vor dem ersten Durchlauf* ausgeführt. Eine hier deklarierte Variable ist *lokal* bzgl. der **for**-Schleife, steht also nur in deren Anweisung(sblock) zur Verfügung.

- ***Bedingung***
  Üblicherweise wird eine Ober- oder Untergrenze für die Indexvariable gesetzt, doch erlaubt C# beliebige logische Ausdrücke. Die Bedingung wird *vor jedem Schleifendurchgang* geprüft. Resultiert der Wert **true**, so findet eine weitere Wiederholung des Anweisungsteils statt, anderenfalls wird die **for**-Schleife verlassen. Folglich kann es auch passieren, dass überhaupt kein Durchlauf zustande kommt.

- ***Aktualisierung***
  Am Ende jedes Schleifendurchgangs (nach Ausführung der Anweisung) wird der Aktualisierungsteil ausgeführt. Hier wird meist die Indexvariable in- oder dekrementiert.

Im folgenden Flussdiagramm sind die semantischen Regeln zur **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 umsetzt, gehören **for**-Schleifenköpfe ohne Vorbereitung oder ohne Aktualisierung, wobei die trennenden Strichpunkte 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.

3.7.3.2 Iterieren über die Elemente einer Kollektion (foreach)

Obwohl wir uns bisher nur anhand von Beispielen mit *Kollektionen* wie Arrays oder Zeichenfolgen beschäftigt haben, kann die einfach aufgebaute **foreach**-Schleife doch hier im Kontext mit den übrigen Schleifen behandelt werden. Hinsichtlich der Steuerungslogik handelt es sich um einen einfachen Spezialfall der **for**-Schleife:

- Es ist eine Kollektion mit einer festen Anzahl von gleichartigen 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*

Das Beispielprogramm `PerST` in Abschnitt 3.7.2.3 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.

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 Zeichenfolgen).

Bei der **foreach**-Schleife wird offenbar das Initialisieren und Inkrementieren der Iterationsvariablen auf nahe liegende Weise automatisiert. Man darf nur *lesend* auf die Iterationsvariable zugreifen, so dass z.B. die folgende Konstruktion verboten ist:

```
foreach (char c in s)
    c = '2';
```

3.7.3.3 *Bedingungsabhängige Schleifen*

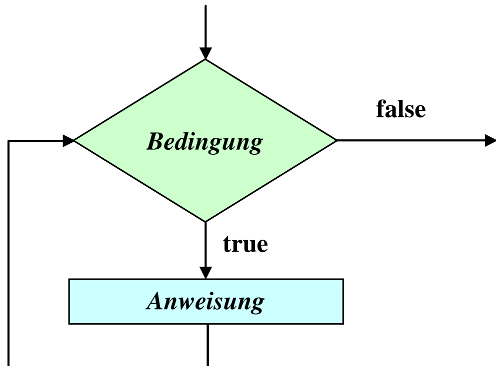
Wie die Erläuterungen zur **for**-Schleife gezeigt haben, ist die Überschrift dieses Abschnitts nicht sehr trennscharf, weil bei der **for**-Schleife ebenfalls eine beliebige Terminierungsbedingung angegeben werden darf. In vielen Fällen ist es eine Frage des persönlichen Geschmacks, welche Wiederholungsanweisung man zur Lösung eines konkreten Iterationsproblems benutzt. Unter der aktuellen Abschnittsüberschrift diskutiert man traditionsgemäß die **while**- und die **do**-Schleife.

3.7.3.3.1 while-Schleife

Die **while**-Anweisung kann als vereinfachte **for**-Anweisung beschreiben kann: Wer im Kopf einer **for**-Schleife auf Vorbereitung und Aktualisierung verzichten möchte, ersetzt besser das Schlüsselwort **for** durch **while** und erhält dann folgende Syntax:

while (*Bedingung*)   *Anweisung*

Wie bei der **for**-Anweisung wird die Bedingung *vor Beginn* eines Schleifendurchgangs geprüft. Resultiert der Wert **true**, so wird die Anweisung ausgeführt, anderenfalls wird die **while**-Schleife verlassen, eventuell noch vor dem ersten Durchgang:



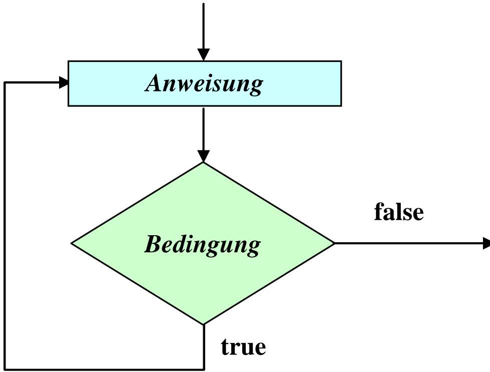
Im obigen Beispielprogramm zur Quadratsummenberechnung (vgl. Abschnitt 3.7.3.1) kann man die **for**-Schleife leicht durch eine **while**-Schleife ersetzen:

Quellcode	Ausgabe

<pre>using System;
class Prog {
  static void Main() {
    double s = 0.0;
    int i = 1;
    while (i <= 5) {
      s += i * i;
      i++;
    }
    Console.WriteLine("Quadratsumme = " + s);
  }
}</pre>
	Quadratsumme = 55

3.7.3.3.2 do-Schleife

Bei der **do**-Schleife wird die Fortsetzungsbedingung *am Ende* der Schleifendurchläufe geprüft, so dass wenigstens *ein* Durchlauf stattfindet:



Das Schlüsselwort **while** tritt auch in der Syntax zur **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. In folgendem Programm wird wie gewohnt 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 Null) per Indexzugriff (mit Hilfe der eckigen Klammern) angesprochen werden:

Quellcode	Ein-/Ausgabe

<pre>using System;
class Prog {
  static void Main() {
    char antwort;
    do {
      Console.Write("Einverstanden? (j/n)? ");
      antwort = Console.ReadLine()[0];
    } while (antwort != 'j' && antwort != 'n');
  }
}</pre>
	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).

3.7.3.4 Endlosschleifen

Bei einer **for**-, **while**- oder **do**-Schleife kann es in Abhängigkeit von der Fortsetzungsbedingung passieren, dass der Anweisungsteil unendlich oft ausgeführt wird. Endlosschleifen sind als gravierende Programmierfehler unbedingt zu vermeiden. 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**.

In folgendem Beispiel resultiert eine Endlosschleife aus einer unvorsichtigen Identitätsprüfung bei **double**-Werten (vgl. Abschnitt 3.5.3):

```
using System;
class Prog {
  static void Main() {
    int i = 0;
    double d = 1.0;
    // besser: while (Math.Abs(d - 0.0) > 1.0e-14) {
    while (d != 0.0) {
      i++;
      d = d - 0.1;
      Console.WriteLine("i = {0}, d = {1}", i, d);
    }
    Console.WriteLine("Fertig!");
  }
}
```

3.7.3.5 Schleifen(durchgänge) vorzeitig beenden

Mit der **break-**Anweisung, die uns schon als Bestandteil der **switch**-Anweisung begegnet ist, kann eine Schleife vorzeitig verlassen werden. Mit der **continue**-Anweisung veranlasst man C#, den aktuellen Schleifendurchgang zu beenden und sofort mit dem nächsten zu beginnen.

In folgendem Beispielprogramm zur (relativ primitiven) Primzahlendiagnose wird die schon in Abschnitt 3.4.1 erwähnte Ausnahmebehandlung per **try-catch** - Anweisung eingesetzt, die später in einem eigenen Kapitel ausführlich behandelt wird. Während wir in der Regel der Einfachheit halber auf eine Prüfung der Eingabedaten verzichten, findet im aktuellen Programm vor allem deshalb eine praxisnahe Validierung statt, weil sich dabei ein sinnvoller **continue**-Einsatz ergibt:

```
using System;
class Primitiv {
  static void Main() {
    bool tg;
    ulong i, mtk, zahl;
    Console.WriteLine("Einfacher Primzahlendetektor\n");
    while (true) {
      Console.Write("Zu untersuchende positive Zahl oder 0 zum Beenden: ");
        try {
          zahl = Convert.ToUInt64(Console.ReadLine());
        } catch {
          Console.WriteLine("\aKeine Zahl (im zulässigen Bereich)!\n");
          continue;
        }
      if (zahl == 0)
        break;
      tg = false;
      mtk = (ulong) Math.Sqrt(zahl);  //Maximaler Teiler-Kandidat
```

```
      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!");
  }
}
```

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. Durch Eingabe der Zahl Null kann das Beispielprogramm beendet werden, wobei die absichtlich konstruierte **while** - „Endlosschleife" per **break** verlassen wird.

Man hätte die **continue**- und die **break**-Anweisung zwar vermeiden können (siehe Übungsaufgabe in Abschnitt 3.7.4), doch werden bei dem vorgeschlagenen Verfahren lästige Sonderfälle (unzulässige Werte, Null als Terminierungssignal) auf besonders übersichtliche Weise abgehakt, bevor der Kernalgorithmus startet.

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 (≥ 1) 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 keinen echten Teiler kleiner oder gleich $\sqrt{z}$ hat, kann man auch jenseits dieser Grenze keinen finden, und z ist eine Primzahl.

Zur Berechnung der Wurzel verwendet das Beispielprogramm die Methode **Sqrt()** aus der Klasse **Math**, über die man sich bei Bedarf in der FCL-Dokumentation informieren kann.

3.7.4 Übungsaufgaben zu Abschnitt 3.7

1) In einer Lotterie gilt folgender Gewinnplan:

- 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 in folgendem 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`?

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
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);
   }
}
```

</td><td>

```
b ist True
Kontr.ausg. von b: False
```

</td></tr>
</table>

3) Erstellen Sie eine Variante des Primzahlen-Diagnoseprogramms aus Abschnitt 3.7.3.5, 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) Verbessern Sie das im Übungsaufgaben-Abschnitt 3.5.11 in Auftrag gegebene Programm zur DM-Euro - Konvertierung so, dass es nicht für jeden Betrag neu gestartet werden muss. Vereinbaren Sie mit dem Benutzer ein geeignetes Verfahren für den Fall, dass er das Programm doch irgendwann einmal beenden möchte.

6) Bei einem **double**-Wert sind maximal 16 signifikante Dezimalstellen garantiert (siehe Abschnitt 3.3.4). Folglich 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 Index *i*, für den man noch erhält:

$$1{,}0 + 2^{-i} > 1{,}0$$

In dem (zur freiwilligen Lektüre empfohlenen) Vertiefungsabschnitt 3.3.5.1 findet sich eine Erklärung für das Ergebnis.

7) In dieser Aufgabe sollen Sie verschiedene Varianten von Euklids Algorithmus zur Bestimmung des größten gemeinsamen Teilers (GGT) zu zwei natürlichen Zahlen *u* und *v* implementieren und die Performanzunterschiede messen. Verwenden Sie als ersten Kandidaten den im Einführungsbeispiel zum Kürzen von Brüchen (Methode `Kuerze()`) benutzten Algorithmus (siehe Abschnitt

1.1.2). Sein offensichtliches Problem besteht darin, dass bei stark unterschiedlichen Zahlen u und v sehr viele Subtraktions-Operationen erforderlich werden.

In der meist benutzten Variante des Euklidischen Verfahrens wird dieses Problem vermieden, indem an Stelle der Subtraktion die Modulo-Operation zum Einsatz kommt, basierend auf dem folgendem 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 1.1.3): Für natürliche Zahlen u und v mit $u > v$ gilt:

$$x \text{ ist gemeinsamer Teiler von } u \text{ und } v$$
$$\Leftrightarrow$$
$$x \text{ ist gemeinsamer Teiler von } u \text{ und } u \,\%\, v$$

Der GGT-Algorithmus per Modulo-Operation läuft für zwei natürliche Zahlen u und v ($u \geq v > 0$) folgendermaßen ab:

> Es wird geprüft, ob u durch v 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 $u \geq v$ ist nicht wesentlich, weil beim Start mit $u < v$ der erste Algorithmusschritt die beiden Zahlen vertauscht.

Um den Zeitaufwand für beide Varianten zu messen, kann man z.B. mit der Struktur **DateTime** aus dem Namensraum **System** arbeiten. Mit ihrer statischen **Now**-Eigenschaft ermittelt man den aktuellen Zeitpunkt und fragt dann die **Ticks**-Eigenschaft dieser **DateTime**-Instanz ab. Der letzte Satz ist zugegebenermaßen recht kompliziert geraten und enthält etliche noch unzureichend erklärte Details. Man erhält jedenfalls mit einer recht einfachen Syntax die Anzahl der 100-Nanosekunden - Intervalle, die seit dem 1. Januar 0001, 00:00:00 vergangen sind, z.B.:

```
long zeit = DateTime.Now.Ticks;
```

Für die Beispielwerte u = 999000999 und v = 36 liefern beide Euklid-Varianten sehr verschiedene Laufzeiten (CPU: Intel Core i3 550):

GGT-Bestimmung mit Euklid (Differenz)	GGT-Bestimmung mit Euklid (Modulo)
Erste Zahl: 999000999	Erste Zahl: 999000999
Zweite Zahl: 36	Zweite Zahl: 36
GGT: 9	GGT: 9
Benöt. Zeit: **201,1719** Millisek.	Benöt. Zeit: **0,9765** Millisek.

4 Klassen und Objekte

Softwareentwicklung mit C# besteht im Wesentlichen aus der Definition von **Klassen**, die aufgrund der vorangegangenen objektorientierten Analyse …

- als Baupläne für Objekte
- und/oder als Akteure

konzipiert werden können. Wenn ein spezieller Akteur im Programm nur *einfach* benötigt wird, kann eine handelnde Klasse diese Rolle übernehmen. Sind hingegen mehrere Individuen einer Gattung erforderlich (z.B. mehrere Brüche im 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 vereinbart bzw. entworfen.

In den Methoden eines Programms werden vordefinierte (z.B. der Standardbibliothek entstammende) oder selbst erstellte Klassen zur Erledigung von Aufgaben verwendet. Meist werden dabei Objekte aus Klassen mit Bauplancharakter erzeugt und mit Aufträgen versorgt. Mit dem „Beauftragen" eines Objekts oder einer Klasse bzw. mit dem „Zustellen einer Botschaft" ist nichts anderes gemeint als ein Methodenaufruf.

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 Softwaretechnologie. Für die in Abschnitt 1 speziell für größere Projekte empfohlene objektorientierte *Analyse* (z.B. mit Hilfe der UML) ist dabei leider keine Zeit (siehe z.B. Balzert 1999).

4.1 Überblick, historische Wurzeln, Beispiel

4.1.1 Einige Kernideen und Vorzüge der OOP

Lahres & Rayman (2009) 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:

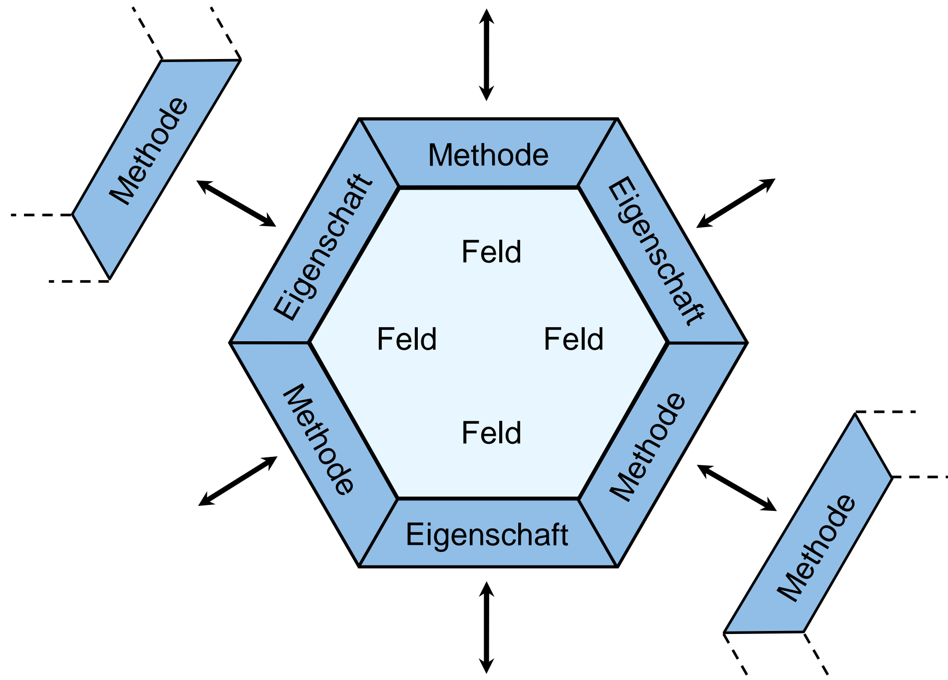
- **Datenkapselung**
  Mit diesem Thema haben wir uns bereits beschäftigt. Das vorhandene Wissen soll gleich vertieft und gefestigt werden.
- **Vererbung**
  Dieses OOP-Element wird gleich vorgestellt und später ausführlich behandelt.
- **Polymorphie**
  Weil die Vorteile der Polymorphie mit den bisherigen Beispielen nicht plausibel vermittelt werden können, schieben wir die Behandlung dieses Themas noch etwas auf.

4.1.1.1 *Datenkapselung und Modularisierung*

In der objektorientierten Programmierung (OOP) wird die traditionelle Trennung von Daten und Operationen aufgegeben. Hier besteht ein Programm 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**, **information hiding**) und so vor ungeschickten Zugrif-

[1] Diese Darstellung ist etwas vereinfachend. Hinter einer Eigenschaft muss nicht unbedingt ein einzelnes Feld stehen.

fen schützen. Die Methoden und Eigenschaften einer Klasse sind hingegen von Außen ansprechbar und bilden ihre **Schnittstelle**. Dies kommt in der folgenden Abbildung zum Ausdruck, die Sie im Wesentlichen schon aus Abschnitt 1 kennen:



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 4.2.5) und damit vor Veränderungen geschützt. Ein Beispiel ist das **double**-Feld **PI** für die Kreiszahl π in der FCL-Klasse **Math**.

Klassen mit Datenkapselung realisieren besser als frühere Software-Technologien (siehe Abschnitt 4.1.2) 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 Projekten mit hoher Komplexität.

Aus der Datenkapselung und der Modularisierung ergeben sich gravierende Vorteile für die Softwareentwicklung:

- **Vermeidung von Fehlern, Erleichterung der Fehlersuche**
  Direkte Schreibzugriffe auf die Felder einer Klasse bleiben den klasseneigenen Methoden und Eigenschaften vorbehalten, die vom Designer der Klasse sorgfältig entworfen wurden. Damit sollten Programmierfehler seltener werden. In unserem `Bruch`-Beispiel haben wir dafür gesorgt, dass unter keinen Umständen der Nenner eines Bruches auf Null gesetzt wird. Anwender unserer Klasse können einen Nenner einzig über die Eigenschaft `Nenner` verändern, die aber den Wert Null nicht akzeptiert. Bei einer anderen Klasse kann es erforderlich sein, dass für eine Gruppe von Feldern bei jeder Änderung gewissen Konsistenzbedingungen eingehalten werden. Treten in einem Programm trotz Datenkapselung Fehler wegen pathologischer Variablenausprägungen auf, sind diese relativ leicht zu lokalisieren, weil nur wenige Methoden verantwortlich sein können. Per Datenkapselung werden also die **Kosten reduziert**, die durch Programmierfehler entstehen.

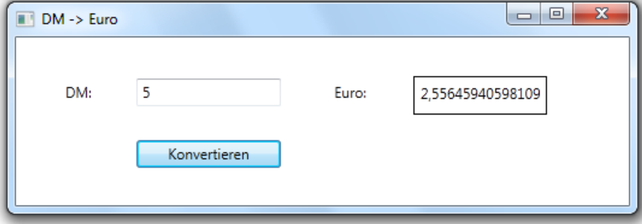
[1] Deutsche Übersetzung: Teile und herrsche!

- **Produktivität durch wiederholt und bequem verwendbare Klassen**
  Selbständig agierende Klassen, die ein Problem ohne überflüssige Anhängigkeiten von anderen Programmbestandteilen lösen, sind potenziell in vielen Projekten zu gebrauchen (Wiederverwendbarkeit). Wer als Programmierer eine Klasse verwendet, braucht sich um deren inneren Aufbau nicht zu kümmern, so dass neben dem Fehlerrisiko auch der Einarbeitungsaufwand sinkt. Wir werden z.B. in GUI-Programmen einen recht kompletten Rich-Text - Editor über eine Klasse aus der Standardbibliothek integrieren, ohne wissen zu müssen, wie Text und Textauszeichnungen intern verwaltet werden.
- **Flexibilität bei der Implementierung**
  Datenkapselung schafft günstige Voraussetzungen für die Wartung bzw. Verbesserung einer Klassendefinition. Solange die Methoden und Eigenschaften der Schnittstelle unverändert bzw. kompatibel bleiben, kann die interne Architektur einer Klasse ohne Nebenwirkungen auf andere Programmteile geändert werden.
- **Erfolgreiche Teamarbeit durch abgeschottete Verantwortungsbereiche**
  In großen Projekten können mehrere Programmierer nach der gemeinsamen Definition von Schnittstellen relativ unabhängig an verschiedenen Klassen arbeiten.

4.1.1.2 Vererbung

Zu den Vorzügen der „super-modularen" Klassenkonzeption gesellt sich in der OOP ein Vererbungsverfahren, das beste Voraussetzungen für die Erweiterung von Softwaresystemen bei rationeller Wiederverwendung der bisherigen Code-Basis schafft: Bei der Definition einer neuen Klasse können alle Merkmale und Handlungskompetenzen (Methoden, Eigenschaften) einer *Basisklasse* übernommen werden. Es ist also leicht möglich, ein Softwaresystem 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.

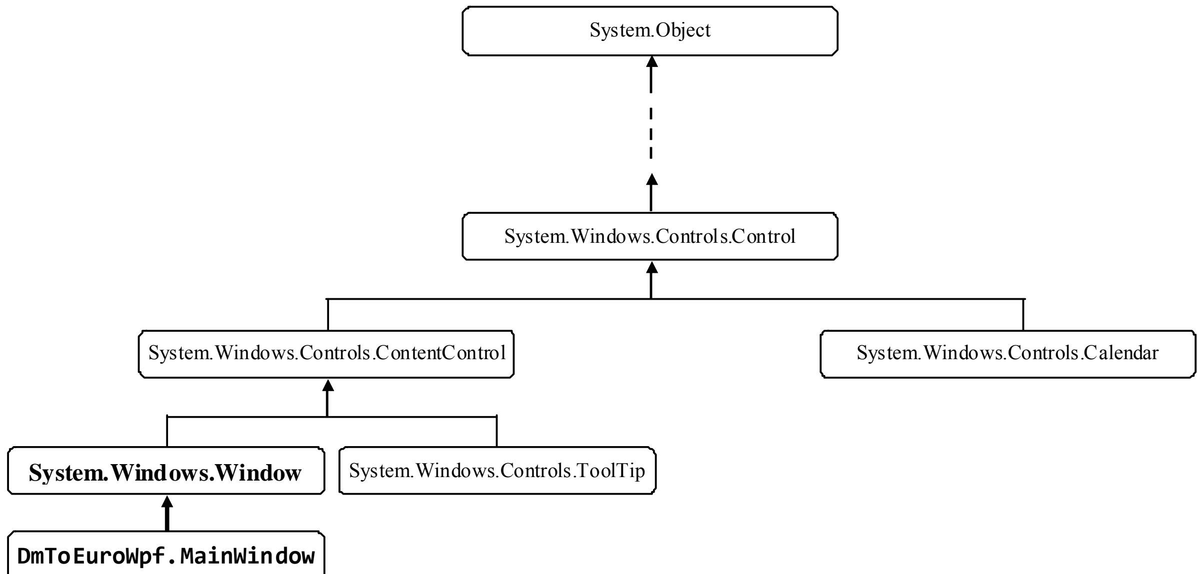
Bei den in Abschnitt 2.2.1.3 erstellten Währungskonverter mit graphischer Benutzerschnittelle in WPF-Technik



haben wir mit Assistentenhilfe die Klasse `MainWindow` definiert als Ableitung der WPF-Klasse **Window**:

```
public partial class MainWindow : Window {
   public MainWindow() {
      InitializeComponent();
   }
   private void button1_Click(object sender, RoutedEventArgs e) {
      double betrag = Convert.ToDouble(textBox1.Text);
      label3.Content = Convert.ToString(betrag / 1.98853);
   }
}
```

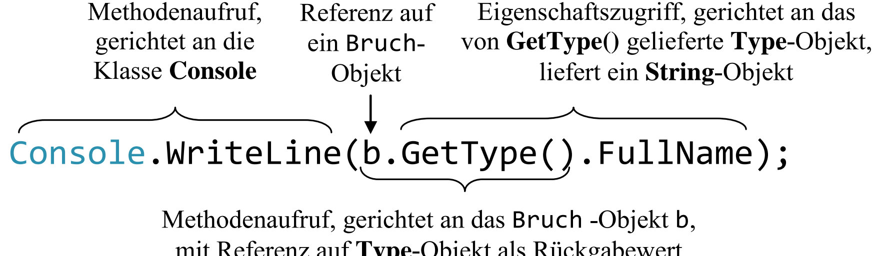
Die Basisklasse **Window** ist selbst Bestandteil eines komplexen Stammbaums, von dem hier nur winziger Ausschnitt zu sehen ist:



Im .NET – Framework 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 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 befragen. Im folgenden Programm wird ein `Bruch`-Objekt (vgl. Abschnitt 1.1) um die entsprechende Auskunft gebeten:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>using System;
class Prog {
   static void Main() {
      Bruch b = new Bruch();
      Console.WriteLine(b.GetType().FullName);
   }
}</pre></td><td><pre>Bruch</pre></td></tr>
</table>

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 **FullName** mitzuteilen. Diese Zeichenfolge mit dem Typnamen (ein Objekt der Klasse **String**[1]) bildet schließlich den Aktualparameter 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 (**FullName**), einer expliziten Referenzvariablen (`b`) und einer impliziten Referenz (**GetType()**-Rückgabewert) genau zu erläutern:



[1] Eine Besonderheit der Klasse **String** ist der aus Bequemlichkeitsgründen vom Compiler unterstützte Aliasname **string** (mit kleinem Anfangsbuchstaben).

Der **FullName**-Eigenschaftszugriff ist im Beispiel nicht unbedingt 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.

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 Objekte beherrschen dank Vererbung die Methode **GetType()**, obwohl in der `Bruch`-Klassendefinition nichts davon zu sehen ist.

4.1.1.3 Realitätsnahe Modellierung

Klassen sind nicht nur ideale Bausteine für die rationelle Konstruktion von Softwaresystemen, sondern erlauben auch eine gute Modellierung des Anwendungsbereichs. In der zentralen Projektphase der objektorientierten Analyse und Modellierung sprechen Softwareentwickler und Auftraggeber dieselbe Sprache, so dass 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 der virtuellen Welt des Computers repräsentieren (z.B. Bildschirmfenster, Mausereignisse).

4.1.2 Strukturierte Programmierung und OOP

In vielen klassischen Programmiersprachen (z.B. C, Fortran, Pascal) sind zur Strukturierung von Programmen zwei Techniken verfügbar, die in weiterentwickelter Form auch bei der OOP genutzt werden:

- **Unterprogrammtechnik**
  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 kompaktere und übersichtlichere Programme, die leichter erstellt, analysiert, korrigiert und erweitert werden können. Praktisch alle traditionellen 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. Mit dem Datentyp **struct** der Programmiersprache C oder dem analogen Datentyp **record** der Programmiersprache Pascal lassen sich problemadäquate Datentypen mit mehreren Bestandteilen konstruieren, die jeweils einen beliebigen, bereits bekannten Typ haben dürfen. So eignet sich etwa für ein Programm zur Adressenverwaltung ein neu definierter Datentyp mit Elementen für Name, Vorname, Telefonnummer etc. Alle Adressinformationen zu einer Person lassen sich dann in *einer* Variablen vom selbst defi-

nierten Typ speichern. Dies vereinfacht z.B. das Lesen, Kopieren oder Schreiben solcher Daten.

Die problemadäquaten Datentypen der älteren Programmiersprachen werden in C# 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 Fortschritte:

- 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 et al. 2000, S. 21), was bei Fehlern zu einer aufwändigen Suche entlang der Verarbeitungskette führen kann.
- Bessere Abbildung des Anwendungsbereichs
- Rationellere (Weiter-)Entwicklung von Software durch die Vererbungstechnik
- Mehr Bequemlichkeit für Bibliotheksbenutzer
  Jede rationelle Softwareproduktion greift in hohem Maß auf Bibliotheken mit bereits vorhandenen Lösungen zurück. Dabei sind die Klassenbibliotheken der OOP einfacher zu verwenden als klassische Unterprogrammbibliotheken.

4.1.3 Auf-`Bruch` zu echter Klasse

In den Beispielprogrammen von Abschnitt 3 wurde mit der Klassendefinition lediglich eine in C# unausweichliche formale Anforderung an Programme erfüllt. Die in Abschnitt 1.1 vorgestellte Klasse `Bruch` realisiert hingegen wichtige Prinzipien der objektorientierten Programmierung. Sie wird nun wieder aufgegriffen und in verschiedenen Varianten bzw. Ausbaustufen als Beispiel verwendet. Darauf basierende Programme sollen Schüler beim Erlernen der Bruchrechnung unterstützen. Eine objektorientierte Analyse der Problemstellung ergab, dass in einer elementaren Ausbaustufe des Programms lediglich eine Klasse zur Repräsentation von Brüchen benötigt wird. Später sind weitere Klassen (z.B. Aufgabe, Übungsaufgabe, Testaufgabe, Schüler, Lernepisode, Testepisode, Fehler) zu ergänzen.

Wir nehmen nun bei der `Bruch`-Klassendefinition im Vergleich zur Variante in Abschnitt 1.1 einige Verbesserungen vor:

- Als zusätzliches Feld erhält jeder Bruch ein `etikett` vom Datentyp **String**. Damit wird eine beschreibende Zeichenfolge verwaltet, die z.B. beim Aufruf der Methode `Zeige()` neben den Feldern `zaehler` und `nenner` auf dem Bildschirm erscheint. Objekte der erweiterten `Bruch`-Klasse besitzen also auch eine Instanzvariable mit *Referenztyp* (neben den **int**-Feldern `zaehler` und `nenner`).
- Weil die `Bruch`-Klasse 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 Feldes 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 Auf-

rufer 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 (vgl. Übungsaufgabe in Abschnitt 3.7.4).

Im folgenden Quellcode der erweiterten `Bruch`-Klasse 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 dem 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("Zaehler: ");
            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 = "";
         for (int i=1; i <= etikett.Length; i++)
            luecke = luecke + " ";
         Console.WriteLine(" {0}   {1}\n {2} -----\n {0}   {3}\n",
                           luecke, zaehler, etikett, nenner);
      }
   }
```

Im Unterschied zur Präsentation in Abschnitt 1.1 wird die `Bruch`-Klassendefinition anschließend gründlich erläutert. Dabei machen die in Abschnitt 4.2 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, so dass wir uns in Abschnitt 4.3 auf elementare Themen beschränken und später noch wichtige Ergänzungen behandeln.

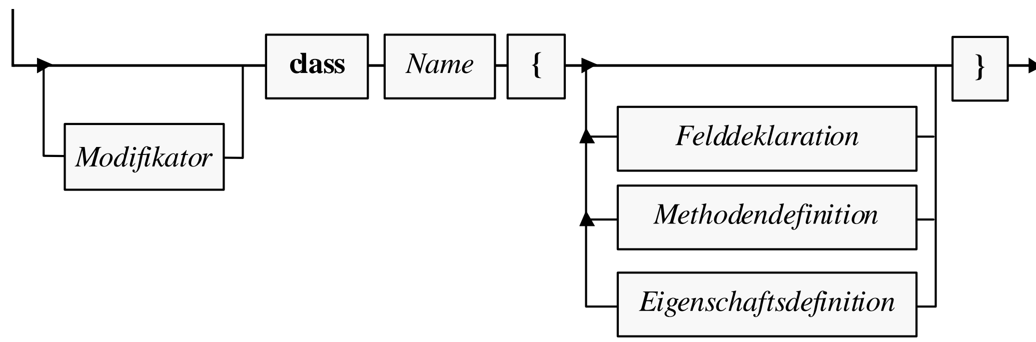
Jede .NET – Anwendung benötigt eine Startklasse mit statischer **Main()**-Methode, die von der CLR (*Common Language Runtime*) beim Programmstart aufgerufen wird. Für die bei diversen Demonstrationen in den folgenden Abschnitten verwendeten Startklassen (mit jeweils spezieller Implementation) werden wir den Namen `BruchRechnung` verwenden, z.B.:

```
using System;
class BruchRechnung {
   static void Main() {
      Console.WriteLine("Kürzen von Brüchen\n------------------\n");
      Bruch b = new Bruch();
      b.Frage();
      b.Kuerze();
      b.Etikett = "Der gekürzte Bruch:";
      b.Zeige();
   }
}
```

In der **Main()**-Methode dieses Programms zum Kürzen von Brüchen wird ein Objekt aus der Klasse `Bruch` erzeugt und mit Aufträgen versorgt.

Wir arbeiten weiterhin mit dem aus Abschnitt 3.1.2.1 bekannten Syntaxdiagramm zur Klassendefinition, das aus didaktischen Gründen einige Vereinfachungen enthält:

Klassendefinition



Zwei Bemerkungen zum Kopf einer Klassendefinition:

- Im Beispiel ist die Klasse `Bruch` 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. `BruchRechnung`), schreibt man der besseren Lesbarkeit wegen die Anfangsbuchstaben aller Wörter groß (*Pascal-Casing*, siehe Abschnitt 3.1.5).

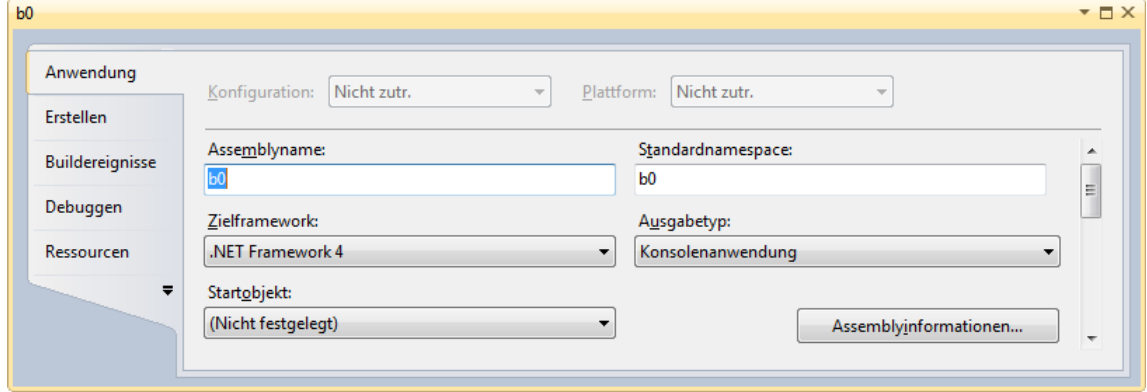
Hinsichtlich der Dateiverwaltung wird vorgeschlagen:

- 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 bei Verwendung der Entwicklungsumgebung Visual C# 2010 Express Edition nach dem Menübefehl

Projekt > Eigenschaften

ändern kann, wenn die vom Assistenten für neue Projekte vergebene Voreinstellung nicht (mehr) gefällt, z.B.:



Im Beispiel entsteht die Assembly-Datei **b0.exe**.

4.2 Instanzvariablen

Die Instanzvariablen (bzw. -felder) einer Klasse besitzen viele Gemeinsamkeiten mit den *lokalen* Variablen, die wir im Abschnitt 3 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.

4.2.1 Sichtbarkeitsbereich, Existenz und Ablage im Hauptspeicher

Von den lokalen Variablen unterscheiden sich die Instanzvariablen (Felder) einer Klasse vor allem bei der *Zuordnung* (vgl. Abschnitt 3.3.3):

- 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:

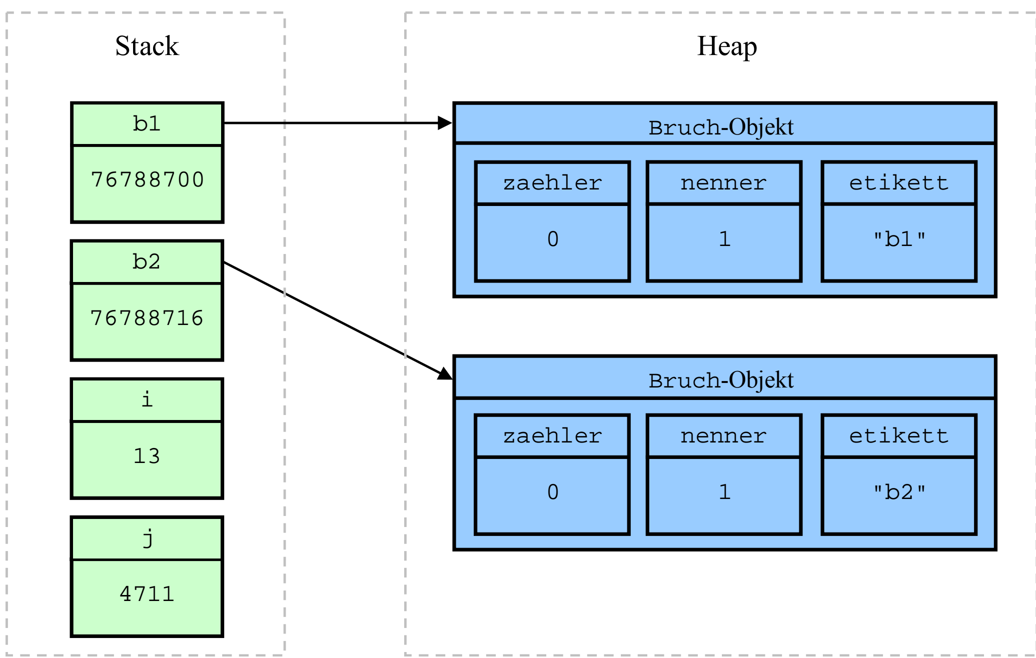
	lokale Variable	**Instanzvariable**
Sichtbarkeit	Eine lokale Variable ist nur in ihrer eigenen Methode sichtbar. Nach der Deklarationsanweisung kann sie in den restlichen Anweisungen des lokalsten Blocks angesprochen werden. Ein eingeschachtelter Block gehört zum Sichtbarkeitsbereich des umgebenden Blocks.	Die Instanzvariablen eines Objekts sind in allen Methoden sichtbar, die eine Referenz zum Objekt kennen. Je nach Schutzstufe ist Methoden fremder Klassen der Zugriff jedoch verwehrt. Instanzvariablen werden in klasseneigenen Instanzmethoden durch gleichnamige lokale Variablen überlagert, können jedoch über das vorgeschaltete Schlüsselwort **this** weiter angesprochen werden (siehe Abschnitt 4.4.5.3).
Lebensdauer	Sie existiert von der Deklaration bis zum Verlassen des lokalsten Blocks.	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 vorhanden ist.
Ablage im Speicher	Sie wird auf dem so genannten **Stack** (deutsch: *Stapel*) abgelegt. Innerhalb des programmeigenen Speichers dient dieses Segment zur Verwaltung 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.

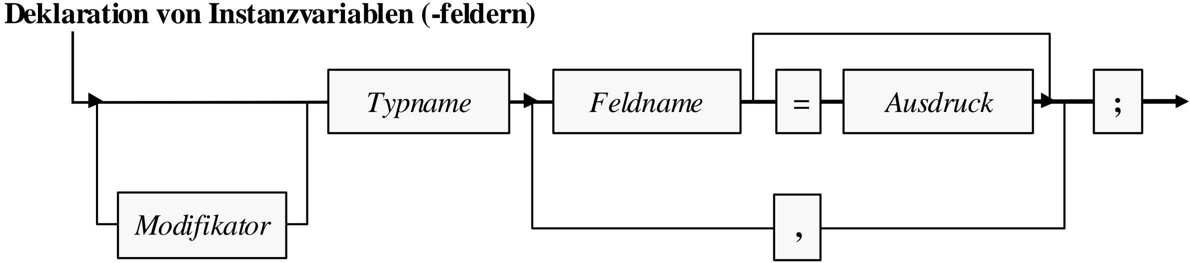


4.2.2 Deklaration mit Wahl der Schutzstufe

Während lokale Variablen im Anweisungsteil einer Methode (oder Eigenschaft) deklariert werden, erscheinen die Deklarationen der Instanzvariablen in der Klassendefinition *außerhalb* jeder Methoden- oder Eigenschaftsdefinition. Man sollte die Instanzvariablen der Übersichtlichkeit halber am Anfang der Klassendefinition deklarieren, wenngleich der Compiler auch ein späteres Erscheinen akzeptiert.

Bei der Deklaration von Instanzvariablen werden zur Spezifikation der **Schutzstufe** (und für andere Zwecke) oft Modifikatoren benötigt (auch mehrere), so dass die Syntax im Vergleich zur Deklaration einer lokalen Variablen entsprechend erweitert werden muss.

Deklaration von Instanzvariablen (-feldern)



In C# besitzen alle Instanzvariablen per Voreinstellung die Schutzstufe **private**, so dass sie nur von klasseneigenen Methoden bzw. Eigenschaften angesprochen werden können. Weil bei den `Bruch`-Feldern diese voreingestellte Datenkapselung erwünscht ist, kommen 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 jeweils eine zugehörige Eigenschaft vorhanden.

Auf den ersten Blick scheint die Datenkapselung nur beim Nenner eines Bruches 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 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 Zähler auf eine Primzahl gesetzt wird.

Trotz ihrer überzeugenden Vorteile soll die Datenkapselung nicht zum Dogma erhoben werden. Sie ist überflüssig, wenn bei einem Feld Lese- und Schreibzugriff erlaubt sind und eine Änderung des Datentyps nicht in Frage kommt. Um *allen* Klassen den Direktzugriff auf ein Feld zu erlauben, wird in seiner Deklaration der Modifikator **public** angegeben, z.B.:

```
public int Zaehler;
```

In Abschnitt 4.9 finden Sie eine Tabelle mit allen verfügbaren Schutzstufen und zugehörigen Modifikatoren.

Bei der Benennung von Instanzvariablen haben sich folgende Regeln etabliert:

- Für Felder mit den Schutzstufen **private**, **protected** oder **internal** wird das *Camel Casing* verwendet.
- Für die (seltenen) Felder mit der Schutzstufe **public** wird das *Pascal Casing* verwendet.

Um private Instanzvariablen gut von lokalen Variablen und Formalparametern (siehe unten) unterscheiden zu können, verwenden manche Programmierer ein Präfix, z.B.:

```
int m_nenner; //das m steht für "Member"
int _nenner;
```

4.2.3 Initialisierung

Während bei lokalen Variablen der Programmierer für die Initialisierung verantwortlich ist, erhalten die Instanzvariablen eines neuen Objekts automatisch folgende Startwerte, falls der Programmierer nicht eingreift:

Datentyp	**Initialisierung**
sbyte, **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**	0
float, **double**, **decimal**	0,0
char	0 (Unicode-Zeichennummer)
bool	**false**
Referenztyp	**null**

Im `Bruch`-Beispiel wird nur die automatische `zaehler`-Initialisierung unverändert übernommen:

- Beim `nenner` eines Bruches wäre die Initialisierung auf Null bedenklich, weshalb eine explizite Initialisierung auf den Wert Eins vorgenommen wird.
- Wie noch näher zu erläutern sein wird, ist **string** in C# *kein* primitiver 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 Wert **null**, zeigt also auf nichts. Weil der `etikett`-Wert **null** z.B. beim Aufruf der `Bruch`-Methode `Zeige()` einen Laufzeitfehler (**NullReferenceException**) zu Folge hätte, wird ein **String**-Objekt mit einer leeren Zeichenfolge erstellt und zur `etikett`-Initialisierung verwendet. Das Erzeugen des

String-Objekts erfolgt *implizit*, indem der **String**-Variablen `etikett` ein Zeichenfolgen-Literal zugewiesen wird.

4.2.4 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()` zu beobachten ist:

```
Console.WriteLine(" {0}   {1}\n {2} -----\n {0}   {3}\n",
                  luecke, zaehler, etikett, nenner);
```

Im Beispiel zeigt sich syntaktisch kein Unterschied zwischen dem Zugriff auf die Instanzvariablen (`zaehler`, `nenner`, `etikett`) und dem Zugriff auf die lokale Variable (`luecke`). Gelegentlich kann es (z.B. der Klarheit halber) sinnvoll sein, einem Instanzvariablennamen über das Schlüsselwort **this** (vgl. Abschnitt 4.4.5.3) eine Referenz auf das handelnde Objekt voranzustellen, z.B.:

```
Console.WriteLine(" {0}   {1}\n {2} -----\n {0}   {3}\n",
                  luecke, this.zaehler, this.etikett, this.nenner);
```

Beim Zugriff auf eine Instanzvariable eines *anderen* Objektes *derselben* Klasse muss dem Variablennamen eine Referenz auf das Objekt vorangestellt werden, wobei die Bezeichner durch den **Punktoperator** zu trennen sind. In der folgenden Zeile 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:

```
nenner = nenner * b.nenner;
```

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 mög-
lich
```

In diesem Abschnitt wurden eine Syntaxregel und ein Designprinzip angesprochen:

- Instanzvariablen des handelnden Objekts können über den Variablennamen angesprochen werden. Beim Zugriff auf Instanzvariablen eines anderen Objekts ist eine Referenzvariable erforderlich.
- Direkte Zugriffe auf Instanzvariablen sind in C# per Voreinstellung nur klasseneigenen Methoden erlaubt. Diese Schutzstufe **private** sollte in der Regel beibehalten werden.

4.2.5 Konstante und schreibgeschützte Felder

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 3.3.8) sondern auch Felder einer Klasse als konstant deklariert werden, wobei der initialisierende Ausdruck *zur Übersetzungszeit* berechenbar sein muss.

Im Zusammenhang mit dem OOP-Prinzip der Datenkapselung sind *konstante und öffentliche* Felder oft eine sinnvolle Möglichkeit, um fremden Klassen ohne Risiko einen syntaktisch einfachen *Lese*zugriff zu gestatten. Im Vergleich zu einer ausschließlich lesend zu verwendenden Eigenschaft besteht der Vorteil des schnelleren Zugriffs, weil kein Methodenaufruf im Spiel ist. In der FCL-Klasse **Math** (Namensraum **System**) ist z.B. das **double**-Feld **PI** als **public** (allgemein verfügbar) und **const** deklariert:

```
public const double PI = 3.14159265358979323846;
```

Konstante Felder einer Klasse sind grundsätzlich statisch, wobei der (somit hier überflüssige) Modifikator **static** *nicht* zusammen mit dem Modifikator **const** verwendet werden darf. Genau genommen passt die Behandlung des Feld-Modifikators **const** also nicht in den Abschnitt 4.2 über Instanzvariablen. Bei Referenztyp-Konstanten ist mit Ausnahme der Klasse **String** als Initialisierungswert lediglich **null** erlaubt, so dass bei konstanten Feldern nur die elementaren Datentypen und der Typ **String** sinnvoll sind.

Soll eine Instanzvariable (von beliebigem Typ) *zur Laufzeit* in einem so genannten Konstruktor (siehe Abschnitt 4.4.3) initialisiert und dann fixiert werden, verwendet man den Modifikator **readonly**, z.B.:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
using System;
class Klaas {
  public readonly int Readomo;
  public Klaas(int i) {
    Readomo = i;
  }
}
class Prog {
  static void Main() {
    Klaas p = new Klaas(7);
    Console.WriteLine(p.Readomo);
  }
}
```

</td><td>

```
7
```

</td></tr>
</table>

Bei den Konstruktoren handelt es sich um spezielle Methoden, die den Namen ihrer Klasse tragen und keinen Rückgabetyp besitzen (auch nicht **void**).

Während ein **const**–deklariertes Feld stets klassenbezogen ist, kann ein **readonly**-deklariertes Feld objekt- oder klassenbezogen sein.

4.3 Instanzmethoden

In einer Bauplan-Klassendefinition werden Objekte entworfen, die eine Anzahl von Verhaltenskompetenzen (Methoden) besitzen, die von anderen Programmbestandteilen (Klassen oder Objekten) per Methodenaufruf genutzt werden können. Objekte sind also Dienstleister, die eine Reihe von Nachrichten interpretieren und mit passendem Verhalten beantworten können.

Wie es im Objekt drinnen aussieht, geht andere Programmierer nichts an (*information hiding*). Seine Instanzvariablen sind bei konsequenter Datenkapselung für Objekte (bzw. Methoden) fremder Klassen unsichtbar. 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 4.5).

Beim Aufruf einer Methode ist oft über so genannte *Parameter* die gewünschte Verhaltensweise näher zu spezifizieren, und bei 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 gut repräsentieren sollen. Wenn ein Programmierer z.B. ein Objekt aus unserer `Bruch`-Klasse verwendet, kann er zur Bildschirmausgabe auf die Methode `Zeige()` zurückgreifen:

```
public void Zeige() {
   string luecke = "";
   for (int i = 1; i <= etikett.Length; i++)
      luecke = luecke + " ";
     Console.WriteLine(" {0}   {1}\n {2} -----\n {0}   {3}\n",
                       luecke, zaehler, etikett, nenner);
}
```

Weil diese Methode auch für fremde Klassen verfügbar sein soll, wird per Modifikator die Schutzstufe **public** gewählt.

Da es vom Verlauf einer Bildschirmausgabe nichts zu berichten gibt, liefert `Zeige()` keinen Rückgabewert. Folglich ist im Kopf der Methodendefinition der Rückgabetyp **void** anzugeben.

Weil die `Zeige()`-Methode nur eine einzige Arbeitsweise beherrscht, sind keine Parameter vorhanden, wobei die leere Parameterliste aber weder bei der Definition noch beim Aufruf der Methode fehlen darf. Leere Rückgaben und Parameterlisten sind keinesfalls die Regel, so dass wir uns bald näher mit den Kommunikationsregeln beim Methodenaufruf beschäftigen müssen.

In der `Zeige()` - Implementierung gelingt mit einfachen Mitteln eine brauchbar formatierte Konsolenausgabe von etikettierten Brüchen, z.B.:

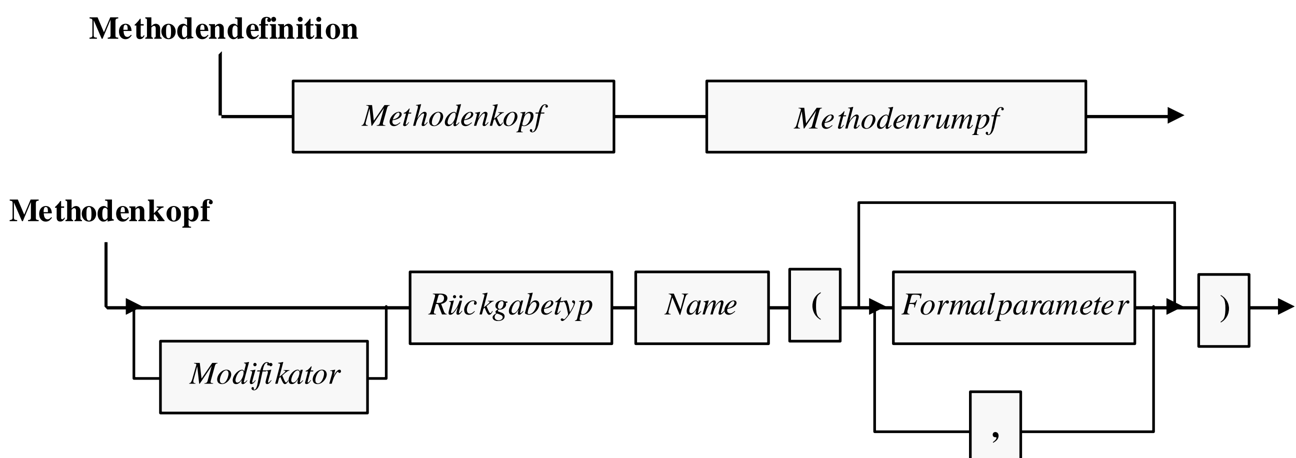
```
                 13
Erster Bruch:  -----
                221
```

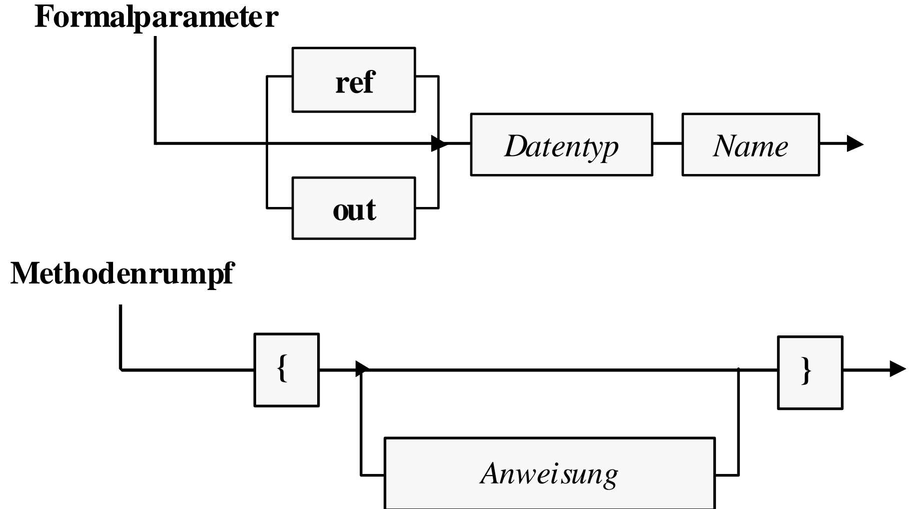
Über und unter dem Etikett steht das **String**-Objekt `luecke`, das als leere Zeichenfolge initialisiert wird. In der **for**-Schleife werden per Plusoperator Leerzeichen angehängt, bis die Länge des Etiketts erreicht ist. Diese Länge wird über die Eigenschaft **Length** des **String**-Objekts `etikett` ermittelt.

Während jedes Objekt einer Klasse seine eigenen Instanzvariablen auf dem Heap besitzt, ist der MSIL-Code der Instanzmethoden jeweils nur *einmal* im Speicher vorhanden und wird von allen Objekten verwendet.

4.3.1 Methodendefinition

Die folgende Serie von Syntaxdiagrammen zur Methodendefinition unterscheidet sich von der Variante in Abschnitt 3.1.2.2 durch eine genauere Erklärung der Formalparameterliste:





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 beim Aufruf zu erkennen.

Während sich bei *Feldern* die Groß-/Kleinschreibung des Anfangsbuchstabens nach einer generell akzeptierten Konvention an der Schutzstufe orientiert (Camel Casing für Felder mit den Schutzstufen **private**, **protected** oder **internal**, Pascal Casing für öffentliche Felder, vgl. Abschnitt 4.2.2), sind die Empfehlungen für Methodennamen weniger einheitlich. Auf der Webseite

http://msdn.microsoft.com/de-de/library/4df752aw%28en-us,VS.71%29.aspx

empfiehlt Microsoft offenbar unabhängig von der Schutzstufe mit einem Großbuchstaben zu beginnen (Pascal Casing):

> .NET Framework General Reference
> **Method Naming Guidelines**
> The following rules outline the naming guidelines for methods:
>
> - Use verbs or verb phrases to name methods.
> - Use Pascal case.

Die Quellcode-Generatoren (Assistenten) im Visual Studio produzieren bei *privaten* Methoden jedoch Namen mit kleinen Anfangsbuchstaben, z.B.:

```
private void button1_Click(object sender, RoutedEventArgs e) { ... }
```

4.3.1.1 Modifikatoren

Bei einer Methodendefinition kann per Modifikator der voreingestellte **Zugriffsschutz** verändert werden. In C# gilt für Methoden gilt wie für Instanzvariablen:

- Voreingestellt ist die Schutzstufe **private**, so dass 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. Im `Bruch`-Beispiel sind alle Methoden für die Verwendung durch beliebige fremde Klassen frei gegeben.

Später werden auch noch Modifikatoren vorgestellt, die nicht den Zugriffsschutz regeln, sondern eine andere Bedeutung haben.

4.3.1.2 Rückgabewerte und return-Anweisung

Für den Informationstransfer von einer Methode an ihren Aufrufer kann neben Referenz- und Ausgabeparametern (siehe Abschnitt 4.3.1.3) 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 in Abschnitt 3.5.2 gelernt, 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 abliefert.

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()`, 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**):

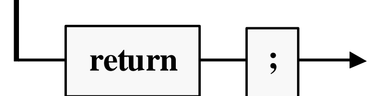
```
public bool Frage() {
   try {
      Console.Write("Zaehler: ");
      int z  = Convert.ToInt32(Console.ReadLine());
      Console.Write("Nenner : ");
      int n = Convert.ToInt32(Console.ReadLine());
      Zaehler = z;
      Nenner = n;
      return true;
   } catch {
      return false;
   }
}
```

Ist der Rückgabetyp einer Methode von **void** verschieden, dann muss im Rumpf dafür gesorgt werden, dass jeder mögliche Ausführungspfad mit einer **return**-Anweisung endet, die einen Wert passenden Typs übergibt.

return-Anweisung für Methoden *mit* Rückgabewert

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



In der `Bruch`-Methode `Frage()` wird am Ende eines störungsfrei durchlaufenen **try**-Blocks der boolesche Wert **true** zurück gemeldet. 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 Null zu setzen, zeigt die Methode `Frage()` übrigens keine besondere Reaktion. Ein kritische Bewertung bleibt der Eigenschaft `Nenner` überlassen (siehe unten).

4.3.1.3 Formalparameter

Parameter wurden Ihnen bisher vereinfachend als Informationen über die gewünschte Arbeitsweise einer Methode vorgestellt. Tatsächlich kennt C# verschiedene Parameterarten, um den Informationsaustausch zwischen einem Aufrufer und einer angeforderten Methode in *beide* Richtungen optimal zu unterstützen.

Im Kopf der Methoden**definition** werden über so genannte **Formalparameter** Daten von bestimmtem Typ spezifiziert, die entweder den Ablauf der Methode beeinflussen, oder durch die Methode verändert werden. Beim späteren **Aufruf** der Methode sind korrespondierende **Aktualparameter** anzugeben (siehe Abschnitt 4.3.2), wobei je nach Parameterart (siehe unten) Variablen oder Ausdrücke in Frage kommen.

In den Anweisungen des Methodenrumpfs sind die Formalparameter wie lokale Variablen zu verwenden, die teilweise (je nach Parameterart) mit den beim Aufruf übergebenen Aktualparameterwerten initialisiert wurden.

Für jeden Formalparameter sind folgende Angaben zu machen:

- **Parameterart**
  Sie werden gleich Wert-, Referenz- und Ausgabeparameter kennen lernen.
- **Datentyp**
  Es sind beliebige Typen erlaubt. Man muss den Datentyp eines Formalparameters auch dann explizit angeben, wenn er mit dem Typ des Vorgängers (linken Nachbarn) übereinstimmt.
- **Name**
  Nach den Empfehlungen aus Abschnitt 3.1.5 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 zu behandeln sind, …
  - können Namenskonflikte mit anderen lokalen Variablen derselben Methode auftreten,
  - werden namensgleiche Instanz- bzw. Klassenvariablen überlagert.
    Diese bleiben jedoch über ein geeignetes Präfix (z.B. **this** bei Objekten) weiter ansprechbar.
- **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 Aktualparameter gemäß ihrer Reihenfolge den Formalparametern zugeordnet werden.

4.3.1.3.1 Wert- bzw. Eingabeparameter

Über einen Wert- bzw. Eingabeparameter werden Informationen in eine Methode kopiert, um diese mit Daten zu versorgen oder ihre Arbeitsweise zu steuern. Als Beispiel betrachten wir folgende Variante der `Bruch`-Methode `Addiere()` Das beauftragte Objekt soll den über **int**-wertige Parameter (`zpar`, `npar`) übergebenen Bruch zu seinem eigenen Wert addieren und optional (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 im Aufruf übergebenen Wert initialisiert wurde.

Methodeninterne Änderungen dieser lokalen Variablen bleiben *ohne* Effekt auf eine als Aktualparameter fungierende Variable der rufenden Programmeinheit. Im folgenden Beispiel übersteht die lokale Variable `iM` 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 iM = 4711;`
`    Prog p = new Prog();`
`    p.WertParDemo(iM);`
`    Console.WriteLine(iM);`
`  }`
`}`
	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) Konstruktion wird es vermieden, im aktuellen Abschnitt 4.3.1 über Details bei der Definition von *Instanzmethoden* zur Demonstration *statische* Methoden zu verwenden. Bei den Parametern und beim Rückgabetyp gibt es allerdings keine Unterschiede zwischen den Instanzmethoden und den Klassenmethoden (siehe Abschnitt 4.6.3).

Als Wertaktualparameter sind nicht nur (initialisierte!) Variablen erlaubt, sondern beliebige Ausdrücke mit einem Typ, der nötigenfalls erweiternd in den Typ des zugehörigen Formalparameters gewandelt werden kann.

4.3.1.3.2 Referenzparameter

Ein Referenzparameter ermöglicht es der aufgerufenen Methode, eine Variable der aufrufenden Programmeinheit zu *verändern*. Die Methode erhält beim Aufruf keine *Kopie* der betroffenen Variablen, sondern die Speicheradresse des Originals. Alle methodenintern über den Formalparameternamen vorgenommenen Modifikationen wirken sich direkt auf das Original aus.

Als Referenzaktualparameter sind nur *Variablen* erlaubt, die vom *selben* Typ wie der Formalparameter und außerdem initialisiert sein müssen. Es findet also keine implizite Typanpassung statt. Auf den garantiert definierten Wert eines Referenzaktualparameters kann die gerufene Methode (vor einer möglichen Veränderung) auch lesend zugreifen. Im Unterschied zu den Wert- bzw. Eingabe-

parametern und den gleich vorzustellenden Ausgabeparametern ermöglichen Referenzparameter also einen Informationsfluss in *beide* Richtungen.

In der Methodendefinition *und* beim Methodenaufruf sind Referenzparameter durch das Schlüsselwort **ref** zu kennzeichnen.

Im folgenden Programm (nach dem aus Abschnitt 4.3.1.3.1 bekannten Strickmuster) tauscht eine Instanzmethode die Werte zwischen den als Referenzaktualparameter übergebenen Variablen:

Quellcode

```
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);
  }
}
```

Ausgabe

```
Vorher:  x = 1, y = 2
Nachher: x = 2, y = 1
```

Abschließend noch ein Satz für Begriffsakrobaten: Wird eine *Referenzvariable* (hat als Inhalt eine Objektadresse) als Referenzaktualparameter übergeben, kann man methodenintern nicht nur auf das Objekt zugreifen (z.B. auf seine Instanzvariablen bei entsprechenden Zugriffsrechten), sondern auch den Inhalt der Referenzvariablen ändern, so dass sie z.B. anschließend auf ein anderes Objekt zeigt. Es ist nur selten sinnvoll, bei einer Methodendefinition Referenzparameter mit Referenztyp (vom Typ einer Klasse) zu verwenden. Erlaubt *ist* diese für Anfänger recht verwirrende Doppelreferenz jedoch. Ein möglicher Einsatzzweck wäre eine methodenintern zu verändernde und zum Aufrufer zurück zu transportierende Zeichenfolge. Wie sich in Abschnitt 5.4.1.1 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 zurück zu transportieren:

Quellcode

```
using System;
class Prog {
   void WertParDemo(ref String spar) {
      Console.WriteLine(spar);
      spar = "NEU";
   }
   static void Main() {
      string sM = "ALT";
      Prog p = new Prog();
      p.WertParDemo(ref sM);
      Console.WriteLine(sM);
   }
}
```

Ausgabe

```
ALT
NEU
```

4.3.1.3.3 Ausgabeparameter

Auch über Ausgabeparameter kann man einer Methode die Veränderung von Variablen der rufenden Programmeinheit ermöglichen.

Als Ausgabeaktualparameter sind nur *Variablen* erlaubt, die vom *selben* Typ wie der Formalparameter sein müssen. 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 Ausgabeparameter vor dem Verlassen der Methode ein Wert zugewiesen wird.

In der Methodendefinition *und* beim Methodenaufruf sind Ausgabeparameter durch das Schlüsselwort **out** zu kennzeichnen, z.B.:

Quellcode:

```
using System;
class Prog {
  void Lies(out int z, out int n) {
    Console.Write("x = ");
    z = Convert.ToInt32(Console.ReadLine());
    Console.Write("\ny = ");
    n = 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));
  }
}
```

Ausgabe:

```
x = 29

y = 5

x % y = 4
```

4.3.1.3.4 Parameterserien variabler Länge

Vielleicht haben Sie sich schon darüber gewundert, dass man beim Aufruf der Methode **Console.WriteLine()** hinter einer geeigneten Formatierungszeichenfolge beliebig viele Ausdrücke durch jeweils ein Komma getrennt als Aktualparameter übergeben darf, z.B.:

```
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 deklariert und durch das Schlüsselwort **params** gekennzeichnet werden muss. In obigen Syntaxdiagrammen wurde diese Option der Einfachheit halber weggelassen. Zwar haben wir uns bisher kaum mit Array-Datentypen beschäftigt, doch sollte das folgende Beispiel hinreichend klären, wie Array-Parameter deklariert und verwendet werden:

Quellcode:

```
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);
  }
}
```

Ausgabe:

```
Die Summe ist = 2,2
Die Summe ist = 5,8
```

4.3.1.4 Methodenrumpf

Über die Verbundanweisung, die den Rumpf einer Methode bildet, haben Sie bereits erfahren:

- Hier werden die Formalparameter wie lokale Variablen verwendet.
- Wert- und Referenzparameter werden von der aufrufenden Programmeinheit initialisiert, so dass diese den Ablauf der Methode beeinflussen kann.
- Über Referenz- und Ausgabeparameter können Variablen der aufrufenden Programmeinheit verändert werden.
- Die **return**-Anweisung dient zur Rückgabe von Werten an den Aufrufer und/oder zum Beenden einer Methodenausführung.

Ansonsten können beliebige Anweisungen unter Verwendung von elementaren und objektorientierten Sprachelementen eingesetzt werden, um den Zweck einer Methode zu realisieren. Definitionen (z.B. von Klassen oder Methoden) sind jedoch *nicht* erlaubt.[1]

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 4.3.3).

4.3.2 Methodenaufruf und Aktualparameter

Beim Aufruf einer Instanzmethode, z.B.:

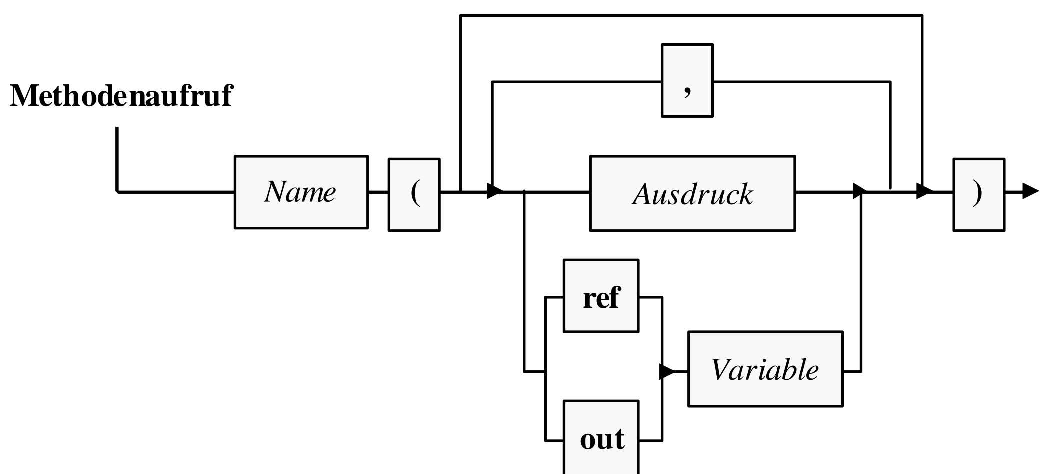
```
b1.Zeige();
```

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 um eine synchron zur Formalparameterliste geordnete Serie von Ausdrücken bzw. Variablen passenden Typs handeln muss.



Es ist grundsätzlich eine Parameterliste anzugeben, ggf. eine leere.

[1] Im Zusammenhang mit Delegaten und anonymen Methoden werden Sie später doch eine Möglichkeit zum Verschachteln von Methodendefinitionen kennen lernen (siehe Abschnitt 9.3.1.4).

Als Beispiel betrachten wir einen Aufruf der in Abschnitt 4.3.1.1 vorgestellten Variante der `Bruch`-Methode `Addiere()`:

```
b1.Addiere(1, 3, true);
```

Liefert eine Methode einen Wert zurück, stellt ihr Aufruf einen verwertbaren **Ausdruck** dar und kann als Argument in komplexeren Ausdrücken auftreten, 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 zu seinem eigenen Wert addieren und das Resultat (bei der Variante aus Abschnitt 4.3.1.3.1 paramtergesteuert) 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();
```

4.3.3 Debug-Einsichten zu (verschachtelten) Methodenaufrufen

Verschachtelte Methodenaufrufe stellen keine Besonderheit, sondern den selbstverständlichen Normalfall dar. Nichtsdestotrotz oder gerade deswegen ist es angemessen, das Geschehen etwas genauer zu betrachten. Anhand der folgenden Bruchrechnungsstartklasse

```
using System;
class BruchRechnung {
   static void Main() {
      Bruch b1 = new Bruch(), b2 = new Bruch();
      b1.Zaehler = 2; b1.Nenner = 8;
      b2.Zaehler = 2; b2.Nenner = 3;
      b1.Addiere(b2);
      b1.Zeige();
   }
}
```

soll mit Hilfe unserer Entwicklungsumgebung untersucht werden, was bei folgender Aufrufverschachtelung geschieht:

- Die statische Methode **Main()** der Klasse `BruchRechnung` ruft die `Bruch`-Instanzmethode `Adddiere()`.

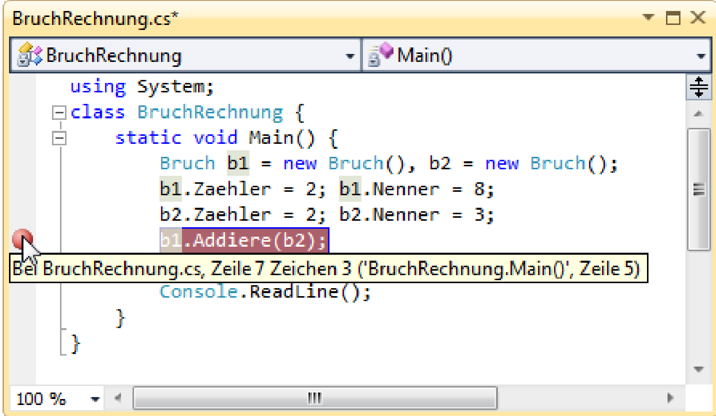
- Die `Bruch`-Instanzmethode `Adddiere()` ruft die `Bruch`-Instanzmethode `Kuerze()`.

```
using System;
class BruchRechnung {
  static void Main() {
    Bruch b1 = new Bruch(), b2 = new Bruch();
    b1.Zaehler = 2; b1.Nenner = 8;
    b2.Zaehler = 2; b2.Nenner = 3;
    b1.Addiere(b2);
    b1.Zeige();
    Console.ReadLine();
  }
}
```

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

```
public void Kuerze() {
  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;
}
```

Wir verwenden dabei die zur Fehlersuche konzipierten Debug-Techniken unserer Entwicklungsumgebung. Das Programm soll nicht komplett ablaufen, sondern an mehreren Stellen durch einen so genannten **Halte-** bzw. **Unterbrechungspunkt** (engl. *breakpoint*) gestoppt werden, so dass wir jeweils die Lage im Hauptspeicher inspizieren können. Um einen Haltepunkt zu setzen oder wieder zu entfernen, setzt man einen Mausklick in die Infospalte neben der betroffenen Anweisung, z.B.



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

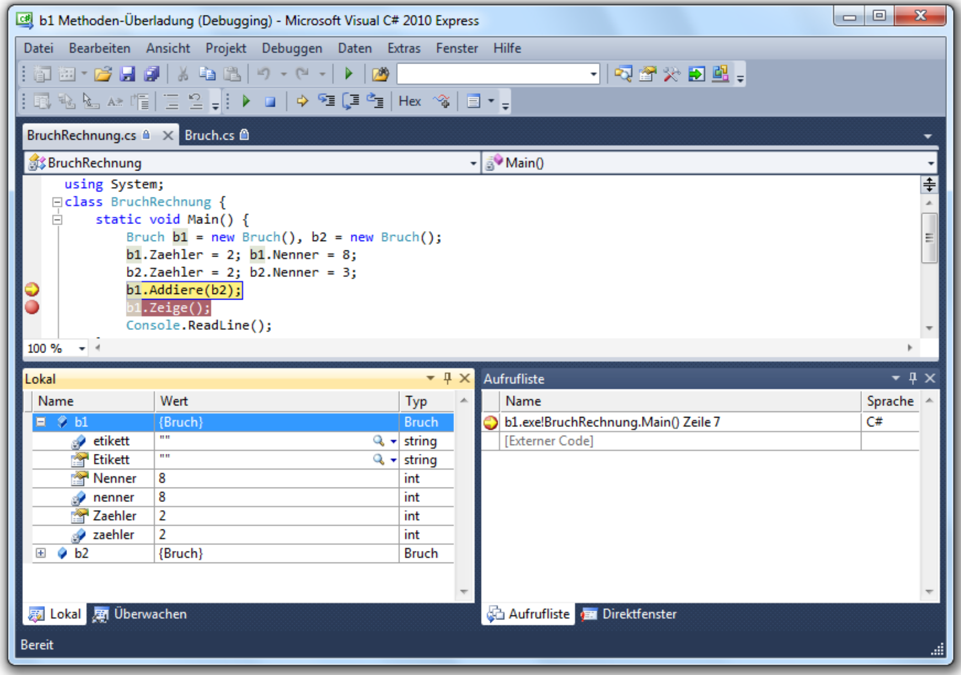
  ```
  ggt = divisor;
  ```

  im Block der **do-while** - Schleife.

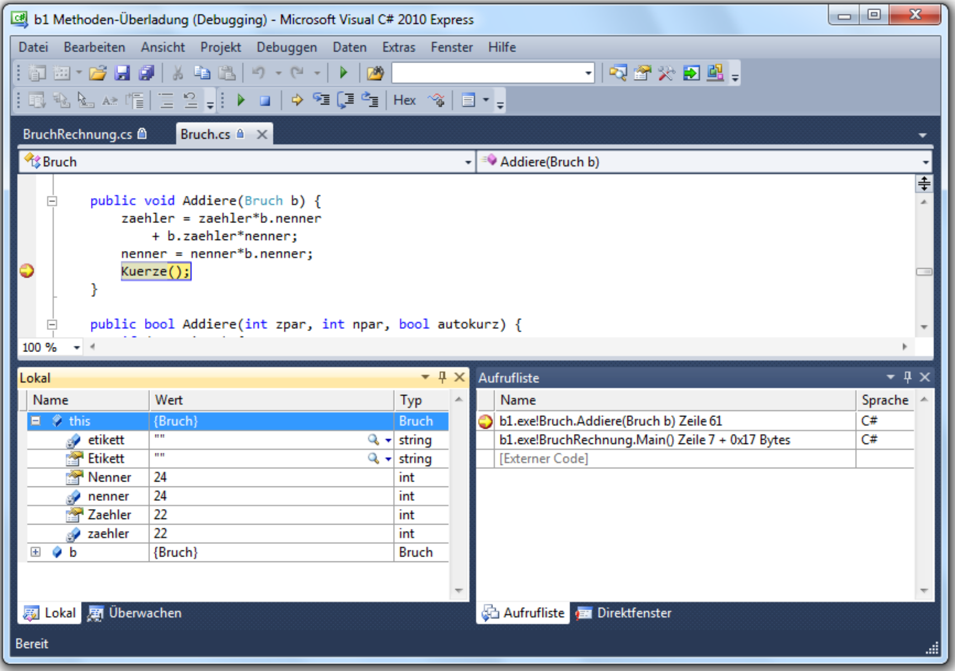
Starten Sie das Programm wie gewohnt über den Schalter ▶, die Funktionstaste **F5** oder den Menübefehl

Debuggen > Debugging starten

Das Programm wird gleich wieder von der Entwicklungsumgebung gestoppt, und im Quellcode-Editor ist der erreichte Haltepunkt markiert. Unten links zeigt das mit **Lokal** betitelte Fenster die beiden lokalen Referenzvariablen der **Main**()-Methode (`b1`, `b2`) und auf Wunsch (nach einem Mausklick auf einen Plus-Schalter neben einer Referenzvariablen) das Innenleben des referenzierten Objekts. Momentan interessiert besonderes das mit **Aufrufliste** betitelte Fenster unter rechts. Hier ist zu erkennen, dass aktuell nur die Einsprungmethode **Main**() in Bearbeitung befindlich ist (mit Daten im Stack-Bereich des Speichers):



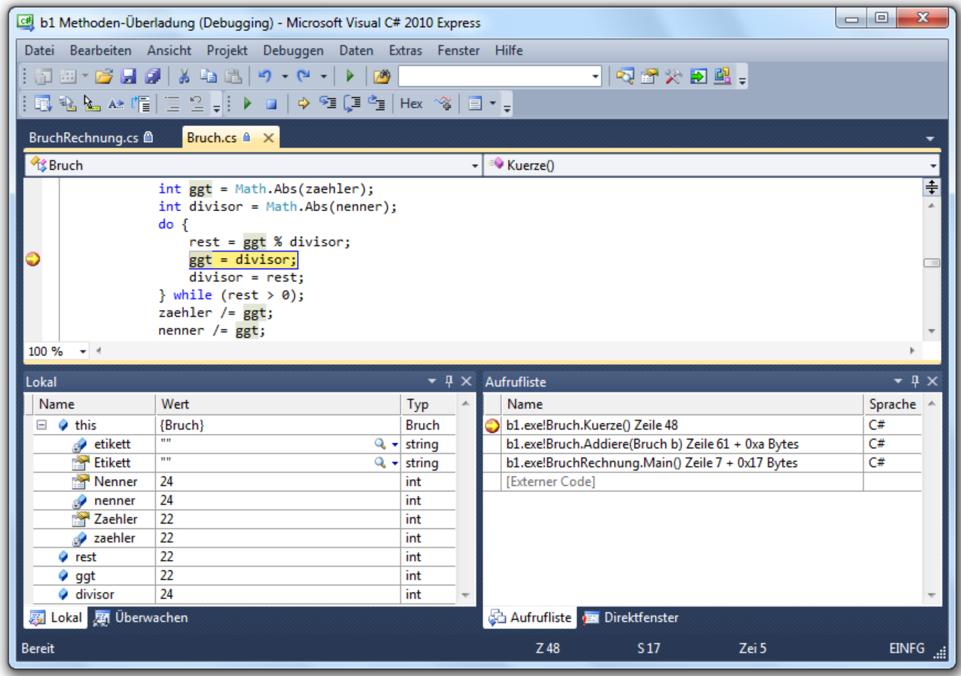
Lassen Sie das Programm mit dem Schalter ▶ oder der Funktionstaste **F5** fortsetzen. Beim Erreichen des zweiten Haltepunkts (Anweisung „`Kuerze();`“ in der Methode `Addiere()`) liegen auf dem Stack die Daten und Verwaltungsinformationen (die *Stack Frames*) der Methoden `Addiere()` und **Main**() übereinander:



Das Fenster unten links zeigt als lokale Variablen der Methode `Addiere()`:

- **this** (Referenz auf das handelnde Objekt)
  Erartungsgemäß ist der Bruch noch nicht gekürzt.
- Parameter `b`

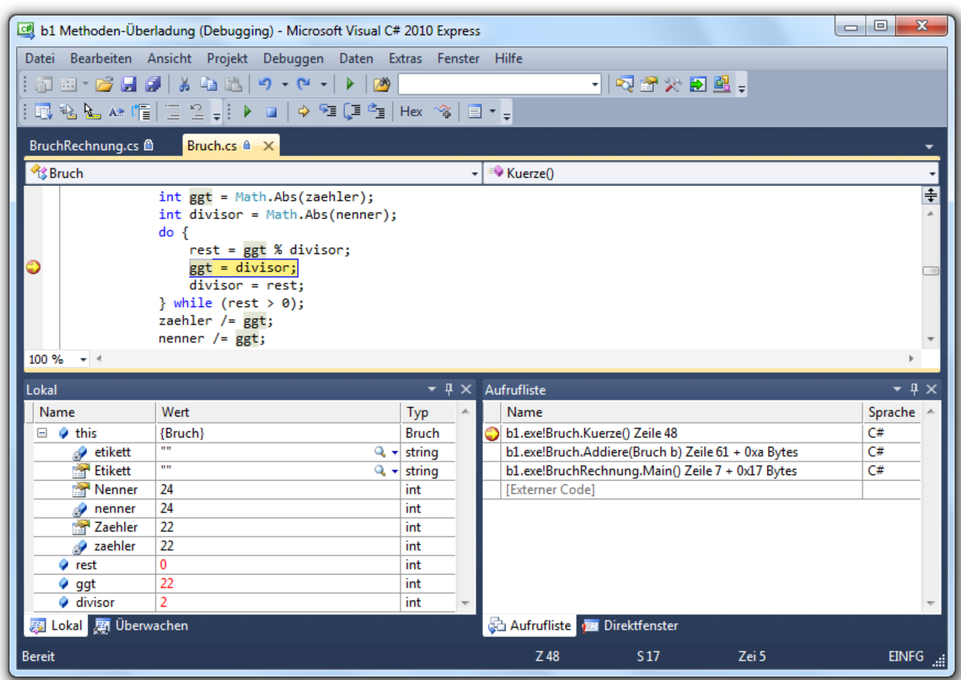
Beim Erreichen des dritten Haltepunkts (Anweisung „`ggt = divisor;`“ in der Methode `kuerze()`) liegen die Stack Frames der Methoden `Kuerze()`, `Addiere()` und **Main**() übereinander:



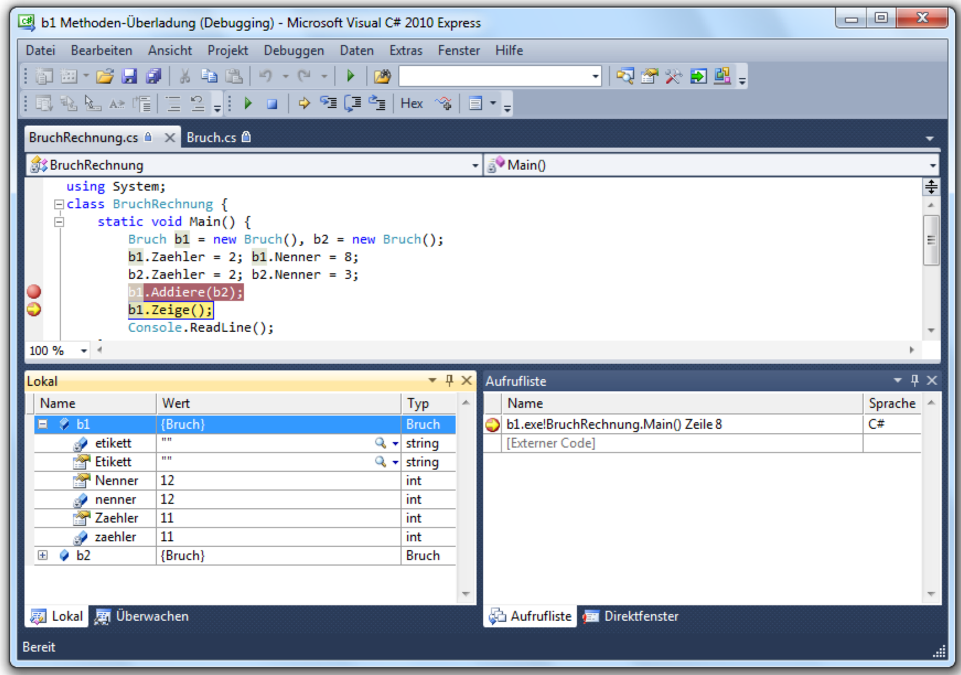
Das Fenster unten links zeigt als lokale Variablen der Methode `Kuerze()`:

- **this** (Referenz auf das handelnde 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.:

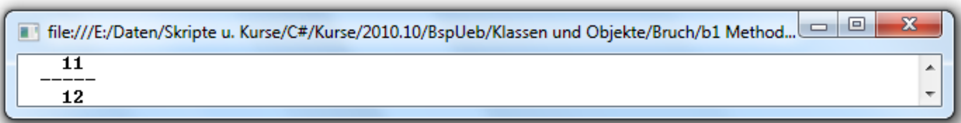


Bei Erreichen des letzten Haltepunkts (Anweisung „`b1.Zeige();`“ in **Main**()) ist nur noch der Stack Frame der Methode **Main**() vorhanden:



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 wird von den kommerziellen Visual Studio – Varianten über den Menübefehl

Debuggen > Alle Haltepunkte löschen

unterstützt, den die Visual C# 2010 Express Edition leider *nicht* anbietet.

Weil der verfügbare Speicher endlich ist, kann es bei der Aufrufverschachtelung und der damit verbundenen Stapelung von Stack Frames zu einem Laufzeitfehler vom Typ **StackOverflow-Exception** kommen. Dies wird aber nur bei einem schlecht entworfenen bzw. fehlerhaften Algorithmus passieren.

4.3.4 Methoden überladen

Die in Abschnitt 4.3.1.1 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 so genannte *Überladung* von Methoden vor.

Eine Überladung ist erlaubt, wenn sich die *Signaturen* der beteiligten Methoden unterscheiden. Zwei Methoden besitzen genau dann *dieselbe* Signatur, wenn die folgenden Bedingungen erfüllt sind:[1]

[1] Bei den später zu behandelnden *generischen* Methoden muss die Liste mit den Kriterien für die Identität von Signaturen erweitert werden.

- Die Namen sind identisch.
- Die Parameterlisten sind gleich lang
- Positionsgleiche Parameter stimmen hinsichtlich Datentyp und Parameterart (Wert-, Referenz- bzw. Ausgabeparameter) überein.

Für die Signatur ist der Rückgabetyp einer Methode ebenso irrelevant wie die Namen ihrer Formalparameter und das beim letzten Formalparameter erlaubte **params**-Schlüsselwort (vgl. ECMA 2006, S. 95). 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 entscheidbar sein, welche Methode aus einer Überladungsfamilie zu verwenden ist.

Ist bei einem Methodenaufruf die angeforderte Überladung nicht eindeutig zu bestimmen, meldet der Compiler einen Fehler.

Von einer Methode unterschiedlich parametrisierte Varianten in eine Klassendefinition aufzunehmen, lohnt sich z.B. in folgenden Situationen:

- 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 float 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. So tauscht z.B. die folgende Methode die Inhalte von zwei Referenzparametern mit beliebigem (natürlich identischem) Datentyp:
  ```
  static void Tausche<T>(ref T a, ref T b) {
    T temp = a;
    a = b;
    b = temp;
  }
  ```
- Für eine Methode sollen unterschiedliche umfangreiche Parameterlisten angeboten werden, sodass zwischen einer bequem aufrufbaren Standardausführung (z.B. mit leerer Parameterliste) und einer individuell gestalteten Ausführungsvariante gewählt werden kann.

4.4 Objekte

Ein zentrales Ziel der OOP ist die Produktion von Software mit hohem Recycling-Potential. Daher wurde im Bruchrechnungsprojekt die recht universell verwendbare `Bruch`-Klasse konzipiert, die schon in Programmen mit stark unterschiedlichen Benutzerschnittstellen (Konsole versus GUI) zum Einsatz kam. In Abschnitt 4.4 geht es darum, wie man Objekte solcher Klassen in die Welt setzen und nutzen kann.

4.4.1 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 erklären, beginnen wir mit einer einfacheren Variante *ohne* Initialisierung:

```
Bruch b;
```

Hier wird die **Referenzvariable** `b` mit dem Datentyp `Bruch` deklariert, die folgende Werte annehmen kann:

- die Adresse eines `Bruch`-Objekts
  In der Variablen wird also 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.
- **null**
  Dieses Referenzliteral steht für einen leeren Verweis. Eine Referenzvariable mit diesem Wert ist nicht undefiniert, sondern zeigt explizit auf nichts.

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 3.3.2):

- Elementare Typen (**bool**, **char**, **byte**, **double**, ...)
  Hier handelt es sich um Werttypen.
- Klassen (Referenztypen)
  Ist eine Variable vom Typ einer Klasse, kann sie die Adresse eines Objekts aus dieser Klasse aufnehmen. Zwar ist der Aufbau einer Objektadresse bei allen Klassen gleich, doch achtet der Compiler strikt darauf, dass eine Referenzvariable nur auf Objekte aus der deklarierten Klasse oder aus einer daraus abgeleiteten Klasse (siehe unten) zeigt.

Später kommen mit den *Strukturen* noch Werttypen dazu, deren Instanzen (wie die Objekte von Klassen) problemadäquat mit beliebig vielen Feldern ausgestattet werden können.

4.4.2 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. in folgendem Ausdruck:

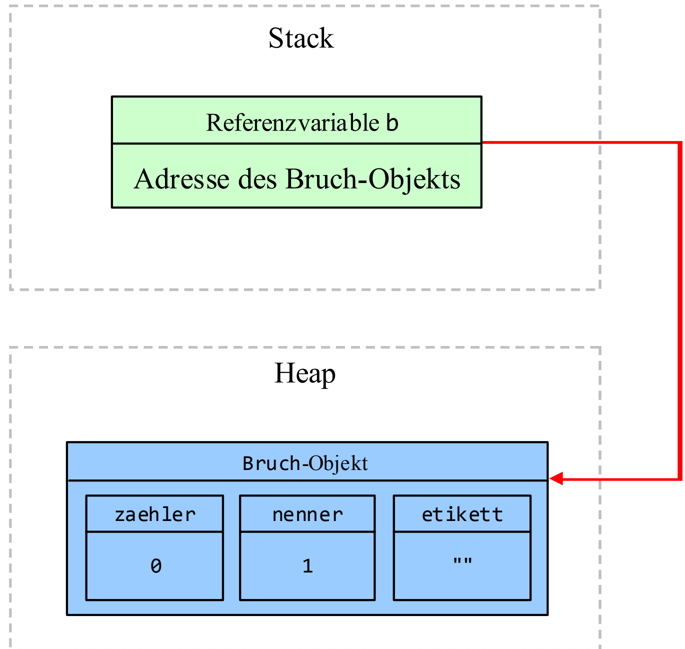
```
new Bruch()
```

Als Operanden erwartet **new** einen Klassennamen, dem eine Parameterliste zu folgen hat, weil er hier als Name eines *Konstruktors* (siehe Abschnitt 4.4.3) aufzufassen ist. Die involvierte Klasse legt den Typ des Ausdrucks fest. Als Wert resultiert eine Referenz, die einen Zugriff auf das neue Objekt (seine Methoden, Eigenschaften, etc.) erlaubt.

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 Speicher des Programms:



Während lokale Variablen bereits beim Aufruf einer Methode (also unabhängig vom konkreten Ablauf) im **Stack**-Bereich des programmeigenen Speichers angelegt werden, entstehen Objekte (mit ihren Instanzvariablen) erst bei der Auswertung des **new**-Operators. Sie erscheinen auch nicht auf dem Stack, sondern werden im **Heap**-Bereich des programmeigenen Hauptspeichers angelegt.

In einem Programm können durchaus *mehrere* Referenzvariablen auf *dasselbe* Objekt zeigen, z.B.:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
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();
  }
}
```

</td><td>

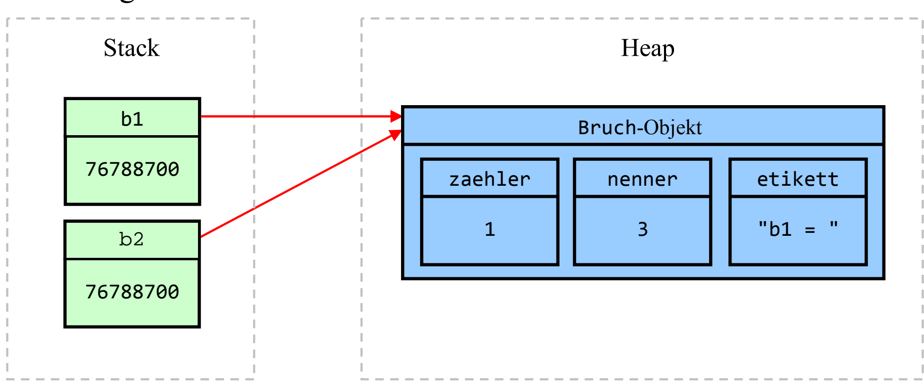
```
        1
b2 =  -----
        3
```

</td></tr>
</table>

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.

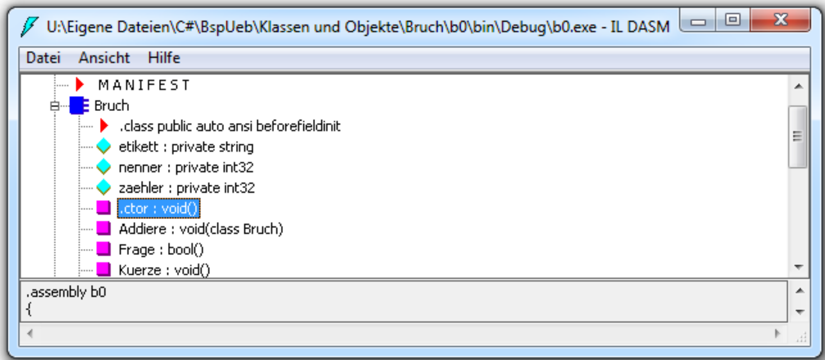
4.4.3 Objekte initialisieren über Konstruktoren

In diesem Abschnitt werden spezielle Methoden behandelt, die beim Erzeugen von neuen Objekten zum Einsatz kommen, um die Instanzvariablen der Objekte zu initialisieren und/oder andere Arbeiten zu verrichten (z.B. Öffnen einer Datei oder Netzwerkverbindung). 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 kommt ein **Standardkonstruktor** zum Einsatz. Weil dieser keine Parameter besitzt, ergibt sich sein Aufruf aus dem Klassennamen durch Anhängen einer leeren Parameterliste, z.B.:

```
Bruch b = new Bruch();
```

Wie der MSIL-Code zum Standardkonstruktor unserer `Bruch`-Klasse (mit dem Namen **.ctor**, Abkürzung für *Konstruktor*) zeigt,



fügt der Compiler automatisch Code für die im Quelltext der Klasse enthaltenen Initialisierungen von Instanzvariablen ein (betroffen: `nenner` und `etikett`):

```
Bruch::.ctor : void()
Suchen  Weitersuchen
.method public hidebysig specialname rtspecialname
        instance void  .ctor() cil managed
{
  // Code size       26 (0x1a)
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  ldc.i4.1
  IL_0002:  stfld      int32 Bruch::nenner
  IL_0007:  ldarg.0
  IL_0008:  ldstr      ""
  IL_000d:  stfld      string Bruch::etikett
  IL_0012:  ldarg.0
  IL_0013:  call       instance void [mscorlib]System.Object::.ctor()
  IL_0018:  nop
  IL_0019:  ret
} // end of method Bruch::.ctor
```

Für eine automatische Null-Initialisierung (vgl. Abschnitt 4.2.3) ist hingegen kein MSIL-Code er-

forderlich. Zur Anzeige des MSIL-Codes wird hier das im Windows-SDK enthaltene und mit der Visual C# 2010 Express Edition automatisch installierte Hilfsprogramm **ILDasm** verwendet.[1]

Am Ende (!) des MSIL-Codes zum Standardkonstruktor wird der parameterlose Konstruktor der Basisklasse aufgerufen, wobei unsere Klasse `Bruch` direkt von der Urahnklasse **Object** im Namensraum **System** abstammt.

Abgesehen von den momentan für uns noch irrelevanten abstrakten Klassen hat der Standardkonstruktor einer Klasse die Schutzstufe **public**, ist also allgemein verfügbar.

In der Regel ist es beim Klassendesign sinnvoll, mindestens einen Konstruktor *explizit* zu definieren, um das individuelle Initialisieren der Instanzvariablen von neuen Objekten zu ermöglichen. Dabei sind folgende Regeln zu beachten:

- Ein Konstruktor trägt denselben Namen wie die Klasse.
- Ein Konstruktor liefert grundsätzlich *keinen* Rückgabewert, und es wird bei der Definition *kein* Typ angegeben, auch nicht der Ersatztyp **void**, mit dem wir bei gewöhnlichen Methoden den Verzicht auf einen Rückgabewert dokumentieren müssen.
- Es darf eine Parameterliste definiert werden, was zum Zweck der Initialisierung ja auch unumgänglich ist.
- Sobald man einen eigenen Konstruktor definiert, steht der Standardkonstruktor *nicht* mehr zur Verfügung.
- Ist weiterhin ein paramameterfreier Konstruktor erwünscht, so muss dieser *zusätzlich* definiert werden.
- Der Compiler fügt bei *jedem* Konstruktor automatisch MSIL-Code für die im Quellcode der Klassendefinition enthaltenen Feldinitialisierungen 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 4.3.4) ist also auch bei Konstruktoren erlaubt.
- Während der Standardkonstruktor die Schutzstufe **public** besitzt, haben explizite Konstruktoren haben wie gewöhnliche Methoden die voreingestellte Schutzstufe **private**. Wenn sie für beliebige fremde Klassen zur Objektkreation verfügbar sein sollen, ist also in der Definition der Modifikator **public** anzugeben.
- Konstruktoren können 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;
}
```

Beachten Sie bitte: 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 beliebige andere Methode einer Klasse muss natürlich auch

[1] Bei der Installation der Visual C# 2010 Express Edition unter Windows 7 (64 Bit) landet das Hilfsprogramm im Ordner
%ProgramFiles(x86)%\Microsoft SDKs\Windows\v7.0A\bin

ein Konstruktor so entworfen sein, dass die Objekte der Klasse unter allen Umständen konsistent und funktionstüchtig sind.

Wenn weiterhin auch ein parameterfreier Konstruktor verfügbar sein soll, muss dieser explizit definiert werden, z.B. mit leerem Anweisungsteil:

```
public Bruch() {}
```

Im folgenden Programm werden beide Konstruktoren eingesetzt:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>using System;
class BruchRechnung {
   static void Main() {
      Bruch b1 = new Bruch(1, 2, "b1 = ");
      Bruch b2 = new Bruch();
      b1.Zeige();
      b2.Zeige();
   }
}</pre></td><td><pre>   1
b1 =   -----
         2

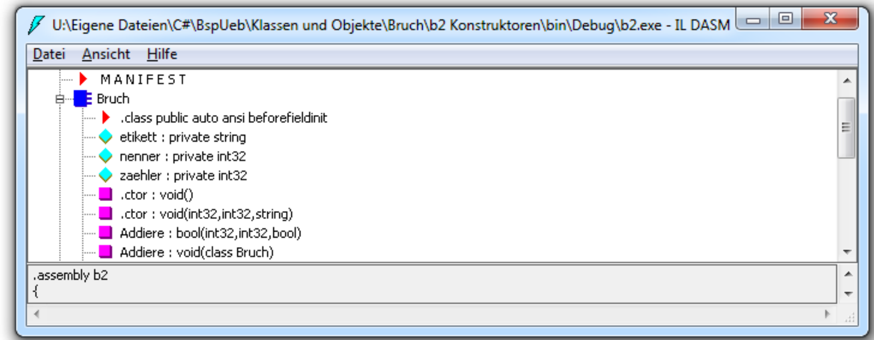
    0

    1</pre></td></tr>
</table>

Von der Regel, dass Konstruktoren nur über den **new**-Operator genutzt werden können, gibt es eine Ausnahme: Zwischen Parameterliste und Anweisungsblock eines Konstruktors darf ein anderer Konstruktor derselben Klasse über das Schlüsselwort **this** aufgerufen werden, z.B.:

```
public Bruch() : this(0, 1, "unbekannt") {}
```

Wie das Windows-SDK - Hilfsprogramm **ILDasm** für die aktuelle Ausbaustufe des Bruchrechnungs-Assemblies zeigt, erscheinen die Konstruktoren einer Klasse unter dem Namen **.ctor** wie gewöhnliche Methoden in der Typ-Metadatentabelle:



Öffentliche Felder, die wegen fehlender Datenkapselung nicht in jeder Situation akzeptabel sind, und öffentliche Eigenschaften (siehe Abschnitt 4.5) können seit C# 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 angegeben, z.B.:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>using System;
class CT {
   public int i;
}
class Prog {
   static void Main() {
      CT ct = new CT() {i = 13};
      Console.WriteLine(ct.i);
   }
}</pre></td><td><pre>13</pre></td></tr>
</table>

Die neue Option wurde unter der Bezeichnung **Objektinitialisierer** zur Unterstützung der LINQ-Technik (*Language Integrated Query*) eingeführt (siehe Abschnitt 21.1.2). Bei Verwendung eines Objektinitialisierers darf eine leere Parameterliste weggelassen werden, z.B.:

```
CT ct = new CT {i = 13};
```

Durch die neuen Objektinitialisierer werden aber Konstruktoren keinesfalls überflüssig, denn ...

- Konstruktoren können neben Wertzuweisungen beliebige andere Initialisierungsarbeiten ausführen (z.B. Dateien öffnen)
- Wie unsere `Bruch`-Klasse mit der Eigenschaft `Nenner` zeigt, ist bei Wertzuweisungen oft eine Kontroll- und Entscheidungslogik erforderlich.

4.4.4 Abräumen überflüssiger Objekte durch den Garbage Collector

Wenn keine Referenz mehr auf ein Objekt zeigt, wird es vom **Garbage Collector** (Müllsammler) der CLR automatisch entsorgt, und der belegte Speicher wird frei gegeben.

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.

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 frei gegeben, sobald die Ausführung der Methode beendet ist. Demgegenüber muss der von Objekten belegte Speicher bei älteren Programmiersprachen (z.B. C++) nach Gebrauch explizit wieder frei gegeben werden. In Anbracht der Objektmassen, die ein typisches Programm (z.B. ein Grafikeditor) benötigt, ist einiger Aufwand erforderlich, um eine Verschwendung von Speicherplatz zu verhindern. Mit seinem vollautomatischen Garbage Collector vermeidet C# lästigen Aufwand und zwei kritische Fehlerquellen:

- Weil der Programmierer keine Verpflichtung (und Berechtigung) zum Entsorgen von Objekten hat, kann es nicht zu Programmabstürzen durch Zugriff auf voreilig vernichtete Objekte kommen.
- Es entstehen keine **Speicherlöcher** (*memory leaks*) durch die vergessene Freigabe des Speichers zu überflüssig gewordenen Objekten.

Sollen die Objekte einer Klasse vor dem Entsorgen noch spezielle Aufräumaktionen durchführen, ist als Gegenstück zu den Konstruktoren ein so genannter **Destruktor** zu definieren, der ggf. vom Garbage Collector aufgerufen werden. Dabei sind folgende Regeln zu beachten:

- Ein Destruktor trägt denselben Namen wie die Klasse, wobei das Tilde-Zeichen (~) voranzustellen ist.
- Ein Destruktor liefert grundsätzlich *keinen* Rückgabewert, und es wird bei der Definition *kein* Typ angegeben.
- Pro Klasse ist nur *ein* Destruktor erlaubt. Dieser muss eine leere Parameterliste haben.
- Destruktoren können nicht aufgerufen werden. Es ist ausschließlich der automatische Aufruf durch den Garbage Collector vorgesehen.
- Bei der Definition eines Destruktors sind Modifikatoren überflüssig und verboten.

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 noch keinen Umgang mit solchen Ressourcen hatten, beschränken wir uns auf ein inhaltsfreies Beispiel, das aber

immerhin die Tätigkeit des Garbage Collectors zum Programmende dokumentiert. In der **Main()**-Methode der Startklasse `KD` wird ein Objekt der Klasse `K2` erzeugt, die von der Klasse `K1` abstammt:

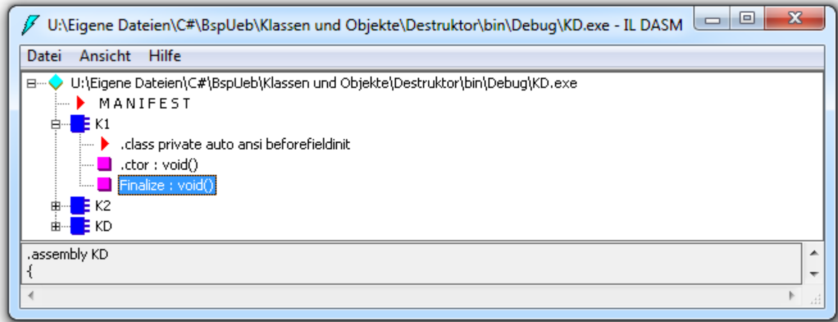
```
using System;

class K1 {
   ~K1() {
      Console.WriteLine("K1-Destruktor");
   }
}

class K2 : K1 {
   ~K2() {
      Console.WriteLine("K2-Destruktor");
   }
}

class KD {
   static void Main() {
      K2 k2 = new K2();
   }
}
```

Wie die Ausgabe zeigt, werden automatisch alle Destruktoren entlang des Stammbaums aufgerufen:

```
C:\Windows\system32\cmd.exe
U:\Eigene Dateien\C#\BspUeb\Klassen und Objekte\Destruktor\bin\Debug>kd
K2-Destruktor
K1-Destruktor
```

Vom Destruktor der Urahnklasse **Object** ist nichts zu sehen, weil er keine Ausgabe macht (und wohl auch sonst nicht viel tut).

Bei einer Assembly-Inspektion mit dem Windows-SDK - Hilfsprogramm **ILDasm** stellt man verwundert fest, dass der Destruktor eigentlich den Namen **Finalize()** trägt:



In C# ist aber die Destruktoren-Syntax (mit Tilde und Klassenname) zu verwenden.

4.4.5 Objektreferenzen verwenden

4.4.5.1 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 werden. 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, so dass die aufgerufene Methode über ihre lokale Referenzvariable auf das Originalobjekt zugreift und dort ggf. Veränderungen vornimmt.[1]

Von den beiden `Addiere()` – Methoden 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.

In obiger `Addiere()` – Methode 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 durchaus 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 Referenzparameter bestimmt wird:

```
public void DuplWerte(Bruch bc) {
   bc.zaehler = zaehler;
   bc.nenner = nenner;
}
```

In folgendem Programm wird das `Bruch`-Objekt `b1` beauftragt, die `DuplWerte()`-Methode auszuführen, wobei als Parameter eine Referenz auf das Objekt `b2` übergeben wird:

Quellcode	Ausgabe

<pre>using System;
class BruchRechnung {
   static void Main() {
      Bruch b1 = new Bruch(1, 2, "b1 = ");
      Bruch b2 = new Bruch(5, 6, "b2 = ");
 	
      b1.Zeige();
      b2.Zeige();
      b1.DuplWerte(b2);
      Console.WriteLine("Nach DuplWerte():\n");
      b2.Zeige();
   }
}</pre>
	<pre> 1
	 b1 = -----
	        2
	
	        5
	 b2 = -----
	        6
	
	Nach DuplWerte():
	
	        1
	 b2 = -----
	        2</pre>

[1] Wertparameter mit Referenztyp arbeiten also analog zu den Referenzparametern (vgl. Abschnitt 4.3.1.3.2), 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 Werttypen zu ermöglichen. Wir werden mit den so genannten Strukturen noch Werttypen kennen lernen, die ähnlich umfangreich sein können wie Klassentypen.

4.4.5.2 *Rückgabewerte mit Referenztyp*

Bisher haben die innerhalb einer Methode erzeugten Objekte das Ende der Methode nicht überlebt, waren jedenfalls anschließend nicht mehr nutzbar. Weil keine Referenz außerhalb der Methode existierte, wurden die Objekte dem Garbage Collector überlassen. Soll ein methodenintern erzeugtes Objekt nach Ende der Methodenausführung weiterhin zur Verfügung stehen, 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 `Bruch`-Klasse 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:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
using System;
class BruchRechnung {
   static void Main() {
      Bruch b1 = new Bruch(1, 2, "b1 = ");
      b1.Zeige();
      Bruch b2 = b1.Klone();
      b2.Zeige();
   }
}
```

</td><td>

```
        1
b1 =  -----
        2

        1
b1 =  -----
        2
```

</td></tr>
</table>

4.4.5.3 *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. Dies ist mit dem Schlüsselwort **this** möglich, das innerhalb einer Instanzmethode wie eine Referenzvariable funktioniert. In folgendem 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 werden soll. Später werden Sie noch weit relevantere **this**-Verwendungsmöglichkeiten kennen lernen.

---

[1] Bei einer für die breitere Öffentlichkeit gedachten Klasse sollte auch eine die Schnittstelle **ICloneable** (siehe Kapitel 8) implementierende Vervielfältigungsmethode angeboten werden, obwohl diese Schnittstelle durch semantische Unklarheit von begrenztem Wert ist, was im Kapitel 8 über Schnittstellen (Interfaces) noch näher erläutert wird.

4.5 Eigenschaften

4.5.1 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;
}
```

Entsprechend 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 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 ist ein **get**- und ein **set**-Block mit nahe liegender Syntax zu implementieren. 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 weit intuitiver zu verwenden ist als korrespondierende Zugriffsmethoden, z.B.:

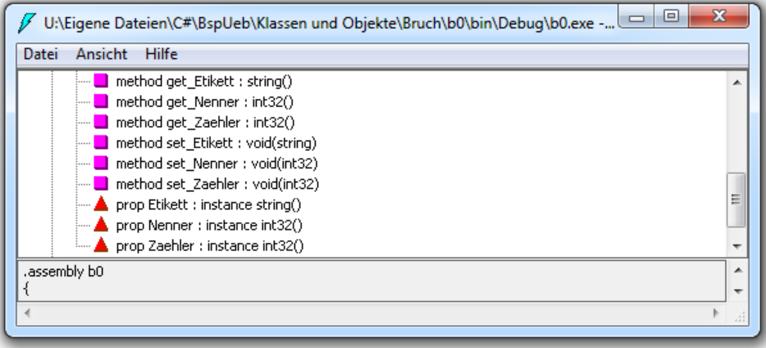
```
b1.Nenner = 2;
Console.WriteLine(b1.Nenner);
```

Sogar die Aktualisierungs-Operatoren (vgl. Abschnitt 3.5.8) werden unterstützt, z.B.:

```
b1.Nenner += 2;
```

Zwar handelt es sich bei den Eigenschaften eher um *syntactic sugar* (Mössenböck 2003, S. 3) als um einen essentiellen Vorteil gegenüber anderen Programmiersprachen wie Java und C++, doch ist diese gelungene Kombination aus objektorientierter Datenkapselung und vertrauter Merkmals-Syntax durchaus zu begrüßen.

Wie eine Assembly-Inspektion mit dem Windows-SDK - Hilfsprogramm **ILDasm** zeigt, erstellt der Compiler zu den Eigenschaften unserer Klasse `Bruch` jeweils ein Paar von Zugriffsmethoden:



4.5.2 Automatisch implementierte Eigenschaften

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** für die gewünschten Zugriffsarten 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` Fall können also die Deklaration

```
int zaehler;
```

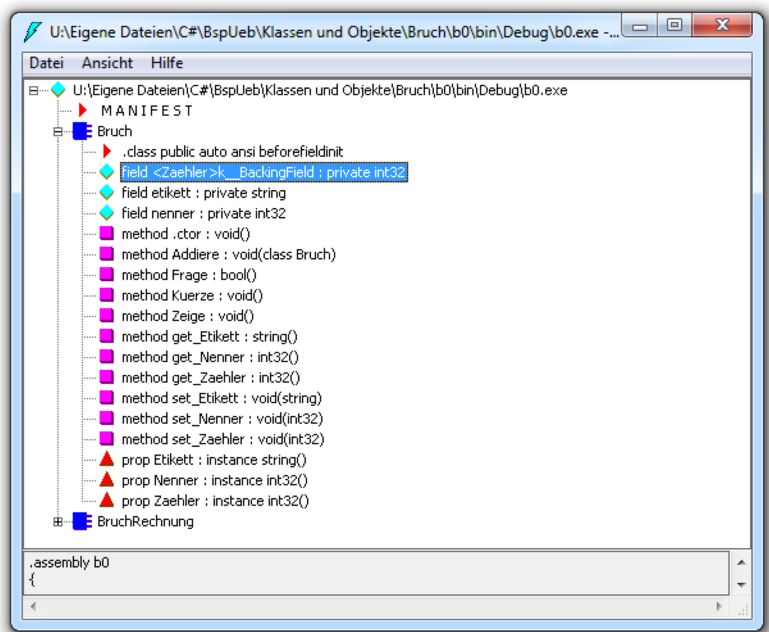
und die Definition

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

äquivalent ersetzt werden durch die Definition:

```
public int Zaehler {get; set;}
```

Wie das Windows-SDK - Hilfsprogramm **ILDasm** zeigt, ergänzt der Compiler automatisch ein privates **Backing Field** mit passendem Typ:



Dieses Feld ist ausschließlich über die Eigenschaft ansprechbar. Bei einer nachträglichen Umstellung des bereits vorhandenen `Bruch`-Quellcodes müssten folglich alle Zugriffe auf die Instanzvariable `zaehler` ersetzt werden.

Im Einstiegsbeispiel ist es allerdings der Transparenz halber sinnvoll, auf automatisch implementierte Eigenschaften zu verzichten.

4.5.3 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] 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**. Bei Tests mit dem .NET-Framework 4.0 (im Dezember 2010) zeigten sich allerdings teilweise Plattform-abhängige Optimierungseffekte:

a) Eigenschafts- bzw. Feldzugriff auf den **Zähler** eines `Bruch`-Objekts:[2]

Zielplattform	Zeitaufwand für 100 Millionen Wertzuweisungen in Millisekunden	
	direkter Feldzugriff	via Eigenschaft
x86	98,6328	97,6563
x64	97,6562	99,6094

b) Eigenschafts- bzw. Feldzugriff auf den **Nenner** eines `Bruch`-Objekts:

Zielplattform	Zeitaufwand für 100 Millionen Wertzuweisungen in Millisekunden	
	direkter Feldzugriff	via Eigenschaft
x86	94,7266	134,7656
x64	98,6328	**292,9688**

Bei der **set**-Methode zum Nenner, die aufgrund der Null-Überwachung länger ausfällt als das Gegenstück zum Zähler, unterlässt die 64-Bit-Version des JIT-Compilers offenbar das Optimieren per Inlining.

Um eine **Read**- bzw. **Write-Only** Eigenschaft zu realisieren, verzichtet man einfach auf die **set**- bzw. die **get**-Implementation.

Aus der `Bruch`-Klassendefinition ist noch die Eigenschaft `Etikett` von Interesse, die den Referenzdatentyp **String** besitzt:

```
public string Etikett {
   get {
      return etikett;
   }
   set {
      if (value.Length <= 40)
         etikett = value;
      else
         etikett = value.Substring(0, 40);
   }
}
```

1 Siehe z.B. http://msdn.microsoft.com/en-us/library/65zdfbdt.aspx
2 Die Zeiten wurden unter Windows 7 gemessen mit der Intel-CPU Core i3 550.

Im **set**-Block wird über das Schlüsselwort **value** das vom Aufrufer als neuer Wert übergebene **String**-Objekt angesprochen. Es informiert in seiner Eigenschaft **Length** (definiert in der Klasse **String**) über die Anzahl der enthaltenen Zeichen. Bei Überlänge wird das **value**-Objekt mit der **Substring()**-Instanzmethode der Klasse **String** aufgeordert, ein neues, auf die ersten 40 Zeichen gekürztes **String**-Objekt zu erzeugen, dessen Adresse schlussendlich den neuen `etikett`-Wert bildet.

4.6 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 der Deklaration bzw. Definition kaum Unterschiede zwischen einem Instanz-Member und dem analogen statischen Member.

Abgesehen vom Standardkonstruktor (siehe Abschnitt 4.4.3) gilt auch bei den statischen Membern für den Zugriffsschutz:

- Voreingestellt ist die Schutzstufe **private**, so dass eine Verwendung nur klasseneigenen Methoden erlaubt ist.
- Durch Modifikatoren kann eine alternative Schutzstufe festgelegt werden (z.B. **public**).

4.6.1 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 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. Im Beispiel wird die (automatisch auf Null initialisierte) Klassenvariable `anzahl` in den beiden Instanzkonstruktoren inkrementiert.

Sofern Methoden *fremder* Klassen (durch den Modifikator **public**) der direkte Zugriff auf eine Klassenvariable gewährt wird, müssen diese dem Variablennamen ein Präfix 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, so dass der direkte Zugriff klasseneigenen Methoden vorbehalten bleibt.

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 4.2.3). Alternative Initialisierungen können in der Variablendeklaration oder im statischen Konstruktor (siehe Abschnitt 4.6.4) 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.

Damit im erweiterten Bruchrechnungsbeispiel fremde Klassen trotz Datenkapselung die Anzahl der bisher erzeugten `Bruch`-Objekte in Erfahrung bringen können, wird noch eine **statische Eigenschaft** mit **public**-Zugriff ergänzt:

```
public static int Anzahl {
   get {
      return anzahl;
   }
}
```

Weil nur die **get**-Funktionalität implementiert ist, können fremde Klassen den `anzahl`-Wert zwar ermitteln, aber nicht verändern.

4.6.2 Wiederholung zur Kategorisierung von Variablen

Mittlerweile haben wir verschiedene Variablensorten kennen gelernt, 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:

- **Lokale Variablen ...**
  werden in Methoden deklariert,
  landen auf dem Stack,
  werden **nicht** automatisch initialisiert,
  sind nur in den Anweisungen des innersten Blocks verwendbar,
  existieren, bis der innerste Block endet.

- **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 (siehe oben).

4.6.3 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 gerade 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 Beispielprogramm 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 (vgl. Abschnitt 1.2.7).

Trotz Ihrer Erfahrung mit diversen **Main()**-Methoden soll auch im Kontext unserer `Bruch`-Klasse das Definieren einer statischen Methode geübt werden. Zur Vereinfachung von Anweisungsfolgen nach dem folgenden Muster

```
Bruch b = new Bruch(0, 1, "Benutzerdefiniert: ");
b.Frage();
b.Kuerze();
```

definieren wir eine Klassenmethode, die eine Referenz auf ein neues `Bruch`-Objekt mit benutzerdefinierten und gekürzten Werten liefert:

```
public static Bruch BenDef(string e) {
  Bruch 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:

<table>
<tr><th>Quellcode</th><th>Eingaben (fett) und Ausgabe</th></tr>
<tr><td><pre>using System;
class BruchRechnung {
  static void Main() {
    Bruch b = Bruch.BenDef("Benutzerdefiniert: ");
    if (b != null)
      b.Zeige();
    else
      Console.WriteLine("b zeigt auf null");
  }
}</pre></td><td><pre>Zaehler: 26
Nenner : 39
                          2
 Benutzerdefiniert: -----
                          3</pre></td></tr>
</table>

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.

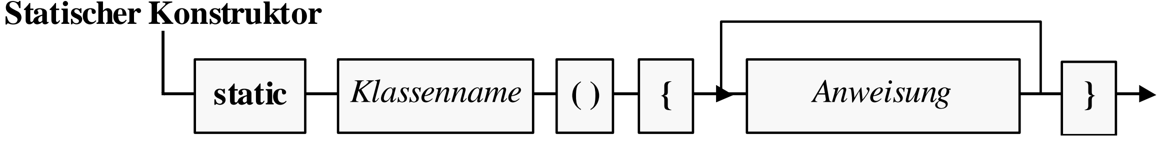
Weil der Modifikator **static** nicht Signatur-relevant ist (vgl. Abschnitt 4.3.4) kann es in einer Klasse zu einer Instanzmethode keine statische Methode mit demselben Namen und derselben Parameterliste geben.

In früheren Abschnitten waren mit *Methoden* stets *objektbezogene* Methoden (*Instanzmethoden*) gemeint. Dies soll auch weiterhin so gelten.

4.6.4 Statische Konstruktoren

Analog zu den Instanzkonstruktoren (siehe Abschnitt 4.4.3), die beim Erzeugen eines Objekts ausgeführt werden und sich um die Initialisierung von Instanzvariablen kümmern, kann für jede Klasse ein statischer Konstruktor zur Initialisierung von Klassenvariablen definiert werden. Er wird beim Laden der Klasse automatisch von der CLR ausgeführt und kann nirgends explizit aufgerufen werden. 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:

Statischer Konstruktor



Im .NET – Framework ist keine Reihenfolge für die Ausführung der bei einem Programm beteiligten statischen Konstruktoren definiert.

In einer etwas gekünstelten Erweiterung des `Bruch`-Beispiels soll der parameterfreie Instanzkonstruktor zufallsabhängige, aber pro Programmlauf 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 4.2.2). Im statischen Konstruktor wird ein Objekt der Klasse **Random** aus dem Namensraum **System** erzeugt und dann per **Next**()-Methodenaufruf mit der Produktion von **int**-Zufallswerten beauftragt:

```
static Bruch() {
   Random 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.:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>using System;
class BruchRechnung {
  static void Main() {
    Bruch b1 = new Bruch(), b2 = new Bruch();
    b1.Zeige(); b2.Zeige();
  }
}</pre></td><td><pre>Klasse Bruch geladen
    1

    2

    1

    2</pre></td></tr>
</table>

4.6.5 Statische Klassen

Besitzt eine Klasse *ausschließlich* statische Methoden, ist das Erzeugen von Objekten nicht sinnvoll. Man kann es mit dem Modifikator **static** in der Klassendefinition verhindern, z.B.

```
public static class Service {
   . . .
}
```

Auch die FCL enthält etliche Klassen, die ausschließlich statische Methoden enthalten und damit nicht zum Erzeugen von Objekten konzipiert sind. Mit der Klasse **Math** aus dem Namensraum **System** haben wir ein wichtiges Beispiel bereits kennen gelernt.

4.7 Vertiefungen zum Thema Methoden

4.7.1 Rekursive Methoden

Innerhalb einer Methode darf man selbstverständlich nach Belieben *andere* Methoden aufrufen. Es ist aber auch zulässig und in vielen Situationen sinnvoll, dass eine Methode *sich selbst* aufruft. Solche *rekursiven* Aufrufe erlauben eine elegante Lösung für ein Problem, das sich sukzessiv auf stets einfachere Probleme desselben Typs reduzieren lässt, bis man schließlich zu einem direkt lösbaren Problem gelangt. Zu einem rekursiven Algorithmus (per Selbstaufruf einer Methode) existiert stets ein äquivalenter *iterativer* Algorithmus (per Wiederholungsanweisung).

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* Realisierungen des Euklidischen Lösungsverfahrens kennen gelernt: In Abschnitt 1.1 wurde ein sehr einfacher Algorithmus benutzt, den Sie später in einer Übungsaufgabe (siehe Abschnitt 3.7.4) durch einen effizienteren Algorithmus (unter Verwendung der Modulo-Operation) 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 ausgelagert wird. Hier ist die iterative Methode `GGTi()` zu sehen:

```
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:

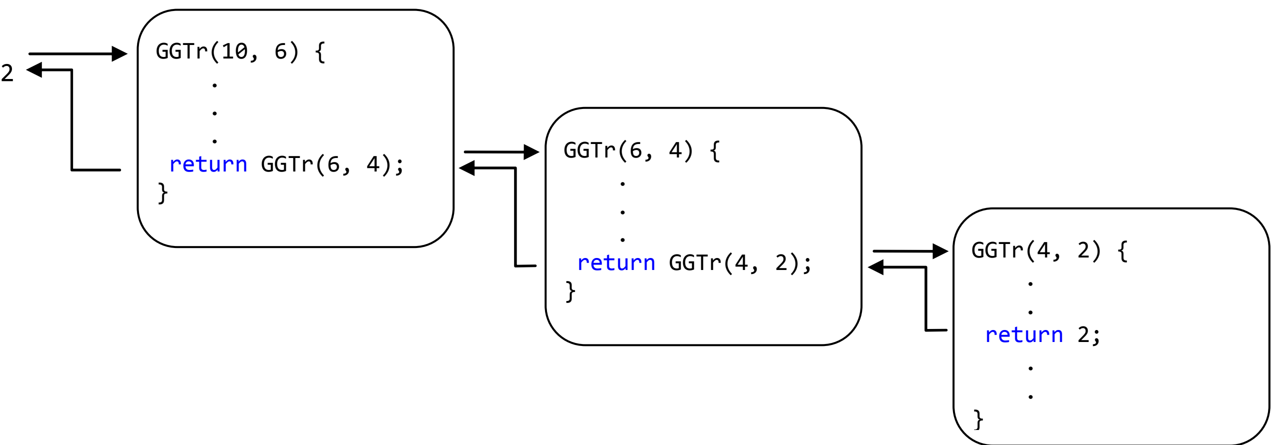
  ```
  return b;
  ```

- 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 recht elegant im Ausdruck der **return**-Anweisung:

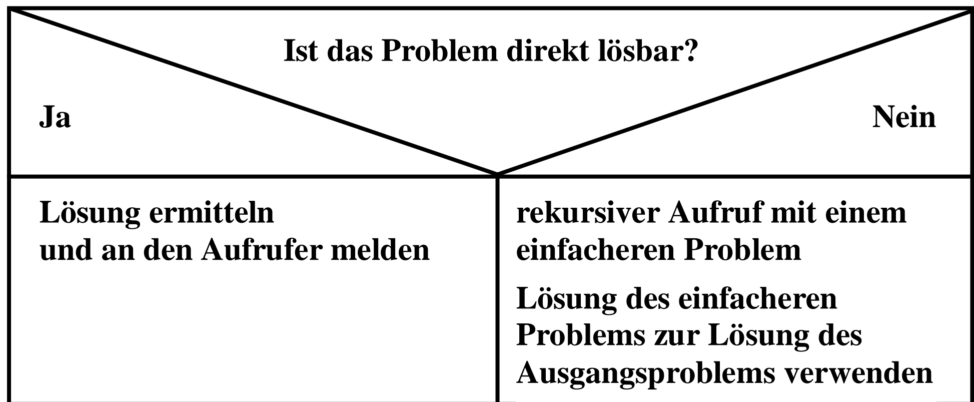
  ```
  return GGTr(b, rest);
  ```

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 in Abschnitt 3.7.4.

Wird die Methode `GGTr()` z.B. mit den Argumenten 10 und 6 aufgerufen, kommt es zu folgender Aufrufverschachtelung:



Generell läuft ein rekursiver Algorithmus nach der im folgenden **Struktogramm** beschriebenen Logik ab:



Im Beispiel ist die Lösung des einfacheren Problems. sogar 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.
```

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.

4.7.2 Operatoren überladen

Nicht nur Methoden können in C# überladen werden, sondern auch die *Operatoren* (+, * etc.), was z.B. in der Klasse **String** mit dem „+“ - Operator geschehen ist, so dass wir Zeichenfolgen bequem 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 = b1.Klone();
b3.Addiere(b2);
```

Es wäre eleganter, wenn derselbe Zweck mit folgender 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, "");
   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 (2003, S. 73ff).

Mit dem überladenen „+“ - Operator lassen sich `Bruch`-Additionen nun sehr übersichtlich formulieren:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
using System;
class BruchRechnung {
  static void Main() {
    Bruch b1 = new Bruch(1, 2, "b1 = ");
    b1.Zeige();
    Bruch b2 = new Bruch(1, 4, "b2 = ");
    b2.Zeige();
    Bruch b3 = b1 + b2;
    b3.Etikett = "Summe = ";
    b3.Zeige();
  }
}
```

</td><td>

```
          1
b1 =    -----
          2

          1
b2 =    -----
          4

             3
Summe =    -----
             4
```

</td></tr>
</table>

Wie das Windows-SDK - Hilfsprogramm **ILDasm** zeigt, resultiert aus unserer Operatorenüberladung im Assembly die statische Methode **op_Addition**():

U:\Eigene Dateien\C#\BspUeb\Klassen und Objekte\Bruch\b7 Operatoren-Überladung\bin\Debug\b7.exe - IL DASM
Datei Ansicht Hilfe
method get_Etikett : string()
method get_Nenner : int32()
method get_Zaehler : int32()
method op_Addition : class Bruch(class Bruch,class Bruch)
method set_Etikett : void(string)
method set_Nenner : void(int32)
method set_Zaehler : void(int32)
.assembly b7
{

4.8 Aggregieren und innere Klassen

4.8.1 Aggregation

Bei den Feldern einer Klasse sind beliebige Datentypen zugelassen, auch Klassentypen. Damit ist es z.B. möglich, vorhandene Klassen als Bestandteile von neuen, komplexeren Klassen zu verwenden. Neben der später noch ausführlich zu behandelnden Vererbung ist diese *Aggregation* von Klassen

eine sehr effektive Technik zur Wiederverwendung von Software bzw. zum Aufbau von komplexen Softwaresystemen. 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.

Wegen der großen Bedeutung der Aggregation soll ihr ein ausführliches Beispiel gewidmet werden, obwohl der aktuelle Abschnitt nur einen neuen Begriff für eine längst vertraute Situation bringt. Schließlich enthält unsere Klasse `Bruch` schon lange das Instanzobjekt `etikett` aus der Klasse **String**. Wir erweitern das Bruchrechnungsprogramm um eine Klasse namens `Aufgabe`, die Trainingssitzungen unterstützen soll. 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, "Das korrekte Ergebnis:");
      antwort = new Bruch();
      Init();
   }

   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;
      }
   }
```

[1] Die betroffenen Personen mögen den Fachterminus *Objekt* nicht persönlich nehmen.

```
      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();
            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 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 folgendem 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

                          9
 Das korrekte Ergebnis: -----
                          10
```

4.8.2 Innere (geschachtelte) Klassen

Eine Klasse darf neben Feldern, Methoden etc. auch Klassendefinitionen enthalten, wobei *innere (geschachtelte) Klassen* entstehen, die sich vor allem für lediglich lokal benötigte Typen eignen. 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 Info() {
  Console.WriteLine("Die Kinder von Familie {0}:\n", name);
  t.Info();
  s.Info();
 }

 class Tochter {
  Familie f;
  string name;
  int alter;

  public Tochter(Familie f_, string name_, int alt_) {
  f = f_;
  name = name_;
  alter = alt_;
  }

  public void Info() {
  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 Info() {
     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() {
      Familie f = new Familie("Müller", "Lea", 7, "Leo", 4);
      f.Info();
   }
}
```

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 privaten Member der umgebenden Klasse zugreifen, was z.B. in der folgenden Anweisung des Beispiels geschieht (`name` ist eine private Instanzvariable in der Klasse `Familie`):
  ```
  Console.WriteLine("  Ich bin der {0}-jährige Sohn {1} von Familie {2}.",
                    alter, name, f.name);
  ```
- Für die privaten Member einer inneren Klasse hat auch die umgebende Klasse keine Zugriffsrechte, weshalb im Beispiel die Konstruktoren und die `Info()`-Methoden der inneren Klassen als **public** definiert werden.
- Innere Klassen besitzen wie die sonstigen Klassen-Member (Felder, Methoden, Eigenschaften etc.) die voreingestellte Schutzstufe **private**.
- Besitzt eine innere Klasse die Schutzstufe **public**, dann können in beliebigen Klassen Objekte 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);
  ```

Aus den obigen Erläuterungen ergeben sich die folgenden Empfehlungen zur Verwendung von inneren Klassen:

- Wenn eine Klasse nicht eigenständig verwendbar, sondern nur zur Modellierung von speziellem Zubehör einer anderen Klasse einsetzbar ist, sollte sie dort als innere Klasse definiert werden. Bei der voreingestellten Schutzstufe **private** tritt die innere Klasse im restlichen Programm nicht in Erscheinung, kann also nicht angesprochen werden und auch keine Namenskollision verursachen.

- Wenn in einer Klasse bestimmte Member-Objekte Zugriff auf private Instanzvariablen des umgebenden Objekts benötigen, hilft die Definition einer inneren Klasse für die zu privilegierenden Member-Objekte.
- Eine innere Klasse mit dem Zugriffsmodifikator **public** zu versehen, ist selten sinnvoll. Eine Ausnahme liegt dann vor, wenn ein Objekt der inneren Klasse über eine Eigenschaft der umgebenden Klasse durch die Außenwelt ansprechbar sein soll.

4.9 Verfügbarkeit von Klassen und Klassenbestandteilen

Nachdem die Datenkapselung mehrfach als wesentlicher Vorzug bzw. 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 kleine Vorgriffe auf das Thema *Vererbung* nötig sind.

Bei einer normalen (nicht geschachtelten) Klasse kennt C# folgende Stufen der Verfügbarkeit (vgl. ECMA 2006, S. 274):

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 in Abschnitt 4.10 demonstriert werden soll.

Für Klassen-Member (Felder, Methoden, Eigenschaften, innere Klassen etc.) unterstützt C# folgende Schutzstufen (siehe ECMA 2006, S. 31):

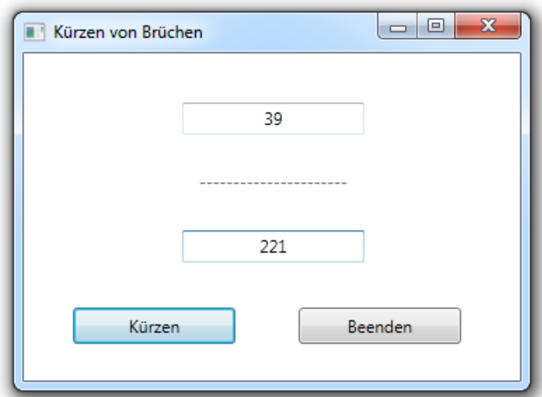
Modifikator(en)	**Der Zugriff ist erlaubt für ...**				
	eigene Klasse (Abk. K) u. innere Klassen	**nicht von K abgel. Klassen im eigenen Assembly**	**von K abgeleitete Klassen**		**sonstige Klassen**
			im eigenen Assembly	**in anderen Assemblies**	
ohne oder **private**	ja	nein	nein	nein	nein
internal	ja	ja	ja	nein	nein
protected	ja	nein	geerbte M.	geerbte M.	nein
protected internal	ja	ja	ja	geerbte M.	nein
public	ja	ja	ja	ja	ja

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 und Delegaten).

4.10 Bruchrechnungsprogramm mit WPF-Bedienoberfläche

Nachdem Sie nun wesentliche Teile der objektorientierten Programmierung mit C# kennen gelernt haben, ist vielleicht ein weiterer Ausblick auf die nicht mehr sehr ferne Entwicklung von Windows-Programmen mit graphischer Benutzerschnittstelle als Belohnung und Motivationsquelle angemessen. Schließlich gilt es in diesem Kurs auch die Erfahrung zu vermitteln, dass Programmieren Spaß machen kann. Wir erstellen nun das schon in Abschnitt 1.1.5 präsentierte Bruchkürzungsprogramm mit graphischer Bedienoberfläche:



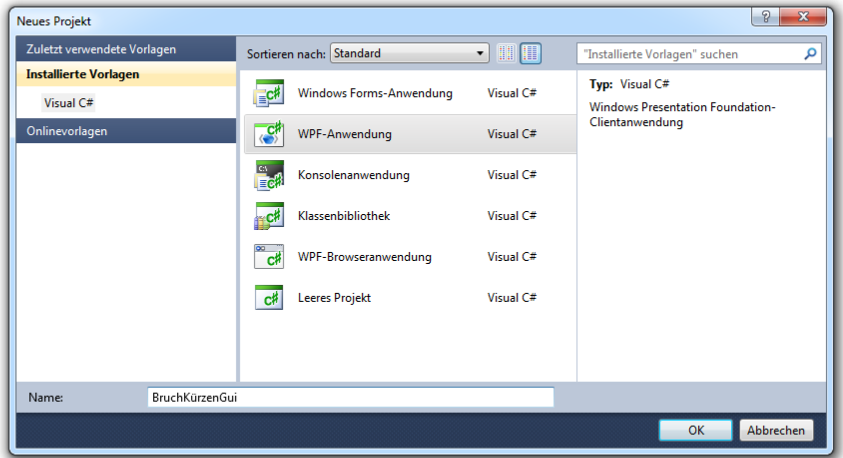
Bei dieser Gelegenheit werden wir unser Wissen über Strukturen und Abläufe bei der Erstellung einer WPF-Anwendung erweitern, um später bei der „offiziellen" Behandlung des Themas schon einige Erfahrungen einbringen zu können. Schließlich gilt es auch, zunehmend besser mit den wichtigsten Werkzeugen unserer Entwicklungsumgebung vertraut zu werden.

4.10.1 Projekt anlegen mit Vorlage *WPF - Anwendung*

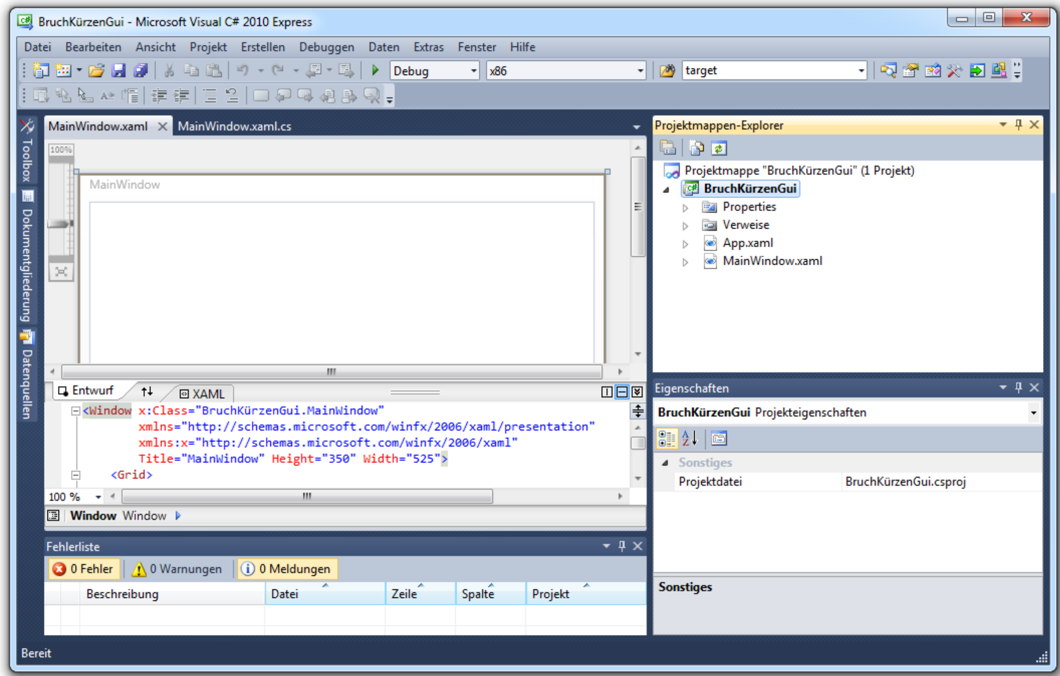
In der folgenden Beschreibung wird die Visual C# 2010 Express Edition verwendet, doch bestehen keine wesentlichen Bedienungsunterschiede zum Visual Studio 2010 Ultimate (vgl. Abschnitt 2.2.1.3). Verwenden Sie nach

Datei > Neues Projekt

für ein neues Projekt mit dem Namen `BruchKürzenGui` die Vorlage **WPF - Anwendung**:



Nach einem Mausklick auf **OK** präsentiert die Entwicklungsumgebung im **WPF-Designer** einen Rohling für das Hauptfenster der entstehenden Anwendung:

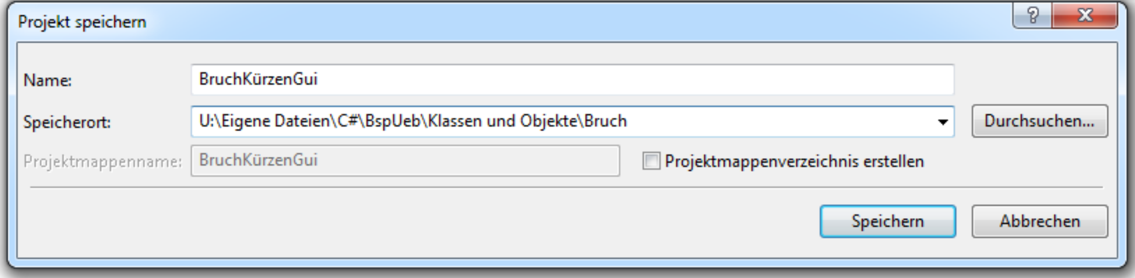


In der oberen Designer-Zone können wir die Bedienoberfläche unseres Programms mit Hilfe von graphischen Werkzeugen erstellen und dazu konfigurierbare Komponenten (Steuerelemente) aus der Toolbox (siehe Abschnitt 4.10.3) übernehmen. Wie gleich zu sehen sein wird, gestalten wir dabei den Auftritt von Objekten aus der neuen Klasse `MainWindow` im Namensraum `BruchKürzenGui`.

Speichern Sie das neue Projekt über

Datei > Alle speichern

in einem Ordner Ihrer Wahl (festgelegt durch Namen und Basisverzeichnis), z.B.:



Durch den Verzicht auf ein **Projektmappenverzeichnis** wählen wir eine „flache" Projektdateiverwaltung ohne Projektunterordner im Projektmappenordner.

4.10.2 Deklaration von WPF-Klassen per XAML

Ein zentrales Merkmal der WPF-Technologie besteht darin, das GUI-Design einer Anwendung durch eine XML-Spezialisierung (*eXtended Markup Language*) namens XAML (*eXtended Application Markup Language*) zu deklarieren. Zu unserem Anwendungsfenster gehört also eine XAML-Datei, die im unteren Teil der Designer-Zone erscheint:

```
<Window x:Class="BruchKürzenGui.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>
```

Das **Window**-Element, das mit dem Tag

```
<Window . . . >
```

startete und mit dem Tag

```
</Window>
```

endet, definiert ein Fenster, also das Erscheinungsbild eines Objekts aus einer anwendungseigenen Klasse, die von der FCL-Klasse **Window** abstammt. Initial enthält es im Beispiel:

- ein Attribut namens **x:Class**, das die zugehörige Klasse samt Namensraum nennt, in unserem Fall also `BruchKürzenGui.MainWindow`
- zwei **xmlns**-Attribute, die Namensräume für die anschließend verwendeten XAML-Elemente angeben
- ein **Title**-Attribut für die Fensterbeschriftung
- die Attribute **Height** und **Width** für die initiale Fenstergröße
- ein **Grid**-Element, das als Container dient und später noch ausführlich besprochen wird

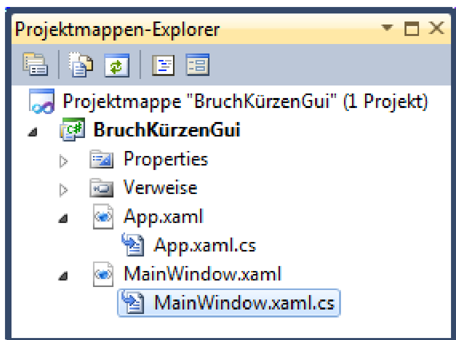
Der graphischen Fenster-Designer ist lediglich ein (willkommenes) Werkzeug zur Bearbeitung der XAML-Datei.

Unsere XAML-Datei 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.g.cs**
  Diese Datei wird automatisch durch Übersetzen der XAML - Deklarationen in C# - Quellcode erstellt und vor unseren Blicken relativ gut verborgen, weil direkte Änderungen durch Programmierer *nicht* vorgesehen sind.
- **MainWindow.xaml.cs**
  Hier werden unsere Beiträge zur Funktionalität des Programms landen.

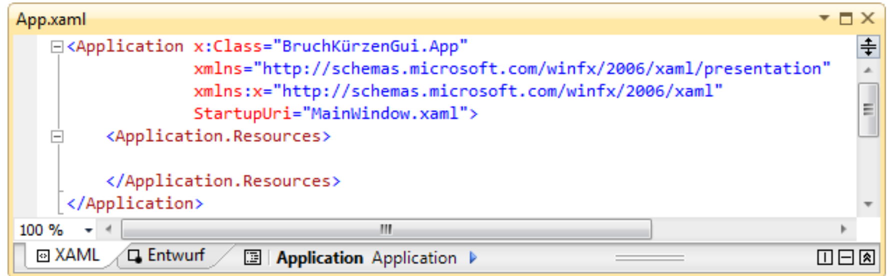
Weitere Details zu den beiden C# - Dateien folgen in Abschnitt 4.10.6.

Der Projektmappen-Explorer zeigt die XAML-Datei und die für Programmierer relevante C# - Datei:



Über die Hauptfensterklasse hinaus benötigt eine WPF-Anwendung auch noch eine Anwendungsklasse, abstammend von der FCL-Klasse **Application**, wobei erneut eine XAML-Deklarationsdatei und zwei C# - Quellcodedateien beteiligt sind. 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 über das Attribut **StartupUri** das beim Programmstart anzuzeigende Fenster festgelegt wird:

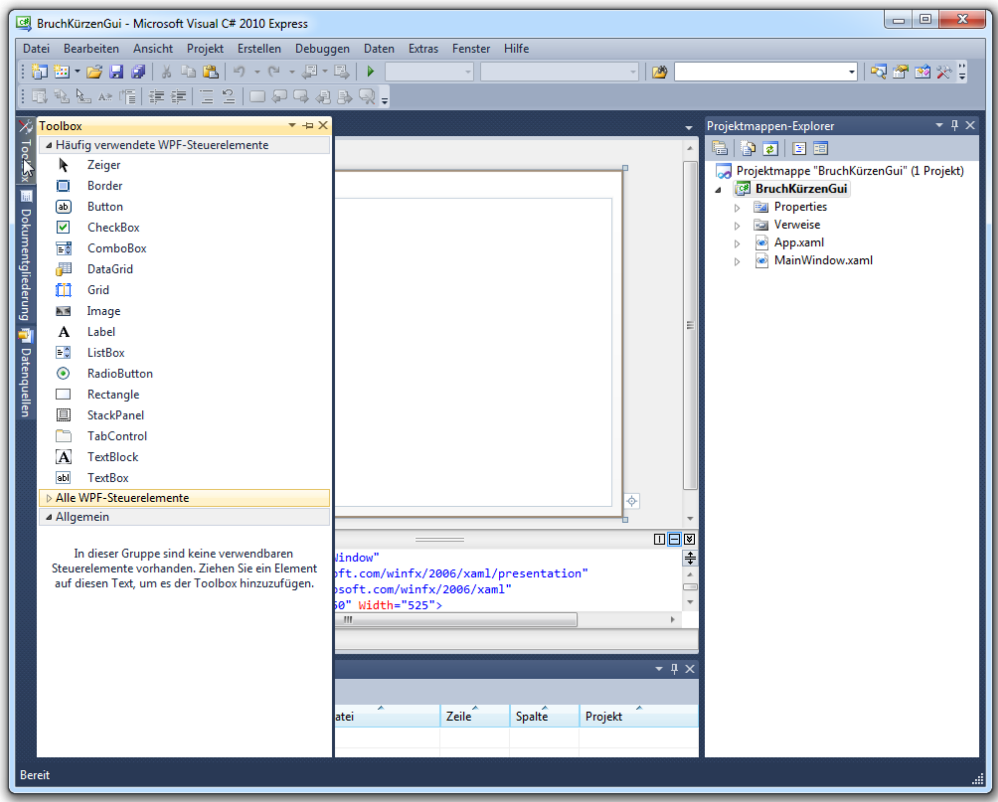


4.10.3 Steuerelemente aus der Toolbox übernehmen

Öffnen Sie das **Toolbox**-Fenster mit dem Menübefehl

Ansicht > Toolbox

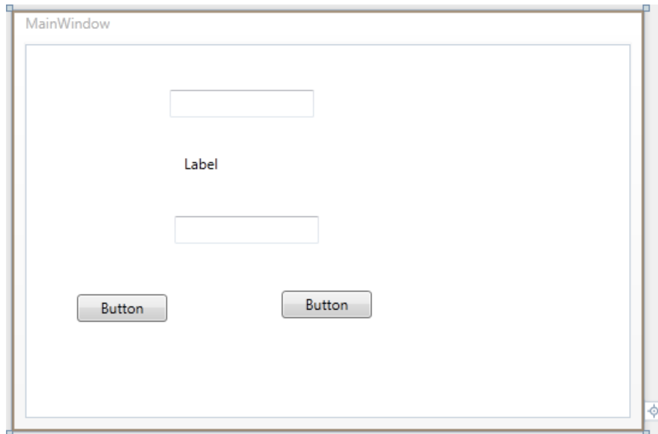
oder per Mauszeiger durch kurzes Verharren auf der **Toolbox**-Schaltfläche am linken Fensterrand:



Erweitern Sie nötigenfalls im **Toolbox**-Fenster die Liste mit den **Häufig verwendeten WPF-Steuerelementen**, und erstellen Sie auf dem Formular zwei **TextBox**-Objekte, ein **Label**-Objekt sowie zwei **Button**-Objekte, ...

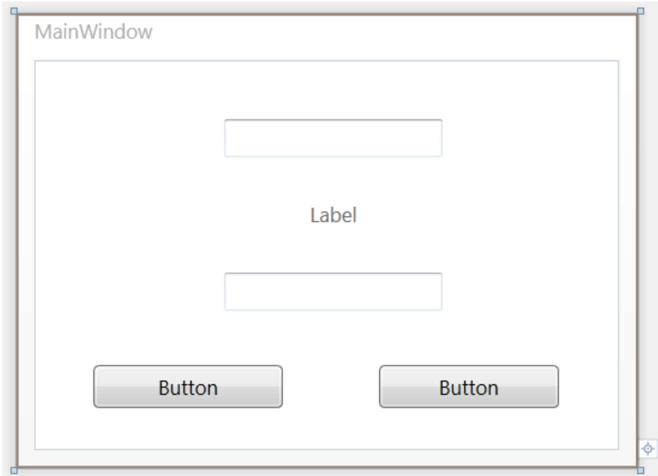
- entweder per Doppelklick auf den jeweiligen **Toolbox**-Eintrag
- oder per Drag & Drop (Ziehen und Ablegen), indem Sie einen linken Mausklick auf den jeweiligen **Toolbox**-Eintrag setzen, die Maus mit gedrückter Taste zum Ziel bewegen und dort die Taste wieder loslassen.

Das Ergebnis sollte ungefähr so aussehen:

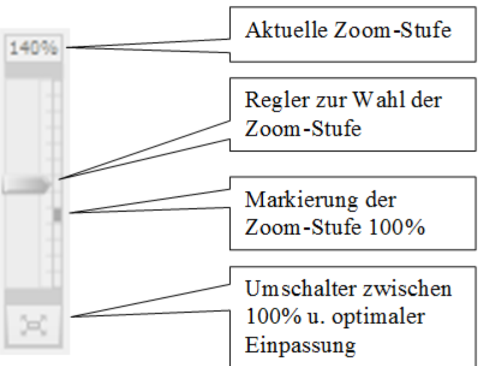


4.10.4 Positionen und Größen der Steuerelemente gestalten

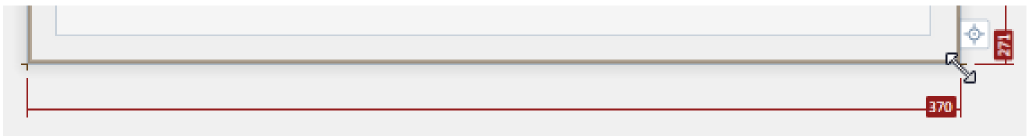
Nun können Sie wie in einem Grafikprogramm die Positionen und Größen der Fensterbestandteile verändern, um das gewünschte Layout zu erzielen, z.B.:



Zur Erleichterung der Arbeit lässt sich mit dem **Zoom-Werkzeug** des WPF-Designers (oben links) eine passende Ansicht einstellen:



Wählen Sie zunächst für das Hauptfenster eine Größe über den Anfasser unten rechts;



Die aktuelle Breite und Höhe in Bildschirm-Pixeln wird mit rot hinterlegter Schrift angezeigt. Achten Sie darauf, dass Sie wirklich das Hauptfenster erwischen (ein Objekt der Klasse `MainWindow`)

und nicht etwa den darin enthaltenen und aus didaktischen Gründen vorläufig ignorierten Container (ein Objekt aus der Klasse **Grid**).

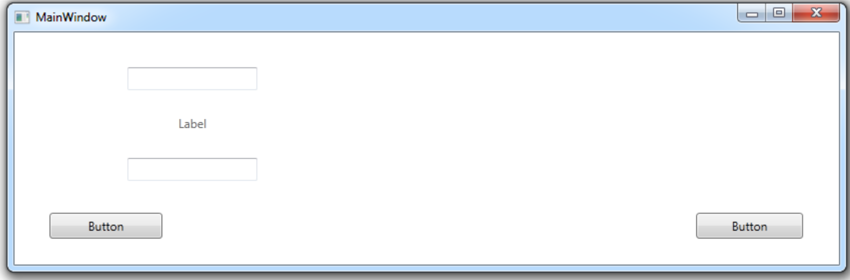
Mit der Schaltfläche

in der rechten unteren Ecke kann man zwischen zwei Optionen für die Hauptfensterfläche zur Laufzeit umschalten:

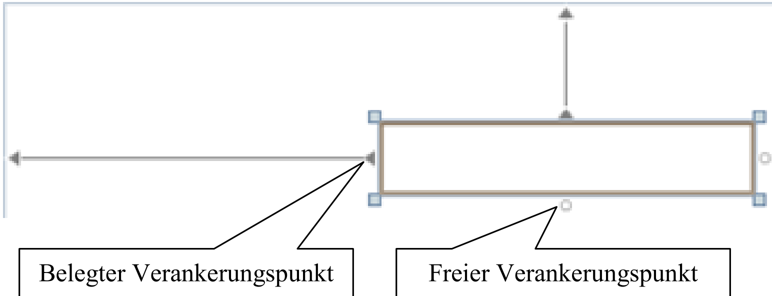
gleiche Größe wie in der Entwurfsansicht bzw. automatische Größenberechnung

Bleiben Sie bitte bei der Voreinstellung .

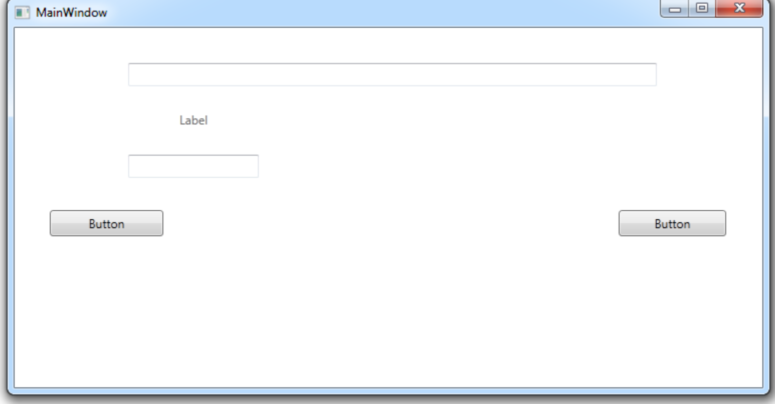
Ändert der Benutzer im laufenden Pogramm die Größe des Hauptfensters, dann hängt das Orts- und Größenverhalten eines Steuerelements davon ab, zu welchen Seiten des umgebenden Containers es einen festen **Randabstand** einhält, z.B.:



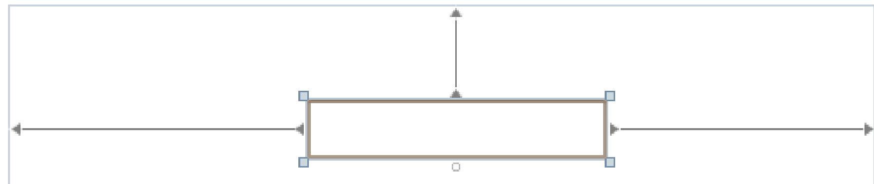
Per Voreinstellung sind die Steuerelemente links und oben angedockt, was bei einem markierten Steuerelement durch Verankerungslinien zum jeweiligen Rand angezeigt wird, z.B.:



Um eine zusätzliche Verankerung vorzunehmen, klickt man auf den zugehörigen Verankerungspunkt. Um eine Dockseite aufzugeben klickt man auf den belegten Verankerungspunkt. Ist die gegenüberliegende Seite frei, springt die Verankerung dorthin. Soll ein Steuerelement z.B. vom horizontalen Größenwachstum des Hauptfensters profitieren, auf vertikale Änderungen aber nicht reagieren,



wählt man die folgende Verankerung:

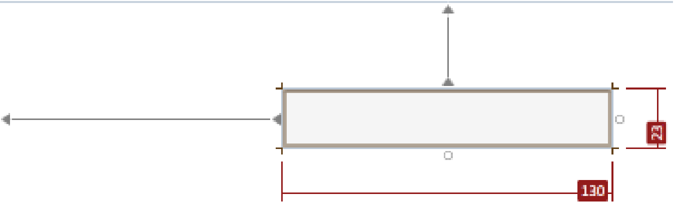


Bei der Positions- und Größenanpassung von Steuerelementen helfen **Ausrichtungs- bzw. Führungslinien**, die im folgenden Beispiel (das **Label**-Objekt wird bewegt) auftauchen und dabei anziehend wirken, sobald die horizontale Position der potentiellen Kandidaten für eine linksbündige Ausrichtung erreicht wird:

Außerdem werden im Beispiel vertikale Abstände angezeigt: Die beiden **TextBox**-Objekte sind 24 bzw. 16 Pixel entfernt.

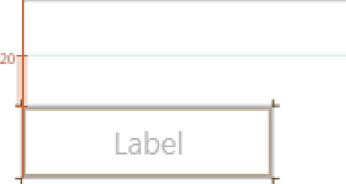
Man kann also z.B. so vorgehen, um die Steuerelemente für Zähler, Nenner und Bruchstrich anzuordnen:

- Größe und Position für das obere Textfeld festlegen, z.B.:

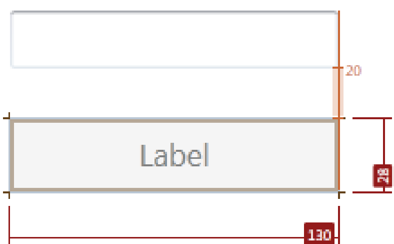


  Die rot hinterlegten Zahlen geben die Breite und Höhe des Textfelds an.

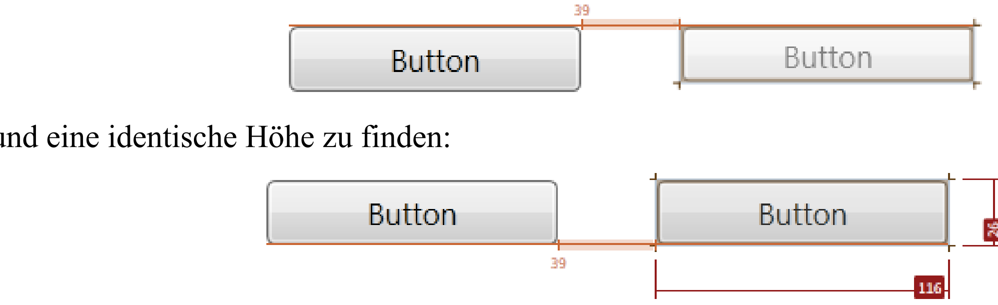
- Das **Label**-Objekt und das untere Textfeld nacheinander ...
  - linksbündig im Abstand von 20 Pixeln unter das jeweils darüber liegende Steuerelement setzen, z.B.:



  - und dieselbe Breite wählen, z.B.:



Bei den **Button**-Objekten helfen Führungslinien dabei, eine identische vertikale Position



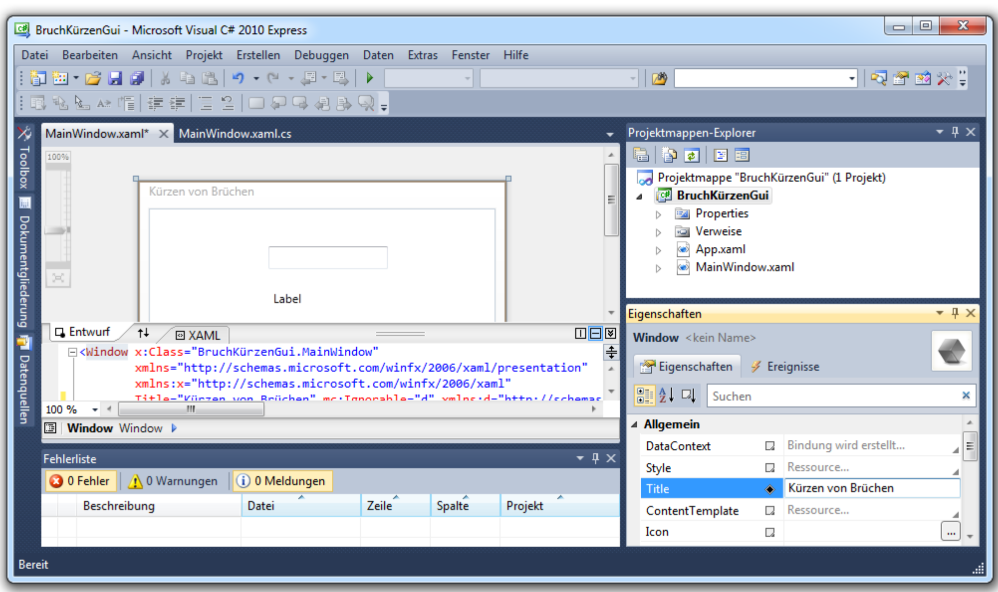
und eine identische Höhe zu finden:

4.10.5 Eigenschaften der Steuerelemente ändern

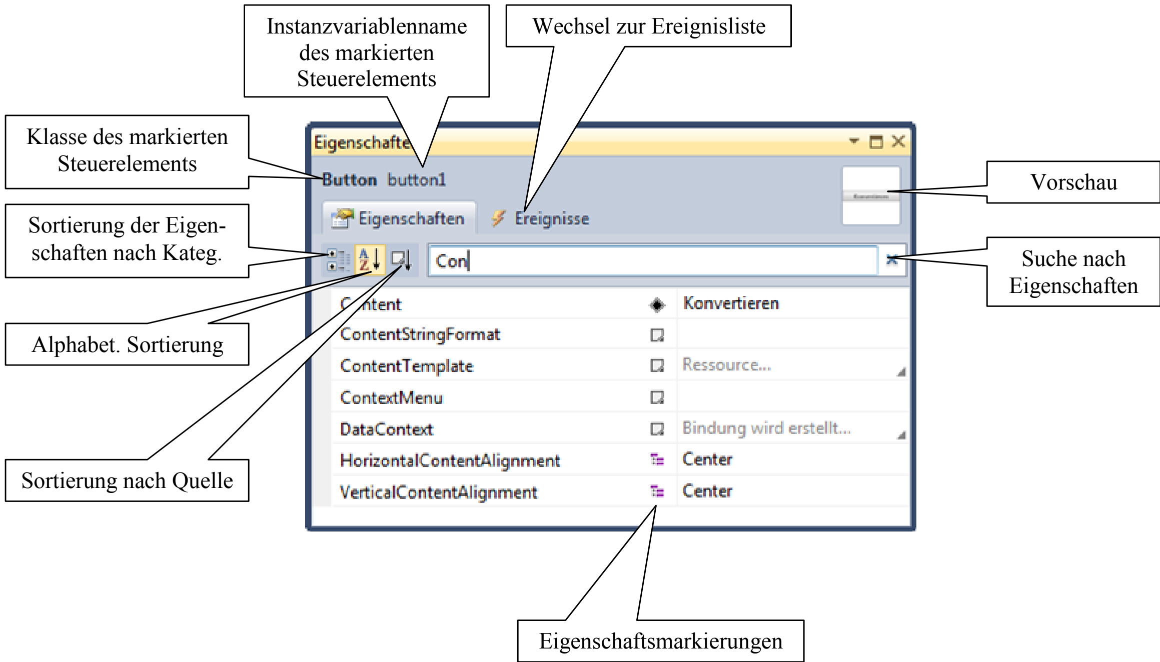
Im Eigenschaftenfenster der Entwicklungsumgebung, das bei Bedarf über das Kontextmenü zu einem Steuerelement, mit der Funktionstaste **F4** oder mit dem Menübefehl

Ansicht > Eigenschaftenfenster

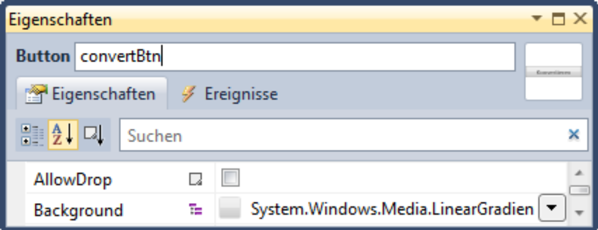
zu öffnen ist, lassen sich diverse Eigenschaften (im Sinne von Abschnitt 4.5!) des markierten Formularobjekts durch direkte Werteingabe festlegen, z.B. der **Titel** des Fensters:



Ist im WPF-Designer ein Steuerelement markiert, bietet das Eigenschaftenfenster folgende Bedienmöglichkeiten:



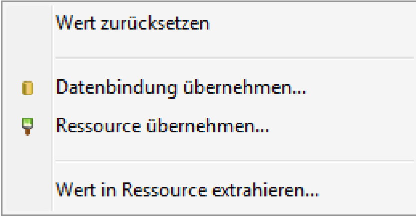
Neben den Eigenschaftsmodifikationen ist insbesondere die Möglichkeit von Relevanz, den Instanzvariablennamen eines Steuerelements zu ändern, ohne die Deklarationsstelle im Quellcode der Klasse kennen zu müssen, z.B.:



Für Neugierige soll hier schon die Bedeutung der Eigenschaftsmarkierungen erläutert werden, obwohl dieses Wissen im aktuellen Beispiel keinen großen Vorteil bringt:

- ◈ Wertquelle lokal (Werteingabe im Eigenschaftenfenster)
- Wertquelle Vererbung
- Wertquelle Ressource (siehe unten)
- Wertquelle Datenbindung (siehe unten)
- Wert ist noch gleich Voreinstellung

Auch das per Mausklick auf eine Eigenschaftsmarkierung zu öffnende Menü mit Optionen zur erweiterten Eigenschaftsbearbeitung ist für uns aktuell noch wenig relevant:



Man kann ...

- den Wert auf die Voreinstellung **zurücksetzen,**
- einen Assistentendialog zur Definition einer **Datenbindung** anfordern,
- die Eigenschaft mit einer **Ressource** verknüpfen,
- aus dem aktuellen Wert eine Ressource erstellen, die sich an anderer Stelle wiederverwenden lässt.

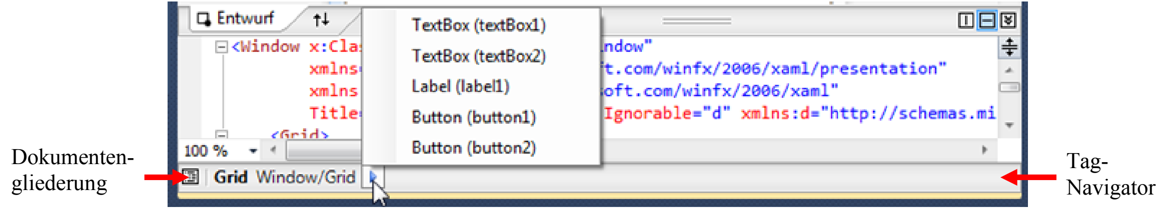
Nach der neugierigen Beschäftigung mit den Bedienmöglichkeiten des Eigenschaftenfensters kehren wir nun den Entwicklungsaufgaben im aktuellen Projekt zurück und ändern …

- die **Content** - Eigenschaft der Schaltflächen, so dass sinnvolle Beschriftungen entstehen,
- die **Content** - Eigenschaft des Labels, um einen behelfsmäßigen Bruchstrich einzuzeichnen,
- für beide Textboxen die Eigenschaft **TextAlignment** auf den Wert **Center**,
- die **IsDefault**-Eigenschaft der linken Schaltfläche, so dass diese im laufenden Programm per **Enter**-Taste angesprochen werden kann. Im WPF-Designer wird eine **Default**-Schaltfläche optisch hervorgehoben (wie später im laufenden Programm):

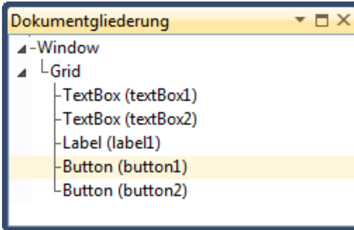
- Beim Verschieben eines Steuerelements im WPF-Designer (siehe Abschnitt 4.10.4) haben wir seine **Margin**-Eigenschaft verändert und so für jede Verankerungsseite einen Randabstand festlegt.

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.
- Unter dem XAML-Fenster befindet sich der **Tag-Navigator**, der zum gerade markierten Element anbietet:
    - den Namen
    - den Pfad beginnend mit der Wurzel **Window**
    - eine Klappliste mit den untergeordneten Elemente, z.B.:



- Ein Schalter am linken Rand des Tag-Navigators öffnet das Fenster der **Dokumentengliederung**, die eine gute Übersicht bietet und einen bequemen Wechsel zwischen den Objekten erlaubt, z.B.:



Um *mehrere* Steuerelemente zu markieren, kann man ...

- im WPF-Designer ein Markierungsrechteck um die gewünschten Teilnehmer ziehen,
- im WPF-Designer oder in der Dokumentengliederung bei gedrückter **Strg**-Taste die Steuerelemente nacheinander anklicken

Die Eigenschaften von mehreren, gleichzeitig markierten Formular-Objekten 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.

4.10.6 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:

- **MainWindow.xaml.cs** im Projektordner
  Hier werden die von Ihnen erstellten Methoden landen (siehe unten).
- **MainWindow.g.cs** im Projektunterordner **...\obj\x86\Debug**
  Hier landet der durch Interpretieren der XAML-Datei **MainWindow.xaml** erstellte C# - Quellcode.[1] In dieser Datei dürfen Sie keine Änderungen vornehmen, um die Entwicklungsumgebung nicht aus dem Tritt zu bringen. Daran erinnert der Namensbestandteil *g* für *generated*.

Dem C# - Compiler wird durch das Schlüsselwort **partial** in der Klassendefinition signalisiert, dass der Quellcode auf mehrere Dateien verteilt ist, z.B.:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
  . . .
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.

Aufgrund Ihrer Tätigkeit im Formulardesigner enthält die Fensterklasse `MainWindow` im Sinne der in Abschnitt 4.8.1 behandelten Aggregation mehrere Objekte anderer Klassen, die Steuerelemente

[1] Beim Studieren von (Fehler)meldungen hilft eventuell das Wissen, dass die Übersetzung der XAML - Deklarationen in C# - Quellcode vom Werkzeug **msbuild.exe** vorgenommen wird.

der graphischen Bedienoberfläche repräsentieren. In der Datei **MainWindow.g.cs** finden sich die Deklarationen der zugehörigen Instanzvariablen:

```
internal System.Windows.Controls.TextBox textBox1;
internal System.Windows.Controls.TextBox textBox2;
internal System.Windows.Controls.Label label1;
internal System.Windows.Controls.Button button1;
internal System.Windows.Controls.Button button2;
```

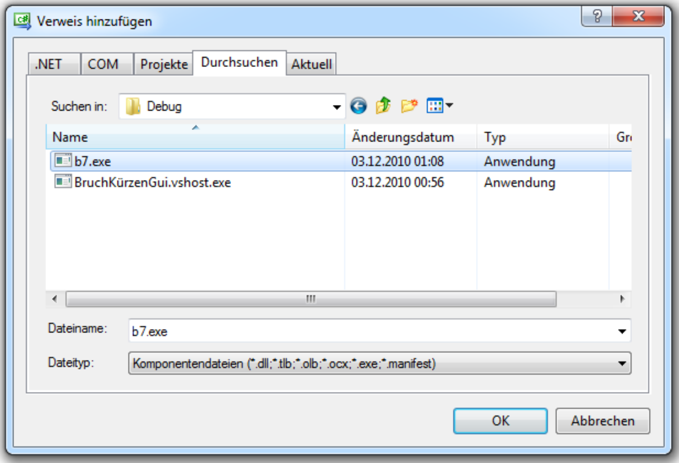
Neben der Klasse `MainWindow` mit der XAML-Datei **MainWindow.xaml** und dem C# - Quellcode-Duo (**MainWindow.xaml.cs**, **MainWindow.g.cs**) enthält das Projekt noch die Klasse `App` mit der XAML-Datei **App.xaml** und dem C# - Quellcode-Duo (**App.xaml.cs**, **App.g.cs**).

Eine weitergehende Analyse des automatisch erstellten Quellcodes sparen wir uns an dieser Stelle.

4.10.7 Assembly mit der Bruch-Klasse einbinden

Gehen Sie folgendermaßen vor, um Objekte unserer Beispielklasse `Bruch` im neuen Programm nutzen zu können:

- Kopieren Sie die Assembly-Datei mit der Klasse `Bruch`, z.B.
  …\BspUeb\Klassen und Objekte\Bruch\b7 Operatoren-Überladung\bin\Debug\b7.exe
  in den Debug-Ausgabeordner Ordner des neuen Projekts, also z.B. nach
  U:\Eigene Dateien\C#\BspUeb\Klassen und Objekte\Bruch\BruchKürzenGui\bin\Debug
  In der Praxis werden sich allgemein benötigte Assembly-Dateien an einem sinnvollen Ort befinden (z.B. im Global Assembly Cache, GAC, siehe Abschnitt 1.2.5.4), so dass sie zur Nutzung in einem konkreten Projekt nicht kopiert werden müssen.
- Nehmen Sie per Projektmappen-Explorer die Assembly-Datei **Bruch.exe** in die Verweisliste des Projekts auf, z.B.:



Die Klasse `Bruch` befindet sich im globalen Namensraum, weil wir auf eine **namespace**-Definition verzichtet haben. Folglich ist beim Zugriff (im Unterschied zu den FCL-Klassen) kein Namensraum anzugeben. Wegen der Gefahr von Namenskollisionen ist dieses Verfahren nicht empfehlenswert bei Klassen(bibliotheken), die in vielen Projekten verwendet werden sollen. Unsere Entwicklungsumgebung definiert grundsätzlich einen Namensraum (abgesehen von der *leeren* Projektvorlage) unter Verwendung des Projektnamens, so auch im aktuellen Beispiel:

```
namespace BruchKürzenGui {
   . . .
}
```

4.10.8 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 `button1` (mit dem Wert *Kürzen* für die Eigenschaft **Content**), so dass die Entwicklungsumgebung in der Quellcodedatei **MainWindow.xaml.cs** die private Instanzmethode `button1_Click()` der Klasse `MainWindow` mit leerem Rumpf anlegt

```
private void button1_Click(object sender, RoutedEventArgs e) {

}
```

und die Datei im Editor öffnet.

Mit Hilfe eines Objekts aus unserer Klasse `Bruch` ist die benötigte Funktionalität leicht zu implementieren, z.B.:

```
private void button1_Click(object sender, RoutedEventArgs e) {
   Bruch b = new Bruch();
   try {
      b.Zaehler = Convert.ToInt32(textBox1.Text);
      b.Nenner = Convert.ToInt32(textBox2.Text);
      b.Kuerze();
      textBox1.Text = b.Zaehler.ToString();
      textBox2.Text = b.Nenner.ToString();
   }
   catch {
      MessageBox.Show("Eingabefehler", "Fehler", MessageBoxButton.OK,
                      MessageBoxImage.Error);
   }
}
```

Der Bequemlichkeit halber wird eine *lokale* `Bruch`-Referenzvariable verwendet, so dass bei jedem Methodenaufruf ein neues Objekt entsteht (vgl. Übungsaufgabe in Abschnitt 4.11).

Tritt im **try**-Block der **try-catch** – Anweisung eine Ausnahme auf (z.B. beim Versuch, irreguläre Benutzereingaben 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 `button2` (mit dem Wert *Beenden* für die Eigenschaft **Content**) den Rohling für seine Click-Ereignisbehandlungsmethode, und ergänzen Sie einen Aufruf der **Window**-Methode **Close()**, die das Hauptfenster schließt und damit das Programm beendet, z.B.:

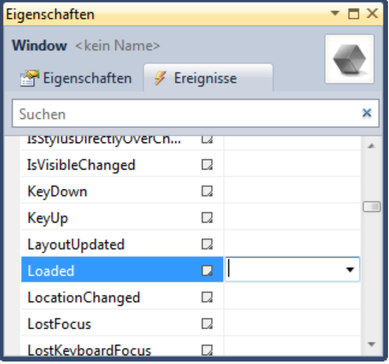
```
private void button2_Click(object sender, RoutedEventArgs e) {
   this.Close();
}
```

Es wäre nett, wenn nach dem Start unseres Programms das Zähler-Textfeld den Tastaturfokus hätte, so dass der Benutzer unmittelbar mit der Werteingabe beginnen könnte, ohne zuvor den Tastaturfokus (z.B. per Maus) setzen zu müssen. Eine Lösungsmöglichkeit besteht darin, die statische Methode **Focus**() der Klasse **Keyboard** aufzurufen, die als Argument eine Referenz auf das privilegierende Objekt erwartet, z.B.:

```
Keyboard.Focus(textBox1);
```

Damit diese Anweisung beim Laden des Hauptfensters ausgeführt wird, stecken wir sie in eine Ereignisbehandlungsmethode zum **Loaded**-Ereignis der Fensterklasse:

- Markieren Sie im WPF-Designer das Hauptfenster.
- Wechseln Sie im Eigenschaftenfenster zur Registerkarte **Ereignisse**.
- Setzen Sie einen Doppelklick auf das 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) {
}
```

- Ergänzen Sie im Rumpf dieser Methode den oben beschriebenen **Focus()**-Aufruf.

Veranlassen Sie mit der Funktionstaste **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 2.2.3). Beachten Sie beim Kopieren/Verschieben Ihres Programms, dass sich eine Assembly-Datei mit der verwendeten Klasse `Bruch` im selben Ordner befinden muss.

4.11 Übungsaufgaben zu Kapitel 4

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 stets 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 bitte den Unterschied zwischen einem **readonly** – deklarierten Feld und einer Eigenschaft ohne **set** – Implementierung, die man auch als *getonly* – Eigenschaft bezeichnen könnte.

3) Im folgenden Programm soll die statische `Bruch`-Eigenschaft `Anzahl` ausgelesen werden:

```
using System;
class BruchRechnung {
  static void Main() {
    Bruch b1 = new Bruch(), b2 = new Bruch();
    Console.WriteLine("Jetzt sind wir " + Bruch.Anzahl);
  }
}
```

Es liegt folgende 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) Die folgende Aufgabe eignet sich nur für Leser mit Grundkenntnissen in linearer Algebra: Erstellen Sie eine Klasse für Vektoren im $\mathbb{R}^2$, die mindestens über Methoden bzw. Eigenschaften mit folgenden Leistungen erfügt:

- **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:

  $$\begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \end{pmatrix}$$

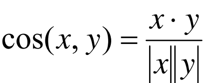
- **Skalarprodukt** zweier Vektoren ermitteln

  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:

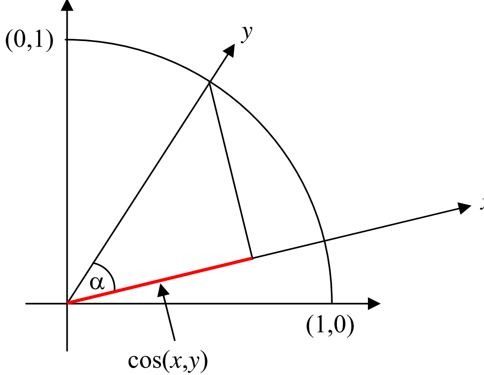
  $$x \cdot y := x_1 y_1 + x_2 y_2$$

- **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]

[1] Dies folgt aus dem Additionstheorem für den Kosinus.



$$\cos(x, y) = \frac{x \cdot y}{|x||y|}$$



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*):

$$deg = \frac{rad}{2\pi} \cdot 360$$

- **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 Vektor-Klasse 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 sicher nicht geeignet.

6) Ersetzen Sie beim GUI-Bruchkürzungsprogramm in Abschnitt 4.10 die lokale `Bruch`-Referenzvariable in der Klick-Ereignisbehandlungsmethode zum Befehlsschalter `button1` (mit dem Wert *Kürzen* für die Eigenschaft **Content**) durch eine Instanzvariable der Formularklasse `Form1`. 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 in dieser Abbildung mit einer Kurzform der Klasse `Bruch`

```
using System;
public class Bruch {
      int zaehler, nenner = 1;
      string etikett = "";
      static int anzahl;

      public Bruch(int zpar, int npar, String epar) {
            zaehler = zpar; Nenner  = npar;
            etikett = epar; anzahl++;
      }
      public Bruch() {anzahl++;}

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

      public void Addiere(Bruch b) {
            zaehler = zaehler*b.nenner + b.zaehler*nenner;
            nenner = nenner*b.nenner;
            Kuerze();
      }
      public Bruch Klone() {
            return new Bruch(zaehler, nenner, etikett);
      }
      public void Kuerze() {  . . . }
      public bool Frage() {  . . . }
      public void Zeige() {
            string luecke = "";
            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) {
            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) {
            Bruch b = new Bruch(0, 1, e);
            if (b.Frage()) {
                  b.Kuerze();
                  return b;
            } else
                  return null;
      }
      public static int Anzahl {
            get {return anzahl;}
      }
}
```

12 Begriffe, und tragen Sie die Positionen in die folgende Tabelle ein

Begriff	Pos.	Begriff	Pos.

Definition einer Instanzmethode mit Referenzrückgabe		Konstruktordefinition	
Deklaration lokale Variable		Deklaration einer Klassenvariablen	
Def. einer Instanzmeth. mit Wertparameter vom Typ einer Klasse		Objekterzeugung	
Deklaration von Instanzvariablen		Definition einer Klassenmethode	
Methodenaufruf		Definition einer Instanzeigenschaft	
Deklaration einer statischen Eigenschaft		Operatorüberladung	

Zum Eintragen benötigen Sie nicht unbedingt eine gedruckte Variante des Manuskripts, sondern können auch das interaktive PDF-Formular

...\BspUeb\Klassen und Objekte\Begriffe lokalisieren.pdf

benutzen. Die Idee zu dieser Übungsaufgabe stammt aus Mössenböck (2003).

5 Weitere .NETte Typen

Nachdem wir uns ausführlich mit elementaren Datentypen und Klassen beschäftigt haben, wird in diesem Abschnitt 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)

5.1 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 das .NET – Framework mit den *Strukturen* auch *Werttypen* zur Verfügung, die in manchen Situationen bei geringerem Ressourcen-Verbrauch eine „echte" Klasse 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, usw.).[1] Eine Variable vom Typ einer Struktur enthält jedoch keine *Referenz* auf ein Heap-Objekt, sondern die *Daten* ihres Typs. Individuen nach dem Bauplan einer Struktur werden *nicht* als Objekte bezeichnet, sondern als **Instanzen**. Weil Objekte als spezielle Instanzen betrachtet werden, erweist es sich als sinnvoll, dass wir bisher stets von *Instanzvariablen* bzw. *-methoden* gesprochen haben.

Eine Struktur eignet sich vor allem bei folgender Konstellation:

- Der Datentyp ist relativ einfach aufgebaut (wenige Member). Microsoft empfiehlt auf einer Webseite mit Richtlinien zur Verwendung von Strukturtypen[2] eine Instanz-Maximalgröße von 16 Bytes.
- Werden in einer zeitkritischen Programmsituation sehr viele Instanzen benötigt, kann sich die Vermeidung von Objektkreationen durch Verwendung eines Strukturtyps lohnen.
- Die Instanzen sollen analog zu den elementaren Typen eine Wertsemantik haben. Bei einer Zuweisung soll also keine Referenz übergeben, sondern der komplette Wert kopiert werden.
- Die Instanzen sollen nicht über das Ende der kreierenden Methode hinaus im Hauptspeicher verbleiben (es sei denn als Member von Objekten).
- Bei einer Struktur muss sichergestellt sein, dass die Nullinitialisierung aller Felder zu einer regulären Instanz führt (siehe unten).
- Bei Strukturen fehlt die Möglichkeit, über die wichtige objektorientierte Technik der Vererbung (siehe Abschnitt 6) eine Hierarchie spezialisierter Typen aufzubauen. Das Implementieren von Schnittstellen (siehe Abschnitt 8) ist aber möglich.

Typische Anwendungsbeispiele für Strukturen:

- Punkte in einem zweidimensionalen Zahlenraum
- Komplexe Zahlen[3]

---

[1] Es ist allerdings kein Destruktor erlaubt.
[2] URL: http://msdn.microsoft.com/de-de/library/y23b5415(en-us).aspx
[3] Dieser mathematische Begriff meint Paare aus reellen Zahlen, für die spezielle Rechenregeln gelten. Wer nicht mathematisch vorbelastet ist, kann das Beispiel ignorieren.

5.1.1 Vergleich von Klassen und Strukturen

Um den gravierenden Unterschied 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. Wir verzichten auf eine Datenkapselung, erlauben also auch fremden Methoden den direkten Zugriff auf die Felder.

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:

```
using System;

public struct Punkt {
   public double X, Y;

   public Punkt(double x_, double y_) {
      X = x_;
      Y = y_;
   }

   public string XY() {
        return "("+X+";"+Y+")";
   }
}
```

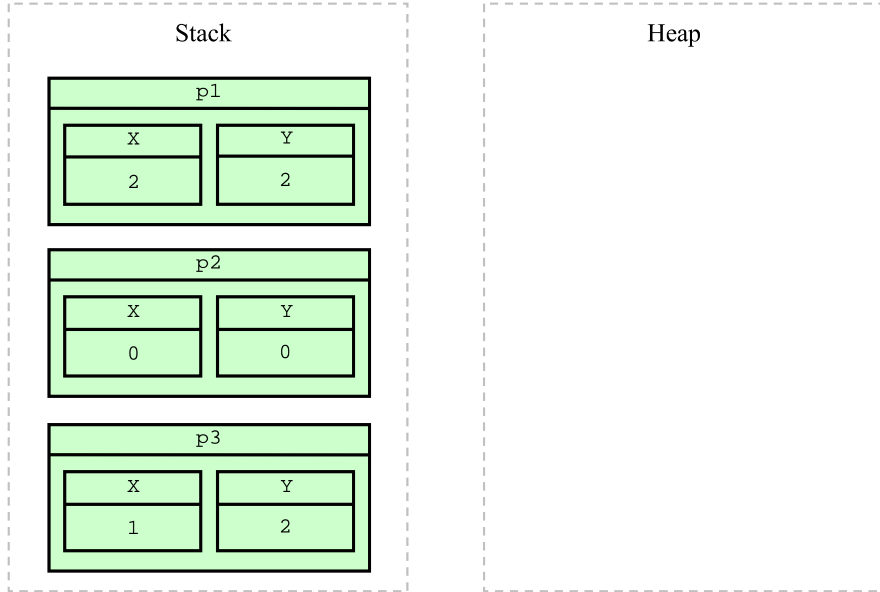
Wie die Objekte von Klassen werden auch Strukturinstanzen per **new**-Operator unter Verwendung eines Konstruktors erstellt. Beim folgenden Einsatz der `Punkt`-Struktur wird die Variable `p1` über den expliziten Konstruktor mit dem Wert (1, 2) initialisiert. Der Punkt `p2` erhält (vom *nicht* verloren gegangenen!) Standardkonstruktor eine Initialisierung auf den Wert (0, 0). Der Punkt `p3` erhält ohne Konstruktor-Beteiligung eine *Kopie* von `p1` (mit allen Instanzvariablen), wobei die spätere Änderung von `p1` ohne Effekt auf `p3` bleibt:

<table>
<tr><th>Quellcode</th><th>Ausgabe (mit <code>Punkt</code> als Struktur)</th></tr>
<tr><td><pre>using System;
class PunktDemo {
   static void Main() {
      Punkt p1 = new Punkt(1, 2);
      Punkt p2 = new Punkt();
      Punkt p3 = p1;
      p1.X = 2;
      Console.WriteLine("p1 = " + p1.XY()+
                  "\np2 = " + p2.XY() +
                  "\np3 = " + p3.XY());
   }
}</pre></td><td><pre>p1 = (2;2)
p2 = (0;0)
p3 = (1;2)</pre></td></tr>
</table>

Nach der Anweisung

```
p1.X = 2;
```

haben wir folgende Situation im Speicher des Programms:



Aus der `Punkt`–Struktur wird durch wenige Quellcode-Änderungen eine *Klasse*:

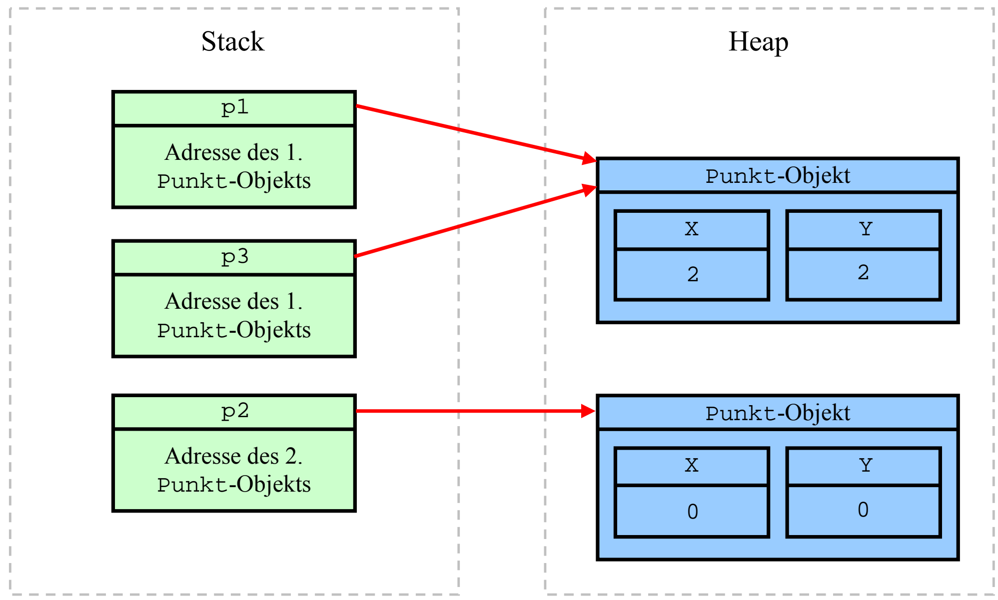
```
using System;

public class Punkt {
   public double X, Y;

   public Punkt(double x_, double y_) {
      X = x_;
      Y = y_;
   }

   public Punkt() { }

   public string XY() {
        return "("+X+";"+Y+")";
   }
}
```

Weil bei Klassen (im Unterschied zu Strukturen) der Standardkonstruktor verloren geht, sobald ein expliziter Konstruktor vorhanden ist (vgl. Abschnitt 4.4.3), hat die Klasse `Punkt` einen parameterlosen Konstruktor erhalten.

Das obige **Main()**-Programm muss beim Wechsel von der `Punkt`-Struktur zur `Punkt`-Klasse *nicht* geändert werden, zeigt aber ein leicht abweichendes Verhalten (siehe letzte Ausgabezeile):

<table>
<tr><th>Quellcode</th><th>Ausgabe (mit <code>Punkt</code> als Klasse)</th></tr>
<tr><td><pre>using System;
class PunktDemo {
   static void Main() {
      Punkt p1 = new Punkt(1, 2);
      Punkt p2 = new Punkt();
      Punkt p3 = p1;
      p1.X = 2;
      Console.WriteLine("p1 = " + p1.XY()+
                  "\np2 = " + p2.XY() +
                  "\np3 = " + p3.XY());
   }
}</pre></td><td><pre>p1 = (2;2)
p2 = (0;0)
p3 = (2;2)</pre></td></tr>
</table>

`p1`, `p2` und `p2` sind nun lokale Referenzvariablen, die auf insgesamt *zwei* Objekte zeigen:



Das erste `Punkt`-Objekt kann über die Referenzvariablen `p1` *und* `p3` angesprochen (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 natürlich dem Aufrufer via Rückgabewert eine Kopie der Strukturinstanz übergeben werden.

Neben der eigenständigen Speicherpersistenz fehlt den Strukturen vor allem die Vererbungstechnik, die wir erst in einem späteren Abschnitt gebührend gründlich behandeln werden. *Jede* Struktur stammt implizit von der Klasse **ValueType** im Namensraum **System** ab, die wiederum direkt von der .NET - Urahn-Klasse **Object** erbt (siehe Abbildung in Abschnitt 5.1.3). Alternative Abstammungen sind bei Strukturen nicht möglich, so dass auch keine Strukturhierarchien entstehen können.

Eine Strukturdefinition unterscheidet sich nur geringfügig von einer Klassendefinition, so dass wir uns an Stelle von Syntaxdiagrammen auf einige Hinweise beschränken können:

- Wie bei Klassen wird die Verfügbarkeit eines Strukturtyps über Modifikatoren geregelt (Voreinstellung: **internal**, Alternative: **public**, vgl. Abschnitt 4.9).
- Das Schlüsselwort **class** wird **struct** ersetzt.
- Bei der Deklaration von Instanzfeldern ist *keine explizite Initialisierung* erlaubt. Verboten ist also z.B.:

  ```
  double delta = 1.0;
  ```

  Felder von Strukturinstanzen werden jedoch wie die Felder von Klasseninstanzen (Objekten) per Voreinstellung mit der typspezifischen Null initialisiert.
- Es sind beliebige viele Konstruktoren erlaubt, wobei der (parameterfreie) Standardkonstruktor *nicht* verloren geht.
- Bei einer Struktur darf kein expliziter parameterloser Konstruktor definiert werden, so dass man das Initialisierungsverbot (siehe oben) nicht per Konstruktor kompensieren kann. Folglich muss bei einer Struktur sichergestellt sein, dass die Nullinitialisierung zu einer regulären Instanz führt.
- Auch bei Strukturen ist eine Datenkapselung möglich. Bei der Deklaration bzw. Definition der Member kann der Zugriffsschutz über die Modifikatoren **private** (Voreinstellung), **pub-**

lic und **internal** reguliert werden. Weil keine Vererbung unterstützt wird, ist der Modifikatoren **protected** verboten.

- Hinter der Strukturdefinition darf wie bei Klassen ein Semikolon stehen.
- Aus dem Vererbungsverbot ergeben sich einige zusätzliche Regeln (siehe ECMA 2006, S. 333).

5.1.2 Zur Eignung von Strukturen im Bruchrechnungs-Projekt

Grundsätzlich lässt sich unsere `Bruch`-Klassendefinition in eine analoge Strukturdefinition übersetzen:

```
using System;

public struct Bruch {
   int zaehler, nenner;
   string etikett;

   public Bruch(int zpar, int npar, String epar) {
      zaehler = zpar;
      nenner = (npar!=0)?npar:1;
      etikett = epar;
   }
   . . .
}
```

Im parametrisierten Konstruktor sind die Eigenschaften (und Methoden) der gerade entstehenden Instanz erst nach der kompletten Initialisierung aller Felder nutzbar, so dass z.B. die Eigenschaft `Nenner` *nicht* wie bei der Klasse `Bruch` zur kontrollierten Initialisierung des zugehörigen Felds genutzt werden kann. Daher wird im parametrisierten Konstruktor die Nenner-Null per Konditionaloperator verhindert.

Werden in einer Methode Variablen von diesem Typ deklariert und initialisiert, entstehen beim Programmablauf keine Objekte auf dem Heap, die über lokale Referenzvariablen (auf dem Stack) anzusprechen sind, sondern es entstehen lokale Variablen mit einer kompletten Struktur als Wert. Vom eingesparten Aufwand ist im folgenden Programm, das lediglich zwei `Bruch`-Instanzen einsetzt, sicher nichts zu spüren:

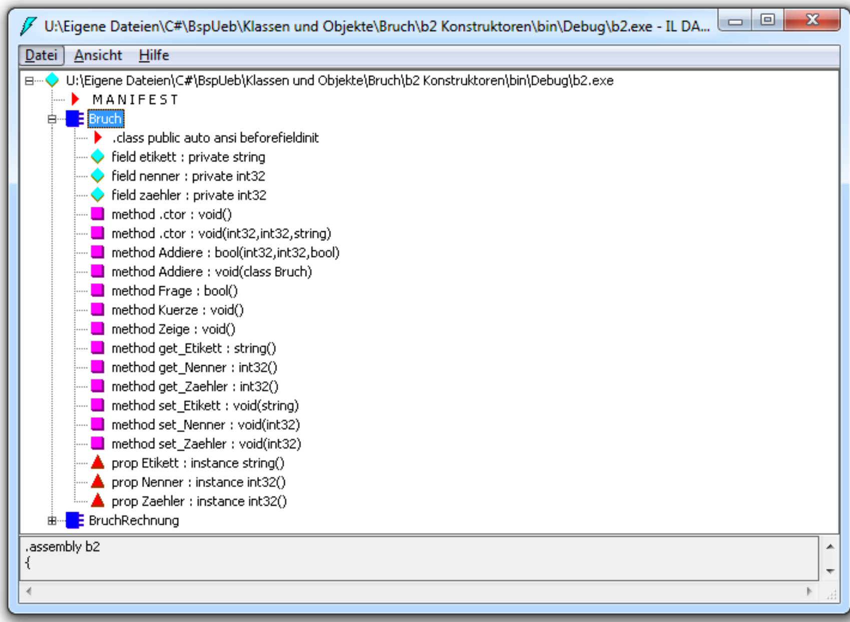
<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>using System;
class BruchRechnung {
   static void Main() {
      Bruch b1 = new Bruch(1,2,"b1 =");
      Bruch b2 = new Bruch();
      b1.Zeige();
      b2.Zeige();
   }
}</pre></td><td><pre>   1
 b1 = -----
       2

   0

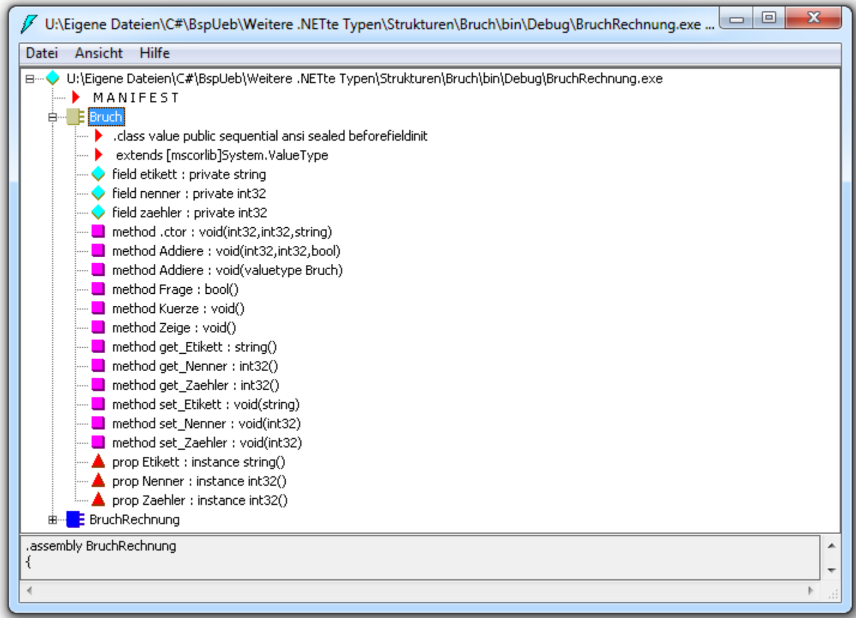
   0</pre></td></tr>
</table>

Man muss man schon eine ziemlich große Anzahl von `Bruch`-Instanzen bzw. `Bruch`-Objekten erzeugen, um nachmessen zu können (z.B. über die Eigenschaft **Ticks** der **DateTime**-Struktur), dass `Bruch`-Objekte einen ca. zwei bis dreifach höheren Zeitaufwand verursachen. Diese Messung bei `Bruch`-Objekten bzw. Instanzen kann allerdings nicht verallgemeinert werden.

Eine Assembly-Inspektion mit dem Windows-SDK - Hilfsprogramm **ILDasm** zeigt im Vergleich zur einer analog ausgestatteten `Bruch`-Klasse



bei der Bruch-Struktur nur wenige Abweichungen:



Neben der alternativen Symbolfarbe für Strukturen stellen wir fest:

- Bei der Struktur ist als Basisklasse **System.ValueType** angegeben, was gleich in Abschnitt 5.1.3 näher erläutert wird.
- In der Member-Liste taucht der parameterfreie Konstruktor *nicht* auf. Er wird offenbar bei Strukturen anders realisiert als bei Klassen.

Weil bei der Deklaration von Strukturfeldern keine Initialisierung erlaubt ist, und außerdem der parameterfreie Konstruktor nicht verändert werden darf, besteht bei parameterfrei konstruierten `Bruch`-Strukturinstanzen ein gravierendes Problem: Im Nenner kann der Wert Null nicht verhindert werden (siehe obige Ausgabe). Generell muss eine Struktur so entworfen werden, dass die Null-Initialisierung durch den Standardkonstruktor zu einer regulären Instanz führt. Bei kritischen Typen (wie z.B. `Bruch`) kann die Regularität trotz der nicht zu verhindernden Null-Initialisierung über Datenkapselung und geeignetes Design von Methoden und Eigenschaften hergestellt werden. Statt Aufwand in die `Bruch`-Struktur zu invertieren, bleiben wir jedoch bei der Realisation durch eine

Klasse. Damit ersparen wir uns Probleme und behalten die früher oder später relevante Vererbungsoption.

5.1.3 Strukturen im Common Type System 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 reservierten Wörter sind lediglich Aliasnamen für vordefinierte Strukturen aus dem FCL-Namensraum **System**:

Aliasname	Struktur
sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32

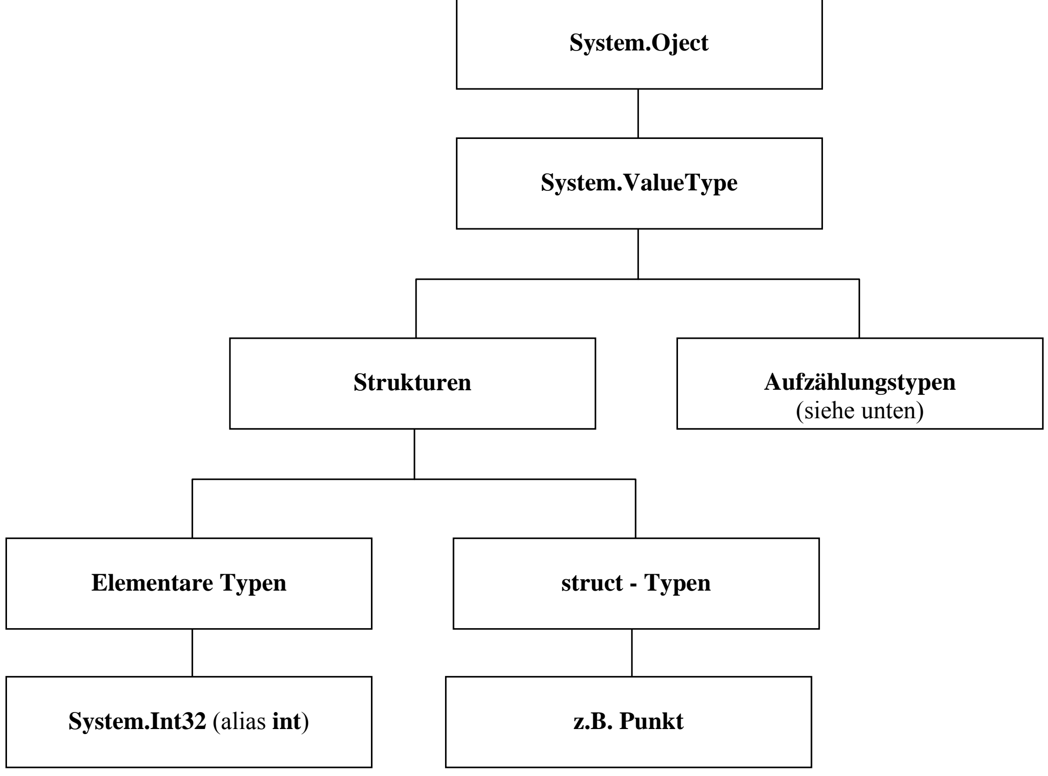
Aliasname	Struktur
long	System.Int64
ulong	System.UInt64
char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

Nun wird z.B. klar, warum die **Convert**-Methode zur Wandlung von Zeichenfolgen in **int**-Werte den Namen **ToInt32**() trägt.

Im Vergleich zu sonstigen Strukturtypen unterstützt C# bei den elementaren Datentypen einige zusätzliche Operationen (vgl. ECMA 2006, S. 110), z.B.:

- Erzeugung von Werten über Literale
- Deklaration konstanter Variablen über das Schlüsselwort **const**

Die folgende Abbildung zeigt, in welcher Beziehung die verschiedenen Werttypen der .NET – Plattform zueinander stehen, und wie sich diese Typen in das streng hierarchisch organisierte **Common Type System** (CTS) mit der Urahnklasse **Object** einfügen:



Im folgenden Beispielprogramm wird beim Ganzzahlliteral 13 (vom Strukturtyp **Int32**) über die von **System.Object** geerbte Methode **GetType()** erfolgreich der 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 der gleich zu beschreibender Raffinesse der .NET - Plattform kann eine Strukturinstanz jederzeit so behandelt werden, als *wäre* sie ein Objekt.

5.2 Boxing und Unboxing

Wie bald im Abschnitt 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 syntaktisch (gemäß Methodendefinition) und damit auch technisch 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** in Positionen mit Formalparametertyp **Object** übergeben.

Diese Zuweisungskompatibilität wird möglich durch ein als **Boxing** bezeichnetes Prinzip: Es sorgt dafür, dass ein Wert bei Bedarf automatisch in ein Objekt einer passenden Hilfsklasse verpackt wird. Somit existiert ein Heap-Objekt mit den Handlungskompetenzen der Klasse **Object**, und der betroffenen Programmeinheit (z.B. der **Console**-Methode **WriteLine()**) kann die benötigte Adresse übergeben werden. Zur Erläuterung der Boxing-Technik bedienen wir uns in Anlehnung an ECMA (2006, S. 114f) einer nicht ganz korrekten, aber hilfreichen Vorstellung: Zum Werttyp `T` nehmen wir die Existenz der folgenden Boxing-Klasse `TBox` an:

```
class TBox {
  public T Wert;
  public TBox(T t) {
    Wert = t;
  }
}
```

Beim automatischen Verpacken eines Werts `w` vom Typ `T` wird implizit über den Ausdruck

```
new TBox(w);
```

ein `TBox`-Objekt auf dem Heap erzeugt. Es nimmt eine *Kopie* des Wertes auf und besitzt alle **Object**-Handlungskompetenzen, kann z.B. die Methode **GetType**() ausführen.

Die Erweiterung bzw. Spezialisierung der Klasse **Object** durch einen Werttyp besteht darin, dass ein per Boxing entstehendes Objekt über die geerbten Merkmale und Handlungskompetenzen hinaus auch noch einen Wert speichern kann.

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:

```
using System;
class Prog {
   static void Main() {
      int i = 4711;
      object iBox = i;
      int j = (int) iBox;
   }
}
```

Dass die Boxing-Technik nicht nur akademische Trockenübungen ermöglicht, werden Sie z.B. im Zusammenhang mit Arrays und anderen Containern erfahren. Dort kann man durch Verwendung des Basistyps **Object** generische (typunabhängige) Behälter schaffen, die auch Daten mit Werttyp aufnehmen können. Die erforderlichen Anpassungs- bzw. Verpackungsmaßnahmen laufen automatisch ab.

Wie das letzte Beispiel zeigt, benötigt der Compiler beim **Unboxing**, also beim Auspacken eines Wertes, eine explizite Casting-Operation mit Angabe des Zieltyps:

```
int j = (int) iBox;
```

Neben dem bisher beschriebenen *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;
```

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}", 13);
```

tatsächlich eine Boxing-Operation stattfindet, kann mit dem Windows-SDK - Hilfsprogramm **IL-Dasm** einen Blick auf den MSIL-Code werfen. Hier wird die Objektkreation zur Verpackung eines **System.Int32** - Werts mit dem OpCode **box** veranlasst:

```
Prog::method Main : void()
Suchen   Weitersuchen
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       20 (0x14)
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ldstr      "i={0}"
  IL_0006:  ldc.i4.s   13
  IL_0008:  box        [mscorlib]System.Int32
  IL_000d:  call       void [mscorlib]System.Console::WriteLine(string,
                                                                 object)
  IL_0012:  nop
  IL_0013:  ret
} // end of method Prog::Main
```

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());
```

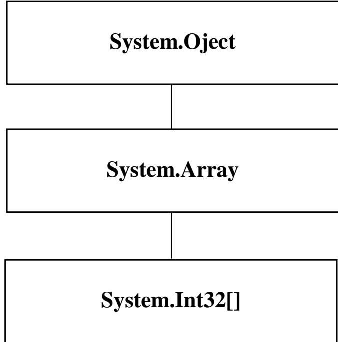
wird per **box**-OpCode ein Objekt erzeugt.

5.3 Arrays

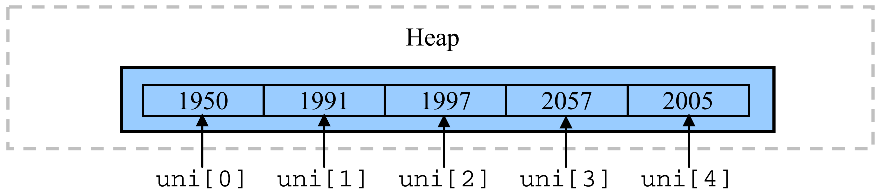
Ein Array ist ein Objekt, das als Instanzvariablen eine feste Anzahl von Elementen desselben Datentyps enthält. Man kann den kompletten Array ansprechen (z.B. als Aktualparameter an eine Methode übergeben), oder auf einzelne Elemente über einen Index zugreifen.

Arrays werden in vielen Programmiersprachen auch *Felder* genannt. In C# bezeichnet man jedoch recht einheitlich die (Instanz-)variablen einer Klasse oder Struktur als *Felder*, so dass 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 werden 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**):



Hier ist als konkretes Objekt aus dieser Klasse ein Array namens `uni` mit fünf **int**-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 Arraynamen den durch eckige Klammern begrenzten Index an, wobei die Nummerierung bei 0 beginnt und bei *n* Elementen folglich mit *n* - 1 endet.

Im Vergleich zur Verwendung einer entsprechenden Anzahl von Einzelvariablen ergibt sich eine erhebliche Vereinfachung der Programmierung:

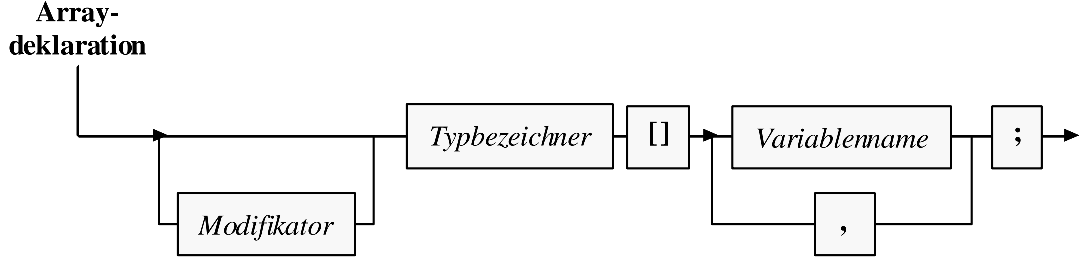
- Weil der Index auch durch einen *Ausdruck* (z.B. durch eine Variable) geliefert werden kann, sind Arrays im Zusammenhang mit den Wiederholungsanweisungen äußerst praktisch.
- Man kann oft die *gemeinsame* Verarbeitung aller Elemente (z.B. bei der Ausgabe in eine Datei) per Methodenaufruf mit Array-Aktualparameter 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.

5.3.1 Array-Referenzvariablen deklarieren

Eine Array-Variable ist vom Referenztyp und wird folgendermaßen deklariert:



Im Vergleich zu der bisher bekannten Variablendeklaration (ohne Initialisierung) ist hinter dem Typbezeichner zusätzlich ein Paar eckiger Klammern anzugeben. In obigem Beispiel kann die Array-Variable `uni` also z.B. folgendermaßen deklariert werden:

```
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** zugewiesen werden.

5.3.2 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`:

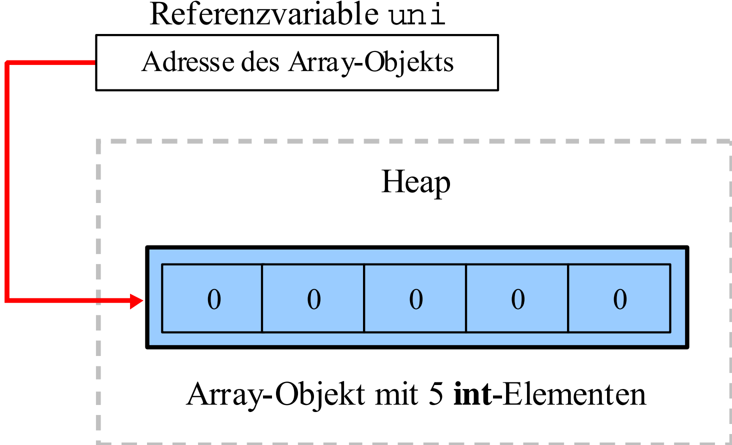
```
uni = new int[max+1];
```

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 kann man 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 folgende Situation:



Weil es sich bei den Array-Elementen um Instanzvariablen eines *Objekts* handelt, erfolgt eine automatische Initialisierung nach den Regeln von Abschnitt 4.2.3. Die **int**-Elemente im Beispiel erhalten folglich den Startwert 0.

Aus der Objekt-Natur eines Arrays folgt unmittelbar, dass er vom Garbage Collector entsorgt wird, wenn keine Referenz mehr vorliegt (vgl. Abschnitt 4.4.4). Um eine Referenzvariable aktiv von einem Array-Objekt zu „entkoppeln", kann man ihr z.B. den Wert **null** (Zeiger auf nichts) 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

Seit der .NET - Version 2.0 erlaubt die statische Methode **Resize**() der Klasse **System.Array** sogar 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 auch bei den in Abschnitt 5.3.8 beschriebenen Kollektionsklassen mit dynamischer Größenanpassung (z.B. **ArrayList**) bei Bedarf im Hintergrund automatisch ausgeführt.

5.3.3 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 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, auf den die Referenzvariable `uni` gemäß obiger Deklaration und Initialisierung zeigt:

```
for (i = 0; i < DRL; i++)
  uni[zzg.Next(5)]++;
```

Den Indexwert liefert die **Random**-Methode **Next()** mit Rückgabetyp **int** (siehe unten).

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

```
int[] uni = new int[5];
```

kein Element `uni[5]`. Ein Zugriffsversuch führt zum Laufzeitfehler vom Typ **IndexOutOfRange-Exception**, 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 Abschnitt 10), wird es vom Laufzeitsystem beendet. Man kann sich in C# generell darauf verlassen, dass jede Überschreitung von Feldgrenzen verhindert wird, so dass 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 Eigenschaft **Length** jederzeit feststellen, z.B.:

<table>
<tr><th>Quellcode</th><th>Eingaben (fett) und Ausgaben</th></tr>
<tr><td>

```
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]);
  }
}
```

</td><td>

```
Länge des Vektors: 3

Wert von Element 0: 7
Wert von Element 1: 13
Wert von Element 2: 4711

7
13
4711
```

</td></tr>
</table>

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.

5.3.4 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 von ihrem Startwert abhängt, kann sie in der Regel echte Zufallszahlen ersetzen. Manchmal ist es sogar von Vorteil, eine Serie über ihren *festen* Startwert reproduzieren zu können. Meist verwendet man aber *variable* Startwerte, z.B. abgeleitet aus einer Zeitangabe. Der Einfachheit halber redet man oft von *Zufallszahlen* und lässt den *Pseudo*-Zusatz weg.

Man kann übrigens mit moderner EDV-Technik unter Verwendung von physikalischen 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 Verwendung des parameterfreien **Random**-Konstruktors entscheidet man sich für einen aus der Systemzeit abgeleiteten Startwert.

Das angekündigte Programm zieht 10000 Zufallszahlen aus der Menge {0, 1, ..., 4} und überprüft die empirische Verteilung dieser Stichprobe:

```
using System;
class UniRand {
   static void Main() {
      const int DRL = 10000;
      int i;
      int[] uni = new int[5];
      Random zzg = new Random();
      for (i = 0; i < DRL; i++)
         uni[zzg.Next(5)]++;
      Console.WriteLine("Absolute Häufigkeiten:");
      for (i = 0; i < 5; i++)
         Console.Write(uni[i] + " ");
      Console.WriteLine("\n\nRelative Häufigkeiten:");
      for (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

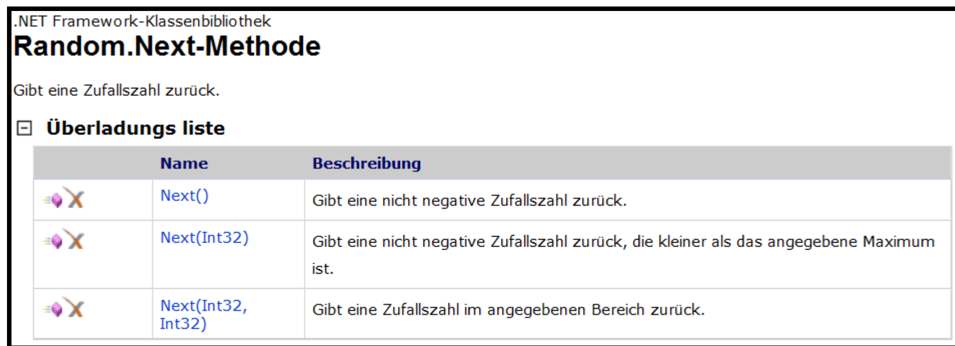
	uni
Chi-Quadrat	1,108[a]
df	4
Asymptotische Signifikanz	**,893**

a. Bei 0 Zellen (,0%) werden weniger als 5 Häufigkeiten erwartet. Die kleinste erwartete Zellenhäufigkeit ist 2000,0.

Über die im Beispielprogramm verwendete Klasse **Random** und deren **Next()**-Methode liefert die FCL-Dokumentation ausführliche Informationen, die von Visual C# 2010 aus z.B. so zu erreichen sind:

- Methodennamen markieren
- Funktionstaste **F1** drücken

Es erscheint ein HTML-Dokument, das die Überladungen der Methode auflistet:



.NET Framework-Klassenbibliothek

Random.Next-Methode

Gibt eine Zufallszahl zurück.

Überladungs liste

	Name	Beschreibung
	Next()	Gibt eine nicht negative Zufallszahl zurück.
	Next(Int32)	Gibt eine nicht negative Zufallszahl zurück, die kleiner als das angegebene Maximum ist.
	Next(Int32, Int32)	Gibt eine Zufallszahl im angegebenen Bereich zurück.

Nach einem Mausklick auf die passende (mittlere) Überladung werden Syntax und Semantik erklärt:

.NET Framework-Klassenbibliothek

Random.Next-Methode (Int32)

Gibt eine nicht negative Zufallszahl zurück, die kleiner als das angegebene Maximum ist.

Namespace: System
Assembly: mscorlib (in mscorlib.dll)

Syntax

C#

```
public virtual int Next(
    int maxValue
)
```

Parameter

maxValue

Typ: System.Int32

Die exklusive obere Grenze der Zufallszahl, die generiert werden soll.*maxValue* muss größer oder gleich 0 (null) sein.

Rückgabewert

Typ: System.Int32

Eine 32-Bit-Ganzzahl mit Vorzeichen, die größer oder gleich 0 (null) und kleiner als *maxValue* ist, d. h., der Bereich der Rückgabewerte umfasst gewöhnlich 0, aber nicht *maxValue*.Wenn jedoch *maxValue* 0 (null) entspricht, wird *maxValue* zurückgegeben.

Eine weitere Anleitung zur Nutzung der sehr guten FCL-Dokumentation ist in diesem Manuskript sicher nicht mehr erforderlich.

5.3.5 Initialisierungslisten

Bei Arrays mit wenigen Elementen ist die Möglichkeit von Interesse, beim Deklarieren der Referenzvariablen eine Initialisierungsliste anzugeben und das Array-Objekt dabei implizit (ohne Verwendung des **new**-Operators) zu erzeugen, z.B.:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>using System;
class Prog {
  static void Main() {
    int[] wecktor = {1, 2, 3};
    Console.WriteLine(wecktor[2]);
  }
}</pre></td><td><pre>3</pre></td></tr>
</table>

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 späteren Objektkreation, z.B.:

```
int[] wecktor;
wecktor = new int[] {1, 2, 3};
```

Die eben (bei der Array-Deklaration mit Initialisierung) noch gültige Schreibweise mit impliziter Objektkreation

```
wecktor = {1, 2, 3}; // Nicht erlaubt!
```

akzeptiert der Compiler allerdings jetzt *nicht* mehr.

5.3.6 Objekte als Array-Elemente

Für die Elemente eines Arrays sind natürlich auch Referenztypen erlaubt. In folgendem Beispiel wird ein Array mit `Bruch`-Objekten erzeugt:

Quellcode:

```
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();
  }
}
```

Ausgabe:

```
        5
 b2 = -----
        6
```

5.3.7 Mehrdimensionale Arrays

5.3.7.1 Rechteckige Arrays

In der linearen Algebra und in vielen anderen Anwendungsbereichen werden auch *mehrdimensionale* Arrays benötigt, z.B.:

Quellcode:

```
using System;
class Prog {
  static void Main() {
    int[,] matrix = new int[4, 3];
    int nrow = matrix.GetLength(0);
    int ncol = matrix.GetLength(1);
    Console.WriteLine("{0} Dimensionen,\n{1} Zeilen, {2} Spalten",
                      matrix.Rank, nrow, ncol);
    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();
    }
  }
}
```

Ausgabe:

```
2 Dimensionen,
4 Zeilen, 3 Spalten
  1  2  3
  2  4  6
  3  6  9
  4  8 12
```

Im Beispiel wird eine *zweidimensionale* Matrix mit 4 Zeilen und 3 Spalten erzeugt, auf deren Zellen man per Doppelindizierung zugreifen kann. Bei der Erzeugung bzw. Verwendung eines mehrdimensionalen rechteckigen Arrays werden die in eckigen Klammern eingeschlossenen Dimensionsangaben bzw. Indexwerte durch Komma getrennt.

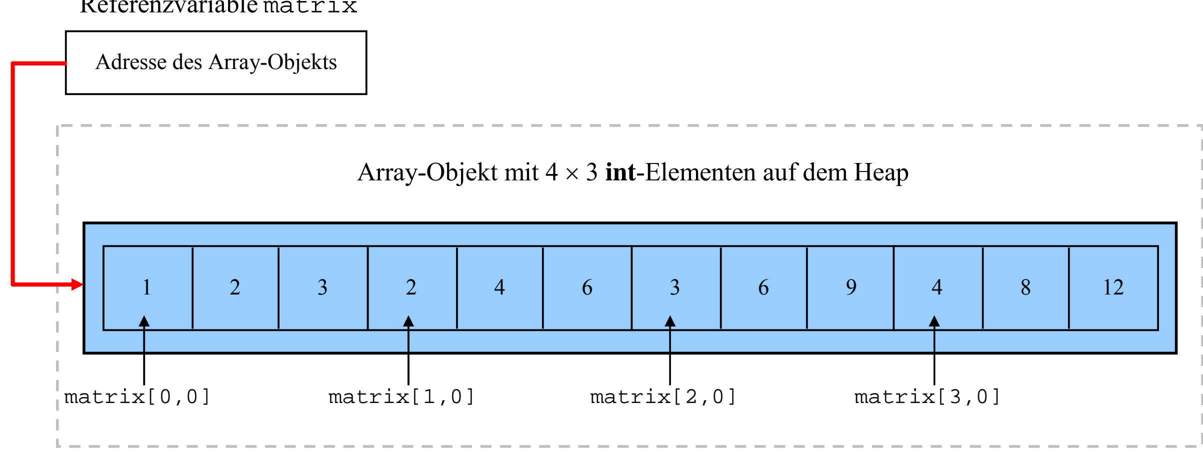
Auch im mehrdimensionalen Fall können Initialisierungslisten eingesetzt werden, z.B.:

```
int[,] matrix = {{1, 4, 2}, {9, 5, 6}, {1, 7, 7}};
```

Weil alle Arrays von der Basisklasse **Array** im Namensraum **System** abstammen, verfügen sie über entsprechende Methoden und Eigenschaften (siehe FCL-Dokumentation), z.B.:

- Die Eigenschaft **Rank** enthält die Anzahl der Dimensionen.
- Über die Methode **GetLength()** informiert ein Array-Objekt über die Anzahl der Elemente in der per Parameter angegebenen Dimension.

Bei den bisher behandelten *rechteckigen* Arrays liegen im Speicher alle Elemente unmittelbar hintereinander, was einen schnellen Indexzugriff ermöglicht:



Wie das Speicherabbild zeigt, ist bei regelmäßigen Arrays jeder beliebige Rang möglich, wobei der vertraute Ausdruck *rechteckig* streng genommen nur bei zweidimensionalen Arrays passt.

5.3.7.2 Mehrdimensionale Arrays mit unterschiedlich großen Elementen

Neben den rechteckigen Arrays unterstützt C# auch „ausgesägte" Exemplare mit unterschiedlich großen Elementen (engl.: *jagged arrays*). So lässt sich etwa eine zweidimensionale Matrix mit unterschiedlich langen Zeilen realisieren:

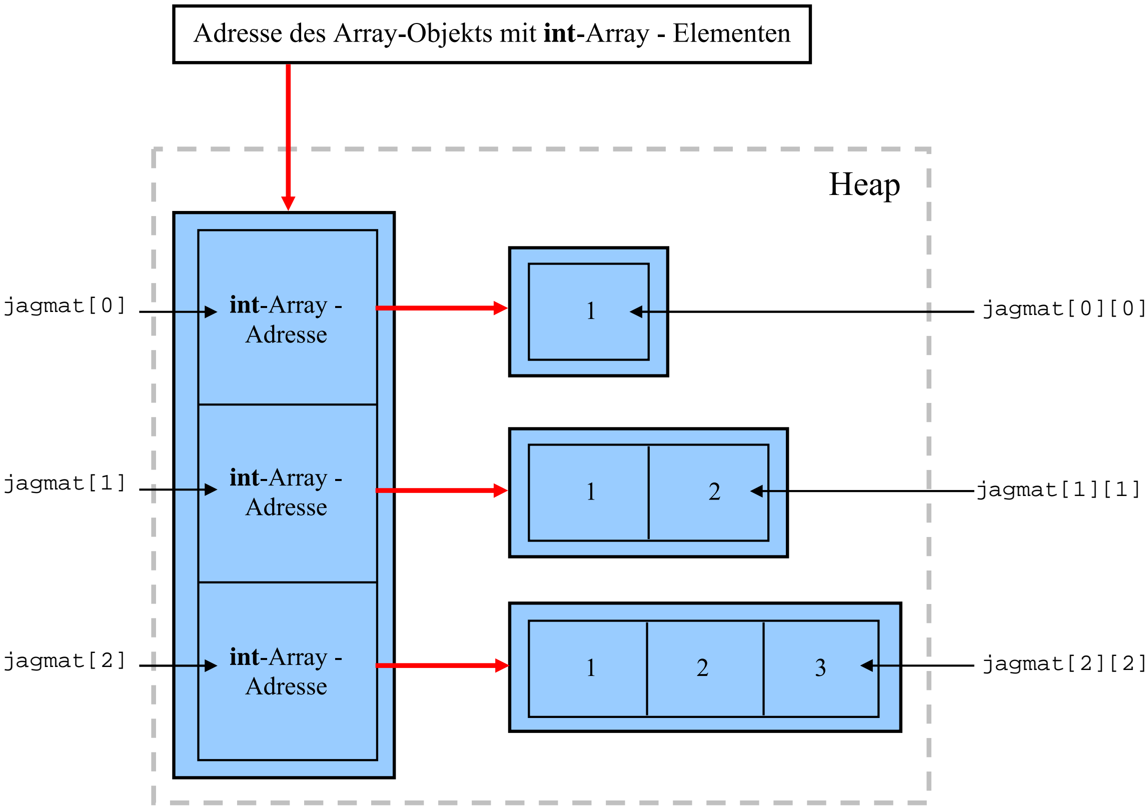
Quellcode

```
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();
    }
  }
}
```

Ausgabe

```
1
1 2
1 2 3
```

Im Unterschied zum **int[,]** - Objekt `matrix` aus dem Beispiel in Abschnitt 5.3.7.1, das doppelt indizierte **int**-Elemente enthält, handelt es sich bei den Elementen des **int[][]** - Objekts `jagmat` um einfach indizierte 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 *erste* Dimension angegeben werden:

```
int[][] jagmat = new int[3][];
```

Anschließend erzeugt man die Zeilenobjekte und legt ihre Adressen in den `jagmat`-Elementvariablen ab, z.B.:

```
jagmat[2] = new int[3] {1, 2, 3};
```

5.3.8 Die Kollektionsklasse ArrayList

Im Namensraum **System.Collections** bietet die FCL etliche Klassen zur flexiblen Verwaltung von Datenbeständen *variablen* Umfangs. 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.:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
using System;
using System.Collections;
class ArrayListDemo {
   static void Main() {
      ArrayList al = new ArrayList();
      string s;
      Console.WriteLine("Was fällt Ihnen zu C#?\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]);
   }
}
```

</td><td>

```
Was fällt Ihnen zu C#?

: Tolle Sache
: Nicht ganz trivial
: Macht Spaß
:

Ihre Anmerkungen:
Tolle Sache
Nicht ganz trivial
Macht Spaß
```

</td></tr>
</table>

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 5.6), kann man per Indexsyntax auf die Elemente zugreifen:

- Das erste Element hat den Indexwert Null.
- 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 *Common Type Systems* der .NET – Plattform. Folglich kann ein **ArrayList**-Container Daten beliebigen Typs aufnehmen, wobei dank Boxing-Technik (siehe Abschnitt 5.2) auch die Werttypen erlaubt sind, z.B.:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
using System;
using System.Collections;
class Prog {
   static void Main() {
      ArrayList al = new ArrayList();
      al.Add("Wort");
      al.Add(3.14);
      al.Add(13);
      foreach (object o in al)
         Console.WriteLine(o);
   }
}
```

</td><td>

```
Wort
3,14
13
```

</td></tr>
</table>

Ein solcher „Gemischtwarenladen“ (allerdings mit *fester* Länge) ist durch Wahl des Element-Datentyps **Object** übrigens auch mit einem einfachen Array zu realisieren.

5.4 Klassen für Zeichenketten

C# bietet für den Umgang mit Zeichenketten, die grundsätzlich aus Unicode-Zeichen bestehen, zwei Klassen an:

- **String** (im Namensraum **System**)
  String-Objekte können nach dem Erzeugen nicht mehr geändert werden. Diese Klasse ist für den *lesenden* Zugriff auf Zeichenketten optimiert.
- **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.

5.4.1 Die Klasse String für konstante 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:

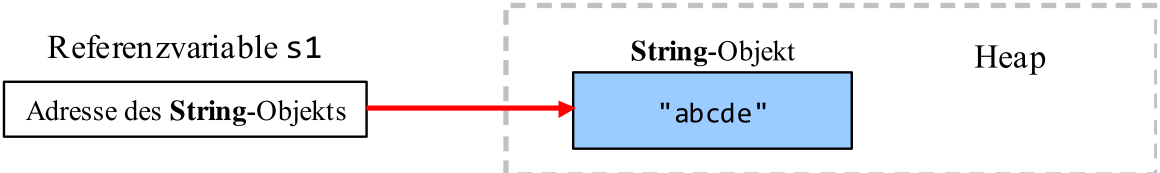
- eine **String**-Referenzvariable namens `s1` angelegt,
- ein neues **String**-Objekt mit dem Inhalt „abcd“ auf dem Heap erzeugt,
- die Adresse des neuen Heap-Objekts in der Referenzvariablen abgelegt.

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, so dass 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 folgende Situation:



5.4.1.1 String als WORM - Klasse

Nachdem ein **String**-Objekt auf dem Heap erzeugt worden ist, kann es nicht mehr geändert werden. In der Überschrift zu diesem Abschnitt wird für diesen Sachverhalt eine Abkürzung aus der Elektronik ausgeliehen: WORM (**W**rite **O**nce **R**ead **M**any). Eventuell werden Sie die Unveränderlichkeit des **String**-Inhalts 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 = testr + "def";`
`    Console.WriteLine("testr = " + testr);`
`  }`
`}`
	`testr = abc`
	`testr = abcdef`

In der Zeile

```
testr = 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.

5.4.1.2 Methoden für String-Objekte

Von den zahlreichen Methoden der Klasse der **String** werden in diesem Abschnitt nur die wichtigsten angesprochen. Für spezielle Anwendungen lohnt sich also ein Blick in die FCL-Dokumentation.

5.4.1.2.1 Verketten von Strings

Weil die Klasse **String** den „+“- Operator geeignet überladen hat (vgl. Abschnitt 4.7.2), taugt er zum Verketten von **String**-Objekten, wobei Operanden beliebiger Datentypen bei Bedarf automatisch in **String**-Objekte konvertiert werden. Wie Sie aus Abschnitt 5.4.1.1 wissen, entsteht beim Verketten von zwei Zeichenfolgen ein *neues* **String**-Objekt. Im folgenden Beispiel 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));`
`  }`
`}`
	`4 + 3 = 7`

5.4.1.2.2 Vergleichen von Strings

Angewandt auf **String**-Variablen vergleichen die Operatoren „==“ und „!=“ *nicht* (wie z.B. in Java) die *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 (siehe Abschnitt 5.4.1.3) zeigen, weil dann Adressen- und Inhaltsvergleich

zum selben Ergebnis kämen. Wie in Abschnitt 5.4.1.3 demonstriert wird, ist bei einer großen Anzahl von **String**-Vergleichen eventuell durch Internalisierung und Verwendung von Adressenvergleichen eine Leistungssteigerung zu erzielen.

Zum Testen auf **lexikographische Priorität** (z.B. beim Sortieren) kann die **String**-Methode **CompareTo()** dienen:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`    String a = "Müller, Anja", b = "Müller, Kurt",`
`           c = "Müller", d = ", Anja";`
`    c = c + d; //a und c sollen nicht auf denselben Pool-String zeigen`
`    Console.WriteLine("< : " + a.CompareTo(b));`
`    Console.WriteLine("= : " + a.CompareTo(c));`
`    Console.WriteLine("> : " + b.CompareTo(a));`
`  }`
`}`
	`< : -1`
	`= : 0`
	`> : 1`

CompareTo() liefert folgende Ergebnisse zurück:

		CompareTo()-Ergebnis
Das befragte **String**-Objekt ist im Vergleich zum Aktualparameter lexikographisch …	kleiner	-1
	gleich	0
	größer	1

5.4.1.2.3 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`

5.4.1.2.4 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* als Kopie.

Mit der Methode

public int IndexOf(string *gesucht*)

kann man einen **String** nach der Existenz einer anderen Zeichenkette befragen. Als Rückgabewert erhält man ...

- nach erfolgreicher Suche: die 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.

5.4.1.2.5 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

<pre>using System;
class Prog {
  static void Main() {
    String a = "Otto", b = "otto";
    Console.WriteLine(a.ToUpper() == b.ToUpper());
    Console.WriteLine(a.ToUpper().IndexOf("T"));
  }
}</pre>
	<pre>True
	1</pre>

In der letzten Anweisung des Beispiels ist der **WriteLine()**-Parameter etwas komplex geraten, so dass vielleicht eine kurze Erklärung angemessen ist:

- Der linke Punktoperator wird zuerst ausgeführt. Dabei erzeugt der Methodenaufruf `a.ToUpper()` ein neues **String**-Objekt und liefert eine zugehörige Referenz.
- Diese Referenz ermöglicht es, dem neuen Objekt Botschaften zu übermitteln, was im Methodenaufruf `IndexOf("T")` geschieht.

5.4.1.3 Interner String-Pool

Wie oben erläutert, bildet ein Zeichenkettenliteral einen Ausdruck vom Typ **String** mit einer Objektadresse als Wert. Für wiederholt im Quellcode auftretende Zeichenkettenliterale lässt dieses

Prinzip eine Verschwendung von Speicherplatz befürchten. Um das zu verhindern, verwaltet die CLR eine als **interner String-Pool** bezeichnete Tabelle für die auf Literalen basierenden **String**-Objekte. Wird zu einem Zeichenkettenliteral eine Objektreferenz benötigt, liefert die CLR nach Möglichkeit die Adresse eines bereits vorhandenen **String**-Objekts. 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:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>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));
   }
}</pre></td><td><pre>True</pre></td></tr>
</table>

Allerdings verhindern die im **String**-Pool vorhandenen Referenzen eine Entsorgung der zugehörigen Objekte durch den Garbage Collector, so dass der gewünschte Speichereinspareffekt *nicht* unbedingt erzielt wird. Seit der .NET - Version 2.0 kann daher die Internalisierung der auf Literalen basierenden **String**-Objekte durch ein Assembly-Attribut (siehe Abschnitt 11.4) abgeschaltet werden. Weil sich bei zukünftigen .NET - Versionen das Internalisierungs-Verhalten der CLR generell eventuell ändert, darf die Korrektheit eines Quellcodes keinesfalls von der aktuellen Praxis abhängen (Richter 2006, S. 274ff).

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 Parameter-String, wird die Adresse dieses Pool-Objekts geliefert.
- Anderenfalls erzeugt **Intern()** ein Objekt mit dem Parameter-String, nimmt es in den internen **String**-Pool auf und liefert seine Adresse als Rückgabe.

Durch Internalisieren kann Speicherplatz gespart werden, wenn viele Strings mit identischem Inhalt zu erwarten sind. Außerdem können **String**-Vergleiche (vgl. Abschnitt 5.4.1.2.2) beschleunigt werden, 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 `N` mal mit einem zufällig gewählten Partner verglichen. Dies geschieht zunächst per Inhaltsvergleich und dann nach dem zwischenzeitlichen Internieren per Adressvergleich:

```
using System;
using System.Text;

class StringIntern {
   public static void Main() {
      const int ANZ = 50000, LEN = 50, N = 50;
      StringBuilder sb = new StringBuilder();
      Random 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;
```

```
      int hits = 0;
      // N * ANZ Inhaltsvergleiche
      for (int n = 1; n <= N; n++)
         for (int i = 0; i < ANZ; i++)
            if (sar[i] == sar[ran.Next(ANZ)])
               hits++;
      Console.WriteLine((N * ANZ)+" Inhaltsvergleiche ("+hits+
            " hits) benoetigen "+((DateTime.Now.Ticks - start)/1.0e4)+
            " Millisekunden");

      start = DateTime.Now.Ticks;
      hits = 0;
      // Internieren
      for (int j = 1; j < ANZ; j++)
         sar[j] = String.Intern(sar[j]);
      Console.WriteLine("Zeit für das Internieren: " +
            ((DateTime.Now.Ticks - start) / 1.0e4) + " Millisekunden");
      // N * ANZ Adressvergleiche
      for (int n = 1; n <= N; n++)
         for (int i = 0; i < ANZ; i++)
            if (((Object)sar[i]) == sar[ran.Next(ANZ)])
               hits++;
      Console.WriteLine((N * ANZ)+" Adressvergleiche ("+hits+
            " hits) benoetigen (inkl. Internieren) "+
            ((DateTime.Now.Ticks - start)/1.0e4)+" Millisekunden");
      Console.ReadLine();
   }
}
```

Beim Erzeugen der Zufallszeichenfolgen kommt ein Objekt der Klasse **StringBuilder** zum Einsatz (siehe Abschnitt 5.4.2).

Um den Identitätsoperator zu Adressvergleichen 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 `N` ab, welche Vergleichsmethode überlegen ist:[1]

	Laufzeit in Millisekunden	
	Inhaltsvergleiche	Internieren u. Adressvergl.
ANZ = 50000, LEN = 50, N = 5	15,625	46,875
ANZ = 50000, LEN = 50, N = 50	265,625	93,75

Erwartungsgemäß ist das Internieren umso rentabeler, 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 **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.

5.4.2 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.

[1] Die Ergebnisse stammen von einem PC mit Intel - CPU Core i3 550 (3,2 GHz).

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 10000-mal verlängert, zunächst mit Hilfe der **String**-Klasse, dann mit Hilfe der **StringBuilder**-Klasse:

```
using System;
using System.Text;
class Prog {
  static void Main() {
    const int N = 10000;
    String s = "*";
    long vorher = DateTime.Now.Ticks;
    for (int i = 0; i < N; i++)
      s = s + "*";
    long diff = DateTime.Now.Ticks - vorher;
    Console.WriteLine("Zeit für String-Manipulation:\t\t"+diff/1.0e4);

    StringBuilder 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);
  }
}
```

Die (in Millisekunden gemessenen) Laufzeiten unterscheiden sich erheblich:[1]

```
Zeit für String-Manipulation:          31,25
Zeit für StringBuilder-Manipulation:   0
```

Ein **StringBuilder**–Objekt kennt u.a. die folgenden Methoden und Eigenschaften (alle **public**):

StringBuilder-Member	Erläuterung
Length	enthält die Anzahl der Zeichen
Append()	Das **StringBuilder**–Objekt wird um die Stringrepräsentation des Argumentes verlängert, z.B.: `t.Append("*");` Es sind **Append**()-Überladungen für zahlreiche Datentypen vorhanden.
Insert()	Die Stringrepräsentation des Argumentes, das von nahezu beliebigem Typ sein kann, wird vom angesprochenen **StringBuilder**–Objekt an einer bestimmten Stelle eingefügt, z.B.: `sb.Insert(4, 3.14);`
Remove()	Ab einer Startposition wird eine Anzahl von Zeichen entfernt, z.B.: `sb.Remove(100, 500);`
Replace()	Ein Zeichen bzw. eine Zeichenfolge des **StringBuilder**–Objekts wird durch anderes Zeichen bzw. eine andere Zeichenfolge ersetzt, z.B.: `sb.Replace("alt", "neu");`

[1] Gemessen auf einem Rechner mit der Intel – CPU Core i3 550 (3,2 GHz).

StringBuilder-Member	Erläuterung
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());`

5.5 Enumerationen

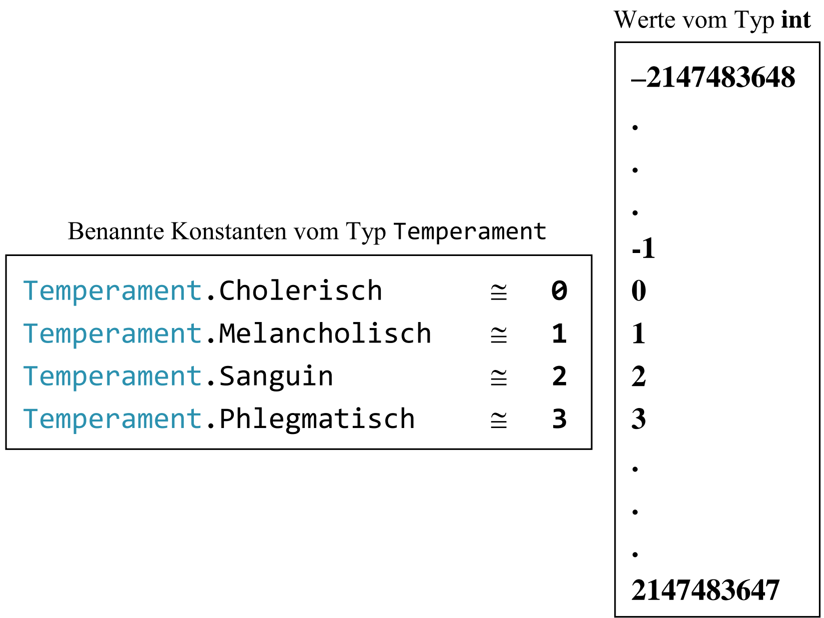
Angenommen, Sie entwerfen eine Klasse namens `Person` und wollen auch den *Charakter* einer Person erfassen. Dabei orientieren sie sich an den vier Temperamentstypen des griechischen Philosophen Hippokrates (ca. 460 - 370 v. Chr.): cholerisch, melancholisch, sanguin, phlegmatisch. Um dieses Merkmal mit 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.:
  ```
  if (otto.Temp == "Flegmatisch") ...
  ```
- Eine **int**-Variable mit der Kodierungsvorschrift 0 = cholerisch, 1 = melancholisch etc.
  Es wird wenig Speicher benötigt, allerdings ist der Quellcode nur für Eingeweihte zu verstehen, und es können leicht fehlerhafte Zuweisungen auftreten, z.B.:
  ```
  if (otto.Temp == 3) ...
  otto.Temp == 13;
  ```

C# bietet mit den **Enumerationen** (**Aufzählungstypen**) eine Lösung, die folgende Vorteile bietet:

- Gut lesbarer Quellcode durch Klartextnamen für die Merkmalsausprägungen
- Falsch geschriebene Klartextnamen werden vom Compiler abgewiesen.
- Geringer Speicherbedarf

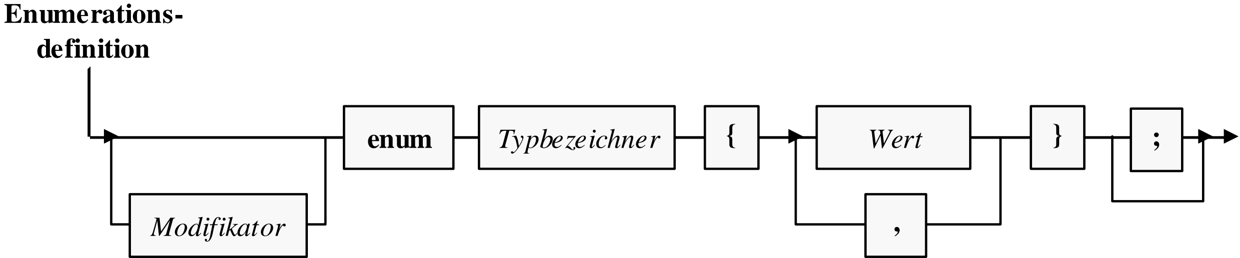
Eine Enumeration basiert auf einem zugrunde liegenden integralen Typ (meist **int**) und enthält eine (meist kleine) Menge von benannten Konstanten dieses Typs, z.B.:



Sofern man auf eine explizite Typkonvertierung 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.Sanguin`).

Bei der Definition eines Aufzählungstyps folgt auf das Schlüsselwort **enum** und den Typbezeichner eine geschweift eingeklammerte Liste mit Namen für die Konstanten:



Wie bei Klassen und Strukturen …

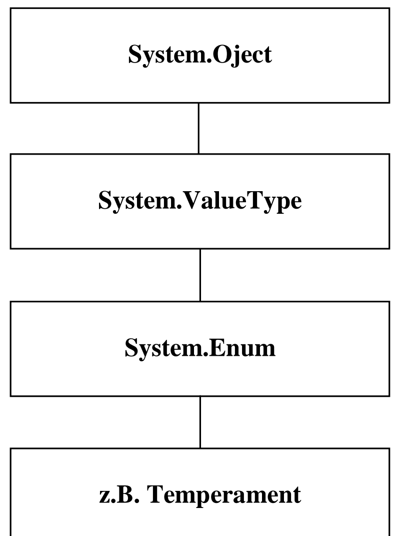
- wird die Verfügbarkeit eines Enumerationstyps über Modifikatoren geregelt (Voreinstellung: **internal**, Alternative: **public**, vgl. Abschnitt 4.9),
- kann optional hinter der schließenden Klammer der Enumerationsdefinition ein Semikolon stehen.

Als Beispiel betrachten wir einen Enumerationstyp zur Erfassung des Charakters von Personen:

```
enum Temperament {Cholerisch, Melancholisch, Sanguin, Phlegmatisch}
```

Per Voreinstellung verwenden Enumerationen den Basistyp **int**, belegen also pro Instanz 4 Byte Speicherplatz, doch kann in der Definition auch ein alternativer Basistyp angegeben werden (siehe FCL-Dokumentation).

Die Enumerationen sind **Werttypen** und folgendermaßen in das CTS (Common Type System) der .NET – Plattform eingeordnet:



Alternative Abstammungen sind bei Enumerationstypen *nicht* möglich; insbesondere kann man eine Enumeration nicht beerben. Die Basisklasse **System.Enum**, die übrigens selbst *keine* Enumeration ist, darf nicht mit dem zugrunde liegenden Typ (meist **int**) verwechselt werden.

Weil Enumerationskonstanten stets mit dem Typnamen qualifiziert werden müssen, ist einige Tipparbeit erforderlich, die aber mit einem gut lesbaren Quellcode belohnt wird, z.B.

```
using System;
enum Temperament {Cholerisch, Melancholisch, Sanguin, Phlegmatisch}
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;
  }
  static void Main() {
    Person otto = new Person("Otto", "Hummer", 35, Temperament.Sanguin);
    if (otto.Temp == Temperament.Sanguin)
        Console.WriteLine("Lustiger Typ!");
  }
}
```

Einer Variable mit 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 werden.

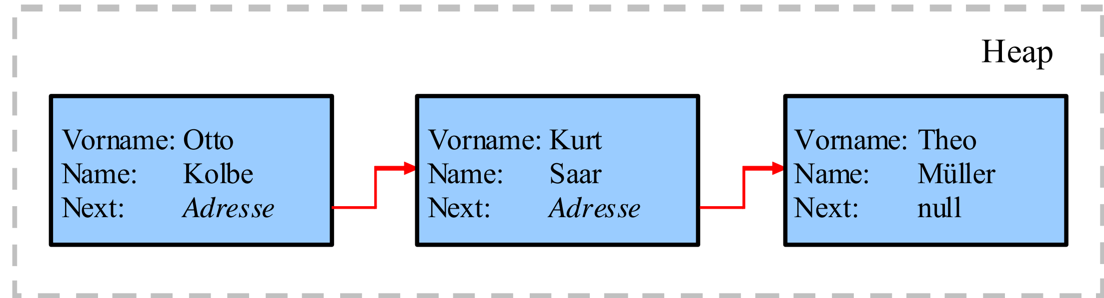
5.6 Indexzugriff bei eigenen Typen zur Verwaltung von Elementen

Bei Arrays sowie bei den Klassen **String** und **ArrayList** hat sich der Indexzugriff auf die Elemente eines Objekts per [ ] - Operator als sehr nützlich bis unverzichtbar erwiesen. Um denselben Komfort für eine eigene Klasse oder Struktur zu realisieren, die zur Verwaltung von zahlreichen Elementen desselben Typs dient, muss man ihr einen so genannten **Indexer** spendieren. Analog zur Situation bei einer Eigenschaft handelt es sich auch bei diesem Member letztlich um ein *Paar von Methoden*. Im aktuellen Abschnitt geht es also *nicht* um einen neuen Datentyp, sondern um ein zusätzliches Ausstattungsdetail für Klassen und Strukturen.

Als Beispiel betrachten wir eine „Datenbank“, die mit einer so genannten *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 beharrliches 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; // Nachfolger des Neulings
          first = value; // Der Neuling wird "Leader"
        } else {
          Person pre = first;
          for (int j = 0; j < i - 1; j++)
            pre = pre.Next;
          value.Next = pre.Next.Next; // Nachfolger des Neulings
          pre.Next = value; // Vorgänger des Neulings
        }
      }
    }
  }
}
```

Für den lesenden oder schreibenden Zugriff auf das *i*-te Listenelement stellt PersonDB einen Indexer zur Verfügung, der eine Person-Referenz liefert (**get**) oder das *i*-te Listenelement durch eine andere Person-Instanz ersetzt (**set**). [1]

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 angegeben.
- Der **set**-Methode wird wie bei Eigenschaften ein impliziter Parameter namens **value** übergeben.
- Indexer können wie Methoden überladen werden, z.B. durch Wahl verschiedener Typen für den Indexparameter.

Vom nicht ganz trivialen PersonDB-Aufbau merkt ein Anwender dieser Klasse nichts, z.B.:

```
using System;
class PersonDbDemo {
  static void Main() {
    PersonDB adb = new PersonDB();
    adb.Add(new Person("Otto", "Kolbe"));
    adb.Add(new Person("Kurt", "Saar"));
    adb.Add(new Person("Theo", "Müller"));
    for (int i = 0; i < adb.Count; i++)
      Console.WriteLine("Nummer {0}: {1} {2}",i,adb[i].Vorname,adb[i].Name);
    Console.WriteLine();
    adb[1] = new Person("Ilse", "Golter");
    for (int i = 0; i < adb.Count; i++)
      Console.WriteLine("Nummer {0}: {1} {2}",i,adb[i].Vorname,adb[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
```

Statt eine eigene Klasse PersonDB mit Listenkonstruktion und Indexer zu entwerfen, hätten wir eine äquivalente „Personenverwaltung" übrigens weit ökonomischer durch Verwendung der in Abschnitt 5.3.8 vorgestellten Kollektionsklasse **ArrayList** realisieren können. In Abschnitt 7.6 sollen Sie im Rahmen einer Übungsaufgabe eine Lösung unter Verwendung der generischen Kollektionsklasse **List**<Person> entwerfen, die zur Verwaltung einer Liste von Elementen *desselben* Typs gegenüber der Klasse **ArrayList** zu bevorzugen ist. Allerdings gehört der Eigenbau einer verketteten Liste zu einer soliden Programmierer-Grundausbildung, so dass sich der Aufwand des PersonDB-Beispiels wohl doch lohnt.

[1] Durch die Aufmerksamkeit und Hilfsbereitschaft von Herrn Reinhard Dämon konnte ein Fehler im **set**-Teil des Indexers behoben werden. Vielen Dank!

5.7 Übungsaufgaben zu Kapitel 5

1) Erstellen Sie eine **struct**-Variante der *Klasse* für zweidimensionale Vektoren, die Sie im Rahmen einer früheren Übungsaufgabe erstellt haben (siehe Abschnitt 4.11).

2) Im folgenden Programm wird den beiden **object**-Variablen `o1` und `o2` derselbe **int**-Wert zugewiesen. Wieso haben die beiden Variablen anschließend nicht denselben Inhalt?

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
using System;
class Prog {
  static void Main() {
    object o1 = 1;
    object o2 = 1;
    Console.WriteLine(o1 == o2);
  }
}
```

</td><td>

```
False
```

</td></tr>
</table>

3) 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.

4) 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 $\{1, 2, 3, ..., K\}$.

- 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 Kandidatenmenge befinden sich nur noch Primzahlen. Um dies einzusehen, nehmen wir an, es gäbe noch eine Zahl $n \leq K$ mit echtem Teiler. Mit zwei positiven Zahlen u, v würde dann gelten:

$$n = u \cdot v \;\text{ und }\; u < b \;\text{ oder }\; v < b \;\text{ (wegen } b > \sqrt{K} \text{ und } n \leq K\text{)}$$

Wir nehmen ohne Beschränkung der Allgemeinheit $u < b$ 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.

Sollen z.B. alle Primzahlen kleiner oder gleich 18 bestimmt werden, so startet man mit folgender Kandidatenmenge:

1	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:

1	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:

1	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:

1, 2, 3, 5, 7, 11, 13 und 17

5) Erstellen Sie eine Klasse für zweidimensionale Matrizen mit Elementen vom Typ **float**. Implementieren Sie eine Methode zum Transponieren einer Matrix und vielleicht noch andere Methoden für elementare Aufgaben der Matrixalgebra.

6) 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 Startwert 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[]**:

```
static void Main(string[] args) {...}
```

- Wie jede andere Methode kann auch **Main**() per **return**-Anweisung spontan beendet werden.

7) 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 die gewünschte Zeilenbreite vorgeben können und auch die Trennzeichen festlegen dürfen, aber nicht müssen (Methoden überladen!).

Weitere Anforderungen an die Methode:

- Das Leerzeichen soll auf jeden Fall trennend wirken.
- Aufeinanderfolgende Trennzeichen sollen wie ein einzelnes Trennzeichen 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 passt nicht in eine Schmal-Zeile, "+
                 "die nur wenige Spalten umfasst.";
      StringUtil.WrapLine(s," -", 40);
      StringUtil.WrapLine(s, 40);
      StringUtil.WrapLine(s);
   }
}
```

Der zweite Methodenaufruf sollte folgende Ausgabe erzeugen:

```
Dieser Satz 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. In folgendem Programm wird die Arbeitsweise demonstriert:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>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);
 }
}</pre></td><td><pre>Dies
ist
der
Beispiel
Satz,
der
zerlegt
werden
soll.</pre></td></tr>
</table>

Die Trennzeichen sind *nicht* in den produzierten Teilzeichenfolgen enthalten, so dass z.B. ein als Trennzeichen definierter Bindestrich verloren geht. Mit dem Enumerationswert

```
StringSplitOptions.RemoveEmptyEntries
```

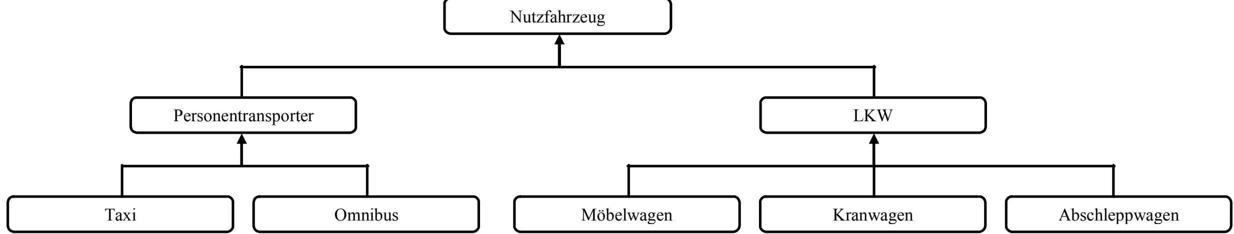
für den zweiten **Split()**-Parameter werden leere Teilzeichenfolgen im resultierenden **String**-Array verhindert.

6 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 Softwareentwicklung im Detail.

Modellierung realer Klassenhierarchien

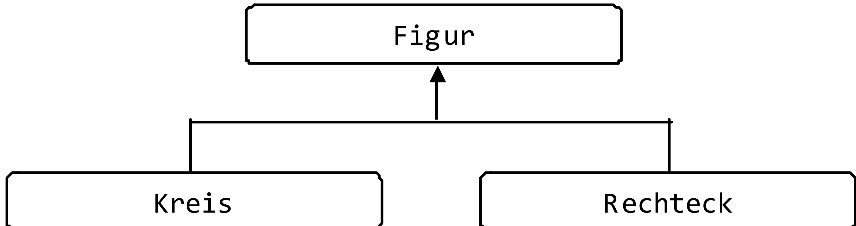
Beim Modellieren eines Gegenstandbereiches durch Klassen, die durch Merkmale (Instanz- und Klassenvariablen) sowie Handlungskompetenzen (Instanz- und Klassenmethoden) gekennzeichnet sind, müssen auch die Spezialisierungs- bzw. Generalisierungsbeziehungen zwischen real existierenden Klassen abgebildet werden. Eine Firma für Transportaufgaben aller Art mag ihre Nutzfahrzeuge folgendermaßen klassifizieren:



Einige Merkmale sind für alle Nutzfahrzeuge relevant (z.B. Anschaffungspreis, momentane Position, maximale Geschwindigkeit), andere betreffen nur spezielle Klassen (z.B. Anzahl der Fahrgäste, maximale Anhängelast, Hebekraft des Krans). Ebenso sind einige Handlungsmöglichkeiten bei allen Nutzfahrzeugen vorhanden (z.B. eigene Position melden), während andere speziellen Fahrzeugen vorbehalten sind (z.B. Fahrgäste befördern, Klaviere transportieren). Ein Programm zur Einsatzplanung und Verwaltung des Fuhrparks sollte diese Klassenhierarchie abbilden.

Übungsbeispiel

Bei unseren Beispielprogrammen bewegen wir uns in einem bescheideneren Rahmen und betrachten meist eine einfache Hierarchie mit Klassen für geometrische Figuren:[1]



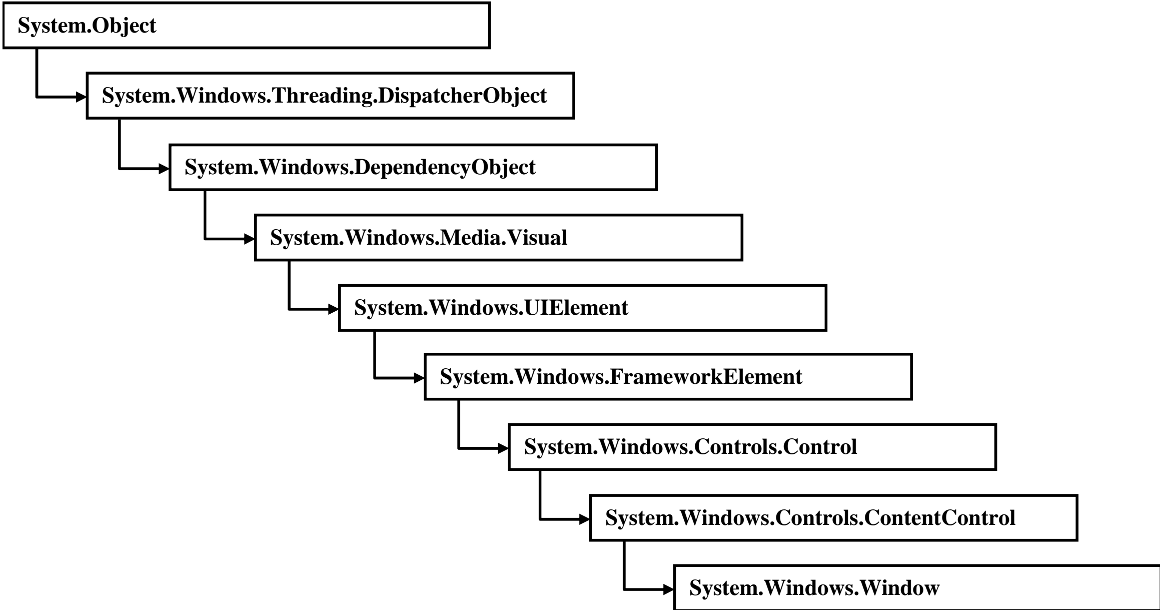
Die Vererbungstechnik der OOP

In objektorientierten Programmiersprachen ist es weder sinnvoll noch erforderlich, jede Klasse einer Hierarchie komplett neu zu definieren. Es steht eine mächtige und zugleich einfach handhabbare **Vererbungstechnik** zur Verfügung: Man geht 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** und kann nach Bedarf Anpassungen bzw. Erweiterungen zur Lösung spezieller Aufgaben vornehmen, z.B.:

[1] Vielleicht haben manche Leser als Gegenstück zum Rechteck (auf derselben Hierarchieebene) die Ellipse erwartet, die ebenfalls zwei ungleiche lange „Hauptachsen“ besitzt. Weiterhin liegt es auf den ersten Blick nahe, den Kreis als Spezialisierung der Ellipse (und das Quadrat als Spezialisierung des Rechtecks) zu betrachten. Wir werden aber in Abschnitt 6.9 über das Liskovsche Substitutionsprinzip genau diese Ableitungen (von Kreis aus Ellipse bzw. von Quadrat aus Rechteck) kritisieren. Daher ist es akzeptabel, an Stelle der Ellipse den Kreis neben das Rechteck zu stellen, um das Erlernen der neuen Konzepte durch ein möglichst einfaches Beispiel zu erleichtern.

- zusätzliche Felder deklarieren
- zusätzliche Methoden oder Eigenschaften definieren
- geerbte Methoden ersetzen, d.h. unter Beibehaltung des Namens umgestalten

Die FCL 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 die Hauptfenster von WPF-Anwendungen:

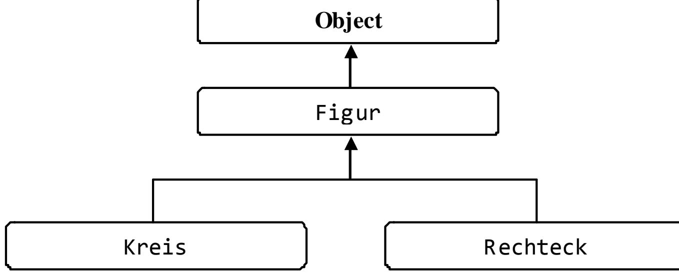


Software-Recycling

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 und dabei doch robuste und wartungsfreundliche Softwaresysteme entstehen. Die verbreitete Praxis, vorhanden Code per *Copy & Paste* in neuen Projekten bzw. Klassen zu verwenden, hat gegenüber einer sorgfältig geplanten Klassenhierarchie offensichtliche Nachteile. Natürlich kann auch C# nicht garantieren, dass jede Klassenhierarchie exzellent entworfen ist und langfristig von einer stetig wachsenden Programmierergemeinde eingesetzt wird.

6.1 Das Common Type System (CTS) des .NET – Frameworks

Im .NET – Framework stammen *alle* Klassen und sonstigen Typen (Strukturen, Enumerationen) von der Klasse **Object** aus dem Namensraum **System** ab. Das gilt sowohl die in der FCL enthaltenen als auch die von uns selbst 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:



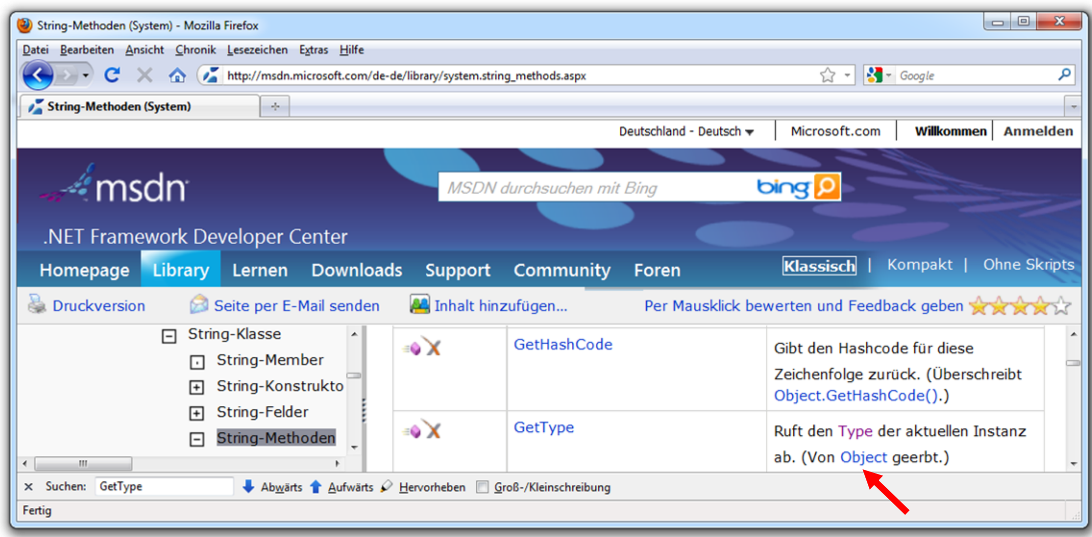
Auch die Strukturen (vgl. Abschnitt 5.1) sind in die globale Hierarchie eingeordnet: Sie stammen implizit von der Klasse **System.ValueType** ab, die wiederum direkt von der Urahnklasse **Object**

erbt. 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** abstammen (vgl. Abschnitt 5.5).

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:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>
using System;
class Prog {
    static void Main() {
        Object o = new Object();
        String s = "abc";
        int i = 13;
        Console.WriteLine(o.GetType() + "\n" +
                          s.GetType() + "\n" +
                          i.GetType());
    }
}
</pre></td><td><pre>
System.Object
System.String
System.Int32
</pre></td></tr>
</table>

In der FCL-Dokumentation zu einem Datentyp sind die Erbstücke gekennzeichnet, z.B. bei der Klasse **String**:



Nach dieser kurzen Beschäftigung mit der ohne eigene Leistungen verfügbaren Standarderbschaft, machen wir uns daran, eigene Vererbungshierarchien aufzubauen.

6.2 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 den Klassennamen durch Doppelpunkt getrennt den Basisklassennamen setzen:

```
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;}
   }
}
```

Die `Kreis`-Klasse erbt die beiden Positionsvariablen sowie die Methode `Wo()` und ergänzt eine zusätzliche Instanzvariable für den Radius samt Eigenschaft mit Lesezugriff für die Öffentlichkeit.

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.

Konstruktoren werden generell *nicht* vererbt, so dass in der `Kreis`-Klasse ein parameterfreier Konstruktor neu definiert wird, obwohl auch die Basisklasse einen solchen Konstruktor besitzt (natürlich mit anderem Namen!).

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 man aber die Mehrfachvererbung wegen einiger Risiken bewusst *nicht* aus C++ übernommen. Einen gewissen Ersatz bietet die in Abschnitt 8 behandelten Schnittstellen (*Interfaces*).

6.3 base-Konstruktoren und Initialisierungs-Sequenzen

Zwar werden Konstruktoren nicht vererbt, doch ist bei der Entstehung einer Instanz eines abgeleiteten Typs ein Konstruktor aus *jeder* Basisklasse entlang der Ahnenreihe 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() {`
`      Figur fig = new Figur(50.0, 50.0);`
`      Console.WriteLine();`
`      Kreis 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 FCL-Designer natürlich keine Kontrollausgabe einbebaut haben. Wir werden die Beiträge der einzelnen Konstruktoren bei der Erstellung eines neuen `Kreis`-Objekts gleich noch genauer analysieren.

Wie schon in Abschnitt 6.2 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 implizit den parameterlosen 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
erfordert 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 parameterlose 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**-Deklaration).

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 Instanzvariablen der Klasse erhalten ggf. den in ihrer Deklaration angegebenen Initialisierungswert. Den zugehörigen MSIL-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 initialisieren, Aufruf des Basisklassenkonstruktors, Anweisungsteil). Diese Rekursion endet mit dem Aufruf eines **Object**-Konstruktors.

Betrachten wir zum Beispiel, was beim Erzeugen eines `Kreis`-Objektes 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.
- 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.
- Der parameterlose **Object**-Konstruktor startet. Die Instanzvariablen der Klasse **Object** erhalten einen Initialisierungswert gemäß **Object**-Klassendefinition. Derzeit sind mir zwar keine **Object**-Instanzvariablen bekannt, doch ist die Existenz von gekapselten Exemplaren durchaus möglich.
- Der Rumpf des parameterlosen **Object**-Konstruktors wird ausgeführt.
- 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.

6.4 Der Zugriffsmodifikator protected

Auf **private**-Member einer Basisklasse haben Methoden einer abgeleiteten Klasse (wie Methoden beliebiger Klassen) *keinen* Zugriff. Um abgeleiteten Klassen besondere Rechte einzuräumen, bietet C# den Zugriffsmodifikator **protected**, welcher den Zugriff durch die eigene Klasse *und* durch alle (direkt oder indirekt) *abgeleiteten* 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 void OLE2Zen() {
        xpos = xpos + radius;
        ypos = ypos + radius;
   }
}
```

Es ist zu beachten, dass hier *geerbte Instanzvariablen* von `Kreis`-Objekten verändert werden. Bei entsprechender Methodenausstattung kann ein `Kreis`-Objekt auch das `xpos`-Feld eines *anderen* `Kreis`-Objekts verändern. Auf das `xpos`-Feld eines `Figur`-Objekts haben die Methoden der `Kreis`-Klasse jedoch *keinen* 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 natürlich nicht nur bei Feldern erlaubt, sondern bei beliebigen Mitgliedern, z.B. bei (instanz- oder klassenbezogenen) *Methoden*:

```
static protected void ProSt() {
   Console.WriteLine("Protected und statisch!");
}
```

6.5 Erbstücke durch spezialisierte Varianten verdecken

6.5.1 Geerbte Methoden, Eigenschaften und Indexern verdecken

Eine geerbte Basisklassenmethode kann in einer abgeleiteten Klasse durch eine Methode mit gleicher Signatur **verdeckt** werden. Zwei Methoden haben genau dann dieselbe **Signatur**, wenn die Namen und die Parameterlisten (hinsichtlich Typ und Art aller Formalparameter) übereinstimmen, während die Rückgabetypen keine Rolle spielen (vgl. Abschnitt 4.3.4).

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]

[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.

```
using System;
public class Kreis : Figur {
   . . .
   public new void Wo() {
      base.Wo();
      Console.WriteLine("Unten Rechts:\t(" + (xpos + 2 * radius) +
                        ", " + (ypos + 2 * radius) + ")");
   }
}
```

Im Definitionskopf der verdeckenden Methode sollte man den Modifikator **new**[1] angeben, um die folgende Warnung des Compilers zu vermeiden:

```
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 diesem Hinweis soll ein ungeplantes Verdecken (z.B. durch Tippfehler) verhindert werden.

Im Anweisungsteil der neuen Methode kann man sich oft durch Rückgriff auf die verdeckte Methode die Arbeit erleichtern, wobei wieder das Schlüsselwort **base** zum Einsatz kommt.

Das folgende Programm schickt an eine `Figur` und an einen `Kreis` jeweils die Nachricht `Wo()`, und beide zeigen ihr artspezifisches Verhalten:

Quellcode

```
using System;
class Prog {
   static void Main() {
      Figur  f = new Figur(10.0, 20.0);
      f.Wo();
      Kreis  k = new Kreis(50.0, 50.0, 10.0);
      k.Wo();
   }
}
```

Ausgabe

```
Oben Links:     (10, 20)

Oben Links:     (50, 50)
Unten Rechts:   (70, 70)
```

Es liegt übrigens *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*.

Ist eine verdeckende Methode als privat deklariert, wird sie nur *klassenintern* verwendet, und in der Schnittstelle der abgeleiteten Klasse bleibt das signaturgleiche Erbstück erhalten, z.B.:

Quellcode

```
using System;
class Basisklasse {
    public void NenneTyp() {
        Console.WriteLine("Basisklasse");
    }
}
class Spezialklasse : Basisklasse {
    new void NenneTyp() {
        Console.WriteLine("Spezialklasse");
    }
    public void DeinTyp() {
        NenneTyp();
    }
}
```

Ausgabe

```
Basisklasse
Basisklasse
Spezialklasse
```

[1] Dieser Modifikator darf nicht mit dem *Operator* **new** verwechselt werden.

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
class Prog {
    static void Main() {
        Basisklasse b = new Basisklasse();
        b.NenneTyp();
        Spezialklasse a = new Spezialklasse();
        a.NenneTyp();
        a.DeinTyp();
    }
}
```

</td><td></td></tr>
</table>

Eine solche Konstruktion ist aber potentiell verwirrend und wohl nur selten nützlich.

Neben objektbezogenen können auch *statische* Methoden verdeckt werden, wobei die Basisklassenvariante durch Voranstellen des Klassennamens angesprochen werden kann, z.B.:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
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();
    }
}
```

</td><td>

```
Basisklasse
Spezialklasse
 abgeleitet von: Basisklasse
```

</td></tr>
</table>

Schließlich ist das Verdecken ist nicht nur bei Methoden erlaubt, sondern auch bei Eigenschaften und Indexern.

6.5.2 Geerbte Felder verdecken

Auch geerbte Instanz- und Klassenvariablen lassen sich verdecken, was aber im Sinne eines möglichst leicht 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 Basisvariable 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`

In der Deklaration einer verdeckenden Variablen sollte man den Modifikator **new**[1] angeben, um die folgende Warnung des Compilers zu vermeiden:

```
Prog.cs(9,6): warning CS0108: "AK.x" blendet den vererbten Member "BK.x" aus.
        Verwenden Sie das new-Schlüsselwort, wenn das Ausblenden vorgesehen war.
```

Mit diesem sehr aufmerksamen Hinweis soll ein ungeplantes Verdecken durch Tippfehler oder Unachtsamkeit verhindert werden.

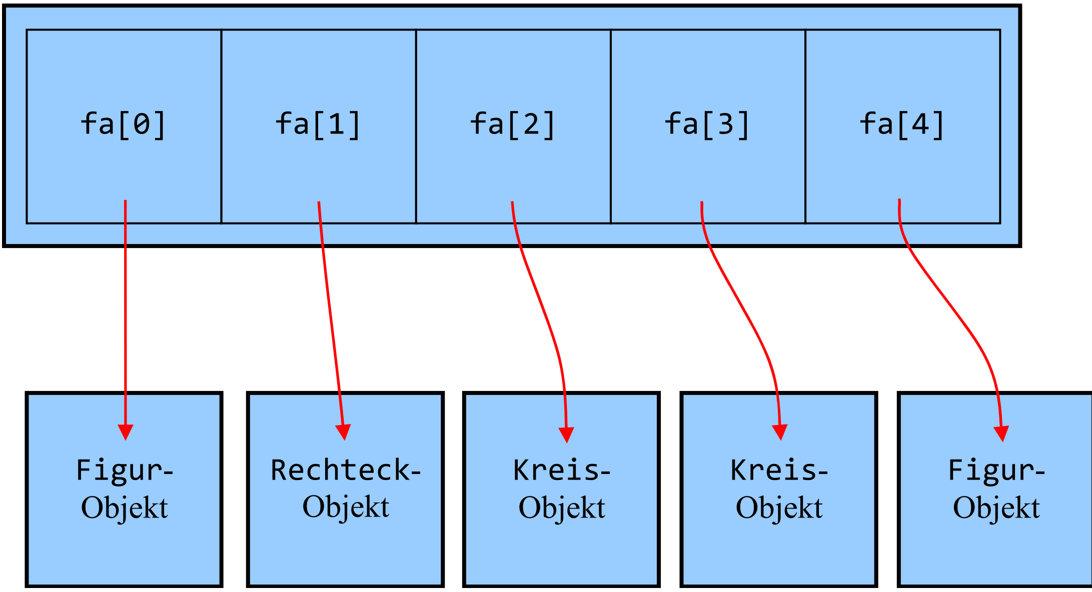
6.6 Verwaltung von Objekten über Basisklassenreferenzen

Eine Basisklassenreferenzvariable darf die Adresse eines beliebigen Unterklassenobjektes 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 in Abschnitt 6.2 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 Graphikprogramms 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:

[1] Dieser Modifikator darf nicht mit dem *Operator* **new** verwechselt werden.



Ein Array vom Typ `Figur` kann Referenzen auf Figuren und Kreise aufnehmen, z.B.:

```
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();
   }
}
```

Bei Ansprache per Basisklassenreferenz führen die Objekte *nicht* die verdeckende, artspezifische `Wo()`-Methode aus, sondern die Basisklassenvariante:

```
Oben Links:     (10, 10)
Oben Links:     (20, 20)
Oben Links:     (30, 30)
Oben Links:     (40, 40)
Oben Links:     (50, 50)
```

In Abschnitt 6.7 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 auf, z.B.:

```
Unbehandelte Ausnahme: System.InvalidCastException: Das Objekt des Typs "Figur" kann
nicht in Typ "Kreis" umgewandelt werden.
```

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);
}
```

An Stelle des gewohnten Typumwandlungsoperators

```
((Kreis)e).Radius
```

kann im Beispiel auch der **as**-Operator eingesetzt werden:

```
(e as Kreis).Radius
```

Die wichtigsten Regeln für den **as**-Operator:

- Der Zieltyp im zweiten Operanden muss ein Referenztyp oder ein **null**-fähiger Werttyp sein (siehe Abschnitt 7.3). 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 ersten 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# 4.0 - Sprachspezifikation (Microsoft 2010a, S. 202).
- Stellt sich zur Laufzeit eine Konvertierung als unmöglich heraus, liefert der **as**-Operator im Unterschied zum gewohnten Typumwandlungsoperator *keinen* Ausnahmefehler, sondern den Ergebniswert **null**. Laut C# 4.0 – 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
  ```

Lahres & Rayman (2009, Abschnitt 5.1.5) bewerten das als *Downcast* bezeichnete explizite Konvertieren in einen spezielleren Typ als Notlösung, die durch ein gutes Programmdesign vermieden werden sollte.

6.7 Polymorphie (Methoden überschreiben)

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 verdeckenden Methoden kommt die *Basisklassenvariante* zum Einsatz.
- Bei überschreibenden Methoden wird die *Unterklassenvariante* benutzt.

Während bei einer *Verdeckung* die auszuführende Methode schon beim Übersetzen festliegt, wird im Fall der *Überschreibung* erst zur Laufzeit mit Hilfe einer Verweistabelle die passende Methode gewählt, was einen erhöhten Speicher- und Zeitaufwand zur Folge hat. Man spricht hier von einer *dynamischen* oder *späten Bindung*.

Werden Objekte aus verschiedenen Klassen über Referenzvariablen eines gemeinsamen Basistyps verwaltet, sind nur Methoden nutzbar, die schon in der Basisklasse definiert sind. Bei überschriebenen Methoden reagieren die Objekte aber jeweils unterklassentypisch auf dieselbe Botschaft. Genau dieses Phänomen bezeichnet man als **Polymorphie**. Wer sich hier mit einem exotischen und nutzlo-

sen 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 dieser Softwaretechnologie (Lahres & Rayman 2009).

Zur Demonstration der Polymorphie definieren wir in der Basisklasse des Figurenbeispiels die Methode `Wo()` mit dem Modifikator **virtual** als überschreibbar:

```
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 geerbte `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; }
   }
   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 in Abschnitt 6.6 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 der Referenzvariablen) fest und wählt die passende Methode aus:

```
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 statischen Methoden sind die Modifikatoren **virtual** und **override** sinnlos und verboten. Das per **new**-Modifikator signalisierte Verdecken einer geerbten statischen Methode ist jedoch sinnvoll und erlaubt.

In der folgenden Tabelle werden die beiden Varianten der Ersetzung einer Basisklassenmethode durch eine signaturgleiche Unterklassenmethode (Verdecken und Überschreiben) in semantischer und syntaktischer Hinsicht gegenübergestellt:

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.
Überschreiben	Ja	Im Kopf der Basisklassenmethode muss der Modifikator **virtual** *und* im Kopf der Unterklassenmethode der Modifikator **override** stehen. Statische Methoden können *nicht* überschrieben werden.

Eine virtuelle Basisklassenmethode kann also verdeckt oder überschrieben werden:

- **Überschreibt** eine abgeleitete Klasse (Modifikator **override**) die Methode, 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 (Modifikator **new**) die Methode, 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.

Beide Ersetzungsarten sind auch bei Eigenschaften und Indexern anwendbar.

Die Flexibilität der Polymorphie ist nicht kostenlos zu haben, und in zeitkritischen Programmsituationen muss eventuell eine hohe Zahl von polymorphen Methodenaufrufen vermieden werden.

6.8 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, müssen die beteiligten Methoden in der Basisklasse vorhanden sein. Wenn es für die Basisklasse zu einer Methode keine sinnvolle Implementierung gibt, erstellt man dort eine **abstrakte** Methode:

- Man beschränkt sich auf den Methodenkopf, dem der Modifikator **abstract** vorangestellt wird.
- 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 hier abstrakt definiert:

```
public abstract class Figur {
   . . .
   public abstract void Skaliere(double faktor);
   . . .
}
```

Abstrakte Methoden sind grundsätzlich virtuell (vgl. Abschnitt 6.7), wobei das Schlüsselwort **virtual** überflüssig und verboten ist.

Enthält eine Klasse mindestens *eine* abstrakte Methode, dann handelt es sich um eine **abstrakte Klasse**, und bei der Klassendefinition muss der Modifikator **abstract** angegeben werden.

Aus einer abstrakten Klasse kann man zwar keine Objekte erzeugen, aber andere Klassen ableiten. Implementiert eine abgeleitete Klasse die abstrakten Methoden, lassen sich Objekte daraus herstellen; anderenfalls ist sie ebenfalls abstrakt.

Wir leiten aus der nunmehr abstrakten Klasse `Figur` die konkreten Klassen `Kreis` und `Rechteck` ab, welche die abstrakte `Figur`-Methode `Skaliere()` implementieren:

```
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 6.7) kommt bei einer abstrakten Basisklassenmethode nur das Überschreiben in Frage, wobei das zugehörige Schlüsselwort **override** anzugeben ist.

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`

Neben den Methoden können auch Eigenschaften und Indexer abstrakt definiert werden. Im Figurenbeispiel soll mit der Eigenschaft `Inhalt` die Möglichkeit geschaffen werden, den Flächeninhalt eines Objekts bequem 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 beim **get**-Block ersetzt ein Semikolon die Implementation. Analog könnte auch ein **set**-Block 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;}
   }
   .  .  .
}
```

Mit Hilfe der in Abschnitt 8.3 vorzustellenden *Schnittstellen* werden wir noch mehr Flexibilität gewinnen und polymorphe Methodenaufrufe auch für Typen *ohne* gemeinsame Basisklasse realisieren.

6.9 Das Liskovsche Substitutionsprinzip (LSP)

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 perfekt übernehmen, d.h. u.a.:

- Das „vertraglich“ zugesicherte Verhalten der Basisklassenmethoden wird auch von den 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 Perspektive 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 müssen. 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 zum Crash kommen, weil z.B. ein automatisch an die Länge eines Transporters angepasstes Objekt unbemerkt an Höhe zulegt und unterwegs gegen eine Brücke stößt.

Derartige Designfehler können vom Compiler nicht verhindert werden. C# bietet alle Voraussetzungen für eine erfolgreiche objektorientierte Analyse und Programmierung, kann aber z.B. eine Verletzung der Ersetzbarkeitsregel nicht verhindern.

6.10 Versiegelte Methoden und Klassen

Gelegentlich möchte man das Überschreiben einer Methode *verhindern*, damit auch ein per Basisklassenreferenz angesprochenes Unterklassenobjekt das Originalverhalten zeigt. Dient etwa die Methode `Passwd()` einer Klasse `Acl` zum Abfragen eines Passworts, will ihr Programmierer even-

tuell verhindern, dass `Passwd()` in einer von `Acl` abstammenden Klasse `Bcl` überschrieben wird. Damit führt auch ein per `Acl`–Referenz angesprochenes `Bcl`–Objekt die `Acl`–Methode `Passwd()` aus.

Um das Überschreiben einer Methode zu verhindern, gibt man in der Definition den Modifikator **sealed** (*versiegelt*) an. Dies ist allerdings nur bei Methoden möglich, die eine virtuelle Methode überschreiben, also auch den Modifikator **override** besitzen, z.B.:

```
class Basis {
   public virtual void Passwd() { }
}

class TopSek : Basis {
   public sealed override void Passwd() { }
}
```

Die Aussagen zum Versiegeln von Methoden gelten analog für Eigenschaften und Indexer.

Die eben für das Versiegeln von Methoden genannten Sicherheitsüberlegungen können auch zum Entschluss führen, eine komplette **Klasse** mit dem Schlüsselwort **sealed** zu fixieren, so dass sie zwar verwendet, aber nicht beerbt werden kann. Für das Versiegeln einer Klasse können aber noch weitere Gründe sprechen, wie das Beispiel der versiegelten FCL-Klasse **String** zeigt.

```
public sealed class String ...
```

6.11 Übungsaufgaben zu Kapitel 6

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:

```
class Figur {
    double xpos = 100.0, ypos = 100.0;
    public Figur(double x, double y) {
        xpos = (x >= 0) ? x : 0;
        ypos = (y >= 0) ? y : 0;
    }
    public Figur() { }
}
```

```
class Kreis : Figur {
    double radius = 75.0;
    public Kreis(double x, double y, double rad) {
        xpos = (x>=0)?x:0;
        ypos = (y>=0)?y:0;
        radius = (rad>=0)?rad:0;
    }
    public Kreis() { }
}
```

Trotzdem erlaubt der Compiler den `Kreis`-Objekten keinen direkten Zugriff auf ihre geerbten Instanzvariablen `xpos` und `ypos` (im initialisierenden `Kreis`-Konstruktor). 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?

7 Typgenerizität und Kollektionen

In C# haben die Felder von Klassen und Strukturen sowie die Parameter von Methoden einen festen Datentyp, so dass der Compiler für Typsicherheit sorgen, d.h. die Zuweisung ungeeigneter Werte bzw. Objekte verhindern kann. Oft werden 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 jeweils zu versorgende Elementtyp konkret festzulegen. Im Ergebnis erhält man durch *eine* Definition zahlreiche konkrete Klassen, wobei die Typsicherheit nicht abgeschwächt wird.

Ein besonders erfolgreiches Anwendungsfeld für Typgenerizität sind die Klassen zur Verwaltung von Listen, Mengen oder Schlüssel-Wert – Tabellen. Für die Objekte dieser Klassen wird im Manuskript alternativ zur offiziellen Bezeichnung *Kollektion* aus sprachlichen Gründen oft auch die Bezeichnung *Container* verwendet. Dass später auch von Containern zur Verwaltung von GUI-Bedienelementen die Rede sein wird, sollte keine Verwirrung stiften.

7.1 Motive für die Einführung generischer Klassen in .NET 2.0

In Abschnitt 5.3.8 haben wir die Klasse **ArrayList** aus Namensraum **System.Collections** als Container für Objekte beliebigen Typs verwendet:

```
ArrayList al = new ArrayList();
al.Add("Text");
al.Add(3.14);
al.Add(13);
```

Im Unterschied zu einem gewöhnlichen Array (siehe Abschnitt 5.3) bietet die Klasse **ArrayList**:

- eine automatische Größenanpassung
- Typflexibilität (durch Verwendung des 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 zwei Nachteile der Typbeliebigkeit:

- Wenn beliebige Objekte zugelassen sind, die intern über Referenzvariablen vom Typ **Object** verwaltet werden, kann der Compiler keine **Typsicherheit** garantieren. Er kann nicht sicher stellen, dass ausschließlich Objekte des gewünschten Typs in den Container eingefüllt werden. Viele Programmierfehler werden erst zur Laufzeit entdeckt.
- Wenn Variablen mit Werttyp verwaltet werden, resultieren leistungsschädliche (Un)Boxing-Operationen.
- Entnommene Objekte können erst nach einer expliziten Typumwandlung die Methoden ihres Klasse ausführen. Die häufig benötigten Typanpassungen sind lästig und fehleranfällig.

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 ArrayList();
  al.Add("Otto");
  al.Add("Rempremerding");
  al.Add('.');
```

```
      int i = 0;
      foreach (Object s in al)
         Console.WriteLine("Laenge von Zeile {0}: {1}\n", ++i, ((String)s).Length);
    }
  }
```

Bevor ein **String**-Element des Containers nach seiner Länge befragt werden kann, ist eine lästige Typanpassung fällig, weil der Compiler nur die Typangabe **Object** kennt:

```
((String)s).Length
```

Beim dritten **Add**()-Aufruf wird ein Wert 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 mangelhaften Typsicherheit nicht bemerken konnte. Beim Versuch, den **char**-Wert als **String**-Objekt zu behandeln, scheitert das Programm am folgenden Ausnahmefehler:

```
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 11.
```

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 analoge 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 Klassen. Durch Verwendung von Typparametern bei der Definition wird die gesamte Handlungskompetenz einer Klasse typunabhängig formuliert. Bei jeder Instantiierung (z.B. beim Erstellen eines Container-Objekts) sind jedoch konkrete Typen anzugeben. Weil der Compiler die konkreten Typen kennt, kann er bei Zuweisungen für Typsicherheit sorgen, und es sind keine lästigen Typumwandlungen erforderlich.

Mit der generischen Klasse **List<T>** im Namensraum **System.Collections.Generic** enthält die FCL seit der Version 2.0 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 List<String>();
      gl.Add("Otto");
      gl.Add("Rempremerding");
      gl.Add(".");
      int i = 0;
      foreach (String s in gl)
         Console.WriteLine("Laenge von Zeile {0}: {1}\n", ++i, s.Length);
      Console.ReadLine();
   }
}
```

Bei der Erstellung eines **List<T>** - Objekts ist an Stelle des Typparameters **T** ein konkreter Datentyp anzugeben:

```
List<String> gl = new List<String>();
```

Jeder Versuch, ein Element mit abweichendem Typ in den Container einzufügen, wird vom Compiler bemerkt und verhindert, z.B.:

```
gl.Add('.');
```

Die Elemente des auf **String**-Objekte spezialisierten Containers beherrschen ohne Typanpassung die Methoden ihrer Klasse, z.B.:

```
foreach (String s in gl)
  Console.WriteLine("Laenge von Zeile {0}: {1}\n",++i,s.Length);
```

Bei einem dynamischen Container für Elemente mit einem festen *Werttyp* erspart die Klasse **List<T>** im Vergleich zu **ArrayList** die zeitaufwändigen (Un)boxing-Operationen. Mit diesem Thema sollen Sie sich im Rahmen einer Übungsaufgabe beschäftigen.

Für Container zur Aufnahme von Elementen mit *unterschiedlichen* Typen ist die Klasse **ArrayList** weiterhin gefragt.

7.2 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 FCL-Namensraum **System.Collections.Generic**), aber keinesfalls auf Container mit ihrer weitgehend inhaltstypunabhängigen Verwaltungslogik beschränkt.

Analog zu den generischen Klassen bietet C# auch generische Strukturen, Schnittstellen und Delegaten. Wir befassen uns in Abschnitt 7 hauptsächlich mit generischen Klassen und Strukturen, machen aber auch schon erste Erfahrungen mit generischen Schnittstellen, die wir bald in Abschnitt 8 vertiefen werden.

7.2.1 Definition

Bei der generischen Klassendefinition verwendet man **Typformalparameter**, die im Kopf der Definition hinter dem Klassennamen zwischen spitzen Klammern und durch Kommata getrennt angegeben werden. Wir erstellen als Beispiel eine generische Klasse namens `EinfachStapel<T>`, die einen LIFO-Stapel (*last-in-first-out*) verwaltet und mit *einem* Typformalparameter für den beliebig wählbaren Elementtyp auskommt. In der Praxis wird man bei solchen Standardaufgaben allerdings eine fertige Container-Klasse aus dem FCL-Namensraum **System.Collections.Generic** verwenden.

```
using System;
public class EinfachStapel<T> {
   int maxHoehe = 5;
   T[] daten;
   int aktHoehe;

   public EinfachStapel() {
      daten = new T[maxHoehe];
   }
   public EinfachStapel(int max) {
      if (max > 0)
         maxHoehe = max;
      daten = new T[maxHoehe];
   }
   public bool Auflegen(T element) {
      if (aktHoehe < maxHoehe) {
         daten[aktHoehe++] = element;
         return true;
      } else
         return false;
   }
```

```
      public bool Abheben(out T element) {
         if (aktHoehe > 0) {
            element = daten[--aktHoehe];
            return true;
         } else {
            element = default(T);
            return false;
         }
      }
   }
```

Mit der Methode `Auflegen()` legt man ein neues Element auf den Stapel, sofern seine Kapazität nicht erschöpft ist. Solange der Vorrat reicht, kann man das jeweils oberste Element `Abheben()`. Ist der Stapel leer, wird dem Ausgabeparameter vom generischem Typ **T** der Wert **default(T)** zugewiesen, d.h.:

- **null**, wenn beim Erstellen des `EinfachStapel`-Objekts für **T** ein Referenztyp angegeben wurde,
- die passende numerische Null, wenn beim Erstellen des `EinfachStapel`-Objekts für **T** ein numerischer Werttyp angegeben wurde,
- eine Strukturinstanz mit komplett auf Null bzw. **null** initialisierten Feldern, wenn beim Erstellen des `EinfachStapel`-Objekts für **T** ein Strukturtyp angegeben wurde.

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 `Auflegen()`

Vielleicht vermissen Sie beim `EinfachStapel` die Größendynamik der Kollektionsklasse **ArrayList** (vgl. Abschnitt 5.3.8). Um dieses automatische Wachstum zu realisieren, könnte man den intern zur Datenspeicherung benutzten Array in leistungsoptimierend geplanten Stufen durch ein größeres Exemplar ersetzen, z.B. mit jeweils doppelter Länge. Dabei wären die bisherigen Elemente zu kopieren. Im Beispiel wurde der Einfachheit halber auf die Größendynamik verzichtet.

In Abschnitt 7.5.2 werden Sie mit der FCL-Klasse **Dictionary<K, V>** ein Beispiel für eine generische Klasse mit *zwei* Typformalparametern kennen lernen.

Bei der Verwendung eines generischen (*offenen*) Typs durch Wahl konkreter Datentypen an Stelle der Typformalparameter entsteht ein *geschlossener* Typ (Richter 2006, S. 385).

7.2.2 Restringierte Typformalparameter

Häufig muss eine generische Klassendefinition bei den konkreten Klassen, welche einen Typparameter konkretisieren dürfen, gewisse Handlungskompetenzen voraussetzen. Soll z.B. ein generischer Container seine Elemente sortieren, dann muss jeder konkrete Elementtyp die *Schnittstelle* **IComparable<T>** erfüllen, d.h. es muss eine Methode namens **CompareTo()** mit folgender Signatur vorhanden sein (hier beschrieben unter Verwendung des Typparameters **T**):

public int CompareTo(T ***element*****)**

In Abschnitt 5.4.1.2.2 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 Typ

verlangt. Mit dem generellen Thema *Schnittstellen* werden wir uns in Abschnitt 8 ausführlich beschäftigen. Dabei wird sich herausstellen, dass zur generischen Schnittstelle **IComparable<T>** auch noch die ältere, nichtgenerische Variante **IComparable** existiert, die eine Methode

public int CompareTo(Object *element***)**

vorschreibt. Weil die generische Variante eine Typprüfung durch den Compiler ermöglicht, ist sie zu bevorzugen.

Wir erstellen nun eine generische Listenverwaltungsklasse, die eingefügte Elemente automatisch einsortiert und daher ihren Typformalparameter auf den Schnittstellentyp **IComparable<T>** einschränkt:

```
using System;
public class SimpleSortedList<T> where T : IComparable<T> {
   const int DEFLEN = 5;
   T[] elements;
   int firstFree;

   public SimpleSortedList(int len) {
      if (len > 0)
         elements = new T[len];
   }
   public SimpleSortedList() {
      elements = new T[DEFLEN];
   }

   public bool Add(T element) {
      if (firstFree == elements.Length)
         return false;
      bool inserted = false;
      for (int i = 0; i < firstFree; i++) {
         if (element.CompareTo(elements[i]) <= 0) {
            for (int j = firstFree; j > i; j--)
               elements[j] = elements[j-1];
            elements[i] = element;
            inserted = true;
            break;
         }
      }
      if (!inserted)
         elements[firstFree] = element;
      firstFree++;
      return true;
   }

   public bool Get(int index, out T element) {
      if (index >= 0 && index < firstFree) {
         element = elements[index];
         return true;
      } else {
         element = default(T);
         return false;
      }
   }
}
```

Bei der Formulierung von Einschränkungen für einen Typparameter wird das Schlüsselwort **where** verwendet, wobei u.a. folgende Regeln gelten:

- Man kann eine Basisklasse vorschreiben.
- Mit dem Schlüsselwort **class** wird vereinbart, dass nur Referenztypen erlaubt sind.
- Mit dem Schlüsselwort **struct** wird vereinbart, dass nur Werttypen erlaubt sind.

- Man kann auch *mehrere* Restriktionen durch Kommata getrennt angeben, die von einem konkreten Typ allesamt zu erfüllen sind.
- Während nur eine Basisklasse vorgeschrieben werden darf, sind beliebig viele Schnittstellen (vgl. Abschnitt 8) erlaubt, die ein konkreter Typ *alle* erfüllen muss.
- Mit dem Listeneintrag **new()** wird für die konkreten Typen ein parameterfreier Konstruktor vorgeschrieben.

Eine ausführliche Darstellung der möglichen Typrestriktionen finden Sie z.B. bei Richter (2006, S. 394ff).

Im folgenden Programm wird aus der offenen Klasse `SimpleSortedList<T>` eine geschlossene Klasse zur Verwaltung einer sortierten **int**-Liste erzeugt:

Quellcode	Ausgabe

`using System;`
`class Prog {`
`   static void Main() {`
`      SimpleSortedList<int> si = new SimpleSortedList<int>(5);`
`      si.Add(11);si.Add(2);si.Add(1);si.Add(4);`
`      int result;`
`      for (int i = 0; i < 4; i++)`
`         if (si.Get(i, out result))`
`            Console.WriteLine(result);`
`   }`
`}`
	1
	2
	4
	11

7.2.3 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 konkretisierte generische Klasse
  ```
  class GenDerived<T> : GenBase<int, double> {
  . . .
  }
  ```
- eine generische Klasse mit kompatiblen Typformalparametern, 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> {
  . . .
  }
  ```
  Im Beispiel hat die konkretisierte Version `GenDerived<int>` die Basisklasse `GenBase<int, int>`.

Offenbar ist die nachträgliche Integration der generischen Typen in das *Common Type System* (CTS) der .NET - Plattform gut gelungen.

7.3 Nullable<T> als Beispiel für generische Strukturen

In diesem Abschnitt wird die generische Struktur **Nullable<T>** aus der FCL vorgestellt:[1]

```
public struct Nullable<T> where T : struct {
   private bool hasValue;
   internal T value;
   public Nullable(T value) {
      this.value = value;
      this.hasValue = true;
   }
   public bool HasValue {
      get {
         return hasValue;
      }
   }
   public T Value {
      get {.  .  .}
   }
   .  .  .
}
```

Mit den Instanzen einer Konkretisierung lassen sich Werte einer Struktur (z.B. Werte eines elementaren Datentyps) so verpacken, dass neben den normalen Werten auch der Ausnahmewert **null** zur Verfügung steht. Er eignet sich etwa dazu, bei einer Variablen einen undefinierten Zustand zu signalisieren. Verpackt man z.B. eine Variablen vom Typ **bool**, erhält man neben den Werten **true** und **false** noch den dritten Wert **null**, der als *unbekannt* interpretiert werden kann.

Im folgenden Beispiel wird eine Variable vom Typ **Nullable<bool>** definiert:

```
Nullable<bool> status;
```

Man kann ihr den Wert **null** zuweisen:

```
status = null;
```

Die **null**-fähige Variante zu einem Strukturtyp lässt sich auch durch den Namen des Grundtyps und ein angehängtes Fragezeichen ausdrücken, z.B.:

```
bool? status;
```

Wie der zu Beginn des aktuellen Abschnitts präsentierte Quellcode zeigt, beschränkt sich der zusätzliche Speicherplatzbedarf einer **null**-fähigen Strukturinstanz im Vergleich zum Grundtyp auf eine **bool**-Variable.

Eine Nullable-Instanz informiert in der booleschen 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.:

```
int?[] 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");
```

Während der Grundtyp implizit in den zugehörigen **Nullable**-Typ konvertiert wird, ist für den umgekehrten Übergang eine *explizite* Konvertierung erforderlich, z.B.:

[1] Der Quellcode stammt aus Microsofts *Shared Source Common Language Infrastructure 2.0.*

```
double? d = 77.7;
double dn = (double) d;
```

Die beim Grundtyp unterstützten **Operatoren** sind auch bei der **null**-fähigen Verschachtelung 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.

```
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 stets **false** ist, z.B. beim Vergleich:

```
0 == null
```

Eine *Instanzvariable* mit **Nullable**-Typ wird mit **null** initialisiert, z.B.:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>using System;
class NullableInit {
   public int gin;
   public int? nin;
}
class Prog {
   static void Main() {
      NullableInit niob = new NullableInit();
      Console.WriteLine(niob.gin);
      Console.WriteLine(niob.nin.HasValue);
   }
}</pre></td><td><pre>0
False</pre></td></tr>
</table>

Mit dem so genannten **Null-Koaleszenz - Operator**, der durch zwei Fragezeichen ausgedrückt wird, lässt sich die Zuweisung einer **null**-fähigen Strukturinstanz an eine Variable des Grundtyps samt Ausnahme für die kritische **null**-Situation bequem formulieren, z.B.:

<table>
<tr><th>Quellcodesegment</th><th>Ausgabe</th></tr>
<tr><td><pre>int? ni = 4;
int i = ni ?? -1;
Console.WriteLine(i);</pre></td><td><pre>4</pre></td></tr>
</table>

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.

Das Bemühen von Compiler und CLR, eine **Nullable**-Instanz wie einen Wert des zugehörigen Grundtyps zu behandeln, geht so weit, dass bei einer **GetType()**-Anfrage der Grundtyp genannt wird, z.B.:

<table>
<tr><th>Quellcodesegment</th><th>Ausgabe</th></tr>
<tr><td><pre>double? d = 3.0;
Console.WriteLine(d.GetType());</pre></td><td><pre>System.Double</pre></td></tr>
</table>

7.4 Generische Methoden

Wenn mehrere überladene Methoden identische Operationen enthalten, stellt *eine* generische Methode oft die bessere Lösung dar. Im folgenden Beispiel wird das Maximum zweier Argumente

geliefert, wobei der gemeinsame Datentyp **T** die Schnittstelle **IComparable<T>** erfüllen, also eine Methode **CompareTo()** besitzen muss:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
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));
  }
}
```

</td><td>

```
int-max:        13
double-max:     47,11
```

</td></tr>
</table>

Man benennt die **Typformalparameter** einer generischen Methode hinter dem Methodennamen zwischen spitzen Klammern und ggf. voneinander durch Kommata getrennt. Sie sind als Datentypen für den Rückgabewert, Parameter und lokale Variablen erlaubt. Wie bei generischen Klassen kann man Restriktionen für Typparameter formulieren (siehe Abschnitt 7.2.2).

7.5 Weitere Klassen aus dem Namensraum System.Collections.Generic

Nach den positiven Erfahrungen mit der generischen Klasse **List<T>** zur bequemen und typsicheren Verwaltung sortierter Elemente eines festen Typs (siehe Abschnitt 7.1), werfen wir noch einen neugierigen Blick auf weitere generische Klassen im Namensraum **System.Collections.Generic**.[1]

7.5.1 Verwaltung einer Menge mit der Klasse HashSet<T>

Die generische Klasse **HashSet<T>** eignet sich zur Verwaltung einer Menge von Elementen, wobei im Unterschied zu einer Liste ...

- einerseits *keine* relevante Anordnung besteht,
- andererseits aber das Auftreten von Dubletten zu verhindern ist.

Ein Objekt dieser Klasse beherrscht u.a. die folgenden Instanzmethoden:

- **public boolean 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 boolean Contains(T** *element***)**
  Diese Methode informiert darüber, ob das fragliche Element in der Menge vorhanden ist.
- **public boolean Remove(T** *element***)**
  Das angegebene Element wird aus der Menge entfernt, falls es dort vorhanden ist. Über den Rückgabewert erfährt man, ob die Menge durch den Aufruf verändert worden ist.
- **public void Clear()**
  Das angesprochene **HashSet<T>** - Objekt entfernt alle Elemente.

---

[1] Verwendet eine Anwendung mehrere nebenläufige *Ausführungsfäden* (**Multithreading**, siehe Kapitel 13), wird eventuell eine Thread-sichere Kollektion aus dem mit .NET 4.0 eingeführten Namensraum **System.Collections.Concurrent** benötigt.

- **public void IntersectWith(IEnumerable<T>** *kollektion***)**
  Das angesprochene **HashSet<T>** - Objekt streicht alle Elemente, die nicht in der Parameterkollektion (ein Objekt einer Klasse, welche die Schnittstelle **IEnumerable<T>** erfüllt), enthalten sind.
- **public void UnionWith(IEnumerable<T>** *kollektion***)**
  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.

Das folgende Programm demonstriert die aufgelisteten Methoden:

Quellcode

```
using System;
using System.Collections.Generic;
class HashSetDemo {
  static void Main() {
    HashSet<char> m1 = new HashSet<char>();
    m1.Add('a'); m1.Add('b'); m1.Add('c');
    Console.WriteLine("Menge 1:");
    foreach (char c in m1)
      Console.Write(c + " ");

    HashSet<char> m2 = new HashSet<char>();
    m2.Add('c'); m2.Add('d'); m2.Add('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'));

    HashSet<char> schnitt = new HashSet<char>(m1);
    schnitt.IntersectWith(m2);
    Console.WriteLine("\nSchnittmenge:");
    foreach (char c in schnitt) Console.Write(c + " ");

    HashSet<char> vereinigung = new HashSet<char>(m1);
    vereinigung.UnionWith(m2);
    Console.WriteLine("\n\nVereinigungsmenge:");
    foreach (char c in vereinigung)  Console.Write(c + " ");

    m1.Clear();
    Console.WriteLine("\n\nMenge 1 nach Clear:");
    foreach (char c in m1) Console.Write(c + " ");
  }
}
```

Ausgabe

```
Menge 1:
a b c

Menge 2:
c d e

b in der Menge 2? False

Schnittmenge:
c

Vereinigungsmenge:
a b c d e

Menge 1 nach Clear:
```

7.5.2 Verwaltung einer Menge von Schlüssel-Wert - Paaren mit der Klasse Dictionary<K, V>

Die generische Klasse **Dictionary<K, V>** eignet sich zur Verwaltung einer Menge von Schlüssel-Wert - Paaren, wobei ...

- keine relevante Anordnung besteht,
- die Eindeutigkeit der Schlüssel garantiert ist.

Ein Objekt dieser Klasse 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 boolean ContainsKey(K** *key***)**
  Diese Methode informiert darüber, ob ein Element mit dem fraglichen Schlüssel vorhanden ist.

- **public boolean RemoveKey(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 verändert worden ist.
- **public void Clear()**
  Das angesprochene **Dictionary<K, V>** - Objekt entfernt alle Elemente.

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 ändern, z.B.:

```
Dictionary<char, int> fred = new Dictionary<char, int>();
fred.Add('c', 1);
fred['c'] = 5;
```

Durchläuft man eine **Dictionary<K, V>** – Kollektion mit der **foreach** – Schleife, ist als Elementtyp die passend konkretisierte generische Struktur **KeyValuePair<K, V>** anzugeben, z.B.:

```
foreach (KeyValuePair<char, int> kvp in fred) {
  Console.WriteLine("{0} : {1}", kvp.Key, kvp.Value);
```

Über die Eigenschaften **Key** bzw. **Value** erreicht man den Schlüssel bzw. Wert einer Instanz.

Das folgende Programm verwendet ein **Dictionary<char, int>** – Objekt dazu, für einen **String** die Häufigkeiten der enthaltenen Zeichen zu ermitteln:

Quellcode

```
using System;
using System.Collections.Generic;

class DictionaryDemo {
   public static void CountLetters(String text) {
      Dictionary<char, int> 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("{0} : {1}", kvp.Key, kvp.Value);
      }

   }

   static void Main() {
      CountLetters("Otto spielt Lotto.");
   }
}
```

Ausgabe

```
O : 1
t : 5
o : 3
  : 2
s : 1
p : 1
i : 1
e : 1
l : 1
L : 1
. : 1
```

7.6 Übungsaufgaben zu Kapitel 7

1) Erstellen Sie eine Variante der in Abschnitt 5.6 vorgestellten Personenverwaltung, wobei die Klasse `PersonDB` (Eigenbau einer verketteten Liste mit Indexer) durch die generische Kollektionsklasse **List<T>** ersetzt wird.

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** zeitaufwändigen (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 Mio. **int**-Werten in einen **ArrayList**- bzw. **List<int>**- Container.

8 Interfaces

Zu vielen Klassen oder Strukturen führt die FCL-Dokumentation hinter dem Namen und einem Doppelpunkt *mehrere* Typen auf, z.B. bei der Klasse **String**:

```
[SerializableAttribute]
[ComVisibleAttribute(true)]
public sealed class String : IComparable,
    ICloneable, IConvertible, IComparable<string>, IEnumerable<char>,
    IEnumerable, IEquatable<string>
```

Um sieben Basisklassen kann es sich nicht handeln, weil C# keine Mehrfachvererbung unterstützt. Außerdem ist der FCL-Dokumentation zu entnehmen, dass die Klasse **String** direkt von der Urahnklasse **Object** abstammt. Am Anfangsbuchstaben *I* sind in der FCL-Dokumentation zuverlässig die von einer Klasse oder Struktur implementierten *Schnittstellen* (englisch: *Interfaces*) zu erkennen. Hierbei handelt es sich um **Verpflichtungserklärungen** von Klassen oder Strukturen gegenüber dem Compiler. Ein Interface definiert eine Reihe von Handlungskompetenzen abstrakt (ohne Implementierung) über Signaturen von Methoden oder anderen ausführbaren Membern (z.B. Eigenschaften). Wenn sich ein Typ zu einem Interface bekennt, muss er die dort geforderten Handlungskompetenzen implementieren. Als Gegenleistung werden seine Instanzen vom Compiler überall akzeptiert, wo die jeweiligen Schnittstellenkompetenzen vorausgesetzt werden.

Die Liste der von einem Typ implementierten Interfaces liefert also wichtige Informationen über die Handlungskompetenzen seiner Instanzen. Diese Informationen sind in erster Linie für den Compiler gedacht, doch sind sie (neben anderen Informationsquellen) auch bei der Verwendung eines Typs durch andere Programmierer relevant. 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.

  C# unterstützt also auch bei den Schnittstellen *generische* Varianten mit Typparametern. Dabei gewinnt man Typsicherheit zur Übersetzungszeit und spart bei Strukturinstanzen (Un)boxing-Operationen (siehe Abschnitt 8.2).

  Weil **String** die Schnittstelle **IComparable** implementiert, muss eine Methode

  public int CompareTo(Object *obj***)**

  vorhanden sein. Um den Vertrag **IComparable<String>** zu erfüllen, wird eine Methode mit der folgenden Signatur benötigt:

  public int CompareTo(String *str***)**

  Seit der C# - Version 2.0 sind beide Überladungen in der Klasse **String** vorhanden. Weil der Compiler solche Aufrufe

  ```
  Console.WriteLine("Alpha".CompareTo(3));
  ```

  nicht verhindern kann, muss die Methode **CompareTo(Object** *obj***)** darauf vorbereitet sein. Sie reagiert sinnvoll mit einem **ArgumentException**-Ausnahmefehler. Aus Kompatibilitätsgründen muss die Klasse **String** an der Methode **CompareTo(Object** *obj***)** und am Bekenntnis zur Schnittstelle **IComparable** festhalten.

  Nun beschäftigen wir uns endlich mit den nützlichen Konsequenzen der **String**-Verpflichtungserklärung. Weil **String**-Objekte die Fähigkeit zum Vergleich mit Artgenossen besitzen, kann z.B. ein Array mit Elementen dieses Typs bequem 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`

- **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 erstaunlicherweise *keine* generische Alternative. Ein Grund könnte darin bestehen, dass die **Clone**()-Methode und damit das **ICloneable**-Interface durch eine Implementierungsunklarheit stark an Nutzen eingebüßt haben. Es geht um den Unterschied zwischen der *tiefen* Kopie, die auch alle direkten und indirekten Member-Objekte dupliziert, und der *flachen* Kopie, die Member-Objekte des Originals weiterverwendet. Bei vorhandenen Klassen ist potentiell unklar, welche **Clone**()-Implementierung sie verwenden. Für eine eigene Klasse kann man eine tiefe Kopie nicht garantieren, sobald Member-Objekte vorhanden sind.

Schnittstellen definieren Verhaltenskompetenzen abstrakt durch Methoden, Eigenschaften und Indexer ohne Anweisungsteil. Außerdem sind Ereignisdeklarationen möglich. Man kann Schnittstellen in erster Näherung als Klassen mit ausschließlich abstrakten Methoden beschreiben. Ein Interface ist also ein **Datentyp**. Es lassen sich zwar keine *Instanzen* von diesem Typ erzeugen, aber *Referenzvariablen* sind erlaubt und als Abstrakationsmittel 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). Dabei werden Methodenaufrufe **polymorph** ausgeführt (späte bzw. dynamische Bindung, siehe Abschnitt 6.7), weil Interface-Methoden grundsätzlich virtuell sind.

Implementiert eine Klasse oder Struktur ein Interface, dann …

- muss sie die im Interface enthaltenen Methoden implementieren,
- werden Variablen von diesem Typ vom Compiler überall akzeptiert, wo der Interface-Datentyp vorgeschrieben ist.

Im Programmieralltag kommen wir auf unterschiedliche Weise mit Schnittstellen in Kontakt, z.B.:

- Bei der Verwendung von Instanzen fremder Typen in eigenen Methodendefinitionen nutzen wir Handlungskompetenzen, die durch Schnittstellen-Verpflichtungen dieser Typen garantiert sind. Es ist nicht unbedingt erforderlich, die fremden Typen namentlich zu kennen. Bei einer eigenen Methodendefinition kann es z.B. sinnvoll sein, Parameterdatentypen über Schnittstellen zu definieren. Damit wird Typsicherheit erreicht, ohne die Erweiterbarkeit eines Software-Systems unnötig zu behindern.
- Implementierung von vorhandenen Schnittstellen in einer eigenen Typdefinition
  Damit werden Variablen dieses Typs vom Compiler überall akzeptiert (z.B. als Aktualparameter), wo die jeweiligen Schnittstellenkompetenzen gefordert sind.
- Definition von eigenen Schnittstellen
  Beim Entwurf eines Software-Systems, das als Halbfertigprodukt (oder Programmgerippe) für verschiedene Aufgabenstellungen durch spezielle Klassen und Strukturen mit bestimmten Verhaltenskompetenzen zu einem lauffähigen Programm komplettiert werden soll, definiert man eigene Schnittstellen, um die Interoperabilität der Typen sicher zu stellen. In diesem Fall spricht man von einem *Framework*. Ein wichtiges „Halbfertigprodukt", aus dem durch Ihre Klassen und Strukturen vollständige Programme entstehen, kennen Sie mit dem .NET - Framework ja schon. Auch bei einem *Entwurfsmuster* (engl.: *design pattern*), das für eine konkrete Aufgabe bewährte Lösungsverfahren vorschreibt, spielen oft Schnittstellen eine wichtige Rolle.

8.1 Interfaces definieren

Wir behandelt zuerst das im Programmieralltag vergleichweise seltene Definieren einer Schnittstelle, weil dabei Inhalt und Funktion gut zu erkennen sind. Allerdings verzichten wir auf ein eigenes Beispiel und betrachten stattdessen die angenehm einfach aufgebaute und außerordentlich wichtige Schnittstelle **IComparable<T>** aus der FCL:

```
public interface IComparable<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 fixierbaren syntaktischen Forderungen meist auch semantische Vorstellungen im Spiel. Der Compiler kann aber z.B. aufgrund der obigen **IComparable<T>**-Definition sinnlose **CompareTo()**-Implementierungen *nicht* verhindern.

Einige Regeln für Schnittstellendefinitionen:

- Modifikator **public**
  Wird **public** *nicht* angegeben, ist die Schnittstelle nur innerhalb ihres Assemblies verwendbar (Schutzstufe **internal**).
- Modifikator **abstract**
  Schnittstellen sind grundsätzlich **abstract**. Der Modifikator **abstract** ist überflüssig und verboten.
- Schlüsselwort **interface**
  Das obligatorische Schlüsselwort dient zur Unterscheidung von Klassen- und Strukturdefinitionen.

- Schnittstellenname
  Per Konvention beginnt der Interfacename mit einem großen *I*. Bei generischen Schnittstellen folgen die Typformalparameter dem Namen zwischen spitzen Klammern und durch Kommata getrennt. Wie bei generischen Klassen können auch bei generischen Schnittstellen Restriktionen für die Typparameter formuliert werden (vgl. Abschnitt 7.2.2).
- Erlaubte Interface-Member
  Als Interface-Member sind nur *instanzbezogene Methoden, Eigenschaften, Indexer* und *Ereignisse* erlaubt. Verboten sind Konstruktoren, Felder sowie statische Member.
- Definition von Methoden, Eigenschaften, Indexern und Ereignissen
  In einer Schnittstelle sind alle Methoden etc. grundsätzlich **public** und **abstract** (folglich auch **virtual**). Die drei Schlüsselwörter sind überflüssig und verboten. Bei der Implementierung einer Schnittstellenmethode etc. (siehe Abschnitt 8.2) muss (und darf) der Modifikator **override** *nicht* angegeben werden, weil keine Alternative zum Überschreiben besteht.

Ein Interface kann andere Interfaces **beerben** (bzw. erweitern), wobei dieselbe Syntax wie beim Ableiten von Klassen zu verwenden ist. In der FCL wird z.B. das Interface **IEnumerable** durch das Interface **ICollection** (beide im Namensraum **System.Collections**) erweitert:

```
public interface ICollection : IEnumerable
{
   void CopyTo(Array array, int index);
   int Count { get; }
   Object SyncRoot { get; }
   bool IsSynchronized { get; }
}
```

Oft werden generische Schnittstellen als Erweiterung der älteren, nicht-generischen Variante definiert, z.B.:

```
public interface IEnumerable<T> : IEnumerable
{
   . . .
}
```

Bei der Implementation einer erweiternden Schnittstelle durch eine Klasse oder Struktur sind auch die Handlungskompetenzen der Basisschnittstellen zu realisieren.

Während bei *Klassen* die Mehrfachvererbung *nicht* unterstützt wird, ist sie bei *Schnittstellen* möglich (und oft auch sinnvoll).

Weil die Schnittstellenhierarchie von der Klassenhierarchie unabhängig ist, kann ein Interface von beliebigen Klassen und Strukturen implementiert werden.

8.2 Interfaces implementieren

Soll für eine Klasse oder Struktur angezeigt werden, dass ihre Instanzen auch die Datentypen bestimmter Schnittstellen erfüllen, müssen die Interfaces 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.

Als Beispiel dient eine Klasse namens `Figur`, die nur begrenzte Ähnlichkeit mit namensgleichen früheren Beispielklassen besitzt. Sie implementiert das Interface **IComparable**`<Figur>`, damit `Figur`-Kollektionen bequem sortiert werden können:

```
using System;
public class Figur : IComparable<Figur> {
   String name = "unbenannt";
   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 int CompareTo(Figur vergl) {
      if (xpos < vergl.xpos)
         return -1;
      else
         if (xpos > vergl.xpos)
            return 1;
      return 0;
   }
}
```

Alle Handlungskompetenzen einer im Kopf angemeldeten Schnittstelle müssen implementiert werden. Bei der generischen Schnittstelle **IComparable<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 obiger Realisation werden Figuren nach der X-Koordinate ihrer linken oberen Ecke verglichen:

- 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 die 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 zur Situation beim Überschreiben von abstrakten Methoden) *nicht* angegeben werden.

Soll eine implementierende Methode überschreibbar sein, ist das Schlüsselwort **virtual** anzugeben.

Eine Klasse kann auf das Implementieren einiger Interface-Handlungskompetenzen verzichten und diese wie auch sich selbst als **abstract** deklarieren.

Weil die `Figur`-Objekte verglichen werden können, gelingt das Sortieren ganzer Kollektionen mit der **List<T>** - Methode **Sort()** mühelos, z.B.:

Quellcode

```
using System;
using System.Collections.Generic;
class Prog {
   static void Main() {
      List<Figur> lf = new List<Figur>();
      lf.Add(new Figur("A", 250.0, 50.0));
      lf.Add(new Figur("B", 150.0, 50.0));
      lf.Add(new Figur("C", 50.0, 50.0));
      Console.WriteLine(lf[0].Name+" "+lf[1].Name+" "+lf[2].Name);
      lf.Sort();
      Console.WriteLine(lf[0].Name+" "+lf[1].Name+" "+lf[2].Name);
}
```

Ausgabe

```
A B C
C B A
```

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 vielen Fällen eine Ersatzlösung, denn:

- Eine Klasse darf beliebig viele Schnittstellen implementieren, so dass 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, wird einer Klasse beim Implementieren von Schnittstellen aber nichts geschenkt, sondern sie gibt Verpflichtungserklärungen ab und muss die entsprechenden Leistungen erbringen.
- Bei Schnittstellen ist Mehrfachvererbung erlaubt.

Im Zusammenhang mit dem Thema *Vererbung* ist noch 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` von der oben vorgestellten Klasse `Figur` abgeleitet, dann arbeitet die statische **Sort()**-Methode der Klasse **Array** auch mit `Kreis`-Objekten, z.B.:

Quellcode

```
using System;
using System.Collections.Generic;
class Prog {
   static void Main() {
      List<Kreis> kl = new List<Kreis>();
      kl.Add(new Kreis("A", 250.0, 50.0, 10.0));
      kl.Add(new Kreis("B", 150.0, 50.0, 20.0));
      kl.Add(new Kreis("C", 50.0, 50.0, 30.0));
      Console.WriteLine(kl[0].Name+" "+kl[1].Name+" "+kl[2].Name);
      kl.Sort();
      Console.WriteLine(kl[0].Name+" "+kl[1].Name+" "+kl[2].Name);
  }
}
```

Ausgabe

```
A B C
C B A
```

Besteht die Wahl, ein generisches oder ein nicht-generisches Interface zu implementieren, sollte in der Regel die generische Variante gewählt werden. Man gewinnt Typsicherheit und vermeidet bei Strukturen zeitaufwändige Boxing-Operationen. Dies soll bei einer einfachen Struktur für Punkte in der Zahlenebene mit einem Vergleich anhand der X-Koordinate 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 MSIL-Code der **Main**()-Methode zeigt, erfordert außerdem jeder *gelungene* **CompareTo**() - Aufruf eine Boxing -Operation:

```
Prog::Main : void()
Suchen   Weitersuchen
IL_0034:  nop
IL_0035:  ldloca.s   p1
IL_0037:  ldloc.1
IL_0038:  box        Punkt
IL_003d:  call       instance int32 Punkt::CompareTo(object)
IL_0042:  call       void [mscorlib]System.Console::WriteLine(int32)
```

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;
   }
}
```

kann die fehlerhafte Quellcodezeile nicht mehr übersetzt werden,

```
Console.WriteLine(p1.CompareTo(13));
```

und ein gelungener **CompareTo()** - Aufruf geht ohne Boxing über die Bühne:

```
Prog::Main : void()
Suchen  Weitersuchen
IL_0034:  nop
IL_0035:  ldloca.s   p1
IL_0037:  ldloc.1
IL_0038:  call       instance int32 Punkt::CompareTo(valuetype Punkt)
IL_003d:  call       void [mscorlib]System.Console::WriteLine(int32)
```

8.3 Interfaces als Referenzdatentypen

Bei der Definition einer Schnittstelle entsteht ein neuer *Referenzdatentyp*, der in Variablendeklarationen und Formalparameterlisten einsetzbar ist. Eine Referenzvariable des neuen Typs kann auf Instanzen einer implementierenden Klasse oder Struktur zeigen, z.B.:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>
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());
   }
}
</pre></td><td><pre>
K1
K2
S
</pre></td></tr>
</table>

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.

Interface-Typen sind grundsätzlich Referenztypen, so dass eine Variable mit einem solchen Datentyp nur eine Objektadresse aufnehmen kann. Wird einer solchen Referenzvariablen eine Strukturinstanz zugewiesen, ist ein Boxing fällig.

Weil die Methoden einer Schnittstelle grundsätzlich virtuell sind, erfolgt ihr Aufruf mit dynamischer Bindung (polymorph). In zeitkritischen Programmsituationen muss eine hohe Zahl von polymorphen Methodenaufrufen und Boxing-Operationen eventuell vermieden werden.

8.4 Explizite Schnittstellenimplementierung

In den bisherigen Beispielen wurden Interface-Verpflichtungserklärungen durch **public**-deklarierte Methoden des implementierenden Typs realisiert. Bei dieser sogenannten *impliziten* Implementation kann es zu Namenskollisionen kommen, wenn …

- ein Typ mehrere Schnittstellen implementiert,
- zwei oder mehrere Schnittstellen eine Methode mit demselben Namen und derselben Parameterliste besitzen, für die aber unterschiedliche Funktionsweisen vorgesehen sind.

Für diese relativ seltene Situation bietet C# die so genannte *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;
   }
}
```

Ihre Objekte können *beide* Methoden ausführen, sofern sie über den passenden Interface-Datentyp angesprochen werden, z.B.:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
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());
   }
}
```

</td><td>

```
13
4711
```

</td></tr>
</table>

Über eine Referenzvariable von eigenen Datentyp angesprochen, ist jedoch *keine* Methode namens `M()` verfügbar, so dass die folgende Anweisung

```
Console.WriteLine(k.M());
```

den Compiler zu der Fehlermeldung veranlasst:

```
"K" enthält keine Definition für "M"
```

Würde die Klasse `K` die Methode `M()` auf bisher gewohnte Weise implementieren, wäre dasselbe Verhalten über Referenzen der Typen `I1`, `I2` und `K` abrufbar.

Weitere Hinweise zur expliziten Implementation finden sich z.B. bei Richter (2006, S. 343).

8.5 Übungsaufgaben zu Kapitel 8

1) Erstellen Sie eine Variante unserer `Bruch`-Klasse (vgl. z.B. Abschnitt 4.1.3), welche die generischen Schnittstellen **IComparable<T>** und **ICloneable** 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?

9 Einstieg in die GUI-Programmierung mit WPF-Technik

9.1 Einstimmung und Orientierung

Mit den Eigenschaften und Vorteilen einer graphischen Benutzeroberfläche (engl.: ***G**raphical **U**ser **I**nterface*) sind Sie sicher sehr gut vertraut. Eine GUI-Anwendung präsentiert dem Benutzer ein oder mehrere Fenster, die neben Bereichen zur Ausgabe bzw. Bearbeitung von programmspezifischen Dokumenten (z.B. Texten oder Grafiken) in der Regel zahlreiche normierte Bedienelemente zur Benutzerinteraktion besitzen (z.B. Menüzeile, Befehlsschalter, Kontrollkästchen, Textfelder, Auswahllisten). Die von einer Plattform (in unserem Fall also vom .NET - Framework) zur Verfügung gestellten Bedienelemente bezeichnet man oft als *Steuerelemente*, *controls* oder *widgets* (Wortkombination aus *window* und *gadgets*). Weil die Steuerelemente intuitiv (z.B. per Maus) und in verschiedenen Programmen weitgehend konsistent zu bedienen sind, erleichtern Sie den Umgang mit moderner Software erheblich.

9.1.1 Vergleich zwischen GUI- und Konsolenanwendungen

Im Vergleich zu Konsolenprogrammen geht es 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 graphischer 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, Touch Screen etc. praktisch permanent in der Lage ist, unterschiedliche Wünsche zu artikulieren. Ein GUI-Programm präsentiert mehr oder weniger viele Bedienelemente und wartet die meiste Zeit darauf, dass eine der zugehörigen Ereignisbehandlungmethoden durch ein (meistens) vom **Benutzer** ausgelöstes **Ereignis** aufgerufen wird.

Im Vergleich zu einem Konsolenprogramm ist bei einem GUI-Programm die dominante Richtung im Kontrollfluss zwischen Anwendung und Laufzeitsystem invertiert. Die Ereignisbehandlungmethoden einer GUI-Anwendung sind Beispiele für so genannte *Call Back - Routinen*. Man spricht auch vom *Hollywood-Prinzip*, weil in dieser Gegend oft nach der Divise kommuniziert wird: „*Don't call us. We call you*".

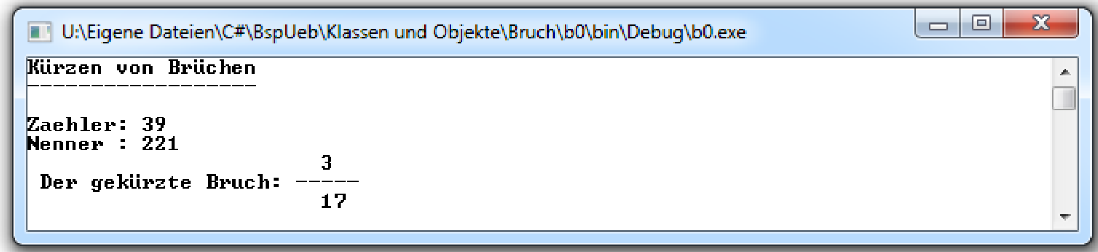
Bei vereinfachter Betrachtungsweise kann man sagen:

- Eine Konsolenanwendung diktiert den Ablauf und erlaubt dem Benutzer gelegentlich eine Eingabe.
- Eine GUI-Anwendung stellt eine Sammlung von Ereignisbehandlungmethoden dar, wobei die zugehörigen Ereignisse vom Benutzer ausgelöst werden, indem er eines der zahlreichen, für ihn verfügbaren Bedienelemente benutzt.

---

[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. in Abschnitt 12.3.3 ein Konsolenprogramm erstellten, das auf Ereignisse im Dateisystem (z.B. auf das Erstellen, Umbenennen oder Löschen von Dateien) reagiert.

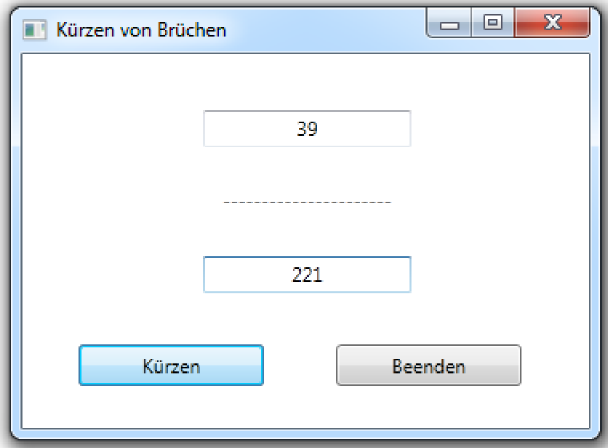
Betrachten wir zur Illustration eine Konsolen- und eine GUI-Anwendung zum Kürzen von Brüchen. Bei der Konsolenanwendung (vgl. Abschnitt 4.1.3)



wird der gesamte Ablauf vom Programm diktiert:

- Es fragt nach dem Zähler.
- Es fragt nach dem Nenner.
- Es schreibt das Ergebnis auf die Konsole.

Im Unterschied zu diesem **programmgesteuerten Ablauf** wird bei der 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**).

9.1.2 GUI-Technologien für die .NET -Plattform

Grundsätzlich ist das Erstellen einer GUI-Anwendung für Windows mit erheblichem Aufwand verbunden. Allerdings enthält die FCL (Framework Class Library) der .NET -Plattform außerordentlich leistungsfähige Klassen zur GUI-Programmierung, deren Verwendung durch Hilfsmittel der Entwicklungsumgebungen (z.B. Fensterdesigner) zusätzlich erleichtert wird.

Momentan (2011) haben .NET – Entwickler zur GUI-Gestaltung die Qual der Wahl zwischen zwei attraktiven Technologien: Das traditionelle, gut ausgestattete **Windows Forms - Framework** hat seit .NET 3.0 eine hausinterne Konkurrenz durch die von Microsoft mit großem Aufwand als neue Grafikbasis entwickelte **Windows Presentation Foundation** (ehemals als *WinFX* oder *Avalon* bezeichnet) erhalten. Als WPF - Vorteile sind vor allem zu nennen:

- Trennung von Oberflächengestaltung und Interaktionslogik durch die neue Möglichkeit, die Benutzeroberfläche in einem XML-Dialekt namens *XAML* zu deklarieren
- Bessere Verbindung von Datenbeständen und GUI-Komponenten
- Attraktive 2D- und 3D-Grafik auf vektorieller Basis (ohne Qualitätsverlust beim Skalieren)

- Einheitliche Behandlung von Steuer- und Graphikelementen (z.B. Mausklickereignisse bei Graphikelementen)
- Hardware - beschleunigte Grafikausgabe
- Multimediale Vielfalt (Audio/Video) und Animationen
- Paralleles GUI-Design für Windows-Anwendungen und Web- bzw. Browser-Anwendungen in der Microsoft-Technik Silverlight

Zwar hat Microsoft hat zugesichert, die WinForms-Plattform noch etliche Jahre zu unterstützten, doch liegt der Entwicklungsschwerpunkt eindeutig bei der Windows Presentation Foundation, so dass ich mich zum Umstieg auf dieses Framework entschieden habe. In der vorherigen Version dieses Manuskripts (Baltes-Götz 2010b) ist eine ausführliche Behandlung der WinForms-Plattform zu finden.

Wir beschränken uns im Kurs auf WPF-*Anwendungen*, die ...

- auf einem PC mit .NET - Laufzeitumgebung installiert
- und in einem eigenständigen Fenster ausgeführt werden.

Davon sind WPF-*Browseranwendungen* zu unterscheiden, die ...

- bei jedem Einsatz aus dem Internet bezogen
- und im Fenster eines WWW-Browsers ausgeführt werden, der die nötige Silverlight-Unterstützung bieten muss.

Das Visual Studio 2010 bietet (auch in der Express-Version) Projektvorlagen für selbständige und Browser-abhängige Anwendungen.

9.2 Elementare Bausteine einer WPF-Anwendung

9.2.1 Eine minimale WPF-Anwendung (ohne XAML)

Bei der rationellen Erstellung einer WPF-Anwendung verwendet man in der Regel den XML-Dialekt XAML, der eine Deklaration von (GUI-)Klassen erlaubt, zur Gestaltung der Bedienoberfläche. Zwar ist das direkte Editieren des XAML-Codes oft eine sinnvolle Option, doch überlässt man komplexe Gestaltungen und Routinearbeiten besser dem WPF-Designer im Visual Studio oder dem auf GUI-Gestaltung spezialisierten (kostenpflichtigen) Microsoft-Produkt *Expression Blend*.

Die Interaktionslogik einer WPF-Anwendung wird vom Anwendungsentwickler über C# - Quellcodedateien realisiert, die in der Regel eine *partielle* Klassendefinition enthalten. Den ergänzenden Part erstellt die Entwicklungsumgebung aufgrund des XAML-Codes. Der gesamte Prozess bei der Erstellung einer WPF-Anwendung ist durchaus nachvollziehbar, und wir werden auch einige neugierige Blicke hinter die Kulissen werfen (siehe Abschnitt 9.4.3).

Um die elementaren Klassen und Abläufe bei einer WPF-Anwendung zu analysieren, vermeiden wir aber zunächst den komplexen Entstehungsprozess via XAML und verwenden ein extrem einfaches, direkt und komplett in C# verfasstes Beispielprogramm. Wir legen im Visual Studio ein neues Projekt mit dem Namen `WpfOhneXaml` an, das die bei einer WPF-Anwendung erforderlichen Verweise auf DLL-Assemblies enthalten soll, aber keinerlei Quellcode- oder XAML-Dateien. Dazu verwenden wir die Vorlage **WPF-Anwendung** an und entfernen mit dem Projektmappen-Explorer die automatisch generierten Bestandteile:

- **App.xaml** (samt der untergeordneten Datei **App.xaml.cs**) und
- **MainWindow.xaml** (samt der untergeordneten Datei **MainWindow.xaml.cs**)
- alle Einträge im Zweig **Properties**

Im Vergleich zur Verwendung der Vorlage **Leeres Projekt** sparen wir uns die Deklaration der Assembly-Verweise und erhalten außerdem eine Anwendung mit dem Ausgabetyp Windows, so dass beim Starten kein Konsolenfenster erscheint (vgl. Abschnitt 2.2.4.2).

Wir legen im Projektmappen-Explorer über den Kontextmenübefehl

Hinzufügen > Neues Element

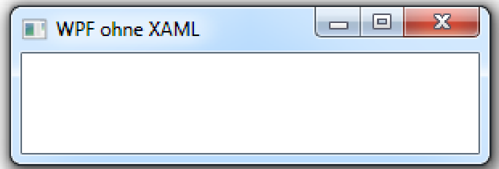
zum Projekt eine neue **Codedatei** namens **WpfOhneXaml.cs** an und definieren dort die Klasse `WpfOhneXaml` mit der Basisklasse **Window** aus dem Namensraum **System.Windows**:

```
using System.Windows;

class WpfOhneXaml : Window {
   WpfOhneXaml() {
      Title = "WPF ohne XAML";
      Width = 300;
      Height = 100;
   }

   [System.STAThreadAttribute]
   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 1.1.4). 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. Beim nun verwendeten WPF-Framework finden vermehrt Aktivitäten *hinter* der Bühne statt, die gleich erläutert werden sollen.

Bei einer WPF-Anwendung muss man die Methode **Main()** mit dem Attribut **System.STAThread-Attribute** dekorieren. Attribute sind Objekte, die mit einer speziellen Syntax (siehe Beispiel) an Klassen, Methoden etc. geheftet werden können, um Informationen für den Compiler oder die CLR bereitzustellen. Nähere Informationen folgen in Kapitel 11. Im konkreten Fall muss ein bestimmtes Threading-Modell angemeldet werden, damit die WPF-Anwendung ausgeführt werden kann.

Einen allzu großen Funktionsumfang sollte man von dem obigen Beispielprogramm nicht erwarten:



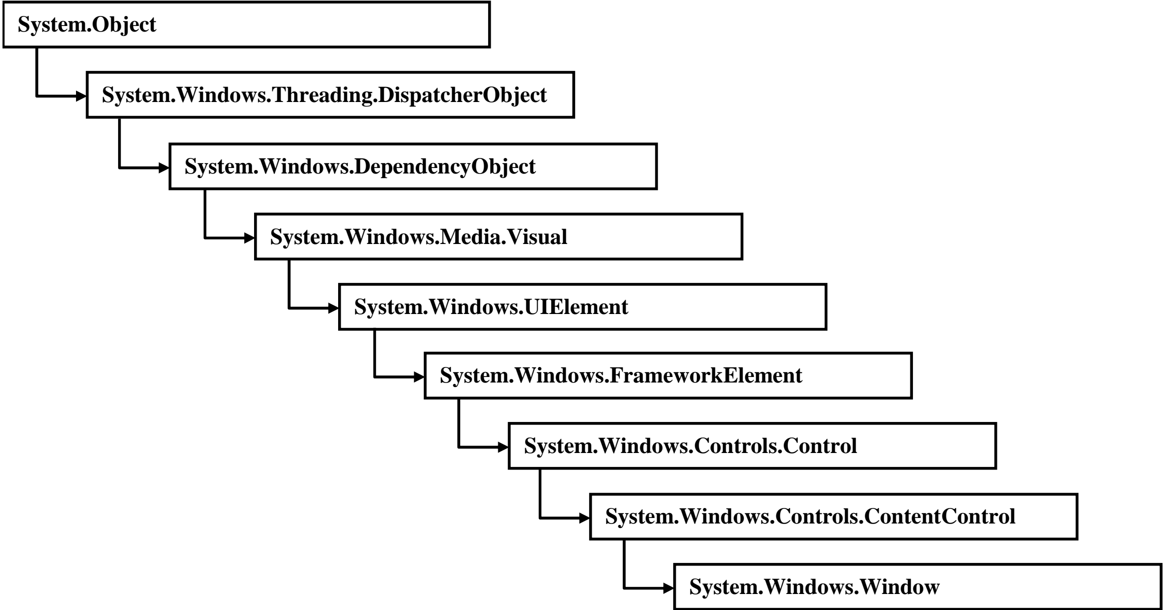
Immerhin kann man das Anwendungsfenster (dank Windows und .NET – Framework) verschieben, seine Größe ändern, die Titelzeilen-Standardschaltflächen zum Minimieren, Maximieren und Beenden benutzen usw.

Im aktuellen Kapitel 9 werden Sie folgende Klassen als wichtige Bestandteile einer WPF-Anwendung kennen lernen:

- **Window**
  Von dieser Klasse stammen alle Anwendungs- oder Dialogfenster ab.
- **Application**
  Diese Klasse stellt für eine WPF-Anwendung wichtige Dienste bereit. In der Regel definiert man eine eigene **Application**-Ableitung als Anwendungsklasse eines WPF-Programms. Ein Objekt dieser Klasse repräsentiert in gewissem Sinn die Anwendung und wird daher im Manuskript als *Anwendungsobjekt* bezeichnet.
- Delegatenklassen (z.B. **RoutedEventHandler**)
  Bei den in Abschnitt 9.3.1 behandelten Delegaten, die unsere Sammlung von C# - Datentypen vervollständigen, handelt es sich um Klassen, deren Objekte auf Methoden mit einer bestimmten Signatur zeigen.
- Klassen für Steuerelemente (z.B. **Label**, **Button**, **TextBox**) und Layoutcontainer zur Verwaltung von Steuerelementen (z.B. **Grid**, **Stackpanel**)

9.2.2 Anwendungsfenster 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:



Die Basisklassen sollen kurz skizziert werden:

- **System.Windows.Threading.DispatcherObject**
  Ein Objekt dieser Klasse ist einem **Dispatcher**-Objekt fest zugeordnet, das wiederum in einem Thread für die Verwaltung der Warteschlange zu erledigender Aufgaben (Methodenaufrufe) zuständig ist. So ist sicher gestellt, dass nur dieser eine Thread direkt auf das **DispatcherObject** zugreifen kann.
- **System.Windows.DependencyObject**
  Diese Klasse steuert die Fähigkeiten zur Behandlung von *Abhängigkeitseigenschaften* (siehe unten) bei.
- **System.Windows.Media.Visual**
  Objekte dieser Klasse besitzen elementare Graphik-Kompetenzen (z.B. Ausgabe auf dem Bildschirm, Trefferdiagnose bei Mausklicks).

- **System.Windows.UIElement**
  Diese Klasse steuert u.a. Kompetenzen bei der Ereignis- und Fokusbehandlung bei.
- **System.Windows.FrameworkElement**
  Diese Klasse implementiert viele Methoden ihrer Basisklasse, wirkt bei der logischen Struktur der WPF-Elemente (siehe unten) mit, bietet nützliche Objektlebensdauerereignisse (z.B. Hinzufügen von untergeordneten Elementen), unterstützt Datenbindung, Ressourcen, Stile und Animationen.
- **System.Windows.Controls.Control**
  Control ist die Basisklasse für viele GUI-Steuerelemente und unterstützt **Control-Template**-Objekte mit Einstellungspaketen zur Oberflächengestaltung sowie einzelne GUI-Eigenschaften wie **Background** (Hintergrundgestaltung) oder **FontFamily**.
- **System.Windows.Controls.ContentControl**
  Ein Objekt der Klasse **ContentControl** kann über seine Eigenschaft **Content** eine beliebige Instanz aufnehmen (z.B. ein GUI-Objekt oder auch eine **DateTime**-Instanz).

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) wird das `WpfOhneXaml`-Objekt `hf` zum Vertreter des **Anwendungs- oder Hauptfensters** im Programm. Unter den Fenstern eines Programms zeichnet sich das Hauptfenster durch folgende Besonderheiten aus:

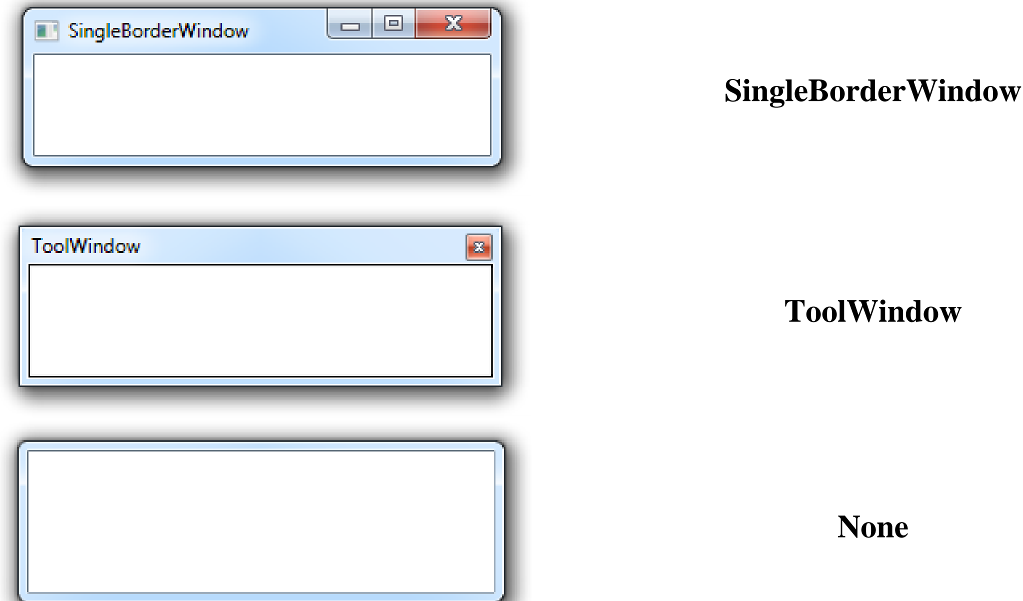
- Über den **Run()**-Aufruf oder die **Application**-Eigenschaft **StartupUri** wird für die Anzeige des Hauptfensters gesorgt. Sein Auftritt muss also *nicht* über die **Window**-Methode **Show()** veranlasst werden.
- Beim Schließen des Hauptfensters wird das Programm beendet.
- In der Regel werden erhebliche Teile der Programm-Funktionalität im Hauptfenster angeboten.

So wie im Konstruktor des Beispielprogramms mit der Anweisung

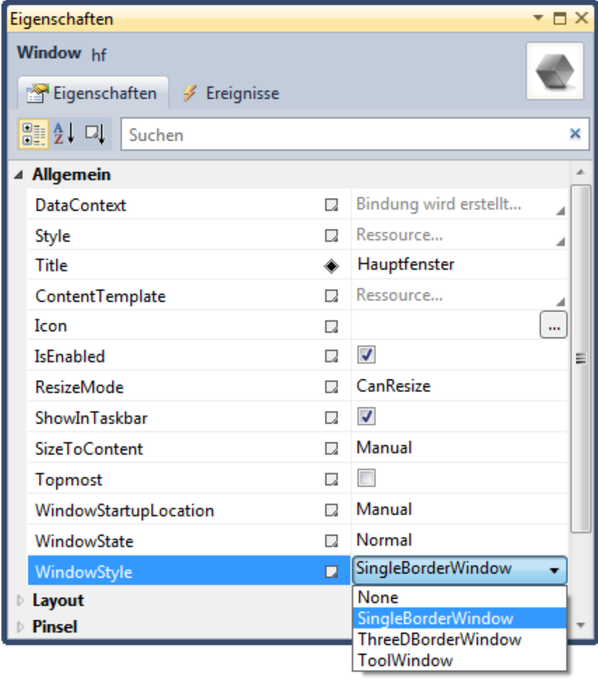
```
Title = "WPF ohne XAML";
```

die Titelzeilenbeschriftung des Fensters über die **Window**-Eigenschaft **Title** gewählt 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 Eigenschaft **WindowStyle** beeinflusst man die Ausstattung der Titelzeile mit Standardschaltflächen und die Rahmengestaltung, z.B.

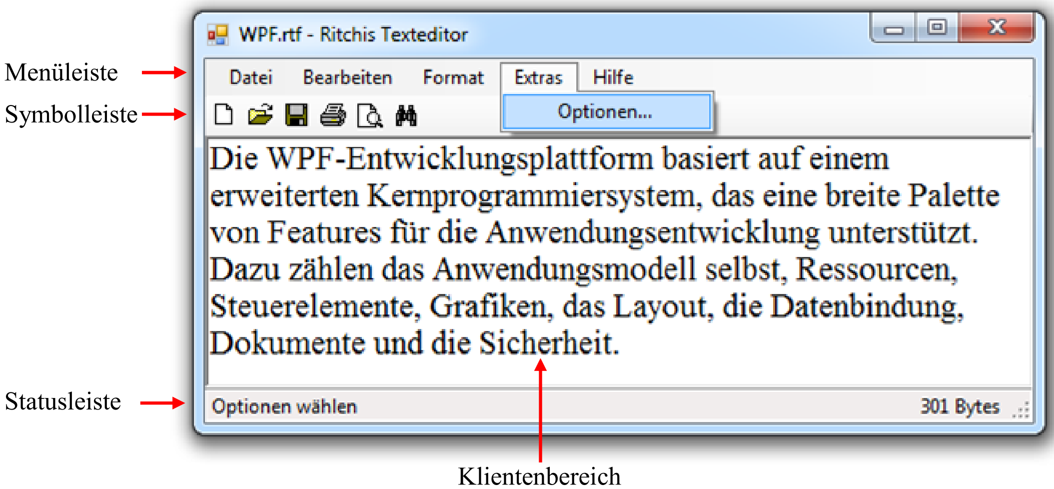


Beim Einsatz des WPF-Designers, mit dem wir schon etliche Erfahrungen gesammelt haben, erleichtert ein spezielles Fenster der Entwicklungsumgebung die Eigenschaftsmodifikation zur Entwurfszeit, z.B.:



Wir verzichten momentan auf diesen Komfort, um die Grundstruktur einer WPF-Anwendung an einem übersichtlichen Beispiel studieren zu können.

Bald werden wir die Fenster unserer Programme mit Bedienelementen wie Menüleiste, Symbolleiste und Statusleiste ausstatten, z.B.:



9.2.3 Windows-Nachrichten und die Klasse Application

Eine WPF-Anwendung wird durch ein Objekt der Klasse **Application** repräsentiert, das z.B. folgende Leistungen erbringt:

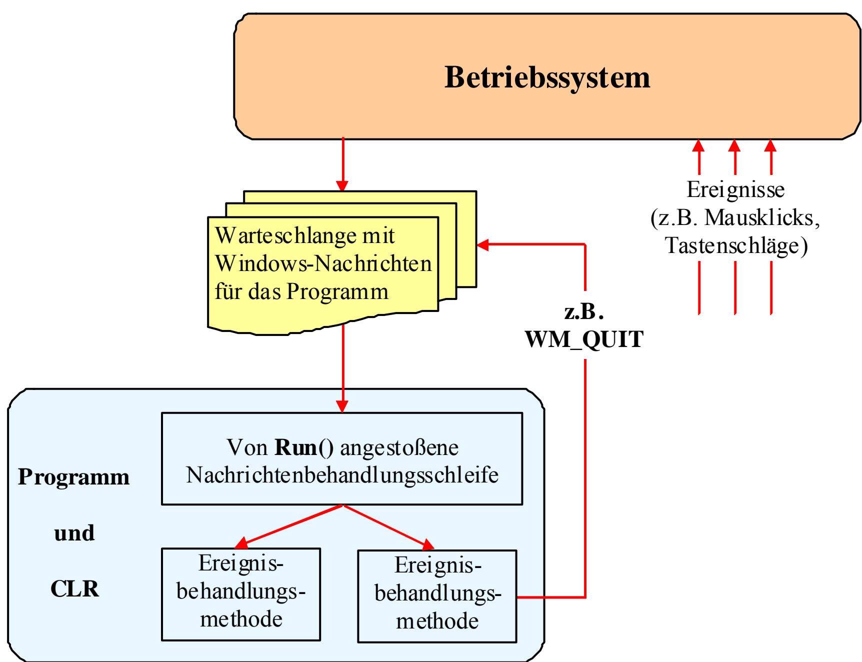
- Das **Application**-Objekt präsentiert wichtige Ereignisse in der Karriere einer Programminstanz, auf die Sie eventuell in einer Behandlungsmethode reagieren möchten (siehe Abschnitt 9.3):
    - **Startup**
      Das Ereignis wird in der Startphase ausgelöst. Eine Behandlungsmethode hat u.a. Gelegenheit, Befehlszeilenparameter auszuwerten.
    - **Activated**, **Deactivated**
      Das Programm (eines seiner Fenster) ist in den Vordergrund geholt bzw. von dort verdrängt worden.
    - **DispatcherUnhandledException**
      Es ist eine unbehandelte Ausnahme aufgetreten, die zur Beendigung des Programms führen wird. Eine Behandlungsmethode kann Widerspruch einlegen, was aber mit Bedacht geschehen sollte, weil die Fortführung eines angeschlagenen Programms riskant sein kann.
    - **SessionEnding**
      Der Benutzer ist im Begriff, sich abzumelden.
    - **Exit**
      Das Programm sieht seinem Ende entgegen.
- Die **Application**-Eigenschaft **MainWindow** zeigt auf das Hauptfenster des Programms, und die Eigenschaft **Windows** zeigt auf eine Liste aller Top-Level – Fenster.
- Die statische **Application**-Eigenschaft **Current** zeigt auf das Anwendungsobjekt.

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 (siehe unten), 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 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 Beispielprogramm z.B. auf die Standardschaltflächen in der Titelzeile (zum Maximieren, Maximieren 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 des Programms:



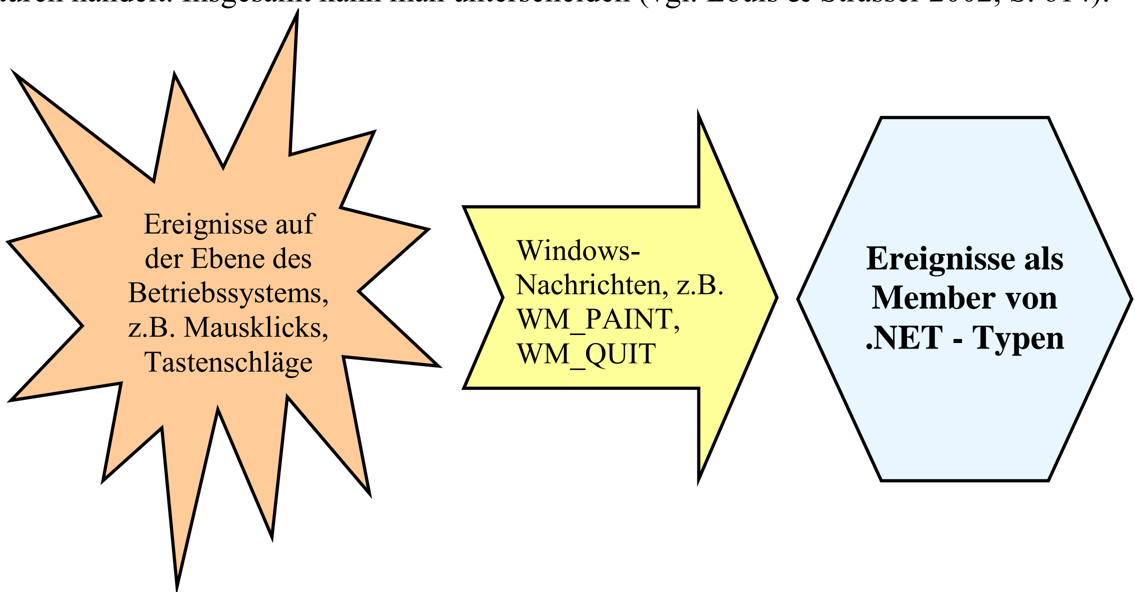
Wird die Nachricht **WM_QUIT** aus der Warteschlange gefischt, endet die Nachrichtenbehandlungsschleife, und der **Run**()-Aufruf kehrt zurück. Oft wird die Nachricht **WM_QUIT** von einer (automatisch eingerichteten) Ereignisroutine zum Hauptfenster abgeschickt, das so auf den Befehl des Benutzers zum Schließen des Fensters reagiert. Man kann die Nachricht aber auch im Programm durch Aufruf der **Application**-Methode **ShutDown**() generieren. Sind beim Eintreffen der **WM_QUIT** – Nachricht noch Fenster einer Anwendung offen, so werden diese allesamt geschlossen.

Fassen wir leicht vereinfachend zusammen, was beim Aufruf der **Application**-Methode **Run()** passiert:

- Das **Hauptfenster** erscheint.
- Das Programm[1] wird um eine **Nachrichtenschleife** erweitert, welche regelmäßig die von Windows für das Programm verwaltete Nachrichtenwarteschlange inspiziert und auf Ereignisse mit dem Aufruf der zuständigen Methode reagiert.
- Beim Eintreffen der Nachricht **WM_QUIT** enden die Nachrichtenschleife und die **Run**()-Methode, wobei auch alle Fenster der Anwendung geschlossen werden. In der Regel stellt eine GUI-Anwendung nach der Rückkehr des **Run**()-Aufrufs ihre Tätigkeit ein (siehe Beispiel). Ein erneuter Aufruf der **Run**()-Methode ist *nicht* möglich.

[1] Genau genommen ist für jeden Thread (vgl. Abschnitt 13), der ein Fenster auf dem Bildschirm präsentiert, eine Nachrichtenwarteschlange und dementsprechend eine Nachrichtenschleife erforderlich.

Weil das Windows-API (*Application Programming Interface*) durch das .NET – Framework gekapselt wird, muss sich ein C# - Programmierer nicht direkt um Windows-Nachrichten kümmern, sondern kann die von vielen Klassen präsentierten **Ereignisse** im .NET – Sinn (siehe unten) durch eigene Methoden behandeln. 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** (siehe unten) registrieren können, wenn sie auf das Ereignis (bzw. auf die zugrunde liegende Windows-Nachricht) regieren möchten. Eine registrierte Methode wird automatisch aufgerufen, wenn das zugehörige Ereignis eintritt. Wir werden uns gleich näher mit den Ereignissen im Sinne des .NET – Frameworks beschäftigen, wobei es sich um spezielle Member von Klassen oder Strukturen handelt. Insgesamt kann man unterscheiden (vgl. Louis & Strasser 2002, S. 614):



9.3 Delegaten und CLR-Ereignisse

9.3.1 Delegaten

In diesem Abschnitt lernen Sie die Delegatentypen kennen, die zunächst etwas abstrakt wirken, aber speziell im Kontext der ereignisorientierten Programmierung unverzichtbar sind. Es handelt sich um Klassen, deren Objekte auf Methoden mit einem bestimmten Rückgabetyp und einer bestimmten Formalparameterliste zeigen. Dazu enthalten Delegatenobjekte eine (ein- oder mehrelementige) Aufrufliste mit kompatiblen Methoden. Diese Liste kann Instanz- und Klassenmethoden enthalten.

Über ein Delegatenobjekt lassen sich mit *einem* Aufruf alle Methoden seiner Aufrufliste nacheinander starten, wobei die zuletzt ausgeführte Methode ggf. den Rückgabewert des Aufrufs liefert.[1]

Vielleicht helfen die folgenden Begriffserläuterungen, Missverständnisse zu vermeiden:

Ein **Delegatentyp** besitzt:

- einen Namen, z.B.
  `DelType`
- eine Signatur, z.B.
  ```
  delegate void DelType(double d, int i);
  ```

[1] Somit bietet C# mit den Delegaten eine objektorientierte Variante der Funktions-Pointer aus anderen Programmiersprachen (z.B. C++, Pascal, Modula).

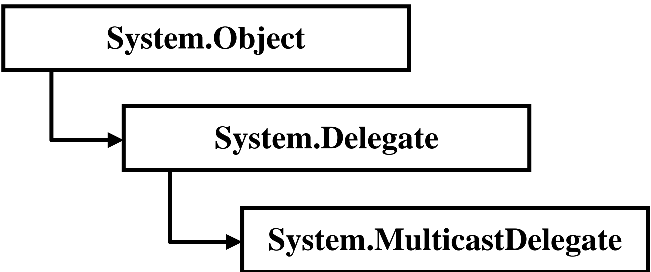
Ein **Delegatenobjekt** besitzt:

- einen Typ, z.B.
  `DelType`
- eine veränderbare Aufrufliste mit kompatiblem Instanz- und Klassenmethoden

Ein **Delegatenvariable** besitzt:

- einen Typ, z.B.
  `DelType`
- einen Wert, z.B.
  null oder
  die Adresse eines `DelType`-Delegatenobjekts

Alle Delegatentypen stammen implizit von der Klasse **MulticastDelegate** im Namensraum **System** ab:



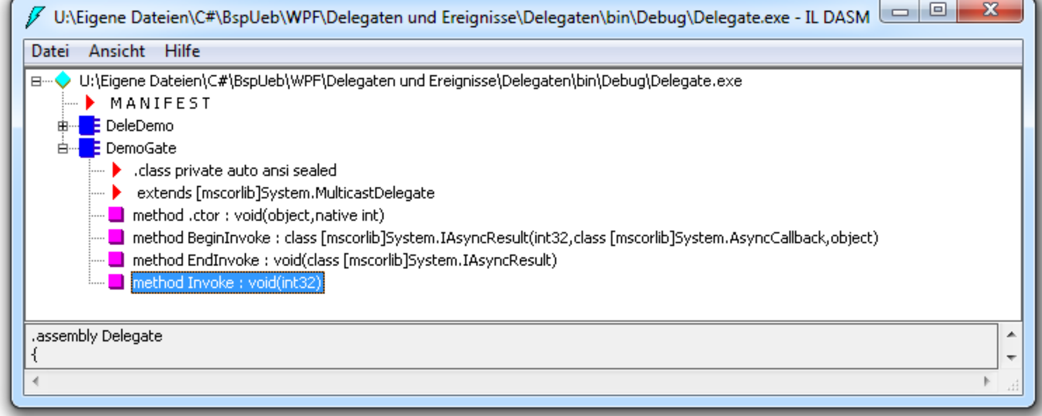
Wir verwenden Delegaten später hauptsächlich in GUI-Programmen, erarbeiten uns die neuen Begriffe aber aus didaktischen Gründen im Rahmen einer simplen Konsolenanwendung.

9.3.1.1 Delegatentypen definieren

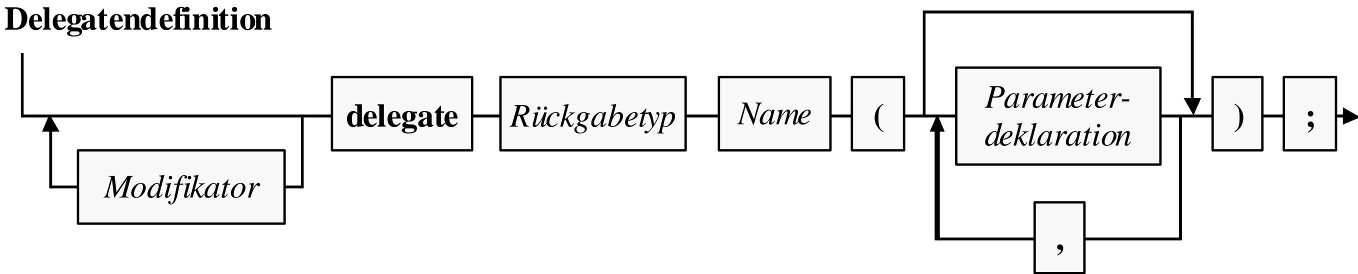
Mit der folgenden Anweisung wird nach dem einleitenden Schlüsselwort **delegate** der Delegatentyp `DemoGate` definiert:

```
delegate void DemoGate(int w);
```

Über Objekte dieses Typs können Instanz- oder Klassenmethoden mit dem Rückgabetyp **void** und einem einzigen Wertparameter vom Typ **int** aufgerufen werden. Eine Inspektion mit dem Windows-SDK - Hilfsprogramm **ILDasm** zeigt, dass die Klasse `DemoGate` aufgrund der obigen Definition u.a. einen Instanzkonstruktor und die Instanzmethode **Invoke**() besitzt:



Das leicht vereinfachte Syntaxdiagramm zur Definition eines Delegatentyps:



9.3.1.2 *Delegatenobjekte erzeugen und aufrufen*

In diesem Abschnitt wird an einem möglichst einfachen Beispiel demonstriert, …

- wie der Aufruf einer Methode,
- welche die Signatur eines bestimmten Delegatentyps besitzt,
- über ein Objekt dieses Delegatentyps erfolgen kann.

Die folgende Klasse

```
class DeleDemo {
   static void SagA(int w){
      for (int i = 1; i <= w; i++)
         Console.Write('A');
      Console.WriteLine();
   }
   static void Main() {
      DemoGate DemoVar = new DemoGate(SagA);
      DemoVar(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 in Abschnitt 9.3.1.1).
- In der **Main**()-Methode wird die lokale Referenzvariable `DemoVar` vom Delegatentyp `DemoGate` deklariert und initialisiert. Über den implizit definierten `DemoGate`-Konstruktor wird ein Objekt des Delegatentyps erzeugt, das auf die Methode `SagA()` zeigt. Die Adresse dieses Objekts landet in der Referenzvariablen:

  ```
  DemoGate DemoVar = new DemoGate(SagA)
  ```

  Einen parameterfreien `DemoGate`-Konstruktor gibt es übrigens *nicht*.

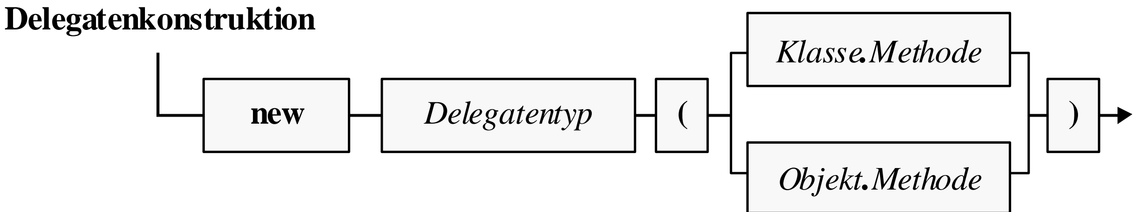
In der Aufrufliste des über die Referenzvariable `DemoVar` ansprechbaren `DemoGate`-Objekts befindet sich ausschließlich die statische Methode `SagA()`. Der Aufruf

```
DemoVar(3);
```

des Delegatenobjekts bewirkt daher die Ausgabe:

```
AAA
```

Natürlich unterstützen Delegatenobjekte nicht nur *statische* Methoden. Dem implizit definierten Konstruktor einer Delegatenklasse übergibt man als einzigen Parameter den Namen einer statischen Methode (nötigenfalls mit vorangestelltem Klasssennamen) *oder* den Namen einer Instanzmethode (nötigenfalls mit vorangestellter Objektreferenz):



Dabei wird an den Methodennamen *keine* Parameterliste angehängt.

Eine Assembly-Inspektion mit dem Windows-SDK - Hilfsprogramm **ILDasm** zeigt, dass die Anweisungen

```
DemoVar = new DemoGate(SagA);
DemoVar(3);
```

in der `DeleDemo`-Methode **Main()**

```
DeleDemo::method Main : void()
Suchen   Weitersuchen
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       61 (0x3d)
  .maxstack  4
  .locals init ([0] class DemoGate DemoVar)
  IL_0000:  nop
  IL_0001:  ldnull
  IL_0002:  ldftn      void DeleDemo::SagA(int32)
  IL_0008:  newobj     instance void DemoGate::.ctor(object,
                                                     native int)
  IL_000d:  stloc.0
  IL_000e:  ldloc.0
  IL_000f:  ldc.i4.3
  IL_0010:  callvirt   instance void DemoGate::Invoke(int32)
```

folgendes bewirken

- Es wird ein `DemoGate`-Objekt erstellt.
- Dessen Adresse landet in der Variablen `DemoVar`.
- Das Objekt wird beauftragt, die `DemoGate`-Methode **Invoke()** auszuführen.

An Stelle der expliziten Delegatenobjektkreation

```
DemoGate DemoVar = new DemoGate(SagA);
```

erlaubt der Compiler auch die Abkürzung:

```
DemoGate DemoVar = SagA;
```

9.3.1.3 Delegatenobjekte kombinieren

Nach dem Motto „Wer *A* sagt, muss auch *B* sagen“ erweitern wir die Klasse `DeleDemo` um die Methode `SagB()`:

```
static void SagB(int w){
  for (int i = 1; i <= w; i++)
    Console.Write('B');
  Console.WriteLine();
}
```

In der **Main()**-Methode ergänzen wir die Anweisung:

```
DemoVar += 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 `DemoVar` referenzierten Objekt kombiniert, wobei ein weiteres `DemoGate`-Delegatenobjekt mit einer *zweielementigen* Aufrufliste entsteht, dessen Adresse schließlich in der Referenzvariablen `DemoVar` landet. Die beiden Delegatenobjekte mit einelementiger Aufrufliste werden zu obsoletem Müll (vgl. ECMA 2006, S. 365). Delegatenobjekte sind ebenso unveränderlich wie z.B. die Objekte der Klasse **String** (vgl. Abschnitt 5.4.1.1).

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.:

```
DemoVar -= SagB;
```

Beim kompletten Entleeren der Aufrufliste entsteht aber kein leeres Delegatenobjekt, sondern die Referenzvariable erhält den Wert **null**.

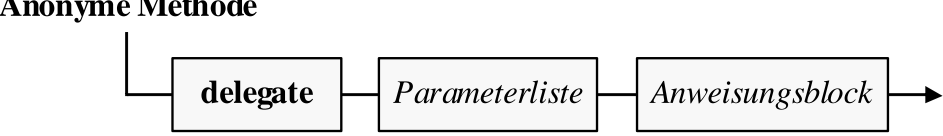
Neben den Aktualisierungsoperatoren += und -= kann man auch den Additions- und den Subtraktionsoperator verwenden, z.B.:

```
DemoVar = DemoVar + new DemoGate(SagB);
```

9.3.1.4 Anonyme Methoden

Statt beim Erzeugen eines Delegatenobjekts den Namen einer vorhandenen, andernorts definierten Methode zu übergeben, kann man seit C# 2.0 auch einen Anweisungsblock setzen, dem das Schlüsselwort **delegate** samt Delegatentyp-konformer Parameterliste vorangeht.

Anonyme Methode



Dies wird in der folgenden Variante des Beispiels aus Abschnitt 9.3.1.2 demonstriert:

Quellcode	Ausgabe

<pre>using System;
 	
delegate void DemoGate(int w);
 	
class AnoMeth {
   static void Main() {
      DemoGate DemoVar =
         delegate(int w) {
            for (int i = 1; i <= w; i++)
               Console.Write('A');
            Console.WriteLine();
         };
      DemoVar(3);
   }
}</pre>
	AAA

Man kann die gewünschte Funktionalität vor Ort realisieren und darf außerdem auf lokale Variablen und Wertparameter der umgebenden Methode zugreifen, was in der folgenden Beispielvariante mit der **Main**()-Variablen `c` passiert:

Quellcode	Ausgabe

`using System;`
`delegate void DemoGate(int w);`
`class AnoMeth {`
`   static void Main() {`
`      char c = 'A';`
`      DemoGate DemoVar =`
`         delegate(int w) {`
`            for (int i = 1; i <= w; i++)`
`               Console.Write(c);`
`            Console.WriteLine();`
`         };`
 	
`      DemoVar(3);`
`   }`
`}`
	`AAA`

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 DemoVar = delegate(int w) {return w;};`
`      Console.WriteLine(DemoVar(3));`
`   }`
`}`
	`3`

Ein Nachteil anonymer Methoden besteht darin, dass sie nicht an anderen Stellen benutzt werden können.

In C# 3.0 wurde zur Unterstützung der LINQ-Technik (siehe Kapitel 21) mit den so genannten *Lambda-Ausdrücken* eine zu den anonymen Methoden funktional äquivalente und dabei deutlich flexiblere Syntax eingeführt, die von Microsoft (2010, Abschnitt 7.15) nachdrücklich bevorzugt wird.

9.3.1.5 Generische Delegaten

Neben generischen Klassen, Strukturen, Schnittstellen und Methoden (siehe Abschnitt 7) unterstützt C# auch generische Delegaten. Das folgende Beispiel aus der FCL

```
public delegate int Comparison<T> (T x, T y);
```

kommt in einer **Sort()**-Überladung der generischen Kollektionsklasse **List<T>** (vgl. Abschnitt 7.1) 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 „Anton“ stets der Größte ist:

Quellcode	Ausgabe

<pre>using System;
using System.Collections.Generic;
class Prog {
   static int Compar(String a, String b) {
      if (a.Equals("Anton"))
         return 1;
      else
         if (b.Equals("Anton"))
            return -1;
         else
            return a.CompareTo(b);
   }
   static void Main() {
      List<String> li = new List<String>();
      li.Add("Fritz"); li.Add("Anton");
      li.Add("Anita"); li.Add("Theo");
      li.Sort(new Comparison<String>(Compar));
      foreach (String s in li)
         Console.WriteLine(s);
   }
}</pre>
	Anita
	Fritz
	Theo
	Anton

An Stelle der expliziten Delegatenobjektkreation

```
li.Sort(new Comparison<String>(Compar));
```

erlaubt der Compiler auch die Abkürzung:

```
li.Sort(Compar);
```

9.3.2 CLR-Ereignisse

Möchte eine .NET – Klasse anderen Programmeinheiten die Möglichkeit geben, zu besonderen Gelegenheiten eine Botschaft, d.h. einen bestimmten Methodenaufruf, zu erhalten, dann bietet sie ein *Ereignis* (engl.: *event*) an. Dabei handelt es sich im Wesentlichen um eine Delegatenvariable, die unter bestimmten Umständen aufgerufen wird.

In diesem Abschnitt beschäftigen wir uns mit der klassischen Ereignis-Technologie im .NET – Framework und sprechen von *CLR-Ereignissen*. In Abschnitt 9.6 kommen die neuen *Routingereignisse* der WPF zur Sprache.

9.3.2.1 Innenarchitektur von CLR-Ereignissen

Ereignisse stellen im .NET – Framework ein wichtiges Kommunikationsmittel dar. Sie ermöglichen es einer Klasse oder einem Objekt, andere Akteure darüber zu informieren, dass etwas Bestimmtes passiert ist. So kann z.B. ein Befehlsschalter-Objekt registrierte Interessenten darüber informieren, dass der Benutzer den Schalter ausgelöst (angeklickt) hat. Das zugehörige Instanzereignis der Klasse **System.Windows.Controls.Button** (siehe unten) heißt **Click**.

Obwohl Ereignisse in der Regel als **public** deklariert werden (siehe Abschnitt 9.3.2.3), resultiert in der veröffentlichenden Klasse stets eine *private* Delegatenvariable. Der Compiler ergänzt jedoch *öffentliche* Zugriffsmethoden zum Erweitern und Verkürzen der Aufrufliste durch fremde Klassen (siehe Beispiel in Abschnitt 9.3.2.3). Diese sind über die Aktualisierungsoperatoren mit dem Ereignisnamen als linkem Argument zu verwenden:

+= nimmt eine Behandlungsmethode in die Aufrufliste des zum Ereignis gehörigen Delegatenobjekts auf

-= entfernt eine Behandlungsmethode aus der Aufrufliste des zum Ereignis gehörigen Delegatenobjekts

Das öffentlich zugänglich Ereignis, das eigentlich ein Paar von Zugriffsmethoden ist, und die privaten Delegatenvariable stehen zueinander in derselben Beziehung wie eine Eigenschaft (vgl. Abschnitt 4.5) und eine zugrunde liegende Instanzvariable. Dementsprechend ähnelt die Syntax zur Definition eines CLR-Ereignisses ähnelt stark einer Eigenschaftsdefinition, wobei die Schlüsselwörter **get** und **set** zu ersetzen sind durch **add** und **remove**. Wir betrachten als Beispiel das von der in Abschnitt 9.2.3 vorgestellten WPF-Klasse **Application** definierte Ereignis **Exit**, das anderen Klassen die Möglichkeit bietet, sich über das bevorstehende (und unabwendbare) Programmende informieren zu lassen:[1]

```
public event ExitEventHandler Exit {
  add {
    VerifyAccess();
    Events.AddHandler(EVENT_EXIT, value);
  }
  remove {
    VerifyAccess();
    Events.RemoveHandler(EVENT_EXIT, value);
  }
}
```

Analog zu den automatisch implementierten Eigenschaften (siehe Abschnitt 4.5.2) kann man auch bei der Ereignisdefinition etliche Routinearbeit dem Compiler überlassen, wie das Ereignis **Activated** aus der Klasse **Application** zeigt:

```
public event EventHandler Activated;
```

Nun sieht die Ereignisdefinition aus wie die Deklaration einer Delegatenvariablen, wobei das Schlüsselwort **delegate** durch **event** ersetzt worden ist.

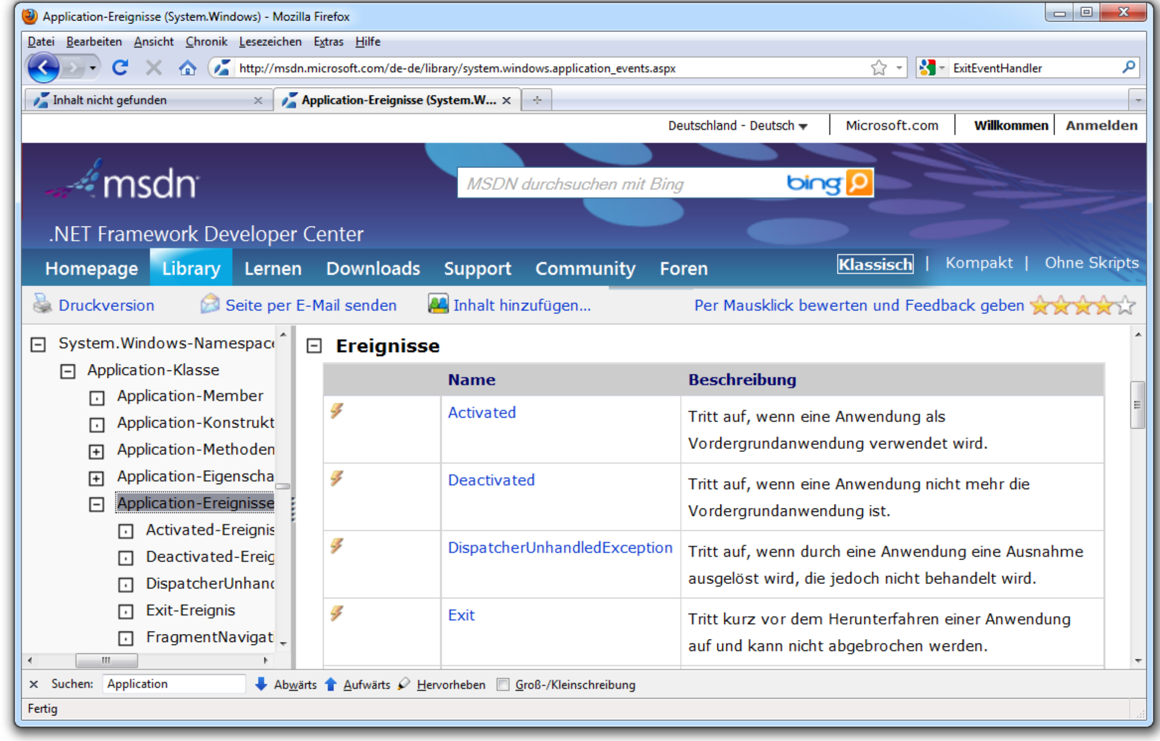
Weil die zu einem Ereignis gehörige Delegatenvariable *privat* ist, können auch abgeleitete Klassen das Ereignis *nicht* über die Delegatenvariable auslösen. In Abschnitt 9.3.2.3 wird demonstriert, wie man abgeleiteten Klassen Kontrollmöglichkeiten einräumen kann.

Näherer Informationen über die Ereignisse der Klasse **Application** finden sich in der FCL-Dokumentation:

---

[1] Der Quellcode stammt der Datei **Application.cs**, die über das *Microsoft Reference Source Code Center* (http://referencesource.microsoft.com/) bezogen wurde. **Events** ist eine private Eigenschaft der Klasse **Application**, die im Lesezugriff ein Objekt der Klasse **EventHandlerList** liefert. Dieses Kollektionsobjekt verwaltet eine Liste von Delegatenobjekten 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 gehörige Delegatenobjekt aus der Kollektion und ruft dann im Sinne von Abschnitt 9.3.1.2 das Delegatenobjekt auf:

```
protected virtual void OnExit(ExitEventArgs e) {
  VerifyAccess();
  ExitEventHandler handler = (ExitEventHandler)Events[EVENT_EXIT];
  if (handler != null) {
     handler(this, e);
  }
}
```



9.3.2.2 *Behandlungsmethoden registrieren*

Um auf ein Ereignis einer .NET - Klasse reagieren zu können, …

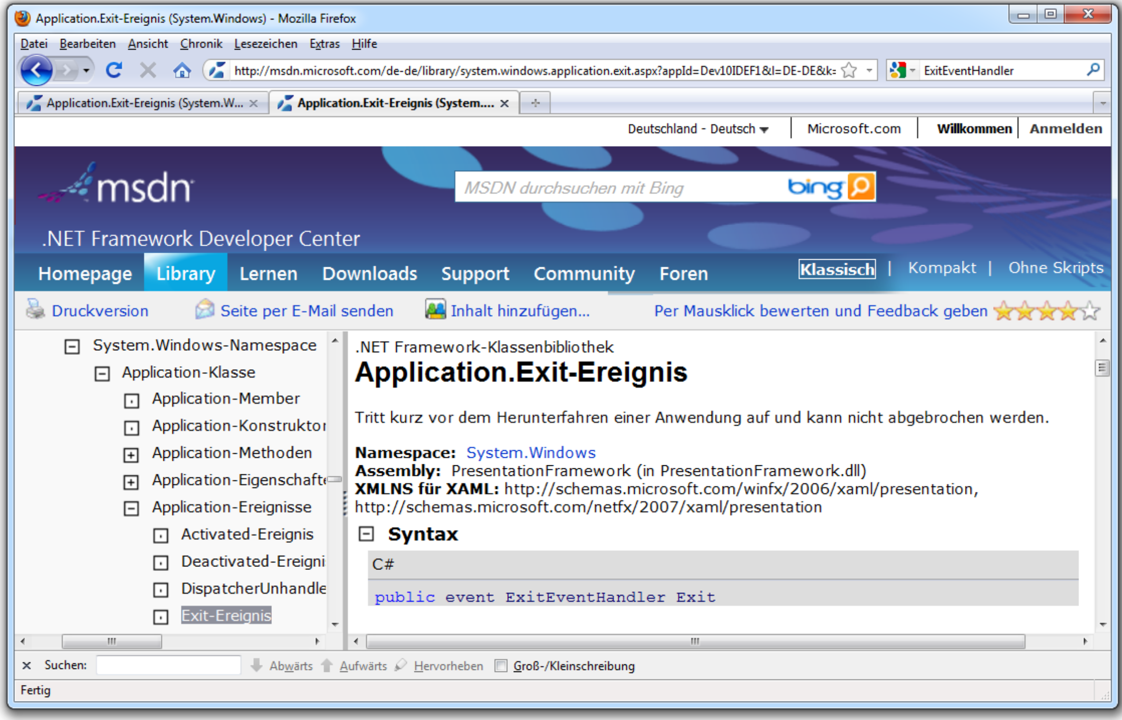
- implementiert man eine Methode, deren Signatur mit dem Delegatentyp des Ereignisses kompatibel ist,
- erzeugt man per **new**-Operator ein Delegatenobjekt, das auf diese Methode zeigt,
- weist man dem Ereignis dieses Delegatenobjekt per „+=“ - Operator zu.
  In Abschnitt 9.3.1.3 wurde im Detail beschrieben, wie bei einer solchen Zuweisung die Aufrufliste des zum Ereignis gehörigen Delegatenobjekts erweitert wird.

In der folgenden Variante des WPF-Einstiegsbeispiels aus Abschnitt 9.2 wird die Methode `AppOnExit()` implementiert und beim **Exit**-Ereignis des **Application**–Objekts zum Programm registriert, um auf das bevorstehende (und unabwendbare) Programmende reagieren zu können:

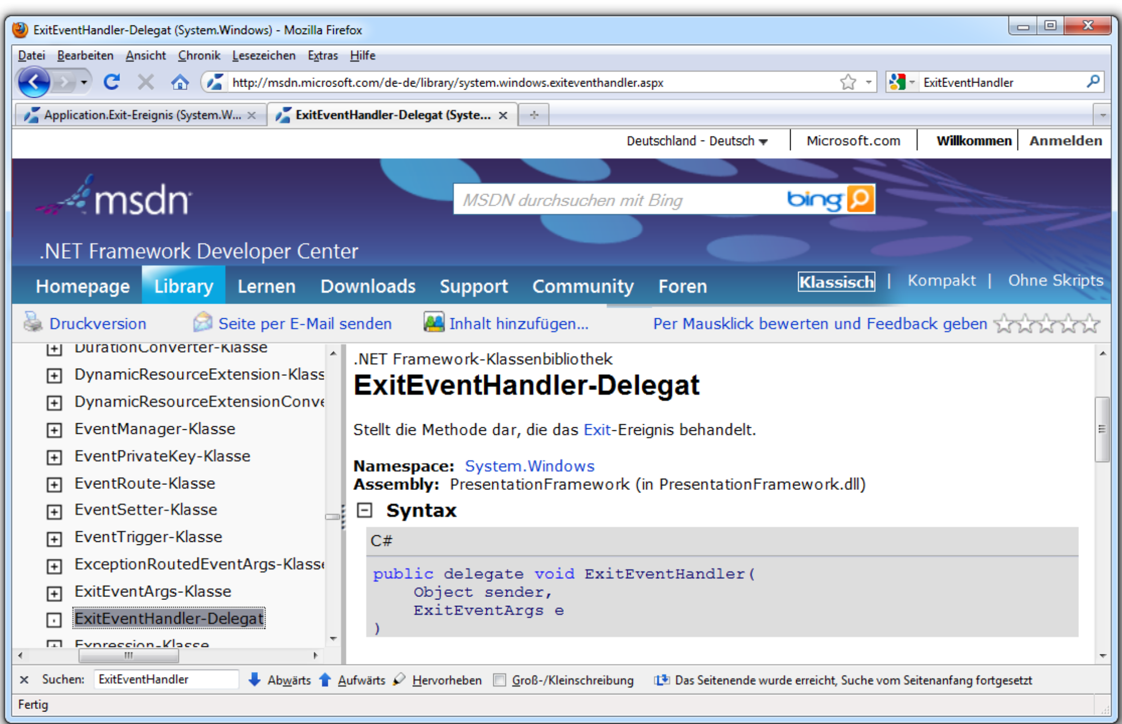
```
using System.Windows;

class ApplicationExit : Window {
   ApplicationExit() {
      Title = "Application-Ereignis Exit";
      Width = 300;
      Height = 100;
   }
   void AppOnExit(object sender, ExitEventArgs e) {
      MessageBox.Show("Vielen Dank für den Einsatz dieser Software!", Title);
   }
   [System.STAThreadAttribute]
   static void Main() {
      ApplicationExit hf = new ApplicationExit();
      Application app = new Application();
      app.Exit += new ExitEventHandler(hf.AppOnExit);
      app.Run(hf);
   }
}
```

Zu welchem Delegatentyp eine Ereignisbehandlungsmethode kompatibel sein muss, erfährt man in

der FCL-Dokumentation. Beim **Application**-Ereignis **Exit** handelt es sich um den Typ **ExitEvent-Handler**:



In der Dokumentation zum Delegatentyp **ExitEventHandler** ist zu erfahren, welchen Rückgabetyp und welche Parameter eine kompatible Methode benötigt, z.B.:



Die **ExitEventHandler**-Signatur verlangt von kompatiblen Behandlungsmethoden zwei Parameter:

- **Object**-Parameter **sender**
  Die aufgerufenen Behandlungsmethoden erhalten über diesen Parameter eine Referenz zum Ereignisemittenden. 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 Quelle (im Beispiel: Objekt App) und das Ereignis (im Beispiel: **Exit**), häufig mit dem Verbindungswort **On**.

In der folgenden Anweisung wird ein **ExitEventHandler**-Delegatenobjekt erzeugt, das auf die Methode AppOnExit() zeigt:

```
app.Exit += new ExitEventHandler(hf.AppOnExit);
```

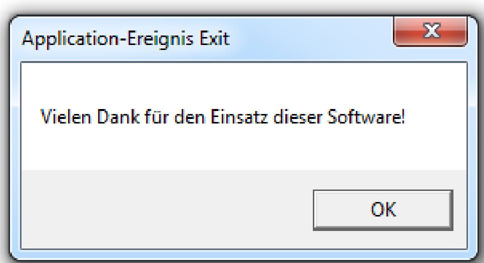
Weil es sich um eine Instanzmethode handelt, ist anzugeben, welches Objekt die Methode ausführen soll.

Statt den beteiligten Delegatenkonstruktor explizit zu notieren, kann man die folgende Abkürzung verwenden:

```
app.Exit += hf.AppOnExit;
```

Das neue **ExitEventHandler**–Delegatenobjekt wird per „+=“ – Operator beim Ereignis registriert, wobei ggf. eine vorhandene Aufrufliste verlängert wird (vgl. Abschnitt 9.3.1.3). Über den „-=“ – Operator lässt sich eine Behandlungsmethode aus der Aufrufliste zu einem Ereignis entfernen.

Im Beispiel kommt die Methode AppOnExit() z.B. zum Einsatz,



wenn der Benutzer auf das Schließkreuz in der Titelzeile des Anwendungsfensters klickt (vgl. Abschnitt 9.2.3).

Wie Sie bereits aus eigener Erfahrung wissen (vgl. z.B. Abschnitt 4.10.8), ist bei der praktischen Arbeit mit dem Visual Studio das Erstellen und Registrieren einer Ereignishandlungsmethode weniger umständlich, als die obige Beschreibung vermuten lässt. Um das Programm Application-Exit samt **Exit**-Ereignisbehandlung semiautomatisch zu erstellen, kann man so vorgehen:

- Neue **WPF-Anwendung** erstellen
- Im WPF-Designer die Hauptfenster-Eigenschaften **Titel**, **Width** und **Height** mit geeigneten Werten versorgen
- Die XAML-Datei zur Klasse App per Doppelklick auf ihren Eintrag im Projektmappen-Explorer öffnen
- Im Eigenschaftsfenster zur Registerkarte **Ereignisse** wechseln
- Doppelklick auf das Ereignis **Exit**
- Die automatisch erstellte und registrierte Ereignisbehandlungsmethode komplettieren, z.B.

```
private void Application_Exit(object sender, ExitEventArgs e) {
   MessageBox.Show("Vielen Dank für den Einsatz dieser Software!");
}
```

9.3.2.3 CLR-Ereignisse anbieten

Das Registrieren eigener Behandlungsmethoden bei Ereignissen von FCL-Klassen ist für uns erheblich relevanter als das Veröffentlichen von Ereignissen in einer eigenen Klassendefinition. Trotzdem erstellen wir ein entsprechendes Beispiel, um einen besseren Einblick in die Technik zu gewinnen. Objekte der folgenden Klasse `EventProd`, die aus der Klasse **Button** für Befehlsschalter abgeleitet ist, ermitteln nach einem Klick auf ihre Schaltfläche zehn ganze Zufallszahlen aus der Menge {0, 1, .. , 9}. Bei jedem Auftreten der Zahl 7 wird das Ereignis `Seven` ausgelöst:

```
class EventProd : System.Windows.Controls.Button {
   internal event MyDelegate Seven;
   Random zzg = new Random();
   protected override void OnClick() {
      base.OnClick();
      for (int i = 1; i <= 10; i++) {
         if (zzg.Next(10) == 7 && Seven != null) {
            MyEventArgs mea = new MyEventArgs();
            mea.Pos = i;
            Seven(this, mea);
            break;
         }
      }
   }
}
```

Warum die überschreibende Methode **OnClick()** bei einem Mausklick auf den Befehlsschalter aufgerufen wird, erfahren Sie später.

Weil wir die Standardimplementation der Methoden zum Erweitern und Reduzieren der Aufrufliste (siehe Abschnitt 9.3.2.1) nicht ändern wollen, sieht die Ereignisdefinition fast so aus wie eine Felddeklaration, wobei aber das Schlüsselwort **event** anzugeben ist:

```
internal event MyDelegate Seven;
```

Als Datentyp kommt nur ein Delegat in Frage, der im Beispiel folgendermaßen definiert wird:

```
delegate void MyDelegate(object sender, MyEventArgs e);
```

Zur Ereignisbeschreibung dient hier die Klasse `MyEventArgs`, die von **EventArgs** abgeleitet wird:

```
class MyEventArgs : EventArgs {
   public int Pos;
}
```

Man darf ein Ereignis nur dann auslösen, wenn tatsächlich ein Delegatenobjekt vorhanden ist:[1]

```
if (zzg.Next(10) == 7 && Seven != null) {
   . . .
   Seven(null, mea);
}
```

Das Delegatenobjekt zu einem Ereignis entsteht beim Registrieren der ersten Behandlungsmethode und verschwindet ggf. beim Entleeren seiner Aufrufliste.

Die folgende WPF-Anwendung (ohne XAML-Beteiligung) besitzt eine Schaltfläche aus der Klasse `EventProd` auf Ihrer Fensteroberfläche:

---

[1] Bei einer Multi Thread - Anwendung (siehe Abschnitt 13) sollte man sogar durch eine geeignete Synchronisation verhindern, dass ein Delegatenobjekt zwischen Existenzprüfung und Aufruf durch einen anderen Thread verworfen wird, z.B. mit der Anweisung:

```
Seven = null;
```

```
class EventConsumer : Window {
   EventConsumer() {
      Height = 75; Width = 284;
      Title = "Schalter sendet Ereignisse";
      StackPanel lm = new StackPanel();
      lm.VerticalAlignment = VerticalAlignment.Center;
      Content = lm;
      EventProd ep = new EventProd();
      ep.Content = "Teste Dein Glück!";
      ep.Width = 200;
      lm.Children.Add(ep);
      ep.Seven += new MyDelegate(this.EventProdOnSeven);
   }

   void EventProdOnSeven(object sender, MyEventArgs e) {
      MessageBox.Show("7 an Position " + e.Pos + " gezogen",
        "Ereignis-Verarbeitung");
   }

   [System.STAThreadAttribute]
   static void Main() {
      new Application().Run(new EventConsumer());
   }
}
```

Im Konstruktor der Klasse `EventConsumer` ist ansonsten zu sehen, wie die bei einem WPF-Fenster beteiligten Objekte instantiiert und initialisiert werden:

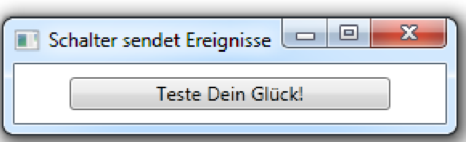
- Das Fenster verwendet zur Verwaltung seiner Steuerelemente einen Layoutcontainer aus der Klasse **StackPanel** (siehe Abschnitt 9.7.2):
  ```
  StackPanel lm = new StackPanel();
  lm.VerticalAlignment = VerticalAlignment.Center;
  ```
  Ein WPF-Layoutcontainer ist z.B. dafür zuständig, bei einer Änderung der Fenstergröße die Positionen und Ausdehnungen der verwalteten Steuerelemente anzupassen.
- Das **StackPanel**-Objekt erhält seine Rolle, indem es der **Content**-Eigenschaft des Fensterobjekts zugewiesen wird:
  ```
  Content = lm;
  ```
- Das Befehlschalter-Objekt `ep` aus der Klasse `EventProd` wird mit folgender Anweisung dem Layoutcontainer `lm` aus der Klasse **StackPanel** anvertraut:
  ```
  lm.Children.Add(ep);
  ```
  Der **Add()**-Methodenaufruf richtet sich über die **StackPanel**-Eigenschaft **Children** an ein Objekt der Klasse **UIElementCollection**, das eine Liste von Objekten der Klasse **UIElement** verwaltet.

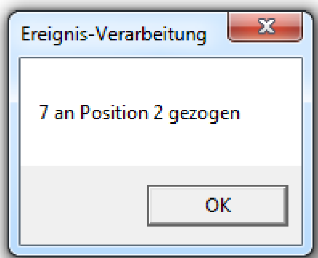
Bei der Anwendungsentwicklung mit XAML-Unterstützung finden solche Instantiierungen und Initialisierungen hinter den Kulissen statt (siehe Abschnitt 9.4.4).

Um auf das Ereignis `Seven` reagieren zu können, implementiert die Klasse `EventConsumer` eine Methode mit kompatibler Signatur und registriert diese beim Ereignis:

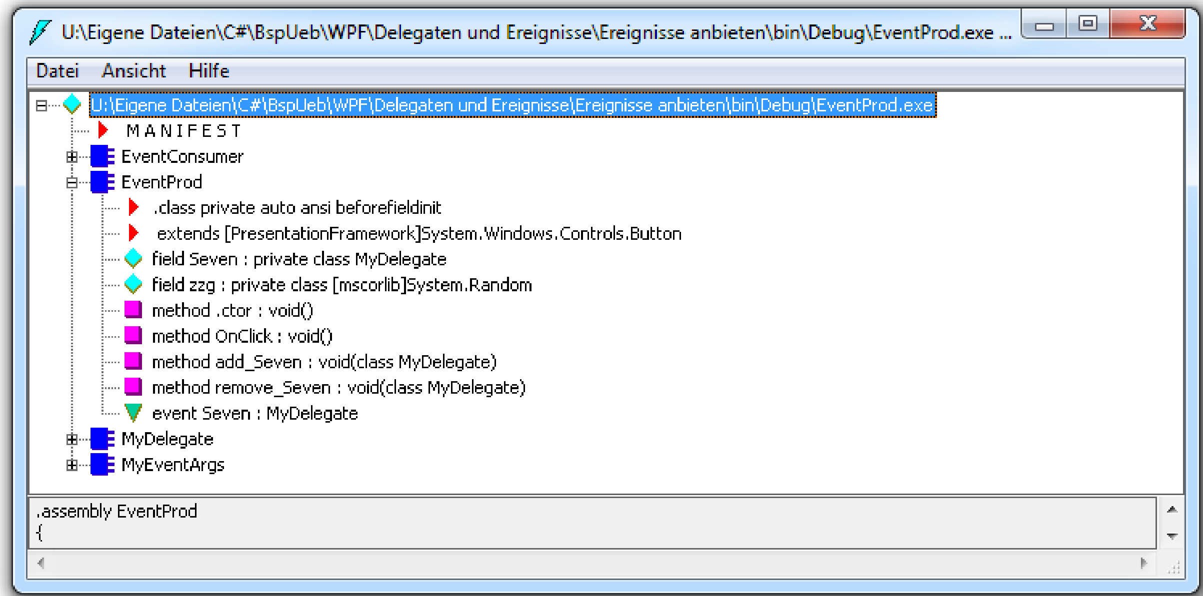
```
ep.Seven += new SimpleDelegate(this.EventProdOnSeven);
```

Bei seiner Reaktion auf die Benachrichtigung wertet der Konsument auch die Ereignisbeschreibung im `MyEventArgs`–Objekt aus, das der Methode `EventProdOnSeven()` beim Aufruf übergeben wird, z.B.:





Eine Assembly-Inspektion mit dem Windows-SDK - Hilfsprogramm **ILDasm** bestätigt die zu Beginn von Abschnitt 9.3.2 formulierte Bemerkung über die Ereignis-Innenarchitektur. In der Klasse `EventProd` befinden sich aufgrund der `Seven`-Ereignisdefinition neben einem privaten Feld vom Typ `MyDelegate` zwei öffentliche Methoden zur Veränderung der Ereignisaufrufliste (`add_Seven()` und `remove_Seven()`):



Wie zu Beginn von Abschnitt 9.3.2 beschrieben, hat auch bei einem Ereignis mit Schutzstufe **public** die beim Auslösen zu verwendende Delegatenvariable stets die Schutzstufe **private**. Damit könne auch abgeleitete Klassen das Ereignis *nicht* über die Delegatenvariable auslösen, was in vielen Situationen durchaus erwünscht ist. Um den abgeleiteten Klassen Kontrollmöglichkeiten einzuräumen, kann man ...

- das Ereignis in einer Methode mit der Schutzstufe **protected** auslösen
  Damit haben abgeleitete Klassen die Möglichkeit, das Ereignis auszulösen.
- die auslösende Methode als virtuell (überschreibbar) definieren
  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 `EventProd` vor:

```
class EventProd : System.Windows.Controls.Button {
   internal event MyDelegate Seven;
   Random zzg = new Random();
   protected override void OnClick() {
      base.OnClick();
      for (int i = 1; i <= 10; i++) {
         if (zzg.Next(10) == 7) {
            MyEventArgs mea = new MyEventArgs();
            mea.Pos = i;
            OnSeven(mea);
            break;
         }
      }
   }
   protected virtual void OnSeven(MyEventArgs mea) {
      Seven(this, mea);
   }
}
```

Im Namen einer Ereignis-auslösenden Methode gibt man in der Regel mach den Präfix **On** das Ereignis an.

Nun kann eine abgeleitete Klasse bei der Ereignisentstehung mitreden und z.B. einen Treffer bei späten Versuchen (`mea.Pos > 5`) für ungültig erklären:

```
class EventProdD : EventProd {
   protected override void OnSeven(MyEventArgs mea) {
      if (mea.Pos <= 5)
         base.OnSeven(mea);
   }
}
```

9.4 Die Extended Application Markup Language (XAML)

Die Extended Application Markup Language (XAML) wurde von Microsoft als XML-Dialekt zur Unterstützung des .NET – Frameworks (ab Version 3.0) entwickelt, wobei der bisherige Einsatzschwerpunkt die Deklaration der Benutzeroberfläche zu einer WPF-Anwendung ist. Während im GUI-Framework *WinForms* die Funktionalität *und* die Gestaltung einer GUI-Anwendung im Quellcode der jeweiligen .NET – Programmiersprache (z.B. C#) zu realisieren sind, hat Microsoft für die WPF-Entwicklung die deklarative Sprache XAML mit folgenden Zielsetzungen entwickelt:

- Vereinfachung der GUI-Gestaltung
- Trennung von Design und Interaktionslogik

Damit bestehen gute Voraussetzungen für die erfolgreiche Zusammenarbeit für Designern und Software-Entwicklern.

9.4.1 Elementare Regeln zum Aufbau einer XML-Datei

Die *eXtended Markup Language* (XML) hat sich als universelles Mittel zur Deklaration von strukturieren Daten etabliert und wird auch im .NET – Framework intensiv verwendet. Besonders auffällig sind:

- Anwendungs- und GUI-Deklaration per XAML
- Anwendungs- und benutzerbezogene 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 eine Deklaration des Dokumententyps, die meist eine Versions- und eine Kodierungsangabe enthält, z.B.

```
<?xml version="1.0" encoding="utf-8" ?>
```

Bei dem im aktuellen Kapitel vornehmlich relevanten Dialekt XAML wird diese Zeile allerdings 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)

Als Bestandteile seiner Startkennung kann ein Element beliebig viele **Attribute** enthalten, die aus Name-Wert - Paaren bestehen. 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 Orientierung des Steuerelements im umgebenden Steuerelement-Container):

```
<Button Name="button1" HorizontalAlignment="Center" VerticalAlignment="Center">
  . . .
</Button>
```

Ein Element *ohne* untergeordnete Elemente (oder sonstige Inhalte) kann sich auf die Startkennung beschränken, die dann mit einer Sequenz aus Leerzeichen und Schrägstrich zu enden hat, z.B. beim folgenden **Image** - Element aus einer XAML-Datei:

```
<Image Source="/Routing;component/rss.gif" VerticalAlignment="Center" />
```

Wie beim C# - Quellcode ist auch beim XML-Code die **Groß-/Kleinschreibung relevant** (mit sehr seltenen Ausnahmen, auf die bei Gelegenheit hingewiesen wird).

Das folgende Beispiel zeigt, wie **Kommentare** in einer XML-Datei untergebracht werden:

```
<!--- Kommentar -->
```

9.4.2 XAML-Kurzbeschreibung

Das Erstellen und Initialisieren der Objekte (Komponenten) der Bedienoberfläche kann auch bei einer WPF-Anwendung per Quellcode erfolgen. Microsoft empfiehlt jedoch, für diesen Zweck den eigens entwickelten XML-Dialekt *Extended Application Markup Language* (XAML) zu verwenden. Bei der von Microsoft empfohlenen WPF-Projektorganisation sind folgenden XAML-Dateien vorhanden:

- Ein XAML-Datei zur Anwendungsklasse
- Für jedes Fenster (jede Fensterklasse) eine XAML-Datei zur Deklaration der GUI-Elemente

Diese Dateien verwenden die UTF-8 – Kodierung und die Namenserweiterung **.xaml**.

9.4.2.1 Wurzelelement

Eine XAML-Datei einhält nur *ein* Wurzelelement, wobei für uns derzeit in Frage kommen:

- **Application**
  Dieses Wurzelelement wird in der XAML-Datei mit der Anwendungsdeklaration verwendet. Es nennt auf jeden Fall die Anwendungsklasse und die XAML-Datei zum Hauptfenster. Außerdem können Behandlungsmethoden zu Ereignissen der Anwendungsklasse und Ressourcen deklariert werden. Als Name der zugehörigen XAML-Datei wird meist **App.xaml** verwendet. Für eine neue WPF-Anwendung erzeugt das Visual Studio 2010 die folgende Datei:

```
<Application x:Class="Mapp.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             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 wählt das Visual Studio 2010 den Dateinamen **MainWindow.xaml**.

Die Startkennung zu einem XAML - Wurzelelement enthält in jedem Fall zwei **xmlns**-Attribute zur Deklaration von **XML-Namensräumen** über die Syntax:

xlmns[**:***prefix*]="*URI*"

Die angegebenen Quellen enthalten die Definitionen der im XAML-Code erlaubten Datentypen. Zur Vermeidung von Namenskollisionen kann ein Präfix angegeben werden. Ohne Präfixangabe wird ein Namensraum zur Voreinstellung für die XAML-Datei. Das geschieht beim Standard für ein neues WPF-Projekt im Visual Studio 2010 mit dem Namensraum für WPF-bezogene Elemente:

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

Der XML-Namensraum wird mit dem Präfix **x** angemeldet

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Die folgenden **x**-Attribute werden oft benötigt:

- **x:Class**
  Im **x:Class**-Attribut wird die zur XAML-Datei gehörige Klasse samt Namensraum genannt.
- **x:Key**
  Die so genannten Ressourcen, die in einer WPF-Anwendung auf einfache Weise die Mehrfachverwendung von Objekten erlauben, erhalten über dieses Attribut einen Schlüssel, unter dem sie später ansprechbar sind.
- **x:Name**
  Über dieses Attribut legt man für die Instanz, die aus der XAML-Laufzeitverarbeitung eines XAML-Elements entsteht, einen Namen fest. Es resultiert eine Instanzvariable mit dem gewählten Namen, die in der Code-Behind – Datei (siehe Abschnitt 9.4.3) einen Instanzzugriff erlaubt. Viele Klassen (z.B. **Button**, **StackPanel**) besitzen eine **Name**-Instanzeigenschaft, die an Stelle von **x:Name** verwendet werden kann.

Beim Erstellen und Initialisieren von Objekten per XAML-Code muss man sich nicht auf WPF-Klassen beschränkt. Stattdessen werden beliebige .NET – Klassen unterstützt, sofern diese über einen parameterlosen und öffentlichen Konstruktor verfügen. Um beliebe CLR-Typen im XAML-Code bequem ansprechen zu können, bezieht man den zuständigen Namensraum ein und wählt dabei ein passendes Präfix, z.B.:

```
<Window x:Class="Mapp.MeinWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:myspace="clr-namespace:Mapp"
        Title="MeinWindow" Height="350" Width="525">
    <Grid>
    </Grid>
</Window>
```

9.4.2.2 Instanzelemente

Ein Instanzelement deklariert eine Instanz (z.B. ein Objekt) zu 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 vorhanden, beendet man die Startkennung mit einer schließenden spitzen Klammer, lässt man die untergeordneten Elemente folgen und setzt eine Endkennung, die zwischen einem Paar spitzer Klammern einen Schrägstrich und den Element- bzw. Typnamen enthält, z.B.:

```
<StackPanel Name="stackPanel1" Orientation="Horizontal">
    <Image Width="20" Source="/Routing;component/rss.gif" VerticalAlignment="Center"/>
    <TextBlock Text="Click to Feed" Margin="5" VerticalAlignment="Center" />
</StackPanel>
```

Ein Instanzelement stellt einen Auftrag an die XAML-Verarbeitung dar, eine Instanz vom angegebenen Typ zu erstellen. Das tatsächliche Erstellen findet zur Laufzeit durch einen Aufruf des parameterlosen Konstruktors zum jeweiligen Typ statt (siehe Abschnitt 9.4.4).

9.4.2.3 Eigenschaftswerte per Attribut oder Eigenschaftselement zuweisen

Viele Instanzeigenschaften können in der Startkennung über die **XML-Attributsyntax** einen Wert erhalten, wobei auf den Namen der Eigenschaft das „="-Zeichen und durch doppelten Anführungszeichen begrenzt eine Zeichenfolge als Wert folgt, z.B.:

```
<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 Beispiel muss aus der Zeichenfolge zum Attribut **Margin**, das einem Steuerelement Abstand zur Umgebung verschafft, eine Instanz vom Typ **System.Windows.Thickness** entstehen.

Die Attributsyntax eignet sich *nicht*, wenn einer Eigenschaft ein komplexes Objekt zugewiesen werden soll, das kein Typkonverter aus einer Zeichenfolge erzeugen kann. 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 ein **StackPanel**-Layoutobjekt zur Verwaltung der **Button**-Oberfläche (siehe unten) zugewiesen:

```
<Button Name="button1" Background="WhiteSmoke">
   <Button.Content>
      <StackPanel Orientation="Horizontal">
          . . .
      </StackPanel>
   </Button.Content>
</Button>
```

Gleich wird sich allerdings herausstellen, dass bei der **Button**-Eigenschaft **Content** eine Vereinfachung möglich ist. Weil diese Eigenschaft bei der Klasse **Button** sehr oft angesprochen werden muss, ist sie als XAML-Inhaltseigenschaft (siehe unten) für **Button**-Elemente festgelegt worden, so dass man auf die Startkennung

```
<Button.Content>
```

und die zugehörige Endkennung verzichten kann.

Die Attributsyntax ist letztlich nur eine Kurzform der Eigenschaftselementsyntax. Z.B. kann man statt

```
<Button Name="button1" Content ="Click to Feed" Background="WhiteSmoke"/>
```

auch schreiben:

```
<Button Name="button1" Content ="Click to Feed">
   <Button.Background>
       <SolidColorBrush Color="WhiteSmoke"/>
   </Button.Background>
</Button>
```

9.4.2.4 Vereinfachte Wertzuweisung bei der Inhaltseigenschaft

Jede Klasse kann von ihren Eigenschaften genau eine als **XAML-Inhaltseigenschaft** deklarieren.[1] Ist ein Eigenschaftselement von diesem Typ anzugeben, kann auf das Kennungspaar gemäß XAML-Eigenschaftselementsyntax verzichtet werden. Folglich kann das obige Beispiel mit einer Deklaration für die **Button**-Eigenschaft **Content**

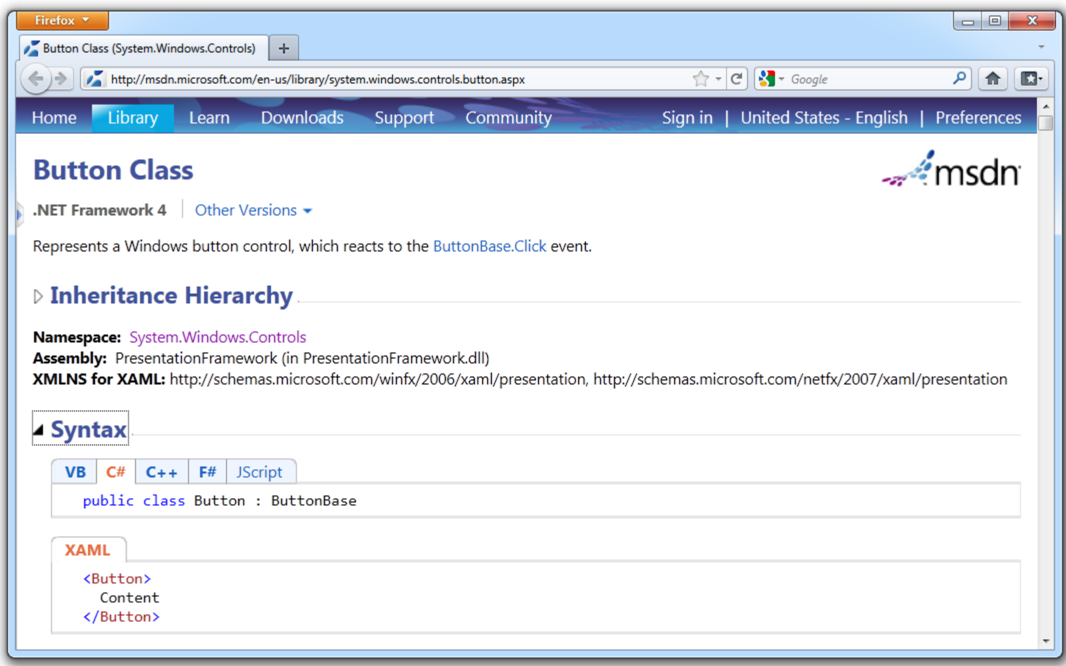
```
<Button Name="button1" Background="WhiteSmoke">
   <Button.Content>
      <StackPanel Orientation="Horizontal">
          . .  .
      </StackPanel>
   </Button.Content>
</Button>
```

einfacher geschrieben werden:

```
<Button Name="button1" Background="WhiteSmoke">
   <StackPanel Orientation="Horizontal">
       . .  .
   </StackPanel>
</Button>
```

In der Dokumentation zu einer WPF-Klasse ist die XAML-Inhaltseigenschaft ggf. an prominenter Stelle ausgewiesen, z.B. bei der Klasse **Button**:

[1] Dazu wird der Klasse das Attribut **DefaultProperty** angeheftet, wobei hier der Begriff *Attribut* im Sinne von Kapitel 11 zu verstehen ist.



Bei der Layoutcontainer-Klasse **StackPanel** ist die (von **Panel** geerbte) Eigenschaft **Children** als XAML-Inhaltseigenschaft deklariert:

```
XAML
<StackPanel>
  Children
</StackPanel>
```

Folglich kann ggf. das Kennungspaar

```
<StackPanel.Children> . . . </StackPanel.Children>
```

weggelassen werden. Weil die Eigenschaft **Children** auf ein Objekt der Klasse **UIElement-Collection** zeigt, ist bei der Wertzuweisung im Wesentlichen eine Liste mit **UIElement**-Objekten anzugeben. In solchen Fällen erlaubt es die **XAML-Auflistungssyntax**, auf ein Instanzelement zum Kollektionsobjekt zu verzichten. In diesem Beispiel

```
<StackPanel Orientation="Horizontal">
   <Image Width="20" Source="/Routing;component/rss.gif" />
   <TextBlock HorizontalAlignment="Right" Text="Click to Feed" />
</StackPanel>
```

werden zwei XAML-Vereinfachungsmöglichkeiten genutzt:

- Inhaltseigenschaft
  Das Eigenschaftselement **StackPanel.Children** wurde weggelassen.
- Auflistungssyntax
  Das **UIElementCollection**-Instanzelement wurde weggelassen.[1]

9.4.2.5 Eine Zeichenfolge als Wert für die Inhaltseigenschaft

Bei manchen XAML-Elementen kann der Parser eine Zeichenfolge verarbeiten, wenn er einen Wert für die Inhaltseigenschaft erwartet, z.B.:

---

[1] Weil die Klasse **UIElementCollection** keinen öffentlichen und parameterfreien Konstruktor anbietet, wäre ein Instanzelement von diesem Typ ohnehin fehlerhaft.

```
<Button Name="button1" Background="WhiteSmoke">
   Click to Feed
</Button>
```

Als Voraussetzung für die Verwendung einer Zeichenfolge als Inhalt muss für den Typ des Instanzelements (im Beispiel: **Button**) eine XAML-Inhaltseigenschaft deklariert sein (im Beispiel: **Content**). Außerdem muss für den Typ dieser Inhaltseigenschaft eine von den folgenden Bedingungen erfüllt sein:[1]

- Dem Typ kann eine Zeichenfolge zugewiesen werden, was z.B. auch beim Typ **Object** (gleich Typ der **Button**–Eigenschaft **Content**) möglich ist.
- Für den Typ ist eine Klasse mit entsprechenden Kompetenzen als Typkonverter definiert.
- Der Typ ist im XAML-Sprachumfang bekannt, was z.B. bei vielen elementaren Datentypen (z.B. **Int32**, **Int64**, **Double**) der Fall ist.

9.4.2.6 Attributsyntax für Ereignisse

Wie die folgende Startkennung zum **Application**-Element aus der XAML-Datei zu einer WPF-*Anwendung* (voreingestellter Dateiname: **App.xaml**) 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 9.4.3).

9.4.2.7 Attributsyntax für Markuperweiterungen

Über die so genannten Markuperweiterungen kann einer Eigenschaft auch per Attributsyntax ein Objekt zugewiesen werden. Man kommt also ohne die etwas umständliche Eigenschaftselementsyntax aus und ist gleichzeitig nicht auf einen Typkonverter angewiesen (siehe Abschnitt 9.4.2.3). Es kommt eine von geschweiften Klammern eingerahmte Zeichenfolge zum Einsatz, die eine spezielle Interpretation erfährt, z.B.:

```
Background="{StaticResource BgColor}"
```

Das Beispiel demonstriert die in WPF-Anwendungen häufig anzutreffende Verwendung einer statischen Ressource per Markuperweiterung. Dies ermöglicht auf einfache Weise die Mehrfachverwendung von Objekten. Zur Komplettversion des Beispiels gehört die folgende Ressourcen-Definition in der XAML-Datei zur Anwendungsklasse (mit **Application**-Wurzelelement):

```
<Application x:Class= ... >
    <Application.Resources>
        <SolidColorBrush x:Key="BgColor" Color="WhiteSmoke"/>
    </Application.Resources>
</Application>
```

Im Ergebnis steht anwendungsglobal ein Objekt der Klasse **SolidColorBrush** als Ressource mit dem Schlüssel `BgColor` zur Verfügung. In der folgenden XAML-Datei zum Hauptfenster dersel-

---

[1] Übernommen von der Webbseite http://msdn.microsoft.com/de-de/library/ms752059.aspx

ben Anwendung wird das **SolidColorBrush**–Objekt per Markuperweiterung der **Window**-Eigenschaft **Background** als statische Ressource zugewiesen:

```
<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>
```

Syntaxregeln für Markuperweiterungen:

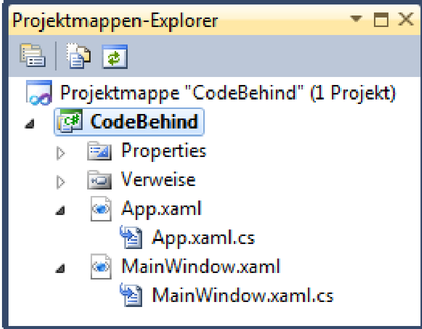
- Man kann eine Liste von Name-Wert – Paare oder eine Liste von Werten angeben, wobei in jedem Fall die Listenelemente durch Kommata zu trennen sind.
- Bei Verwendung von Name-Wert – Paaren (mit Gleichheitszeichen dazwischen) kommt der parameterlose Konstruktor zum Einsatz.
- Bei Verwendung von Werten (ohne Gleichheitszeichen dazwischen) kommt ein parametrisierter Konstruktor zum Einsatz, so dass alle Werte den korrekten Typ und die korrekte Position haben müssen.

9.4.3 Code-Behind -Dateien

Ein vom Visual Studio 2010 über die Vorlage **WPF-Anwendung** 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 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 Interaktionslogik der Anwendung zuständig, werden im Wesentlichen vom Entwickler (also von Ihnen!) erstellt und als *Code-Behind – Dateien* bezeichnet.

Sie enthalten nach etlichen **using**-Direktiven zum Import von .NET - Namensräumen jeweils eine partielle Klassendefinition im Namensraum des Projekts:

- **App.xaml.cs**
  In der Datei **App.xaml.cs** findet sich eine partielle Definition der (vom Visual Studio so getauften) Anwendungslasse App, die von der FCL-Klasse **Application** (vgl. Abschnitt 9.2.3) abstammt:

  ```
  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 Sie im Visual Studio eine Ereignisbehandlung zum Anwendungsobjekt anfordern, 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** erhalten Sie z.B. so:

- Datei **App.xaml** per Doppelklick auf den Eintrag im Projektmappen-Explorer öffnen
- Im Eigenschaftenfenster die Registerkarte **Ereignisse** wählen
- Doppelklick auf das Ereignis **Exit**

Dabei erhält in der zugehörigen XAML-Datei das Wurzelelement **Application** einen entsprechenden Wert für das Attribut **Exit**:

```
<Application x:Class="CodeBehind.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">
```

- **MainWindow.xaml.cs**

  In der Datei **MainWindow.xaml.cs** findet sich eine partielle Definition der (vom Visual Studio so getauften) Anwendungslasse `MainWindow`, die von der FCL-Klasse **Window** (vgl. Abschnitt 9.2.2) abstammt:

```
using System;
  . . .
namespace CodeBehind {
  /// <summary>
  /// Interaktionslogik für MainWindow.xaml
  /// </summary>
  public partial class MainWindow : Window {
     public MainWindow() {
        InitializeComponent();
     }
  }
}
```

Wenn Sie im Visual Studio eine Ereignisbehandlung zu einem Steuerelement auf der Fensteroberfläche anfordern, 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 button1_Click(object sender, RoutedEventArgs e) {
     }
}
```

Diesen Methodenrumpf zum **Click**-Ereignis der Klasse **Button** erhalten Sie z.B. so:

  - Datei **MainWindow.xaml** samt Fenster-Werkstück per Doppelklick auf den Eintrag im Projektmappen-Explorer öffnen
  - Toolbox-Werkzeug (z.B. über den Menübefehl **Ansicht > Toolbox**) öffnen und ein **Button**-Objekt per Drag & Drop auf die Fensterfläche befördern
  - Doppelklick auf das **Button**-Objekt

Dabei erhält in der zugehörigen XAML-Datei **MainWindow.xaml** das zum **Button**-Objekt gehörige Element einen entsprechenden Wert für das Attribut **Click**:

```
<Button  Click="button1_Click" ... />
```

Im nächsten Abschnitt ist zu sehen, wo die Ergänzungsstücke zu den partiellen Klassendefinitionen in **App.xaml.cs** bzw. **MainWindow.xaml.cs** stecken.

9.4.4 XAML-Verarbeitung beim Erstellen und Starten einer WPF-Anwendung

In diesem Abschnitt werfen wir einen Blick hinter die WPF-Kulissen, wobei interessante Details der technischen Realisation erkennbar werden. So wird z.B. die Startmethode **Main()** lokalisiert, die auch bei einer C# - Anwendung mit WPF-Bedienoberfläche ihre übliche Rolle in der Startphase spielt. Allerdings ist dieses Hintergrundwissen für die Praxis der Anwendungsentwicklung mit dem Visual Studio nicht erforderlich, so dass 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**) vom XAML–Compiler **xamlc.exe** in eine binäre (validierte und effizient zu verarbeitende) Variante mit der Namenserweiterung **.baml** übersetzen (z.B. **MainWindow.baml**). Diese Dateien landen bei einem Projekt mit der Zielplattform **x86** und der Build-Konfiguration **Debug** im Projektunterordner

...\obj\x86\Debug

Die BAML-Dateien zu den Fensterklassen werden als so genannte *Ressourcen* (siehe unten) in das 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 gehörig) eine C# - Quellcodedatei mit einer partiellen Klassendefinition, bei einer WPF-Anwendung mit der Anwendungsklasse `App` und der *einem* Fenster (Hauptfensterklasse `MainWindow`) also die Dateien:

- **MainWindow.g.cs**
  In der partiellen Klassendefinition zum Hauptfenster

  ```
  public partial class MainWindow :
    System.Windows.Window, System.Windows.Markup.IComponentConnector {
     . . .
  }
  ```

  werden die Referenzvariablen zu den Steuerelementen des Fensters deklariert, z.B.:

  ```
  internal System.Windows.Controls.Button button1;
  ```

  Die Variablennamen stammen aus den **Name** – oder **x:Name** - Attributen der zugehörigen XAML-Instanzelemente (vgl. Abschnitt 9.4.2.1).

  Außerdem findet sich hier die im Fensterklassenkonstruktor (siehe Code-Behind – Datei) aufgerufene Methode **InitializeComponent()**. Diese sorgt durch einen Aufruf der statischen **Application**-Methode **LoadComponent()** für das Laden der BAML-Ressource und damit für das Instantiieren und Initialisieren der zugehörigen Objekte:

```
public void InitializeComponent() {
  if (_contentLoaded) {
   return;
  }
  _contentLoaded = true;

  System.Uri resourceLocater =
    new System.Uri("/WpfPro;component/mainwindow.xaml", System.UriKind.Relative);
   .  .  .
  System.Windows.Application.LoadComponent(this, resourceLocater);
   .  .  .
}
```

Schließlich enthält die partielle `MainWindow`–Klassendefinition noch die Methode **Connect**(), die den Vertrag der Schnittstelle **System.Windows.Markup.IComponentConnector** erfüllt und die `MainWindow`–Referenzvariablen mit den BAML-Objekten (Steuerelementen) verbindet:

```
void System.Windows.Markup.IComponentConnector.Connect(int connectionId,
                                                       object target) {
    switch (connectionId)
    {
    case 1:
    this.button1 = ((System.Windows.Controls.Button)(target));

    #line 6 "..\..\..\MainWindow.xaml"
    this.button1.Click += new System.Windows.RoutedEventHandler(this.button1_Click);

    #line default
    #line hidden
    return;
    }
    this._contentLoaded = true;
}
```

Die Klasse `MainWindow` implementiert die Methode **Connect()** *explizit*, sodass der Schnittstellenname angegeben werden muss und kein Zugriffsmodifikator gesetzt werden darf (siehe Abschnitt 8.4). In **Connect**() werden ggf. auch die Ereignisbehandlungsmethoden zu den Steuerelementen registriert, welche in der Regel in der vom Programmierer gepflegten Code-Behind - Datei **MainWindow.xaml.cs** (siehe Abschnitt 9.4.3) mit dem Rest der `MainWindow`-Klassendefinition implementiert werden.

- **App.g.cs**
  Hier findet sich die partielle Definition der von **System.Windows.Application** abgeleiteten Anwendungsklasse mit dem Namen `App`:

```
public partial class App : System.Windows.Application {
   .  .  .
  public void InitializeComponent() {
     .  .  .
    #line 4 "..\..\..\App.xaml"
    this.Exit += new System.Windows.ExitEventHandler(this.Application_Exit);
     .  .  .
    #line 4 "..\..\..\App.xaml"
    this.StartupUri = new System.Uri("MainWindow.xaml", System.UriKind.Relative);
     .  .  .
  }

  [System.STAThreadAttribute()]
  public static void Main() {
    App app = new WpfApplication1.App();
    app.InitializeComponent();
    app.Run();
  }
}
```

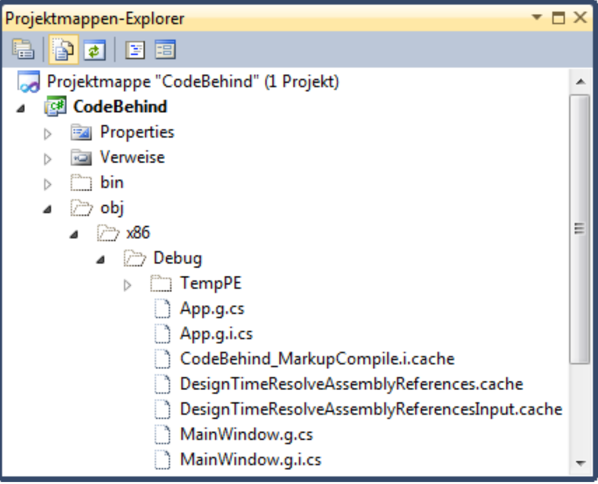
Um für den automatischen Start des Hauptfensters zu sorgen, wird in der `App`-Methode **InitializeComponent()** (nicht zu verwechseln mit der gleichnamigen Methode in der Klasse `MainWindow`) der **Application**-Eigenschaft **StartupUri** ein **Uri**-Objekt zugewiesen, das auf der XAML-Deklaration der Hauptfensterklasse basiert. Die Methode **InitializeComponent()** ist auch dafür zuständig, Behandlungsmethoden für `App`-Ereignisse zu registrieren.

Schließlich findet sich in der partiellen, automatisch erstellten `App`-Klassendefinition die Startmethode **Main()**. Die Abweichungen von der manuell erstellten **Main()** – Methode unserer minimalen WPF-Anwendung in Abschnitt 9.2.1 halten sich in Grenzen.

Die Behandlungsmethoden für `App`-Ereignisse werden in der Code-Behind – Datei **App.xaml.cs** zur Anwendungsklasse implementiert.

Gehen Sie folgendermaßen vor, wenn Sie die generierten Quellcode-Dateien im Visual Studio öffnen wollen:

- Projekt erstellen lassen (z.B. mit dem Schalter **F6**)
  Neben den **g**-Dateien gibt es noch **g.i**–Dateien, die vom Visual Studio für die Intellisense-Funktionalität benötigt werden. Während die **g.i** – Dateien bei jeder Änderung am Projekt vom Visual Studio aktualisiert werden, erhalten die **g**-Dateien nur beim Erstellen den aktuellen Stand.
- Im Projektmappen-Explorer über den Symbolschalter **alle Dateien anzeigen** lassen
- Pfad mit den generierten Dateien öffnen, z.B.:



- Gewünschte Datei per Doppelklick im Quellcode-Editor öffnen.

9.5 Spaß- und Motivationszufuhr

Die letzten Abschnitte waren eventuell nicht für alle Leser vergnüglich und motivationsfördernd. Damit nicht ausgerechnet im eigentlich Spaß versprechenden Kapitel über die Programmierung graphischer Benutzeroberflächen ein Handtuch fliegt, schieben wir einen Abschnitt ein, der hoffentlich Entspannung bringt und neue Motivation zuführt. Wir erstellen wir 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* bezeichnet man eine im Internet angebotene, per URL ansprechbare Datei, die neue Themen kurz beschreibt und jeweils einen Link zur Vollinformation bietet. Als Dateiformat dient eine XML-Variante namens RSS, aktuell in der Version 2.0.

Die deutsche Übersetzung *Zufuhr* für das englische Wort *feed* ist übrigens in die Abschnittsüberschrift eingeflossen.

Laut Wikipedia (http://de.wikipedia.org/wiki/RSS#Entwicklung) enthält eine RSS-Datei der Version 2.0 Item-Elemente mit folgendem Aufbau:

```
<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>
```

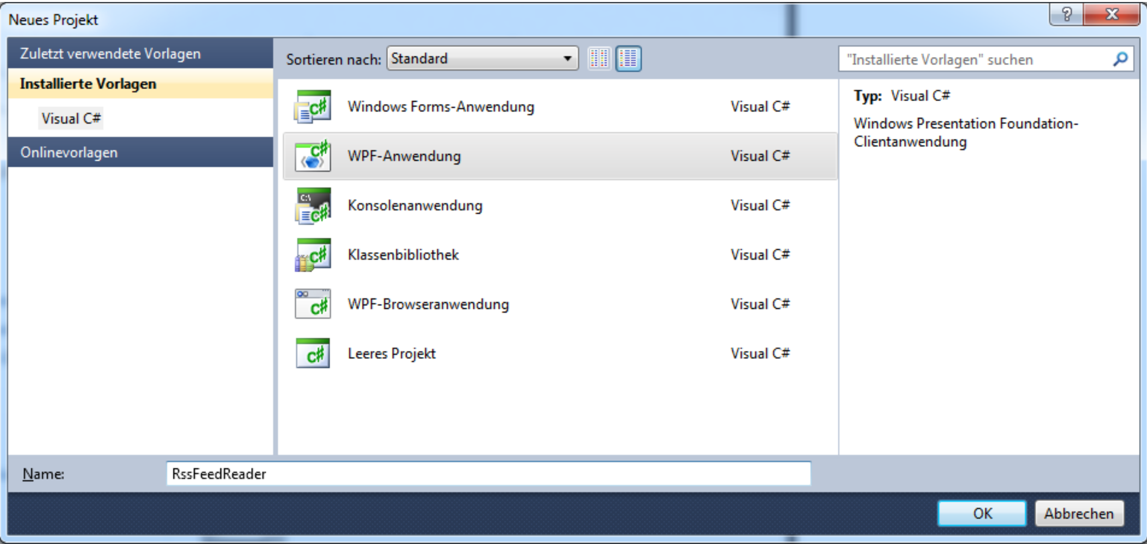
Beim RSS-Feed – Projekt werden wir unser Wissen über WPF-Programme anwenden und stabilisieren. Es kommen aber einige bisher unbehandelte WPF-Themen vor (z.B. Datenbindung, Vorlagen), so dass Sie bei der späteren Behandlung schon auf erfolgreiche Anwendungserfahrungen zurückblicken können. Außerdem üben wir den Umgang mit dem WPF-Designer im Visual Studio 2010, der uns beim Erstellen von XAML-Deklarationsdateien sehr gut unterstützt.

9.5.1 Projekt anlegen mit Vorlage *WPF - Anwendung*

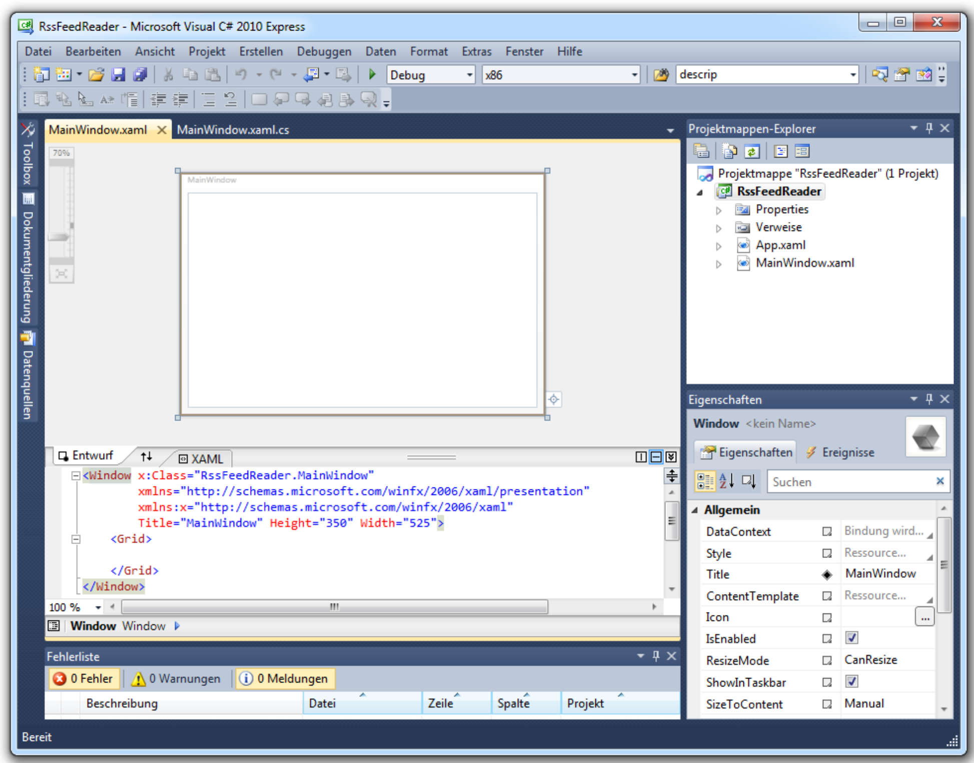
Verwenden Sie nach

Datei > Neues Projekt

für ein neues Projekt mit dem Namen `RssFeedReader` die Vorlage **WPF - Anwendung**:



Nach einem Mausklick auf **OK** präsentiert die Entwicklungsumgebung im **WPF-Designer** einen Rohling für das Hauptfenster der entstehenden Anwendung:



In der oberen Designer-Zone können wir die Bedienoberfläche unseres Programms mit Hilfe von graphischen Werkzeugen erstellen und dazu konfigurierbare Komponenten (Steuerelemente) aus der Toolbox übernehmen. Wie gleich zu sehen sein wird, gestalten wir dabei den Auftritt von Objekten aus der neuen Klasse `MainWindow` im Namensraum `RssFeedReader`.

Wie Sie mittlerweile wissen, besteht ein zentrales Merkmal der WPF-Technologie darin, das GUI-Design einer Anwendung durch XAML-Code zu deklarieren. Zu unserem Anwendungsfenster gehört die XAML-Datei **MainWindow.xaml**, die im unteren Teil der Designer-Zone erscheint:

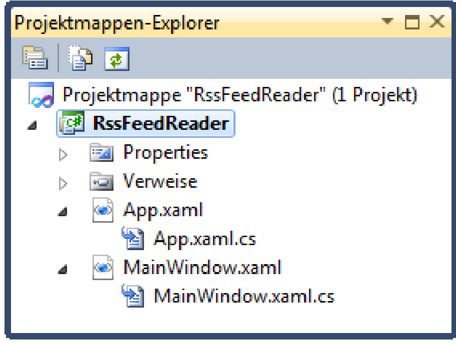
```
<Window x:Class="RssFeedReader.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

Das **Window**-Wurzelelement der XAML-Datei definiert ein WPF-Fenster, also das Erscheinungsbild der Objekte aus einer von **Window** abgeleiteten Klasse. Initial enthält es im Beispiel:

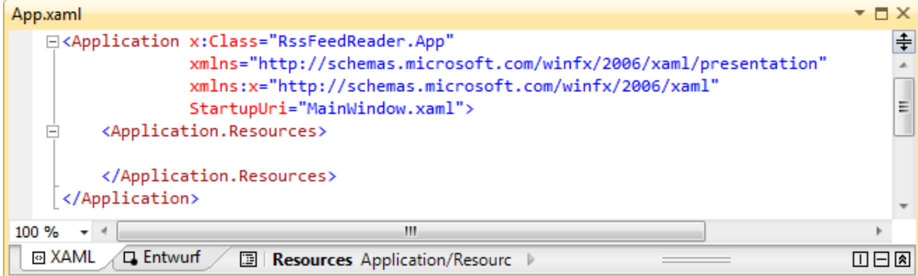
- ein Attribut namens **x:Class**, das die zugehörige Klasse samt Namensraum nennt, in unserem Fall also `RssFeedReader.MainWindow`
- zwei **xmlns**-Attribute, die Namensräume für die verwendeten XAML-Elemente angeben
- ein **Title**-Attribut für die Fensterbeschriftung
- die Attribute **Height** und **Width** für die initiale Fenstergröße
- ein **Grid**-Element, das als Container dient und später noch ausführlich besprochen wird

Der graphischen Fenster-Designer ist lediglich ein (willkommenes) Werkzeug zur Bearbeitung der XAML-Datei.

Die bearbeitete XAML-Datei **MainWindow.xaml** beschreibt das Erscheinungsbild von Objekten der Klasse `MainWindow`. Zu dieser Klasse gehört auch die Code-Behind - Datei **MainWindow.xaml.cs** (vgl. Abschnitt 9.4.3), welche die von uns in C# zu erstellende Interaktionslogik (insbesondere die Ereignisbehandlungsmethoden) aufnehmen wird. Der Projektmappen-Explorer zeigt die XAML- und die zugehörige C# - Datei:



Über die Hauptfensterklasse hinaus benötigt eine WPF-Anwendung auch noch eine Anwendungsklasse, abstammend von der FCL-Klasse **Application**, wobei erneut eine XAML-Deklarationsdatei und eine C# - Code-Behind - Datei beteiligt sind. Bei unserem geplanten Beispielprogramm müssen wir uns allerdings um beide nicht kümmern. Wer die XAML-Datei **App.xaml** neugierig per Doppelklick auf ihren Eintrag im Projektmappen-Explorer öffnet,



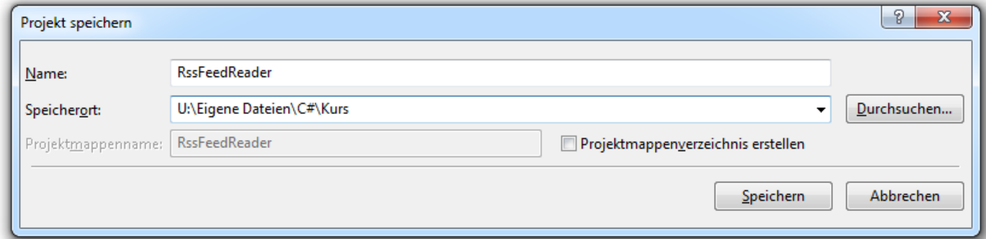
findet im **Application**-Wurzelelement u.a. die die folgenden Attribute:

- **x:Class**
  enthält den Namen der zur XAML-Datei gehörigen .NET - Klasse samt Namensraum, im Beispiel: `RssFeedReader.App`
- **StartupUri**
  legt das beim Programmstart anzuzeigende Fenster fest

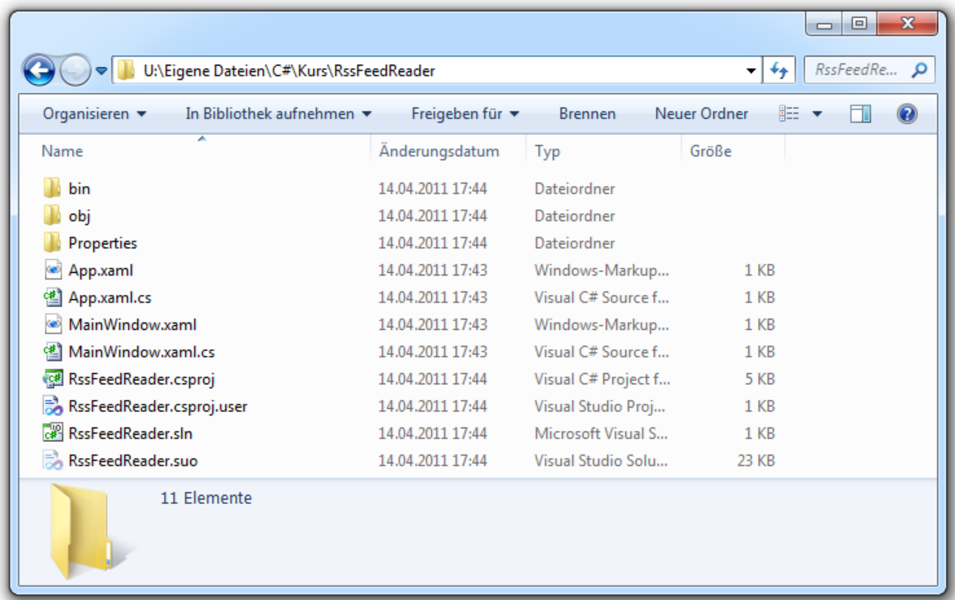
Speichern Sie das neue Projekt über

Datei > Alle speichern

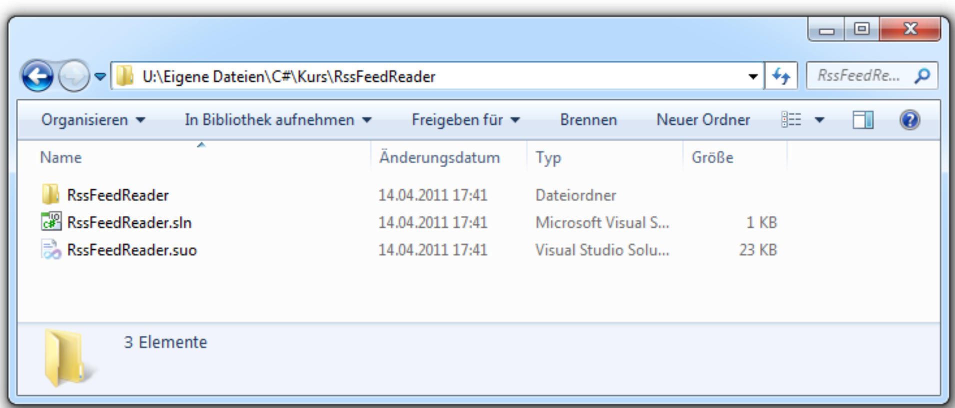
in einem Ordner Ihrer Wahl (festgelegt durch den **Namen** und den **Speicherort** mit dem Basisverzeichnis), z.B.:



Durch den Verzicht auf ein **Projektmappenverzeichnis** wählen wir eine „flache“ Projektdateiverwaltung, weil keine Zusammenfassung von mehreren Projekten zu einer Mappe geplant ist:



Dies wäre die Alternative *mit* einem **Projektmappenverzeichnis**:

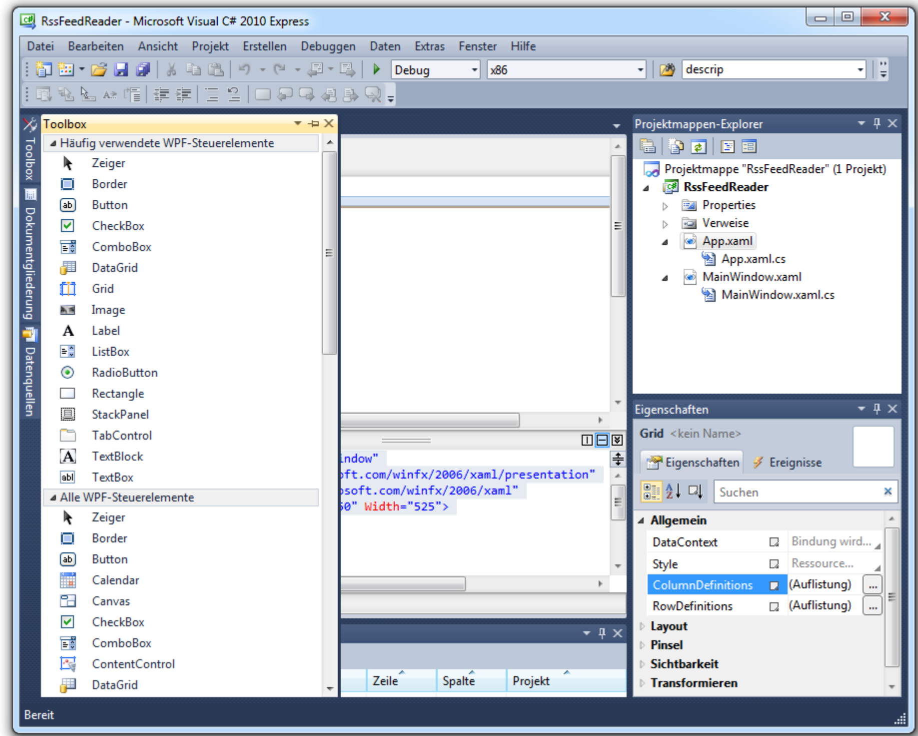


9.5.2 Steuerelemente aus der Toolbox übernehmen

Öffnen Sie das **Toolbox**-Fenster mit dem Menübefehl

Ansicht > Toolbox

oder per Mauszeiger durch kurzes Verharren auf der **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:

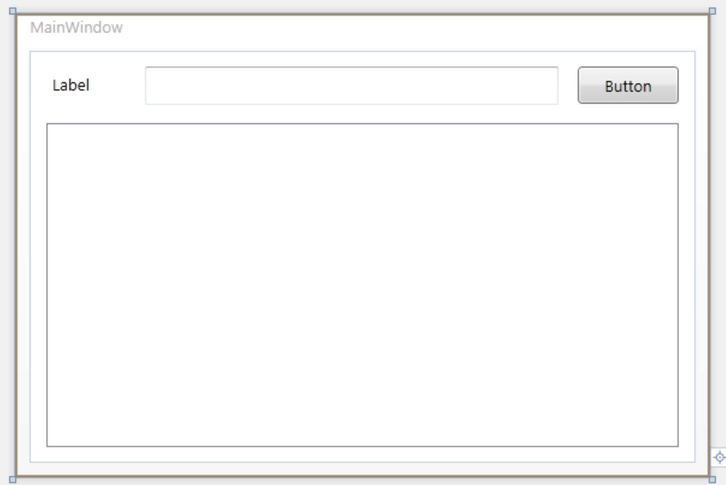
- ein **Label**-Objekt
  Es soll den im rechts daneben stehenden Texteingabefeld erwarteten Inhalt beschreiben.
- ein **TextBox**-Objekt
  Hier können die Benutzer die Feed-Adresse eintragen.
- ein **Button**-Objekt
  Damit fordern die Benutzer das Laden einer RSS-Datei an.
- ein **ListBox**-Objekt
  Hier sollen die RSS-Items angezeigt werden.

Die Übernahme eines Steuerelements aus der Toolbox gelingt ...

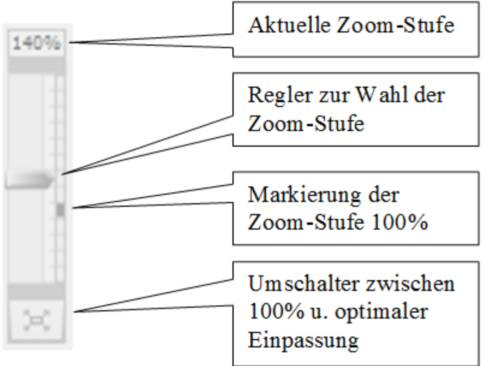
- per Doppelklick auf den jeweiligen **Toolbox**-Eintrag
- oder per Drag & Drop (Ziehen und Ablegen)

9.5.3 Positionen, Größen und Verankerungspunkte der Steuerelemente

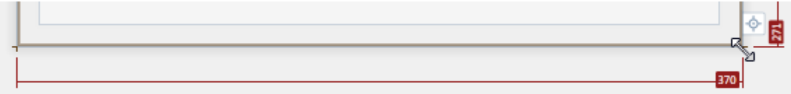
Nun können Sie wie in einem Grafikprogramm die Positionen und Größen der Fensterbestandteile verändern, um das gewünschte Layout zu erzielen, z.B.:



Zur Erleichterung der Arbeit lässt sich mit dem **Zoom-Werkzeug** des WPF-Designers (oben links) eine passende Ansicht einstellen:



Wählen Sie zunächst für das Hauptfenster eine initiale Größe über den Anfasser unten rechts;



Die aktuelle Breite und Höhe in Bildschirm-Pixeln wird mit rot hinterlegter Schrift angezeigt. Achten Sie darauf, dass Sie wirklich das Hauptfenster 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**).

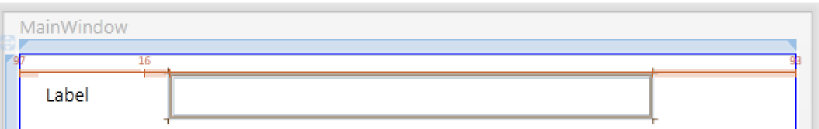
Bei aktivem Hauptfenster kann man mit der Schaltfläche

in der rechten unteren Ecke zwischen zwei Optionen für die Hauptfensterfläche zur Laufzeit umschalten:

gleiche Größe wie in der Entwurfsansicht bzw. automatische Größenberechnung

Bleiben Sie bei der Voreinstellung .

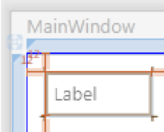
Bei der Abstands- und Größenanpassung von Steuerelementen helfen **Ausrichtungs- bzw. Führungslinien**. Im folgenden Beispiel (das **TextBox**-Objekt wird bewegt) taucht eine horizontale Linie mit anziehender Wirkung auf, sobald die vertikale Position des **Label**-Objekts erreicht wird, so dass man den beiden Steuerelementen leicht eine gemeinsame Oberkante geben kann:



Außerdem werden im Beispiel horizontale Abstände angezeigt.

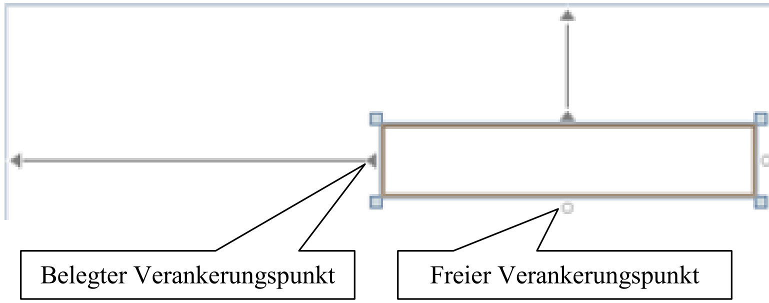
Man kann also z.B. so vorgehen, um die Steuerelemente für den RSS-Feed - Reader anzuordnen:

- Das **Label**-Objekt in die oberer linke Ecke setzen und dabei die vorgeschlagenen Randabstände von 12 Punkten nach links und rechts akzeptieren:



  Das **Label**-Objekt sollte breit genug sein, um eine geeignete Beschriftung (z.B. *Feed-URL:*) aufzunehmen.
- Das **TextBox**-Objekt wird mit dem eben beschrieben Verfahren rechts neben das Label gesetzt.
- Das **Button**-Objekt wird zwischen **TextBox**-Objekt und Fensterrand gesetzt, so dass die drei Bedienelemente im Kopfbereich eine gemeinsame Oberkante und Höhe haben.
- Beim **ListBox**-Objekt übernimmt man die vorgeschlagenen 12-Punkt-Abstände zu den Fensterrändern (links, unten, rechts) und wählt einen passenden Abstand zu den Bedienelementen im Kopfbereich.

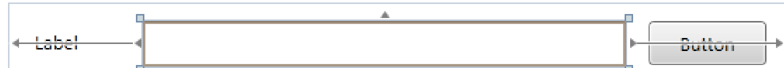
Ändert der Benutzer im laufenden Pogramm die Größe des Hauptfensters, dann hängt das Orts- und Größenverhalten eines Steuerelements davon ab, zu welchen Seiten des umgebenden Containers es einen festen **Randabstand** einhält. Per Voreinstellung sind die Steuerelemente links und oben angedockt, was bei einem markierten Steuerelement durch Verankerungslinien zum jeweiligen Rand angezeigt wird, z.B.:



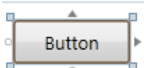
Um eine zusätzliche Verankerung vorzunehmen, klickt man auf den zugehörigen freien Verankerungspunkt. Um eine Dockseite aufzugeben klickt man auf den belegten Verankerungspunkt. Ist die gegenüberliegende Seite frei, springt die Verankerung dorthin.

Bei unserem RSS-Feed - Reader werden die Benutzer sicher oft die Fenstergröße ändern, und wir müssen durch geschickt gewählte Verankerungspunkte dafür sorgen, dass die verfügbare Gesamtfläche optimal auf die Steuerelemente aufgeteilt wird:

- Beim **Label** können die voreingestellten Verankerungspunkte (links und oben) beibehalten werden.
- Das **TextBox**-Objekt sollte links, oben und rechts verankert werden, so dass es von horizontaler Größenzunahme profitiert, auf vertikale Größenzunahme aber nicht reagiert:



- Der Befehlsschalter sollte rechts und oben verankert werden, also bei beliebiger Veränderung der Fenstergröße bei konstanter eigener Größe in der rechten oberen Ecke verbleiben:



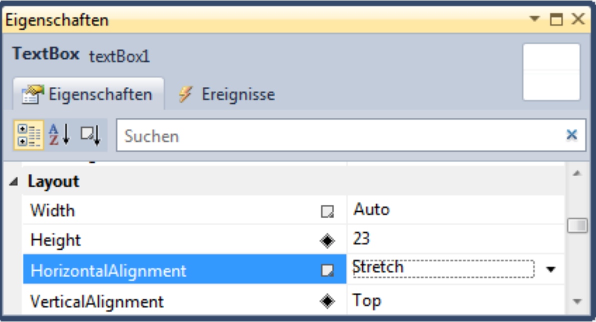
- Das **ListBox**-Objekt hat den größten Platzbedarf und sollte bei jeder Größenzunahme profitieren. Es wird daher an allen vier Seiten verankert.

Mit einem gelegentlichen Blick auf das XAML-Fenster können Sie feststellen, dass bei der intuitiv-kreativen Arbeit mit dem WPF-Designer letztlich der XAML-Code modifiziert wird. Zum **TextBox**-Objekt sollte ungefähr das folgende XAML-Element entstanden sein:

```
<TextBox Height="23" Margin="98,12,108,0" Name="textBox1" VerticalAlignment="Top" />
```

Es hat ...

- per **Name**-Attribut einen Instanzvariablennamen erhalten, unter dem das Objekt im Programm ansprechbar sein wird.
- per **Height**-Attribut eine Höhe erhalten
  Sollte Ihnen die mit Maus und Augenmaß gefunden Einstellung missfallen, können Sie den gewünschten Wert direkt im XAML-Fenster eintragen. Das Visual Studio hält das Designer- und das XAML-Fenster stets synchron. Selbst die in einem Fenster vorgenommene Objektmarkierung wirkt sich sofort auch auf das andere Fenster aus.
- per **Margin**-Attribut zum linken, oberen und rechten Seite des umgebenden Containers einen Randabstand erhalten
  Dass wir durch eine Zeichenfolge mit vier kommagetrennten Zahlen eine Instanz der Struktur **System.Windows.Thickness** deklarieren können, ist übrigens einem Typkonverter zu verdanken.
- per **VerticalAlignment**-Attribut eine Befestigung am oberen Fensterrand erhalten
  Wächst die Höhe des Fensters, bleibt das TextBox-Objekt an oberen Rand. Ein **HorizontalAlignment**–Attribut fehlt. Damit ist Wert **Stretch** gewählt, wie ein Blick auf das Eigenschaftenfenster bestätigt:

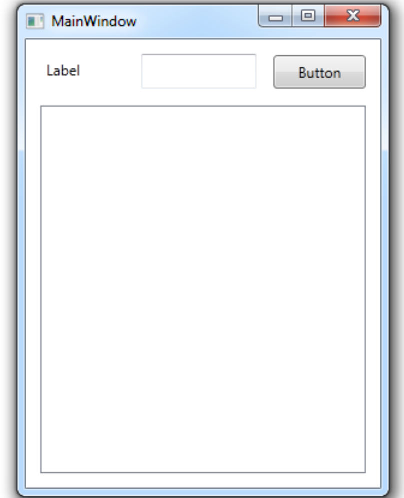
  Es befindet sich in der unteren rechten Ecke der Entwicklungsumgebung und ist bei Bedarf mit dem Menübefehl

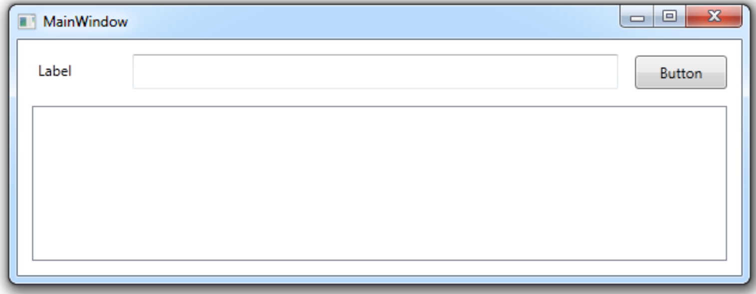
  Ansicht > Eigenschaftsfenster

  zu öffnen. Die Werte zu den Attributen **VerticalAlignment** und **HorizontalAlignment** hat das Visual Studio aus den vier Verankerungspunkten zum Steuerelement abgeleitet, die wir mit Mausklicks eingestellt haben.

Als Maßeinheit für alle Größenangaben dienen geräteunabhängige Pixel (= 1/96 Zoll).

Weil das Programm von Anfang an startfähig ist, kann man sein Verhalten bei variabler Fenstergröße leicht überprüfen, z.B.:



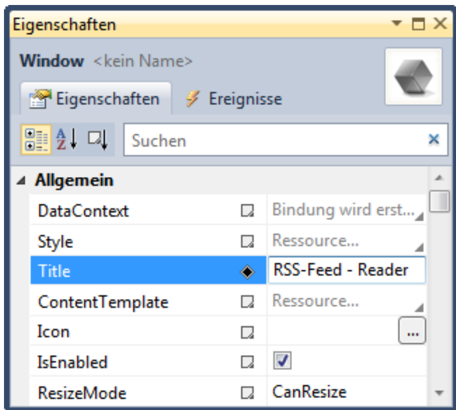


9.5.4 Eigenschaften der Steuerelemente ändern

Im Eigenschaftsfenster der Entwicklungsumgebung, das sich per Voreinstellung in der unteren rechten Ecke befindet und bei Bedarf über das Kontextmenü zu einem Steuerelement oder mit dem Menübefehl

Ansicht > Eigenschaftsfenster

zu öffnen ist, lassen sich diverse Eigenschaften des markierten Steuerelements durch direkte Werteingabe festlegen, z.B. der **Titel** des Fensters:



Nehmen Sie bitte noch folgende Einstellungen vor:

- Wählen Sie geeignete Texte für die **Content** – Eigenschaften des **Label**- und des **Button**-Objekts, so dass sinnvolle Beschriftungen entstehen.
- Die **Text**-Eigenschaft des **TextBox**-Objekts sollte als Initialwert die Adresse des RSS-Feeds mit Neuigkeiten für Benutzer der Entwicklungsumgebung Visual Studio Express erhalten:

  http://www.microsoft.com/germany/msdn/rss/dc_vsexpress.xml
- Aktivieren Sie das Kontrollkästchen zur **IsDefault**-Eigenschaft der Schaltfläche, so dass der Schalter im laufenden Programm auch per **Enter**-Taste angesprochen werden kann.

Wie Sie wissen, enthält die XAML-Datei zu einem Fenster alle erforderlichen Informationen zum Erzeugen und Initialisieren der enthaltenen Objekte. Folglich haben sich auch unsere Arbeiten mit dem Eigenschaftenfenster auf den XAML-Code ausgewirkt. Sie sollten jetzt ungefähr das folgende **Button**-Element sehen:

```
<Button Content="Laden" Height="23" HorizontalAlignment="Right" Margin="0,12,12,0"
  Name="button1" VerticalAlignment="Top" Width="75" IsDefault="True" />
```

Im **Content**-Attribut hat sich der Text geändert, und das neue Attribut **IsDefault** besitzt den Wert **True**. Dieser Wert kann übrigens ausnahmsweise groß oder klein geschrieben werden, wobei ein spezielles Wissen des XAML-Parsers um den Datentyp der zugehörigen **Button**-Eigenschaft zur Anwendung kommt.

Zwar ist eine Eigenschaftsmodifikation zur *Entwurfszeit* besonders bequem per Eigenschaftsfenster zu bewerkstelligen, doch muss man trotzdem wissen, wie der Eigenschaftszugriff per C# - Anweisung erfolgt, weil viele Steuerelementeigenschaften auch zur *Laufzeit* (dynamisch) geändert werden müssen.

9.5.5 Automatisch erstellter Quellcode

Aufgrund Ihrer kreativen Tätigkeit pflegt die Entwicklungsumgebung nicht nur den XAML-Code zum Hauptfenster, sondern erstellt auch Quellcode zu einer neuen Klasse namens `MainWindow`, die von der Klasse **Window** im Namensraum **System.Windows** abgeleitet wird. 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, die jeweils eine partielle Klassendefinition enthalten (vgl. die Abschnitte 9.4.3 und 9.4.4):

- **MainWindow.xaml.cs**
  Diese Datei wird zwar von der Entwicklungsumgebung angelegt, aber doch im Wesentlichen vom Programmierer gefüllt. Hier werden die von Ihnen zu erstellenden Ereignisbehandlungmethoden landen.
- **MainWindow.g.cs**
  Hier landet ausschließlich automatisch generierter Quellcode, und Sie sollten in dieser Datei keine Änderungen vornehmen, um die Entwicklungsumgebung nicht aus dem Tritt zu bringen.

Hier ist die Datei **MainWindow.xaml.cs** zu sehen, die zwar vom Visual Studio angelegt, aber im Wesentlichen von Ihnen erstellt wird:

```
using System;
   .  .  .
using System.Windows.Shapes;

namespace RssFeedReader {
   /// <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 (siehe Abschnitt 9.4.4).

Aufgrund unserer Tätigkeit im WPF-Designer enthält die Fensterklasse `MainWindow` mehrere Objekte anderer Klassen, die Steuerelemente der graphischen Bedienoberfläche repräsentieren. In der Datei **MainWindow.g.cs** finden sich die Deklarationen der zugehörigen Instanzvariablen:

```
internal System.Windows.Controls.Label label1;
internal System.Windows.Controls.TextBox textBox1;
internal System.Windows.Controls.Button button1;
internal System.Windows.Controls.ListBox listBox1;
```

Eine weitergehende Analyse des automatisch erstellten Quellcodes sparen wir uns.

9.5.6 Click-Ereignisbehandlung zum Befehlsschalter (Teil 1)

Wir erstellen eine Ereignisbehandlungsmethode, die durch das Betätigen des Befehlsschalters (per Mausklick oder **Enter**-Taste) ausgelöst wird. Sie 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, so dass die Entwicklungsumgebung in der Datei **MainWindow.xaml.cs** die Instanzmethode `button1_Click()` der Klasse `MainWindow` mit leerem Rumpf anlegt

```
private void button1_Click(object sender, RoutedEventArgs e) {

}
```

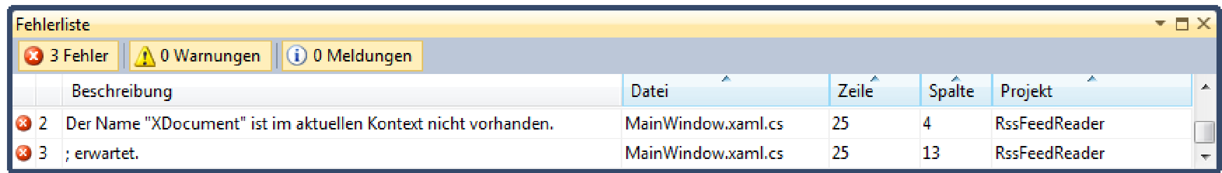
und die Quellcodedatei im Editor öffnet.

Wir befördern die vom Benutzer gewünschte RSS-Datei in ein Objekt der FCL-Klasse **XDocument** und verwenden für das nicht triviale, mit einem Internetzugriff verbundene Laden der RSS-Datei die statische **XDocument**-Methode **Load**(), wobei die **Text**-Eigenschaft (mit Datentyp **String**) des **TextBox**-Bedienelements als Parameter übergeben wird.

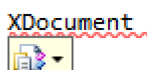
Der optimistisch als Start der ersten Anweisung in der Methode `button1_Click()` eingetippte Klassennamen wird von der Entwicklungsumgebung rot unterschlängelt,

```
private void button1_Click(object sender, RoutedEventArgs e) {
    XDocument
}
```

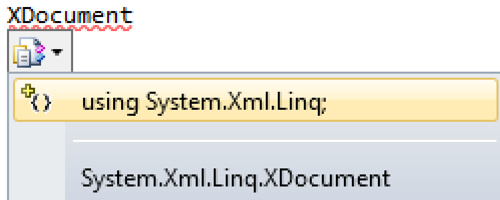
und das Fenster **Fehlerliste** informiert:



Sobald die Maus auf das kleine blaue Rechteck unter dem kritisierten Klassennamen zeigt, erscheint ein Symbol, das in einer aufklappbaren Liste Hilfe bereithält:



Nach einem Klick auf das Dreieck wird vorgeschlagen, entweder den Namensraum **System.Xml.Linq**, in dem sich die Klasse **XDocument** befindet, per **using**-Direktive zu importieren, oder dem Klassenamen einen Namensraumpräfix voranzustellen:



Weil die Klasse **XDocument** in unserem Programm mindestens zweimal angesprochen werden muss, lassen wir eine **using**-Direktive zu dieser Klasse einfügen (Vorschlag 1). Nun komplettieren wir mit Intellisense-Unterstützung die Anweisung:

```
XDocument feed = XDocument.Load(textBox1.Text);
```

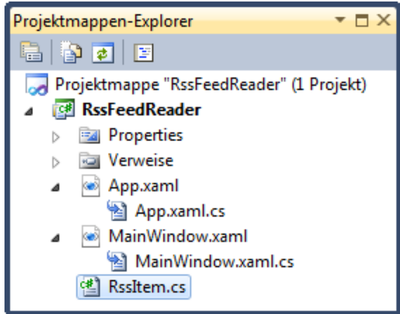
Um ein **ListBox**-Steuerelement zu füllen, kann man seiner Eigenschaft **ItemsSource** ein Objekt aus einer Klasse zuweisen, welche die Schnittstelle **IEnumerable** im Namensraum **System.Collections** erfüllt. Unter einem solchen Kollektionsobjekt können Sie sich einen größendynamischen Array vorstellen. Zwecks Typsicherheit sollte man ein Kollektionsobjekt einen festem Elementtyp verwenden, also mit der generischen Schnittstelle **IEnumerable<*Elementtyp*>** arbeiten. In unserem Fall treten RSS-Items als Elemente auf, wobei wir eine modellierende Klasse (z.B. mit dem Namen `RssItem`) erst noch definieren müssen. Um die Hilfsbereitschaft und Kompetenz der Entwicklungsumgebung zu testen, starten wir mit der Deklaration einer Referenzvariablen vom Typ `IEnumerable<RssItem>` unter Verwendung der noch fehlenden Klasse `RssItem`:

```
private void button1_Click(object sender, RoutedEventArgs e) {
    XDocument feed = XDocument.Load(textBox1.Text);
    IEnumerable<RssItem>

}
```

Klasse für "RssItem" generieren

Neuen Typ generieren...

Wählt man das erste Hilfsangebot, erstellt das Visual Studio eine Klasse mit gewünschten Namen in einer eigenen Datei, die im Projektmappen-Explorer erscheint:



In der automatisch erstellten Klassendefinition

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace RssFeedReader {
   class RssItem {
   }
}
```

müssen wir noch Eigenschaften für den Titel, die Kurzbeschreibung und den URL eines RSS-Items ergänzen, was gleich mit Hilfe der Entwicklungsumgebung geschehen soll. Wir vervollständigen zunächst in der Ereignisbehandlungsmethode `button1_Click()` die Anweisung mit der Deklaration einer lokalen Variablen vom Typ `IEnumerable<RssItem>`:

```
IEnumerable<RssItem> items;
```

Nun widmen wir uns der nicht ganz trivialen Aufgabe, die im **XDocument**-Objekt `feed` enthaltenen RSS-Items zu extrahieren und als Objekte der neuen Klasse `RssItem` in das Kollektionsobjekt `items` vom Typ `IEnumerable<RssItem>` einzufüllen. Dazu verwenden wir eine Technologie namens **LINQ to XML**, die momentan nur oberflächlich bzw. effektorientiert erläutert werden kann. Die **XDocment**-Instanzmethode **Descendents**() liefert ein Objekt, das die Schnittstelle **IEnumerble<XElement>** erfüllt. Per LINQ-Abfragesyntax (unter Verwendung der Schlüsselwörter **from**, **in**, **select**) erstellen wir daraus ein Kollektionsobjekt aus einer Klasse, welche die Schnittstelle `IEnumerable<RssItem>` erfüllt. Erweitern Sie bitte die Methode `button1_Click()` um die folgende Anweisung:

```
items = from item in feed.Descendants("item")
```

```
        select new RssItem() {
           Title = item.Element("title").Value,
           Description = item.Element("description").Value,
           Url = item.Element("link").Value
        };
```

In der LINQ-Abfragesyntax werden 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.:

```
private void button1_Click(object sender, RoutedEventArgs e) {
   XDocument feed = XDocument.Load(textBox1.Text);
   IEnumerable<RssItem> items;
   items = from item in feed.Descendants("item")
           select new RssItem() {
              Title = item.Element("title").Value,
                iption = item.Element("description").Value,
           };
}
```

Eigenschaftsstub für "Title" in "RssFeedReader.RssItem" generieren
Feldstub für "Title" in "RssFeedReader.RssItem" generieren

Die generierte Definition zur Eigenschaft `Title`

```
class RssItem {
   public string Title { get; set; }
}
```

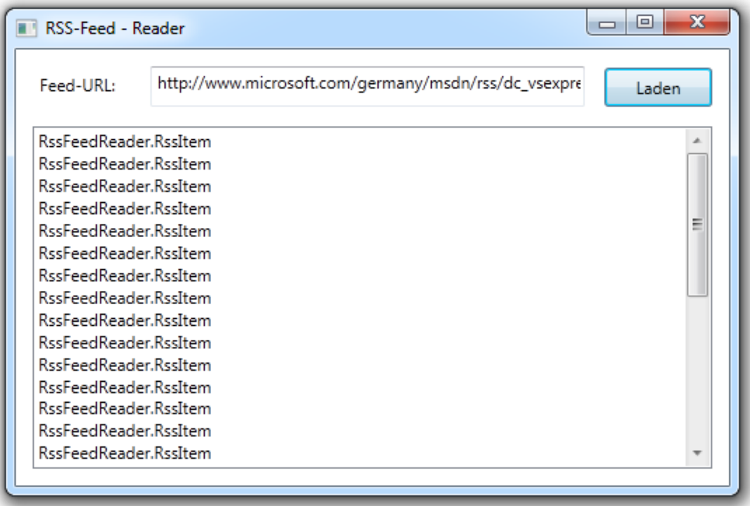
ist durchaus vollständig, weil der C# - Compiler seit der Version 3.0 das zugehörige private Feld automatisch erstellen kann. Analog erhalten wir ohne große Anstrengungen die komplette Definition der Klasse `RssItem`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace RssFeedReader {
   class RssItem {
      public string Title { get; set; }
      public string Description { get; set; }
      public string Url { get; set; }
   }
}
```

Nun können wir am Ende der Ereignisbehandlungsmethode `button1_Click()` der **ListBox**-Eigenschaft **ItemsSource** das Kollektionsobjekt mit den `RssItem`-Elementen zuweisen:

```
private void button1_Click(object sender, RoutedEventArgs e) {
   XDocument feed = XDocument.Load(textBox1.Text);
   IEnumerable<RssItem> items;
   items = from item in feed.Descendants("item")
           select new RssItem() {
              Title = item.Element("title").Value,
              Description = item.Element("description").Value,
              Url = item.Element("link").Value
           };
   listBox1.ItemsSource = items;
}
```

Ein Startversuch mit dem Laden des voreingestellten RSS-Feeds zeigt, dass wir auf einem guten Weg sind:



9.5.7 Formatierung der Listenelemente per DataTemplate-Objekt

Bislang zeigt das **ListBox**-Objekt zu jedem 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 Hauptfenster:

```
<ListBox Margin="12,50,12,12" Name="listBox1">
  <ListBox.ItemTemplate>
    <DataTemplate>
        <StackPanel Name="stackPanel1" Orientation="Vertical">
          <TextBlock Name="textBlock1" Text="{Binding Path=Title}" Margin="1"
            TextWrapping="Wrap" FontSize="14" FontWeight="Bold" Foreground="DarkMagenta"
            TextAlignment="Center" />
          <TextBlock Name="textBlock2" Text="{Binding Path=Description}"
            Margin="1,1,1,4" TextWrapping="Wrap" TextTrimming="WordEllipsis"
            MaxHeight="50" TextAlignment="Justify" />
        </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Im XAML-Element **ListBox** wird die Eigenschaftselementsyntax (vgl. Abschnitt 9.4.2.3) verwendet, um den Wert der **ListBox**-Eigenschaft **ItemTemplate** zu deklarieren. 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 unten) mit vertikaler Orientierung, um zwei **TextBlock**-Objekte (siehe unten) übereinander zu präsentieren.

Ein **TextBlock**-Objekt erhält seine Daten über die Datenbindungstechnologie, die zu den WPF-Glanzlichtern gehört und später noch ausführlich zu behandeln ist. Durch die folgende Attributsyntax mit einer Markuperweiterung (vgl. Abschnitt 9.4.2.7)

```
Text="{Binding Path=Title}"
```

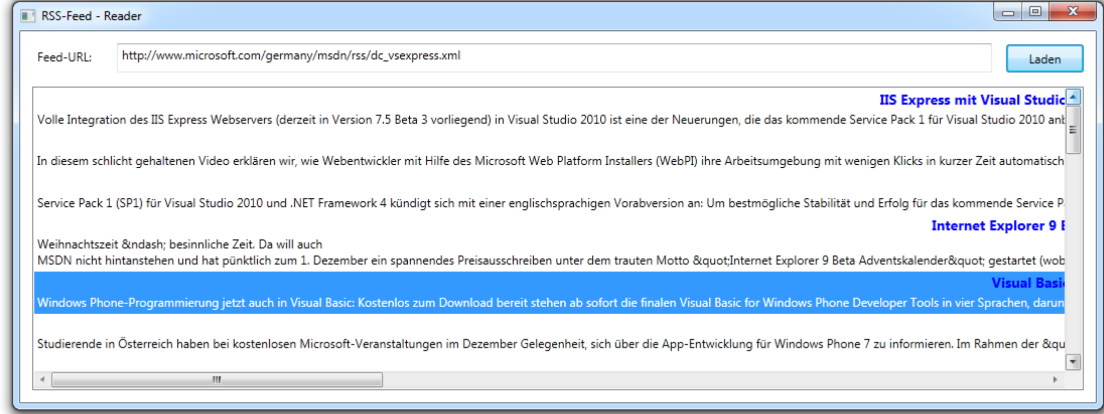
wird ein Objekt der Klasse **Binding** beauftragt, aus dem aktuellen Element der zum **ListBox**-Objekt gehörigen Kollektion den Wert der Eigenschaft `Title` zu extrahieren. Das Kollektionsobjekt wird

in der **Click**-Ereignismethode zum Befehlsschalter (siehe Abschnitt 9.5.6) erzeugt und der **ListBox**-Eigenschaft **ItemsSource** zugewiesen:

```
listBox1.ItemsSource = items;
```

Die weiteren **TextBlock** – XAML-Attribute dienen der Formatierung.

Der bei **ListBox**-Steuerelementen per Voreinstellung vorhandene horizontale Rollbalken ist bei unserem RSS-Feed - Reader für eine unpraktische Textpräsentation verantwortlich



und wird daher über das folgende Attribut zum **ListBox**-Element der XAML-Datei **MainWindow.xaml** abgeschaltet:

```
<ListBox . . . ScrollViewer.HorizontalScrollBarVisibility="Disabled">
   . . .
</ListBox>
```

HorizontalScrollBarVisibility ist eine so genannte *Abhängigkeitseigenschaft* der Klasse **Scroll-Viewer**, und Sie werden noch erfahren, warum man mit ihrer Hilfe für ein Objekt der Klasse **ListBox** den horizontalen Rollbalken beeinflussen kann.

9.5.8 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 Abschnitt 10 und zeigen ggf. im **catch**-Block eine Fehler-meldung an, welche auf die **Message**-Eigenschaft des **Exception**-Objekts zugreift:

```
try {
   XDocument feed = XDocument.Load(textBox1.Text);
   IEnumerable<RssItem> items;
   items = from item in feed.Descendants("item")
         select new RssItem() {
            Title = item.Element("title").Value,
            Description = item.Element("description").Value,
            Url = item.Element("link").Value
         };
   listBox1.ItemsSource = items;
} catch (Exception ex) {
   listBox1.ItemsSource = null;
   MessageBox.Show(this, ex.ToString(), ex.Message);
}
```

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;
```

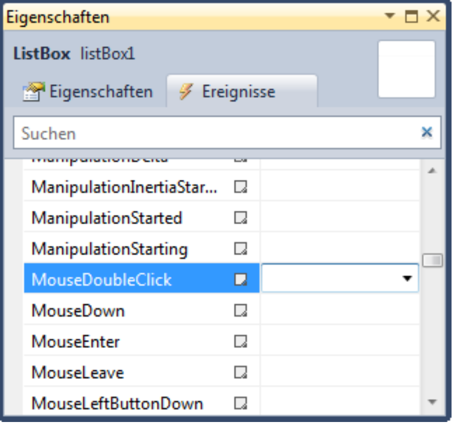
und reaktivieren vor dem Verlassen der Methode den vorherigen Cursor. Damit dieses Restaurieren unter allen Umständen, insbesondere auch nach einem Fehler im **try**-Block, ausgeführt wird, setzen wir die erforderliche Anweisung in einen **finally**-Block, der die **try-catch** – Anweisung erweitert:

```
finally {
   Cursor = oldCursor;
}
```

9.5.9 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 Eigenschaftsfenster zur Registerkarte **Ereignisse**.
- Setzen Sie einen Doppelklick auf das Ereignis **MouseDoubleClick**:



- Daraufhin wird in der Quellcodedatei **MainWindow.xaml.cs** die Instanzmethode `listBox1_MouseDoubleClick()` zu unserer Fensterklasse `MainWindow` angelegt.

Im Rumpf dieser Methode verwenden wir die statische Methode **Start**() der Klasse **Process** im Namensraum **System.Diagnostics** dazu, das Betriebssystem zu beauftragen, mit einem geeigneten Programm den Link im aktuell gewählten **ListBox**-Element zu öffnen. Weil dabei einiges schief gehen kann, verwenden wir eine **try-catch** – Anweisung im Vorgriff auf Abschnitt 10 und zeigen ggf. im **catch**-Block eine Fehlermeldung an, welche auf die **Message**-Eigenschaft des **Exception**-Objekts zugreift:

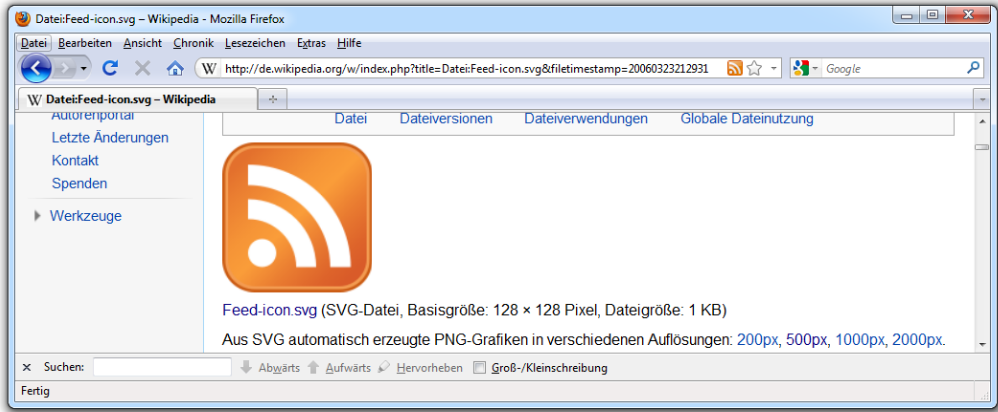
```
private void listBox1_MouseDoubleClick(object sender, MouseButtonEventArgs e) {
   RssItem item = (RssItem)listBox1.SelectedItem;
   // Exist. das SelectedItem? Liefert seine Url-Eigenschaft einen nicht-leeren String?
   if (item != null && !String.IsNullOrEmpty(item.Url))
      try {
         System.Diagnostics.Process.Start(item.Url);
      } catch (Exception ex) {
         MessageBox.Show(this, ex.ToString(), ex.Message);
      }
}
```

9.5.10 Symbole für die Anwendung und das Hauptfenster

Abschließend sollen das Programm und sein Hauptfenster noch ein attraktives Symbol erhalten. Wir beziehen von der Wikipedia-Webseite

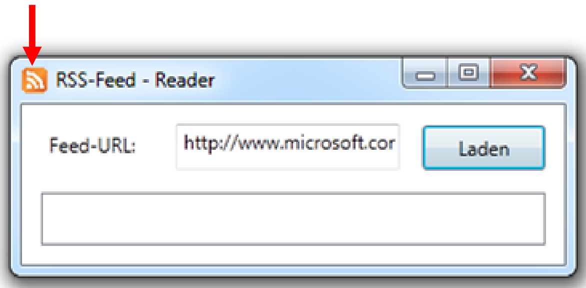
http://de.wikipedia.org/wiki/RSS

eine Bitmap-Datei mit einem RSS-Emblem im PNG-Format (***P****ortable* ***N****etwork* ***G****raphics*) mit einer 500 × 500 Pixelmatrix:



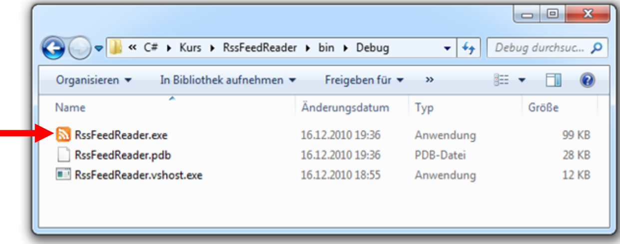
Daraus lässt sich z.B. mit der Freeware **IcoFX**[1]

eine Windows-Symboldatei (Namenserweiterung **.ico**) erstellen, die sich für das Fenstersymbol



und für das Anwendungssymbol

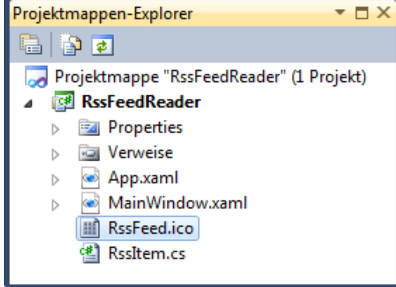
[1] Verfügbar auf der Webseite http://icofx.ro/



eignet.

Gehen Sie im Visual Studio folgendermaßen vor, um aus der **ico**-Datei das Anwendungssymbol und das Fenstersymbol zu beziehen:

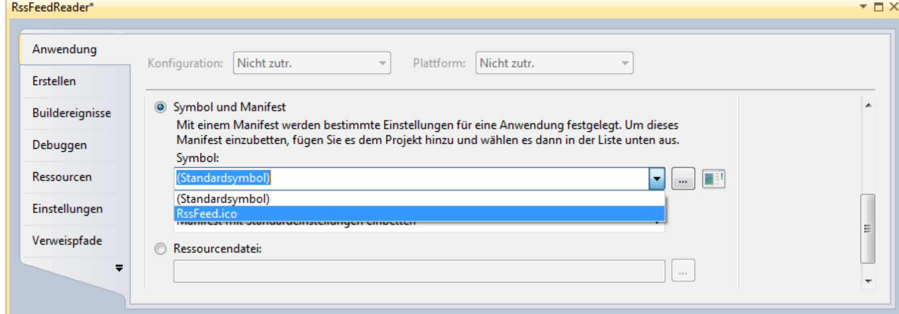
- Kopieren Sie die ICO-Datei in den Projektordner.
- Nehmen Sie die ICO-Datei in das Projekt auf (Item **Hinzufügen > Vorhandenes Element** aus dem Kontextmenü zum Projekt). Anschließend wird die Datei im Projektmappen-Explorer angezeigt:



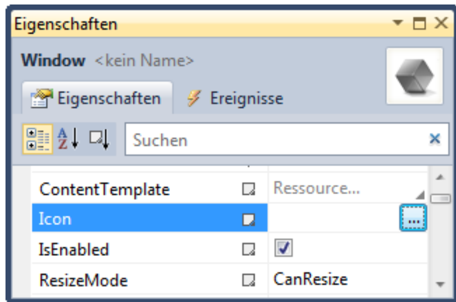
- Öffnen Sie das Fenster mit den Projekteigenschaften über den Menübefehl

  Project > Eigenschaften

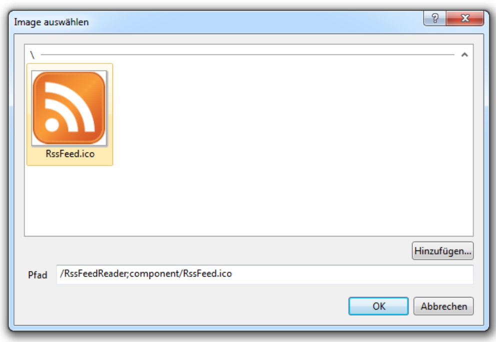
  und wählen Sie im Bereich **Anwendung** das **Symbol**:



- Markieren Sie im WPF-Designer das Hauptfenster, und wählen Sie per Eigenschaftsfenster als Wert der Eigenschaft **Icon**



  nach einem Klick auf den Erweiterungsschalter das Symbol:



9.5.11 Selbstkritik, Ausblick

Wie Sie sicher schon bemerkt haben, lässt sich das Anwendungsfenster nicht verschieben, während eine RSS-Datei geladen wird. Offenbar ist der GUI-Thread ausgelastet und kann nicht auf Anweisungen zur Änderungen seines Auftritts reagieren. Mit der in Kapitel 13 zu erlernenden Multithreading-Programmierung können wir solche Probleme vermeiden.

9.6 Routingereignisse

Auf der WWW-Seite

http://msdn.microsoft.com/de-de/library/ms742806.aspx

liefert Microsoft zwei Charakterisierungen für Routingereignisse.

- **Funktionale Definition**
  Ein Routingereignis ist ein Ereignistyp, der Handler für mehrere Listener in einer Elementstruktur aufrufen kann, und nicht nur für das Objekt, von dem das Ereignis ausgelöst wurde.
- **Implementierungsdefinition**
  Ein Routingereignis ist ein CLR-Ereignis, das auf einer Instanz der **RoutedEvent**-Klasse basiert und vom Windows Presentation Foundation (WPF)-Ereignissystem verarbeitet wird.

Hinsichtlich der automatischen Weiterleitung entlang einer Hierarchie von potentiellen Interessenten zeigt das WPF-Ereignissystem eine Verwandtschaft zur Kommunikation von Ausnahmefehlern durch das Laufzeitsystem (siehe Abschnitt 10).

Wenn man eine Ereignisbehandlungsmethode bei der Ereignisquelle selbst registriert, was wir bisher getan haben, dann bleibt der Unterschied zwischen normalen CLR-Ereignissen und WPF-Ereignissen unsichtbar und irrelevant. Wir haben z.B. für den in Abschnitt 9.5 entwickelten RSS-Feed – Reader mit Assistentenhilfe (in der Code-Behind – Datei zur Hauptfensterklasse `MainWindow`) die folgende Behandlungsmethode zum **Click**-Ereignis der WPF-Klasse **Button** erstellt:

```
private void button1_Click(object sender, RoutedEventArgs e) {
   . . .
}
```

Diese wird in der vom XAML-Compiler generierten Quellcodedatei **MainWindow.g.cs** (vgl. Abschnitt 9.4.4) wie bei einem gewöhnlichen CLR-Ereignis (vgl. Abschnitt 9.3.2.2) „an der Quelle" beim Ereignis registriert:

```
this.button1.Click += new System.Windows.RoutedEventHandler(this.button1_Click);
```

Um diese Verwendung von Routingereignissen mit der vertrauten C# - Syntax für CLR-Ereignisse zu ermöglichen, bieten die meisten WPF-Ereignisse einen Wrapper. Diesen Hinweis nehmen wir vorläufig nur dankbar zur Kenntnis. Relevant wird das Thema erst dann, wenn man eigene WPF-Ereignisse definieren möchte.

Neu am WPF-Ereignissystem ist die Möglichkeit, Behandlungsmethoden nicht (nur) bei der Ereignisquelle (hier: **Button**-Objekt `button1`) zu registrieren, sondern bei beliebigen Knoten (auch mehreren), die im Baum der hierarchisch verschachtelten GUI-Elemente auf der Route von der Ereignisquelle bis zur Wurzel (hier: Hauptfensterobjekt aus der von **Window** abstammenden Klasse `MainWindow`) auftreten.

Die beiden wichtigsten Anwendungsszenarios für Routingereignisse:

- Durch die Ereignisweiterleitung ist es möglich, für mehrere Ereignisquellen mit einem gemeinsamen Knoten im GUI-Baum eine **gemeinsame Ereignisbehandlung durch den übergeordneten Knoten** zu realisieren.
- Eine zweite Hauptanwendung betrifft die **Ereignisbehandlung bei zusammengesetzten Steuerelementen** mit verschachtelten GUI-Elementen. Ein unmittelbar (z.B. von einem **MouseDown**-Ereignis) betroffenes Element kann reagieren, ohne das Ereignis zu „verbrauchen“. In der Hierarchie übergeordnete Elemente erhalten ebenfalls Gelegenheit zur Reaktion. Es besteht aber auch die Möglichkeit, ein Ereignis in einer Behandlungsmethode als behandelt zu deklarieren, so dass übergeordnete GUI-Elemente nicht mehr darauf reagieren. Oft ist es sinnvoll, ein alternatives, für die übergeordneten Elemente besser geeignetes Ereignis zu feuern.

9.6.1 Ein kurzer Blick hinter die Kulissen des WPF-Ereignissystems

Eben war vom Wrapper die Rede, den ein WPF-Ereignis anbieten kann, damit die gewohnte Ereignissyntax verwendbar bleibt. Der folgende Quellcode zeigt, die in der WPF-Klasse **ButtonBase** das Routingereignis **ClickEvent** (ein Objekt der Klasse **RoutedEvent**) unter dem Namen **Click** beim WPF-Ereignissystem registriert wird:

```
public static readonly RoutedEvent ClickEvent = EventManager.RegisterRoutedEvent(
   "Click", RoutingStrategy.Bubble, typeof(RoutedEventHandler), typeof(ButtonBase));
```

Allerdings würde die Zeichenfolge **Click** nicht genügen, um das Routingereignis mit einem CLR-**Wrapper** zu versorgen. Dazu wird das CLR-Ereignis **Click** mit der in Abschnitt beschriebenen Syntax definiert, wobei die explizit realisierten **add**- und **remove**-Methoden für die Verbindung zum WPF-Ereignissystem sorgen:[1]

```
public event RoutedEventHandler Click {
   add {
      AddHandler(ClickEvent, value);
   }
   remove {
      RemoveHandler(ClickEvent, value);
   }
}
```

9.6.2 Routingstrategien

Nach der verwendeten Weiterleitungsstrategie sind folgende Kategorien von WPF-Ereignissen 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 Wurzelelement im GUI-Baum (z.B. ein **Window**-Objekt) zur Behandlung angeboten. Anschließend durchläuft das Ereignis die hierarchisch absteigende Route bis zur

[1] Der Quellcode stammt der Datei **Application.cs**, die über das *Microsoft Reference Source Code Center* (http://referencesource.microsoft.com/) bezogen wurde.

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 erhalten die übergeordneten Elemente im GUI-Baum bis hinauf zur Wurzel Gelegenheit zur Reaktion.
  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 9.6.4).
  Beispiele: **MouseEnter**-Ereignis der Klasse **UIElement**

Ein Tunnel- oder Blasenereignis ist also *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.

Die „echten“ 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 des Elementbaums bis zur Ereignisquelle wird nach einer Behandlungsmethode gesucht.
- Wenn das Tunnelereignis seine Endstation erreicht, wird das zugehörige Blasenereignis ausgelöst.
- Wird das Tunnelereignis hingegen als behandelt markiert, wird das zugehörige Blasenereignis *nicht* ausgelöst. Tunnelereignisse dienen meist dazu, eine Ereignisbehandlung zu stoppen.
- 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.

Eine wichtige Ausnahme vom Paarungsprinzip ist das Blasenereignis **Click** der Klasse **ButtonBase**, zu dem kein Tunnelereignis existiert.

9.6.3 Eine Beobachtungsstudie

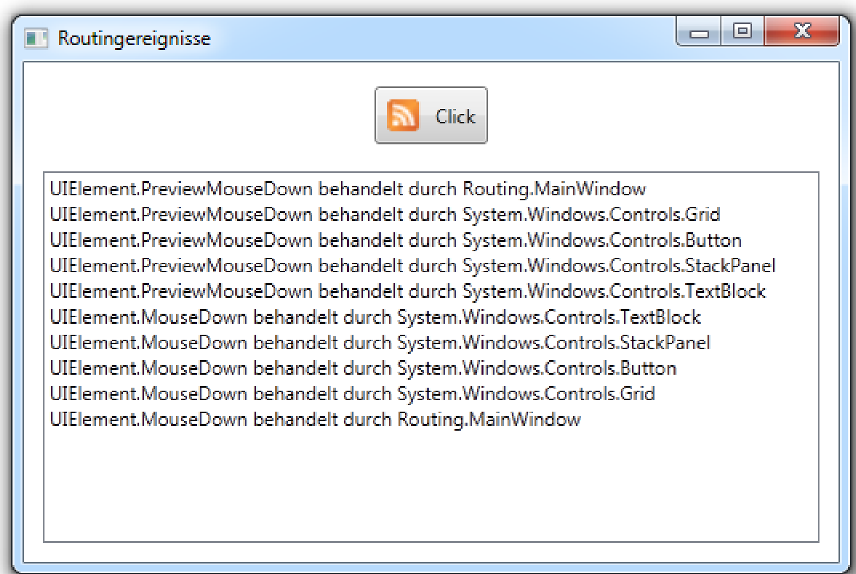
Um das WPF-Ereignissystem bei der Arbeit beobachten zu können, erstellen wir eine Anwendung mit dem folgenden Baum von GUI-Elementen:

Ein Wurzelelement aus der von **Window** abstammenden Klasse `MainWindow`,
darin ein Layoutcontainer aus der Klasse **Grid** (siehe Abschnitt 9.7.1),
in der **Grid**-Zelle (0, 0) u.a. ein Befehlsschalter aus der Klasse **Button**,
darin ein Layoutcontainer aus der Klasse **StackPanel** (siehe Abschnitt 9.7.2),
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 der Ereignisse dient. Das Programm soll die folgenden Routingereignisse bei allen betroffenen GUI-Elementen beobachten:

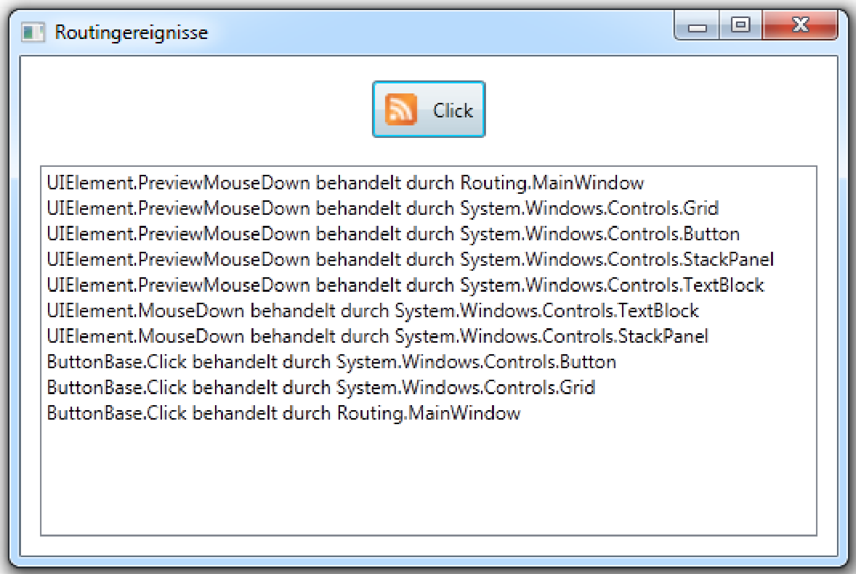
- Tunnelereignis **PreviewMouseDown** aus der Klasse **UIElement**
- Blasenereignis **MouseDown** aus der Klasse **UIElement**
- Blasenereignis **Click** aus der Klasse **ButtonBase** (Basisklasse von **Button**)

Beim folgenden Einsatz



hat das **TextBlock**-Objekt wegen eines *rechten* Mausklicks das Tunnelereignis **PreviewMouse-Down** gefeuert. Das Ereignis ist gemäß Abschnitt 9.6.2 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 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 **MouseEvent** 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 des GUI-Baums nimmt.

Das Programm verwendet für die drei beobachteten Ereignisse jeweils eine Behandlungsmethode, die das als Ereignislauscher verantwortliche GUI-Element am ersten Aktualparameter (**sender** aus der Klasse **Object**) erkennt und protokolliert. Bei einer WPF-Anwendung mit XAML-Unterstützung bringt man die Behandlungsmethoden in der **Code-Behind** – Datei mit der partiellen Definition unter:

```
using System;
   . . .
using System.Windows.Shapes;

namespace Routingereignisse {
   public partial class MainWindow : Window {
      public MainWindow() {
         InitializeComponent();
      }

      private void HandlePreviewMouseDown(object sender, MouseButtonEventArgs e) {
         listBox1.Items.Add("UIElement.PreviewMouseDown behandelt durch " +
                            sender.ToString());
      }

      private void HandleMouseDown(object sender, MouseButtonEventArgs e) {
         listBox1.Items.Add("UIElement.MouseDown behandelt durch " +
                            sender.ToString());
      }

      private void HandleClick(object sender, RoutedEventArgs e) {
         listBox1.Items.Add("ButtonBase.Click behandelt durch " +
                             sender.ToString());
      }
   }
}
```

Um die Protokolleinträge vom **ListBox**-Steuerelement anzeigen zu lassen, werden sie dem zugrunde liegenden Kollektionsobjekt aus der der Klasse **ItemCollection** übergeben, das über die **ListBox**-Eigenschaft **Items** ansprechbar ist und u.a. die Methode **Add()** beherrscht.

Das Registrieren der Ereignisbehandlungsmethoden ist per XAML-Code erheblich einfacher zu bewerkstelligen als mit der C# - Ereignissyntax:

```
<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="button1" HorizontalAlignment="Center" Margin="198,23,192,252"
                PreviewMouseDown="HandlePreviewMouseDown" MouseDown="HandleMouseDown"
                Click="HandleClick">
            <StackPanel Name="stackPanel1" Orientation="Horizontal"
                        PreviewMouseDown="HandlePreviewMouseDown"
                        MouseDown="HandleMouseDown">
                <Image  Name="image1" Width="20" Margin="5"
                        Source="/Routing;component/rss.gif" VerticalAlignment="Center"
                        PreviewMouseDown="HandlePreviewMouseDown"
                        MouseDown="HandleMouseDown"/>
                <TextBlock HorizontalAlignment="Right" Margin="5" Name="textBlock1"
                           Text="Click" VerticalAlignment="Center"
                           PreviewMouseDown="HandlePreviewMouseDown"
                           MouseDown="HandleMouseDown"/>
            </StackPanel>
        </Button>
        <ListBox Margin="12,76,12,12" Name="listBox1" />
    </Grid>
</Window>
```

Wie in Abschnitt 9.4.2.6 beschrieben, kann die XAML-Attributsyntax auch für Ereignisse verwendet werden, wobei der jeweilige Methodenname als Zeichenfolgen-Literal anzugeben ist.

Aufmerksamkeit verdient allerdings die **Click**-Ereignis - Registrierung für das **Window**- bzw. 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** unbekannt ist. Dies ist möglich, wenn ein so genanntes **angefügtes Ereignis** (engl.: *attached event*) vorliegt, was beim **Click**-Ereignis der Fall ist. Bei der Registrierung ist verständlicherweise dem Ereignisnamen der Name der definierenden Klasse (im Beispiel: **ButtonBase**) voranzustellen.

Das komplette Projekt ist im folgenden Ordner zu finden

...\BspUeb\WPF\Delegaten und Ereignisse\Routingereignisse\Beobachtungsstudie

9.6.4 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, d.h.:

- Wenn eine Ereignisinstanz auf ihrer Route bei einem Objekt der Klasse vorbeikommt, wird die statische Behandlungsmethode ausgeführt.
- 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`[1], die eine statische **ClickEvent**-Behandlungsmethode definiert und beim WPF-Ereignissystem registriert:

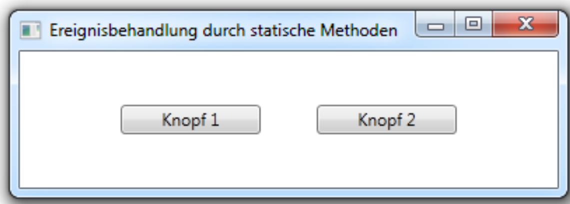
```
class MaiButton : System.Windows.Controls.Button {
   protected static void StaticClickEventHandler(object sender, RoutedEventArgs e) {
      MessageBox.Show("Statischer Click-Handler der Klasse MaiButton");
   }
   static MaiButton () {
      EventManager.RegisterClassHandler(typeof(MaiButton), ClickEvent,
         new System.Windows.RoutedEventHandler(StaticClickHandler), 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 9.6.1) als CLR-Ereignis namens **Click** angesprochen werden kann.

Die folgende WPF-Anwendung (handgestrickt ohne XAML)

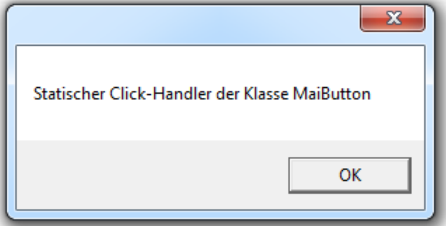
```
class StaticEventHandling : Window {
   StaticEventHandling() {
      Height = 200; Width = 400;
      Title = "Ereignisbehandlung durch statische Methoden";
      StackPanel lm = new StackPanel();
      lm.Orientation = Orientation.Horizontal;
      lm.HorizontalAlignment = HorizontalAlignment.Center;
      lm.VerticalAlignment = VerticalAlignment.Center;
      this.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 System.Windows.RoutedEventHandler(this.knopf1Click);
   }

   private void knopf1Click(object sender, RoutedEventArgs e) {
      MessageBox.Show("Click-Handler von Knopf 1");
   }

   [System.STAThreadAttribute]
   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:

[1] Das Beispiel entstand an einem 1. Mai.



Wurde der links Schalter getroffen, findet danach die objektbezogene Ereignisbehandlung statt:

9.7 Layoutcontainer

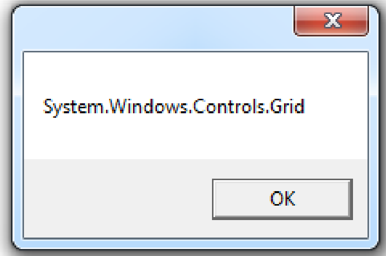
Die in unseren WPF-Anwendungen definierten Fenster (in der Regel **Window**-Abkömmlinge) besitzen mehr oder weniger viele Steuerelemente. Zur Verwaltung der Steuerelemente (z.B. Neuberechnung von Positionen und Größen bei einer Änderung der Fenstergröße) dient ein Layoutcontainer. Bei einer neuen WPF-Anwendung wird für das Hauptfenster ein Layoutcontainer aus der Klasse **System.Windows.Controls.Grid** vorgeschlagen. Dies zeigt ein Blick auf den XAML-Code des Hauptfensters:

```
<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>
```

Dass dieses Objekt dem Hauptfenster über dessen **Content**-Eigenschaft zugewiesen wird, lässt sich z.B. über eine Behandlungsmethode zum **Window**-Ereignis **Loaded**

```
private void Window_Loaded(object sender, RoutedEventArgs e) {
   MessageBox.Show(this.Content.ToString());
}
```

nachweisen:



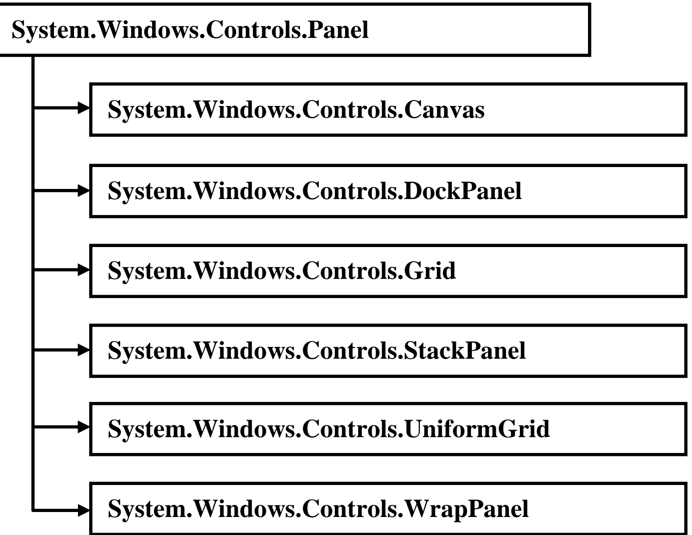
Wenn wir im WPF-Designer Steuerelemente per Drag & Drop aus der Toolbox-Palette auf das Hauptfenster platzieren, landen diese de facto im **Grid**-Element, das für Layout-Angelegenheiten zuständig ist, z.B.:

```
<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" Loaded="Window_Loaded">
    <Grid>
        <Label Content="Label" Height="28" HorizontalAlignment="Left"
               Margin="44,40,0,0" Name="label1" VerticalAlignment="Top" />
    </Grid>
</Window>
```

Statt in die Entwurfsansicht des Fensters kann man die Steuerelemente beim Ziehen und Ablagen übrigens auch direkt in den XAML-Code einfügen, wobei XAML-Code und Fenster-Vorschau von der Entwicklungsumgebung stets synchron gehalten werden.

Von der Gitterstruktur, die der Klassennamen **Grid** erwarten lässt, ist noch nichts zu sehen. Es ist nur eine Zeile und eine Spalte vorhanden. Eingefügte Steuerelemente landen in der **Grid**-Zelle (0, 0). Wie man die enorme **Grid**-Flexibilität nutzt, erfahren Sie ansatzweise in Abschnitt 9.7.1.

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.

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.

9.7.1 Grid

9.7.1.1 Zeilen und Spalten definieren

Um die Zeilen bzw. Spalten eines **Grid**-Objekts per XAML zu definieren, ergänzt man im **Grid**-Element ein untergeordnetes Element mit dem Namen **Grid.RowDefinitions** bzw. **Grid.ColumnDefinitions** (vgl. Abschnitt 9.4.2.3 über die XAML-Syntax für Eigenschaftselemente.

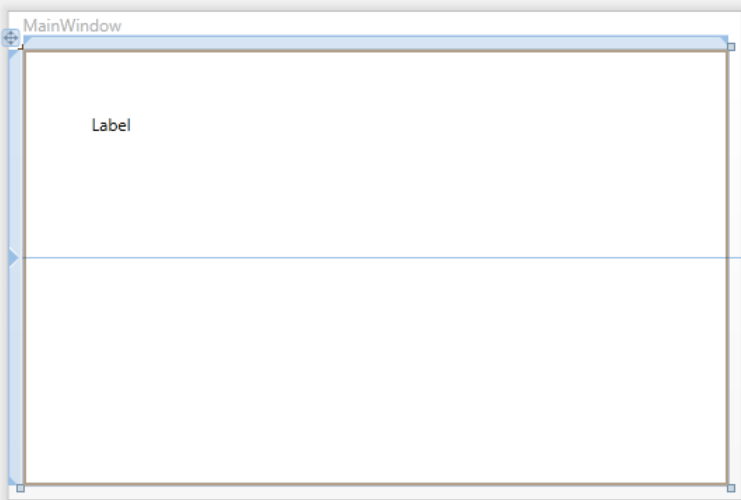
Im Element **Grid.RowDefinitions** werden schließlich einzelne Gitterzeilen durch **RowDefinition**-Elemente vereinbart, z.B.:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition/>
```

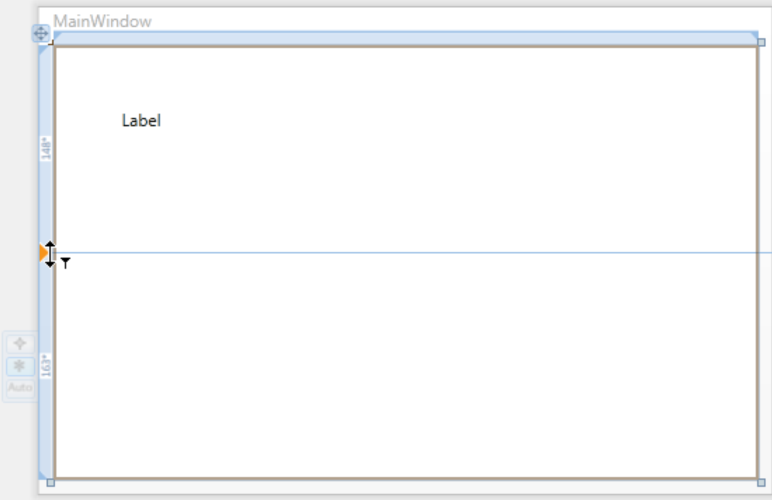
```
            <RowDefinition/>
        </Grid.RowDefinitions>
            .  .  .
    </Grid>
```

Die **Grid**-Eigenschaft **RowDefinitions** ist vom Typ **System.Windows.Controls.RowDefinitionCollection**, und die XAML-Auflistungssyntax (vgl. Abschnitt 9.4.2.5) 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. Per Mausklick auf die Zeilendefinitionszone eines Grid-Objekts



positioniert man eine Trennlinie:



Dabei wird automatisch im XAML-Code die Liste der **RowDefinition**-Elemente erweitert und nötigenfalls auch ein Element von Typ **Grid.RowDefinitions** erstellt.

Analog lässt sich auch die Spaltenstruktur vereinbaren:

```
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="210*" />
            <ColumnDefinition Width="293*" />
        </Grid.ColumnDefinitions>
            .  .  .
    </Grid>
```

Um per WPF-Designer eine Zeilen- oder Spaltendefinition zu **entfernen**, packt man die zugrunde liegende Trennlinie mit der Maus (linke Maustaste drücken und festhalten) und bewegt sie aus dem

Fenstervorschaubereich heraus. Sobald das Loslassen der Maustaste zum Löschen der Trennlinie führen wird, nimmt der Mauszeiger folgende Gestalt an:

9.7.1.2 *Platzaufteilung*

Bei der Zeilendefinition per Mausklick erhalten die Zeilen einen Wert für ihr **Height**-Attribut:

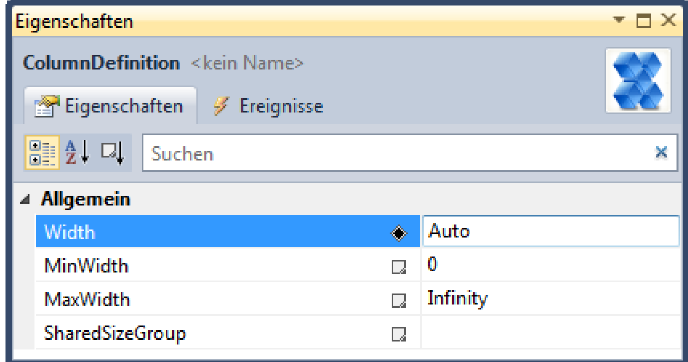
```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="150*" />
        <RowDefinition Height="161*" />
    </Grid.RowDefinitions>
      .  .  .
</Grid>
```

Ein angehängtes Sternchen macht aus den Werten Proportionalitätsfaktoren, so dass im Beispiel bei jeder Ausdehnung des **Grid**-Containers die beiden Zeilenhöhen das Verhältnis (150/161) beibehalten würden.

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 Angabe oder bei der Anforderung
  ```
  <ColumnDefinition Width="*" />
  ```
  beansprucht die Spalte bzw. Zeile den gesamten noch nicht vergebenen Platz.
  Verhalten sich alle Konkurrenten so, dann 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.
- Besitzen alle Spalten bzw. Zeilen einen Wert in Pixeln, werden die angeforderten Pixel ausgegeben, solange der Vorrat reicht, so dass 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 Steuerelemente.

Statt den **Width**- bzw. **Height**-Wert im XAML-Fenster einzutragen, kann man bei passender Markierung auch das Eigenschaftsfenster benutzen, z.B.



Bei einer leeren Platzangabe im Eigenschaftsfenster wird automatisch im XAML-Code ein Sternchen gesetzt.

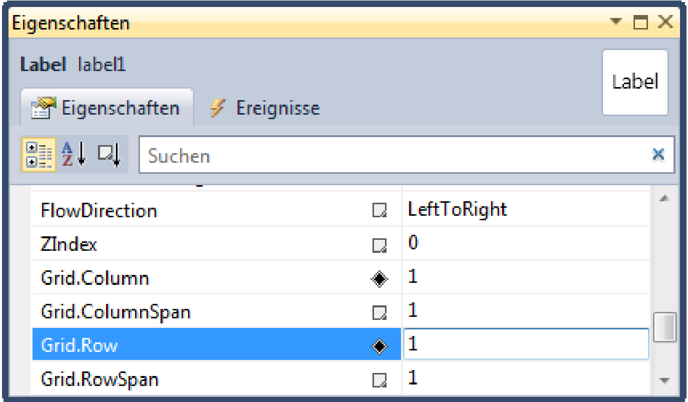
In der Designerzone kann für eine Zeile oder Spalte per Mauszeiger eine Palette mit den drei Alternativen zur Ausdehnungsdefinition (feste Höhe, Proportionalitätsfaktor, **Auto**) angefordert werden:

9.7.1.3 Platzanweisung

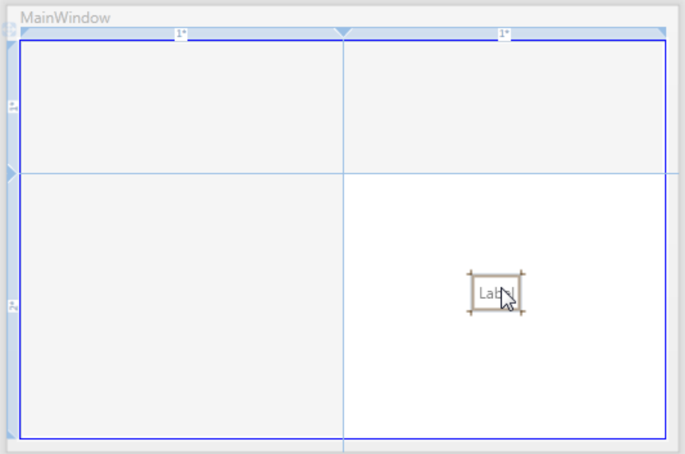
Besitzt ein per **Grid**-Container verwaltetes Steuerelement keine Ortsangabe, landet es in der **Grid**-Zelle (0, 0), also oben links. Um für ein Steuerelement einen alternative Zelle festzulegen, sind im zugehörigen XAML-Element die Attribute **Grid.Row** bzw. **Grid.Column** mit der 0-basierten Nummer der Zeile bzw. Spalte versorgen. Hier wird ein **Label**-Objekt in die Zelle (1, 1) gesetzt:

```
<Grid>
   . . .
    <Label Content="Label" Height="28" HorizontalAlignment="Left"
           Margin="44,40,0,0" Name="label1" VerticalAlignment="Top"
           Grid.Column="1" Grid.Row="1" />
</Grid>
```

An Stelle der direkten Werteingabe kann im Visual Studio bei passender Markierung das Eigenschaftsfenster benutzt werden:



Das bequemste Verfahren zur Wahl einer **Grid**-Zelle bietet der WPF-Designer: Steuerelement per Maus packen und am Ziel ablegen, z.B.:



9.7.1.4 Angefügte Eigenschaften

Bei der eben vorgestellten XAML-Syntax zur Wahl einer **Grid**-Zelle für ein Steuerelement

```
<Grid>
   . . .
    <Label Content="Label" Height="28" HorizontalAlignment="Left"
           Margin="44,40,0,0" Name="label1" VerticalAlignment="Top"
           Grid.Column="1" Grid.Row="1" />
</Grid>
```

fällt auf:

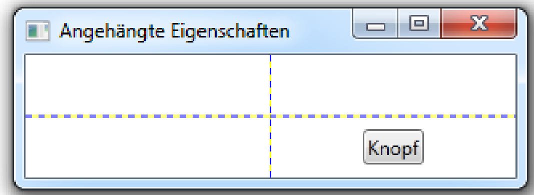
- Während sich in der Steuerelementklasse **Label** zu XAML-Attributen wie **Content**, **Height**, **HorizontalAlignment** etc. jeweils eine korrespondierende Eigenschaft findet, sucht man in der Klassendefinition die Eigenschaften **Row** und **Column** vergeblich.
- Bei den Wertzuweisungen tritt „in statischer Manier“ der Klassenname **Grid** auf.

Grid.Row und **Grid.Column** sind so sogenannte *angefügte Eigenschaften*, (*attached properties*) die von der Klasse **Grid** zur Verfügung gestellt werden, damit die im Layoutcontainer verwalteten GUI-Elemente ihre Positionswünsche formulieren können. Entsprechende Eigenschaften in einer Steuerelement-Basisklasse (z.B. **System.Windows.Controls.Control**) zu definieren, wäre wenig sinnvoll, weil durchaus nicht alle Layoutcontainer über eine Matrixstruktur verfügen.

Hinter der bequemen XAML-Syntax stecken die statischen **Grid**-Methoden **SetRow**() und **SetColumn**(), die als Parameter jeweils eine Steuerelementreferenz 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:

```
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);
}
```

Das **Button**-Objekt erscheint in der gewünschten Zelle des **Grid**-Layoutcontainers:



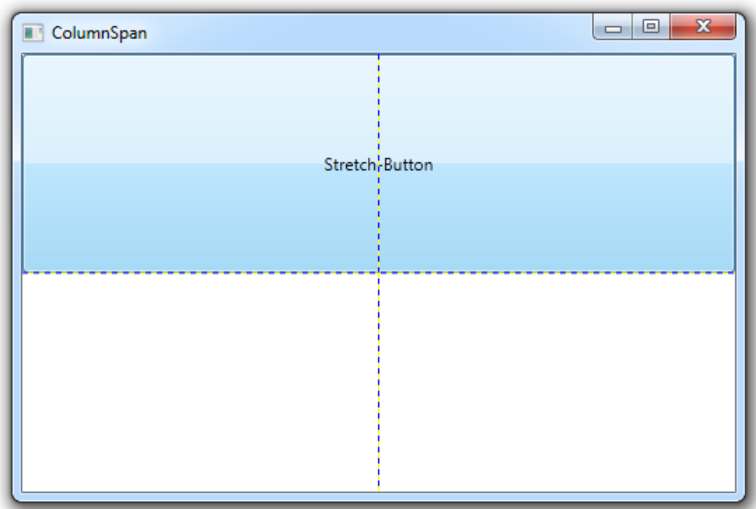
Man legt über die statischen Methodenaufrufe für ein Steuerelement **Grid**-global eine Wunschzelle fest, wobei sich natürlich ein Steuerelement zu einem Zeitpunkt nur in *einem* Container *unmittelbar* befinden kann.

9.7.1.5 Mehrzellige Elemente

Es 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="MainWindow" Height="350" Width="525">
    <Grid ShowGridLines="True">
        <Grid.ColumnDefinitions>
            <ColumnDefinition /> <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition /> <RowDefinition />
        </Grid.RowDefinitions>
        <Button Name="button1" Content="Stretch-Button" Grid.ColumnSpan="2"
                HorizontalAlignment="Stretch" VerticalAlignment="Stretch" />
    </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]



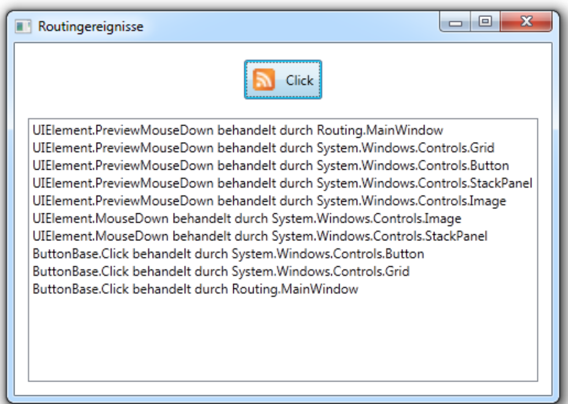
Wenn Sie sich mit der Tabellendefinition in HTML auskennen, kommt ihnen das Attribut **Grid.ColumnSpan** sicher vertraut vor, z.B.:

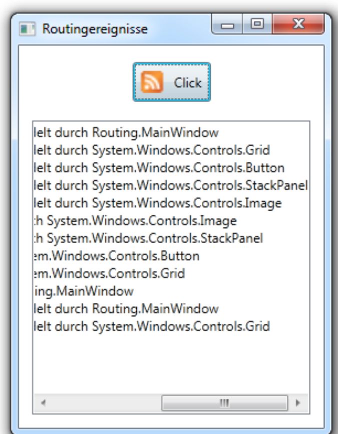
```
<table border="1" width="500" height="500">
    <tr height="100">
        <td colspan="2">
Erstreeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeckt sich über 2 Spalten
        </td>
    </tr>
 ...
</table>
```

Dass sich mehrere Steuerelemente eine **Grid**-Zelle teilen können, haben Sie schon mehrfach beobachtet (z.B. in Abschnitt 9.6.3). Bei einer neuen WPF-Anwendung im Sinne der entsprechenden

[1] Bei der Werten **true** und **false** für Eigenschaften von Typ **bool** ist in XAML die Groß-/Kleinschreibung ausnahmsweise irrelevant.

Projektvorlage unserer Entwicklungsumgebung hat das Hauptfenster zunächst einen einzelligen **Grid**-Container, und alle aus der Toolbox per Drag & Drop übernommenen Steuerelemente landen in der einzigen Zelle. Sofern die Größen und Verankerungen der Elemente geschickt gewählt werden, resultiert ein sinnvoll nutzbares Fenster, z.B.:





Im den meisten Fällen ist es bei einer **Grid**-Zelle mit mehreren Elemente allerdings sinnvoller, einen eingeschachtelten Layoutcontainer in Betrieb zu nehmen (siehe Abschnitt 9.7.1.5).

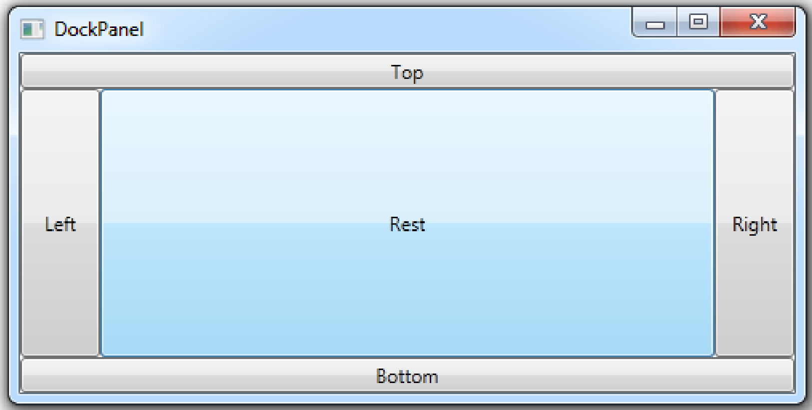
9.7.2 DockPanel

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. 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.

Bei der folgenden Fensterklassendeklaration

```
<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="250" Width="500">
    <DockPanel>
        <Button DockPanel.Dock="Top">Top</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:



Ob die vier Ecken von den horizontalen oder den vertikalen Steuerelementen eingenommen werden, hängt von der Aufnahmereihenfolge ab.

9.7.3 StackPanel

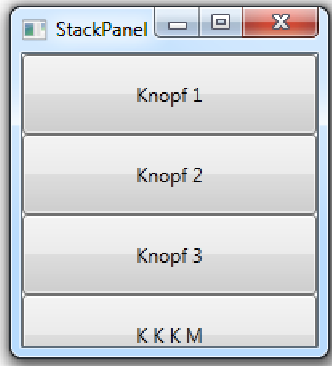
Mit einem Layoutcontainer der Klasse **StackPanel** realisiert man einen simplen vertikalen oder horizontalen Stapel von Steuerelementen. Die Elemente erhalten die gewünschten Ausdehnungen, solange der Vorrat an Pixeln reicht. Ist beim vertikalen Stapel die Gesamthöhe bzw. beim horizontalen Stapel die Gesamtbreite unzureichend, dann gilt: Wer zuerst kommt, malt zuerst. Eine Restraumvergabe wie beim **DockPanel** findet nicht statt.

In der jeweils orthogonalen Richtung werden die Steuerelemente über die gesamte Ausdehnung des Layoutcontainers gestreckt.

Durch die folgende XAML-Deklaration

```
<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>
```

werden vier **Button**-Objekte übereinander gestapelt und dabei über die gesamte Breite gestreckt:



Um einen *horizontalen* Stapel zu erhalten, setzt man die **StackPanel**-Eigenschaft **Orientation** auf den Wert **Horizontal**:

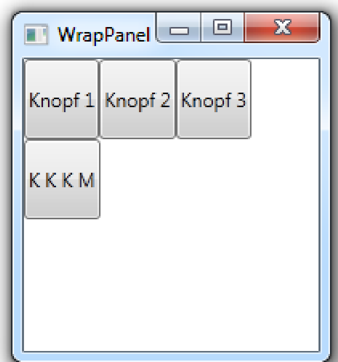
```
<StackPanel Orientation="Horizontal">
. . .
</StackPanel>
```

9.7.4 WrapPanel

Beim **WrapPanel** kommt im Vergleich zum **StackPanel** ein automatischer Spalten- bzw. Zeilenumbruch hinzu. Durch die folgende XAML-Deklaration

```
<WrapPanel Orientation="Horizontal">
    <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>
</WrapPanel>
```

werden vier **Button**-Objekte nebeneinander gestapelt, bis der Platz erschöpft und daher ein Zeilenumbruch erforderlich ist:



9.7.5 UniformGrid

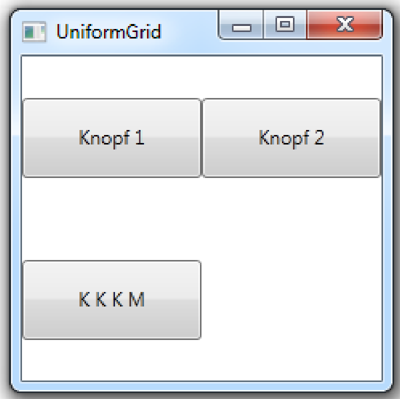
Im Vergleich zum sehr flexiblen **Grid**-Layoutcontainer bestehen beim **UniformGrid**-Container folgende Unterschiede:

- Die Anzahl der Zeilen und Spalten ist gleich.
- Diese Zahl wird nach Bedarf ermittelt.
  Ist die Zahl der Elemente größer als 4 und kleiner als 10, erhält man z.B. einen (3 × 3) - Layoutcontainer.

Aus der folgenden XAML-Deklaration

```
<UniformGrid>
    <Button Height="50">Knopf 1</Button>
    <Button Height="50">Knopf 2</Button>
    <Button Height="50">K K K M</Button>
</UniformGrid>
```

resultiert ein Container mit (2 × 2) gleich großen Zellen. Die drei Elemente werden nacheinander auf die Zellen verteilt, wobei der Spaltenindex schneller läuft:



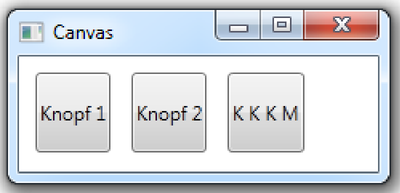
9.7.6 Canvas

Der **Canvas**-Container stellt eine Zeichenfläche ohne jede Layout-Logik bei Größenänderungen zur Verfügung und sollte nur in sehr speziellen Situationen eingesetzt werden. Immerhin werden den verwalteten Elementen die angefügten Eigenschaften **Canvas.Left** und **Canvas.Top** zur Positionierung geboten. Wer unbedingt will, kann mit diesem Container nach schlechter alter Sitte Steuerelemente in liebevoller Kleinarbeit montieren.

Aus der folgenden XAML-Deklaration

```
<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>
```

resultiert das Fenster:



9.7.7 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.:

```
<Grid ShowGridLines="True">
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center"
                Height="Auto" Width="Auto" Name="stackPanel1" >
    </StackPanel>
</Grid>
```

Weil die angefügten Eigenschaften **Grid.Row** und **Grid.Column** fehlen, wird jeweils der Wert 0 angenommen.

Das **StackPanel**–Objekt soll ...

- sich in seiner Zelle horizontal und vertikal zur Mitte hin orientieren,
- seine Breite und Höhe automatisch an den enthaltenen Steuerelementen orientieren.

Es werden drei Button-Objekte in den **StackPanel**-Container eingefügt:

```
<StackPanel HorizontalAlignment="Center" VerticalAlignment="Center"
            Height="Auto" Width="Auto" Name="stackPanel1" >
    <Button Content="Button 1" Height="23" Name="button1" Width="75" Margin="5" />
    <Button Content="Button 2" Height="23" Name="button2" Width="75" Margin="5" />
    <Button Content="Button 3" Height="23" Name="button3" Width="75" Margin="5" />
</StackPanel>
```

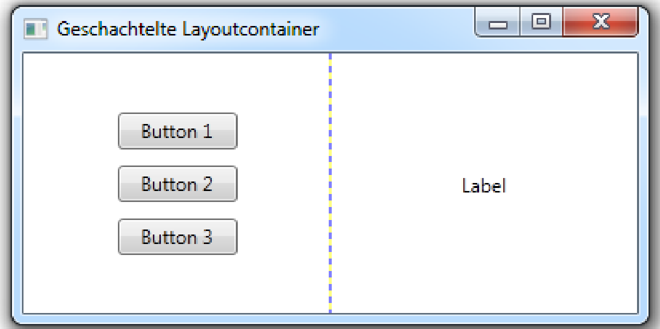
Die **Button**–Objekte sind ...

- 75 Pixel breit und 23 Pixel hoch,
- auf allen Seiten von einem freien Streifen mit 5 Pixeln Breite 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" Name="label1" />
```

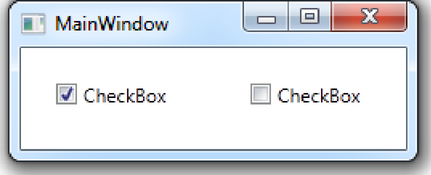
Das Ergebnis:



9.8 Basiswissen über Steuerelemente

Die WPF bietet zahlreiche Klassen zur Realisation von Steuerelementen (Schaltflächen, Textfeldern, Kontrollkästchen, Listen etc.) an, so dass sich für zahllose Programmieraufgaben auf recht bequeme Weise ergonomische und attraktive Bedienoberflächen erstellen lassen. Als Besonderheiten dieser Klassen (z.B. im Vergleich zu Klassen wie **String** oder **Math**) sind zu nennen:

- Ihre Objekte können als (Kind-)fenster **auf dem Bildschirm auftreten** und dabei selbständig **mit dem Benutzer interagieren**. Wenn wir z.B. ein Kontrollkästchen in ein Fenster einbauen, erscheint bzw. verschwindet bei einem Mausklick des Benutzers die Markierung,

  ohne dass wir uns um diese Anpassung der Optik kümmern müssten.
- Die Steuerelemente kommunizieren über **Ereignisse** (im Sinn von Abschnitt 9.3.2) mit anderen Klassen. Will man z.B. über die gesetzte Markierung bei einem Kontrollkästchen informiert werden, registriert man eine Behandlungsmethode bei seinem **Checked**–Ereignis.

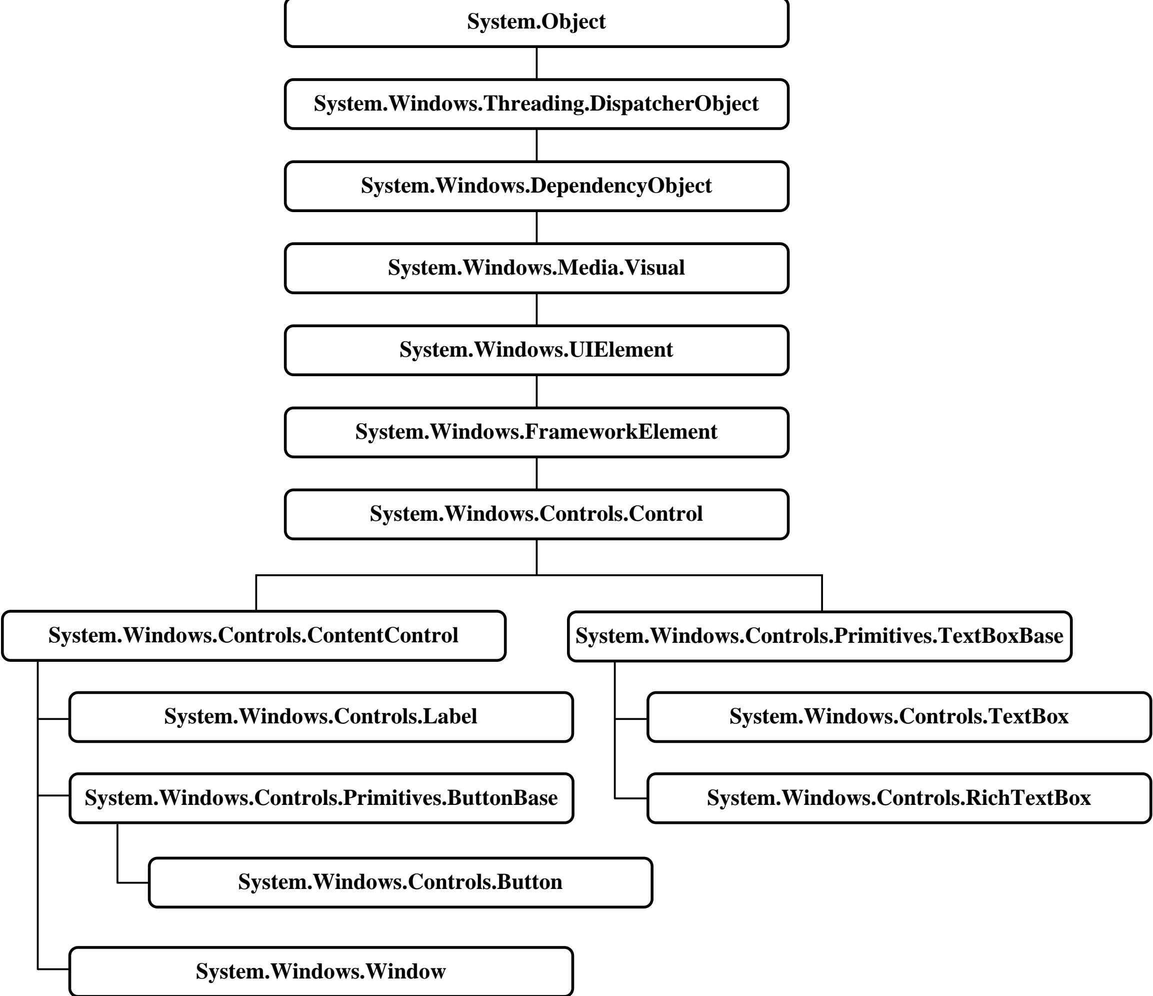
- Ihre **Eigenschaften** (z.B. **Text**, **Height**, **HorizontalAlignment**, **Focusable**) können zur Entwurfszeit über Werzeuge der Entwicklungsumgebungen konfiguriert werden (siehe z.B. Abschnitt 4.10.5).

Als WPF-Besonderheiten gegenüber anderen GUI-Frameworks sind zu erwähnen:

- Die Steuerelemente sind nur für ihr Verhalten zuständig und kümmern sich nicht selbst um ihr Erscheinungsbild, sondern überlassen Vorlage-Objekten diese Aufgabe (siehe Sells & Griffiths 2007, S. 139 und Kapitel 9). Daraus resultieren für die Anwendungsentwicklung relativ einfache Möglichkeiten, das Erscheinungsbild zu beeinflussen.
- Bei den GUI-Elementen ist zu unterscheiden zwischen **Steuerelementen** (z.B. Befehlsschalter, Texteingabefelder) und **Grafikelementen** (z.B. Bilder, Ellipsen), wobei letztere keine Interaktionsfähigkeiten besitzen.

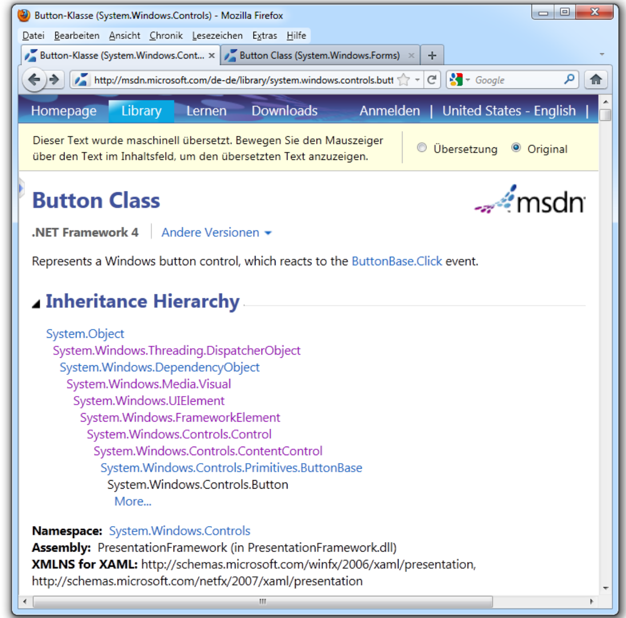
9.8.1 Abstammungsverhältnisse

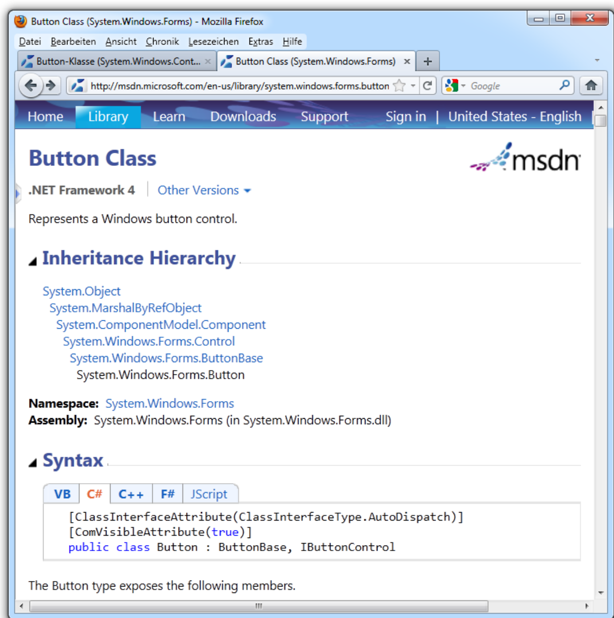
In der .NET - Klassenhierarchie stammen die Steuerelemente (**Button**, **TextBox** etc.) von der Basisklasse **System.Windows.Controls.Control** ab, während rein optische GUI-Elemente (**Image**, **Ellipse** etc.) von der Basisklasse **System.Windows.FrameworkElement** abstammen:



Die von **System.Windows.Controls.ContentControl** abstammenden Klassen beherrschen das WPF-Inhaltsmodell, können also untergeordnete Elemente aufnehmen und sind nicht auf bloße Beschriftung eingeschränkt.

Zu vielen Steuerelemente existieren im früheren GUI-Framework WinForms und in der WPF namensgleiche Klassen, was beim Studium der FCL-Dokumentation zu beachten ist, z.B. bei den beiden **Button**-Klassen.





9.8.2 Einsatz von Steuerelementen in WPF-Anwendungen

Wir besprechen zunächst anhand eines einfachen Beispielprogramms mit **Button**-Steuerelement,

wie man ...

- ein Steuerelement als Memberobjekt in eine Fensterklasse aufnimmt,
- seine Position auf dem Fenster festlegt,
- Eigenschaften zur Entwurfszeit konfiguriert,
- Behandlungsmethoden bei Ereignissen des Steuerelements registriert.

9.8.2.1 Steuerelement als Memberobjekt in eine Fensterklasse aufnehmen

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 ein **Grid**-Container zuständig, der sich vorläufig auf die Zelle (0, 0) beschränkt (nur eine Zeile bzw. Spalte hat). Im XAML-Code zur Fensterklasse erscheint im **Grid**-Element ein Eintrag für das neue Steuerelement, z.B.:

```
<Grid>
    <Button Content="Button" Height="23" HorizontalAlignment="Left"
            Margin="96,48,0,0" VerticalAlignment="Top" Width="75"
            Name="button1" />
```

```
</Grid>
```

Dem XAML-Code ist auch zu entnehmen,

- dass ein Objekt der Klasse **Button** entsteht,
- dass dieses Objekt über den Referenzvariablennamen `button1` ansprechbar ist.
  In einer ernsthaften Anwendung werden Sie die vorgegebenen Namen vermutlich durch individuelle und aussagekräftige Alternativen ersetzen.

Wir haben schon in Abschnitt 9.4.3 untersucht, das beim Programmstart ein WPF-Bestandteil (der BAML-Parser) eine Binär-Variante der XAML-Datei auswertet und z.B. das **Button**-Objekt erzeugt.

9.8.2.2 Steuerelement positionieren

Es ist zumindest schon angeklungen (siehe Abschnitt 9.7.7), dass die WPF verschachtelte Layoutcontainer ermöglicht, und dass Steuerelemente in der Regel in untergeordnete Container eingefügt werden.

Für die Position eines Steuerelements im umgebenden Container bzw. in der Container-Zelle sind folgende Eigenschaften relevant:

- **Margin**
  Mit dieser Eigenschaft legt man fest, wie viel Platz um das Steuerelement herum frei bleiben soll, wobei unterschiedlich detaillierte Angaben möglich sind:
  - mit *einer* Zahl ...
    legt man für alle Seiten denselben Rand fest
  - von *zwei* Zahlen ...
    legt die erste die die beiden horizontalen und die zweite die beiden vertikalen Ränder fest
  - *vier* Zahlen ...
    beziehen sich auf den linken, oberen, unteren und den rechten Rand.
- **HorizontalAlignment**
  Diese Eigenschaften bestimmen die horizontale Ausrichtung eines Steuerelements in Bezug auf den umgebenden Container. Es sind vier Werte (aus der Enumeration **HorizontalAlignment** im Namensraum **System.Windows**) möglich:
  - **Left**
    Das Steuerelement ist am linken Rand verankert.
  - **Right**
    Das Steuerelement ist am rechten Rand verankert.
  - **Stretch**
    Das Steuerelement ist am linken und am rechten Rand verankert, so dass es seine horizontale Ausdehnung zusammen mit dem umgebenden Container verändert, sofern der Container eine entsprechende Funktionalität besitzt.
  - **Center**
    Das orientiert sich horizontal an der Mitte des umgebenden Containers.
- **VerticalAlignment**
  Diese Eigenschaften bestimmen die vertikale Ausrichtung eines Steuerelements in Bezug auf den umgebenden Container. Es sind vier Werte (aus der Enumeration **VerticalAlignment** im Namensraum **System.Windows**) möglich:
  - **Top**
    Das Steuerelement ist am oberen Rand verankert.
  - **Bottom**

Das Steuerelement ist am unteren Rand verankert.
- **Stretch**
  Das Steuerelement ist am oberen und am unteren Rand verankert, so dass es seine vertikale Ausdehnung zusammen mit dem umgebenden Container verändert, sofern der Container eine entsprechende Funktionalität besitzt.
- **Center**
  Das orientiert sich vertikal an der Mitte des umgebenden Containers.

9.8.2.3 Steuerelement konfigurieren

Neben der Position in Bezug auf den umgebenden Container besitzt ein Steuerelement noch zahlreiche weitere Eigenschaften, von denen viele zur Entwurfszeit über das Eigenschaftsfenster der Entwicklungsumgebung oder direkte Einträge im XAML-Code zu gestalten sind, z.B. bei einem **Button**-Objekt:

- **Content**
  Mit der Eigenschaft **Content** legt man den Inhalt der Button-Oberfläche fest, z.B. auf eine Beschriftung:
  ```
  <Button Content="Farbe wechseln" ... />
  ```
- **Width**, **Height**
  Mit den Eigenschaften **Width** und **Height** wählt man eine Breite und eine Höhe. Man kann jeweils neben einer Pixelzahl auch den Wert **Auto** angeben. Dann wird die Breite bzw. Höhe passend zum Inhalt und zum angeforderten Innenrand (siehe unten) gewählt.
- **Padding**
  Mit dieser Eigenschaft legt man eine frei zu haltende Zone am inneren Rand des Steuerelements fest. Die Angaben sind analog zur Eigenschaft **Margin** zu machen (siehe Abschnitt 9.8.2.2).
- **HorizontalContentAlignment**, **VerticalContentAlignment**
  Mit diesen Eigenschaften bestimmt man die Position des Inhalts innerhalb der rechteckigen Steuerelementfläche. Es sind dieselben Werte möglich wie bei **HorizontalAlignment** bzw. **VerticalAlignment** (siehe Abschnitt 9.8.2.2).

9.8.2.4 Ereignisbehandlungsmethoden

Nach Bedarf werden in der Code-Behind – Datei zur Fensterklasse Behandlungsmethoden definiert und bei Ereignissen des Fensters oder der Steuerelemente registriert. Im Beispiel soll das **Click**-Ereignis des **Button**-Objekts behandelt werden. Dazu wählen wir bei markiertem **Button**-Objekt im Eigenschaftsfenster die Registerkarte Ereignisse und setzen einen Doppelklick auf den Eintrag **Click**. Weil das **Click**-Ereignis bei einem Befehlsschalter das Standardereignis ist, führt ein Doppelklick auf den Schalter im WPF-Designer zum selben Ziel: Die Datei **MainWindow.xaml.cs** erscheint im Editor mit einer vorbereiteten Ereignisbehandlungsmethode,

```
private void button1_Click_1(object sender, RoutedEventArgs e) {
}
```

die wir nur noch vervollständigen müssen:

```
private void button1_Click_1(object sender, RoutedEventArgs e) {
  if (Background == Brushes.Beige)
    Background = Brushes.LightGray;
  else
    Background = 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 Ereignisbehandlungsmethode muss im XAML-Code des Hauptfensters deklariert werden, was angenehm einfach ist und zudem meist von der Entwicklungsumgebung erledigt wird:

```
<Button Content="Farbe wechseln" Height="Auto" Width="Auto"
        HorizontalAlignment="Center" VerticalAlignment="Center"
        Name="button1" Margin="10" Padding="10" Click="button1_Click_1" />
```

Wie Sie aus Abschnitt 9.4.3 wissen, ist die vom XAML-Compiler erstellte Datei **MainWindow.g.cs** für das Registrieren der **Click**-Behandlungsmethode per C# Ereignissyntax zuständig. Wir finden die erwartete Anweisung (vgl. Abschnitt 9.3.2.2) in der Methode **Connect**:

```
void System.Windows.Markup.IComponentConnector.Connect(int connectionId, object target) {
    switch (connectionId)
    {
    case 1:
    this.button1 = ((System.Windows.Controls.Button)(target));
      .  .  .
    this.button1.Click += new System.Windows.RoutedEventHandler(this.button1_Click_1);
      .  .  .
    return;
    }
    this._contentLoaded = true;
}
```

9.8.2.5 Ereignisse und On-Methoden

Zur bisher beschriebenen Technik zur Ereignisbehandlung gibt es eine oft benutzte Alternative, die nun vorgestellt werden soll. Bei den für uns relevanten Ereignissen wird vom Laufzeitsystem jeweils eine Methode aufgerufen, deren Name aus dem Präfix **On** und dem Ereignisnamen besteht, z.B.

- **OnClick()**
- **OnMouseDown()**

Weil die **On**-Methoden als **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 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 folgenden Fensterklasse mit dem XAML-Code

```
<Window x:Class="OnMouseDownWindow.MainWindow"
           xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
           xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
           Title="On-Methoden - Demo" Height="200" Width="300" MouseDown="Window_MouseDown">
       <Grid>
           <Label Name="label1"/>
       </Grid>
   </Window>
```

ist die **OnMouseDown**-Methode überschrieben und außerdem die Methode `Window_MouseDown()` beim **MouseDown**-Ereignis des Fensters registriert:

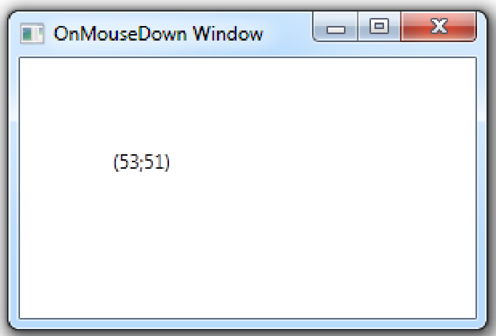
```
using System;
   . . .
using System.Windows.Shapes;

namespace OnMouseDownWindow {
```

```
    public partial class MainWindow : Window {
       public MainWindow() {
          InitializeComponent();
       }
       protected override void OnMouseDown(MouseButtonEventArgs e) {
          base.OnMouseDown(e);
       }
       private void Window_MouseDown(object sender, MouseButtonEventArgs e) {
          label1.Margin = new Thickness(e.GetPosition(this).X, e.GetPosition(this).Y,0,0);
          label1.Content = "(" + e.GetPosition(this).ToString() + ")";
       }
    }
 }
```

Momentan enthält die überschriebene Methode **OnMouseDown**() lediglich einen Aufruf der basisklassenmethode, der grundsätzlich wegen der dort eventuell erfolgenden Aktionen erfolgen sollte. Die `Window_MouseDown()`...

- legt die Position des Labels über dessen **Margin**-Eigenschaft (siehe Abschnitt 9.8.2.2) neu auf die Klickstelle fest, die der Methode über Eigenschaften eines **MouseButtonEvent-Args**-Objekts bekannt wird
- und verarbeitet die Koordinaten der Klickstelle zum neuen Wert der **Content**-Eigenschaft des Labels.

Aufgrund der tätigen Methode `Window_MouseDown()` kann das Programm die Koordinaten einer Klickstelle am Ort des Geschehens anzeigen, z.B.:



Das gelingt ebenso gut, wenn an Stelle der registrierten Behandlungsmethode `Window_Mouse-Down()` die überschreibende On-Methode tätig wird:

```
protected override void OnMouseDown(MouseButtonEventArgs e) {
   base.OnMouseDown(e);
   label1.Margin = new Thickness(e.GetPosition(this).X, e.GetPosition(this).Y,0,0);
   label1.Content = "(" + e.GetPosition(this).ToString() + ")";
}

private void Window_MouseDown(object sender, MouseButtonEventArgs e) {
}
```

Zu Beginn einer überschreibenden On-Methode sollte unbedingt die Basisklassenvariante aufgerufen werden, weil dort oft das Ereignis ausgelöst wird, z.B. in der Methode **OnClick**() der Klasse **ButtonBase**:[1]

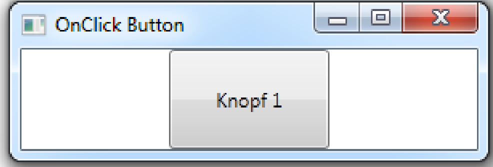
[1] Der Quellcode stammt der Datei **ButtonBase.cs**, die über das *Microsoft Reference Source Code Center* (http://referencesource.microsoft.com/) bezogen wurde.

```
protected virtual void OnClick() {
   RoutedEventArgs newEvent = new RoutedEventArgs(ButtonBase.ClickEvent, this);
   RaiseEvent(newEvent);
   MS.Internal.Commands.CommandHelpers.ExecuteCommandSource(this);
}
```

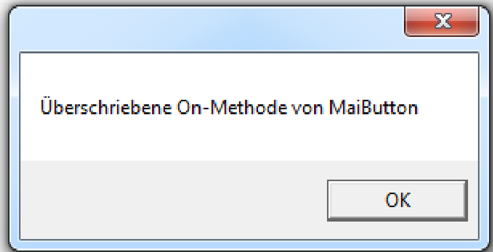
Bei einer Überschreibung ohne Basisklassenmethodenaufruf, werden beim Ereignis registrierte Methoden abgehängt, z.B.:

```
using System;
using System.Windows;
using System.Windows.Controls;


class MaiButton : System.Windows.Controls.Button {
   protected override void OnClick() {
      MessageBox.Show("Überschriebene On-Methode von MaiButton");
   }
}

class OnClickButton : Window {
   OnClickButton() {
      Height = 100; Width = 300;
      Title = "OnClick Button";
      MaiButton k1 = new MaiButton();  k1.Content = "Knopf 1";  k1.Width = 100;
      this.AddChild(k1);
      k1.Click += new System.Windows.RoutedEventHandler(this.knopf1Click);
   }

   private void knopf1Click(object sender, RoutedEventArgs e) {
      MessageBox.Show("Click-Handler von Knopf 1");
   }

   [System.STAThreadAttribute]
   static void Main() {
      new Application().Run(new OnClickButton());
   }
}
```

Aufgrund des Programmierfehlers wird bei einem Mausklick auf den Schalter



nur noch die **OnClick**()-Überschreibung ausgeführt:



Mit der korrekten Überschreibung

```
protected override void OnClick() {
   base.OnClick();
   MessageBox.Show("Überschriebene On-Methode von MaiButton");
}
```

bleiben andere Ereignisbehandlungsmethoden im Spiel, z.B.:

Bei der **Window**-Methode **OnMouseDown()** hat die Unterlassung des Basismethodenaufrufs übrigens keine erkennbaren Konsequenzen. Halten Sie sich trotzdem daran, zu Beginn einer überschreibenden Methode die Basisklassenvariante aufzurufen.

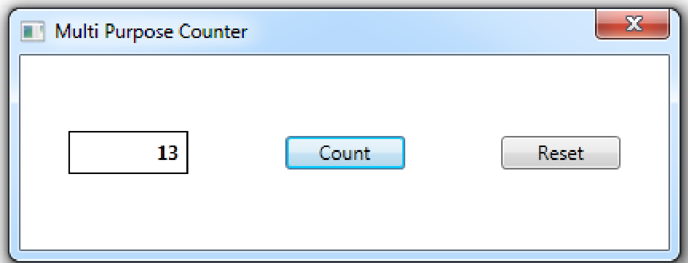
9.8.3 Standardkomponenten

9.8.3.1 Befehlsschalter

Befehlsschalter werden der WPF durch die Klasse **Button** realisiert.

9.8.3.1.1 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="ButtonOnClick">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </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 das **XAML**-Attribut **ResizeMode** (also letztlich die gleichnamige **Window**-Eigenschaft) den Wert **NoResize**.

Die **Height**-Eigenschaften der Steuerelemente werden nicht spezifiziert, und WPF orientiert sich infolgedessen an der Schriftgröße (mit ca. 5 Pixeln Randabstand).

Um für das **Label**-Steuerelement eine fette Schrift zu wählen, erhält das **XAML**-Attribut **FontWeight** den Wert **Bold**. Außerdem erhält das Label einen Rahmen und die (bei einer Zahlenanzeige übliche) rechtsbündige Textausrichtung:

```
BorderThickness="1" BorderBrush="Black"
HorizontalContentAlignment="Right"
```

Die für das **Click**-Ereignis *beider* **Button**-Steuerelemente zuständige Ereignisbehandlungsmethode `ButtonOnClick()` wertet die **Source**-Eigenschaft des zweiten Parameters (vom Typ **RoutedEventArgs**) aus und orientiert ihr Verhalten an der Ereignisquelle:

```
private void ButtonOnClick(object sender, RoutedEventArgs e) {
   long anzahl = long.Parse((String)label.Content);
   if (e.Source== count)
      anzahl++;
   else
      anzahl = 0;
   label.Content = anzahl.ToString();
}
```

Weil es um ein *Routingereignis* geht, kann die Behandlungsmethode dem gemeinsamen **StackPanel**-Container zugeordnet werden, statt sie bei beiden **Button**-Objekten registrieren zu müssen:

```
<Grid Button.Click="ButtonOnClick">
    . . .
</Grid>
```

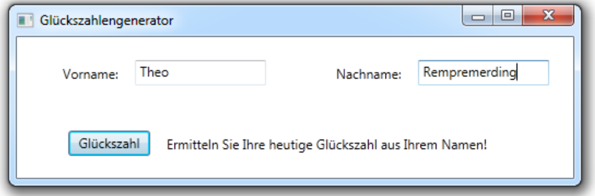
9.8.3.1.2 Standardschaltfläche, Eingabefokus und Zugriffstaste

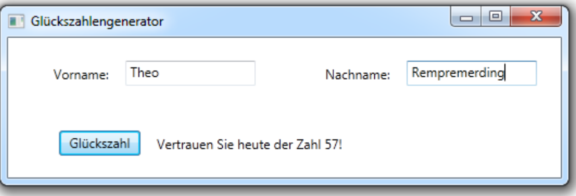
Nach dem Start sehen die beiden Schalter des Beispielprogramms verschieden aus,

und der optisch hervorgehobene Schalter verfügt über das Privileg, auf die Enter-Taste wie auf einen Mausklick zu reagieren, solange keine andere Schaltfläche den Eingabefokus (siehe unten) besitzt. Infolgedessen kann ein Multi-Counter - Benutzer nach dem Start sofort die Enter-Taste zum Zählen verwenden, statt nach der Maus greifen zu müssen. Verantwortlich für dieses Privileg der **Standardschaltfläche** ist das XAML-Attribut (bzw. die **Button**-Eigenschaft) **IsDefault**:

```
<Button ... IsDefault="True"/>
```

Im Beispielprogramm zum **TextBox**-Steuerelement (siehe Abschnitt 9.8.3.3) erhält der Schalter zur Anforderung einer persönlichen Glückszahl die Rolle der Standardschaltfläche. Folglich kann der Schalter auch dann per **Enter**-Taste ausgelöst werden, wenn eines der Textfelder den Eingabefokus besitzt (erkennbar an der Texteinfügemarke):





Im Glückszahlenbeispiel 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" />
```

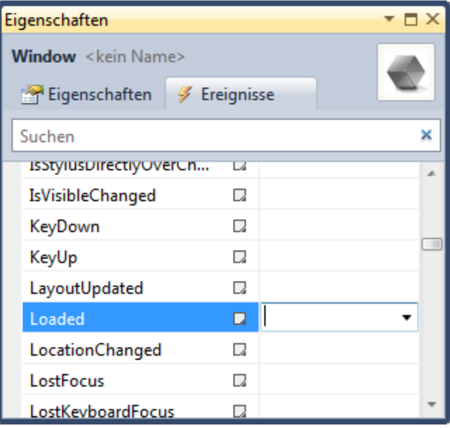
Analog zur Standardschaltflächenernennung über die Eigenschaft **IsDefault** kann ein **Button**-Objekt über die Eigenschaft **IsCancel** auch als **Escape-Schaltfläche** festgelegt werden. Ihr **Click**-Ereignis wird per **Esc**-Taste ausgelöst, sofern keine andere Schaltfläche des Formulars den Eingabefokus besitzt.

Nachdem der `count`-Schalter einen Mausklick erhalten, besitzt er den **Eingabefokus**. Zwar ändert sich (in Windows 7) die Optik nicht, doch nun kann der Schalter auch per Leertaste angesprochen werden. Es wäre nett, wenn der `count`-Schalter schon beim Programmstart den Eingabefokus hätte. Eine Lösungsmöglichkeit besteht darin, die statische Methode **Focus()** der Klasse **Keyboard** aufzurufen, die als Argument eine Referenz auf das privilegierende Objekt erwartet, z.B.:

```
Keyboard.Focus(count);
```

Damit diese Anweisung beim Laden des Hauptfensters ausgeführt wird, stecken wir sie in eine Ereignisbehandlungsmethode zum **Loaded**-Ereignis der Fensterklasse:

- Markieren Sie im WPF-Designer das Hauptfenster.
- Wechseln Sie im Eigenschaftenfenster zur Registerkarte **Ereignisse**.
- Setzen Sie einen Doppelklick auf das 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) {
}
```

- Ergänzen Sie im Rumpf dieser Methode den oben beschriebenen **Focus()**-Aufruf.

Sobald die Fokusverwaltung per **Tabulator**-Taste verwendet wurde, ist ein **Buttton**-Objekt im Fokusbesitz an einer eine innen liegende gepunktete Linie zu erkennen:

Soll das **Click**-Ereignis eines Befehlsschalters über einen **Alt-Tastenbefehl** (eine **Zugriffstaste**) auslösbar sein, 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>
```

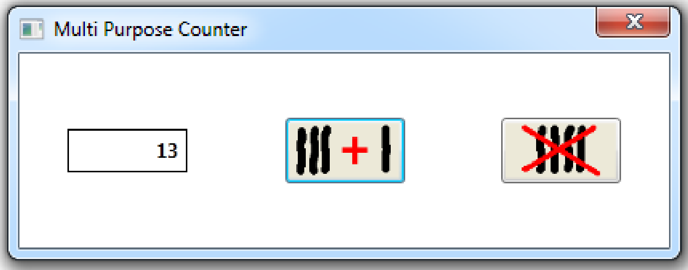
Man wählt die auslösende Alt-Ergänzungstaste, indem man dem zugehörigen Buchstaben im **AccessText**-Inhalt einen Unterstrich voranstellt. Spätestens nach einmaligem Drücken der Alt-Taste ist der Buchstabe im laufenden Programm durch Unterstreichung hervorgehoben, z.B.:

Weil bei **Button**-Objekten häufig eine Zugriffstaste benötigt wird, kann der XAML-Compiler das **AccessText**-Element automatisch einfügen, solange man eine simple Beschriftung anstelle eines zusammengesetzten **Button**-Inhalts verwendet: Setzen Sie einfach in der Beschriftung einen Unterstich vor den Buchstaben zur auslösende Alt-Ergänzungstaste, z.B.:

```
<Button Name="reset" Content="_Reset" Width="75" Grid.Column="2"
    HorizontalAlignment="Center" VerticalAlignment="Center" Click="ButtonOnClick"/>
```

9.8.3.1.3 Bitmaps auf Schaltflächen

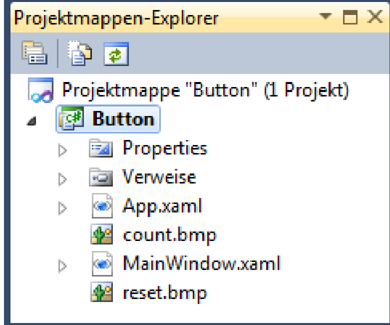
Um dem oben vorgestellten Mehrzweck-Zählprogramm ein individuelles Design zu geben, kann man die Beschriftungen der Schaltflächen durch Grafiken ersetzen (oder ergänzen), z.B.:



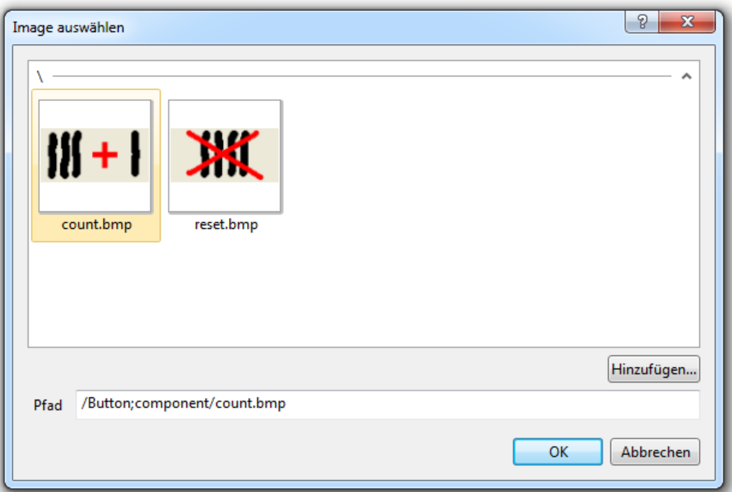
Für das Beispiel sind zunächst Bitmap-Dateien mit einer passenden Pixelmatrix zu erstellen (hier gewählt: 50 Zeilen und 100 Spalten), was z.B. mit der Windows-Zugabe *Paint* geschehen kann.

Gehen Sie im Visual Studio folgendermaßen vor, um die Bitmap-Dateien bequem auf die Schaltflächen zu platzieren:

- 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:



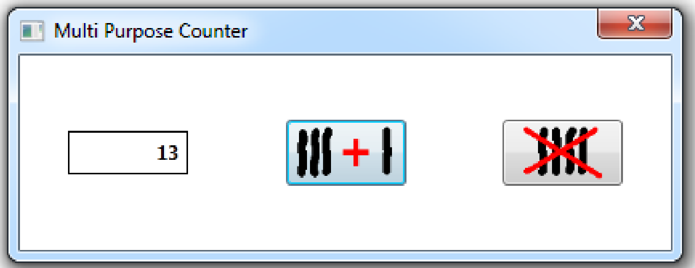
- Ersetzen Sie im XAML-Code bei den Button-Elementen das **Content**-Attribut mit der Beschriftung durch ein Inhaltselement von Typ **Image**.
- Eine Bitmap-Datei als Quelle für die **Image**-Eigenschaft **Source** festzulegen, gelingt besonders bequem über das Eigenschaftsfenster der Entwicklungsumgebung, z.B.:



- Es resultiert der XAML-Code:

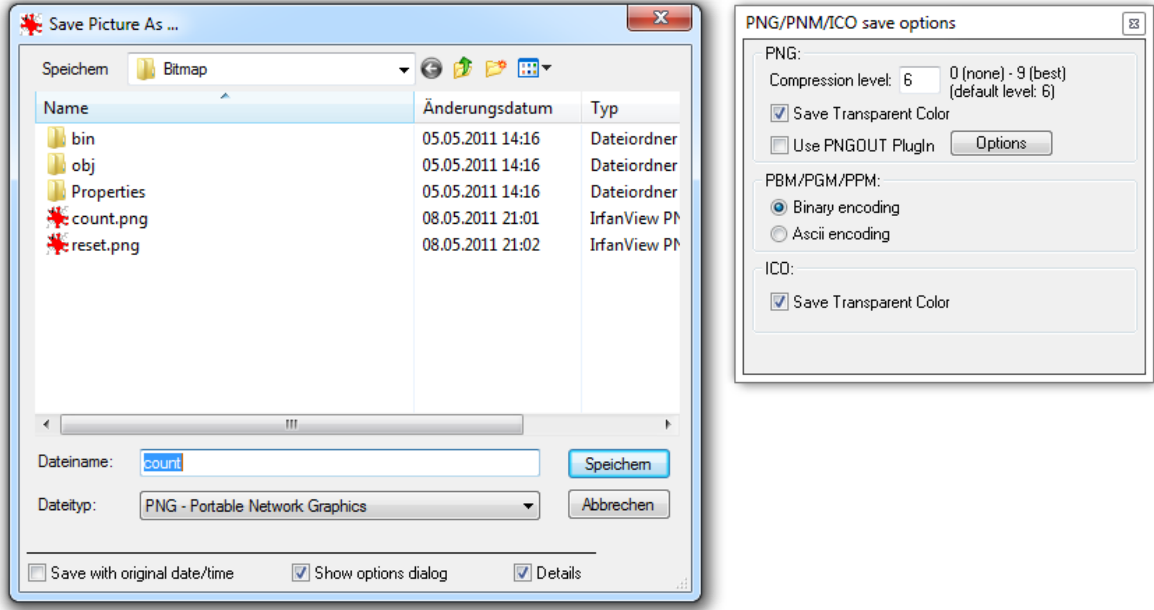
```
<Button Name="count" Width="75" Grid.Column="1" . . . IsDefault="True">
    <Image Source="/Button;component/count.bmp" />
</Button>
```

Macht man die Hintergrundfarbe der verwendeten Bitmaps transparent, harmonieren die bemalten Schalter unabhängig vom eingestellten Windows-Design optisch mit den restlichen Fensterbestandteilen, z.B.:



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/



Soll ein **Button**-Steuerelement eine **Image**-Oberfläche *und* eine Zugriffstaste erhalten, stapelt man zum Bild noch ein **AccessText**-Element (siehe Abschnitt 9.8.3.1.2) auf die Schaltfläche, z.B.:

```
<Button Name="reset" Width="75" Height="40" Grid.Column="2"
        HorizontalAlignment="Center"  VerticalAlignment="Center">
    <StackPanel Orientation="Horizontal">
        <Image Source="/Button;component/reset.png" /> <AccessText>_Reset</AccessText>
    </StackPanel>
</Button>
```

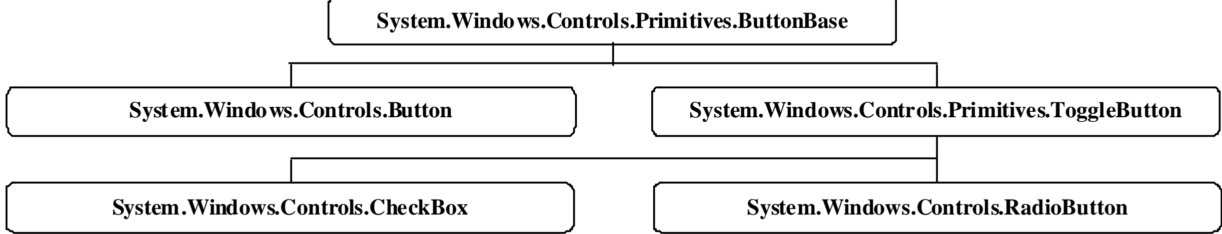
Durch die Breiten von **Button**-Objekt und Bitmap wird ein optischer Auftritt des **AccessText**-Elements verhindert. Die explizite **Button**-Höhe wurde erforderlich, um ein vertikales Strecken durch den horizontalen **StackPanel**-Layoutcontainer zu verhindern (vgl. Abschnitt 9.7.3).

9.8.3.2 Kontrollkästchen und Optionsfelder

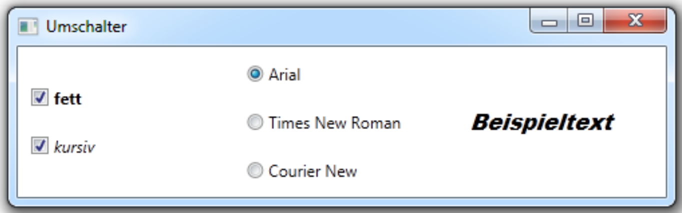
In diesem Abschnitt werden zwei Umschalter vorgestellt:

- 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 in Abschnitt 9.8.3.1 vorgestellten Klasse **Button** von der abstrakten Basisklasse **ButtonBase** ab:



In folgendem Programm kann für den Text einer **Label**-Komponente über zwei Kontrollkästchen der Schriftschnitt und über ein Optionsfeld die Schriftart gewählt werden:



Beim Fensterdesign per XAML-Code kommt ein dreispaltiger **Grid**-Container zum Einsatz. In den beiden ersten Spalten arbeitet jeweils ein **StackPanel**-Container 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 Width="*" /> <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <StackPanel  VerticalAlignment="Center" Button.Click="CheckBoxOnClick">
            <CheckBox Content="fett" Height="16" Margin="10"
                      Name="chkBold" FontWeight="Bold" />
            <CheckBox Content="kursiv" Height="16" Margin="10" Name="chkItalic"
                      FontWeight="Normal" FontStyle="Italic"/>
        </StackPanel>
        <StackPanel  Grid.Column="1" VerticalAlignment="Center"
                     Button.Click="RadioButtonOnClick">
            <RadioButton Name="rbArial" Content="Arial" Height="16" Margin="10"
                         IsChecked="True" />
            <RadioButton Name="rbTimesNewRoman" Content="Times New Roman" Height="16"
                         Margin="10"/>
            <RadioButton Name="rbCourierNew" Content="Courier New" Height="16"
                         Margin="10" />
        </StackPanel>
        <Label  Name="lblTextbeispiel" Grid.Column="2" Content="Beispieltext"
                Margin="10" VerticalAlignment="Center"
                FontFamily="Arial" FontWeight="Normal" 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 mit dem normalen Schnitt in 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**).

Bei beiden Kontrollkästchen (`cbBold` und `cbItalic` genannt[1]) ist dieselbe **Click**-Behandlungsmethode `CheckBoxOnClick()` registriert, die für das separate Ein- bzw. Ausschalten der Schriftattribute **fett** und *kursiv* sorgt:

```
private void CheckBoxOnClick(object sender, RoutedEventArgs e) {
   if (chkBold.IsChecked == true)
      lblTextbeispiel.FontWeight = FontWeights.Heavy;
```

[1] Im Beispielprogramm wird die so genannte *Ungarische Notation* zur Bezeichnung der Instanzvariablennamen verwendet, wobei ein Präfix den Datentyp angibt (z.B. `lblTextbeispiel`). Die in früheren Zeiten der Windows-Programmierung sehr verbreitete Konvention gilt mittlerweile als obsolet und veraltet. Wir verwenden sie im aktuellen Beispiel trotzdem.

```
    else
      lblTextbeispiel.FontWeight = FontWeights.Normal;
    if (chkItalic.IsChecked == true)
      lblTextbeispiel.FontStyle = FontStyles.Italic;
    else
      lblTextbeispiel.FontStyle = FontStyles.Normal;
  }
```

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" Button.Click="CheckBoxOnClick">
    <CheckBox Content="fett" . . . /> <CheckBox Content="kursiv" . . . />
</StackPanel>
```

Im Beispielprogramm wird für alle **RadioButton**-Objekte eine gemeinsame **Click**-Behandlungsmethode verwendet:

```
private void RadioButtonOnClick(object sender, RoutedEventArgs e) {
  if (rbArial.IsChecked == true)
    lblTextbeispiel.FontFamily = ffArial;
  else
    if (rbTimesNewRoman.IsChecked == true)
      lblTextbeispiel.FontFamily = ffTNR;
    else
      lblTextbeispiel.FontFamily = ffCourierNew;
}
```

Hier werden drei Objekte vom Typ **System.Windows.Media.FontFamily** verwendet, die über initialisierte Instanzvariablen ansprechbar sind:

```
public partial class MainWindow : Window {
  FontFamily ffArial = new FontFamily("Arial");
  FontFamily ffTNR = new FontFamily("Times New Roman");
  FontFamily ffCourierNew = new FontFamily("Courier New");
  . . .
}
```

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:

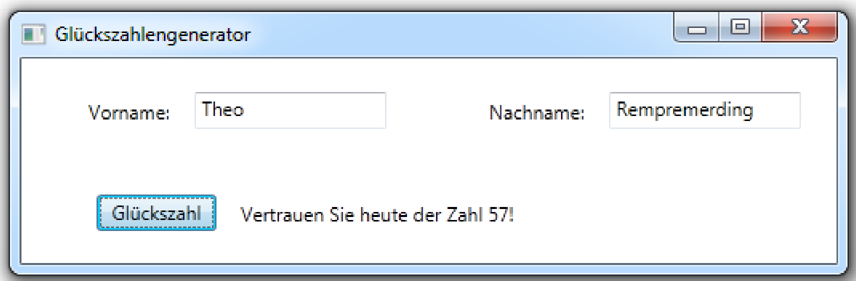
```
<StackPanel Grid.Column="1" VerticalAlignment="Center" Button.Click="RadioButtonOnClick">
    <RadioButton Name="rbArial" . . ./> <RadioButton Name="rbTimesNewRoman" . . . />
    <RadioButton Name="rbCourierNew" . . . />
</StackPanel>
```

Das komplette Projekt finden Sie im Ordner

…\BspUeb\WPF\Steuerelemente\Umschalter

9.8.3.3 Texteingabefelder

Kurze Texteingaben der Benutzer erfasst man in der WPF mit Steuerelementen der Klasse **TextBox**. Auf dem bereits in Abschnitt 9.8.3.1.2 präsentierten Formular eines Programms zur Berechnung der persönlichen Glückszahl in Abhängigkeit vom Vor- und Nachnamen werden zwei **TextBox**-Objekte verwendet:



Das GUI-Design und die Ereignisregistrierung 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="TextBoxOnTextChanged">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" /> <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" /> <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <StackPanel  HorizontalAlignment="Center" VerticalAlignment="Center" Margin="10"
                 Width="Auto" Orientation="Horizontal">
      <Label Content ="Vorname:" Height="28" Margin="10"/>
      <TextBox Height="23" Name="vorname" Width="120" />
    </StackPanel>
    <StackPanel  Grid.Column="1" HorizontalAlignment="Center" VerticalAlignment="Center"
                 Margin="10" Width="Auto" Orientation="Horizontal">
        <Label Content ="Nachname:" Height="28" Margin="10"/>
        <TextBox Height="23" Name="nachname" Width="120" />
    </StackPanel>
    <StackPanel Grid.ColumnSpan="2" Grid.Row="1" Grid.Column="0" HorizontalAlignment="Center"
                Margin="10" VerticalAlignment="Center" Width="Auto" Orientation="Horizontal">
        <Button Content="Glückszahl" Height="23" Name="button1" Width="75"
                Click="button1_Click" IsDefault="True" Focusable="False" />
        <Label Height="28" Name="info" Margin="10" Width="320" />
    </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 button1_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 geändert hat:

```
private void TextBoxOnTextChanged(object sender, TextChangedEventArgs e) {
   if (valToRemove) {
      info.Content = strInst;
      valToRemove = false;
   }
}
```

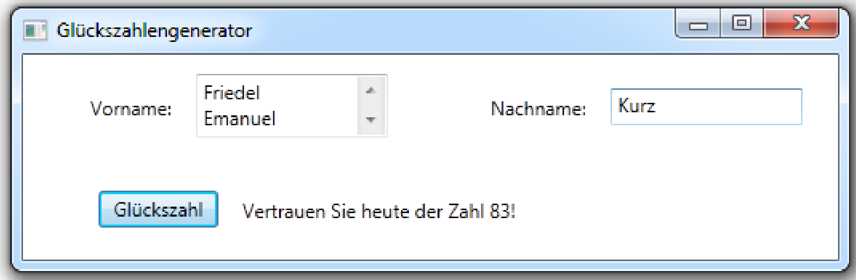
Weil es um ein *Routingereignis* geht, kann die Behandlungsmethode dem gemeinsamen **Grid**-Container zugeordnet werden, statt sie bei *beiden* **TextBox**-Objekten registrieren zu müssen:

```
<Grid TextBox.TextChanged="TextBoxOnTextChanged">
    . . .
</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**-Funktionen

Um ein **mehrzeiliges Textfeld** zu erhalten,



aktiviert man per **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 Height="40" Name="vorname" Width="120" TextChanged="TextBoxOnTextChanged"
         TextWrapping="Wrap" AcceptsReturn="True" VerticalScrollBarVisibility="Visible"/>
```

Die Ernennung das **Button**-Objekts zur Standardschaltfläche (über den Wert **true** der Eigenschaft **IsDefault**, siehe Abschnitt 9.8.3.1.2) verliert ihre Wirkung, sobald im **Enter**-berechtigen Texteingabefeld einmal die **Enter**-Taste gedrückt worden ist.

Trotz der **Multiline**-Option eignet sich die Klasse **TextBox** nur für kurze Texteingaben. Wir werden später mit Hilfe des leistungsfähigeren Steuerelements **RichTextBox** einen kompletten Texteditor erstellen.

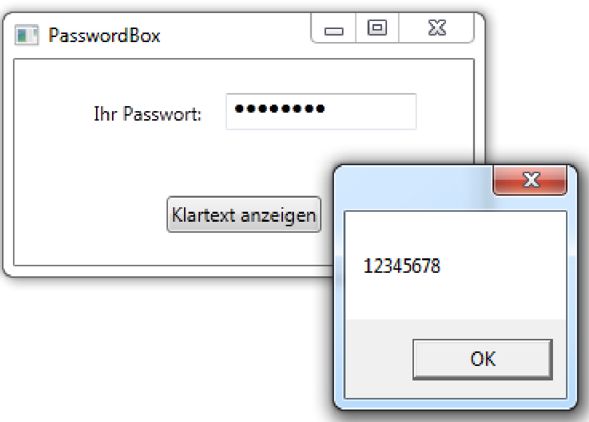
Das vollständige Projekt finden Sie im Ordner

...\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.:

- Anzeige eines Symbols statt der Passwortzeichen
- Eigenschaft **Text** durch die Eigenschaft **Password** ersetzt
- Kein Schreiben in die Zwischenablage

Trotz des geänderten Eigenschaftsnamens ist ein erfasstes Passwort im Klartext vorhanden:

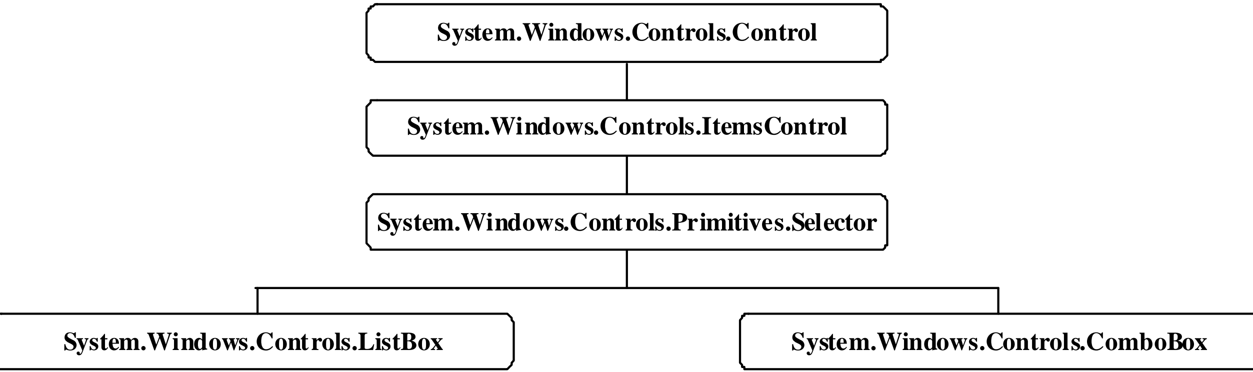


Der XAML-Code zum **PasswordBox**-Objekt:

```
<PasswordBox Height="23" Name="pw" Width="120" Password="default"/>
```

9.8.3.4 Listen- und Kombinationsfelder

In diesem Abschnitt werden die Steuerelementklassen **ListBox** und **ComboBox** vorgestellt, die einen gemeinsamen Stammbaum besitzen:

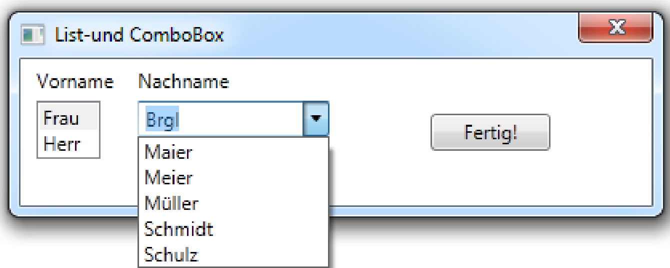


Ein **ListBox**-Steuerelement präsentiert eine Liste von Elementen, die per Mausklick oder geschickte Tastaturbedienung ausgewählt werden können.

Das **ComboBox**-Steuerelement bietet eine Kombination aus einem einzeiligen Textfeld und einer Liste. Um seine Wahl zu treffen, hat der Benutzer *zwei* Möglichkeiten (Wert **true** für die Eigenschaft **IsEditable** vorausgesetzt, siehe unten):

- den Text der gewünschten Option eintragen
- die versteckte Liste aufklappen und die gewünschte Option wählen

In folgendem Programm wird die Angabe des Nachnamens durch eine Liste mit den häufigsten Namen erleichtert:



Ein **ListBox**-Steuerelement kann per XAML-Code über die Auflistungssyntax (siehe Abschnitt 9.4.2.4) mit Elementen versorgt werden

```
<ListBox Name="listBox1" Margin="10,0,0,0" Width="40" HorizontalAlignment="Left">
    <TextBlock>Frau</TextBlock>
    <TextBlock>Herr</TextBlock>
</ListBox>
```

Diese landen in einem Objekt der Klasse **ItemCollection**, das über die (schreibgeschützte) **ListBox**-Inhaltseigenschaft **Items** ansprechbar ist. Das Kollektionsobjekt besitzt etliche Fertigkeiten, z.B. zum Sortieren seiner Elemente nach einer Eigenschaft der Laufzeitklasse:

```
listBox1.Items.SortDescriptions.Add(
            new SortDescription("Text", ListSortDirection.Descending));
```

Weil das **ItemCollection**-Objekt seine Elemente vom Typ **Object** verwaltet, kann beliebige „vorzeigbare" Typen verwenden, z.B.

```
<ListBox Name="listBox1" Margin="10,0,0,0" Width="40"
         HorizontalAlignment="Left" SelectedIndex="0">
    <sys:String>Frau</sys:String>
    <TextBlock>Herr</TextBlock>
</ListBox>
```

Damit dem XAML-Compiler im Beispiel die Klasse **String** und das Namensraumpräfix **sys** bekannt sind, ist eine XML-Namensraumdeklaration (vgl. Abschnitt 9.4.2.1) im **Window**-Wurzelelement erforderlich:

```
xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

Für die Klasse **TextBlock** spricht im Vergleich zur Klasse **String** z.B. die Möglichkeit, nach der **Text**-Eigenschaft zu sortieren.

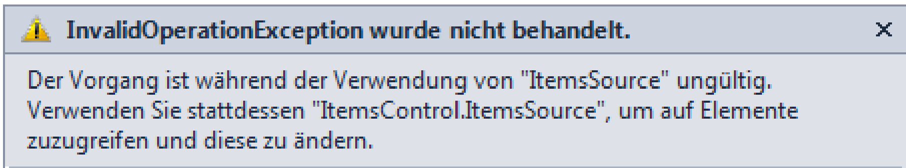
Über die Kollektionsmethoden **Add**(), **Remove**() usw. lässt sich die Elementenliste zur Laufzeit per Programm verändern, z.B.:

```
listBox1.Items.Add(new TextBlock{ Text = "Neu" });
```

Oft befinden sich die in einem **ListBox**-Steuerelement anzuzeigenden Elemente bereits in einer Kollektion und es soll eine Datenbindung (siehe Kapitel 15) zwischen dieser vorhandenen Kollektion als Quelle und dem **ListBox**-Objekt als Ziel vorgenommen werden. Dazu wird das Kollektionsobjekt der **ListBox**-Eigenschaft **ItemSource** zugewiesen. Besonders geeignet sind Objekte der Klasse **ObservableCollection<String>** wegen der Fähigkeit, Änderungen der Listenzusammenstellung per Ereignis an das **ListBox**-Steuerelement zu melden:

```
items = new System.Collections.ObjectModel.ObservableCollection<String>();
listBox1.ItemsSource = items;
```

Sobald der **ItemsSource**-Eigenschaft ein Kollektionsobjekt zugewiesen wurde, ist es nicht mehr möglich, die „eingebaute", per **Items**-Eigenschaft ansprechbare Kollektion des **ListBox**-Steuerelements zu verändern. Ein Versuch führt zu einer **InvalidOperationException**, z.B.:



Im obigen Beispielprogramm mit **ListBox**- und **ComboBox**-Objekt zeigen sich im XAML-Code nur wenige Unterschiede zwischen den beiden **ItemsContol**-Abkömmlingen:

```
<ComboBox Name="comboBox1" Margin="10,0,0,0" Width="120" HorizontalAlignment="Left"
        IsEditable="True">
    <TextBlock>Maier</TextBlock>
     . . .
    <TextBlock>Schulz</TextBlock>
</ComboBox>
```

Ob ein **ComboBox**-Objekt ein Texteingabefeld präsentiert, hängt von der Eigenschaft **IsEditable** ab, die im Beispiel auf den Wert **true** gesetzt wird.

Wie sich in der **Click**-Behandlungsmethode zur Schaltfläche zeigt, ist beim **ListBox**-Objekt die aktuelle Wahl über die Eigenschaft **SelectedItem** ansprechbar:

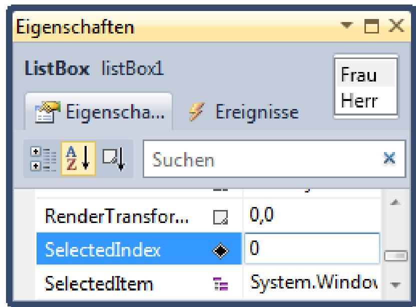
```
private void button1_Click(object sender, RoutedEventArgs e) {
   MessageBox.Show("Guten Tag, " + ((TextBlock)listBox1.SelectedItem).Text + " " +
      comboBox1.Text);
}
```

Weil diese Eigenschaft den Datentyp **Object** besitzt, ist im Beispiel die Textextraktion erst nach einer Typumwandlung möglich. Das **ComboBox**-Objekt stellt seinen aktuell sichtbaren Inhalt über die Eigenschaft **Text** zur Verfügung.

Wenn Sie die die Items eines **ListBox**- oder **ComboBox**-Objekts im Programm ändern wollen (siehe unten), kommt die **ItemsControl**-Eigenschaft **Items** ins Spiel. Sie zeigt auf ein Objekt der Kollektionsklasse **System.Windows.Controls.ItemCollection**, das die Liste der Elemente enthält und diverse Verwaltungsmethoden bietet, z.B.:

- **public int Add(Objekt** ***element*****)**
  Die Liste wird um *ein* Element erweitert.
- **public void Remove(Object** ***element*****)**
  Das angegebene Element wird aus der Liste entfernt.
- **public void Clear()**
  Es werden *alle* Elemente entfernt.

Über die Eigenschaft **SelectedIndex** kann man dafür sorgen, dass der Anwenderbequemlichkeit halber schon beim Öffnen des Fensters ein Listenelement markiert ist, z.B.:



Über dieselbe Eigenschaft lässt sich auch der Indexwert zur aktuellen Auswahl feststellen. Bei einem **ComboBox**-Steuerelement ist allerdings die **String**-Eigenschaft **Text** weit relevanter (siehe oben).

Im obigen Beispielprogramm sind die durchaus zahlreich vorhandenen **ListBox**- bzw. **ComboBox**-Ereignisse (z.B. **SelectionChanged**) nicht vor Interesse.

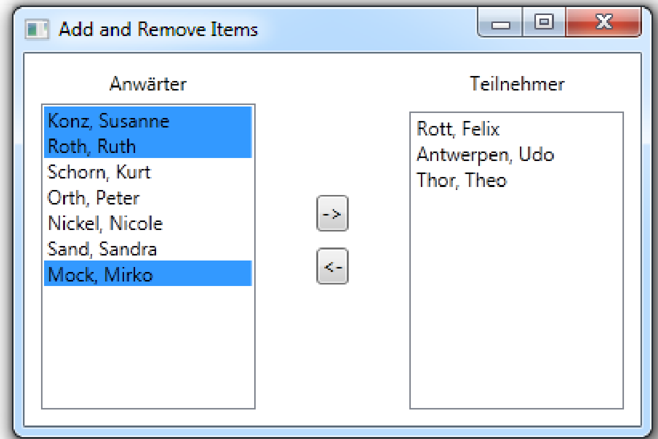
Das vollständige Projekt finden Sie im Ordner

…\BspUeb\WPF\Steuerelemente\Listen\List- und ComboBox

In einem weiteren Beispielprogramm sollen folgende Techniken demonstriert werden:

- Mehrfachauswahl (nur bei **ListBox**-Objekten verfügbar)
- Dynamische Veränderung von Listen

Wir befüllen ein erstes **ListBox**-Objekt mit den Namen von Anwärtern und halten ein zweites **ListBox**-Objekt für die ausgewählten Teilnehmer bereit. Auf dem Anwendungsfenster stehen 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" ResizeMode="CanResize">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="2*" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="2*" />
        </Grid.ColumnDefinitions>
        <DockPanel VerticalAlignment="Stretch">
            <Label Content="Anwärter" DockPanel.Dock="Top" Margin="0,5,0,0"
                   HorizontalAlignment="Center" VerticalAlignment="Top"/>
            <ListBox Name="listeAnwaerter" Margin="10,0,10,10" Height="Auto"
                     HorizontalAlignment="Stretch" 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="ButtonOnClick">
            <Button Content="->" Height="23" HorizontalAlignment="Center" Margin="5"
                    Name="cmdRein" VerticalAlignment="Center" Width="Auto" />
            <Button Content="&lt;-" Height="23" HorizontalAlignment="Center" Margin="5"
                    Name="cmdRaus" VerticalAlignment="Center" Width="Auto"/>
        </StackPanel>
        <DockPanel Grid.Column="2" VerticalAlignment="Stretch">
            <Label Content="Teilnehmer" DockPanel.Dock="Top" Margin="5"
                   HorizontalAlignment="Center" VerticalAlignment="Top"/>
            <ListBox Name="listeTeilnehmer" Margin="10,0,10,10" Height="Auto"
                     HorizontalAlignment="Stretch" SelectionMode="Extended">
            </ListBox>
        </DockPanel>
    </Grid>
</Window>
```

Dem dreispaltigen Top-Level - Container vom Typ **Grid** sind untergeordnet:

- Links und rechts jeweils ein **DockPanel**-Container, das ein **Label**- und ein **ListBox**-Objekt verwaltet, wobei das zuletzt eingefügte **ListBox**-Objekt den gesamten unverbrauchten Raum einnimmt.
- In der Mitte ein **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.

Beim Rücktransportschalters muss die als Beschriftung erwünschte öffnende spitze Klammer wegen der Kollision mit dem XAML-Syntaxelement etwas umständlich notiert werden:

```
Content="&lt;-"
```

In der Ereignismethode 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* Elemente zeigt.

```
private void ButtonOnClick(object sender, RoutedEventArgs e) {
   Object[] tempAnwaerter = new Object[listeAnwaerter.SelectedItems.Count];
   Object[] tempTeilnehmer = new Object[listeTeilnehmer.SelectedItems.Count];
if (e.Source == cmdRein) {
      listeAnwaerter.SelectedItems.CopyTo(tempAnwaerter, 0);
      foreach (object anw in tempAnwaerter) {
         listeAnwaerter.Items.Remove(anw);
         listeTeilnehmer.Items.Add(anw);
      }
   } else {
      listeTeilnehmer.SelectedItems.CopyTo(tempTeilnehmer, 0);
      foreach (object anw in tempTeilnehmer) {
         listeTeilnehmer.Items.Remove(anw);
         listeAnwaerter.Items.Add(anw);
      }
   }
}
```

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 Grid.Column="1" VerticalAlignment="Center"
            Button.Click="ButtonOnClick">
    <Button Content="->" . . . /> <Button Content="&lt;-" . . . />
</StackPanel>
```

In der Ereignismethode kann daher die Ereignisquelle *nicht* über den Parameter **sender** festgestellt werden, 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 finden Sie im Ordner

…\BspUeb\WPF\Steuerelemente\Listen\AddRemoveItems

9.8.3.5 ToolTip

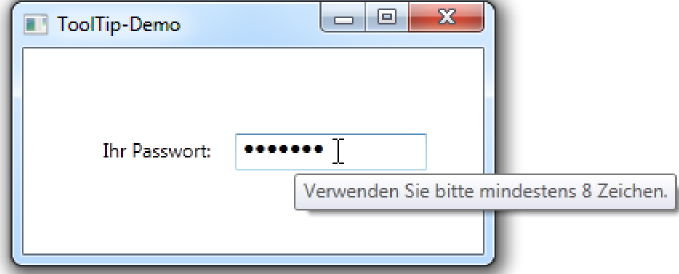
Über ein **ToolTip**-Objekt realisiert man eine in der Regel nur vorübergehend sichtbare Infoanzeige zu einem Steuerelement:

- Die Anzeige erscheint in der Nähe des zu erläuternden 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: 5000 Millisekunden).

Statt im XAML-Code ein Eigenschaftselement vom Typ **ToolTip** zu deklarieren, kann 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:



Um die voreingestellte Anzeigedauer von 5000 Millisekunden zu verändert, benutzt man die angefügte Eigenschaft **ShowDuration** der Klasse **ToolTipService**, z.B.:

```
<PasswordBox ToolTip="Verwenden Sie bitte mindestens 8 Zeichen."
             ToolTipService.ShowDuration="10000"
             Height="23" Name="pw" Width="120" Password="default" />
```

Über angefügte Eigenschaften der Klasse **ToolTipService** lassen sich noch einige weiter **ToolTip**-Verhaltensweisen ändern (z.B. Startverzögerung, Ausstattung mit einem Schlagschatten).

9.9 Zusammenfassung zu Kapitel 9

Charakteristische Merkmale einer GUI-Anwendung sind:

- Verwendung von **Steuerelementen** zur Benutzerinteraktion (z.B. Schalter, Optionsfelder, Menüs etc.)
- Unterstützung von **grafikorientierten Eingabegeräten** (z.B. Maus, Touch Screen)
- **Benutzergesteuerter**, **ereignisorientierter** Ablauf

> Eine GUI-Anwendung wartet die meiste Zeit darauf, dass eine Ihrer Methoden über ein (meist vom Benutzer ausgelöstes) Ereignis aufgerufen wird.

Leicht übertreibend kann man sagen:

Ein Konsolenprogramm	kommandiert den Benutzer, benutzt das Laufzeitsystem.
Ein GUI-Programm:	lässt den Benutzer zwischen zahlreichen Bedienelementen wählen, ist eine Ansammlung von Ereignisbehandlungsroutinen (Call Back - Routinen), die vom Laufzeitsystem aufgerufen werden.

Die WPF bietet leistungsfähige Klassen zur GUI-Programmierung. Als elementare WPF-Klassen haben Sie kennen gelernt:

- **Window**
  Alle Fenster einer Windows-Anwendung mit WPF-Technik werden über Objekte einer von **Window** abstammenden Klasse verwaltet. Das **Hauptfenster** einer WPF-Anwendung spielt eine besondere Rolle:
  - Es wird beim Programmstart automatisch angezeigt.
  - Ein sinnvoll entworfenes WPF-Programm endet beim Schließen des Hauptfensters.
- **Application**
  Diese Klasse bietet u.a. essentielle Methoden zum Betreiben einer WPF-Anwendung. Durch den Aufruf der **Application**–Methode **Run()** kommt **Nachrichtenschleife** der Anwendung in Gang.

Delegaten

Ein Objekt eines Delegatentyps zeigt auf eine Methode (oder auf eine Liste von Methoden) mit einer bestimmten Signatur. Beim Aufruf eines Delegatenobjekts werden alle Methoden in seiner Aufrufliste nacheinander ausgeführt.

CLR-Ereignisse

Bei einem allgemeinen CLR - Ereignis ist eine private Delegatenvariable im Spiel, die auf ein Delegatenobjekt mit einer Aufrufliste zeigt. Bei bestimmten Gelegenheiten (z.B. bei einem Klick auf einen Befehlsschalter) wird die Delegatenvariable vom Ereignisanbieter aufgerufen. Um auf ein Ereignis reagieren zu können, registriert man dort eine Behandlungsmethode:

- Man definiert eine Methode mit passender Signatur
- und befördert diese per Aktualisierungsoperator „+=“ in die Aufrufliste zum Ereignisobjekt. Hinter dem überladenen Aktualisierungsoperator steckt eine öffentliche Methode, die den regulierten Zugriff auf das Ereignisobjekt erlaubt.

Routingereignisse

Ein CLR-Ereignis kann nur an der Quelle behandelt werden. Demgegenüber werden die Routingereignisse der WPF allen Knoten verfügbar gemacht, die sich im Baum der hierarchisch verschachtelten GUI-Elemente auf der Route von der Ereignisquelle bis zur Wurzel (bei unseren Beispielen: Hauptfensterobjekt aus einer von **Window** abstammenden Klasse) befinden. Je nach Routingstrategie startet die Route an der Wurzel (Tunnelereignis) oder an der Quelle (Blasenereignis). So wird es etwa ermöglicht, einem hochrangigen Container eine Behandlungsmethode für das **Button**-Ereignis **Click** anzuheften, die somit für mehrere enthaltene **Button**-Objekte zuständig ist. Bei den uneigentlichen Routingereignissen mit der Routingstrategie *Direkt* findet *keine* Ereignisweiterleitung statt.

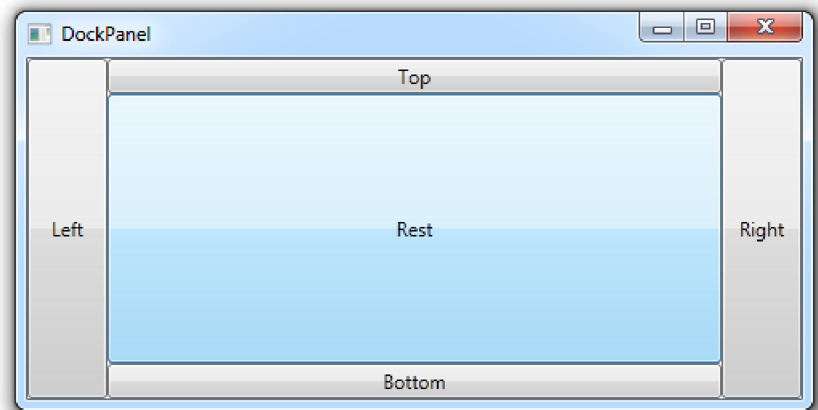
Steuerelemente

Die WPF bietet zahlreiche Klassen zur Realisation von Steuerelementen (z.B. Befehlsschalter, Textfelder, Kontrollkästchen, Auswahllisten) an. Objekte dieser Klassen besitzen einige Besonderheiten:

- Sie erscheinen als (Kind-)fenster eines Containers auf dem Bildschirm.
- Sie interagieren selbständig mit dem Benutzer.
- Sie kommunizieren über Ereignisse mit anderen Klassen.
- Ihre Eigenschaften (z.B. **Text**, **Height**, **HorizontzalAlignment**) können zur Entwurfszeit über Werzeuge der Entwicklungsumgebungen konfiguriert werden.

9.10 Übungsaufgaben zu Kapitel 9

1) Ändern Sie das **DockPanel**-Beispielprogramm in Abschnitt 9.7.2 so ab, dass die vier Ecken von den vertikalen Steuerelementen belegt werden:



10 Ausnahmebehandlung

Durch Programmierfehler (z.B. versuchter Feldzugriff mit ungültigem Indexwert) oder durch besondere Umstände (z.B. z.B. irreguläre Eingabedaten, Speichermangel, unterbrochene Netzverbindungen) kann die reguläre Ausführung einer Methode scheitern. Solche Probleme sollten entdeckt, behoben oder zusammen mit hilfreichen Informationen an den Aufrufer der Methode und eventuell schlussendlich an den Benutzer gemeldet werden, statt zu einem Absturz und sogar zu einem Datenverlust zu führen.

C# bietet ein modernes Verfahren zur Meldung und Behandlung von Problemen: 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 wird über das Problem informiert und mit relevanten Daten für die Behandlung versorgt.

Die Initiative beim Auslösen einer Ausnahme kann ausgehen …

- von der **CLR**
  Wir dürfen annehmen, dass die CLR praktisch immer stabil bleibt. Entdeckt sie einen Fehler, der nicht zu schwerwiegend ist und vom Benutzerprogramm behoben werden kann (z.B. versuchter Feldzugriff mit ungültigem Indexwert), wirft sie ein Ausnahmeobjekt aus einer Klasse, die meist von **System.Exception** abstammt.
- vom **Benutzerprogramm**, wozu auch die verwendeten Bibliotheksklassen gehören
  In jeder Methode kann mit der **throw**-Anweisung (siehe Abschnitt 10.5) eine Ausnahme erzeugt werden.

Die unmittelbar von einer Ausnahme betroffene Methode steht oft am Ende einer Sequenz verschachtelter Methodenaufrufe, und entlang der Aufrufersequenz haben die beteiligten Methoden jeweils folgende Reaktionsmöglichkeiten:

- Ausnahmeobjekt abfangen und das Problem behandeln
  Dabei ist der im Ausnahmeobjekt enthaltene Unfallbericht von Nutzen. Scheidet die Fortführung des ursprünglichen Handlungsplans auch nach der Ausnahmebehandlung aus, sollte erneut ein Ausnahmeobjekt geworfen werden, entweder das ursprüngliche oder ein informativeres.
- Ausnahmeobjekt ignorieren und 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 eigene Ausnahmeklassen definiert.

Man kann von keinem Programm erwarten, dass es unter allen widrigen Umständen normal funktioniert. Doch müssen Datenverluste verhindert werden, und der Benutzer sollte nach Möglichkeit eine nützliche Information zum aufgetretenen Problem erhalten. Bei vielen Methodenaufrufen ist es realistisch und erforderlich, auf ein Scheitern vorbereitet zu sein. Dies folgt schon aus **Murphy's Law** (zitiert nach Wikipedia):

> „Whatever can go wrong, will go wrong."

10.1 Unbehandelte Ausnahmen

Findet die CLR zu einer geworfenen Ausnahme entlang der Aufrufersequenz bis hinauf zur **Main()**-Methode keine Behandlungsroutine, wird das Programm mit einer Fehlermeldung beendet. Das folgende Programm soll die Fakultät zu einer Zahl berechnen, die beim Start als Befehlszeilenargument übergeben wird. Dabei beschränkt sich die **Main()**-Methode auf die eigentliche Fakultätsberechnung und überlässt die Konvertierung und Validierung des übergebenen Strings der Methode `Kon2Int()`. Diese wiederum stützt sich bei der Konvertierung auf die statische Methode **ToInt32**() der FCL-Klasse **Convert**:

```
using System;

class Fakul {
   static String instr;

   static int Kon2Int() {
      int arg = Convert.ToInt32(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);
      } else
         instr = args[0];
      int argument = Kon2Int();
      if (argument != -1) {
         double fakul = 1.0;
         for (int i = 1; i <= argument; i++)
               fakul = fakul * i;
         Console.WriteLine("Fakultät von {0}: {1}", instr, fakul);
      } else
         Console.WriteLine("Keine ganze Zahl im Intervall [0, 170]: " + instr);
   }
}
```

Das Programm ist durchaus bemüht, einige kritische Situationen zu vermeiden. **Main()** überprüft, ob `args[0]` tatsächlich vorhanden ist, bevor dieser String beim Aufruf der Methode `Kon2Int()` als Parameter verwendet wird. Damit wird verhindert, dass es zu einer **IndexOutOfRange-Exception** kommt, wenn der Benutzer das Programm *ohne* Befehlszeilenargument startet. In diesem Fall reagiert **Main()** mit dem sofortigen Abbruch des Programms durch Aufruf der Methode **Environment.Exit()**, der als Aktualparameter ein **Exitcode** übergeben wird. Dieser landet beim Betriebsystem und steht in der Umgebungsvariablen ERRORLEVEL zur Verfügung, z.B.:

```
>fakul
Kein Argument angegeben

>echo %ERRORLEVEL%
1
```

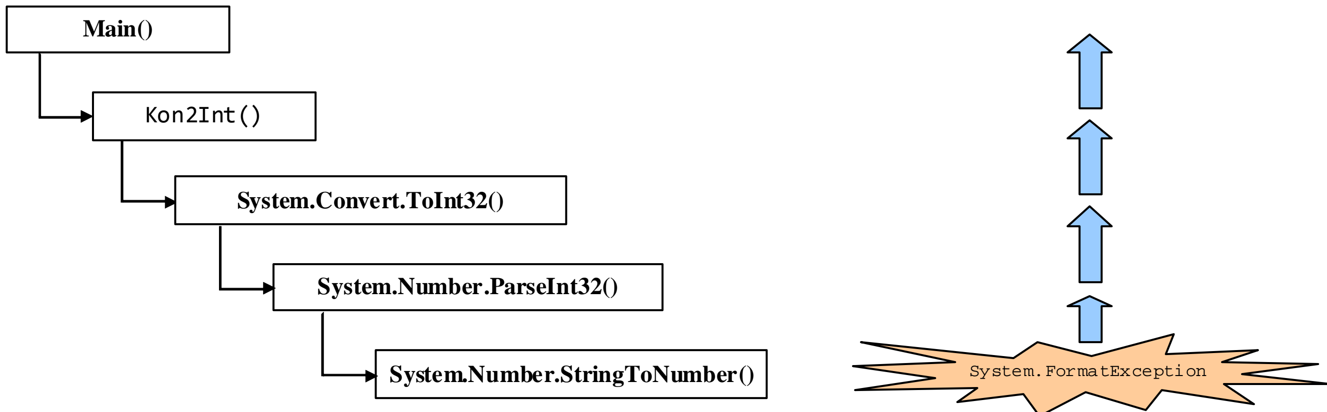
Nach einem störungsfrei verlaufenen Programmeinsatz enthält ERRORLEVEL den Exitcode 0.

Die Reaktion auf ein fehlendes Befehlszeilenargument kann als akzeptabel gelten:

- An Stelle einer für Benutzer irritierenden und wenig hilfreichen Ausnahmemeldung durch das Laufzeitsystem erscheint eine verwertbare Information.
- Falls das Programm von einem anderen Programm (z.B. einer Kommando-Prozedur) 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()** erkennt die spezielle Bedeutung dieses Rückgabewerts, so dass z.B. unsinnige Fakultätsberechnungen für negative Argumente vermieden werden. Diese traditionelle Fehlerbehandlung per Rückgabewert ist nicht grundsätzlich als überholt und ineffizient einzustufen, aber in vielen Situationen doch der gleich vorzustellenden Kommunikation über Ausnahmeobjekte unterlegen (siehe Abschnitt 10.3 zum Vergleich der beiden Kommunikationsverfahren).

Trotz seiner präventiven Bemühungen ist das Programm leicht aus dem Tritt zu bringen, indem man es mit einer nicht konvertierbaren Zeichnfolge füttert (z.B. „vier"). Die zunächst betroffene, FCL-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:



Weil kein Aufrufer eine geeignete Behandlungsroutine bereithält, endet das Programm mit einer Fehlermeldung:

```
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 System.Convert.ToInt32(String value)
   bei Fakul.Kon2Int(String s) in U:\Eigene Dateien\ ... \Fakul.cs:Zeile 7.
   bei Fakul.Main(String[] args) in U:\Eigene Dateien\ ... \Fakul.cs:Zeile 21.
```

10.2 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.

10.2.1 Die try-catch-finally - Anweisung

In C# wird die Behandlung von Ausnahmen über die **try-catch-finally** - Anweisung unterstützt:

```
try {
    Überwachter Block mit Anweisungen für den normalen Programmablauf
}
catch (Ausnahmeklasse Parametername) {
    Anweisungen für die Behandlung von Ausnahmen der ersten Ausnahmeklasse
}
// Optional können weitere Ausnahmen abgefangen werden:
catch (Ausnahmeklasse Parametername) {
    Anweisungen für die Behandlung von Ausnahmen der zweiten Ausnahmeklasse
}

    . . .

// Optionaler Block mit Abschluss- bzw. Bereinigungsarbeiten.
// Bei vorhandenem finally-Block, ist kein catch-Block erforderlich.
finally {
    Anweisungen, die unabhängig vom Auftreten einer Ausnahme ausgeführt werden
}
```

Die Anweisungen für den ungestörten Ablauf setzt man in den **try**-Block. 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.

Weil es der obigen Syntaxbeschreibung im Quellcodedesign trotz Unterstützung durch Kommentare an Präzision fehlt, sollen Sie in einer Übungsaufgabe ein Syntaxdiagramm erstellen (siehe Abschnitt 10.7).

10.2.1.1 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 Parameterliste eines **catch**-Blocks jedoch 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 die zugehörige Anweisung 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-catch-finally** - Anweisung wird maximal *ein* **catch**-Block ausgeführt. Weitere Details zum Programmablauf bei der Ausnahmebehandlung folgen in Abschnitt 10.2.2.

Nun zu den angekündigten Möglichkeiten, den Kopf eines **catch**-Blocks zu vereinfachen:

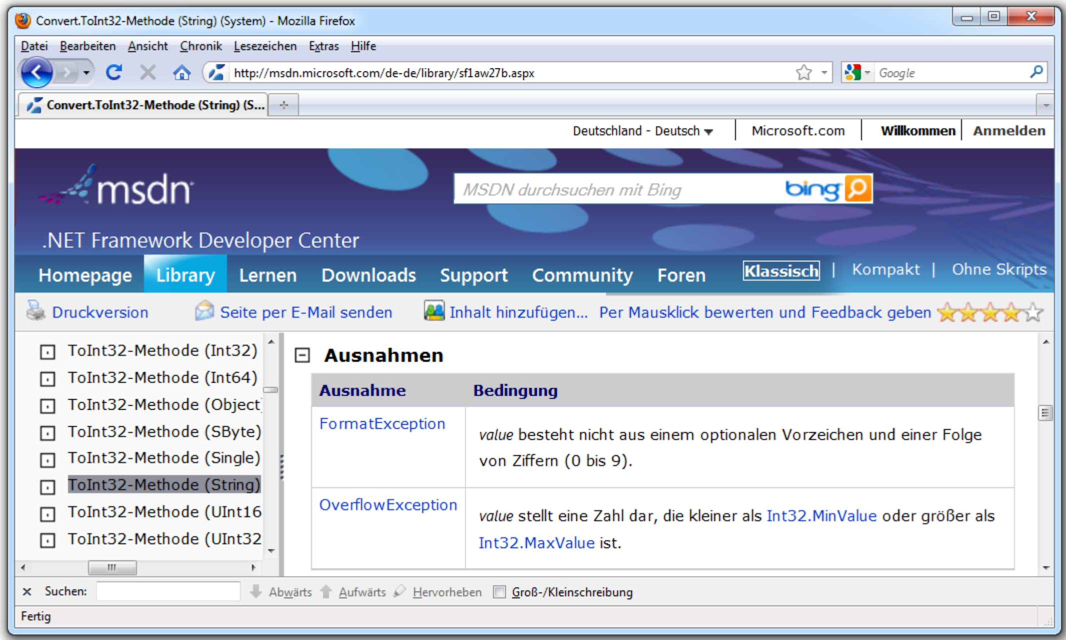
- Man kann auf die Angabe eines Formalparameter*namens* verzichten, hat dann aber im **catch**-Block kein Ausnahmeobjekt (mit Unfallbericht, siehe unten) zur Verfügung:

  ```
  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***
> **}**

Welche Ausnahmen von den Methoden eines FCL-Typs zu befürchten sind, erfährt man in der Dokumentation, z.B. bei der Methode **Convert.ToInt32(String)**:



Neben der bereits besprochenen **System.FormatException** ist bei **Convert.ToInt32()** auch eine Ausnahme aus der Klasse **System.OverflowException** möglich. Sie wird geworfen, wenn sich bei der Konvertierung eine ganze Zahl außerhalb des **int**-Wertebereichs ergibt (vgl. Abschnitt 3.6.1).

In der folgenden Variante der Methode `Kon2Int()` werden die von **Convert.ToInt32()** zu erwartenden Ausnahmen abgefangen. Die **OverflowException** wird lediglich entfernt, wobei `Kon2Int()` den Wert -1 zurückliefert (wie bei einem **int**-Wert außerhalb von [0, 170]). Im **FormatException** - Handler wird versucht, durch sukzessives Streichen des jeweils letzten Zeichens eine interpretierbare Teilzeichenfolge zu gewinnen. Bei Misserfolg landet wiederum der Wert -1 beim Aufrufer. Weil beim Reparaturversuch mit hoher Wahrscheinlichkeit mehrere Fehlversuche zu erwarten sind, ist die per Ausnahmeobjekt kommunizierende Methode **Convert.ToInt32()** aus Performanzgründen ungeeignet. Stattdessen wird die Methode **Int32.TryParse()** verwendet, die traditionell per **bool**-Rückgabewert über die Konvertierbarkeit berichtet und den resultierenden Wert per **out**-Parameter übergibt.

```
static int Kon2Int() {
   int arg = -1;
   try {
      arg = Convert.ToInt32(instr);
   } catch (FormatException) {
      String str = instr;
      bool ok = false;
      while (str.Length > 1 && !ok) {
         str = str.Substring(0, str.Length - 1);
         ok = Int32.TryParse(str, out arg);
      }
      if (ok)
         instr = str;
      else
         arg = -1;
   } catch (OverflowException) {}
```

```
    if (arg >= 0 && arg <= 170)
      return arg;
    else
      return -1;
  }
```

Man kann sich fragen, ob `Kon2Int()` nicht komplett auf den potentielle Ausnahmewerfer **Convert.ToInt32**() und damit auch auf die Ausnahmebehandlung per **try-catch** - Anweisung verzichten und ausschließlich die mit Rückgabewert arbeitende Methode **Int32.TryParse**() verwenden sollte. Nach den Performanzüberlegungen in Abschnitt 10.3 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.

Der **catch**-Block beschränkt sich nicht auf eine pure Ausgabe zum Existenznachweis, sondern unternimmt einen ernsthaften Reparaturversuch. Je nach Algorithmus kommen als Aufgaben für einen **catch**-Block auch in Frage:

- **Roll Back**
  Man kann versuchen, bereits realisierte und aufgrund der Ausnahme nunmehr unerwünschte Effekte des unterbrochenen **try**-Blocks wieder rückgängig zu machen.
- **Informationsvermittlung**
  Viele **catch**-Blöcke betätigen sich als Informationsvermittler und werfen selbst eine Ausnahme, um dem Aufrufer einen leichter verständlichen Unfallbericht zu liefern (siehe Abschnitt 10.6).

Das Beispielprogramm, dessen **Main**()-Methode im Vergleich zur Version in Abschnitt 10.1 unverändert geblieben ist, endet aufgrund der Verbesserungen in `Kon2Int()` nun beim Auftreten einer **FormatException** (z.B. wegen des Befehlszeilenarguments „vier“) mit der Meldung:

```
Keine ganze Zahl im Intervall [0, 170]: vier
```

Die **catch**-Blöcke in `Kon2Int()` verwenden das von **Convert.ToInt32**() geworfene Ausnahmeobjekt *nicht* und verzichten daher auf einen Parameternamen.

10.2.1.2 finally

Der **finally**-Block wird in jedem Fall ausgeführt, also …

- nach der ungestörten Ausführung des **try**-Blocks
  Auch ein vorzeitiges Verlassen der Methode durch eine **return**-Anweisung im **try**-Block verhindert nicht die Ausführung des **finally**-Blocks.
- nach einer Ausnahmebehandlung in einem **catch**-Block
- nach dem Auftreten einer unbehandelten Ausnahme

Dies ist der ideale Ort für Anweisungen, die unter möglichst allen Umständen ausgeführt werden sollen, z.B. zur Freigabe von Ressourcen wie Datei - und Netzverbindungen. Wir verwenden (dem Kapitel 12 über Dateibearbeitung vorgreifend) zur **finally**-Demonstration eine statische Methode, die aus einer Textdatei pro Zeile eine **double**-Zahl zu lesen versucht, um den Mittelwert der vorhandenen Zahlen zu berechnen:

```
static void Mean(String dateiname) {
   StreamReader sr = null;
   FileStream fs = null;
   try {
      fs = new FileStream(dateiname, FileMode.Open);
   } catch {
      Console.WriteLine("Fehler beim Öffnen der Datei {0}", dateiname);
      throw;
   }
   try {
      String s;
      int n = 0;
      double summe = 0.0;
      sr = new StreamReader(fs);
      while ((s = sr.ReadLine()) != null) {
         summe += Convert.ToDouble(s);
         n++;
      }
      Console.WriteLine("Deskriptive Statistiken zur Datei {0}\n", dateiname);
      Console.WriteLine("Anzahl:\t" + n);
      Console.WriteLine("Summe:\t" + summe);
      Console.WriteLine("Mittel:\t" + summe / n);
   } catch {
      Console.WriteLine("Fehler beim Lesen der Datei {0}", dateiname);
      throw;
   } finally {
      sr.Close();
   }
}
```

Das Ergebnis eines erfolgreichen Aufrufs:

```
Deskriptive Statistiken zur Datei daten.txt

Anzahl: 18
Summe:  90
Mittel: 5
```

In der ersten **try**-Anweisung (ohne **finally**-Block) werden die vom **FileStream** - Konstruktor, der eine vorhandene Datei öffnen soll, zu erwartenden Ausnahmen behandelt:

- **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.

Der **catch**-Block schreibt eine Fehlermeldung und wirft per **throw** - Anweisung (siehe Abschnitt 10.5) dieselbe Ausnahme erneut, so dass die Methode `Mean()` beendet und der Aufrufer über das Scheitern informiert wird.

Um den momentan interessanten Fall einer Störung *nach* dem erfolgreichen Öffnen der Datei geht es erst in der zweiten **try**-Anweisung. Eine geöffnete Datei sollte möglichst früh per **Close()**-Aufruf geschlossen werden, um andere Programme möglichst wenig zu behindern. Das muss auch für den Ausnahmefall sicher gestellt werden, indem das Schließen in einem **finally**-Block stattfindet. Im Beispiel kommt es zu einer Ausnahme bei geöffneter Datei, wenn die Methode **Convert.ToDouble()** auf eine nicht konvertierbare Zeichenfolge trifft (siehe Abschnitt 10.1). Weil der **Close()**-Aufruf im **finally**-Block steht, wird er auf jeden Fall ausgeführt. Stünde er z.B. am Ende des **try**-Blocks, bliebe im eben geschilderten Ausnahmefall die Datei geöffnet bis zum Programmende.

10.2.2 Programmablauf bei der Ausnahmebehandlung

Findet das Laufzeitsystem für eine Ausnahme in der aktuellen 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.

10.2.2.1 Beispiel

In folgendem Beispiel dürfen Sie allerdings *keine* optimierte Einsatzplanung erwarten. Es soll einige Programmabläufe demonstrieren, die sich bei Ausnahmen auf verschiedenen Stufen einer Aufrufhierarchie ergeben können. 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:

```
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 Aufruf in eine **try**-Anweisung mit **catch**-Block für die **ArithmeticException** ein, die z.B. bei einer Division durch Null auftritt. `Calc()` benutzt die Klassenmethode **Convert.ToInt32()** sowie den Modulo-Operator in einem **try**-Block, wobei nur die (potentiell von **Convert.ToInt32()** zu erwartende) **FormatException** 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 beim Ausführen der Anweisung

```
erg = 10 % Convert.ToInt32(instr);
```

eine **FormatException** an `Calc()` gemeldet (z.B. wegen Befehlszeilenargument „acht“ von **Convert.ToInt32()** geworfen), dann kommt der zugehörige **catch**-Block zum Einsatz. Dann folgen:

- **finally**-Block in `Calc()`
- restliche Anweisungen in `Calc()`

An **Main()** wird keine Ausnahme gemeldet, also werden 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 wenig überzeugenden Ausgabe

```
10 % acht = 0
```

kommt es, weil die **FormatException** in `Calc()` nicht *sinnvoll* behandelt wird. Das aktuelle Beispiel soll ausschließlich 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 Handler. Bevor die Methode verlassen wird, um entlang der Aufrufsequenz nach einem geeigneten Handler zu suchen, wird noch ihr **finally**-Block ausgeführt. Im 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 gar 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:

```
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()
```

10.2.2.2 Komplexe Fälle

Es ist oft erforderlich, **try**-Anweisungen zu schachteln, wobei sowohl innerhalb eines **try**- als auch innerhalb eines **catch**-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.

10.3 Ausnahmeobjekte im Vergleich zur traditionellen Fehlerbehandlung

Die konventionelle Fehlerbehandlung verwendet meist die Rückgabewerte von Methoden zur Berichterstattung über Probleme bei der Ausführung von Aufträgen. Ein Rückgabewert kann …

- ausschließlich zur Fehlermeldung dienen
  Meist wird dann ein ganzzahliger **Returncode** mit Datentyp **int** verwendet, wobei die Null einen erfolgreichen Ablauf meldet, während andere Zahlen 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()` im Beispielprogramm)

Sollen z.B. drei Methoden, deren Rückgabewerte ausschließlich zur Fehlermeldung dienen, nacheinander aufgerufen werden, dann wird die vom Algorithmus diktierte simple Sequenz:

```
M1();
M2();
M3();
```

nach Ergänzen der Fehlerbehandlung zu einer länglichen und recht unübersichtlichen Konstruktion (nach Mössenböck 2005, S. 254):

```
returncode = M1();
if (returncode == 0) {
   returncode = M2();
   if (returncode == 0) {
      returncode = M3();
      // Behandlung für diverse M3()-Fehler
      if (returncode == 1) {
         . . .
      }
         . . .
   }
   else {
      // Behandlung für diverse M2()-Fehler}
}
else {
   // Behandlung für diverse M1()-Fehler
}
```

Mit Hilfe der Ausnahmetechnik bleibt beim Kernalgorithmus die Übersichtlichkeit erhalten. Wir nehmen nun an, dass die drei Methoden `M1()`, `M2()` und `M3()` durch Ausnahmeobjekte über Fehler informieren:

```
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
}
```

Ein gut gesetzter Rückgabewert nutzt natürlich nichts, wenn sich der Aufrufer nicht darum kümmert.

Neben dem unübersichtlichen Quellcode und der ungesicherten Beachtung eines Rückgabewerts ist am klassischen Verfahren zu bemängeln, dass eine Fehlerinformation aufwändig entlang der Aufrufersequenz nach oben gemeldet werden muss, wenn sie nicht an Ort und Stelle behandelt werden soll.

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 diesem Fall verwendet die klassische Fehlerbehandlung eine **Statusvariable** als Kommunikationsmittel, wobei die Beachtung ebenso wenig garantiert ist wie bei einem Returncode.

Gegenüber der konventionellen Fehlerbehandlung hat die Kommunikation über Ausnahmeobjekte u.a. folgende Vorteile:

- Bessere Lesbarkeit des Quellcodes
  Mit Hilfe einer **try-catch-finally** - Konstruktion erreicht man eine Trennung zwischen den Anweisungen für den normalen Programmablauf und den diversen Ausnahmebehandlungen, so dass der Quellcode übersichtlich bleibt.
- Automatische Weitermeldung bis zur bestgerüsteten Methode
  Oft ist der unmittelbare Aufrufer nicht gut gerüstet zur Behandlung einer Ausnahme, z.B. nach dem vergeblichen Öffnen einer Datei. Dann soll eine „höhere“ Methode über das weitere Vorgehen entscheiden.

- Bessere Fehlerinformationen für den Aufrufer
  Über ein **Exception**-Objekt kann der Aufrufer beliebig genau über einen aufgetretenen Fehler informiert werden, was bei einem klassischen Rückgabewert nicht der Fall ist.
- Garantierte Beachtung von Ausnahmen
  Im Unterschied zu Returncodes oder Fehlerstatusvariablen können Ausnahmen nicht ignoriert werden. Reagiert ein Programm nicht darauf, wird es vom Laufzeitsystem beendet.

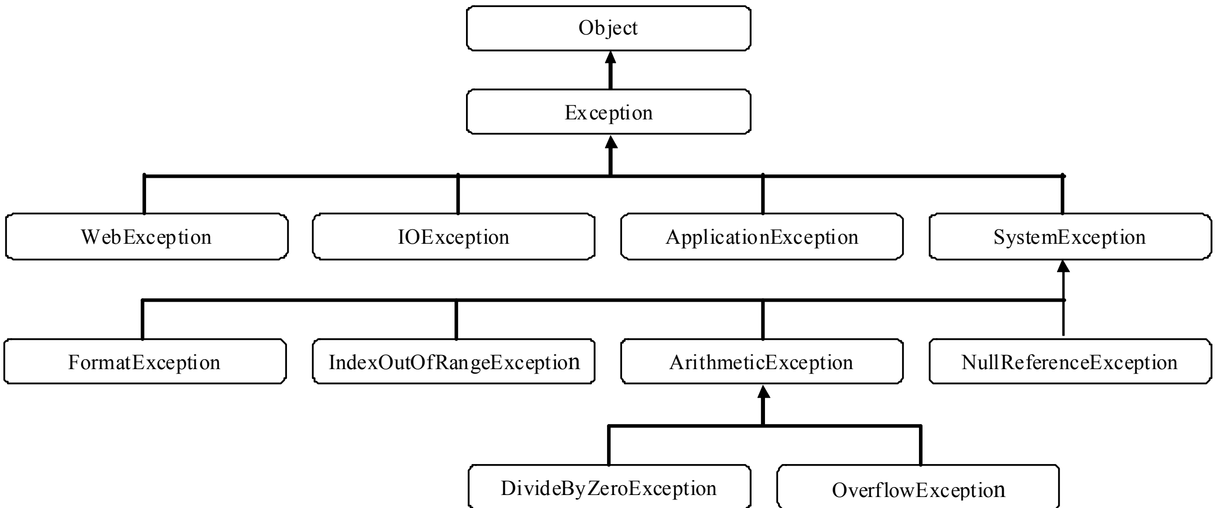
Wie die Realisation des **catch**-Blocks zur **FormatException** in Abschnitt 10.2.1 gezeigt hat, ist die Fehlermeldung per Ausnahmeobjekt dem klassischen Rückgabewert nicht grundsätzlich überlegen. Wenn ein Problem mit erheblicher Wahrscheinlichkeit auftritt, also keinesfalls ungewöhnlich ist, …

- sollte eine routinemäßige, aktive Kontrolle stattfinden
- eine auf das Problem stoßende Methode per Rückgabewert kommunizieren, also davon ausgehen, dass der Aufrufer mit dem Problem rechnet und daher den Rückgabewert beachtet.

Bei Fehlern mit geringer Wahrscheinlichkeit haben jedoch häufige, meist überflüssige Kontrollen Performanzeinbußen und einen unübersichtlichen Quellcode zur Folge. Hier sollte man es besser auf eine Ausnahme ankommen lassen. Eine Überwachung per Ausnahmetechnik verursacht praktisch nur dann Kosten, wenn tatsächlich eine Ausnahme geworfen wird. Diese Kosten sind allerdings deutlich größer als bei einer Fehleridentifikation auf traditionelle Art.

10.4 Ausnahme-Klassen im .NET - Framework

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* Ausnahmen durch Wahl einer entsprechend breiten Basisklasse abgefangen 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 Methodenaufruf
  ```
  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. Am Ende von Abschnitt 10.1 war schon die **StackTrace**-Eigenschaft der **FormatException** zu sehen, die in der ersten Variante des Fakultätsprogramms bei einer irregulären Zeichenfolge von der Methode **Number.StringToNumber()** geworfen wird:

```
bei System.Number.StringToNumber(String str, ..., Boolean parseDecimal)
bei System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
bei System.Convert.ToInt32(String value)
bei Fakul.Kon2Int(String instr) in U:\Eigene Dateien\ ... \Fakul.cs:Zeile 7.
bei Fakul.Main(String[] args) in U:\Eigene Dateien\ ... \Fakul.cs:Zeile 21.
```

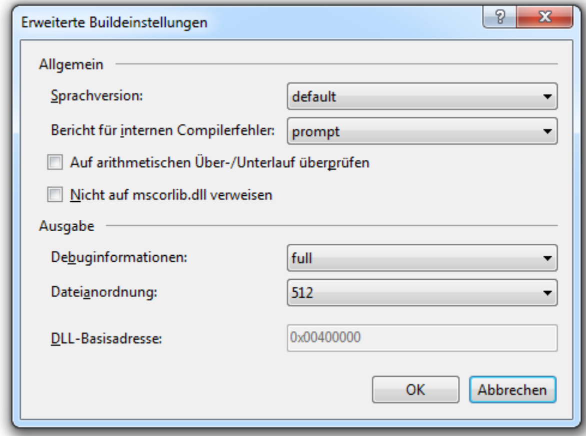
  Dateinamen und Zeilennummern enthält die Aufrufreihenfolge nur dann, wenn das betroffene Assembly mit aktiver Debug-Option übersetzt worden ist. Bei direkten Aufruf des Compilers in einem Konsolenfenster ist die Option **debug** anzugeben, z.B.:

```
csc /debug Sequenzen.cs
```

  Im Visual Studio 2010 (Express Edition) stellt man den Umfang der **Debuginformationen** über

  Projekt > Einstellungen > Erstellen > Erweitert

  ein, wobei die folgende Voreinstellung gilt:



  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 10.6). Um dem Aufrufer auch die ursprüngliche Ausnahme zur Verfügung zu stellen, kann man ihre Adresse in der Eigenschaft **InnerException** mitliefern.

- **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 Typ **DictionaryEntry**) unterbringen. Für die Schlüssel wählt man in der Regel den Datentyp **String**, für die Werte einen geeigneten Typ. Im folgenden Beispiel werden drei Einträge in die **Data**-Liste eines neuen Ausnahmeobjekts aufgenommen:

```
if (arg < 0 || arg > 170) {
   BadFakulArgException bfa =
      new BadFakulArgException("Wert ausselhalb [0, 170]");
   bfa.Data.Add("Input", instr);
   bfa.Data.Add("Type", 3);
   bfa.Data.Add("Value", arg);
   throw bfa;
}
```

Die **ToString()**-Methode eines **Exception**-Objekts liefert:

- den Namen der Ausnahmeklasse
- **Message**-Zeichenfolge (die beim Erzeugen der Ausnahme formulierte Fehlermeldung)
- **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.
```

10.5 Ausnahmen werfen (throw)

Unsere eigenen Methoden 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. In folgender Variante der Methode `Kon2Int()` aus dem Standard-beispiel von Kapitel 10 wird ein Ausnahmeobjekt aus der Klasse **ArgumentOutOfRange-Exception** erzeugt und geworfen, wenn die erfolgreiche Interpretation des Parameters `instr` ein unzulässiges Fakultätsargument ergibt:

```
static int Kon2Int() {
   int arg;
   arg = Convert.ToInt32(instr);
   if (arg < 0  ||  arg > 170)
      throw new ArgumentOutOfRangeException("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. Wie im Beispiel benutzt man oft den **new**-Operator mit nachfolgendem Konstruktor, um vor Ort das Ausnahmeobjekt zu erzeugen und die Referenz zu liefern.

Im Beispiel wird ein **ArgumentOutOfRangeException**-Konstruktor mit *drei* Parametern verwen-det, wobei Name und Wert des irregulären Arguments sowie eine Fehlermeldung anzugeben sind. Aus dem Aufruf

```
try {
   argument = Kon2Int();
} 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. In diesem Fall wird die gerade behandelte Ausnahme erneut geworfen. Dieses Verhalten kommt z.B. in Frage, wenn eine Methode auf eine Ausnahme reagieren und einen Lösungsversuch unternehmen möchte, aber das Problem nicht aus Welt schaffen kann und daher ihren Aufrufer informieren muss.

Statt die ursprüngliche Ausnahme in einem **catch**-Block erneut zu werfen, kommt auch die Verwendung einer anderen Ausnahmeklasse in Frage, die aufgrund der bisherigern Analyse besser geeignet erscheint. Damit die ursprüngliche Ausnahme als Anlage beigefügt werden kann, bieten die FCL-Ausnahmeklassen einen Konstruktor mit einem Parameter vom Typ **Exception**. Ein Handler kann ggf. über die Eigenschaft **InnerException** auf die Anlage zugreifen.

In der obigen `Kon2Int()`–Variante wird auf die Behandlung der von **Convert.ToInt32** potentiell zu erwartenden Ausnahmen verzichtet. Folglich muss ein `Kon2Int()`–Anwender mit folgenden Ausnahmeklassen rechnen:

- **FormatException**
- **OverflowException**
- **ArgumentOutOfRangeException**

In der `Kon2Int()`–Dokumentation muss darüber informiert werden.

Dass eine Methode die *selbst* geworfenen Ausnahmen auch wieder auffängt, ist nicht unbedingt der Standardfall, aber in manchen Situationen eine praktische Möglichkeit, von verschiedenen potentiellen Schadstellen aus zur selben Ausnahmebehandlung zu verzweigen, z.B.:

Quellcode:

```
using System;
class Prog {
   static void Nix(bool cond) {
      try {
         if (cond)
            throw new Exception("A");
         else
            throw new Exception("B");
      }
      catch (Exception e) {
         Console.WriteLine(e.Message);
      }
   }
   static void Main() {
      Nix(true);
   }
}
```

Ausgabe:

```
A
```

10.6 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 keinesfalls auf die im .NET – Framework vordefinierten Ausnahmeklassen beschränken, sondern kann eigene Ausnahmen definieren, z.B.:

```
using System;

public class BadFakulArgException : Exception {
   int type, value = -1;
   String input;

   public BadFakulArgException() {
   }
   public BadFakulArgException(String message) : base(message) {
   }
   public BadFakulArgException(String message, Exception innerException)
      : base(message, innerException) {
   }
   public BadFakulArgException(String message, String input_, int type_, int value_)
      : this(message, input_, type_, value_, null) {
   }
   public BadFakulArgException(String message, String input_,
                     int type_, int value_, Exception innerException)
        : this(message, innerException) {
      input = input_;
      if (type_ >= 0 && type <= 2)
         type = type_;
      if (type_ == 3 && (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 `BadFakulArgException` an Microsofts Empfehlungen für selbst definierte Ausnahmeklasse (siehe z.B. http://msdn.microsoft.com/de-de/library/87cdya3t.aspx):

- Als Basisklasse sollte **System.Exception** verwendet werden.
- Der Klassenname sollte mit dem Wort *Exception* enden.
- Die folgenden *allgemeinen Konstruktoren* sollten mit **public** - Verfügbarkeit implementiert werden:
  - Ein parameterfreier Konstruktor
  - Ein Konstruktor mit einem **String**-Parameter für die Fehlermeldung
  - Einen Konstruktor mit einem **String**- und einem **Exception**-Parameter für eine Fehlermeldung und den Verweis auf ein inneres Ausnahmeobjekt, das zuvor aufgefangen wurde und nun in ein informativeres Ausnahmeobjekt als Anlage aufgenommen wird.

Beim parameterlosen `BadFakulArgException`-Konstruktor beschränken wir uns auf den (impliziten) Aufruf des parameterlosen Basisklassenkonstruktors. Bei der restlichen Konstruktoren rufen wir explizit eine passende Überladung des Basisklassenkonstruktors auf, um die Instanzvariablen hinter den Eigenschaften **Message** und **InnerException** initialisieren zu können.

Über ein Objekt der handgestrickten Ausnahmeklasse `BadFakulArgException` kann ausführlich über Probleme mit Argumenten 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: Zeichenfolge kann nicht konvertiert werden
  - 2: **int**-Überlauf
  - 3: **int**-Wert außerhalb [0, 170]
- in der Eigenschaft `Value` steht das Konvertierungsergebnis (falls vorhanden, sonst -1)

Die endgültige `Kon2Int()`-Version kümmert sich um alle von **Convert.ToInt32()** zu befürchtenden Ausnahmen und wirft bei allen Fehlerursachen eine spezielle `BadFakulArgException`:

```
static int Kon2Int() {
   int arg = -1;
   try {
      arg = Convert.ToInt32(instr);
      if (arg < 0  || arg > 170)
         throw new BadFakulArgException("Wert ausselhalb [0, 170]", instr, 3, arg);
      else
         return arg;
   }
   catch (FormatException e) {
      String str = instr;
      bool ok = false;
      while (str.Length > 1 && !ok) {
         str = str.Substring(0, str.Length - 1);
         ok = Int32.TryParse(str, out arg);
      }
      if (ok) {
         instr = str;
         return arg;
      } else
         throw new BadFakulArgException("Fehler beim Konvertieren", instr, 1, -1, e);
   } catch (OverflowException e) {
      throw new BadFakulArgException("Integer-Überlauf", instr, 2, -1, e);
   }
}
```

Den in **catch**-Blöcken geworfenen `BadFakulArgException`-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");
      Environment.Exit(1);
   } else
      instr = args[0];

   try {
      argument = Kon2Int();
   }
   catch (BadFakulArgException e) {
      Console.WriteLine("Message:\t" + e.Message);
      Console.WriteLine("Fehlertyp:\t{0}\nZeichenfolge:\t{1} ", e.Type, e.Input);
      Console.WriteLine("Wert:    \t" + e.Value);
      if (e.InnerException != null)
         Console.WriteLine("Orig. Message:\t" + e.InnerException.Message);
      Environment.Exit(1);
   }
```

```
      double fakul = 1.0;
      for (int i = 1; i <= argument; i++)
         fakul = fakul * i;
      Console.WriteLine("Fakultät von {0}: {1}", instr, fakul);
   }
```

Bei einem Programmstart mit dem Kommandozeilenargument „vier“ resultiert z.B. die Ausgabe:

```
Message:        Fehler beim Konvertieren
Fehlertyp:      1
Zeichenfolge:   vier
Wert:           -1
Orig. Message:  Die Eingabezeichenfolge hat das falsche Format.
```

Zum Transport von speziellen 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 **Dictionary-Entry**) unterbringen (siehe Abschnitt 10.4). Bei der Klasse `BadFakulArgException` könnten wir uns auf die folgende Standarddefinition beschränken:

```
public sealed class BadFakulArgException : Exception {
   public BadFakulArgException() {
   }
   public BadFakulArgException(String message) : base(message) {
   }
   public BadFakulArgException(String message, Exception innerException)
      : base(message, innerException) {
   }
}
```

In die **Data**-Liste eines `BadFakulArgException`-Objekts lassen sich Schlüssel-Wert - Einträge mit den benötigten Zusatzinformationen z.B. per **Add()** aufnehmen:

```
BadFakulArgException bfa =
   new BadFakulArgException("Fehler beim Konvertieren", e);
bfa.Data.Add("Input", instr);
bfa.Data.Add("Type", 1);
bfa.Data.Add("Value", -1);
```

Ein **catch**-Block kann per Indexer

```
Console.WriteLine("Wert = {0}", e.Data["Value"]);
```

oder mit Hilfe der **DictionaryEntry**-Eigenschaften **Key** und **Value**

```
foreach (DictionaryEntry de in e.Data) {
   Console.WriteLine("{0}\t:\t{1}", de.Key, de.Value);
```

auf die **Data**-Listeneinträge eines Ausnahmeobjekts zugreifen.

10.7 Übungsaufgaben zu Kapitel 10

1) Erstellen Sie ein Syntaxdiagramm zur **try-catch-finally** - Anweisung (vgl. Abschnitt 10.2.1).

2) Im Beispielprogramm zur Demonstration von möglichen Sequenzen bei der Ausnahmebehandlung (siehe Abschnitt 10.2.2) verzichtet die Methode `Calc()` darauf, die potentiell von der Methode **Convert.ToInt32**() zu erwartende **OverflowException** abzufangen (vgl. Abschnitt 10.2.1). 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 weiter gerechnet wird, und erst am Ende eines längeren Rechenweges das Ergeb-

nis **NaN** auftaucht (in der Ausgabe: **n. def.**). In folgendem Beispiel wird eine Methode namens `Log2()` zur Berechnung des dualen Logarithmus[1] verwendet, welche auf die FCL-Methode **Math.Log()** zurückgreift und daher bei ungeeigneten Argumenten (≤ 0) als Rückgabewert **Double.NaN** liefert.

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
using System;

class DuaLog {
   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);
   }
}
```

</td><td>

```
n. def.
```

</td></tr>
</table>

Erstellen Sie eine Version, die bei ungeeigneten Argumenten eine **ArgumentOutOfRange-Exception** wirft.

4) Erstellen Sie eine Variante zu der in Abschnitt 7.2.1 vorgestellten generischen Stapelverwaltungsklasse mit zusätzlichen Methoden `AuflegenEx()` und `AbhebenEx()`, die bei besetztem bzw. leerem Stapel eine **InvalidOperationException**-Ausnahme werfen. Die bereits vorhandenen Methoden `Auflegen()` und `Abheben()` signalisieren solche Probleme mit dem Rückgabewert **false**. Im vorliegenden Fall sind beide Kommunikationstechniken akzeptabel (vgl. Abschnitt 10.3), und bei der erweiterten Klasse hat der Anwender die Wahl.

---

[1] Für eine positive Zahl a ist ihr Logarithmus zur Basis b (> 0) definiert durch:

$$\log_b(a) := \frac{\ln(a)}{\ln(b)}$$

Dabei steht ln() für den natürlichen Logarithmus zur Basis e (Eulersche Zahl).

11 Attribute

An Typen (Klassen, Strukturen, Schnittstellen, usw.), Member (Methoden, Eigenschaften, usw.), Parameter und Rückgabewerte von Methoden sowie an Assemblies und Module kann man *Attribute* anheften, um zusätzliche Metainformationen bereit zu stellen, 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 legt die Attributobjekte per Serialisierung (siehe Abschnitt 12.2.3) in der Metadatentabelle des erzeugten Assemblies ab. 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 für Interessenten verfügbar sind.

Ein Attribut beeinflusst das Laufzeitverhalten eines Programms über seine Signalwirkung auf Methoden, welche sich über die Existenz bzw. Ausgestaltung des Attributs informieren und ihr Verhalten daran orientieren. Wir lernen also eine weitere Technik zur Kommunikation zwischen Programmbestandteilen kennen. In komplexen objektorientierten Softwaresystemen (Frameworks) spielt generell die als *Reflexion* (engl.: *reflection*) bezeichnete Ermittlung von Informationen über Typen und Instanzen zur Laufzeit eine zunehmende Rolle. Dabei leisten Attribute einen wichtigen Beitrag.

Man kann die Attribute auch als Option zur *deklarativen Programmierung* auffassen. Sie ergänzen die im C# - Sprachumfang verankerten *Modifikatoren* für Typen, Methoden etc. und bieten dabei eine enorme Flexibilität.

Von *deklarativer Programmierung* sprachen wir auch bei der Gestaltung einer WPF-Bedienoberfläche per XAML-Code (siehe Abschnitt 4.10.2). Die aktuell behandelten, im 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.

In der FCL wird von Attributen reichlich Gebrauch gemacht, was z.B. die Dokumentation zur Klasse **String** zeigt:

```
[SerializableAttribute]
[ComVisibleAttribute(true)]
public sealed class String : IComparable,
    ICloneable, IConvertible, IComparable<string>, IEnumerable<char>,
    IEnumerable, IEquatable<string>
```

Hier wird über die Klasse **String** ausgesagt, dass ihre Objekte serialisiert werden dürfen (siehe Abschnitt 12.2.3), und dass die Klasse für die Zusammenarbeit mit traditionellen Windows-Komponenten (nach dem *Component Object Model*) geeignet ist. Was das konkret bedeutet, wird sich bei der noch ausstehenden Beschäftigung mit Objektserialisierung bzw. COM-Interoperabilität zeigen.

Wir müssen uns nicht auf die Auswertung und Vergabe von FCL-Attributen beschränken, sondern können auch eigene Attribute definieren und verwenden.

Gelegentlich wird im Zusammenhang mit unserem aktuellen Thema von *benutzerdefinierten Attributen* gesprochen (siehe z.B. Richter 2006, S. 403ff), und wichtige Methoden zur Auswertung von Attributen (siehe Abschnitt 11.2) führen das Wort *Custom* in ihrem Namen. Es ist nicht ganz klar (uns auch nicht sonderlich wichtig), ob durch diese Wortwahl die vom Anwendungsprogrammierer erstellten Attributklassen den bereits in der FCL definierten Attributklassen gegenübergestellt werden sollen, oder ob sich der Gegenbegriff auf Metadaten bezieht, die vom .NET - Framework verwaltet werden und die in keinem Zusammenhang mit der Klasse **System.Attribute** stehen. Das Manuskript orientiert sich bei der Begriffsverwendung an der C# 4.0 - Sprachspezifikation (Micro-

soft 2010a, Kap. 17). Dort ist nur von *Attributen* die Rede, und dabei sind die von **System.Attribute** abstammenden Klassen gemeint, von wem auch immer programmiert.

11.1 Attribute vergeben

Sollen z.B. die Benutzer einer Klassenbibliothek dazu gebracht werden, einer neuen Klasse den Vorzug vor einer alten zu geben, kann man der alten Klassendefinition zwischen eckigen Klammern das Attribut **Obsolete** voranstellen, so dass der Compiler bei Verwendung dieser Klasse automatisch eine Warnung ausgibt. Dies geschieht z.B. beim Übersetzen der folgenden Quelle:

ObsoleteClass.cs

Prog | Main()

```
using System;
[Obsolete]
class MyClass {
    public static void Tell() {
        Console.WriteLine("Hallo!");
    }
}
class MyNewClass {
    public static void Tell() {
        Console.WriteLine("Hallo, Wilhelm!");
    }
}
class Prog {
    static void Main() {
        MyClass.Tell();
    }
}
```

"MyClass" ist veraltet.

100 %

Im Editor der Visual C# 2010 Express Edition wird der unerwünschte Zugriff auf die obsolete Klasse unterschlängelt, und der Compiler moniert:

```
ObsoleteClass.cs(15,9): warning CS0612: "MyClass" ist veraltet.
```

Wer als Klassenbibliotheksdesigner und -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 MyNewClass")]
```

Dann meldet der Compiler beim unerwünschten Zugriff:

```
ObsoleteClass.cs(15,9): warning CS0618: "MyClass" ist veraltet: "Benutzen Sie
        bitte MyNewClass"
```

Bei der Vergabe eines Attributes wird durch einen Konstruktor der Attributklasse ein Objekt erstellt, das die Metadaten der Trägers ergänzt.

Je nach verwendeter Konstruktorüberladung sind Aktualparameter anzugeben, wobei eine leere Aktualparameterliste allerdings weggelassen werden kann. Als weitere syntaktische Besonderheit darf bei der Vergabe eines Attributs der Namensteil *Attribute* im Klassennamen weglassen werden. Im Beispiel kommt also die Klasse **ObsoleteAttribute** (aus dem Namensraum **System**) zum Einsatz.

Neben Klassen können auch andere Programmbestandteile mit Attributen versehen werden, z.B. Methoden:

ObsoleteMethod.cs

Prog | Main()

```
using System;

class MyClass {
    [Obsolete("Der Support für Tell() läuft aus. Benutzen Sie bitte TellEx()")]
    public static void Tell() {
        Console.WriteLine("Hallo!");
    }

    public static void TellEx() {
        Console.WriteLine("Hallo, Wilhelm!");
    }
}

class Prog {
    static void Main() {
        MyClass.Tell();
    }
}
```

[veraltet] void MyClass.Tell()

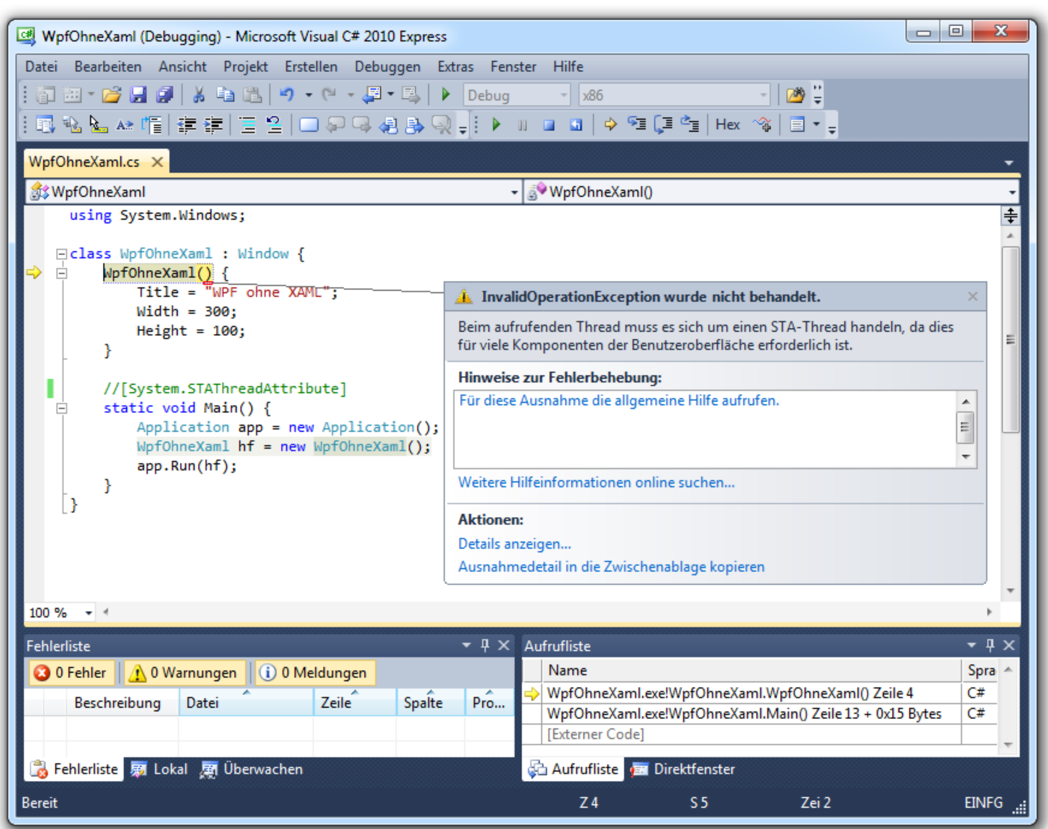
Warnung:
"MyClass.Tell()" ist veraltet: "Der Support für Tell() läuft aus. Benutzen Sie bitte TellEx()"

100 %

Beim Übersetzen dieses Quellecodes meldet der Compiler:

```
ObsoleteMethod.cs(16,3): warning CS0618: "MyClass.Tell()" ist veraltet: "Der
          Support für Tell() läuft aus. Benutzen Sie bitte TellEx()"
```

Bei einer WPF-Anwendung muss die **Main**()-Methode generell mit dem **STAThreadAttribut** dekoriert werden, weil ansonsten schon der Fensterklassenkonstruktor an einer **InvalidOperationException** scheitert, z.B.:



Wurde die Anwendung aus dem Visual Studio (z.B. mit **F5**) im Debug-Modus gestartet, liefert die Entwicklungsumgebung ...

- den Typ der Ausnahme
- die Unfallstelle
- Tipps zur Lösung des Problems

Um am Projekt weiterarbeiten zu können, muss der Debug-Modus beendet werden (z.B. mit **Umschalt+F5**).

Mit dem Attribut **STAThreadAttribut** wird signalisiert, dass bei der Kooperation mit dem Component Object Model (COM), der noch sehr weit verbreiteten Windows-Komponententechnologie, das COM-Threading-Modell STA (*Singlethread-Apartment*) zum Einsatz kommen soll, um Synchronisationsprobleme bei Steuerelementen zu verhindern.

Im weiteren Verlauf von Kapitel 11 wird noch klarer, dass man bei der Vergabe von Attributen in der Regel nicht nur den Quellcode kommentiert, sondern signalisierend den Programmablauf beeinflusst, sofern andere Akteure (z.B. andere Typen oder die CLR) die Attribute kennen und bei ihrem Verhalten berücksichtigen.

11.2 Attribute per Reflexion auswerten

Das .NET - Framework 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, z.B. bei einer Klasse oder einer Methode:

Quellcode:

```
using System;
using System.Reflection;

[Obsolete("Benutzen Sie bitte MyNewClass")]
class MyClass {
 [Obsolete("Der Support für Tell() läuft aus. Benutzen Sie bitte TellEx()")]
 public static void Tell() {
  Console.WriteLine("Hallo!");
 }
}

class Prog {
 static void Main() {
  Type typeMC = typeof(MyClass);
  Type typeObs = typeof(ObsoleteAttribute);
  if (Attribute.IsDefined(typeMC, typeObs))
   Console.WriteLine("Der Typ {0} ist obsolet", typeMC.Name);

  MemberInfo[] mi = typeMC.FindMembers(MemberTypes.Method,
   BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public,
   Type.FilterName, "Tell");
  if (Attribute.IsDefined(mi[0], typeObs))
   Console.WriteLine("\nDie Methode {0} ist obsolet", mi[0].Name);
 }
}
```

Ausgabe:

```
Der Typ MyClass
 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 Attribut-Träger:
  `Attribute.IsDefined(typeMC, typeObs)`
- Im zweiten Aufruf wird die Anwesenheit eines Methodenattributs untersucht:
  `Attribute.IsDefined(mi[0], typeObs)`

Als zweiter Parameter ist das **Type**-Objekt zur fraglichen Attributklasse anzugeben. Im Beispiel werden die beiden beteiligten **Type**-Objekte per **typeof**-Operator erzeugt. Anders als bei der Attributvergabe (siehe Abschnitt 11.1) ist auch der Name der Attributsklasse vollständig (inkl. Namenbestandteil *Attribute*) zu schreiben.

Das im zweiten **IsDefined()**-Aufruf benötigte **MemberInfo**-Objekt zur `MyClass`-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 öffentliche Instanz- und Klassen-Member zugelassen.
- Der dritte Parameter sorgt im Beispiel für eine namensorientierte Filterung.
- Im letzten Parameter spezifiziert man den geforderten Namen, wobei auch das Jokerzeichen * genutzt werden kann.

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 (per Deserialisierung) aus den Assembly-Metadaten, so dass 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 MyClass {
  [Obsolete("Der Support für Tell() läuft aus. 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 MyClass:

Die Methode Tell() ist obsolet
Message: Der Support für Tell() läuft aus. 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 benutzerdefinierten Attribute ermittelt, wobei *geerbte* Attribute nicht interessieren (Wert **false** für den Parameter **inherit**):

Quellcode	Ausgabe

<pre>using System;
 	
[Obsolete] [Serializable]
class MyClass {
  public static void Tell() {
   Console.WriteLine("Hallo!");
  }
}
 	
class Prog {
  static void Main() {
   Type type = typeof(MyClass);
   Attribute[] atar =
     Attribute.GetCustomAttributes(type, false);
   foreach (Attribute at in atar)
    Console.WriteLine(at);
  }
}</pre>
	<pre>System.ObsoleteAttribute
	System.SerializableAttribute</pre>

Mit alternativen Überladungen lassen sich Assemblies, Member, Parameter etc. analog untersuchen.

11.3 Attribute definieren

Bei einer eigenen Attributklasse sollte man …

- die Basisklasse **System.Attribute** verwenden,
- den Klassennamen mit dem Wort *Attribute* enden lassen.

Um den Compiler darüber zu informieren, welchen Programmbestandteilen das Attribut angeheftet werden darf, verwendet man ein Attribut aus der Klasse **AttributeUsageAttribute**. Im folgenden Beispiel erhält der Konstruktorparameter **validOn** den Wert **AttributeTargets.Class**, so dass 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 {
   int level;
   public NonsenseAttribute(int level_) {
      level = level_;
   }
   public int Level {
      get {return level;}
   }
}
```

Mit dieser Eigenschafts-Initialisierung innerhalb der Konstruktor-Parameterliste kommt eine spezielle Syntax unter Verwendung von Name-Wert - Paaren zum Einsatz. Man spricht hier von **Namensparametern**, die *nach* den regulären Parametern (nun **Positionsparameter** genannt) in beliebiger Reihenfolge stehen dürfen. 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* besteht übrigens seit C# 3.0 auch für andere Klassen eine analoge Initialisierungsmöglichkeit (siehe Abschnitt 4.4.3 und 21.1.2).

Bei Positions- oder Namensparameter von **Attribut**-Konstruktoren sind ausschließlich die folgenden Datentypen erlaubt:

- **bool**, **byte**, **char**, **short**, **int**, **long**, **float**, **double**
- **System.Object**, **System.String**, **System.Type**
- Aufzählungstypen
- Eindimensionale Arrays mit einem Elementtyp aus der obigen Liste

11.4 Attribute für Assemblies und Module

Auch Assemblies und Module 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.:

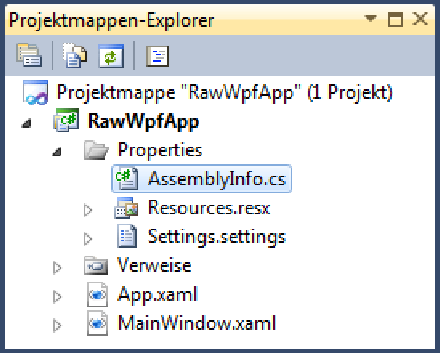
```
[assembly: AssemblyCompany("Marco Saft")]
[assembly: AssemblyProduct("YourTools")]
[assembly: AssemblyVersion("1.4.2.3")]
```

Hier wird jeweils das vom Compiler zu erzeugende Assembly als Träger des nachfolgenden Attributs festgelegt.

Derartige Assembly-Attribut - Deklarationen erzeugt das Visual Studio beim Erstellen eines neuen Projekts unter Verwendung der Vorlage **WPF-Anwendung** automatisch, wobei die Entwicklungsumgebung eine Datei namens **AssemblyInfo.cs** anlegt, z.B. (**using**-Direktiven und Kommentare aus Platzgründen weggelassen):

```
[assembly: AssemblyTitle("RawWpfApp")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("Universität Trier")]
[assembly: AssemblyProduct("RawWpfApp")]
[assembly: AssemblyCopyright("Copyright © Universität Trier 2011")]
[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")]
```

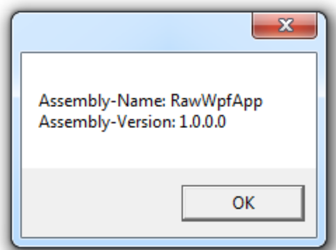
Zur Modifikation dieser Attribute öffnen man die Datei **AssemblyInfo.cs** über den Projektmappe-nexplorer:



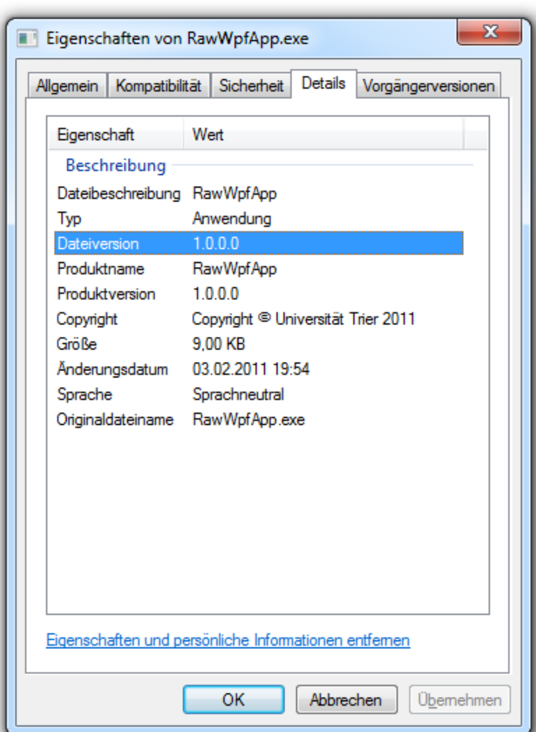
In einem Programm kann man auf einige Attribute des gerade ausgeführten Assemblies über Eigenschaften des zugehörigen **AssemblyName**–Objekts zugreifen, z.B.:

```
private void button1_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 folgende MessageBox:



Außerdem erscheinen die Assembly-Attribute im Eigenschaftsdialog einer Assembly-Datei, z.B.:



11.5 Eine Auswahl nützlicher FCL-Attribute

11.5.1 Bitfelder per FlagsAttribute

Bei einem Enumerationstyp (vgl. Abschnitt 5.5) signalisiert der Designer 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 Null und Eins). Ein Enumerationswert kodiert also die Ausprägungen von *k* dichotomen Merkmalen. Bei der Enumeration **ModifierKeys** aus dem Namensraum **System.Windows.Input** stehen die ersten vier Bits (mit den dezimalen Wertigkeiten 2^0, 2^1, 2^2 und 2^3) für die Vorschalttasten **Alt**, **Strg**, **Umschalt** und **Windows**:[1]

```
[Flags]
[ValueSerializer(typeof(ModifierKeysValueSerializer))]
[TypeConverter(typeof(ModifierKeysConverter))]
public enum ModifierKeys {
   None = 0,
   Alt = 1,
   Control = 2,
   Shift = 4,
   Windows = 8
}
```

- 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. Bei der in Abschnitt 9.8.2.2 erwähnten Enumeration **HorizontalAlignment** kodieren die ersten vier nicht-negativen **int**-Werte jeweils eine horizontale Orientierung eines WPF-Steuerelements gegenüber dem umgebenden Container (**Left**, **Center**, **Right**, **Stretch**).
- Eine bitweise ODER-Verknüpfung der Werte ist zwar syntaktisch erlaubt, aber meist sinnlos. Z.B. liefert

```
HorizontalAlignment.Left | HorizontalAlignment.Right
```

dasselbe Ergebnis wie

```
HorizontalAlignment.Right
```

In der FCL-Enumerationsbasisklasse **Enum** wird die Existenz des **Flags**-Attributs per **IsDefined()**-Aufruf (vgl. Abschnitt 11.2) überprüft[2]

```
eT.IsDefined(typeof(System.FlagsAttribute), false) ...
```

und ggf. in der **ToString()** - Überschreibung für jeden Wert der Enumeration eine kommaseparierte Liste der Merkmale mit einem angeschalteten Bit ausgegeben. Weil alle Enumerationen diese **ToString()** – Überschreibung erben, kann im folgenden Programm demonstriert werden, dass die Summe von zwei **ModifierKeys** -Werten wieder ein sinnvoller Wert dieses Typs ist:

---

[1] Der Quellcode wurde aus dem Mono-Projekt (http://www.mono-project.com/Main_Page) übernommen.
[2] Der Quellcode stammt aus Microsofts *Shared Source Common Language Infrastructure 2.0*.

Quellcode	Ausgabe

`using System;`
`using System.Windows;`
`using System.Windows.Input;`
 	
`class Prog {`
`   static void Main() {`
`      Console.WriteLine("ModifierKeys-Werte:");`
`      for (ModifierKeys i = 0; (int)i <= 8; i++)`
`         Console.WriteLine(" {0}: {1}", (int)i, i);`
`      Console.WriteLine("\nHorizontalAlignment-Werte:");`
`      for (HorizontalAlignment i = 0; (int)i <= 8; i++)`
`         Console.WriteLine(" {0}: {1}", (int)i, i);`
`   }`
`}`
	`ModifierKeys-Werte:`
	` 0: None`
	` 1: Alt`
	` 2: Control`
	` 3: Alt, Control`
	` 4: Shift`
	` 5: Alt, Shift`
	` 6: Control, Shift`
	` 7: Alt, Control, Shift`
	` 8: Windows`
	
	`HorizontalAlignment-Werte:`
	` 0: Left`
	` 1: Center`
	` 2: Right`
	` 3: Stretch`
	` 4: 4`
	` 5: 5`
	` 6: 6`
	` 7: 7`
	` 8: 8`

Bei der Enumeration **HorizontalAlignment** gilt die analoge Aussage jedoch nicht.

11.5.2 Unions per StructLayoutAttribute und FieldOffsetAttribute

Bei der in Abschnitt 3.3.5.1 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 von Typ **float** und **int** definiert:

```
[StructLayout(LayoutKind.Explicit)]
public struct Union {
   [FieldOffset(0)]
   public float f;
   [FieldOffset(0)]
   public int i;
}
```

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 Null beginnen zu lassen. Die beiden Typen (**float**, **int**) haben denselben Platzbedarf von vier Bytes (siehe Abschnitt 3.3.4).

Das Programm schreibt den vom Benutzer angegebenen **float**-Wert in das `f`-Feld einer `Union`-Instanz und liest anschließend über das `i`-Feld der Instanz aus demselben Speicherbereich einen **int**-Wert, der über bitorientierte Operatoren (siehe Abschnitt 3.5.6) untersucht werden kann:

```
using System;
using System.Runtime.InteropServices;

class FloatBits {
   static void Main() {
      Union uni = new Union();
      float f;
      Console.Write("float: ");
      f = Convert.ToSingle(Console.ReadLine());
      uni.f = f;
      int bits = uni.i;
      Console.WriteLine("\nBits:\n1 12345678 12345678901234567890123");
      for (int i = 31; i >= 0; i--) {
         if (i == 30 || i == 22)
            Console.Write(' ');
         if ((1 << i & bits) != 0)
            Console.Write('1');
         else
            Console.Write('0');
      }
   }
}
```

11.6 Übungsaufgaben zu Kapitel 11

1) Ergänzen Sie im Beispiel von Abschnitt 11.3 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.

12 Ein- und Ausgabe über Datenströme

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 per Bildschirm beschränkt und müssen allmählich alternative Quellen bzw. Senken kennen lernen (z.B. Dateien, Netzwerkverbindungen, Datenbankserver). Im aktuellen Kapitel beschränken wir uns auf einfache Verfahren, um Werte elementarer Typen (z.B. **int**, **double**), Zeichenfolgen oder beliebige Objekte in *Dateien* zu schreiben bzw. von dort lesen. Wesentliche Teile der erlernten Techniken werden aber auch im späteren Kapitel über Netzwerkprogrammierung verwendbar sein.

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:

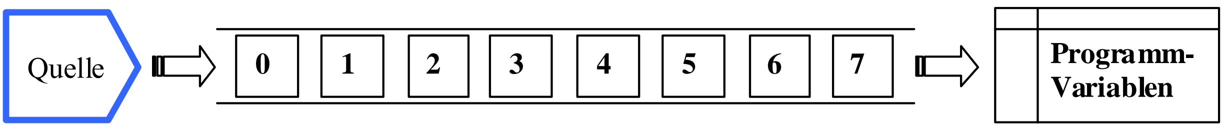
```
using System.IO;
```

12.1 Datenströme aus Bytes

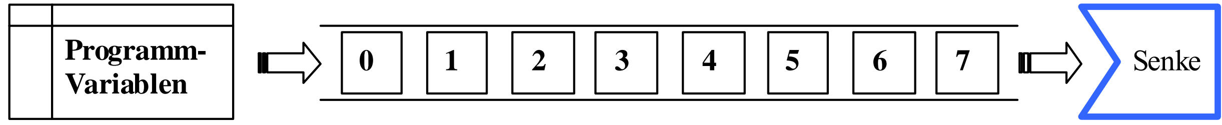
12.1.1 Das Grundprinzip

Im .NET - Framework wird die Ein- und Ausgabe von Daten über so genannte *Ströme* (engl.: ***streams***) abgewickelt.

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 geeigneten Klassen des Namensraums **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.

Mit dem Datenstromkonzept wird bezweckt, Anweisen zur Ein- oder Ausgabe von Daten möglichst unabhängig von den Besonderheiten konkreter Datenquellen und –senken formulieren zu können.

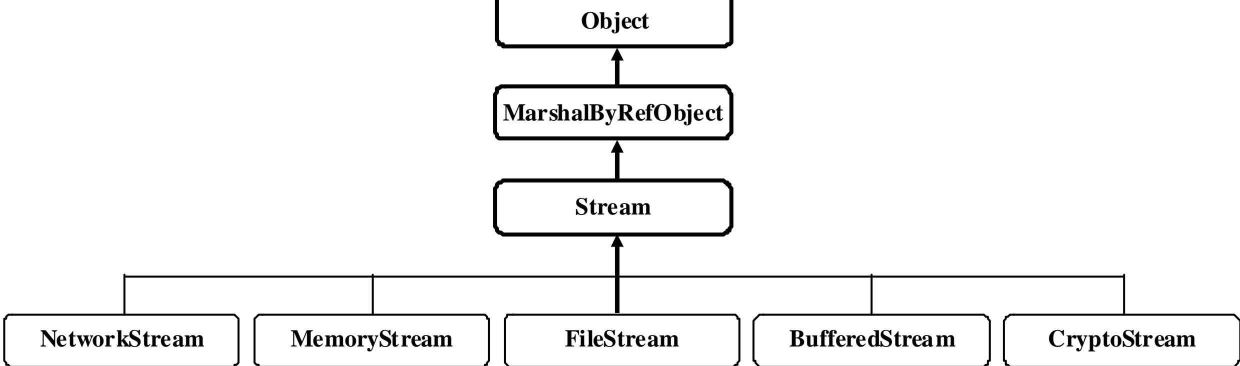
12.1.2 Beispiel

Nach so vielen allgemeinen bzw. abstrakten Bemerkungen wird es Zeit für ein konkretes Beispiel, wobei der Einfachheit halber auf Sinn und Ausnahmebehandlung verzichtet wird. Das folgende Programm erstellt eine Datei, schreibt einen **byte**-Array hinein, liest die Daten wieder zurück und löscht schließlich die Datei:

Quellcode	Ausgabe

<pre>using System;
using System.IO;
 	
class FSDemo {
   static void Main() {
      String name = "demo.bin";
      byte[] arr = {0,1,2,3,4,5,6,7};
      FileStream fs = new FileStream(name, FileMode.Create);
      fs.Write(arr, 0, arr.Length);
      fs.Position = 0;
      fs.Read(arr, 0, arr.Length);
      foreach (byte b in arr)
         Console.WriteLine(b);
      fs.Close();
      File.Delete(name);
   }
}</pre>
	<pre>0
	1
	2
	3
	4
	5
	6
	7</pre>

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 wiedergegeben:



Mit den Details des Beispielprogramms beschäftigen wir uns gleich.

12.1.3 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 12.1.2 wird die Position der geöffneten Datei mit der folgenden Anweisung auf den Dateianfang zurück gesetzt:
  ```
  fs.Position = 0;
  ```
- **public int ReadByte()**
  Mit dieser Methode wird ein Strom-Objekt aufgefordert, *ein* Byte per Rückgabewert vom Typ **int** zu liefern und seine Position entsprechend zu erhöhen. Ist das Ende des Stroms erreicht, wird *keine* Ausnahme geworfen, sondern der Rückgabewert -1 geliefert.
- **public int Read(byte[]** *buffer*, **int** *offset*, **int** *count***)**
  Mit dieser Methode wird ein Strom-Objekt aufgefordert, *count* Bytes zu liefern, im **byte**-Array *buffer* ab Position *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 Strom 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 im **byte**-Array *buffer* ab Position *offset* liegen, und seine Position entsprechend zu erhöhen.
- **public void Flush()**
  Viele Strom-Objekte verwenden beim Schreiben aus Performanzgründen einen Puffer, um die Anzahl der zeitaufwendigen Zugriffe auf eine angeschlossene Senke (z.B. Datei) möglichst gering zu halten. Mit der Methode **Flush()** verlangt man die sofortige Ausgabe des Puffers, so dass der komplette Inhalt für Abnehmer zur Verfügung steht. Bei der Klasse **FileStream** wird eine voreingestellte Puffergröße von 4096 Bytes benutzt.
- **public void Close()**
  Mit dem Schließen eines Datenstroms per **Close()**-Aufruf oder durch eine alternative Technik beschäftigen wir uns gleich in Abschnitt 12.1.4.
- **public long Length {get;}**
  Mit dieser Eigenschaft wird die Länge des Stroms (z.B. die Dateigröße) in Bytes angesprochen.
- **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.:

Quellcode-Segment	Ausgabe

`Console.WriteLine("CanRead:  " + fs.CanRead);`
`Console.WriteLine("CanSeek:  " + fs.CanSeek);`
`Console.WriteLine("CanWrite: " + fs.CanWrite);`
	`CanRead:  True`
	`CanSeek:  True`
	`CanWrite: True`

12.1.4 Schließen von Datenströmen

Mit der Methode **Close()** wird ein Datenstrom geschlossen, was in den von **Stream** abgeleiteten Klassen folgende Maßnahmen beinhaltet:

- Wenn ein Puffer vorhanden ist (z.B. bei der Klasse **FileStream**), wird er durch einen automatischen **Flush()**-Aufruf entleert.
- Betriebssystem-Ressourcen (z.B. Datei-Handles, Netzwerk-Sockets) werden freigegeben.

Beispiel:

```
fs.Close();
```

Nach einem **Close()**-Aufruf existiert das angesprochene .NET – Objekt weiterhin, doch führen Lese- bzw. Schreibversuche zu Ausnahmefehlern. Rufen Sie **Close()** also *nicht* auf, wenn solche Ausnahmefehler möglich sind, weil im Programm noch weitere Referenzen auf das **Stream**-Objekt existieren.

Derartige Pannen sind ausgeschlossen, wenn Sie ein überflüssig gewordenes Datenstromobjekt dem Garbage Collector überlassen, was Richter (2006, S. 501) nachdrücklich empfiehlt. Über die Finalisierungsmethode (vgl. Abschnitt 4.4.4) der von **Stream** abgeleiteten Klassen ist sichergestellt, dass

vor dem Entfernen eines Objekts per Garbage Collector alle verwendeten Ressourcen (Dateien, Netzwerkverbindungen) freigegeben werden. Dabei kommt die Methode **Dispose()** zum Einsatz, die auch bei einem **Close()**-Aufruf die eigentlichen Aufräumungsarbeiten (inklusive Entleeren des Puffers) verrichtet. Die Klasse **Stream** muss eine **Dispose**()-Methode bereithalten, weil sie das Interface **IDisposable** (man sagt auch: *das Beseitigungsmuster*) implementiert.

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 12.2.2). Es kommt zu Datenverlusten wenn der Garbage Collector bei seinen Aufräumarbeiten (ohne garantierte Reihenfolge!) das **Stream**-Objekt *vor* dem **StreamWriter**–Objekt beseitigt.

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 (wegen einer Endlosschleife oder eines unerreichbaren Sperrobjekts, siehe Abschnitt 13.2).[1] Insgesamt sollte man sich bei einem puffernden Ausgabestrom nicht auf das Finalisieren durch die CLR verlassen und stattdessen (per **Flush()**-, **Dispose()**- oder **Close()**-Aufruf) die Pufferinhalte rechtzeitig in die angeschlossene Senke befördern.

Außerdem ist es oft wichtig, (Ein- oder Ausgabe-)Ströme durch ein explizites **Close()** so früh wie möglich zu schließen, um (exklusive) Zugriffe durch andere Interessenten zu ermöglichen.

Im Beispiel `FSDemo` von Abschnitt 12.1.2 ist das Schließen der Datei erforderlich, um sie anschließend löschen zu können.

Trotz der modernen Softwaretechnologie im .NET - Framework ist also einige Aufmerksamkeit erforderlich, um Fehler durch voreilige oder vergessene **Close()**-Aufrufe zu vermeiden.

Mit der **using**-Anweisung (nicht zu verwechseln mit der **using**-*Direktive* zum Importieren eines Namensraums) kann man einen Block definieren und Objekte erzeugen, die nur innerhalb des **using**-Blocks gültig (referenziert) sind. Beim Verlassen des Blocks werden automatisch per **Dispose()**-Aufruf alle verwendeten Ressourcen (Dateien, Netzwerkverbindungen) freigegeben, z.B.:

```
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 FileStream(name, FileMode.Create)) {
      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);
  }
}
```

[1] Auf den folgenden MSDN-Webseite werden Details zum Finalisierungsverhalten der CLR dokumentiert: http://msdn.microsoft.com/en-us/library/system.object.finalize.aspx

Per **using**-Anweisung wird das Schließen eines Stroms elegant gelöst, wobei der Compiler im Hintergrund eine **try-finally** - Anweisung erstellt. In manchen Fällen ist es aber sinnvoll bzw. erforderlich, die von Ein-/Ausgabe – Methoden zu erwartenden Ausnahmen zu behandeln.

12.1.5 Ausnahmen behandeln

Weil Methodenaufrufe bei Datenstromobjekten diverse Ausnahmen produzieren können (z.B. **FileNotFoundException**, **UnauthorizedAccessException**, **IOExeption**), sind sie am besten in einem **try** – Block untergebracht. Wenn ein **Close**()-Aufruf angemessen ist (vgl. die Warnungen und Empfehlungen in Abschnitt 12.1.4), gehört er in den **finally**-Block der **try** – Anweisung, damit er auf jeden Fall ausgeführt wird (vgl. Abschnitt 10.2.1.2). Alternativ kann eine **using**-Anweisung für das garantierte Schließen sorgen (siehe Abschnitt 12.1.4). In der folgenden Variante des Einstiegsbeispiels ist das Abfangen der Ausnahmen immerhin angedeutet:

```
using System;
using System.IO;

class FSDemo {
   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.Create);
         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 {}
   }
}
```

Mit Hilfe der **using**-Anweisung (siehe Abschnitt 12.1.4) lässt sich dasselbe Verhalten mit weniger Schreibaufwand realisieren:

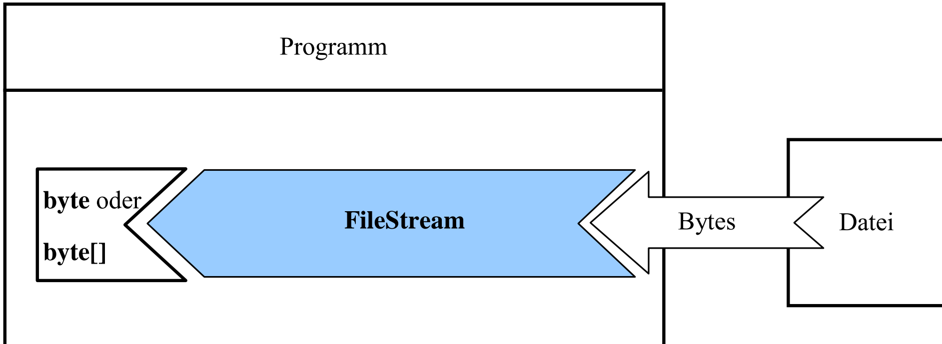
```
static void Main() {
   String name = "demo.bin";
   byte[] arr = { 0, 1, 2, 3, 4, 5, 6, 7 };
   try {
      using (FileStream fs = new FileStream(name, FileMode.Create)) {
         fs.Write(arr, 0, arr.Length);
         fs.Position = 0;
         fs.Read(arr, 0, arr.Length);
         foreach (byte b in arr)
            Console.WriteLine(b);
      }
      File.Delete(name);
   } catch (Exception e) {
      Console.WriteLine(e.Message);
   }
}
```

Nun steht der **Close**()-Aufruf im **finally**-Block der vom Compiler aus der **using**-Anweisung erstellten **try-finally** – Anweisung.

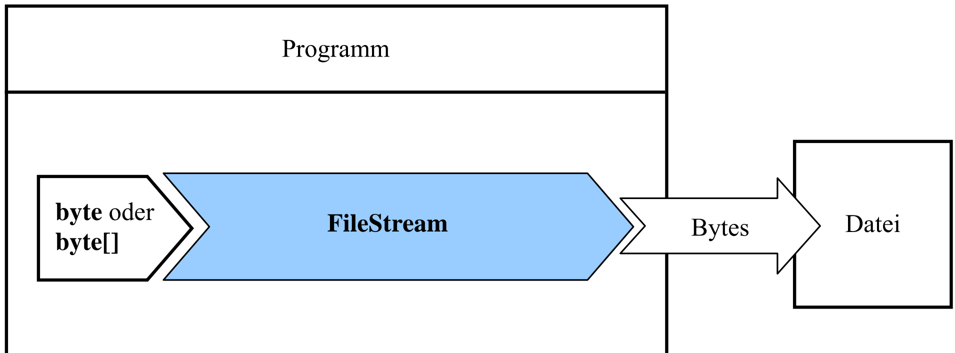
Bei den Beispielen im weiteren Verlauf von Kapitel 12 werden wir der Einfachheit halber meist auf eine Ausnahmebehandlung verzichten.

12.1.6 FileStream

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 eine per Pfadnamen identifizierte Datei erstellt und zum Lesen und/oder Schreiben geöffnet:

```
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 zahlreiche Überladungen des Konstruktors.

FileStream-Objekte verwenden einen Puffer, um die Anzahl der Zugriffe auf die angeschlossene Datei möglichst gering zu halten. Beim Schließen einer Datei (durch den Garbage Collector oder einen expliziten **Close**()-Aufruf) werden gepufferte Schreibvorgänge automatisch ausgeführt. Einige Überladungen des Konstruktors bieten einen Parameter, um die voreingestellte Puffergröße von 4096 Bytes zu ändern.

12.1.6.1 Öffnungsmodus

Beim Erstellen eines **FileStream**-Objekts kann man den Öffnungsmodus über einen Wert des Enumerationstyps **FileMode** wählen:

Modus	Beschreibung
Append	Die Datei wird geöffnet oder neu erzeugt.
Create	Ist die Datei noch *nicht* vorhanden, wird sie angelegt (wie bei **CreateNew**), andernfalls wird sie überschrieben (wie bei **Truncate**).
CreateNew	Es wird eine neue Datei angelegt, oder eine **IOEexeption** geworfen, falls eine Datei mit dem gewünschten Namen bereits existiert.
Open	Es wird eine vorhandene Datei geöffnet, oder eine **IOEexeption** 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 **IOEexeption** 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 vor dem ersten Byte, und es bestehen einige Anforderungen an die Schreibberechtigung für das **FileStream**-Objekt (siehe Abschnitt 12.1.6.2).

Mit einem Öffnungsmodus wählt man also 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 12.1.6.2)?

12.1.6.2 Zugriffsmöglichkeiten für das erstellte FileStream-Objekt

Bei einigen Überladungen des **FileStream**-Konstruktors lassen sich über einen Parameter vom Enumerationstyp **FileAccess** die Zugriffsmöglichkeiten für den eigenen Prozess vereinbaren, wobei auf Verträglichkeit mit dem Eröffnungsmodus zu achten ist. Es sind folgende Alternativen verfügbar:

- **FileAccess.Read**
- **FileAccess.Write**
- **FileAccess.ReadWrite**

In folgendem Beispiel wird eine Datei zum Lesen geöffnet:

```
FileStream fs = new FileStream("demo.bin", FileMode.Open,
                               FileAccess.Read);
```

Die in Abschnitt 12.1.6.1 beschriebenen Öffnungsmodi 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

12.1.6.3 Zugriffsmöglichkeiten für andere Interessenten

Für eine per **FileStream**-Konstruktor geöffnete Datei dürfen andere Interessenten (im eigenen oder in einem fremden Prozess) per Voreinstellung simultan *lesend* zugreifen. Bei einigen **FileStream**-Konstruktorüberladungen kann man die Freigabe für den gemeinsamen Zugriff über einen Parameter vom Enumerationstyp **FileShare** regeln. Dabei sind im Wesentlichen die folgenden Alternativen verfügbar:

- **FileShare.None**
- **FileShare.Read**
- **FileShare.Write**
- **FileShare.ReadWrite**

In folgendem Beispiel wird die gemeinsame Nutzung komplett verweigert:

```
FileStream fs = new FileStream("demo.bin", FileMode.Create,
                                           FileAccess.ReadWrite,
                                           FileShare.None);
```

Das Betriebssystem spricht allerdings auch noch ein Wörtchen mit und wird z.B. nicht erlauben, dass eine Datei in zwei Prozessen gleichzeitig zum Schreiben geöffnet ist.

Beim (etwas ungewöhnlichen) Simultanzugriff *innerhalb* einer .NET –Anwendung, realisiert über zwei verschiedene **FileStream**-Objekte, ist zu beachten, dass *beide* Objekte über einen geeigneten **FileShare**-Parameterwert im Konstruktor den Partner berechtigen müssen (Eller & Kofler, 2005, S. 369). Diese Konstellation

```
FileStream fs1=new FileStream(name,FileMode.Create,FileAccess.ReadWrite,FileShare.Read);
   . . .
FileStream fs2=new FileStream(name,FileMode.Open,FileAccess.Read);
```

scheitert mit einem Ausnahmefehler, weil `fs2` den voreingestellten **FileShare**-Wert **Read** verwendet, also dem zuvor tätig gewordenen `fs1` kein Schreibrecht zugesteht. So klappt es:

```
FileStream fs2=new FileStream(name,FileMode.Open,FileAccess.Read,FileShare.ReadWrite);
```

12.2 Verarbeitung von Daten mit höherem Typ

Bisher haben wir uns auf das Schreiben und Lesen von *Bytes* beschränkt. Im Abschnitt 12.2 lernen Sie Verfahren kennen, um Daten mit einem beliebigen (aus mehreren Bytes bestehenden) Typ (z.B. **int**, **double**, **String**, beliebige Klasse oder Struktur) in Dateien zu schreiben oder von dort zu lesen.

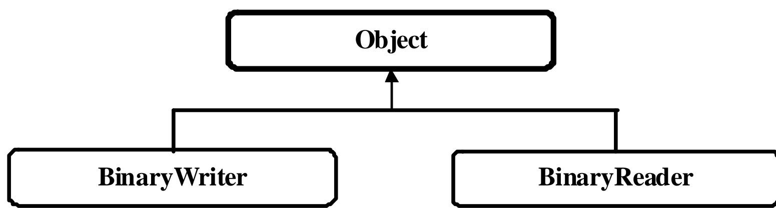
Bei den Dateien auf einem Rechner kann man unterscheiden:

- **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. Für den menschlichen Konsum ist eine Binärdateien allerdings nicht geeignet. Öffnet man sie mit einem Texteditor, ist nur eine wirre Folge von (Sonder-)zeichen zu sehen. In Abschnitt 12.2.1 werden die zum Schreiben bzw. Lesen binärer Daten 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 Zeichenkodierung 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. In Abschnitt 12.2.2 werden die zum Schreiben bzw. Lesen von Zeichenfolgen konstruierten Klassen **StreamWriter** bzw. **StreamReader** vorgestellt.

In Abschnitt 12.2.3 beschäftigen wir uns mit dem Schreiben und Lesen von kompletten Objekten (oder auch Strukturinstanzen), wobei eine Binärdatei oder eine XML-formatierte Textdatei zum Einsatz kommen kann.

12.2.1 Schreiben und Lesen im Binärformat

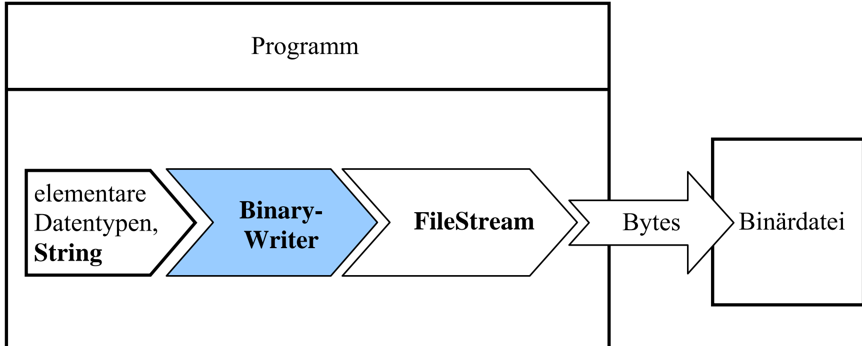
Um Werte von einem beliebigen elementaren Datentyp (z.B. **int**, **double**) sowie Zeichenfolgen in einen binär organisierten Strom (z.B. in eine Binärdatei) zu schreiben bzw. aus einem Binärstrom mit bekanntem Aufbau zu lesen, verwendet man ein Objekt der Klasse **BinaryWriter** bzw. **BinaryReader**.



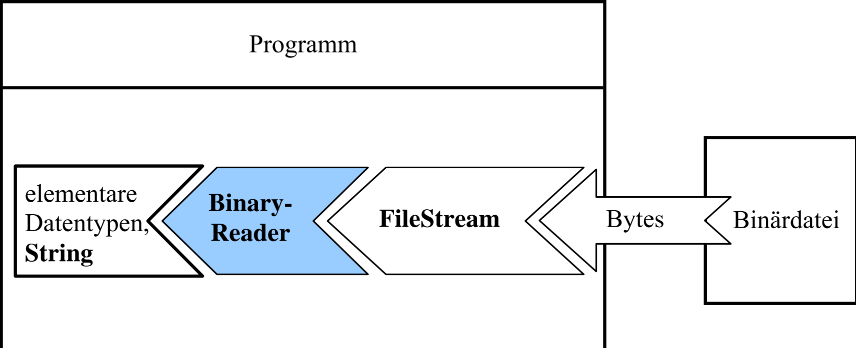
Die beiden Klassen stammen *nicht* von der in Abschnitt 12.1 behandelten Klasse **Stream** ab, verwenden aber für die Verbindung mit einer Datenquelle oder –senke ein **Stream**-Objekt, das im Konstruktor anzugeben ist, z.B.:

public BinaryWriter(Stream *ausgabestrom***)**

Im Unterschied zu den „bidirektionalen" **Stream**-Klassen sind für das Schreiben bzw. Lesen von elementaren Datenwerten „gerichtete" Klassen zuständig. Schreibt man per **BinaryWriter** in eine Datei, entsteht 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";
      FileStream fso = new FileStream(name, FileMode.Create);
      BinaryWriter bw = new BinaryWriter(fso);
      bw.Write(4711);
      bw.Write(3.1415926);
      bw.Write("Nicht übel");
      bw.Close();

      FileStream fsi = new FileStream(name, FileMode.Open, FileAccess.Read);
      BinaryReader br = new BinaryReader(fsi);
      Console.WriteLine(br.ReadInt32() + "\n" +
                        br.ReadDouble() + "\n" +
                        br.ReadString());
      br.Close();
   }
}
```

Um das Schreiben und das Lesen unabhängig voneinander vorzuführen, wird im Beispiel jeweils ein eigenes **FileStream**-Objekt mit passenden **FileMode**- bzw. **FileAccess**- Konstruktorparameterwerten erstellt. Es wäre möglich, mit *einem* **FileStream**-Objekt zu arbeiten und dessen **Position**-Eigenschaft nach dem Schreiben wieder auf Null zu setzen (siehe Beispiel in Abschnitt 12.1.1).

Die von beiden **FileStream**-Objekten verwendete Datei wird zunächst mit dem **FileMode.Create** geöffnet. Nach den Schreibzugriffen per **Write**()-Methode wird die Datei per **Close**()-Aufruf geschlossen, damit das anschließende Öffnen mit dem **FileMode.Open** gelingt. Der **Close**()-Aufruf kann sich an das **FileStream**- oder an das **BinaryWriter**-Objekt richten, wobei er im letztgenannten Fall durchgereicht wird.

Durch den **Close**()-Aufruf wird der Schreibpuffer des **FileStream**-Objekts `fso` geleert, so dass 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.[1] 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 Überladung der Methode **Write**() besitzt, sind in der Klasse **BinaryReader** typspezifisch benannte Lesemethoden vorhanden (z.B. **ReadInt32**(), **ReadDouble**()).

Die Ausgabe des Programms:

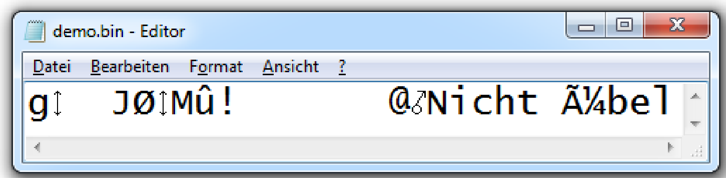
```
4711
3,1415926
Nicht übel
```

Es macht wenig Sinn, die vom Beispielprogramm erzeugte Binärdatei mit einem Texteditor zu öffnen:

---

[1] Z.B werden bei einem Aufruf der **Write**()-Methode mit **int**-Parameter die vier Bytes sofort an den Ausgabestrom übergeben (Quellcode aus Microsofts *Shared Source Common Language Infrastructure 2.0*):

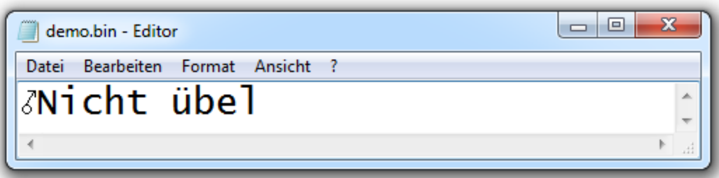
```
public virtual void Write(int value) {
   _buffer[0] = (byte) value;
   _buffer[1] = (byte)(value >> 8);
   _buffer[2] = (byte)(value >> 16);
   _buffer[3] = (byte)(value >> 24);
   OutStream.Write(_buffer, 0, 4);
}
```



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 *Ausgabekodierung* der Zeichen relevant, die im Arbeitsspeicher eines Rechners bekanntlich in Unicode-Kodierung vorliegen. Die Klasse **BinaryWriter** verwendet per Voreinstellung dasselbe **UTF8Encoding** wie die später vorzustellenden **TextWriter**-Klassen. Offenbar ist die vom Windows-Texteditor unterstellte Kodierung bei den meisten Zeichen mit dem **UTF8Encoding** kompatibel. Über einen **BinaryWriter**-Konstruktor mit entsprechendem Parameter stehen auch andere Kodierungen zur Verfügung:

public BinaryWriter(Stream *ausgabestrom*, **Encoding** *kodierung*)

Schreibt man per **BinaryWriter** an Stelle der gemischten Ausgaben ausschließlich Text, kann der Windows-Texteditor beim Öffnen der Ergebnisdatei die zugrunde liegende Kodierung übrigens besser erkennen:



Trotz der gemeinsamen Kodierungsvoreinstellung gibt es zwischen der Klasse **BinaryWriter** und den **TextWriter**-Klassen doch 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 - Kodierung:

- 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 - Kodierung, siehe Abschnitt 12.2.2) 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

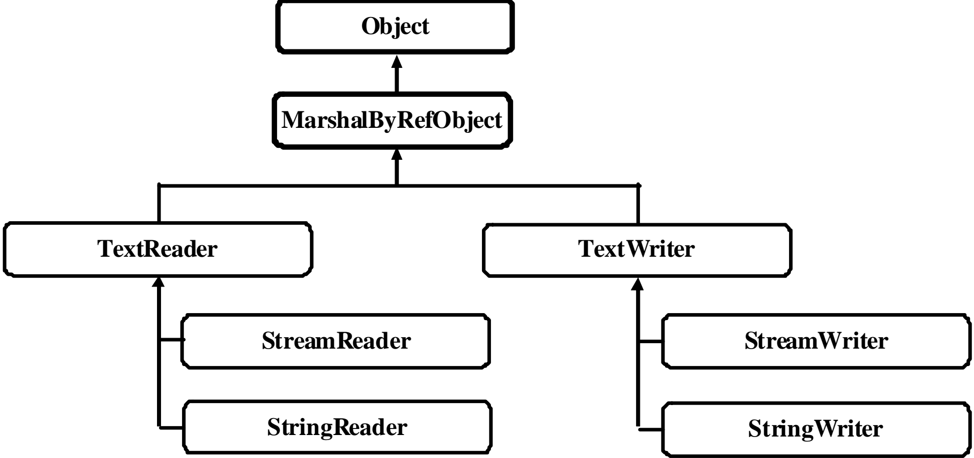
Es besteht kein Risiko, wenn ein Gespann aus einem **BinaryWriter**- und einem **FileStream**-Objekt dem Garbage Collector anheim fallen, obwohl beim automatischen Finalisieren (ohne garantierte Reihenfolge!) zuerst das **FileStream**-Objekt beseitigt werden könnte:

- **BinaryWriter**–Objekte besitzen keinen lokaler Puffer, der beim Abräumen geleert werden müsste.
- In der Klasse **BinaryWriter** ist keine Finalisierungsmethode vorhanden, so dass beim Abräumen kein Zugriff auf das zugrunde liegende (und eventuell nicht mehr existente) **Stream**-Objekt stattfinden kann.

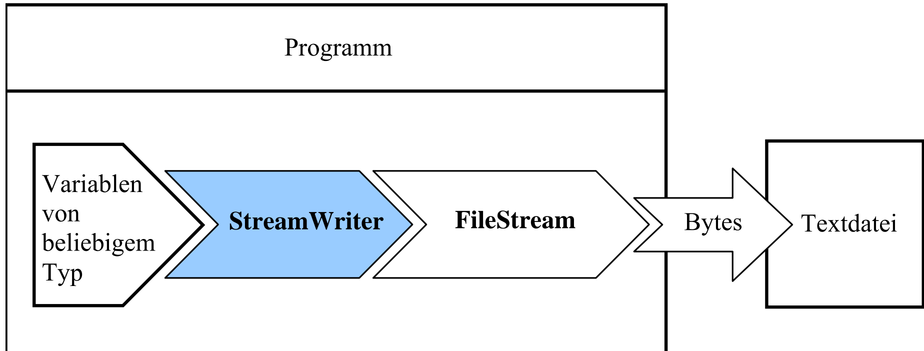
Objekte der anschließend behandelten Klasse **StreamWriter** müssen aufgrund ihres lokalen Puffers jedoch unbedingt *vor* dem zugrunde liegenden **Stream**-Objekt geschlossen werden, was nur durch einen **Close**()-Aufruf (oder einen äquivalenten **Dispose**()-Aufruf) an das **StreamWriter**–Objekt sicher gestellt ist (eventuell per **using**-Block automatisiert, vgl. Abschnitt 12.1.4).

12.2.2 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, so dass bei der Versorgung von numerischen Variablen aus textuellen Eingabedaten etwas Eigeninitiative gefragt ist (siehe Übungsaufgabe in Abschnitt 12.4). Beide Klassen sind abstrakt, doch bietet die FCL auch konkrete Ableitungen für **Stream**- bzw. **String**-Objekte als Senken bzw. Quellen:

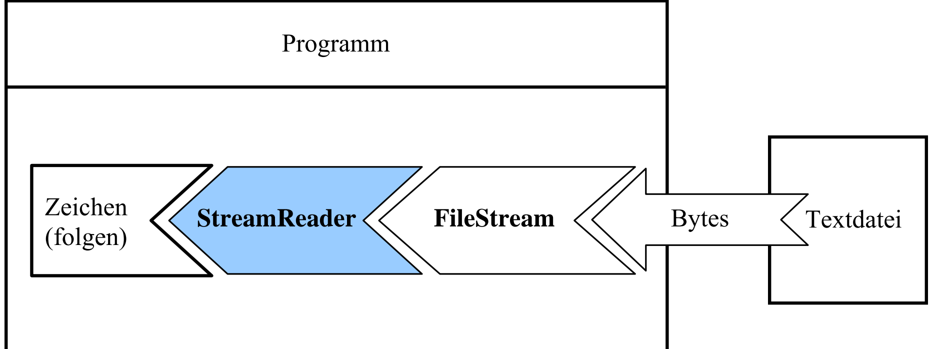


Um in eine Textdatei zu schreiben bzw. von dort zu lesen, verwendet man Objekte der Klassen **StreamWriter** bzw. **StreamReader**, die jeweils über eine Instanzvariable mit einem **FileStream**-Objekt verbunden sind (analog zu den Klassen **BinaryWriter** und **BinaryReader**).[1] Beim Schreiben haben wir also 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.

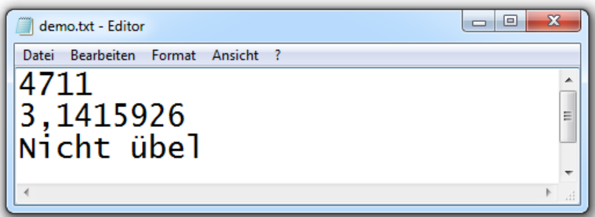
Das folgende Beispielprogramm schreibt einen **int**- und einen **double**-Wert sowie eine Zeichenfolge per **StreamWriter** über ein implizit erzeugtes **FileStream**-Objekt in eine Datei. Anschließend werden die Daten über einen **StreamReader** eingelesen, der sich auf ein explizit erzeugtes **FileStream**-Objekt stützt:

```
using System;
using System.IO;

class StreamWrtRd {
   static void Main() {
      String name = "demo.txt";
      StreamWriter sw = new StreamWriter(name);
      sw.WriteLine(4711);
      sw.WriteLine(3.1415926);
      sw.WriteLine("Nicht übel");
      sw.Close();

      StreamReader sr = new StreamReader(
                            new FileStream(name, FileMode.Open, FileAccess.Read));
      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());
      }
      sr.Close();
   }
}
```

Das Beispielprogramm liefert folgende Ausgabe:

```
Inhalt der Datei U:\Eigene Dateien\C#\EA\StreamWrtRd\bin\Debug\demo.txt:

1:      4711
2:      3,1415926
3:      Nicht übel
```

Auch die erzeugte Textdatei ist ansehnlich:



Bei den **TextWriter**-Methoden **Write()** und **WriteLine()** treffen wir im Wesentlichen auf dieselben Signaturen wie bei den gleichnamigen **Console**-Methoden, die Ihnen aus zahlreichen Beispielen vertraut sind.

Im Unterschied zu einem **BinaryWriter** (siehe Abschnitt 12.2.1) besitzt ein **StreamWriter** einen lokalen Puffer (Datentyp: **char[]**, voreingestellte Größe: 1024). Daher muss ein **StreamWriter** unbedingt nach Gebrauch per **Close()** (oder **Dispose()**) geschlossen werden. Das zugrunde liegende **Stream**-Objekt wird dabei automatisch ebenfalls geschlossen. Es wäre riskant, das Schließen dem Garbage Collector zu überlassen, für den keine Arbeitsreihenfolge garantiert ist. Wenn er das **Stream**-Objekt *vor* dem **StreamWriter**–Objekt schließt, kann letzteres seinen Puffer nicht mehr ausgeben.

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).[1]

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 interne Puffer in den Ausgabestrom geschrieben
- und dann ein **Flush()**-Aufruf an das **Stream**-Objekt gerichtet.

Hinweise zu einigen **TextReader**-Methoden:

- **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, der ...
  - das Steuerzeichen Carriage Return (0x000D),
  - das Steuerzeichen Line Feed (0x000A),
  - eine Sequenz aus den Sonderzeichen Carriage Return und Line Feed,
  - die in der statischen **Environment.NewLine** hinterlegte Zeichenfolge,
  - oder das End des Stroms

  folgt. 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 oder aber den Wert -1, wenn der Strom kein Zeichen mehr enthält, und verschiebt die Position des Eingabestroms entsprechend.

[1] Die folgende Implementierung der **Write()**-Methode mit **char**-Parameter aus der Klasse **StreamWriter** zeigt, wie sich die öffentliche **AutoFlush**-Eigenschaft über das private **autoFlush**-Feld auf das Pufferungsverhalten auswirkt (Quellcode aus Microsofts *Shared Source Common Language Infrastructure 2.0*):

```
public override void Write(char value) {
  if (charPos == charLen) Flush(false, false);
  charBuffer[charPos++] = value;
  if (autoFlush) Flush(true, false);
}
```

- **public int Peek()**
  Diese Methode liefert wie **Read()** die Unicode-Nummer des nächsten Zeichens oder aber den Wert -1, wenn der Strom kein Zeichen mehr enthält. Die Position des Stroms bleibt dabei aber unverändert.

Per Voreinstellung schreiben bzw. lesen die **StreamWriter** bzw. –**Reader** Unicode-Zeichen unter Verwendung der Platz sparenden **UTF8**-Kodierung. Bei diesem Schema werden die Unicode-Zeichen durch eine variable Anzahl von Bytes kodiert. 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 Kodierungen einstellen, z.B.:

```
FileStream fs = new FileStream("unicode.txt", FileMode.Create);
StreamWriter swUnicode = new StreamWriter(fs, Encoding.Unicode);
```

Die statische Eigenschaft **Unicode** der Klasse **Encoding** im Namensraum **System.Text** zeigt auf ein Objekt der Klasse **UnicodeEncoding**, das die Kodierung übernimmt. Es verzichtet auf Platzsparmaßnahmen und verwendet für *jedes* Zeichen 2 Bytes.

Auch bei Objekten der Klasse **StreamReader** lässt sich die voreingestellte **UTF8**-Kodierung über alternative Konstruktoren ersetzen, was z.B. beim Lesen der häufig anzutreffenden Textdateien mit **ANSI-Kodierung** erforderlich ist. Im folgenden Programm

```
using System;
using System.IO;
using System.Text;

class AnsiTextLesen {
   static String name = "AnsiText.txt";
   static void Main() {
      FileStream fs = new FileStream(name, FileMode.Open, FileAccess.Read);
      StreamReader sr = new StreamReader(fs);
      Console.WriteLine("Mit UTF8-Kodierung gelesen:");
      while (sr.Peek() >= 0)
         Console.WriteLine(sr.ReadLine());
      fs.Position = 0;
      sr = new StreamReader(fs, Encoding.Default);
      Console.WriteLine("\nMit ANSI-Kodierung gelesen:");
      while (sr.Peek() >= 0)
         Console.WriteLine(sr.ReadLine());
      fs.Close();
   }
}
```

wird beim Lesen einer Textdatei zuerst die UTF8- und dann die ANSI-Kodierung unterstellt. Bei einer Datei mit ANSI-Kodierung und folgendem Inhalt

```
ANSI-kodierte Umlaute: üöä
```

---

[1] Im Bereich von 0 bis 127 befinden sich im Unicode dieselben Zeichen wir im ASCII-Code (*American Standard Code for Information Interchange*) aus der Steinzeit der Datenverarbeitung

resultiert die Ausgabe:

```
Mit UTF8-Kodierung gelesen:
ANSI-kodierte Umlaute: ???

Mit ANSI-Kodierung gelesen:
ANSI-kodierte Umlaute: üöä
```

Über die statische **Encoding**-Eigenschaft **Default** erhält man eine zur aktuellen ANSI-Codepage des Betriebssystems passende Kodierung.

12.2.3 Serialisieren von Objekten

Wer objektorientiert programmiert, möchte natürlich auch objektorientiert speichern und laden. Erfreulicherweise können in C# Objekte tatsächlich in der Regel genau so einfach wie Instanzen mit elementarem Typ in einen Datenstrom geschrieben bzw. von dort gelesen werden. Die keinesfalls triviale Übersetzung eines Objekts mit all seinen Instanzvariablen und den enthaltenen (d.h. von Instanzvariablen referenzierten) Objekten in einen Bytestrom bezeichnet man recht treffend als *Objektserialisierung*. Beim Einlesen werden alle Objekte mit ihren Instanzvariablen wiederhergestellt und die Referenzen zwischen den Objekten in den Ausgangszustand gebracht. Die FCL-Dokumentation spricht von (De)serialisieren eines *Objektdiagramms*. Dank Autoboxing lassen sich auch Strukturinstanzen (de)serialisieren.

Das Serialisieren der Instanzen eines Typs muss explizit bei der Definition über das Typ-Attribut **Serializable** erlaubt werden. Dies darf nicht unbedacht erfolgen, z.B. weil die Serialisierbarkeit aller Datentypen von Instanzvariablen erforderlich ist. Aus nahe liegenden Gründen haben die FCL-Designer das Attribut **Serializable** als *nicht* vererbbar definiert, so dass es nicht auf abgeleitete Klassen übertragen wird.

Bei Bedarf können einzelne Felder über das Attribut **NonSerialized** ausgeschlossen werden. Dies kommt z.B. in Frage, wenn ...

- ein Feld aus Sicherheitsgründen nicht in den Ausgabestrom gelangen soll,
- ein Feld temporäre Daten enthält, so dass ein Speichern überflüssig bzw. sinnlos ist,
- ein Feld einen nicht-serialisierbaren Datentyp hat.
  Wird ein solches Feld nicht von der Serialisierung ausgeschlossen, kommt es ggf. zu einer **SerializationException**.

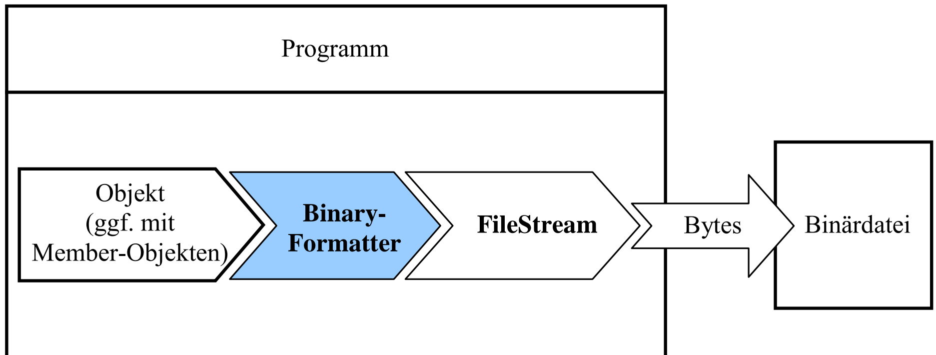
Über das Implementieren der Schnittstelle **ISerializable** gewinnt der Typdesigner Kontrolle über die (De)serialisierung, muss aber dazu die **ISerializable**-Methode **GetObjectData()** implementieren.

Die im folgenden Quellcode definierte Klasse `Kunde` ist als serialisierbar deklariert, wobei jedoch für das Feld `stimmung` eine Ausnahme gemacht wird:

```
using System;
using System.Runtime.Serialization;

[Serializable]
public class Kunde {
   int nr;
   string vorname;
   string name;
   int nkaeufe;
   double aussen;
   [NonSerialized]
   int stimmung;
```

```
    public Kunde(int nr_, string vorname_, string name_, int stimmung_,
                 int nkaeufe_, double aussen_) {
      nr = nr_;
      vorname = vorname_;
      name = name_;
      stimmung = stimmung_;
      nkaeufe = nkaeufe_;
      aussen = aussen_;
    }

    public void prot() {
      Console.WriteLine("Kundennummer: \t" + nr);
      Console.WriteLine("Name: \t\t" + vorname + " " + name);
      Console.WriteLine("Stimmung: \t" + stimmung);
      Console.WriteLine("Anz.Einkäufe: \t" + nkaeufe);
      Console.WriteLine("Aussenstände: \t" + aussen+ "\n");
    }
  }
```

Den recht anspruchsvollen Job der (De)Serialisierung übernimmt 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**), die ein kompaktes Binärformat verwendet. Beim Abspeichern in eine Datei resultiert die folgende Verarbeitungskette:



Im folgenden Programm wird ein ...

Array-Objekt vom Typ `Kunde`
  zusammen mit den beiden `Kunde`–Elementobjekten
    mitsamt den jeweils enthaltenen **String**-Objekten,
    aber ohne das Feld `stimmung`

(de)serialisiert:

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

class Serialisierung {
  static string name = "demo.bin";
  static void Main() {
    Kunde[] kunden = new Kunde[2];
    kunden[0] = new Kunde(1, "Fritz", "Orth", 1, 13, 426.89);
    kunden[1] = new Kunde(2, "Ludwig", "Knüller", 2, 17, 89.10);

    Console.WriteLine("Zu sichern:\n");
    foreach (Kunde k in kunden)
      k.prot();
```

```
      FileStream fs = new FileStream(name, FileMode.Create);
      IFormatter bifo = new BinaryFormatter();

      bifo.Serialize(fs, kunden);

      fs.Position = 0;
      Console.WriteLine("\nRekonstruiert:\n");
      Kunde[] desKunden = (Kunde[]) bifo.Deserialize(fs);
      foreach(Kunde k in desKunden)
         k.prot();
      fs.Close();
   }
}
```

Pro **Serialize()**-Aufruf wird *ein* Wurzelobjekt mitsamt den Werttyp-Feldern sowie den direkt oder indirekt referenzierten Objekten, also ein komplettes Objektdiagramm, geschrieben. Um weitere Objektdiagramme in denselben Datenstrom zu befördern, sind entsprechend viele Aufrufe erforderlich. Das Beispielprogramm hätte die `Kunde`-Objekte auch einzeln serialisieren können.

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.

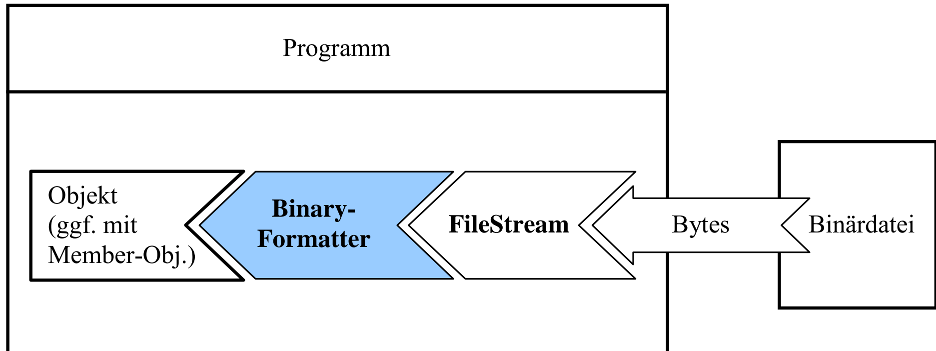
Weil **Deserialize()** den Rückgabewert **Object** hat, ist eine Typumwandlung erforderlich. Ein **Deserialize()**-Aufruf liest das nächste im Datenstrom befindliche Wurzelobjekt samt Anhang (also *ein* Objektdiagramm). Um weitere Wurzelobjekte aus demselben Datenstrom zu lesen, sind entsprechend viele Aufrufe erforderlich.

Das Beispielprogramm produziert folgende Ausgabe:

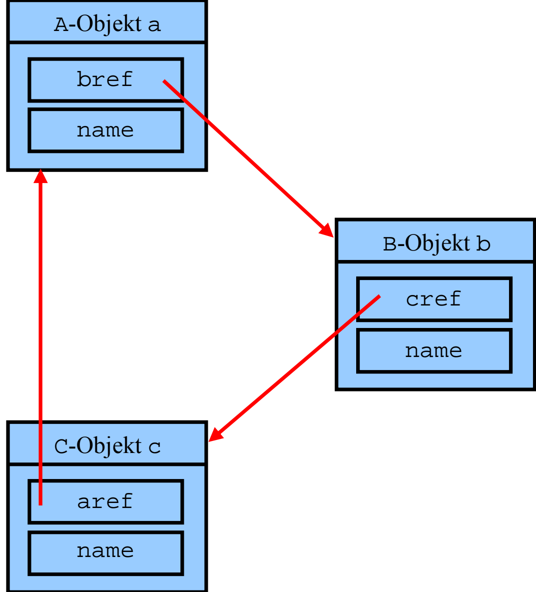
```
Zu sichern:

Kundennummer:    1
Name:            Fritz Orth
Stimmung:        1
Anz.Einkäufe:    13
Aussenstände:    426,89

Kundennummer:    2
Name:            Ludwig Knüller
Stimmung:        2
Anz.Einkäufe:    17
Aussenstände:    89,1


Rekonstruiert:

Kundennummer:    1
Name:            Fritz Orth
Stimmung:        0
Anz.Einkäufe:    13
Aussenstände:    426,89

Kundennummer:    2
Name:            Ludwig Knüller
Stimmung:        0
Anz.Einkäufe:    17
Aussenstände:    89,1
```

Die Instanzvariable `stimmung` der eingelesenen Kunden besitzen den Initialwert Null, während die übrigen Elementvariablen bei der (De)serialisierung ihre Werte behalten.

In der folgenden Abbildung wird die Rekonstruktion der Objekte skizziert:



Auch zirkuläre Referenzen wie in folgender 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,

```
FileStream 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, so dass der **WriteLine**()-Aufruf zu folgender 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

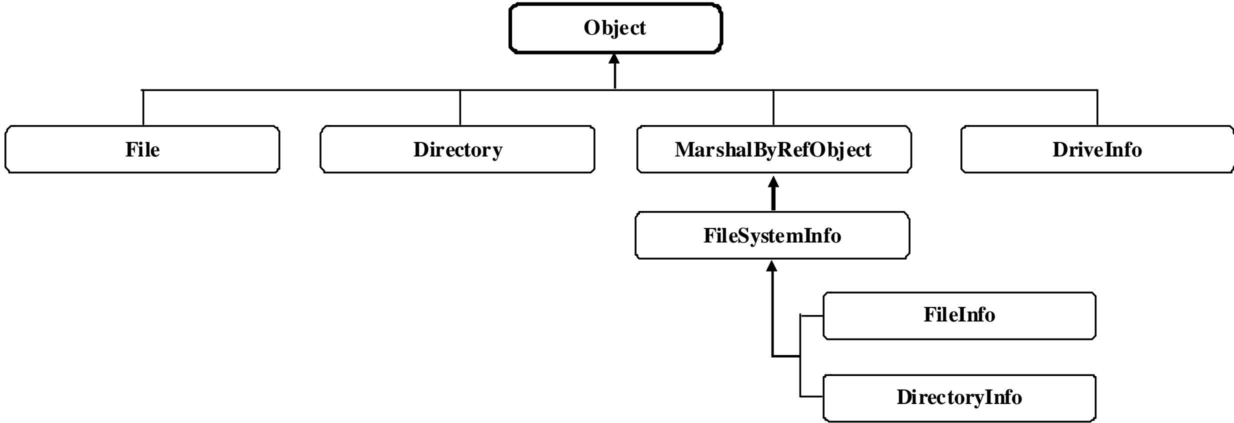
```
b == na.bref
```

den Wert **false** besitzt.

Ist eine XML-basierte Ausgabe gefragt, verwendet man zum Serialisieren die Klasse **XmlSerializer**, die allerdings im Unterschied zur Klasse **BinaryFormatter** keine privaten Felder und keine zirkulären Referenzen unterstützt.

12.3 Verwaltung von Dateien und Verzeichnissen

Zur Verwaltung von Dateien bzw. Verzeichnissen enthält die FCL jeweils eine Klasse mit statischen Methoden (**File** bzw. **Directory**) sowie eine Klasse mit Instanzmethoden (**FileInfo** bzw. **DirectoryInfo**):



Man kann Dateien bzw. Verzeichnisse erstellen, kopieren, löschen, umbenennen und verschieben sowie diverse Datei- bzw. Verzeichnisattribute einsehen und verändern. Im Zusammenhang mit dem **TreeView**-Steuerelement werden wir später noch die Klasse **DriveInfo** behandeln, die ein komplettes Laufwerk repräsentiert.

12.3.1 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 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";

      FileStream fs = File.Create(PFAD1);
      fs.Close();

      StreamWriter sw = File.CreateText(PFAD1);
      sw.WriteLine("File-Demo");
      sw.Close();

      File.Copy(PFAD1, PFAD2, true);

      if (File.Exists(PFAD3))
         File.Delete(PFAD3);

      File.Move(PFAD1, PFAD3);
```

```
      File.SetCreationTime(PFAD3, new DateTime(2005, 12, 29, 22, 55, 44));
      File.SetLastWriteTime(PFAD3, new DateTime(2005, 12, 29, 22, 55, 44));

      FileInfo fi = new FileInfo(PFAD3);
      Console.WriteLine("Die Datei          {0} wurde\n  erstellt:         {1}"+
         "\n  zuletzt geändert: {2}", fi.Name, fi.CreationTime, fi.LastWriteTime);

      fi.Delete();
   }
}
```

Wie in Abschnitt 3.3.9.5 über die Syntax von Zeichenkettenliteralen besprochen, wird mit dem Präfix „@" vor einem Zeichenkettenliteral die Auswertung von Escape-Sequenzen abgeschaltet, so dass die in Windows-Pfadnamen üblichen Rückwärtsschrägstrich nicht mehr durch Verdoppeln von ihrer Sonderfunktion befreit werden müssen.

Anschließend werden wichtige Methoden 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, so dass die Datei anschließend mit **FileMode.Create** geöffnet ist. Die Anweisung
  ```
  FileStream fs = File.Create(PFAD1)
  ```
  ist äquivalent mit
  ```
  FileStream fs = new FileStream(PFAD1, FileMode.Create)
  ```
- **public static StreamWriter CreateText(String** *pfadname***)**
  Die **File**-Methode **CreateText()** erzeugt eine Datei und ein **StreamWriter**–Objekt (mit UTF8-Kodierung). Es entsteht auch das vermittelnde **FileStream**-Objekt, und die Datei ist anschließend mit **FileMode.Create** geöffnet. Die Anweisung
  ```
  StreamWriter sw = File.CreateText(PFAD1)
  ```
  ist äquivalent mit
  ```
  StreamWriter sw = new StreamWriter(PFAD1)
  ```
- **public static bool Exists(String** *pfadname***)**
  Mit **File.Exists()** überprüft man die Existenz einer Datei.
- **public static void Delete(String** *pfadname***)**
  Mit der **File**-Methode **Delete()** kann man eine Datei löschen.
- **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 SetLastWriteTime(String** *pfadname***, DateTime** *letzteÄnderung***)**
  Mit der 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). Exemplarisch wird im Beispiel das Löschen einer Datei per **FileInfo**-Objekt vorgeführt.

Wie die Ausgabe des Programms zeigt, lassen sich wichtige Dateieigenschaften leicht fälschen:

```
Die Datei           nn.txt wurde
  erstellt:         29.12.2005 22:55:44
  zuletzt geändert: 29.12.2005 22:55:44
```

12.3.2 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\";

    Directory.SetCurrentDirectory(DIR1);
    Directory.CreateDirectory(DIR2);

    DirectoryInfo di = new DirectoryInfo(".");
    FileInfo[] fia = di.GetFiles("*.txt");
    Console.WriteLine("txt-Dateien in {0}\n", Directory.GetCurrentDirectory());
    Console.WriteLine("{0, 20} {1, 20}", "Name", "Letzte Änderung");
    foreach (FileInfo fi in fia)
        Console.WriteLine("{0, 20} {1, 20}", fi.Name, fi.LastWriteTime);

    DirectoryInfo[] dia = di.GetDirectories();
    Console.WriteLine("\n\nOrdner in {0}\n", 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.Delete(DIR2, true);
  }
}
```

Wichtige statische Methoden der Klasse **Directory**:

- **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.
- **public static bool Exists(String** *pfadname***)**
  Mit **Exists()** überprüft man die Existenz eines Ordners.
- **public static DirectoryInfo CreateDirectory(String** *pfadname***)**
  public static void Delete(String *pfadname***)**
  Zum Erzeugen bzw. Löschen eines Ordners stehen die Methoden **CreateDirectory()** bzw. **Delete()** bereit. Mit der angegebenen **Delete()** - Überladung lässt sich nur ein *leerer* Ordner löschen. Bei einer alternativen Überladung mit Parameter vom Typ **bool** kann man auch ein rekursives Löschen von Unterverzeichnissen und Dateien erzwingen.

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***)**
  Bei Aufruf seiner Instanzmethode **GetFiles()** liefert ein **DirectoryInfo**-Objekt einen Array mit **FileInfo**-Objekten zu allen Dateien im Ordner mit passendem Namen. Erlaubte Jokerzeichen im Dateiauswahlfilter:

Jokerzeichen	Bedeutung
?	ersetzt genau ein beliebiges Zeichen
*	steht für eine beliebige (eventuell) Zeichenfolge

- **public DirectoryInfo[] GetDirectories(String** *filter***)**
  Analog liefert die Instanzmethode **GetDirectories()** einen Array mit **DirectoryInfo**-Objekten zu den Unterordnern.

12.3.3 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 (Extension **.txt**) in dem per Befehlszeilenargument angegebenen Ordner die Änderungen beim Datum des letzten Zugriffs und beim Namen:

```
using System;
using System.IO;

public class TxtWatcher {
   static void Main(String[] args) {
      FileSystemWatcher watcher = new FileSystemWatcher(args[0]);

      // Zu Überwachen: Ändern und Umbenennen von Dateien
      watcher.NotifyFilter = NotifyFilters.LastWrite | NotifyFilters.FileName;
      // Filter für Dateinamen
      watcher.Filter = "*.txt";

      // Ereignisbehandlungsroutinen registrieren
      watcher.Changed += new FileSystemEventHandler(FsoChanged);
      watcher.Created += new FileSystemEventHandler(FsoChanged);
      watcher.Deleted += new FileSystemEventHandler(FsoChanged);
      watcher.Renamed += new RenamedEventHandler(FsoRenamed);

      // 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');
   }

   // Ereignisroutinen implementieren
   static void FsoChanged(Object source, FileSystemEventArgs e) {
      Console.WriteLine("Datei: " +  e.Name + " " + e.ChangeType);
   }

   static void FsoRenamed(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' + Enter

Überwachter Ordner: U:\Eigene Dateien\C#\EA

Datei Neu Textdokument.txt Created
Datei Neu Textdokument.txt umbenannt in neu.txt
Datei neu.txt Changed
Datei neu.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.

12.4 Übungsaufgaben zu Kapitel 12

1) Erstellen Sie bitte 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

Auf obige Daten sollte Ihr Programm ungefähr so reagieren:

```
Mittelwertsberechnung für die Datei daten.txt

Warnung: Token 2 in Zeile 6 ist keine Zahl.

           Variable          Mittelwert          Valide Werte
                  1               7,833                     6
                  2               4,000                     5
                  3             282,500                     6
```

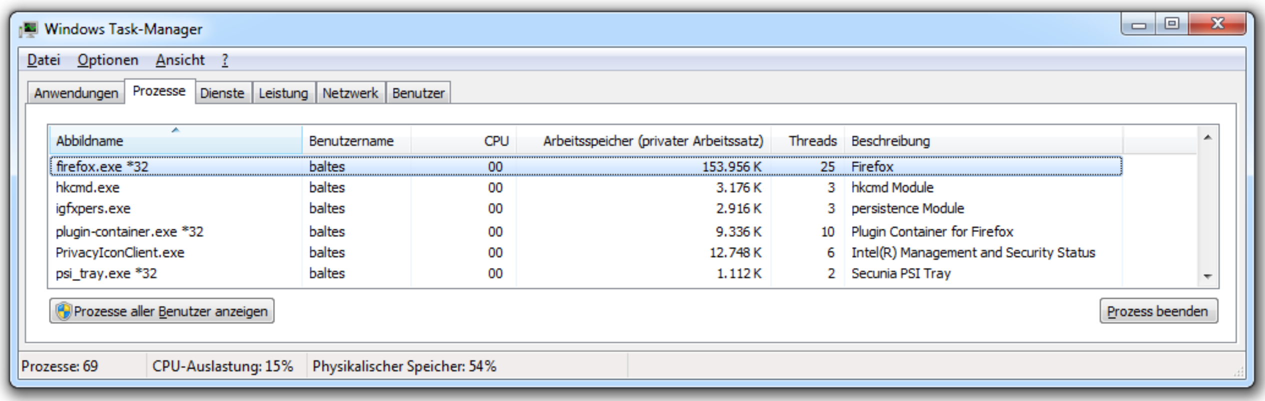
2) Wie kann man den Quellcode des folgenden Programms vereinfachen und dabei auch noch die Laufzeit erheblich reduzieren?

```
using System;
using System.IO;

class AutoFlushDemo {
   static void Main() {
      long zeit = DateTime.Now.Ticks;;
      StreamWriter sw = new StreamWriter("demo.txt");
      sw.AutoFlush = true;
      for (int i = 1; i < 30000; i++) {
         sw.WriteLine(i);
      }
      sw.Close();
      Console.WriteLine("Zeit: "+((DateTime.Now.Ticks-zeit)/1.0e4) + " Millisek.");
   }
}
```

13 Multithreading

Wir sind längst daran gewöhnt, dass moderne Betriebssysteme mehrere Programme (Prozesse) parallel betreiben können, so dass 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. Sofern nur *ein* Prozessor vorhanden ist, der den einzelnen Programmen bzw. Prozessen reihum vom Betriebssystem zur Verfügung gestellt wird, reduziert sich zwar die Ausführungsgeschwindigkeit jedes Programms im Vergleich zum Solobetrieb, 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 ein Blick in die Prozessliste mit dem Task-Manager zeigt, sind z.B. bei einer typischen Firefox-Sitzung ca. 20 Threads aktiv, wobei die Anzahl ständig schwankt, z.B.:



Bei einer GUI-Anwendung sorgt die Multithreading-Technik generell dafür, dass die Bedienoberfläche auch dann noch 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. 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 werden, um die CPU-Kerne auszulasten und Zeit zu sparen. Mittlerweile (2011) sind 4 Kerne guter Standard und dank Intels Hyper-Threading - Technologie sieht das Betriebssystem bei einer Quad-Core - CPU in der Regel sogar 8 logische Kerne. Multi-Core - CPUs erhöhen den Druck auf die Software-Entwickler, per Multithreading für gut skalierende Anwendungen zu sorgen, die auf einem Quad-Core - Rechner deutlich schneller als auf einem Single-Core - Rechner laufen.

Beim Multithreading ist allerdings eine sorgfältige Einsatzplanung erforderlich, denn:

- Thread-Wechsel sind mit einem gewissen Zeitaufwand verbunden und sollten daher nicht zu häufig stattfinden.
- Das Laufzeitsystem wird durch die Verwaltung von Threads zusätzlich belastet.
- In der Regel erfordert das Synchronisieren von Threads einige Aufmerksamkeit beim Programmierer (siehe Abschnitt 13.2). Hier kommt es oft zu Fehlern, die aufgrund variabler Ergebnisse schwer zu analysieren sind.

Während jeder *Prozess* einen eigenen Adressraum besitzt, laufen die *Threads* eines Programms im selben Adressraum ab. Sie haben einen gemeinsamen Heap-Speicher, wohingegen jeder Thread als selbständiger Kontrollfluss bzw. Ausführungsfaden aber einen eigenen Stack-Speicher benötigt.

Bei C# ist die Multithreading-Unterstützung in Sprache, Standardbibliothek (siehe Namensraum **System.Threading**) 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.

Wir erarbeiten uns in diesem Kapitel zunächst ein Multithreading-Basiswissen durch den Einsatz von dedizierten, für einen bestimmten Zweck erstellten Threads.

In der Praxis geht es darum, mit möglichst wenigen Threads eine gute Performanz zu erzielen, wobei im .NET - Framework traditionell das *Asynchronous Programming Model* (APM) eine wichtige Rolle spielt. Statt für eine konkrete Aufgabe (z.B. Bedienung eines Webzugriffs) jeweils einen neuen Thread zeitaufwändig zu erzeugen und anschließend wieder zu zerstören, kommt hier ein Pool von Arbeits-Threads zum Einsatz. Eingehende Aufträge werden einem freien Thread zugeteilt oder in eine Warteschlange gestellt.

Das APM ist bei vielen -NET - Klassen implementiert, und wir müssen es zu nutzen wissen. Speziell bei der asynchronen Ein- und Ausgabe (Dateisystem, Netzwerk, Datenbankzugriff) wird es in absehbarer Zeit eine Standardlösung bleiben. Seit .NET 4.0 steht zur performanten und skalierbaren Nutzung von Mehrkernsystemen die Task Parallel Library (TPL) zur Verfügung, auf die sich eventuell die Weiterentwicklung der Multithreading-Technik konzentrieren wird.

Über weitere Multithreading - Optionen in C# informiert z.B. eine sehr komplette und aktuelle WWW-Seite von Joe Albahari (2010).[1]

13.1 Threads erzeugen

13.1.1 Die Klasse Thread

Ein Thread wird in C# über ein Objekt der gleichnamigen Klasse aus dem Namensraum **System.Threading** realisiert. Das gilt sowohl für die dedizierten (zur Bewältigung einer speziellen Aufgabe erstellten) Threads, als auch für die Pool-Threads.

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 dem **Thread**-Konstruktor übergeben wird, z.B.:

```
Thread pt = new Thread(new ThreadStart(pro.Run));
```

Man kann sich das explizite Notieren des Delegaten-Konstruktors sparen:

```
Thread pt = new Thread(pro.Run);
```

[1] http://www.albahari.com/threading/

13.1.2 Produzent - Konsument - Beispiel

Diese Thread-Kreation stammt aus einem „betriebswirtschaftlichen“ Beispielprogramm mit einem Objekt aus einer Klasse `Produzent` und einem Objekt aus einer Klasse `Konsument`, die auf einen Lagerbestand einwirken, der von einem Objekt der Klasse `Lager` gehütet wird. Produzent und Konsument entfalten ihre Tätigkeit jeweils im Rahmen einer Methode namens `Run()`, die in einem eigenen Thread läuft (siehe unten).

13.1.2.1 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:

- `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.

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 `MANZ`):

```
using System;
using System.Threading;

public class Lager {
 int bilanz;
 int anz;
 const int MANZ = 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 < MANZ) {
   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 < MANZ) {
      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 < 40000; 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();
    }
   }
```

13.1.2.2 Die Klassen `Produzent` und `Konsument`

Produzent und Konsument kommen mit einer recht 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 als Konstruktor-Parameter eine entsprechende Referenz.

Neben dem Konstruktor ist jeweils nur eine Methode namens `Run()` vorhanden, welche den Delegatentyp **ThreadStart** erfüllt und als Startmethode für den Produzenten- bzw. Konsumenten-Thread dient. 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. Zwischen zwei Aufträgen machen die `Run()`-Methoden eine Pause von zufallsabhängiger Länge, indem Sie die statische **Thread**-Methode **Sleep**() aufrufen. Diese befördert den Thread vom Zustand **Running** in den Zustand **WaitSleepJoin** (siehe unten). 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 nicht unbedingt als erster beim Lager an kommt.

13.1.3 Sekundäre 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**)
  ```
  Konsument kon = new Konsument(lager);
  ```

Schließlich erhalten die Threads einen Namen und werden gestartet:

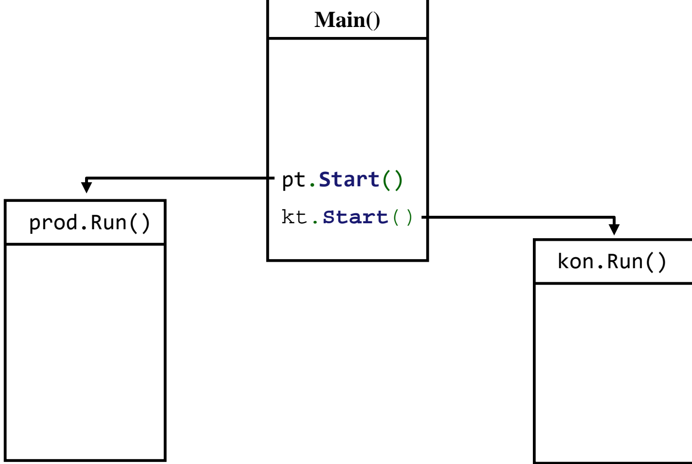
```
pt.Name = "Produzent";
kt.Name = "Konsument";
pt.Start();
kt.Start();
```

Der **Thread** `pt` startet mit der Ausführung der `Run()`-Methode durch das Objekt `pro` aus der Klasse `Produzent`, und der **Thread** `kt` startet mit der Ausführung der `Run()`-Methode durch das Objekt `kon` aus der Klasse `Konsument`.

Beide Threads laufen im *Vordergrund* ab, und ein Prozess gilt als aktiv, solange er mindestens einen aktiven Vordergrund-Thread besitzt. Ist ein Thread über seine Eigenschaft **IsBackground** als **Hintergrund-Thread** markiert, kann er einen Prozess nicht aufrecht erhalten, sondern wird ggf. automatisch mit dem letzten Vordergrund-Thread beendet.

Unmittelbar vor dem Ende der **Main()**-Methode sind **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 (`prod.Run()` bzw. `kon.Run()`), sofern sie nicht zuvor abgebrochen werden (siehe unten).



Die beiden Aufrufe der **Thread**-Methode **Start()** kehren praktisch unmittelbar zurück, und anschließend endet mit der **Main()**-Methode auch der primäre Thread. Die beiden sekundären Threads leben weiter bis zum Ende ihrer jeweiligen Startmethode, und das Programm endet mit seinem letzten Vordergrund-Thread.

Dass beim Ende der **Main()**-Methode die einzigen Referenzen auf die **Thread**-Objekte (`pt` und `kt`) verschwinden, spielt keine Rolle.

Ein Thread endet, wenn seine Startmethode abgearbeitet ist. Er befindet sich dann im Zustand **Stopped** und kann *nicht* erneut gestartet werden. Im Beispiel passiert dies, sobald eine der `Run()`-Methoden zu Beginn eines **while** - Schleifendurchgangs ein geschlossenes Lager festgestellt, wenn also der Methodenaufruf `lager.Ergaenze()` bzw. `lager.Liefere()` zum Rückgabewert **false** führt.

Der primäre Thread des Programms ist zu diesem Zeitpunkt ebenfalls bereits Geschichte, weil er mit der `Lager`-Methode **Main()** seine Tätigkeit einstellt. Folglich endet das Programm, wenn Produzenten- und Konsumenten-Thread sich verabschiedet haben.

13.1.4 Klassen aus dem Anwendungsbereich und aus der Informatik

Das Beispiel mit seinem recht reichhaltigen Objekt-Ensemble demonstriert, 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 Aufgaben dienen. Man kann einen Thread als *Ausführungsfaden* oder *Aktivitätsträger*[2] bezeichnen,

---

[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 ebenfalls beendet.

[2] http://de.wikipedia.org/wiki/Thread_(Informatik)

wobei der zweite Ausdruck eine nette Veranschaulichung liefert. Wir haben früher gelegentlich von der *objektorientierten Bühne* gesprochen und können nun zur Veranschaulichung von Multithreading zum Plural übergehen, in einem sekundären *Aktivitätsträger* also eine zusätzliche Bühne mit eigenem Handlungsablauf sehen.

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, so dass in einem Ausführungsfaden beliebig viele Akteure tätig werden können. Wir reden zwar reden 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 einzelner Akteur (z.B. ein Objekt) in mehreren Threads arbeiten, wenn er entsprechende Botschaften erhält. Im Beispiel kommt das `Lager`-Objekt sowohl im Produzenten- als auch im Konsumenten-Thread zum Einsatz: Die Methoden `Ergaenze()` und `Liefere()` erhöhen oder reduzieren den Lagerbestand, aktualisieren die Anzahl der Lagerveränderungen und protokollieren jede Maßnahme. Dazu besorgen sie sich mit der stationären **Thread**-Methode **CurrentThread()** eine Referenz auf den aktuell ausgeführten Thread 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.

13.2 Threads synchronisieren

In einem typischen Ablaufprotokoll des Produzenten - Konsumenten - Programms zeigen sich einige Ungereimtheiten, verursacht durch das unkoordinierte Agieren der beiden 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 Synchronisation der Zugriffe erforderlich.

13.2.1 Die lock-Anweisung

Am Anfang des oben wiedergegebenen Ablaufprotokolls stehen zwei „wirre“ Einträge, die folgendermaßen durch eine so genannte **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 das Laufzeitsystem 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.

Offenbar muss verhindert werden, dass während eines Lagerzugriffs ein Thread-Wechsel stattfindet. Dies ist in C# leicht zu realisieren, indem per **lock**-Anweisung der kritische Anweisungsblock mit einer Sperre versehen wird, z.B.:

```
Object lockObject = new Object();
  . . .
public bool Ergaenze(int add) {
 lock(lockObject) {
  if (anz < MANZ) {
   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 (2006, S. 642ff) rät aber zur Verwendung eines *privaten* Member-Objekts, weil ein öffentlich bekanntes Sperrobjekt zum Blockieren der Anwendung durch eine unbefugte oder ungeschickte Verwendung führen kann. Im Beispiel wird dieser Vorschlag realisiert.

Im Beispiel muss auch der kritische Bereich in der Methode `Liefere()` durch *dasselbe* Objekt gesperrt werden. Beim Betreten eines geschützten Bereichs setzt ein Thread per **lock** die Sperre. Man kann sich vorstellen, dass er den einzigen Schlüssel für die von einem Objekt geschützten Bereiche erwirbt. Jedem anderen Thread wird der Zutritt verwehrt, und er muss warten. Beim Verlassen eines geschützten Bereichs wird die Sperre aufgehoben, und ggf. kann ein wartender Thread seine Arbeit fortsetzen. Ein Anweisungsblock mit **lock**-reguliertem Zugang wird auch als *synchronisiert* bezeichnet.

Um andere Threads möglichst wenig zu behindern, muss ein Sperrobjekt so schnell wie möglich wieder frei gegeben werden.

Aufgrund der Thread-Synchronisation per Lock-Objekt produziert unser Beispielprogramm keine wirren Protokolleinträge mehr, doch es kommt nach wie vor zu einem negativen Lagerzustand:

```
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!
```

Wenn eine komplette Methode in den geschützten Bereich einbezogen werden soll, der (bei Instanzmethoden) vom handelnden Objekt (anzusprechen mit **this**) bzw. (bei statischen Methoden) vom **Type**-Objekt zur Klasse (anzusprechen mit **typeof**(*Klasse*)) überwacht wird, kann der Methode das **MethodImplAttribute** vorangestellt werden, z.B.:[1]

[1] Vorbild für diese Sperrtechnik war wohl der Modifikator **synchronized** für Methoden in der Programmiersprache Java.

```
[MethodImplAttribute(MethodImplOptions.Synchronized)]
public bool Ergaenze(int add) {
 if (anz < MANZ) {
  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 dabei ein öffentliches Sperrobjekt in Kauf nehmen, das Blockaden durch ungeschickte oder böswillige Software ermöglicht (siehe oben).

13.2.2 Die Klasse Monitor

Für den synchronisierten Zugriff von Threads auf geschützte Bereiche sorgt letztlich die Klasse **Monitor** aus dem Namensraum **System.Threading**. Sie eignet sich weder zum Ableiten neuer Klassen noch zum Erzeugen von Objekten, spielt aber neben der Klasse **Thread** eine zentrale Rolle bei Multithreading-Anwendungen im .NET - Framework. Die **lock**-Anweisung

```
lock(sperre) {
   ...
}
```

wird vom Compiler umgesetzt in:

```
Monitor.Enter(sperre);
try {
   ...
} finally{
   Monitor.Exit(sperre);
}
```

Hier kommen zwei statische **Monitor**-Methoden zum Einsatz:

- **Enter()**
  Die Sperre wird gesetzt.
- **Exit()**
  Die Sperre wird aufgehoben. Durch den **Exit()**-Aufruf im **finally**-Block ist sichergestellt, dass der kritische Block auch nach einem Ausnahmefehler verlassen wird.

Um einer Blockade vorzubeugen, kann sich ein Thread mit der **Monitor**-Methode **TryEnter()** um den exklusiven Zugang zum kritischen Block bewerben. Diese Methode endet auf jeden Fall sofort und informiert mit einem Rückgabewert vom Typ **bool** darüber, ob die Berechtigung erteilt worden ist.

13.2.3 Die Klasse Mutex

Die Klasse **Mutex** (engl. *mutual exclusion*) erbringt ähnliche Leistungen wie die Klasse **Monitor** und erlaubt auch eine Thread-Synchronisation über Prozessgrenzen hinweg (z.B. zwischen verschiedenen Programmen). Diese Flexibilität ist allerdings mit einem erheblichen Zeitaufwand verbunden, was speziell bei häufigem Thread-Wechsel von Bedeutung ist. Wir benötigen nur die prozessinterne Thread-Synchronisation und verzichten daher auf den Einsatz der Klasse **Mutex**.

13.2.4 Koordination per Wait() und Pulse()

Mit Hilfe der statischen **Monitor**-Methoden **Wait()** und **Pulse()** können in unserem betriebswirtschaftlichen Beispiel negative Lagerbestände verhindert werden. Trifft eine Konsumenten-Anfrage auf einen unzureichenden Lagerbestand, dann wird der zugehörige Thread mit der Methode **Wait()** in den Zustand **WaitSleepJoin** versetzt:

```
public bool Liefere(int sub) {
 lock (lockObject) {
  if (anz < MANZ) {
   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);
   return true;
  } else {
   Console.WriteLine("\nLieber " + Thread.CurrentThread.Name +
    ", es ist Feierabend!");
   return false;
  }
 }
}
```

Mit dem **Wait()**-Aufruf wird das exklusive Zutrittsrecht für den synchronisierten Block zurückgegeben, so dass im Beispiel der Produzenten-Thread zum Zug kommt und für Nachschub sorgen kann. Als Parameter wird im **Wait()**-Aufruf das synchronisierende Objekt angegeben.

Um eine erfolgreiche Kooperation zu gewährleisten, muss der Produzenten-Thread nach jeder Lieferung die **Monitor**-Methode **Pulse()** aufrufen, um den Konsumenten-Thread zu reaktivieren, d.h. in den Zustand **Running** zu versetzen:

```
public bool Ergaenze(int add) {
 lock (lockObject) {
  if (anz < MANZ) {
   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;
  }
 }
}
```

Als Parameter wird im **Pulse()**-Aufruf das synchronisierende Objekt angegeben, das die Warteschlange verwaltet. Der reaktivierte Konsumenten-Thread bewirbt sich wieder um Prozessorzeit und Lagerzugangsberechtigung. Weil keinesfalls sicher ist, dass Konsument nach der Reaktivierung

einen ausreichenden Vorrat antrifft, findet sein **Wait()** - Aufruf in einer **while**-Schleife mit einleitender Bedingungsprüfung statt.

Nun produziert das Beispielprogramm nur noch realistische Lagerprotokolle, z.B.:

```
Das Lager ist offen (Bestand: 100)

Nr.  1:  Produzent ergänzt   29 um 02:26:27 Uhr. Stand: 129
Nr.  2:  Konsument entnimmt  82 um 02:26:27 Uhr. Stand: 47
Nr.  3:  Produzent ergänzt   51 um 02:26:29 Uhr. Stand: 98
Nr.  4:  Konsument entnimmt  21 um 02:26:29 Uhr. Stand: 77
Nr.  5:  Produzent ergänzt   70 um 02:26:32 Uhr. Stand: 147
Nr.  6:  Konsument entnimmt  15 um 02:26:33 Uhr. Stand: 132
Nr.  7:  Produzent ergänzt   40 um 02:26:34 Uhr. Stand: 172
Nr.  8:  Konsument entnimmt  85 um 02:26:35 Uhr. Stand: 87
Nr.  9:  Konsument entnimmt  27 um 02:26:38 Uhr. Stand: 60
Nr. 10:  Produzent ergänzt   15 um 02:26:38 Uhr. Stand: 75
!!!!!!!  Konsument muss warten: Keine  81 Einheiten vorhanden um 02:26:39 Uhr.
Nr. 11:  Produzent ergänzt    7 um 02:26:41 Uhr. Stand: 82
Nr. 12:  Konsument entnimmt  81 um 02:26:41 Uhr. Stand: 1
!!!!!!!  Konsument muss warten: Keine   5 Einheiten vorhanden um 02:26:42 Uhr.
Nr. 13:  Produzent ergänzt   37 um 02:26:43 Uhr. Stand: 38
Nr. 14:  Konsument entnimmt   5 um 02:26:43 Uhr. Stand: 33
!!!!!!!  Konsument muss warten: Keine  43 Einheiten vorhanden um 02:26:45 Uhr.
Nr. 15:  Produzent ergänzt   73 um 02:26:47 Uhr. Stand: 106
Nr. 16:  Konsument entnimmt  43 um 02:26:47 Uhr. Stand: 63
!!!!!!!  Konsument muss warten: Keine  75 Einheiten vorhanden um 02:26:48 Uhr.
Nr. 17:  Produzent ergänzt   33 um 02:26:50 Uhr. Stand: 96
Nr. 18:  Konsument entnimmt  75 um 02:26:50 Uhr. Stand: 21
!!!!!!!  Konsument muss warten: Keine  78 Einheiten vorhanden um 02:26:51 Uhr.
Nr. 19:  Produzent ergänzt   75 um 02:26:52 Uhr. Stand: 96
Nr. 20:  Konsument entnimmt  78 um 02:26:52 Uhr. Stand: 18

Lieber Konsument, es ist Feierabend!

Lieber Produzent, es ist Feierabend!
```

An Stelle der Methode **Pulse()**, die den Thread mit dem ältesten **Wait()**-Aufruf anspricht, ist oft die Methode **PulseAll()** sinnvoller, die alle wartenden Threads weckt.

Die **Monitor**-Methoden **Wait()**, **Pulse()** und **PulseAll()** dürfen nur in einem synchronisierten Block aufgerufen werden.

13.2.5 Andere Verfahren zur Thread-Koordination

An Stelle der **Monitor**-Methoden **Wait()**, **Pulse()** und **PulseAll()**, die nur in einem synchronisierten Block aufgerufen werden dürfen und daher unmittelbar im Anschluss an die Techniken zur Sicherung der Zugriffsexklusivität dargestellt wurden, kommen oft einfachere Verfahren zur Koordination von Threads in Frage.

13.2.5.1 Ein Schläfchen in Ehren

In diesem Zusammenhang kann man auch die statische **Thread**-Methode **Sleep()** noch einmal erwähnen, mit der sich der aufrufende Thread für eine per Parameter bestimmte Anzahl von Millisekunden in den Zustand **WaitSleepJoin** versetzt und den restlichen Threads das Feld überlässt, z.B.:

```
Thread.Sleep(1000 + rand.Next(3000));
```

13.2.5.2 Weck mich, wenn Du fertig bist

Will ein Thread `t1` in den Wartezustand wechseln, bis der Thread `t2` seine Tätigkeit beendet hat, kann er diesen Plan durch einen Aufruf der **Thread**-Methode **Join()** realisieren, z.B.:

Quellcode:

```
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\n");
   }

   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 beenden mit Enter");
      Console.Read();
   }

   static void Main() {
      t1 = new Thread(new ThreadStart(M1));
      t2 = new Thread(new ThreadStart(M2));
      t1.Start(); t2.Start();
   }
}
```

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 beenden mit Enter
```

13.2.5.3 Signalisierungsobjekte

Über ein Signalisierungsobjekt der Klasse **AutoResetEvent** kann sich ein Thread in den einstweiligen Ruhezustand versetzen, bis ein anderer Thread das Signal „auf Grün“ stellt. Beide Threads sprechen dasselbe **AutoResetEvent**–Objekt an, wobei der erste Thread seinen Weckauftrag über die Methode **WaitOne()** erteilt und der zweite Thread über die Methode **Set()** einen wartenden Thread reaktiviert. Wie bei einer Drehkreuztür mit kostenpflichtigem Einzeldurchlass kann pro **Set()**-Aufruf nur *ein* Thread reaktiviert werden, wobei das gesetzte Signal automatisch verbraucht wird (Auto-Reset, siehe Klassenname).

Soll stattdessen bei der Reaktivierung eine Tür für *alle* auf das Signal wartenden Threads geöffnet werden, verwendet man statt der Klasse **AutoResetEvent** das Gegenstück **ManualResetEvent**. Bei einem Signalisierungsobjekt dieser Klasse bleibt ein gesetztes Signal erhalten, bis es über die Methode **Reset()** storniert wird.

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); // Das 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 gewckt 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 folgende Ausgaben:

```
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.
```

Per Konstruktorparameter wird ein **AutoResetEvent**–Objekt mit initial abgeschalteten Signal erzeugt:

```
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, 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 bereits gesetzt, bleibt ein weiterer **Set**()-Aufruf allerdings folgenlos.

Beim einem Aufruf der Methode **WaitAny()** bzw. **WaitAll()** 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.

Es ist zu beachten, dass ein Thread beim **WaitOne()**-, **WaitAny()-** oder **WaitAll**()-Aufruf in seinem Besitz befindliche Sperrobjekte *nicht* freigibt. Beim Aufruf der **Monitor**-Methode **Wait()** findet eine solche Freigabe jedoch statt, was im Produzenten-Konsumenten - Beispiel von Abschnitt 13.2.4 von essentieller Bedeutung ist.

Die Klassen **AutoResetEvent** und **ManualResetEvent** stammen (wie die Klasse **Mutex**, siehe Abschnitt 13.2.3) von der Klasse **System.Threading.WaitHandle**, verwenden Ressourcen des Betriebssystems und erlauben eine Thread-Koordination über Prozessgrenzen hinweg. Dies hat aber einen erheblichen Zeitaufwand beim Thread-Wechsel zur Folge, sodass ein Einsatz beim kurz getakteten Start-Stopp – Betrieb *nicht* ratsam ist.

Seit der .NET – Version 4.0 bietet die Klasse **ManualResetEventSlim**, die von **System.Object** abstammt, eine zur Klasse **ManualResetEvent** analoge Funktionalität. Weil auf Betriebsystem-Ressourcen und die Option zur Überschreitung von Prozessgrenzen verzichtet wird, ist ein sehr flotter Thread-Wechsel möglich.

Über ein Objekt der Klasse **CountdownEvent**, kann sich ein Thread reaktivieren lassen, wenn eine Signal *k* mal gesetzt worden ist (siehe Übungsaufgabe in Abschnitt 13.10).

13.2.5.4 Die Klasse ReaderWriterLock

Wenn die zu synchronisierenden Threads auf die gemeinsame Ressource meist lesend und selten schreibend zugreifen, bietet die Klasse **ReaderWriterLock** eine performantere Alternative zur Klasse **Monitor**:

- Nachdem ein schreibwilliger Thread die Methode **AquireWriterLock()** erfolgreich aufgerufen hat, ist kein weiterer Zugriff möglich, bis der die Schreibberechtigung mit einem Aufruf der Methode **ReleaseWriterLock()** zurück gegeben wird.
- Nachdem ein lesewilliger Thread die Methode **AquireReaderLock()** erfolgreich aufgerufen hat, ist kein Schreibzugriff möglich, während auch andere Threads eine Leseberechtigung erwerben können. Hat jeder Thread mit Leseberechtigung die Methode **ReleaseReader-Lock()** aufgerufen, kann wieder eine Schreibberechtigung erteilt werden.

Läuft die im Schreib- oder Leseantrag anzugebende maximale Wartezeit ab, wird der Methodenaufruf mit einer **ApplicationException** abgebrochen.

Im folgenden Beispiel greift ein Thread schreibend auf eine Ressource (einen **double**-Wert) zu, während 3 andere Threads nur lesend zugreifen:

```
using System;
using System.Threading;

class ReaderWriterLockDemo {
   static Thread rt;
   static double dwert = 13.0;
   static ReaderWriterLock rwl = new ReaderWriterLock();

   static void WM() {
      Console.WriteLine("*** Der Schreib-Thread startet.");
      for (int i = 1; i <= 3; i++) {
         rwl.AcquireWriterLock(1000);
            dwert++;
         Console.WriteLine("*** Schreibzugriff " + i + ". Neuer Wert: " + dwert);
         rwl.ReleaseWriterLock();
         Thread.Sleep(1000);
      }
      Console.WriteLine("*** Der Schreib-Thread endet.");
   }

   static void RM() {
      Console.WriteLine(Thread.CurrentThread.Name + " startet.");
      for (int i = 1; i <= 3; i++) {
         rwl.AcquireReaderLock(1000);
            Console.WriteLine(Thread.CurrentThread.Name + " ermittelt Wert " + dwert);
         rwl.ReleaseReaderLock();
         Thread.Sleep(1000);
      }
      Console.WriteLine(Thread.CurrentThread.Name + " endet.");
   }

   static void Main() {
      new Thread(WM).Start();
      for (int i = 0; i < 3; i++) {
         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:

```
*** Der Schreib-Thread startet.
*** Schreibzugriff 1. Neuer Wert: 14
RT0 startet.
RT0 ermittelt Wert 14
RT1 startet.
RT1 ermittelt Wert 14
RT2 startet.
RT2 ermittelt Wert 14
*** Schreibzugriff 2. Neuer Wert: 15
RT0 ermittelt Wert 15
RT1 ermittelt Wert 15
RT2 ermittelt Wert 15
*** Schreibzugriff 3. Neuer Wert: 16
RT0 ermittelt Wert 16
RT1 ermittelt Wert 16
RT2 ermittelt Wert 16
*** Der Schreib-Thread endet.
RT0 endet.
RT1 endet.
RT2 endet.
```

13.3 Threads stoppen

Ein Thread endet „auf natürlichem Weise“ mit der zugrunde liegenden Startmethode. Um ihn früher zu stoppen, kann man die **Thread**–Methode **Abort()** aufrufen. Diese befördert den Thread in den Zustand **AbortRequested** und löst eine **ThreadAbortException** aus, so dass die betroffenen Methoden per Ausnahmebehandlung für einen sinnvollen Abgang sorgen können. Am Ende eines entsprechenden **catch**-Blocks wird die Ausnahme automatisch erneut ausgelöst, so dass im Fall einer Aufrufverschachtelung auch vorgeordnete Methoden Gelegenheit zu Terminierungsmaßnahmen erhalten.

Als Beispiel soll eine kundenunfreundliche Variante des Produzenten-Lager-Konsumenten - Programms dienen: Der Lagerverwalter terminiert den Konsumenten-Thread, sobald dieser mit einem Wunsch über den Lagerbestand hinausgeht:

```
public bool Liefere(int sub) {
   lock (lockObject) {
   if (anz < MANZ) {
    if (bilanz < sub) {
     Console.WriteLine("!!!!!!! {1,10} fordert  {2, 3}" +
     " um {3} Uhr und wird abgewiesen.",
     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 Konsumententen-Thread in der Zustand **Stopped**, 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 abgewiesen.
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!
```

Ein Thread im Zustand **AbortRequested** kann aber durchaus die Terminierung vermeiden und in den Zustand **Running** zurückkehren. Dazu muss er lediglich in seiner **ThreadAbortException**-Ausnahmebehandlung die **Thread**-Methode **ResetAbort()** aufrufen, was in der folgenden Variante unseres Beispiels der Konsument 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);
}
```

13.4 Thread-Lebensläufe

In diesem Abschnitt wird zunächst die Vergabe von Arbeitsberechtigungen für konkurrierende Threads behandelt. Dann fassen wir unsere Kenntnisse über die verschiedenen Zustände eines Threads und über Anlässe für Zustandswechsel zusammen.

13.4.1 Scheduling und Prioritäten

Die Zuweisung von Rechenzeit auf den real oder logisch vorhandenen CPU-Kernen an die Threads im Zustand **Running** überlässt die CLR dem Betriebssystem, wo für diese Aufgabe der so genannte **Scheduler** zuständig ist.

Er 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**:

- **Highest**
- **AboveNormal**
- **Normal**
- **BelowNormal**
- **Lowest**

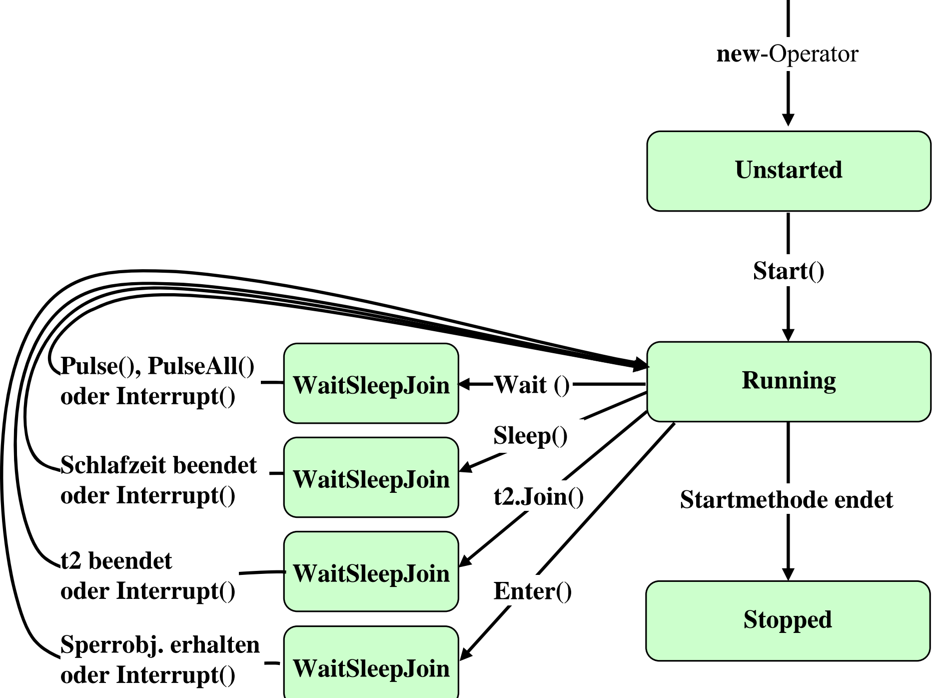
Per Voreinstellung haben Threads die Priorität **Normal**.

Der Scheduler bevorzugt Threads mit höherer Priorität in einem *strengen* Sinn und verwendet bei gleicher Priorität ein **preemtives Zeitscheibenverfahren**. Auf einem *Einprozessor*-System resultiert folgendes Verhalten:

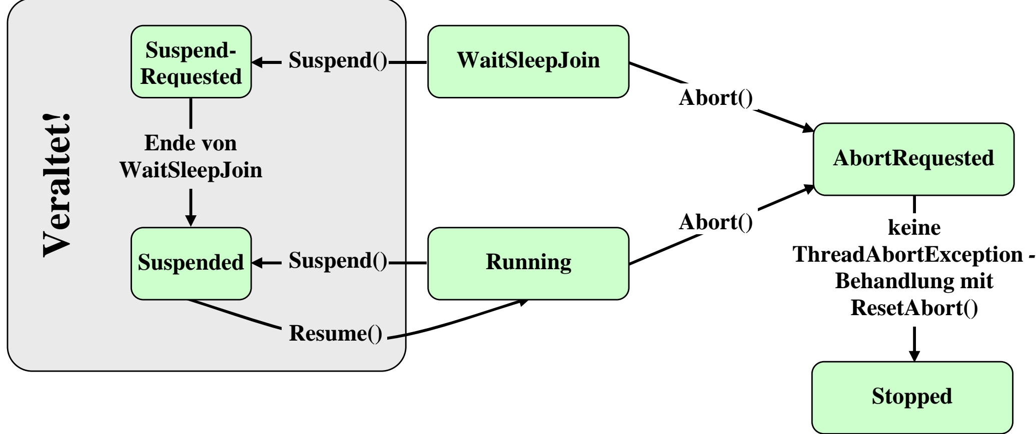
- Ein Thread kann nur dann Zugang zum Prozessor erhalten, wenn *kein* Thread mit höherer Priorität arbeitswillig ist. Ein *Verhungern* (engl. *starvation*) von Threads mit niedriger Priorität, die permanent den Kürzeren ziehen, wird also *nicht* verhindert.
- Die Threads gleicher Priorität werden reihum (*Round-Robin*) jeweils für eine festgelegte Zeitspanne ausgeführt.

13.4.2 Zustände von Threads

In der folgenden Abbildung werden wichtige Thread-Zustände (bezeichnet durch Werte der Enumeration **ThreadState**) und Anlässe für Zustandsübergänge dargestellt:



Der Übersichtlichkeit halber folgt eine separate Abbildung für das Abbrechen bzw. Unterbrechen eines Threads:



Einige in den Abbildungen enthaltene **Thread**-Instanzmethoden wurden bisher noch nicht angesprochen:

- **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 prophylaktisch erfolgen. Begibt sich der angesprochene Thread nie in den Zustand **WaitSleepJoin**, bleibt der **Interrupt()**-Aufruf ohne Folgen. Um auf eine **ThreadInterruptedException** vorbereitet zu sein, müssen **WaitSleepJoin**-einleitende 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 diesem 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 13.5) führen kann.

Weitere Hinweise:

- **Running** ist ein arbeitswilliger Thread auch dann, wenn er gerade auf die Zuteilung eines Prozessors durch das Betriebssystem wartet.
- Gemäß der FCL-Dokumentation[1] zur Enumeration **ThreadState** ist ein Thread auch dann im Zustand **WaitSleepJoin**, wenn er auf ein Sperrobjekt wartet (z.B. nach dem Methodenaufruf **Monitor.Enter()**, siehe Abschnitt 13.2.2) oder wenn er auf ein Synchronisierungsobjekt (z.B. aus der Klasse **AutoResetEvent**) wartet (siehe Abschnitt 13.2.5.3). Manche Autoren sprechen bei einer solchen Lage von einem *blockierten* Thread.
- Einige Thread-Zustände können *gleichzeitig* bestehen. Erhält z.B. ein Thread im Zustand **WaitSleepJoin** einen **Abort()**-Aufruf, bestehen simultan die Zustände **WaitSleepJoin** und **AbortRequested** bis der Thread den Zustand **WaitSleepJoin** verlässt und dann mit der **ThreadAbortException** konfrontiert wird.

13.5 Deadlock

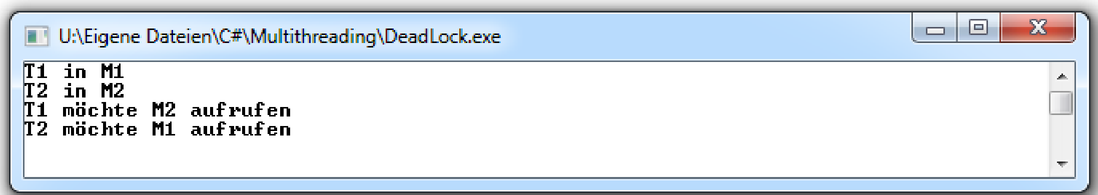
Wer sich beim Einsatz von Sperrobjekten zur Thread-Synchronisation ungeschickt anstellt, kann einen genannten *Deadlock* produzieren, wobei sich Threads gegenseitig blockieren. Im folgenden Beispiel begeben sich die Threads `T1` und `T2` jeweils in einen exklusiven Block, der durch die Objekte `lock1` bzw. `lock2` geschützt ist:

```
using System;
using System.Threading;

class DeadLock {
   static object lock1 = new object();
   static object lock2 = new object();
```

[1] http://msdn.microsoft.com/de-de/library/system.threading.threadstate.aspx

```
      static void M1() {
         lock (lock1) {
            Console.WriteLine(Thread.CurrentThread.Name+" in M1");
            Thread.Sleep(100);
            Console.WriteLine(Thread.CurrentThread.Name+" möchte M2 aufrufen");
            M2();
         }
      }

      static void M2() {
         lock (lock2) {
            Console.WriteLine(Thread.CurrentThread.Name+" in M2");
            Thread.Sleep(100);
            Console.WriteLine(Thread.CurrentThread.Name+" möchte M1 aufrufen");
            M1();
         }
      }

      static void Main() {
         Thread t1 = new Thread(new ThreadStart(M1));
         Thread t2 = new Thread(new ThreadStart(M2));
         t1.Name="T1"; t1.Start();
         t2.Name="T2"; t2.Start();
      }
   }
```

Ein kurzes Schläfchen sorgt dafür, dass beide Threads ihren „eigenen" synchronisierten Block ungestört betreten können. Anschließend versuchen beide, in den jeweiligen fremden Block zu gelangen, und das Programm hängt fest:



Das vorgestellte Problem hat folgende Struktur:

- Thread `T1` besetzt den vom Objekt `lock1` geschützten Block.
- Thread `T2` besetzt den vom Objekt `lock2` geschützten Block.
- 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. Thread `T2` wartet also darauf, dass Thread `T1` diesen Block verlässt.

13.6 Treadpool

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 von .NET-Framework zur Verfügung gestellte Pool von Ein-/Ausgabe - und Arbeits-Threads genutzt werden. Eingehende Aufträge gelangen in eine Warteschlange des Pools und werden vom nächsten freien Thread aus dem „Bereitschaftsteam" übernommen.

Es ist wohl nur selten sinnvoll, mit der **ThreadPool**-Methode **SetMaxThreads()** die Entscheidung der CLR über die angemessene Zahl von Ein-/Ausgabe - bzw. Arbeits-Threads im Pool zu beeinflussen.

Anstelle eines Threadpool - Auftrags ist ein eigener (gem. Abschnitt 13.1 erstellter) Thread z.B. dann erfoderlich, ...

- wenn ein *Vordergrund*-Thread benötigt wird.
  Im Pool sind nur Hintergrund-Threads tätig, die nach dem Ende des letzten Vordergrund-Threads von der CLR automatisch gestoppt werden.
- wenn eine spezielle Thread-Priorität gewünscht ist.
  Alle Pool-Threads haben eine *normale* Priorität (siehe Abschnitt 13.4.1).

Soll eine Methode im Hintergrund durch einen Arbeits-Thread ausgeführt werden, muss sie dem Delegatentyp **WaitCallback**:

public delegate void WaitCallback(Object *state***)**

entsprechen, z.B.

```
static void HintergrundAktion(Object anz) {
   double summe;
   for (int i = 0; i < (int)anz; i++) {
      summe = 0.0;
      for (int j = 0; j < 10000000; j++)
         summe += zzg.Next(100);
      Console.WriteLine("Aktuelle Zufallssumme: " + summe);
   }
   Console.WriteLine("\nDer Arbeits-Thread ist fertig");
}
```

Diese Methode berechnet eine wählbare Anzahl von Summen, jeweils bestehend aus reichlich vielen Zufallszahlen, die von einem **Random**-Objekt erstellt werden.

Zur Übergabe eines **WaitCallback**-Arbeitsauftrags an den Threadpool dient die **ThreadPool**-Methode **QueueUserWorkItem**() mit 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 so festlegen, wie viele Zufallssummen berechnet werden sollen.

In der folgenden **Main()**-Methode wird ein Arbeitsauftrag in die Warteschlange des Threadpools gestellt. Statt im Vordergrund andere Arbeiten auszuführen, legt sich der primäre Thread schlafen:

```
static void Main() {
   bool b = ThreadPool.QueueUserWorkItem(HintergrundAktion, 5);
   Console.WriteLine("Arbeitsauftrag erfolgreich in die "+
      "Threadpool-Warteschlange gestellt: {0}\n",b);
   Console.WriteLine("Der primäre Thread schläft 5 Sekunden lang\n");
   Thread.Sleep(5000);
   Console.WriteLine("\nDer primäre Thread ist aufgewacht.\n");
   Console.ReadLine();
}
```

Ein Ergebnis dieser „Arbeitsteilung“:

```
Arbeitsauftrag erfolgreich in die Threadpool-Warteschlange gestellt: True

Der primäre Thread schläft 5 Sekunden lang

Aktuelle Zufallssumme: 495119495
Aktuelle Zufallssumme: 495009812
Aktuelle Zufallssumme: 494964065
Aktuelle Zufallssumme: 494957046
```

```
Aktuelle Zufallssumme: 494947456

Der Arbeits-Thread ist fertig

Der primäre Thread ist aufgewacht.
```

13.7 Asynchronous Programming Model

In diesem Abschnitt wird das **APM**–Entwurfsmuster (*Asynchronous Programming Model*) vorgestellt, das die Verwendung von Pool-Threads einfacher und flexibler macht. Es bietet u.a. die folgenden Vorteile:

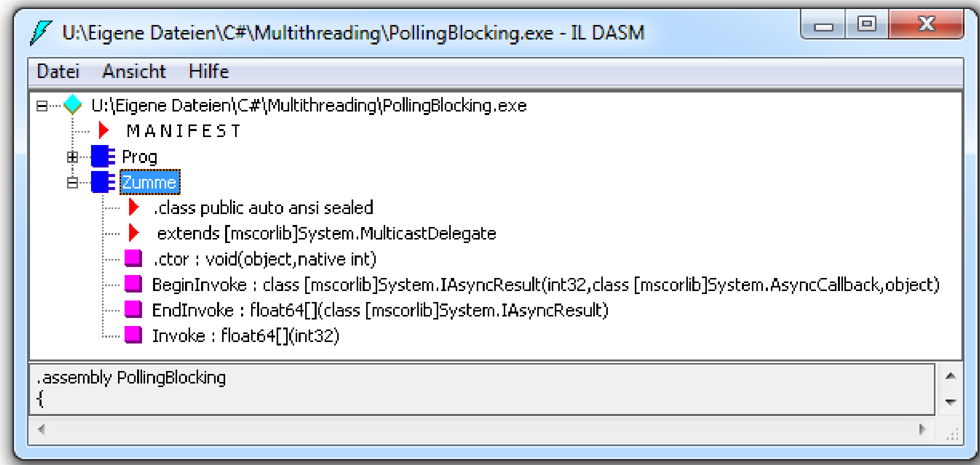
- Statt der Einschränkung auf den Delegatentyp **WaitCallback** können Methoden mit Rückgabe und Parametern per Pool-Thread ausgeführt werden.
- Der initiierende Thread kann sich über den Abschluss der Hintergrundtätigkeit informieren.

13.7.1 Die Delegatenmethoden BeginInvoke() und EndInvoke()

Zu jeder Delegatendefinition, z.B.:

```
public delegate double[] Zumme(int anz);
```

erstellt der Compiler eine Klasse mit den Methoden **BeginInvoke()** und **EndInvoke()**, wie die folgende **IlDasm**-Ausgabe für den Typ `Zumme` 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 `Zumme` erfüllt:

```
static double[] RandomSums(int anz) {
   double[] erg = new double[anz];
   for (int i = 0; i < anz; i++) {
      for (int j = 0; j < 10000000; 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()` unten als *Arbeitsmethode*.

Wir erstellen aus `RandomSums()` ein Delegatenobjekt vom Typ `Zumme` (gleich als *Arbeitsdelegat* bezeichnet) und beauftragen dieses Objekt, seine Methode **BeginInvoke()** auszuführen:

```
Zumme rs = new Zumme(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 13.2.5.3), das über die Eigenschaft **AsyncWaitHandle** angesprochen werden kann.

BeginInvoke() besitzt alle Parameter der Delegatendefinition (im Beispiel: **int**) und außerdem am Ende seiner Parameterliste:

- **AsyncCallBack**
  Dieser Delegatentyp hat 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**
  Dieses Objekt ist in der **IAsyncResult**-Rückgabe 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 enthaltenen **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 Arbeiten abgeschlossen, kann man beim Arbeitsdelegaten per **EndInvoke()**-Aufruf die Ergebnisse anfordern, wobei das Operationsobjekt als Parameter zu übergeben ist:

```
double[] da = z.EndInvoke(ar);
```

Die **EndInvoke()**-Methode hat denselben Rückgabewert wie die die Arbeitsmethode (im Beispiel: die **double[]**).

Neben der Übergabe von Arbeitsergebnissen des Pool-Threads hat die **EndInvoke()**-Methode noch weitere Aufgaben:

- Ist bei der Hintergrundaktivität eine Ausnahme aufgetreten, wird diese von **EndInvoke()** erneut geworfen, so dass sie vom aufrufenden Thread zu Kenntnis genommen und bearbeitet werden kann.
- Etwaige Aufräumungsarbeiten (z.B. zur Freigabe von Ressourcen) sollten von **EndInvoke()** ausgeführt werden.

Statt in einer **while**-Schleife zwischen zwei **IsCompleted**-Abfragen den primären Thread schlafen zu legen (siehe oben), könnten wir die Wartezeit für nützliche Arbeiten verwenden. Damit wäre eine APM-Nutzung erreicht, die Richter (2006, S. 612) als *Polling* bezeichnet und als ineffizient einstuft.

Eine noch weniger empfehlenswerte Alternative besteht darin, ohne Prüfung des Bearbeitungszustands die **EndInvoke()**-Methode des Arbeitsdelegaten aufzurufen, weil diese Methode auf das Auftragsende wartet und damit den primären Thread blockiert.

Mit der **Rückruftechnik** kommen wir ohne 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) {
   Zumme z = (Zumme)ar.AsyncState;
   Console.WriteLine("Report der ermittelten Summen:");
   double[] da = z.EndInvoke(ar);
   foreach (double zs in da)
      Console.WriteLine(" "+zs);
}
```

Das zugehörige Delegatenobjekt wird im **BeginInvoke()**-Aufruf als dritter Parameter übergeben, z.B.:

```
rs.BeginInvoke(3, new AsyncCallback(Report), rs);
```

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.
- 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[] Zumme(int anz);

class Prog {
   static Random zzg = new Random();

   static double[] RandomSums(int anz) {
      . . .
   }
   static void Report(IAsyncResult ar) {
      . . .
   }
```

```
    static void Main() {
      Zumme rs = new Zumme(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.
```

Über die Rückruftechnik gewinnen wir an Flexibilität bei der Verwertung von Ergebnissen einer asynchronen Auftragsabwicklung, weil nach Beendigung der Arbeitsmethode automatisch die Rückrufmethode gestartet wird. Doch ist zu betonen, dass die Rückrufmethode im Hintergrund-Thread abläuft. Wenn der initiierende Thread auf den erfolgreichen Abschluss eines Hintergrund-Threads angewiesen ist, muss irgendeine Synchronisation stattfinden.

Bei Richter (2006, S. 621) finden sich wichtige 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 frei gegeben werden. Wie das obige Beispiel zeigt, kann der **EndInvoke()**-Aufruf in der Rückrufmethode und somit im Hintergrund-Thread erfolgen, so dass der aufrufende Thread auf keinen Fall warten muss.[1]
- Für jedes Operationsobjekt sollte **EndInvoke()** nur einmal aufgerufen werden, weil diese Methode eventuell Aufräumungsarbeiten ausführt, so dass ein erneuter Aufruf scheitern könnte.

13.7.2 Asynchrone Schreib- und Leseoperationen bei Dateien

Für das asynchrone Lesen bzw. Schreiben stellt die Klasse **Stream** im Namensraum **System.IO** die Methodenpaare **BeginRead()** und **EndRead()** bzw. **BeginWrite()** und **EndWrite()** zur Verfügung. Allerdings ist mir auf diesem Weg *keine* asynchrone Dateiausgabe mit Hilfe der Klasse **FileStream** gelungen. Unter Windows 7 (64 Bit) mit .NET 4.0 zeigte **BeginWrite()** ein synchrones (blockierendes) Verhalten, wobei Regeln des Betriebsystems dem angestrebten asynchronen Verhalten im Weg zu stehen scheinen.[2]

Über das im Abschnitt 13.7.1 beschrieben APM-Verfahren gelingt die Dateiausgabe im Hintergrund über einen Pool-Thread aber doch. Das folgende Programm schreibt auf diese Weise 1 GB in

---

[1] Eine Ausnahme von der Pflicht zum **EndInvoke()**-Aufruf lernen Sie in Abschnitt 13.8.1 beim **BeginInvoke()**-Aufrufe an das **Dispatcher**-Objekt des GUI-Threads kennen.

[2] Auf der WWW-Seite http://support.microsoft.com/default.aspx?scid=kb%3Ben-us%3BQ156932 schildert Microsoft mögliche Gründe.

eine Datei, vertrödelt aber die gewonnene Freizeit bis zu einer positiven **IsCompleted**-Abfrage mit **Sleep**-Aufrufen:

```
using System;
using System.IO;

public delegate void FileWriteDelegate(String name, byte[] bar);

class AsyncFileWrite {
   static void AsynWrite(String name, byte[] bar) {
      FileStream s = null;
      try {
         s = new FileStream(name, FileMode.Create);
         s.Write(bar, 0, bar.Length);
      } finally {
         s.Close();
      }
   }

   static void Main() {
      String name = "demo.bin";
      byte[] arr = new byte[1073741824];

      FileWriteDelegate awd = new FileWriteDelegate(AsynWrite);
      Console.WriteLine("Das asynchrone Schreiben startet.\n");
      IAsyncResult iar = awd.BeginInvoke(name, arr, null, null);

      int i = 0;
      while (!iar.IsCompleted) {
         System.Threading.Thread.Sleep(1000);
         Console.WriteLine("{0} Sek. Zeit zum Erledigen anderer Arbeiten", ++i);
      }

      awd.EndInvoke(iar);
      Console.WriteLine("\nFertig!");
   }
}
```

Ein typisches Ablaufprotokoll:

```
Das asynchrone Schreiben von 1 GB startet.

1 Sek. Zeit zum Erledigen anderer Arbeiten
2 Sek. Zeit zum Erledigen anderer Arbeiten
   .  .  .
10 Sek. Zeit zum Erledigen anderer Arbeiten
11 Sek. Zeit zum Erledigen anderer Arbeiten

Fertig!
```

13.8 Timer

Bei manchen Programmen müssen bestimmte Aktivitäten in festen Zeitabständen wiederholt werden (z.B. Aktualisieren der Zeitanzeige in einem Fenster, Überprüfung der Funktionstüchtigkeit eines Servers). In der FCL stehen mehrere Klassen zur Verfügung, deren Objekte mit dem regelmäßigen Aufrufen bestimmter Methoden beauftragt werden können.[1] Anschließend werden zwei Klassen vorgestellt:

[1] Auf der WWW-Seite http://msdn.microsoft.com/en-us/magazine/cc164015.aspx werden die FCL-Timer beschrieben, wobei allerdings an Stelle des GUI-Frameworks WPF noch die ältere WinForms – Variante unterstützt wird.

- Die Klasse **Timer** im Namensraum **System.Threading** kommt in Frage, wenn die regelmäßig auszuführenden Aktionen aufwändig sind und in einem Pool-Thread ablaufen sollten.
- Die Klasse **DispatcherTimer** im Namensraum **System.Windows.Threading** ist im Rahmen von WPF-Anwendungen bequemen einsetzbar. Allerdings laufen die regelmäßig auszuführenden Aktionen im GUI-Thread ab, so dass bei hohem Zeitbedarf eine unergonomisch zäh reagierende Bedienoberfläche resultieren kann.

13.8.1 Regelmäßige Hintergrundaktivitäten

Mit Hilfe der Klasse **Timer** im Namensraum **System.Threading** kann man die CLR beauftragen, eine Methode regelmäßig in einem Pool-Thread auszuführen. Im folgenden Beispiel

```
tim = new Timer(this.HintergrundAktion, null, 0, 1000);
```

kommt eine Konstruktor-Überladung mit folgenden Parameter-Datentypen zum Einsatz:

- **TimerCallback** *callback*
  Dieser Delegatentyp verlangt für die regelmäßig auszuführende Methode folgende Bauart:
  void TimerCallback (Object *state)*
  Im obigen Beispiel kommt die vereinfachte Syntax mit impliziter Konstruktion des Delegatenobjekts zum Einsatz.
- **Object** *state*
  Die regelmäßig auszuführenden Methode erhält beim Aufruf das im zweiten Konstruktorparameter anzugebende Objekt, so dass ihr Verhalten gesteuert werden kann. Ist (wie im Beispiel) kein Parameter erforderlich, übergibt man eine **null**-Referenz.
- **long** *dueTime*
  Die Zeit bis zum ersten Aufruf in Millisekunden
- **long** *period*
  Die Zeit zwischen zwei Aufrufen in Millisekunden

Bei aufwendigen 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 13.2). 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.:

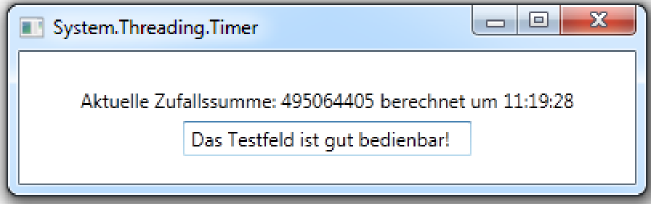
```
tim.Change(0, 2000);
```

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 berechnet die Methode `HintergrundAktion` alle 10 Sekunden die Summe aus 300 Millionen Zufallszahlen:

```
using System;
using System.Windows;
using System.Threading;
using System.Windows.Controls;

delegate void LabelUpdateDelegate(String s);
```

```
class ThreadingTimer : Window {
   Label anzeige;
   System.Threading.Timer tim;
   Random zzg = new Random();
   LabelUpdateDelegate laup;

   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);

      laup = new LabelUpdateDelegate(this.UpdateLabel);
      tim = new Timer(this.HintergrundAktion, null, 0, 10000);

      TextBox eingabe = new TextBox();
      eingabe.Width = 180;
      lm.Children.Add(eingabe);
   }

   void HintergrundAktion(object info) {
      double zs = 0;
      for (int i = 0; i < 300000000; i++)
         zs += zzg.Next(100);
      anzeige.Dispatcher.BeginInvoke(laup, zs.ToString());
   }

   void UpdateLabel(String s) {
      anzeige.Content = "Aktuelle Zufallssumme: " + s + " berechnet um " +
                        DateTime.Now.ToLongTimeString();
   }

   [System.STAThreadAttribute]
   static void Main() {
      new Application().Run(new ThreadingTimer());
   }
}
```

Trotz des erheblichen CPU-Zeitverbrauchs durch die Hintergrundaktivität reagiert die Benutzeroberfläche des Programms verzögerungsfrei:



Im .NET-Framework sind die meisten Eigenschaften und Methoden der Steuerelemente bewusst *nicht* Thread - sicher, um Leistungsverluste durch die Thread-Synchronisation zu vermeiden. Folglich müssen Zugriffe auf Steuerelemente dem Thread vorbehalten bleiben, der sie erzeugt hat. Zugriffe durch fremde Threads werden von der CLR verhindert.

Wie kann nun aber ein Hintergrund-Thread Aktualisierungen der Benutzeroberfläche veranlassen? Dies ist die übliche Lösung:

- Man erstellt eine GUI - Aktualisierungsmethode (im Beispiel: `UpdateLabel()`),
- definiert einen passenden Delegatentyp (im Beispiel: `LabelUpdaterDelegate`),
- erzeugt ein Delegatenobjekt (`new LabelUpdateDelegate(this.UpdateLabel)`)
- und veranlasst den GUI-Thread über die **Invoke**()-Methode oder die die **BeginInvoke**()-Methode des zuständigen **Dispatcher**-Objekts, das Delegatenobjekt auszuführen.

Beim **Invoke**()-Aufruf wartet der Hintergrund-Thread, bis der GUI-Thread die Aktualisierungsmethode beendet hat. Beim **BeginInvoke**()-Aufruf wartet der Hintergrund-Thread *nicht* auf das Ende der Aktualisierung durch den GUI-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. Während zu jedem **BeginInvoke**() - Aufruf an ein *Delegatenobjekt* (siehe Abschnitt 13.7.1) der zugehörig **End-Invoke**() - Aufruf durchgeführt werden sollte, ist im vorliegenden Fall ausnahmsweise *kein* **End-Invoke**()-Aufruf erforderlich.

13.8.2 Regelmäßige Aktivitäten im GUI-Thread

Im Namensraum **System.Windows.Threading** ist eine Klasse namens **DispatcherTimer** vorhanden. Diese arbeitet ereignisorientiert und ist sehr bequem in eine GUI-Anwendung einzubinden, 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 = "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 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;
      for (int i = 0; i < 300000000; i++)
         zs += zzg.Next(100);
      anzeige.Content = "Aktuelle Zufallssumme: " + zs + " berechnet um " +
                        DateTime.Now.ToLongTimeString();
   }
```

```
        [System.STAThreadAttribute]
        static void Main() {
            new Application().Run(new WPFTimer());
        }
    }
```

Diesmal laufen die vom Timer angestoßenen Methodenaufrufe jedoch im GUI-Thread ab, und bei gleicher Rechenlast resultiert eine unergonomisch zähe Bedienoberfläche. Im Beispiel sind nach jedem Tick-Ereignis mehrere Sekunden lang keine Eingaben in das Textfeld möglich.[1]

Für das regelmäßige Starten von Methoden mit geringem Rechenzeitbedarf ist die Klasse **DispatcherTimer** aber durchaus geeignet. und wegen der einfachen Verwendbarkeit attraktiv.

13.9 Task Parallel Library (TPL)

Die in .NET 4.0 neu eingeführte *Task Parallel Library* bringt neue Klassen, die speziell zu optimalen Nutzung von Mehrkernsystemen konzipiert wurden. Für jede selbständig auszuführende Teilaufgabe entsteht ein Objekt der neuen Klasse **Task**. Zwar wird jede **Task** letztlich von einem Thread aus dem Poolpool erledigt, doch muss keinesfalls für jede **Task** ein eigener Arbeitsauftrag in die Warteschlange des Threadpools gestellt werden (vgl. Abschnitt 13.6). Vielmehr wird unter Berücksichtigung der verfügbaren CPU-Kerne eine leistungsoptimierende Anzahl von Threadpool-Aufträgen ermittelt.

Neben dieser Leistungsoptimierung bietet die TPL eine gute Kontrolle über die **Task**-Objekte (z.B. Warten auf Beendigung, Abbrechen, Ausnahmebehandlung), so dass **Task**-Objekte in vielen Situationen gegenüber der direkten Verwendung von **Thread**-Objekten oder direkten Aufträgen an den Threadpool zu bevorzugen sind.

Daher empfiehlt Microsoft, in Anwendungen für die Zielplattform .NET 4.0 Nebenläufigkeit möglichst per TPL zu realisieren.[2]

Eine komplette Behandlung der TPL würde den Rahmen dieses Manuskripts sprengen. Weiterführende Informationen finden Sie z.B. im online verfügbaren Buch von Campbell et al. (2010).

13.9.1 Implizite Task-Erstellung über statische Methoden der Klasse Parallel

Über die statische Methode **Invoke**()der Klasse **Parallel** aus dem Namensraum **System.Threading.Tasks** startet man eine Serie von Aktionen, wobei die CLR über das Ausmaß der Parallelisierung entscheidet. Man übergibt die Aufgaben per Serienparameter vom Typ **Action[]**, wobei **Action** ein Delegatentyp mit folgender Signatur ist:

public delegate void Action()

Im folgenden Programm erfüllt die Methode `RandomSum()`, die eine Summe von 10 Millionen Pseudozufallszahlen berechnet und zusammen mit dem ausführenden Thread ausgibt, den Delegatentyp **Action**:

---

[1] Auf einem Rechner mit Intel-CPU Core i3 550.
[2] Siehe http://msdn.microsoft.com/en-us/library/dd537609.aspx

```
using System;
using System.Threading;
using System.Threading.Tasks;

class ParallelInvoke {
   static int anz = 16;

   static void RandomSum() {
      Random zzg = new Random();
      double erg = 0.0;
      for (int j = 0; j < 10000000; j++)
         erg += zzg.NextDouble();
      Console.WriteLine("RandomSum " + erg + " berechnet von Thread " +
                        Thread.CurrentThread.ManagedThreadId);
   }

   static void Main() {
      Action[] av = new Action[anz];
      for (int i = 0; i < anz; i++)
         av[i] = new Action(RandomSum);
      long start = DateTime.Now.Ticks;
      Parallel.Invoke(av);
      Console.WriteLine("Nach dem Invoke-Aufruf. Benötigte Zeit im Sek.: " +
                        (DateTime.Now.Ticks - start) / 1.0e7);
   }
}
```

In der Methode **Main()** wird ein **Action**-Array mit Delegatenobjekten gefüllt, deren Aufgabe darin besteht, die Methode `RandomSum()` aufzurufen. Per **Invoke**()-Aufruf werden **Task**-Objekte erstellt und durch eine geeignete Anzahl von Threads ausgeführt. Beim anschließend protokollierten Programmeinsatz auf einem Rechner mit zwei realen und vier virtuellen Kernen (Intel Core i3 mit Hyperthreading) sind 16 **Task**-Objekte auf fünf Threads verteilt worden:

```
RandomSum 4998337,2350927 berechnet von Thread 10
RandomSum 5001517,81784143 berechnet von Thread 13
RandomSum 4999684,53862916 berechnet von Thread 12
   . . .
RandomSum 5000057,54340198 berechnet von Thread 14
RandomSum 5000321,52146498 berechnet von Thread 10
RandomSum 4999582,45087807 berechnet von Thread 11
RandomSum 5000075,24225269 berechnet von Thread 13
Nach dem Invoke-Aufruf. Benötigte Zeit in Sek.: 1,1660156
```

Diese Zahl ergibt sich aus dem Vergleich der Kennungen, die man über die **Thread**-Eigenschaft **ManagedThreadId** in Erfahrung bringt.

Der **Invoke()** endet erst dann, wenn alle Aufgaben erledigt sind. Es ist keine Reihenfolge der Bearbeitung garantiert.

Um den Leistungsvorteil der TPL im Vergleich zur äußerst simplen Kreation eines eigenen Threads für jede Zufallssummenberechnung beobachten zu können, lassen wir durch eine Programmvariante 1000 Summen aus lediglich 100000 Zufallszahlen berechnen. Es liegen also viele Aufgaben von jeweils relativ kleinem Umfang vor. Wir vergleichen den Zeitaufwand des **Invoke**()-Aufrufs mit der offenbar ineffizienten Kreation eines eigenen Threads für jede Zufallssummenberechnung:

```
for (int i = 0; i < anz; i++)
  (new Thread(RandomSum)).Start();
```

Wie erwartet schneidet die TPL durch ihre Beschränkung auf wenige Threads günstiger ab:

```
Zeitaufwand per Invoke in Sek.: 0,711914
Zeitaufwand für Einzelne Threads in Sek.: 4,7880859
Zeitaufwand bei Threadpool-Items in Sek.: 0,7011718
```

Im Vergleich zur Kreation von Threadpool-WorkItems

```
for (int i = 0; i < anz; i++)
   ThreadPool.QueueUserWorkItem(RandomSumTP);
```

kann die TPL im aktuellen Beispiel allerdings *nicht* punkten.

Muss eine Aufgabe für eine Folge von Indexwerten, aber nicht unbedingt in der natürlichen Reihenfolge, ausgeführt werden, erlaubt die statische **Parallel**-Methode **For()** eine Parallelisierung. Es ist eine Methode zu definieren, die den Delegatentyp **Action<int>** erfüllen, also von folgender Bauart sein muss:

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, 100, RandomSum);
```

Das Delegatenobjekt wird für jeden Indexwert bzw. bei jedem Schleifendurchgang aufgerufen und erhält den Indexwert als Aktualparameter. Man darf sich aber keinesfalls darauf verlassen, dass die Aufrufe in der Reihenfolge der Indexwerte stattfinden.

Von der folgenden Methode werden Zufallssummen in Abhängigkeit von einem Startwert des Pseudozufallszahlengenerators ermittelt und protokolliert. Als Startwert verwendet die Methode den **int**-wertigen Parameter, den sie beim Aufruf erhält:

```
static void RandomSum(int i) {
   Random zzg = new Random(i);
   double erg = 0.0;
   for (int j = 0; j < 1000000; j++)
      erg += zzg.NextDouble();
   Console.WriteLine("RandomSum " + erg + " mit Startwert " + start +
                     " berechnet von Thread " + Thread.CurrentThread.ManagedThreadId);
}
```

Dieser **Parallel.For()** - Aufruf liefert Exemplare für die Startwerte von 1 bis 99:

```
ParallelLoopResult plr = Parallel.For(1, 100, RandomSum);
```

Über die Rückgabe ist ggf. der kleinste Schleifenindex zu erfahren, der zum Abbruch per **Break** geführt hat.

Analog erstellt die **Parallel**-Methode **ForEach()** eine Serie von **Task**-Objekten aus einer Methode und passenden Argumenten in einer Kollektion.

13.9.2 Explizite Task-Objekte

In Abschnitt 13.9.1 haben wir statische Methoden der Klasse **Parallel** verwendet, die im Hintergrund Objekte der Klasse **Task** aus dem Namensraum **System.Threading.Tasks** erzeugen. Die Verwendung explizit erstellter **Task**-Objekte bringt den Vorteil, dass man über diverse Methoden (z.B. **Wait()**) und Eigenschaften (z.B. **Status**) während der gesamten Bearbeitungszeit Informationen und Einfluss gewinnen kann.

13.9.2.1 Aufgaben ohne bzw. mit Rückgabe erstellen

Statt ein **Task**-Objekt per Konstruktor zu erzeugen 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 Objekt der Klasse **TaskFactory** ist über die statische **Task**-Eigenschaft **Factory** ansprechbar, z.B.:

```
Task task = Task.Factory.StartNew(SampleMean);
```

Soll eine Aufgabe bzw. die zugrunde liegende 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 TaskDemo {
   static int sg = 10000;

   static double SampleMean() {
      Random zzg = new Random();
      double erg = 0.0;
      for (int j = 0; j < sg; j++)
         erg += 5.0 + zzg.NextDouble() * 10.0;
      return erg / sg;
   }

   static void Main() {
      Task<double> task = Task.Factory.StartNew<double>(SampleMean);
      Console.WriteLine("Stichprobenmittel = " + task.Result);
   }
}
```

entsteht aus der Methode `SampleMean()`, die aus einer Stichprobe mit `sg` Zufallszahlen den Mittelwert bestimmt, mit Hilfe der Methode **StartNew<double>()** ein Objekt der Klasse **Task<double>**:

```
Task<double> task = Task.Factory.StartNew<double>(SampleMean);
```

Task<*TResult*>-Objekte machen ihr Resultat ggf. über die Eigenschaft **Result** verfügbar, z.B.:

```
Console.WriteLine("Stichprobenmittel = " + task.Result);
```

Ein Zugriff auf die **Result**-Eigenschaft einer noch nicht abgeschlossenen Aufgabe versetzt den Aufrufer in den Wartezustand.

13.9.2.2 Parameterabhängige Aufgaben

Soll eine Aufgabe bzw. die zugrunde liegende Methode in Abhängigkeit von einem Parameter tätig werden, ist ein alternativer Delegatentyp und eine entsprechende **StartNew**<*TResult*>() - Überladung zu verwenden.

Bei den oben verwendeten Methoden `RandomSum()` bzw. `SampleMean()`, welche die Summe bzw. den Mittelwert aus einer Serie von Zufallszahlen berechnen, ist bei Verwendung in *mehreren* Aufgaben zu kritisieren, dass sie den voreingestellten, aus der Systemzeit abgeleiteten Startwert für den Pseudozufallszahlengenerator verwenden. So kann es leicht passieren, dass mehrere kurz nacheinander gestartete Aufgaben dieselben Pseudozufallszahlen verwenden. Dies wird bei der folgenden `SampleMean()`-Variante vermieden, die einen per Parameter übergebenen Startwert verwendet:

```
static double SampleMean(Object obj) {
   int start = (int) obj;
   Random zzg = new Random(start);
   double erg = 0.0;
   for (int j = 0; j < sg; j++)
      erg += 5.0 + zzg.NextDouble()*10.0;
   return erg/sg;
}
```

Sie erfüllt den Delegatentyp **Func<Object, double>** und kann daher in der folgenden **Factory.StartNew**<*TResult*>() - Überladung

public Task<*TResult*> **StartNew**<*TResult*>**(Func<Object,** *TResult*> *function*, **Object** *state*)

dazu verwendet werden, **Task<double>** - Objekte mit Startinformationen zu erstellen, z.B.:

```
Task.Factory.StartNew<double>(SampleMean, i);
```

13.9.2.3 Fortsetzungsaufgaben

Über diverse Instanzmethoden der Klassen **Task**, **Task<*TResult*>** und **TaskFactory** kann man eine Aufgabe erstellen, die gestartet wird, sobald eine andere Aufgabe oder ein Array mit anderen Aufgaben abgeschlossen ist. Wir setzen unser Beispiel zur Untersuchung von Zufallszahlen fort und erzeugen einen Array mit `anz` Aufgaben, die jeweils einen Stichprobenmittelwert berechnen:

```
Task<double>[] sampleMeanTaskArray = new Task<double>[anz];
for (int i = 0; i < anz; i++)
   sampleMeanTaskArray[i] = Task.Factory.StartNew<double>(SampleMean, i);
```

Dank moderner Rechenleistung und TPL sollte es kein Problem darstellen, z.B. aus 10000 Stichproben vom jeweiligen Umfang 10000 parallel den Mittelwert zu berechnen. Wenn alle Stichprobenmittel vorliegen, sollen diese in einer Fortsetzungsaufgabe untersucht werden. Die zugrunde liegende Methode `Summary()` erfüllt den Delegatentyp **Action<Task<double>[]>**:

```
static void Summary(Task<double>[] sampleMeanTaskArray) {
   double x;
   double min = Double.MaxValue;
   double max = Double.MinValue;
   int n = sampleMeanTaskArray.Length;
   for (int j = 0; j < n; j++) {
      x = sampleMeanTaskArray[j].Result;
      if (x < min) min = x;
      if (x > max) max = x;
   }
   Console.WriteLine("\nAnzahl = {0,5:d}\nUmfang = {1,5:d}\nmin. Mittel = {2,10:f7}"+
      "\nmax. Mittel = {3,10:f7}\nSpannweite = {4,10:f7}", n, sg, min, max, (max-min));
}
```

Sie erhält per Parameter den Array mit den Aufgaben zum Berechnen der Stichprobenmittelwerte und ermittelt daraus Minimum, Maximum und Spannweite (Maximum - Minimum). So lässt sich z.B. beobachten, wie die Spannweite von der Anzahl und vom Umfang der Stichproben abhängt.

Aus dem folgenden Aufruf der **Factory**-Methode **ContinueWhenAll<double>()** resultiert eine neue Aufgabe, welche asynchron die Methode `Summary()` ausführt, sobald alle Aufgaben im Array `sampleMeanTaskArray` erledigt sind:

```
Task.Factory.ContinueWhenAll<double>(sampleMeanTaskArray, Summary);
```

Um 10000 Stichproben vom Umfang 10000 auszuwerten, benötigt ein Rechner mit der Intel-CPU Core i3 keine Sekunde:

```
Anzahl = 10000
Umfang = 10000
min. Mittel =  9,8943772
max. Mittel = 10,1317129
Spannweite =  0,2373357
Zeit im Sek.: 0,75
```

13.9.2.4 Warten (auf Ausnahmen)

Für einen Thread, der eine Aufgabe gestartet hat, gibt es zwei Gründe, die **Wait()** - Methode dieses **Task**-Objekts aufzurufen:

- Das weitere Verhalten des initiierenden Threads hängt vom Ergebnis der Aufgabe ab, so dass deren Beendigung abgewartet werden muss.

- Bei der Aufgabenbearbeitung sind Ausnahmen zu erwarten, die vom initiierenden Thread bearbeitet werden sollen, damit sie nicht zum Beenden des Programms führen.

Bei *mehreren* initiierten Aufgaben kommen aus analogen Gründen die **Task**-Methoden **WaitAll()** und **WaitAny()** in Frage.

Tritt bei der Aufgabenbearbeitung eine Ausnahme auf, wird diese dem initiierenden Thread als innere Ausnahme eines **AggregateException**-Objekts zugestellt. Durch den Aufruf der **Wait()**-Methode im Rahmen einer **try**-Anweisung kann die **AggregateException** abgefangen und analysiert werden. Abhängig vom Typ der inneren Ausnahme kommen als Maßnahmen in Frage:

- Innere Ausnahme behandeln
- **AggregateException** erneut werfen

Im folgenden Beispielprogramm beschränkt sich die Aufgabenmethode auf das Werfen einer Ausnahme, wobei zwei selbst definierte Ausnahmeklassen auftreten (vgl. Abschnitt 10.6).

```
using System;
using System.Threading.Tasks;

public sealed class HarmlessException : Exception {
   public HarmlessException(String message) : base(message) {}
}

public sealed class SeriousException : Exception {
   public SeriousException(String message)  : base(message) {}
}

class WaitDemo {
   static void Dummy() {
      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(Dummy);
      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. Dabei ist eine Methode verlangt, die einen Parameter vom Typ **Exception** entgegen nimmt und eine Rückgabe vom Typ **bool** abliefert. Bei Aufruf erhält diese Methode das innere Ausnahmeobjekt und soll per Rückgabewert darüber informieren, ob eine Behandlung stattgefunden hat (**true**) oder nicht (**false**). Beim Rückgabewert **false** wird die **AggregateException** automatisch neu geworfen.

Wird im Beispiel bei der Aufgabenbearbeitung eine `HarmlessException` geworfen, findet eine Behandlung statt, und das Programm wird normal weitergeführt:

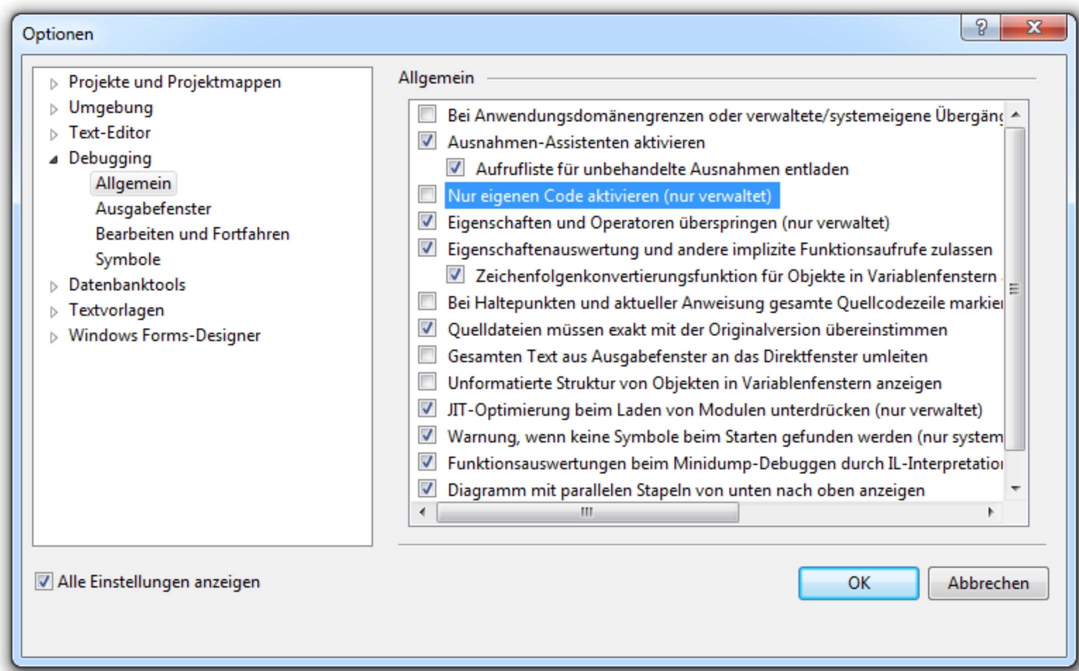
```
Harmlos
Normales Ende
```

Im Falle einer `SeriousException` unterbleibt eine Behandlung, und das Programm wird von der CLR beendet.

Damit Sie das versprochene Verhalten bei der Ausführung des Beispielprogramms im Rahmen der Entwicklungsumgebung auch tatsächlich beobachten können, müssen Sie nach

Extras > Optionen > Debugging > Allgemein

die Markierung des Kontrollkästchens **Nur eigenen Code aktivieren** entfernen:



In diesem Abschnitt konnte nur die Grundlogik der Ausnahmebehandlung bei Verwendung der Task Parallel Library dargestellt werden. Für praktische Anwendungen sind weitere Informationen erforderlich (siehe z.B. http://msdn.microsoft.com/de-de/library/dd997415.aspx).

13.9.2.5 Aufgaben abbrechen

Mit Hilfe der TPL-Typen **CancellationTokenSource** und **CancellationToken** kann eine Aufgabe aufgefordert werden, ihre Tätigkeit einzustellen. Ein Objekt der Klasse **CancellationTokenSource**

```
CancellationTokenSource cts = new CancellationTokenSource();
```

stellt über seine **Token**-Eigenschaft eine Instanz vom Typ **CancellationToken** als „Teilnahmekärtchen“ am Terminierungsverfahren zur Verfügung:

```
CancellationToken ct = cts.Token;
```

Um die beteiligten Aufgaben zu unterbrechen, ruft man die **CancellationTokenSource** -Methode **Cancel**() auf, z.B.:

```
cts.Cancel();
```

Daraufhin wird die **IsCancellationRequested**-Eigenschaft der **CancellationToken**-Instanzen auf **true** gesetzt.

Das einer Aufgabe zugeordnete **CancellationToken** kann in einem passend parametrisierten **Task**-Konstruktor oder in geeigneten Überladungen der **TaskFactory**-Methode **StartNew()** gesetzt werden, z.B.:

```
Task task = Task.Factory.StartNew(Rumoren, ct, ct);
```

Ist das Terminierungssignal schon vor Beginn der Bearbeitung gesetzt, wird die Aufgabe durch die TPL-Logik vom Zustand **WaitingToRun** sofort in den Zustand **Canceled**, ohne je den Zustand **Running** erlebt zu haben.

In der Regel muss das Terminierungssignal auch innerhalb der Delegaten-Methode überwacht werden. Um eine Kopie dorthin zu befördern, verwendet man meist einen Lamda-Ausdruck.[1] Weil das Manuskript diese Technik erst in Abschnitt 21.1.4 behandelt, verwenden wir einen Aufgabendelegaten mit **Object**-Parameter:

```
static void Punktieren(Object obj) {
   CancellationToken ct = (CancellationToken ) obj;
   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 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 Klasse **OperationCanceled-Exception** Verwendung findet. Dem Ausnahmeobjekt wird das **CancellationToken** der Aufgabe mit auf den Weg gegeben, so dass der Ausnahmefänger die Identität der Aufgabe feststellen kann, die kooperativ auf das Terminierungssignal reagiert hat.

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() {
   CancellationTokenSource cts = new CancellationTokenSource();
   CancellationToken ct = cts.Token;

   // Aufgrund der lokalen Referenz unterbleibt die GC, solange wir uns für
   // das Task-Objekt interessieren.
   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] Dabei erhält die Delegatenmethode die **CancellationToken**-Instanz der Aufgabe per Parameter:

```
static void Rumoren(CancellationToken ct) {
    . . .
}
```

Im **StartNew()**-Aufruf wird ein **Action**-Delegat per Lamda-Ausdruck geliefert:

```
Task task = Task.Factory.StartNew( () => Punktieren(ct), ct);
```

Dieser Lambda-Ausdruck stellt eine parameterlose Methode ohne Rückgabe dar und erfüllt somit den Delegatentyp **Action**. Er besteht aber aus einem Aufruf der Methode `Punktieren()` und kann die **CancellationToken**-Instanz übergeben, weil ein Lambda-Ausdruck auf die lokalen Variablen im Kontext zugreifen darf.

```
    try {
      task.Wait();
    } catch (AggregateException ae) {
      foreach (Exception ie in ae.InnerExceptions)
        Console.WriteLine("Message der inneren Ausnahme:   "+ie.Message);
    }
    Console.WriteLine("Aufgabe abgebrochen. Zustand:   "+task.Status);
  }
```

Beim folgenden Programmeinsatz hatte sich der Benutzer schon nach ca. einer Minute an den Punkten satt gesehen:

```
Aufgabe gestartet. Zustand:     WaitingToRun.
Stoppen mit Enter

Aufgabe in Bearbeitung
.............................................................

Zustand der Aufgabe vor Cancel: Running
Signal zum Abbrechen gesetzt.

.
Signal zum Abbrechen erhalten

Message der inneren Ausnahme:   Eine Aufgabe wurde abgebrochen.
Aufgabe abgebrochen. Zustand:   Canceled
```

CancellationToken ist eine Struktur, und eine **CancellationTokenSource** liefert somit *Kopien* ihres Tokens für die beteiligten Aufgaben. Weil keine Objekte auf dem für alle Threads zugreifbaren Heap-Speicher als Token verwendet werden, resultiert Thread-Sicherheit bei der Terminierungsprozedur (Michaelis 2010, S. 721).

13.10 Übungsaufgaben zu Kapitel 13

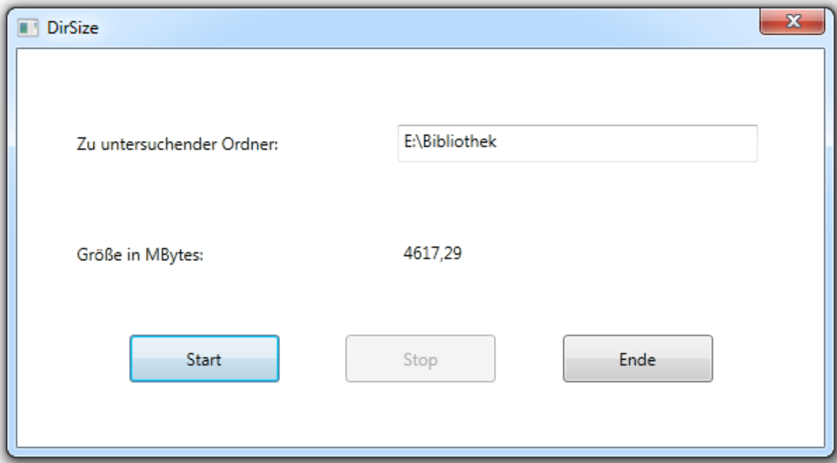
1) Welche von den folgenden Aussagen sind richtig?

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. Bei der **lock**-Anweisung
   ```
   lock(sperre) {
   ...
   }
   ```
   ist sichergestellt, dass der kritische Block auch beim Auftreten einer Ausnahme verlassen wird.
4. Ist für einen Thread die **Abort()**-Methode aufgerufen worden, ist sein Ende nicht mehr zu verhindern.

2) Objekte der Signalisierungsklasse **CountdownSignal** 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 **CountdownSignal** 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 statet:
.. . .  .  .   .   .    .    .     .     .      .      .       .        .
```

3) Erstellen Sie eine GUI-Anwendung, welche die Größe eines Ordners (inklusive aller Unterordner) in einem eigenen Thread berechnet, so dass die Bedienelemente des Formulars stets verzögerungsfrei reagieren, z.B.:



14 Netzwerkprogrammierung

Die FCL enthält zahlreiche Klassen zur Netzwerkprogrammierung, wobei man zwischen einem bequemen Zugriff auf Netzwerkressourcen über Standardprotokolle der Anwendungsebene (z.B. HTTP) und einer Programmierung auf elementaren Protokollebenen mit einer entsprechend weiter reichenden Kontrolle wählen kann.

In diesem Manuskript können wichtige Einsatzfelder für die Netzwerkprogrammierung *nicht* behandelt werden, z.B.:

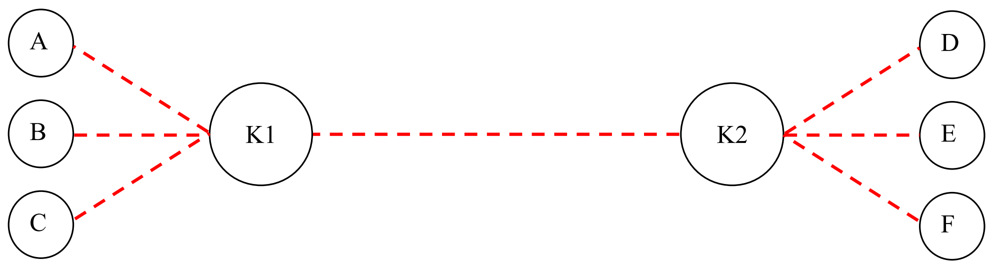
- **Server-Anwendungen zur dynamischen Erstellung individueller HTML-Seiten**
  Auf der Serverseite werden (oft in Kooperation mit einer Datenbankanwendung) HTML-Seiten aufgrund einer speziellen Anforderung individuell erstellt und dann zum klientenseitigen Browser gesandt. In Abschnitt 14.2.3 werden wir uns allerdings mit Klientenprogrammen befassen, die individuell erstellte HTML-Seiten anfordern.
- **Webdienste**
  Bei dieser Technik kommunizieren lose gekoppelte Programme, die mit unterschiedlichen Techniken (z.B. Programmiersprachen) erstellt worden sind und auf verschiedenen Rechnern ablaufen über das Internet miteinander unter Beachtung eines in der ***W**eb **S**ervices **D**escription **L**anguage* (WSDL) formulierten Vertrags. Für die ausgetauschten Daten wird das XML-Format verwendet. So kann z.B. eine lokale Anwendung vom Server einer Fluglinie Daten über Verbindungen, freie Plätze und Preise beschaffen.

14.1 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 für die korrekte Weiterleitung der Pakete zuständig:



Von der Anwendungsebene (z.B. Versandt einer E-Mail über einen SMTP-Server (*Simple Mail Transfer Protocol*)) bis zur physikalischen 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.

14.1.1 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) eingeordnet werden.

1. Physikalische 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.

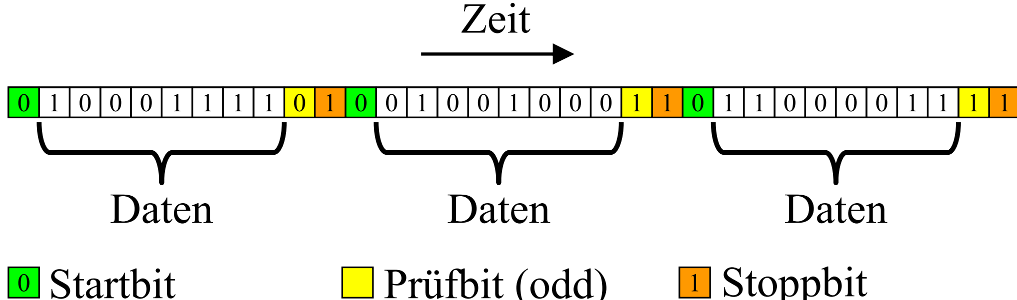
2. Link-Ebene (gesicherte Frame-Übertragung, z.B. per Ethernet)

Hier wird vereinbart, wie zwischen zwei direkt verbundenen Stationen ein *Frame* zu übertragen ist, der aus einer Anzahl von Bits besteht und durch eine Prüfsumme gesichert ist. In der Regel gehören zum Protokoll dieser Ebene auch Start- und Endmarkierungen, damit sich die beteiligten Geräte rechtzeitig auf eine Informationsübertragung einstellen können.

Im Beispiel der seriellen Datenübertragung kann folgender Frame-Aufbau verwendet werden:

Startbit:	0
8 Datenbits	
Parität:	odd (siehe unten)
Stoppbit:	1

In folgender Abbildung sind drei 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 (siehe Spurgeon 2000, S. 40ff):

- Der Header enthält u.a. die MAC-Adressen (*Media Access Control*) von Sender und Empfänger. Diese Level-2 - Adressen sind nur für die Subnetz-interne Kommunikation relevant.
- Es können zwischen Daten im Umfang von 46 bis 1500 Byte transportiert werden.

3. Netzwerkebene (Paketübertragung, z.B. per IP)

Die Frames der zweiten Ebene hängen von der verwendeten Netzwerktechnik ab, so dass auf der Strecke vom Absender bis zum Empfänger in der Regel *mehrere* Frame-Architekturen beteiligt sind (z.B. bei der LAN-Verbindung zum hausinternen Router eine andere als auf der Telefonstrecke zum DSL-Provider). Auf der dritten Ebene kommen hingegen Informations-**Pakete** zum Einsatz, die auf der gesamten Strecke (im Intra- und/oder im Internet) unverändert bleiben und beim Wechsel der Netzwerktechnik in verschiedene Schicht 2 - Container umgeladen werden (siehe Abschnitt 14.1.2).

Durch die Protokolle der Schicht 3 sind u.a. folgende Aufgaben zu erfüllen:

- **Adressierung (über Subnetzgrenzen hinweg gültig)**
  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 Paketes zum Ziel. Vermittlungsrechner (sog. *Router*) entscheiden darüber, welchen Weg ein Paket nehmen soll.
- **Datenflusskontrolle**
  Eine weitere Aufgabe der dritten Protokollebene besteht in der Datenflusskontrolle zur Vermeidung von Überlastungen.

Bei aktuellen Netzwerken kommt auf der Ebene 3 überwiegend das **IP-Protokoll** zum Einsatz. Seine Pakete bezeichnet man auch als **IP-Datagramme**. In der heute noch üblichen IP-Version 4 (IPv4) besteht eine Adresse aus 32 Bits, üblicherweise durch vier per Punkt getrennte Dezimalzahlen (aus dem Bereich von 0 bis 255) dargestellt, z.B.:

192.168.178.12

Bei der (schon seit vielen Jahren) kommenden IP-Version 6 (IPv6) besteht eine Adresse aus 128 Bits, welche durch acht per Doppelpunkt getrennte Hexadezimalzahlen (aus dem Bereich von 0 bis FFFF) dargestellt werden, z.B.:

2001:88c7:c79c:0000:0000:0000:88c7:c79c

Innerhalb eines Blocks dürfen führende Nullen weggelassen werden, z.B.:

2001:88c7:c79c:0:0:0:88c7:c79c

Eine Gruppe aufeinanderfolgender Blöcke mit dem Wert 0000 bzw. 0 darf durch zwei Doppelpunkte ersetzt werden, z.B.:

2001:88c7:c79c::88c7:c79c

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 versandten *Echo Requests* und die zugehörigen Antworten zählen zu den ICMP - Nachrichten.

4. Transportschicht (gesicherte Paketübertragung, z.B. per TCP)

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 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 spezielle Anwendung auf Rechner A mit einer speziellen Anwendung auf Rechner B kommunizieren kann, werden so genannte **Ports** verwendet. Hierbei handelt es sich um Zahlen zwi-

schen 0 und 65535 ($2^{16} - 1$), die eine kommunikationswillige bzw. -fähige Anwendung identifizieren. So wird es z.B. möglich, auf einem Rechner verschiedene Serverprogramme zu installieren, die trotzdem von Klienten aufgrund ihrer verschiedenen Ports (z.B. 21 für einen FTP-Server, 80 für einen WWW-Server) gezielt angesprochen werden können. Während die Ports von 0 bis 1023 ($2^{10} - 1$) für Standarddienste fest definiert sind, werden die höheren Ports nach Bedarf vergeben, z.B. zur temporären Verwendung durch kommunikationswillige Klientenprogramme.

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 für die Kommunikation zugeteilte Portnummer

Weitere Eigenschaften einer TCP-Verbindung:

- Das TCP-Protokoll stellt eine *virtuelle Verbindung* zwischen zwei Anwendungen her.
- Auf beiden Seiten steht eine als **Socket** 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.

Von den Internet-Protokollen ist auch das **User Datagram Protocol** (UDP) auf der Ebene 5 anzusiedeln. 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 Paketversandt 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

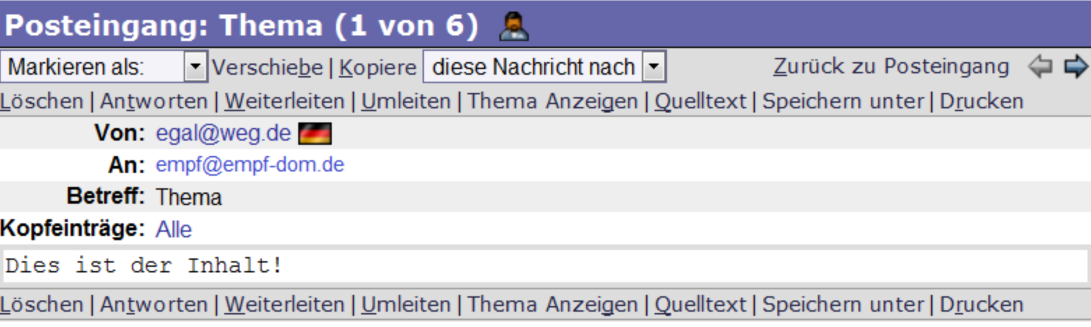
Hier geht es z.B. um die Verschlüsselung oder Komprimierung von Daten. Die TCP/IP - Protokollfamilie kümmert sich nicht darum, sondern überlässt derlei Arbeiten den Anwendungen.

7. Anwendung (Protokolle für Endbenutzer-Dienstleistungen, z.B. per HTTP oder SMTP)

Hier wird für verschiedene Dienste festgelegt, wie Anforderungen zu formulieren und Antworten auszuliefern sind. Einem SMTP-Serverprogramm (*Simple Mail Transfer Protocol*), das an Port 25 lauscht, kann ein Klientenprogramm z.B. folgendermaßen eine Mail übergeben:

Klient	Serverantwort
telnet srv.srv-dom.de 25	220 srv.srv-dom.de ESMTP Postfix
HELO mainpc.client-dom.de	250 srv.srv-dom.de
MAIL FROM:egal@weg.de	250 2.1.0 Ok
RCPT TO:empf@srv-dom.de	250 2.1.5 Ok
DATA	354 End data with <CR><LF>.<CR><LF>
From: egal@weg.te	
To: empf@srv-dom.de	
Subject: Thema	
Dies ist der Inhalt!	
.	250 2.0.0 Ok: queued as 43A7D6D91AC
QUIT	221 Bye

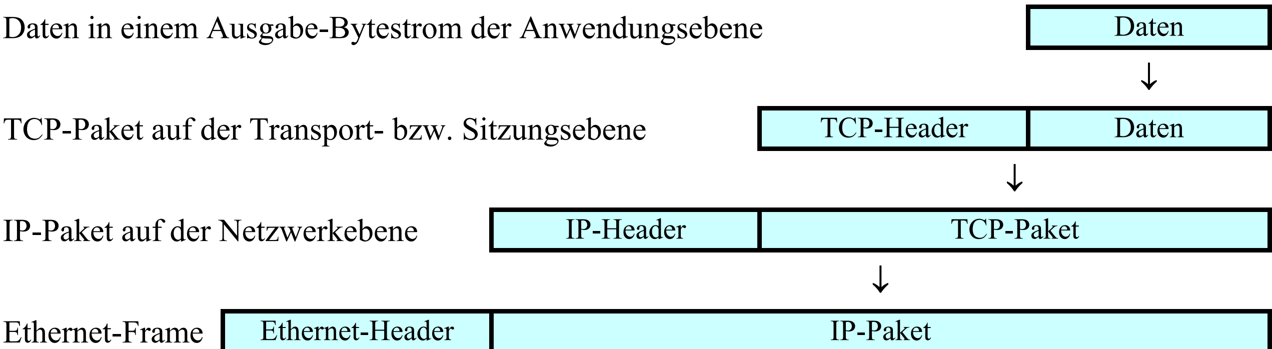
Der Mailempfänger glaubt hoffentlich nicht an die angezeigten Adressen:



Noch häufiger als das SMTP-Protokoll kommt im Internet auf Anwendungsebene das HTTP-Protokoll (*Hyper Text Transfer Protocol*) für den Austausch zwischen Web-Server und -Browser zum Einsatz.

14.1.2 Zur Funktionsweise von Protokollstapeln

Möchte eine Anwendung (genauer: eine aktive Anwendungsinstanz) auf dem Rechner A über ein TCP/IP – Netzwerk eine gemäß zugehörigem Anwendungsebenen-Protokoll (z.B. SMTP) zusammengestellte Sendung an eine korrespondierende Anwendung auf dem Rechner B schicken, dann übergibt sie eine Serie von Bytes an die TCP-Schicht von Rechner A, welche daraus TCP-Pakete erstellt. Wir beschränken uns auf den einfachen Fall, dass alle Daten in *ein* TCP-Paket passen, und machen die analoge Annahme auch für alle weiteren Neuverpackungen:



Wichtige Bestandteile des TCP-Headers sind:

- Die Portnummern der Quell- und Zielanwendung
- TCP-Flags
  Hierzu gehört z.B. das zur Gewährleistung der Auslieferung von TCP-Paketen benutzte ACK-Bit. Weil es bei allen Paketen einer Verbindung mit Ausnahme des initialen Pakets gesetzt ist, kann z.B. eine Firewall-Software an diesem Bit erkennen, ob ein von Außen eintreffendes Paket zur (unerwünschten) Verbindungsaufnahme dienen soll.

Das TCP-Paket wird weiter „nach unten“ durchgereicht zur IP-Schicht, die ihren eigenen Header ergänzt, der u.a. folgende Informationen enthält:

- Die IP-Adressen von Quell- und Zielrechner
- Typ des eingepackten Protokolls (z.B. TCP oder UDP)
- Time-To-Live (TTL)
  Beim Routing kann es zu Schleifen kommen. Damit ein Paket nicht ewig rotiert, startet es mit einer Time-To-Live - Angabe mit der maximalen Anzahl von erlaubten Router - Passagen, die von jedem Router dekrementiert wird. Muss ein Router den TTL-Wert auf Null setzen, verwirft er das Paket und informiert den Absender eventuell per ICMP-Paket über den Vorfall.

Wenn der nächste Router auf dem Weg zum Zielrechner über ein lokales Netzwerk mit Ethernet-Technik erreicht wird, muss das IP-Paket in einen Ethernet-Frame verpackt werden, wobei der Ethernet-Header z.B. die MAC-Adressen des Empfängers (hier: des Routers) und des Senders (hier: der Netzwerkkarte in Rechner A) aufnimmt.

Auf dem Rechner B wird der umgekehrte Weg durchlaufen: Jede Schicht entfernt ihren eigenen Header und reicht den Inhalt an die nächst höhere Ebene weiter, bis die übertragenen Daten schließlich in einem Eingabestrom der zuständigen Anwendung (identifiziert über die Portnummer im TCP-Header) gelandet sind.

14.1.3 Optionen zur Netzwerkprogrammierung in C#

C# (bzw. das .NET – Framework) 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 (MSIL) möglich. Entsprechende Dienste aus dem Windows-API können jedoch über die später zu behandelnde *PInvoke* - Technik genutzt werden.

Die in Abschnitt 14 zu behandelnden FCL-Klassen zur Netzwerkprogrammierung befinden sich meist in den Namensräumen **System.Net** und **System.Net.Sockets**, wobei ein Verweis auf das GAC - Assembly **System** erforderlich ist.

14.2 Internet - Ressourcen per Request/Response – Modell nutzen

Auf Internet-Ressourcen, die über einen so genannten **Uniform Resource Identifier** (**URI**) ansprechbar sind, kann man in C# genau so einfach zugreifen wie auf lokale Dateien.

Ein URI wie z.B.

http://www.egal.de:81/cgi-bin/beispiel/cgi.pl?vorname=Kurt

ist folgendermaßen aufgebaut:

Syntax:	*Protokoll*	**://**	*User:Pass@* (optional)	*Rechner*	**:***Port* (optional)	*Pfad*	**?***URL-Parameter* (optional)
Beispiel:	http	://		www.egal.de	:81	/cgi-bin/beispiel/cgi.pl	?vorname=Kurt

Die URL-Parameter dienen zur Anforderung von individuellen bzw. dynamisch erstellten Webseiten unter Verwendung der GET-Methode aus dem HTTP-Protokoll (siehe Abschnitt 14.2.3.3). 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.:

http://www.cs.tut.fi/~jkorpela/forms/methods.html#fund

14.2.1 Statische Webinhalte anfordern

Für den Zugriff auf eine per URI beschriebene Internet-Ressource bietet das .NET – Framework die **Request/Response – Architektur**. Man ruft zunächst die statische Methode **Create**() der Klasse **WebRequest** mit einem URI als Parameter auf, um ein Objekt aus einer zum Protokoll passenden **WebRequest** – Ableitung erzeugen zu lassen, z.B.:

```
WebRequest request = WebRequest.Create("http://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, 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; U; Windows NT 6.1; de; rv:1.9.2.13) " +
  "Gecko/20101203 Firefox/3.6.13 (.NET CLR 3.5.30729; .NET4.0E)";
```

Mit der **WebRequest** – Methode **GetResponse()** fordert man die Antwort des Servers an, z.B.

```
WebResponse response = request.GetResponse();
```

Das resultierende Objekt aus einer **WebResponse** – Unterklasse 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.:

```
Stream responseStream = response.GetResponseStream();
```

Zum Lesen einer Serverantwort mit bestimmtem Zeichensatz eignet sich ein entsprechend konfigurierter **StreamReader**:

```
StreamReader reader;
String cs = (((HttpWebResponse)response).CharacterSet).ToLower();
switch (cs) {
  case "iso-8859-1":
  case "utf-8": reader = new StreamReader(responseStream,Encoding.GetEncoding(cs));
              break;
  default:      reader = new StreamReader(responseStream,Encoding.ASCII);
              break;
}
```

Ein Aufruf der **StreamReader** – Methode **ReadLine()** liefert die nächste Zeile der Serverantwort:

```
String s;
while ((s = reader.ReadLine()) != null) {
    Console.WriteLine(s);
    Console.ReadLine();
}
```

Nach dem Lesen der Serverantwort sollte man die **WebResponse**-Methode **Close()** aufrufen, um die Ressourcen der Verbindung sofort frei zu geben:

```
response.Close();
```

Das folgende Programm zeigt die Schritte im Zusammenhang und verzichtet dabei der Einfachheit halber auf die bei Netzverbindungen sehr empfehlenswerte Ausnahmebehandlung:

```
using System;
using System.IO;
using System.Net;
using System.Text;
```

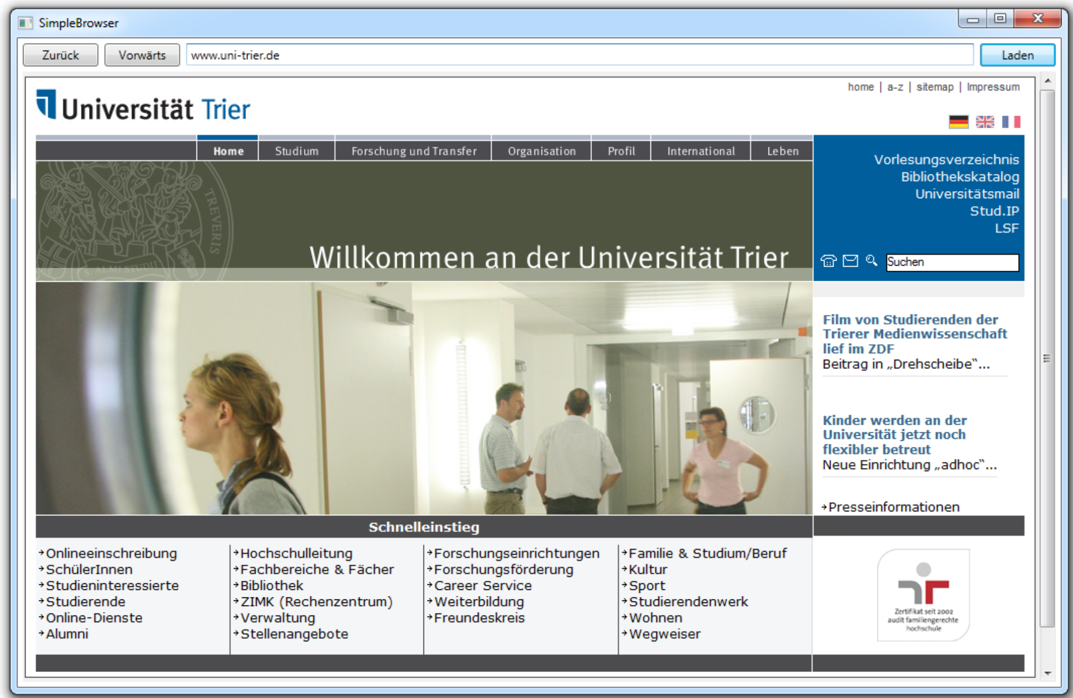
```
class RequestRespone {
   static void Main() {
      // Request-Objekt zu URI erzeugen
      WebRequest request = WebRequest.Create("http://www.uni-trier.de/");

      // Protokollspezifische Request-Konfiguration
      ((HttpWebRequest)request).UserAgent =
        "Mozilla/5.0 (Windows; U; Windows NT 6.1; de; rv:1.9.2.13) " +
        "Gecko/20101203 Firefox/3.6.13 (.NET CLR 3.5.30729; .NET4.0E)";

      // Response anfordern
      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 Inhhalt");
      Console.ReadLine();

      // Zugriff auf den Strom mit der Serverantwort
      Stream content = response.GetResponseStream();

      // StreamReader mit passender Kodierung erstellen
      StreamReader reader;
      String cs = (((HttpWebResponse)response).CharacterSet).ToLower();
      switch (cs) {
         case "iso-8859-1":
         case "utf-8": reader = new StreamReader(content, Encoding.GetEncoding(cs));
            break;
         default: reader = new StreamReader(content);
            break;
      }

      // Serverantwort zeilenweise ausgeben
      String s;
      while ((s = reader.ReadLine()) != null) {
         Console.WriteLine(s);
      }

      // Verbindung schließen
      response.Close();
   }
}
```

Ein Programmlauf (am 17.2.2011) bringt folgendes Ergebnis:

```
Letzte Änderung:        17.02.2011 18:11:40
Zeichensatz:            iso-8859-1

<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html
     PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="de" lang="de">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<!--
        This website is powered by TYPO3 - inspiring people to share!
   .  .  .
   .  .  .
```

Im .NET - Namensraum **System.Windows.Controls** befindet sich die Steuerelementklasse **Web-Browser** mit der Fähigkeit, HTML zu rendern. Sie sollen als Übungsaufgabe einen kleinen Web-

browser erstellen, der HTML-Seiten attraktiver anzeigen kann als die obige Konsolenanwendung, z.B.:



14.2.2 Datei-Download per HTTP - Protokoll

Per Request/Response – Technologie und HTTP - Protokoll kann man nicht nur HTML-Seiten von einem Server beziehen, sondern auch Binärdateien herunter laden. Wir nutzen zu diesem Zweck die Klasse **WebClient**, welche durch Verpacken der ohnehin schon recht bequemen Klassen **WebRequest** und **WebResponse** den Komfort auf die Spitze treibt:

```
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 WebClient();
         Console.WriteLine("Download startet");
         client.DownloadFile(args[0], args[1]);
         Console.WriteLine("Fertig!");
      } catch(Exception e) {
         Console.WriteLine(e.Message);
      }
   }
}
```

Man verwendet die Methode **DownloadFile()** und überlässt Routinearbeiten wie das Öffnen und Schließen von Dateien den FCL - Programmierern. Im ersten Parameter gibt man die Webadresse (den URI) an und im zweiten Parameter den lokalen Dateinamen. Das Beispielprogramm erwartet diese Informationen in Befehlszeilenargumenten.

Hier ist ein erfolgreicher Einsatz zu sehen:

```
filedownload http://www.uni-trier.de/fileadmin/urt/doku/spss/v18/SPSS18.zip spss18.zip

U:\Eigene Dateien\C#\Netzwerk>filedownload http://www.uni-trier.de/fileadmin/urt
/doku/spss/v18/SPSS18.zip spss18.zip
Download startet
Fertig!
```

Ist mehr Flexibilität gefragt, kommt die *direkte* Verwendung der Klassen **WebRequest** und **WebResponse** (bzw. der entsprechenden Ableitungen) in Frage. Das folgende Download-Programm wertet drei Befehlszeilenargumente aus:

- Webadresse der Quelldatei
- lokaler Dateiname
- Datum der letzten Änderung

Das Datum wird für die **HttpWebRequest**-Eigenschaft **IfModifiedSince** verwendet, so dass der Server eine Fehlermeldung liefert, wenn das Änderungsdatum der angeforderten Datei weiter zurück liegt. 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) {
      WebResponse response = null;
      FileStream fs = null;
      const int BUFSIZE = 4096;
      DateTime date;

      if (args.Length < 3) {
         Console.WriteLine("Aufruf: FileDownload <URI> <Dateiname> <jjjj.mm.tt>");
         Console.ReadLine();
         return;
      }

      try {
         date = new DateTime(Convert.ToInt32(args[2].Substring(0, 4)),
                             Convert.ToInt32(args[2].Substring(5, 2)),
                             Convert.ToInt32(args[2].Substring(8, 2)));
      } catch {
         Console.WriteLine("Datumsformat: jjjj.mm.tt");
         return;
      }

      try {
         WebRequest request = WebRequest.Create(args[0]);
         ((HttpWebRequest)request).IfModifiedSince = date;
         response = request.GetResponse();
      } catch (Exception e) {
         Console.WriteLine(e.Message);
         return;
      }
```

```
        try {
           Stream webStream = response.GetResponseStream();
           byte[] buffer = new byte[BUFSIZE];
           fs = new FileStream(args[1], FileMode.Create);
           int bytesRead;
           int sumRead = 0;
           Console.WriteLine("Download beginnt ...");
           while ((bytesRead = webStream.Read(buffer, 0, buffer.Length)) > 0) {
              fs.Write(buffer, 0, bytesRead);
              sumRead += bytesRead;
           }
           Console.WriteLine("Fertig: " + sumRead + " Bytes übertragen");
        } catch (Exception e) {
           Console.WriteLine(e.Message);
        } finally {
           response.Close();
           if(fs != null)
              fs.Close();
        }
     }
  }
```

Bei diesem Einsatz wurde die gesuchte Datei in einer neuen Version angetroffen:

```
filedownloadwr http://www.uni-trier.de/fileadmin/urt/doku/spss/v18/SPSS18.zip spss18.zip 2010.05.01

U:\Eigene Dateien\C#\Netzwerk>filedownloadwr http://www.uni-trier.de/fileadmin/u
rt/doku/spss/v18/SPSS18.zip spss18.zip 2010.05.01
Download beginnt ...
Fertig: 8899 Bytes übertragen
```

Hat sich die fragliche Datei seit dem genannten Datum nicht geändert, erhält man die Meldung:

```
Der Remoteserver hat einen Fehler zurückgegeben: (304) Nicht geändert.
```

Seit der .NET –Version 2.0 wird auch das FTP-Protokoll (von den Klassen **FtpWebRequest**, **FtpWebResponse** und **WebClient**) unterstützt, so dass ein Datei-Download von einem FTP-Server nun ebenfalls leicht zu bewerkstelligen ist.

14.2.3 Dynamische erstellte Webseiten per GET oder POST anfordern

14.2.3.1 Überblick

WWW-Server halten in der Regel nicht nur statische HTML-Seiten und sonstige Dateien bereit, 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). WWW-Nutzer äußern ihre Wünsche, indem sie per Browser (z.B. Mozilla-Firefox, MS Internet Explorer) eine Formularseite (mit Eingabeelementen wie Textfeldern, Kontrollkästchen usw.) ausfüllen und ihre Daten zum **WWW-Server** übertragen. Dieses Programm (z.B. Apache HTTP Server, MS Internet Information Server) analysiert und beantwortet Formulardaten aber nicht selbst, sondern überlässt diese Arbeit externen Anwendungen, die in unterschiedlichen Programmier- bzw. Skriptsprachen erstellt werden können (z.B. ASP.NET, Java, PHP oder Perl). Traditionell kooperieren WWW-Server und Ergänzungsprogramm über das so genannte **Common Gateway Interface** (**CGI**), wobei das Ergänzungsprogramm bei jeder Anforderung neu gestartet und nach dem Erstellen der HTML-Antwortseite wieder beendet wird. Mittlerweile werden jedoch Lösungen bevorzugt, 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. Außerdem wird bei den genannten Lösungen die nicht sehr

wartungsfreundliche Erstellung kompletter HTML-Antwortseiten über Ausgabeanweisungen der jeweiligen Programmiersprache vermieden. Stattdessen können in *einem* Dokument statische HTML-Abschnitte mit Bestandteilen der jeweiligen Programmiersprache zur dynamischen Produktion individueller Abschnitte kombiniert werden. Wir werden anschließend der Einfachheit halber alle Verfahren zur dynamischen Produktion individueller HTML-Seiten als *CGI - Lösungen* bezeichnen. Eine wichtige Gemeinsamkeit dieser Verfahren besteht darin, dass die Browser zur Formulierung ihrer Anforderungen (Requests) die Methoden **GET** und **POST** aus dem HTTP - Protokoll benutzen (siehe unten).

Wir beschränken uns darauf, CGI - Anwendungen durch *klientenseitige* C# - Programme (statt durch einen WWW-Browser) als zu nutzen. Wie zu Beginn von Abschnitt 14 verabredet, können wir uns aus Zeitgründen mit der relevanteren *Realisation* von CGI – Lösungen durch *serverseitige* C# - Programme (bisher per ASP.NET – Framework, neuerdings per *Windows Communication Foundation* (WCF)) *nicht* beschäftigen.

Es besteht eine Verwandtschaft zu den oben erwähnten *Webdiensten*, die allerdings keine HTML-Seiten für Browser produzieren, sondern XML - Dateien. Von dieser interessanten Technik zur Erstellung von verteiltem Anwendungen kann in diesem Kurs weder die server- noch die klientenseitige Programmierung behandelt werden.

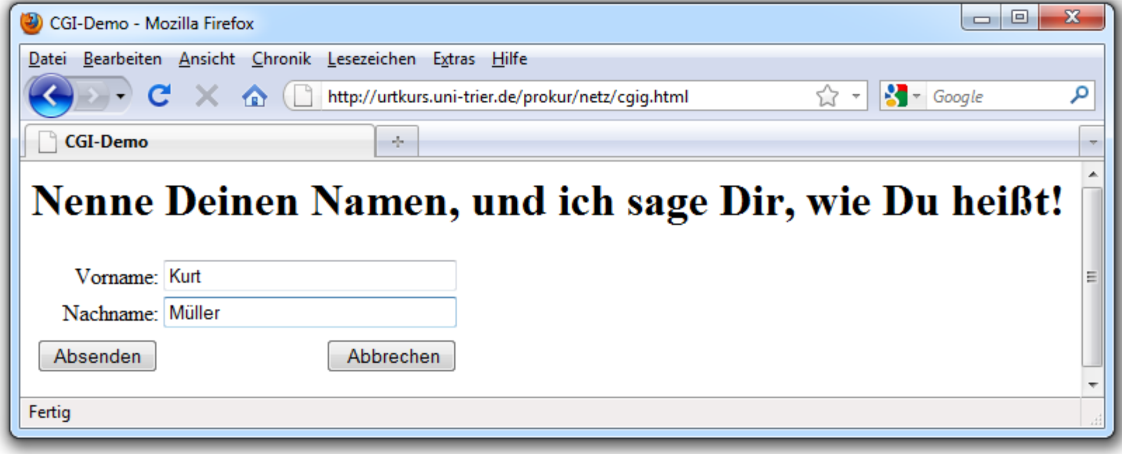
14.2.3.2 Arbeitsablauf

Beim CGI - Einsatz sind üblicherweise folgende Programme beteiligt:

- WWW-Browser
- WWW-Server
  In der Regel läuft dieses Programm auf einem fremden Rechner im Internet.
- CGI-Lösung
  Wir bezeichnen der Einfachheit halber jedes Verfahren zur dynamischen HTML-Produktion als *CGI - Lösung*.

Der Browser zeigt eine vom Server erhaltene HTML-Seite mit Formularelementen an, über die Benutzer eine CGI-Anfrage formulieren und abschicken können. Zur Erläuterung technischer Details betrachten wir ein sehr einfaches Formular, das ein in PHP realisiertes CGI-Skript auf einem WWW-Server an der Universität Trier anspricht.[1]

In diesem Browser-Fenster



[1] PHP ist eine auf das Erstellen von dynamischen Webseiten spezialisierte Programmiersprache.

ist die HTML-Seite zu sehen, die über den URI

http://urtkurs.uni-trier.de/prokur/netz/cgig.html

abrufbar ist und den folgenden HTML-Code mit 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>
```

Klickt der Benutzer nach dem Ausfüllen der Textfelder auf den Schalter **Absenden**, werden seine Eingaben mit der Syntax

Name=Wert

als Parameter für die CGI - Software zum WWW-Server übertragen. Zwei Felder werden jeweils durch ein &-Zeichen getrennt, so dass im obigen Beispiel mit den Feldern `vorname` und `nachname` (siehe HTML-Quelltext) folgende Sendung resultiert:

`vorname=Kurt&nachname=M%FCller`

Den Umlaut „ü" kodiert der Browser automatisch durch ein einleitendes Prozentzeichen und seinen Hexadezimalwert im Zeichensatz der Webseite. Wenn im HTML-Code kein Zeichensatz angegeben ist (wie im Beispiel), stellt der Browser eine Vermutung an (im Beispiel mit dem Ergebnis: ISO 8859-1). Analog werden andere Zeichen behandelt, die nicht zum Standard - ASCII-Code gehören. Weitere Regeln dieser so genannten **URL-Kodierung**:

- Leerzeichen werden durch ein „+" ersetzt.
- Für die mit einer speziellen Bedeutung belasteten Zeichen (also &, +, = und %) erscheint nach einem einleitenden Prozentzeichen ihr Hexadezimalwert im Zeichensatz.

Auf gleich noch näher zu erläuternde Weise übergibt der WWW-Server die Formulardaten an das im **action**-Attribut der Formulardefinition angegebene externe Programm oder Skript. Im Beispiel handelt es sich um folgendes **PHP**-Skript, das wenig kreativ aus den übergebenen Namen einen Gruß formuliert:

```
<?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>";
?>
```

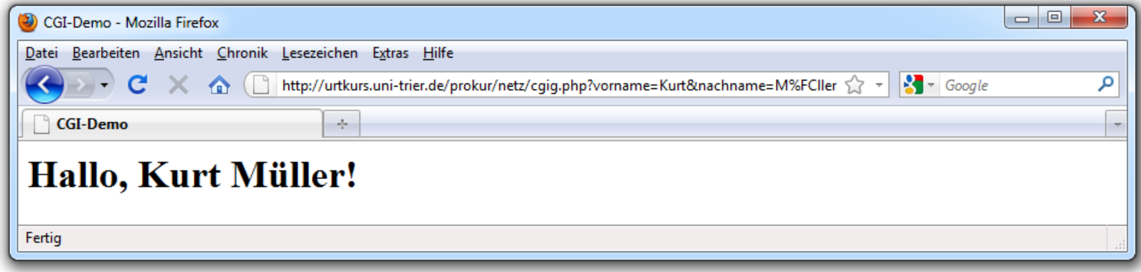
Im Skript wird die auszugebende HTML-Seite über **echo**-Kommandos an die Standardausgabe geschickt, und der WWW-Server befördert die PHP-Produktion über das HTTP-Protokoll an den Browser, der den empfangenen HTML-Quellcode

Quelltext von: http://urtkurs.uni-trier.de/prokur/netz/cgig.php?vorname=Kurt&nachname=M%FCller - Mozilla Firefox

Datei Bearbeiten Ansicht Hilfe

```
<html>
<head><title>CGI-Demo</title>
</head>
<body>
<h1>Hallo, Kurt Müller!</h1>
</body>
</html>
```

anzeigt:



14.2.3.3 GET

Zum Versandt von CGI-Parametern an einen WWW-Server kennt das HTTP-Protokoll zwei Verfahren (**GET** und **POST**), die nun vorgestellt und dabei auch gleich in C#- Programmen realisiert werden sollen.

Bei der GET-Technik, die man im **form** - Tag einer HTML-Seite durch die Angabe

```
method="get"
```

wählt, schickt der Browser die Name-Wert - Paare als URI-Bestandteil hinter einem trennenden Fragezeichen an den WWW-Server. Weil nach Eintreffen der Antwortseite die zugrunde liegende Anforderung in der Adresszeile des Browsers erscheint, kann die GET-Syntax dort inspiziert werden (siehe oben).

Aus der Integration der HTTP-Parameter in die Anforderung an den WWW-Server ergibt sich eine Längenbeschränkung, wobei die konkreten Maximalwerte vom Server und vom Browser abhängen. Man sollte vorsichtshalber eine Anforderungsgesamtlänge von 255 Zeichen einhalten und ggf. das POST-Verfahren verwenden, das keine praxisrelevante Längenbeschränkung kennt.

Der WWW-Server schreibt die HTTP-Parameter in eine Umgebungsvariable namens **QUERY_STRING** und stellt auf analoge Weise der CGI-Software gleich noch weitere Informationen zur Verfügung, z.B.:

```
QUERY_STRING="vorname=Kurt&nachname=M%3Fller"
REMOTE_PORT="1211"
REQUEST_METHOD="GET"
```

In obigem PHP-Skript erfolgt der der Zugriff auf die Parameter in der Umgebungsvariablen **QUERY_STRING** über den superglobalen Array **$_GET**.

Um in C# eine CGI-Software anzusprechen, die per GET mit Parametern versorgt werden möchte, genügt ein Objekt der komfortablen Klasse **WebClient**, das in seiner Methode **DownloadData()** den Job erledigt:

```
using System;
using System.Net;
using System.Text;
using System.Web;

class CgiGet {
   static void Main() {
      String cgiuri = "http://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(), Encoding.Default);

      Console.WriteLine("Geben Sie bitte Ihren Nachnamen an");
      Console.Write("Nachname: ");
      String nachName = HttpUtility.UrlEncode(Console.ReadLine(), Encoding.Default);
      Console.WriteLine("\nDie Anforderung wird übertragen ...");

      WebClient client = new WebClient();
      byte[] antwort = null;
      try {
         antwort = client.DownloadData(cgiuri + "?vorname=" + vorName +
            "&nachname=" + nachName);
         String s = Encoding.Default.GetString(antwort);
         Console.WriteLine("\nAntwort:\n" + s);
      } catch (Exception e) {
         Console.WriteLine(e.Message);
      }
   }
}
```

Mit der statischen **HttpUtility**-Methode **UrlEncode()** wird für die korrekte URL-Kodierung der Umlaute etc. gesorgt (siehe Abschnitt 14.2.3.2). Weil die Klasse **HttpUtility** (aus dem Namensraum **System.Web**) im GAC-Assembly **System.Web** implementiert ist, benötigt der Compiler einen entsprechenden Verweis. Für die korrekte Interpretation der via Konsole erhaltenen Zeichenfolgen sorgt ein **Encoding**-Objekt (angesprochen über die statische Eigenschaft **Default** der Klasse **Encoding**).

Dasselbe **Encoding**-Objekt wandelt in seiner **GetString()**-Methode den vom Webserver gelieferten Byte-Array in einen **String**. Statt somit eine zum lokalen System kompatible Kodierung ungeprüft anzunehmen, sollte man bei einer unbekannten CGI-Lösung die (hoffentlich vorhandenen) Zeichensatz - Metainformationen der Serverantwort auswerten, z.B. über die **WebClient**-Eigenschaft **ResponseHeaders**, die ein Objekt der Klasse **WebHeaderCollection** anspricht.

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:
<html>
<head><title>CGI-Demo</title>
</head>
<body>
<h1>Hallo, Kurt Müller!</h1>
</body>
</html>
```

14.2.3.4 POST

Beim **POST**-Verfahren, das man im **form** - Tag einer HTML-Seite durch die Angabe

```
method="post"
```

wählt, werden die Parameter (im selben Format wie beim GET-Verfahren) mit Hilfe des WWW-Servers zur *Standardeingabe* der CGI-Software übertragen. Was genau gemäß HTTP-Protokoll zu tun ist, braucht C# - Programmierer nicht zu interessieren. Man schreibt die HTTP-Parameter über **Add**()-Aufrufe in ein Objekt der Klasse **NameValueCollection**

```
NameValueCollection formData = new NameValueCollection();
formData.Add("vorname",vorName);
formData.Add("nachname",nachName);
```

und verwendet dieses Objekt neben dem statischen URI-Anteil als Parameter für die **WebClient**-Instanzmethode **UploadValues**():

```
WebClient client = new WebClient();
byte[] antwort = null;
antwort = client.UploadValues(cgiuri, formData);
```

Ein Objekt der Klasse **Encoding.UTF8** hat sich dabei bewährt, die von **UploadValues()** als Rückgabe gelieferte Server-Antwort in seiner **GetString()**-Methode korrekt in ein **String**-Objekt zu wandeln:[1]

```
String s = Encoding.UTF8.GetString(antwort);
```

Insgesamt unterscheidet sich das POST-Beispielprogramm nur wenig von der GET-Variante (siehe Abschnitt 14.2.3.3):

```
using System;
using System.Collections.Specialized;
using System.Net;
using System.Text;
using System.Web;

class CgiPost {
   static void Main() {
      String cgiuri = "http://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();

      NameValueCollection formData = new NameValueCollection();
      formData.Add("vorname", vorName);
      formData.Add("nachname", nachName);

      Console.WriteLine("\nDie Anforderung wird übertragen ...");
      WebClient client = new WebClient();
      byte[] antwort = null;
```

[1] Die Bewährungsprobe bestand aus einem Nachnamen mit Umlaut (*Müller*). Während im GET-Beispielprogramm (siehe Abschnitt 14.2.3.3), das denselben Nachnamen URL-kodiert versandt hatte, die Server-Antwort per **Encoding.Default** in ein ansehnliches **String**-Objekt konnte, gelang dies im POST-Beispielprogramm per **Encoding.UTF8**. Eigentlich sollten die beteiligten (De-)Kodierungen auf dem Weg vom Klienten zum Server und zurück näher analysiert werden.

```
        try {
           antwort = client.UploadValues(cgiuri, formData);
           String s = Encoding.UTF8.GetString(antwort);
           Console.WriteLine("\nAntwort:\n" + s);
        } catch (Exception e) {
           Console.WriteLine(e.Message);
        }
     }
  }
```

Das angesprochene PHP-Skript unterscheidet sich ebenfalls kaum von der GET-Variante: Anstelle des superglobalen Arrays **$_GET** ist der analoge Array **$_POST** zu verwenden.

14.3 IP-Adressen bzw. Host-Namen ermitteln

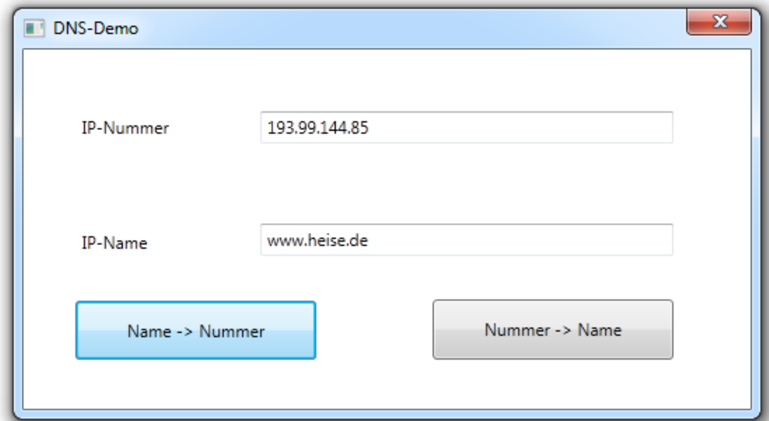
Jeder an das Internet angeschlossen Rechner verfügt über (mindestens) eine **IP-Adresse** (32-bittig in IPv4, 128-bittig in IPv6) sowie über einen **Host-Namen**, wobei die Zuordnung vom **Domain Name System** (**DNS**) geleistet wird.

Die statische Methode **GetHostEntry**() der FCL-Klasse **Dns** (im Namensraum **System.NET**) mit den beiden Überladungen

public static IPHostEntry GetHostEntry(IPAddress *adresse*)
public static IPHostEntry GetHostEntry(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 Host-Namen bzw. den Namen zu einer Adresse) zu ermitteln. Man erhält ein **IPHostEntry**-Objekt, das in seinen Eigenschaften **AddressList** bzw. **HostName** die gewünschten Daten bereit hält.

Im folgenden Programm werden beide Konvertierungsrichtungen verwendet:



Aufgrund der graphischen Oberfläche ist der Quelltext deutlich länger als bei den bisherigen Beispielprogrammen in Abschnitt 14. Daher werden nur die Ereignisbehandlungsmethoden zu den beiden Schaltflächen wiedergegeben:

```
private void cbToNumber_Click(object sender, RoutedEventArgs e) {
   if (tbName.Text.Length == 0)
      return;
   try {
      IPHostEntry host = Dns.GetHostEntry(tbName.Text);
      tbNumber.Text = host.AddressList[0].ToString();
   } catch (Exception ex) {
      MessageBox.Show(ex.ToString(), "Fehler");
   }
}
```

```
private void cbToName_Click(object sender, RoutedEventArgs e) {
   if (tbNumber.Text.Length == 0)
      return;
   try {
      IPHostEntry host = Dns.GetHostEntry(tbNumber.Text);
      tbName.Text = host.HostName;
   } catch (Exception ex) {
      MessageBox.Show(ex.ToString(), "Fehler");
   }
}
```

Mit den **Dns**-Methoden **BeginGetHostEntry()** , und **EndGetHostEntry()** lässt sich eine DNS-Anfrage asynchron durchführen (vgl. Abschnitt 13.6).

Im .NET – Framework nimmt man für jeden Rechner eine ganze Liste von IP-Adressen an, was bei der heutigen Vielfalt von Netzwerkadaptern in Standard-PCs (z.B. LAN, VPN) und in Anbetracht der bei Server-PCs oft mehrfach vorhandenen Ethernet-Anschlüsse durchaus realistisch ist. Das obige Programm zeigt nur die erste IP-Adresse an:

```
tbNumber.Text = host.AddressList[0].ToString();
```

Den vollständigen (mit Visual Studio - Unterstützung erstellten) Quellcode finden Sie im Ordner

...\BspUeb\Netzwerk\DNS

14.4 Socket-Programmierung

Unsere bisherigen Beispielprogramme im Kapitel 14 haben hauptsächlich WWW-Inhalte von Servern bezogen und dazu FCL-Klassen benutzt, die ein fest „verdrahtetes“ Anwendungsprotokoll (meist HTTP) realisieren. 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 14.1.1) 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 im .NET - Framework durch Klassen im Namensraum **System.Net.Sockets** realisiert wird.

TCP-Programmierer müssen sich nicht um IP-Pakete kümmern, weder die Zustellung noch die Integrität oder die korrekte Reihenfolge überwachen, sondern (nach den Regeln eines Protokolls der Anwendungsschicht) Bytes in einen Ausgabestrom einspeisen bzw. aus einen Eingabestrom entnehmen und interpretieren. Dabei ist der Ausgabestrom des Senders virtuell mit dem Eingabestrom des Empfängers verbunden.

Wir beschäftigen uns in diesem Abschnitt mit der Erstellung von Klienten- *und* Serveranwendungen. Ein wesentlicher Unterschied zwischen 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 zugewiesenen Port benutzt.

14.4.1 TCP-Server

Wir erstellen einen Server, der am TCP-Port 13000 lauscht und anfragenden Klienten die aktuelle Tageszeit mitteilt. Im Konstruktoraufruf für das zentrale Objekt der Klasse **TcpListener** geben wir neben der Portnummer auch eine IP-Adresse an, z.B.

```
TcpListener server = null;
int svrPort = 13000;
IPAddress ip = Dns.GetHostEntry("localhost").AddressList[1];
   . . .
server = new TcpListener(ip, svrPort);
```

Vom veralteten Konstruktoraufruf *ohne* IP-Adresse und der damit erforderlichen automatischen Wahl einer IP-Adresse aus der Liste lokal verfügbarere 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 liefert über das **AddressList**-Element mit dem Index 0 die IPv6-Adresse und über das Element mit dem Index 1 die IPv4-Adresse.

Nach dem Start des Servers

```
server.Start();
```

wird dieser durch Aufruf seiner Methode **AcceptTcpClient()** beauftragt, auf eine Verbindungsanfrage zu lauern:

```
tcpClient = server.AcceptTcpClient();
```

Während der Wartezeit ist der aktuelle Thread durch den **AcceptTcpClient()**-Aufruf blockiert. Dieser endet erst bei einer Verbindungsanfrage und liefert dann ein (zum Senden und Empfangen geeignetes) **TcpClient**-Objekt zurück, das als Verpackung für die beiden folgenden (sicher von Ihnen erwarteten) Objekte dient:

- ein **Socket**-Objekt, ansprechbar über die **TcpClient** - Eigenschaft **Client**
- ein **NetworkStream**-Objekt, erreichbar über die **TcpClient**-Methode **GetStream()**

Im Beispielprogramm wird anschließend eine Zeichenfolge mit Datum und Uhrzeit unter Verwendung der ASCII-Kodierung in einen Byte-Array gewandelt und in dieser Form über das **NetworkStream**-Objekt an die Gegenstelle gesendet. Bei einer realen Server-Anwendung ist für den Informationsaustausch ein Anwendungsebenen-Protokoll zu verabreden oder zu beachten, und das Senden von Daten sollte durch einen **StreamWriter** erleichtert werden.

Am Ende der Klientenbedienung richtet man einen **Close()**-Methode an das **TcpClient** – Objekt,

```
tcpClient.Close();
```

um die folgenden Abschlussarbeiten auszulösen:

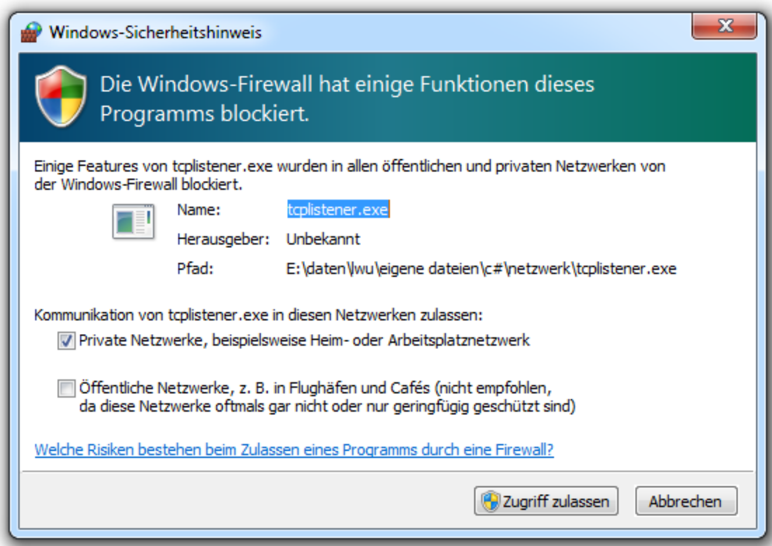
- Das interne **NetworkStream** - Objekt erhält einen **Dispose()** – Aufruf.
- Das interne **Socket** - Objekt wird aufgefordert, noch anstehenden Übertragungen auszuführen.
- Das interne **Socket** - Objekt erhält einen **Close()** – Aufruf, und die Referenz auf dieses Objekt wird auf **null** gesetzt. Dies beendet die Verbindung zum Klienten und gibt die Netzwerk-Ressourcen (z.B. den belegten Port) an das Betriebsystem zurück. Die explizite Rückgabe von Ressourcen ist vor allem dann relevant, wenn ein Programm anschließend weiter aktiv bleiben soll. Bei Beendigung eines Programms werden die belegten Ressourcen automatisch frei gegeben.

Das Programm bedient in einer **while**-Schleife beliebig viele Klienten nacheinander:

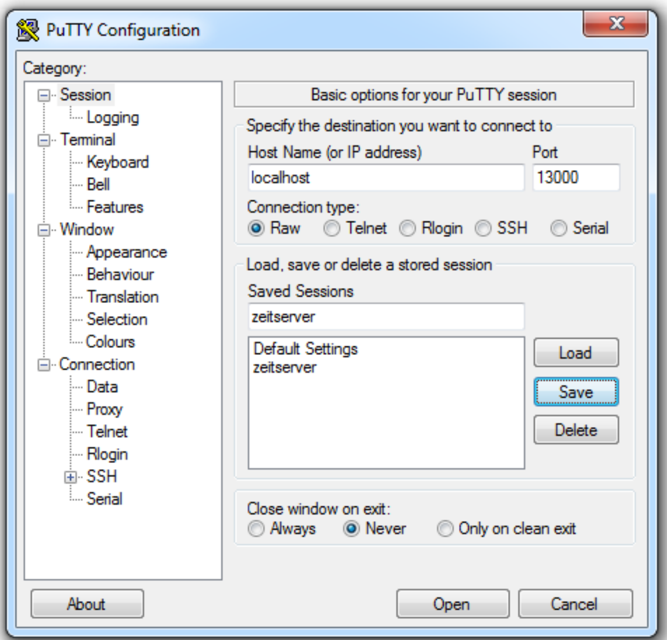
```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class TcpListenerDemo {
   static void Main() {
      TcpListener server = null;
      int svrPort = 13000;
      IPAddress ip = Dns.GetHostEntry("localhost").AddressList[1];
      TcpClient tcpClient = null;
      NetworkStream stream = null;

      try {
         server = new TcpListener(ip, svrPort);
         server.Start();
         Console.WriteLine("Zeitserver lauscht seit " + DateTime.Now + " (IPv4: " +
              ip + ", Port: " + svrPort + ")");
         while (true) {
            tcpClient = server.AcceptTcpClient();// Achtung: Aufruf blockiert!
            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);
            tcpClient.Close();
         }
      } catch (Exception e) {
         Console.WriteLine(e);
      }
   }
}
```

Der Kürze halber verzichten wir auf ein Verfahren zum regulären Beenden des Programms, so dass es (samt **TcpListener**) rabiat abgebrochen werden muss.

Sobald *nicht* die interne IP-Adresse (127.0.0.1 bei IPv4 bzw. ::1 bei IPv6, verbunden mit dem Host-Namen *localhost*) verwendet wird, ist für den Betrieb des Zeit-Servers eine Firewall-Ausnahme erforderlich:



Der Zeit-Server kann z.B. per über das kostenlose Terminalprogramm **PuTTY**[1] angesprochen werden. Wenn Server und Klient auf dem lokalen Rechner laufen, eignet sich die folgende Sitzungskonfiguration:



Der Server antwortet:

```
PuTTY (inactive)
18.02.2011 01:11:22
```

und protokolliert seine Dienstleistungen:

```
U:\Eigene Dateien\C#\Netzwerk\TcpListener.exe
Zeitserver lauscht seit 13.10.2011 02:49:14 (IPv4: 127.0.0.1, Port: 13000)

13.10.2011 02:49:42 Anfrage von
  IP-Nummer:     127.0.0.1
  Port:          54476

13.10.2011 02:51:20 Anfrage von
  IP-Nummer:     127.0.0.1
  Port:          54572

13.10.2011 02:52:00 Anfrage von
  IP-Nummer:     127.0.0.1
  Port:          54610
```

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.

[1] http://www.putty.org/

14.4.2 TCP-Klient

Als Gegenstück zum eben präsentierten Zeit-Server wird nun ein passendes Klienten-Programm entwickelt, wobei ein Objekt der 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 = 13000;
  . . .
TcpClient tcpClient = new TcpClient(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 Daten im angesprochenen Netzwerkstrom ankommen. Seit der .NET - Version 2.0 besteht die Möglichkeit, über die **NetworkStream** - Eigenschaft **ReadTimeout** eine Zeitspanne in Millisekunden festzulegen, nach deren Ablauf der Leseversuch mit einer **IOExecption** abgebrochen werden soll.

Bei der Wandlung von Bytes in Unicode-Zeichen ist die korrekte Kodierung zu verwenden, z.B.:

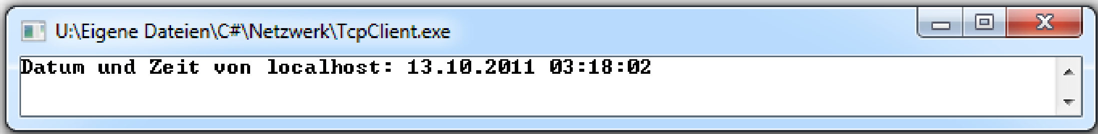
```
string sZeit = Encoding.ASCII.GetString(bZeit, 0, nRead);
```

Welche Abschlussarbeiten ein **TcpClient** – Objekt nach einem **Close()** – Aufruf erledigt, wurde schon in Abschnitt 14.4.1 erläutert.

Der Quellcode im Überblick:

```
using System;
using System.Net.Sockets;
using System.Text;

class TcpClientDemo {
   static void Main() {
      String svr = "localhost";
      int port = 13000;
      TcpClient tcpClient = null;
      try {
         tcpClient = new TcpClient(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);
      } finally {
         // TcpClient samt Socket und NetworkStream schließen
         if (tcpClient != null)
            tcpClient.Close();
      }
   }
}
```

Sofern Netz und Server mitspielen, liefert das Programm Datum und Uhrzeit, z.B.:



14.4.3 Simultane Bedienung mehrerer Klienten

Bei einer ernsthaften Server-Programmierung kommt man an einer **Multithreading** - Lösung nicht vorbei, damit mehrere Klienten simultan bedient werden können. Als Beispiel erstellen wir einen Echo-Server, der alle zugesandten Bytes unverändert zurück schickt. Den „Service“ für einen Klienten übernimmt ein Objekt der Klasse `EchoHandler`, das in seiner `Run()`-Methode aus dem Netzwerkstrom liest und auch dorthin schreibt:

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

public class EchoHandler {
   TcpClient tcpClient;
   ThreadPoolEchoServer server;
   int clientID;
   NetworkStream stream;
   public EchoHandler(ThreadPoolEchoServer server_,TcpClient tcpClient_, int clientID_) {
      server = server_;
      tcpClient = tcpClient_;
      clientID = clientID_;
      stream = tcpClient.GetStream();
      stream.ReadTimeout = 10000;
   }
   void Close() {
      tcpClient.Close();
      server.CheckOut(clientID);
   }
   public void Run(object obj) {
      int n;
      byte[] buffer = new byte[80];
      try {
         while (true) {
            n = stream.Read(buffer, 0, buffer.Length);
            stream.Write(buffer, 0, n);
         }
      } catch {
      } finally {
         Close();
      }
   }
}
```

Wenn ein Klient länger als 10 Sekunden stumm bleibt, läuft die **ReadTimeout** - Zeitspanne des beteiligten **NetworkStream**-Objekts ab:

```
stream.ReadTimeout = 10000;
```

In diesem Fall endet die `Run()` - Methode nach einigen Aufräumungsarbeiten in der per **finally**-Block aufgerufenen `Close()` - Methode.

Das Dienstleistungszentrum wird durch ein Objekt der Klasse `ThreadPoolEchoServer` realisiert. Seine `Run()` - Methode wird vom primären Thread ausgeführt:

```
public void Run() {
   try {
      while (true) {
         TcpClient tcpClient = tcpListener.AcceptTcpClient();
         lock (this) {
            nc++;
            nr++;
            Console.WriteLine("\n" + DateTime.Now + "  Klient " +
               nr + " akzeptiert. Aktiv: " + nc + "\n  IP-Nummer:\t" +
               (tcpClient.Client.RemoteEndPoint as IPEndPoint).Address + "\n  Port:  \t" +
               (tcpClient.Client.RemoteEndPoint as IPEndPoint).Port);
         }
         EchoHandler echoHandler = new EchoHandler(this, tcpClient, nr);
         ThreadPool.QueueUserWorkItem(echoHandler.Run);
      }
   } catch (Exception e) {
      lock (this) {
         Console.WriteLine(e);
      }
   }
}
```

Sobald der Lauscher vom Typ **TcpListener** eine Klientenanfrage feststellt, wird ein `EchoHandler` - Objekt erzeugt, das im Konstruktoraufruf folgende Informationen erhält:

- eine Referenz auf das Dienstleistungszentrum
  So kann der `EchoHandler` die `ThreadPoolEchoServer` - Instanzmethode `CheckOut()` aufrufen, um das Ende einer Klientenversorgung zu melden.
- eine Referenz auf das **TcpClient** - Objekt, das von der **TcpListener** - Methode **AcceptTcp-Client()** erzeugt worden ist
- eine Verbindungsnummer vom Typ **int**

Nach dem Erstellen des **EchoHandler** - Objekts wird seine `Run()`-Methode als Arbeitsauftrag in die Warteschlange des Threadpools eingereiht (vgl. Abschnitt 13.6).

In der folgenden Situation bedient der `ThreadPoolEchoServer` drei Klienten, wobei die Klasse `Dauersender` zum Einsatz kommt, die Sie als Übungsaufgabe erstellen sollen (siehe Abschnitt 14.5). `Dauersender`-Objekte versenden pro Sekunde eine zufällige Ziffernsequenz und protokollieren die Server-Antwort.

```
MultiClientEchoServer.exe
U:\Eigene Dateien\C#\Netzwerk>MultiClientEchoServer.exe
18.02.2011 02:15:23  MultiClientEchoServer gestartet

18.02.2011 02:17:52  Klient 1 akzeptiert. Aktiv: 1
  IP-Nummer:     127.0.0.1
  Port:          51336

18.02.2011 02:18:04  Klient 1 ausgeschieden. Noch aktiv: 0

18.02.2011 02:18:44  Klient 2 akzeptiert. Aktiv: 1
  IP-Nummer:     127.0.0.1
  Port:          51380

18.02.2011 02:19:30  Klient 3 akzeptiert. Aktiv: 2
  IP-Nummer:     127.0.0.1
  Port:          51418

18.02.2011 02:19:31  Klient 3 ausgeschieden. Noch aktiv: 1

18.02.2011 02:19:51  Klient 4 akzeptiert. Aktiv: 2
  IP-Nummer:     127.0.0.1
  Port:          51438

18.02.2011 02:20:01  Klient 5 akzeptiert. Aktiv: 3
```

```
Sendung 247: 1796858341    Echo: 1796858341
Sendung 248: 8022558268    Echo: 8022558268
Sendung 249: 3386055634    Echo: 3386055634
Sendung 250: 6948725809    Echo: 6948725809
Sendung 251: 7410677005    Echo: 7410677005
Sendung 252: 5389078353    Echo: 5389078353
Sendung 253: 9290486442
```

Das komplette Projekt ist im folgenden Ordner zu finden

...\BspUeb\Netzwerk\MultiClientEchoServer\ThreadPoolEchoServer

14.5 Übungsaufgaben zu Kapitel 14

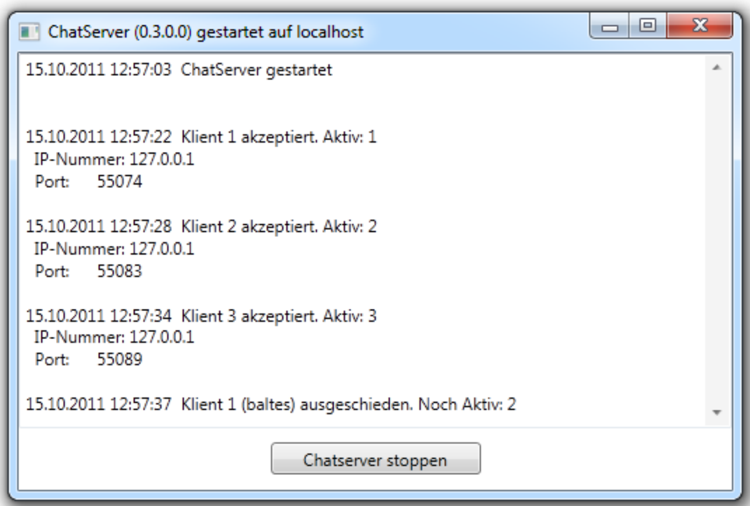
1) Erstellen Sie mit Hilfe der Steuerelementklasse **WebBrowser** (im Namensraum **System.Windows.Controls**) eine WPF-Anwendung, die eine **TextBox** zur Eingabe einer Webadresse bietet, HTML anzeigt, sinnvoll auf Änderungen der Formulargröße reagiert und elementare Navigationsmöglichkeiten bietet (siehe Bildschirmphoto in Abschnitt 14.2.1).

2) Erstellen Sie einen TCP-Klienten, der den in Abschnitt 14.4.3 vorgestellten Multithread-Echoserver durch das regelmäßige Absenden von jeweils zehn zufällig gewählten Ziffern beschäftigt und die Server-Antworten protokolliert.

3) Erstellen Sie eine ChatRoom - Anwendung mit Server- und Klientenprogramm. Die Serveranwendung sollte …

- an einem TCP-Port auf Verbindungswünsche warten,
- mehrere Klienten simultan bedienen (jeden Klienten in einem eigenen Thread),
- Sendungen eines Klienten an alle aktiven Klienten übertragen,
- wichtige Ereignisse (z.B. An- und Abmeldungen von Klienten) protokollieren, z.B. mit einem mehrzeiligen **TextBox**-Steuerelement,
- ein **Button**-Steuerelement anbieten zum Beenden des Programms, wobei die aktiven Klienten über das Dienstende zu benachrichtigen sind,

Ein typisches Server-Verlaufsprotokoll könnte so aussehen:

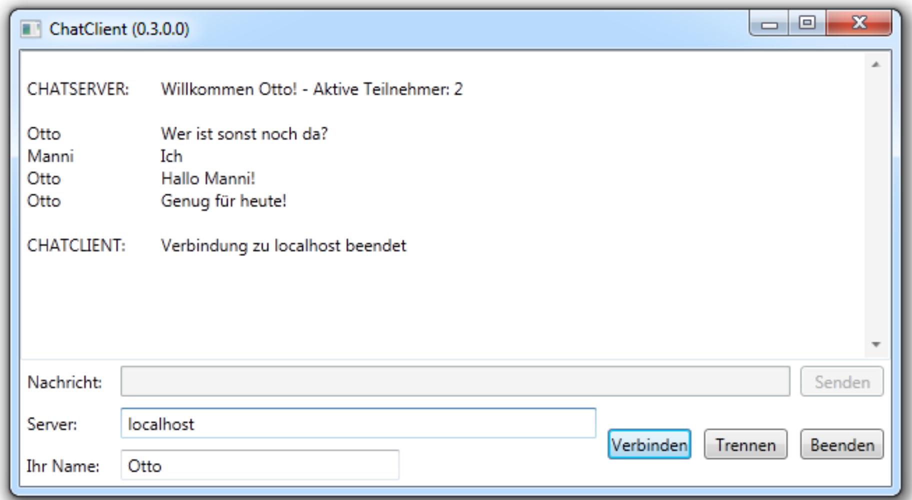


Die Klientenanwendung sollte …

- **TextBox**-Steuerelemente anbieten für:
  - den Servernamen
  - den Wunschnamen des Chat-Teilnehmers
  - die abzuschickenden Nachrichten
- **Button**-Steuerelemente bieten …
  - zum An- und Abmelden beim Server
  - zum Abschicken der Nachricht
  - zum Beenden des Programms
- zum gewünschten Server eine Verbindung mit dem gewünschten Benutzernamen aufbauen
- in einem speziellen Thread ständig für eingehende Nachrichten empfangsbereit sein

- eingehende Nachrichten protokollieren, z.B. mit einem mehrzeiligen **TextBox**-Steuerelement

Die Benutzeroberfläche der Klientenanwendung sollte ungefähr so aussehen:



Im ChatRoom - Anwendungsprotokoll sollen folgende Nachrichten eine spezielle Bedeutung haben:

#Quit!	Ein Klient meldet sich ab.
#Bye!	Serverantwort auf die Abmeldung eines Klienten
#UserName='*name*'!	Ein Klient nennt seinen Wunschnamen.
#EndOfService!	Der Server informiert die Klienten über den bevorstehenden Dienstschluss.

Die Benutzbarkeit des Programms kann durch folgende Maßnahmen gesteigert werden:

- Je nach Programmzustand sollten die gerade nicht verwendbaren Bedienelemente gesperrt sein (z.B. die Nachrichten - **TextBox** bei fehlender Verbindung).
- Mit der **UIElement** - Methode **Focus()** setzt man den Eingabefokus auf die gerade benötigte **TextBox**, z.B. nach dem Verbindungsaufbau auf die Nachrichten - **TextBox**.
- Über die **Button**-Eigenschaft **IsDefault** verbindet man die **Enter**-Taste mit der gerade am ehesten benötigten Schaltfläche.

15 Datenbindung

In den meisten WPF-Anwendungen dürfte 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 weitaus größere Rolle spielen als 3D-Grafiken und Animationen. Erfreulicherweise bietet die WPF gerade beim Thema Datenbindung eine exzellente Unterstützung für die Entwickler.

15.1 Quelle und Ziel einer Datenbindung

Als **Ziel** einer Datenbindung kommt jedes Objekt einer von **System.Windows.DependencyProperty** abstammenden Klasse in Frage. Ein Beispiel ist das Feld **TextProperty** der Steuerelementklasse **TextBox**-Feld, das in der Öffentlichkeit über die **TextBox**-Eigenschaft **Text** ansprechbar ist.

Als **Quelle** einer Datenbindung kann jede öffentliche Eigenschaft eines Objekts fungieren. In einfachen Fällen treten gewöhnliche CLR-Objekte oder Steuerelemente auf. In einer Anwendung zur Anzeige und Verwaltung großer Datenbestände kommen Quellobjekte aus der ADO.NET – Klasse **DataSet** zum Einsatz (siehe Abschnitt 20.6.2.3).

Wir beschränken und im Kapitel 15 meist auf Objekte als Datenquellen und verwenden in einem Beispielprogramm die folgende Klasse `Person`, die im Wesentlichen aus Sells & Griffiths (2007, S. 171) stammt:

```
using System;
using System.ComponentModel;

namespace DatenbindungObjekt {
   class Person : INotifyPropertyChanged {

      public event PropertyChangedEventHandler PropertyChanged;

      protected void ValueChanged(String property) {
         if (PropertyChanged != null ) {
            PropertyChanged(this, new PropertyChangedEventArgs(property));
         }
      }

      String name;
      public String Name {
         get { return name; }
         set {
            if (name == value)
               return;
            else {
               name = value;
               ValueChanged("Name");
            }
         }
      }
```

```
        int  alter;
        public int Alter {
           get { return alter; }
           set {
              if (alter == value)
                 return;
              if (0 < value  &&  value < 130) {
                 alter = value;
                 ValueChanged("Alter");
              }
           }
        }
      }
   }
}
```

Damit Quellobjekte der Klasse `Person` im Rahmen der WPF-Datenbindungstechnik Eigenschaftsänderungen an verbundene Zielsteuerelemente melden können, implementiert die Klasse das Interface **INotifyPropertyChanged**. Somit ist sie verpflichtet, das Ereignis **PropertyChanged** anzubieten. Es wird in der Methode `ValueChanged()` ausgelöst, die in den **set**-Klauseln der beiden Eigenschaften `Name` und `Alter` zum Einsatz kommt.

Im Beispielprogramm soll ein generisches Kollektionsobjekt zur Verwaltung von Personen zum Einsatz kommen. Um Problemen mit der Syntax für generische Klassen im XAML-Kontext aus dem Weg zu gehen, definieren wir nach dem Vorschlag von Sells & Griffiths (2007, S. 201) einen Aliasnamen für die Klasse **List**<`Person`>:

```
class Personen : List<Person> { }
```

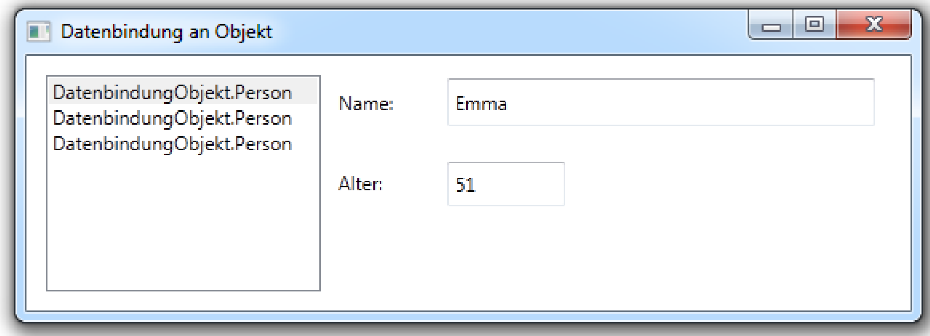
Um per XAML ein `Personen`-Objekt anzulegen, zu bevölkern und der Anwendung zur Verfügung zu stellen, deklarieren wir in der Datei **App.xaml** einen Namensraum mit Präfix `dbo` und eine Anwendungs-Ressource mit dem Namen `Gruppe`:

```
<Application x:Class="DatenbindungObjekt.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:dbo="clr-namespace:DatenbindungObjekt"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
        <dbo:Personen x:Key="Gruppe">
            <dbo:Person Name="Emma" Alter="51" />
            <dbo:Person Name="Emanuel" Alter="23" />
            <dbo:Person Name="Elvira" Alter="34" />
            . . .
        </dbo:Personen>
    </Application.Resources>
</Application>
```

15.2 Datenkontext und Bindung

15.2.1 Die Eigenschaft DataContext

Der **FrameworkElement**-Eigenschaft **DataContext** kann ein Quellobjekt für Datenbindungszwecke zugewiesen werden. Im **ListBox**-Steuerelement und den beiden **TextBox**-Steuerelementen der folgenden Anwendung



sollen Daten aus der Ressource `Gruppe` (ein Objekt vom Typ **List**`<Person>` alias `Personen`) angezeigt bzw. verändert werden. Statt jedem einzelnen Steuerelement dieses Quellobjekt bekannt zu geben, sollte es der **DataContext**–Eigenschaft des umgebenden Layoutcontainers zugewiesen werden. Im Beispiel stammt das Quellobjekt aus einer Ressource, und es kommt die Attributsyntax für Markuperweiterungen (vgl. Abschnitt 9.4.2.7) zum Einsatz:

```
<Window x:Class="DatenbindungObjekt.MainWindow" .... >
   <Grid DataContext="{StaticResource Gruppe}">
      . . .
   </Grid>
   . . .
</Window>
```

15.2.2 Die Klasse Binding

Eine zentrale Rolle bei der Verknüpfung von Quelle und Ziel spielt ein Objekt der Klasse **Binding** mit dem folgenden Stammbaum:

System.Object
 System.Windows.Markup.MarkupExtension
  System.Windows.Data.BindingBase
   System.Windows.Data.Binding

Sie besitzt u.a. die folgenden Eigenschaften:

- **Path**
  Im relativ einfachen Fall eines CLR-Objekts als Quelle, auf den wir uns meist beschränken, erhält die **Path**-Eigenschaft den Namen der zu verbindenden Eigenschaft des Quellobjekts, z.B.:
  ```
  Path="Alter"
  ```
- **ElementName**
  Dieser Eigenschaft wird der Name eines Steuerelements zugewiesen, das als Quellobjekt dient, z.B.:
  ```
  ElementName="textBox2"
  ```
- **Source**
  Über diese Eigenschaft kann ein Objekt als Bindungsquelle festgelegt werden.
- **Mode**
  Über diese Eigenschaft wird der Bindungsmodus festgelegt, z.B.:
  ```
  Mode="OneWay"
  ```
- **Converter**
  Über diese Eigenschaft lässt sich ein Objekt vereinbaren, das die Werte zwischen Quelle und Ziel übersetzen soll.

Um eine Datenbindung einzurichten wird dem Ziel ein **Binding**-Objekt mit impliziter oder expliziter Quellangabe zugewiesen, was meist im XAML-Code geschieht.

Im **Binding**-Objekt zur **Text**-Eigenschaft im **TextBox**-Element für die Anzeige und Änderung des Namens wird per **Path**-Attribut die zu verbindende Quelleigenschaft genannt:

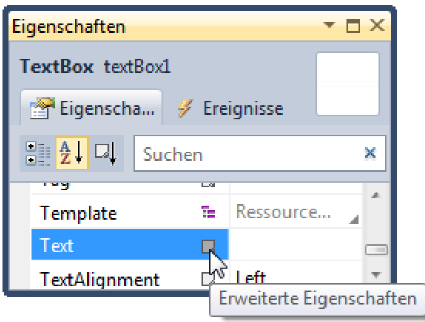
```
<TextBox Text="{Binding Path=Name}" . . . >
```

Für den Rest sorgt die WPF-Datenbindungstechnik:

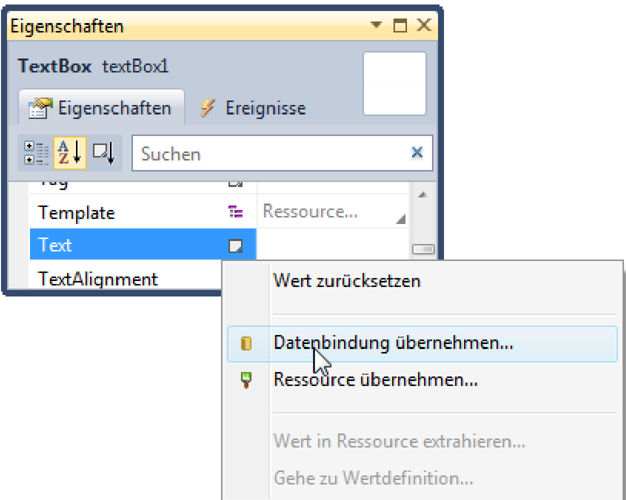
- Es wird im Elementbaum nach oben ein Quellobjekt gesucht, das der **DataContext**-Eigenschaft eines Steuerelements zugewiesen worden ist.
- Ist das Quellobjekt eine Kollektion, wird deren aktuelles Element ermittelt (im Beispiel: die aktuelle Person).

15.2.3 Unterstützung durch die Entwicklungsumgebung

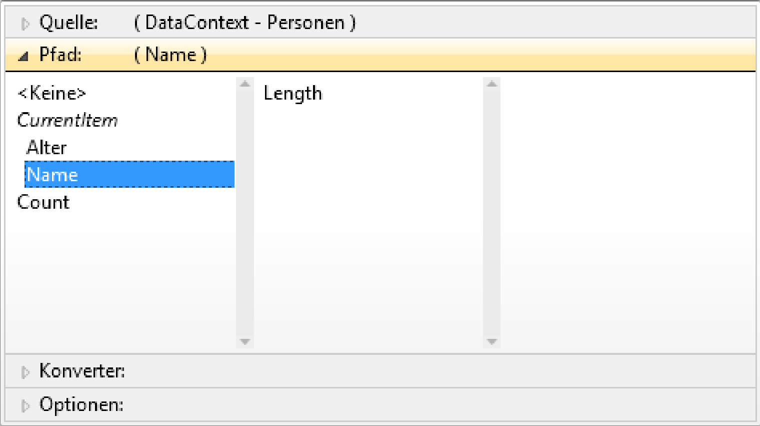
Aufgrund der Datenkontext-Definition kann unsere Entwicklungsumgebung das Einrichten einer Datenbindung unterstützen. Klicken Sie bei markiertem **TextBox**-Element im Eigenschaftsfenster neben der **Text**-Eigenschaft auf das Symbol mit dem Tool-Tip **Erweiterte Eigenschaften**:



Es erscheint ein Menü, aus dem das Item **Datenbindung übernehmen** zu wählen ist:



Wenn Sie im nächsten Fenster



noch einen Doppelklick auf die zu verbindende Eigenschaft `Name` im aktuellen Element (*CurrentItem*) der Quell-Kollektion aus der Klasse `Personen` setzen, erscheint im XAML-Code zum **TextBox**-Element ein **Text**-Attribut mit **Binding**-Objekt:

```
Text="{Binding Path=Name}"
```

15.2.4 Binding-Objekte ohne Pfadangabe

Im **ListBox**-Element erhält die Eigenschaft **ItemsSource** ein unspezifiziertes **Binding**-Objekt, das insbesondere keine **Path**-Angabe besitzt:

```
<ListBox ItemsSource="{Binding}" Name="lbPersonen" IsSynchronizedWithCurrentItem="True"
         Margin="12,12,0,12" HorizontalAlignment="Left" Width="172">
  . . .
</ListBox>
```

Damit ist implizit das gesamte, im Elementbaum über das **DataContext**-Attribut zur **Grid**-Layout-container passend deponierte Kollektionsobjekt aus der Ressource `Gruppe` zugewiesen (vgl. Abschnitt 15.2.1):

```
<Grid DataContext="{StaticResource Gruppe}">
    . . .
</Grid>
```

Über den Wert **true** zur Eigenschaft **IsSynchronizedWithCurrentItem** wird dafür gesorgt, dass die aktuelle **ListBox**-Markierung zum aktuellen Listenelement wird. Weil die beiden **TextBox**-Steuerelemente Eigenschaften des aktuellen Listenelements anzeigen, sind alle Steuerelemente synchronisiert.

Um die noch unbefriedigende Anzeige im **ListBox**-Steuerelement kümmern wir uns im nächsten Abschnitt.

15.3 Formatierung per DataTemplate-Objekt

Bislang zeigt das **ListBox**-Objekt zu jedem Item lediglich den Datentyp an (offenbar die Produktion der Methode **ToString**(), welche die Klasse `Person` 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:

```
<ListBox ItemsSource="{Binding}" Name="lbPersonen" IsSynchronizedWithCurrentItem="True"
         Margin="12,12,0,12" HorizontalAlignment="Left" Width="172">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Name="stackPanel1" Orientation="Horizontal">
                <TextBlock Text="{Binding Path=Name}"/>
                <TextBlock Text=" ("/>
                <TextBlock Text="{Binding Path=Alter}"/>
                <TextBlock Text=")"/>
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

Im XAML-Element **ListBox** wird die Eigenschaftselementsyntax (vgl. Abschnitt 9.4.2.3) verwendet, um den Wert der **ListBox**-Eigenschaft **ItemTemplate** zu deklarieren. 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 9.7.3) mit horizontaler Orientierung, um vier **TextBlock**-Objekte nebeneinander zu präsentieren. Zwei **TextBlock**-Objekte erhalten ihre Daten per **Binding**-Objekt mit **Path**-Angabe, z.B.:

```
<TextBlock Text="{Binding Path=Name}"/>
```

In welchem Objekt die Eigenschaft `Name` steckt, ermittelt die WPF-Datenbindungstechnik durch eine Suche im Elementbaum (siehe oben).

15.4 Bindungsmodus

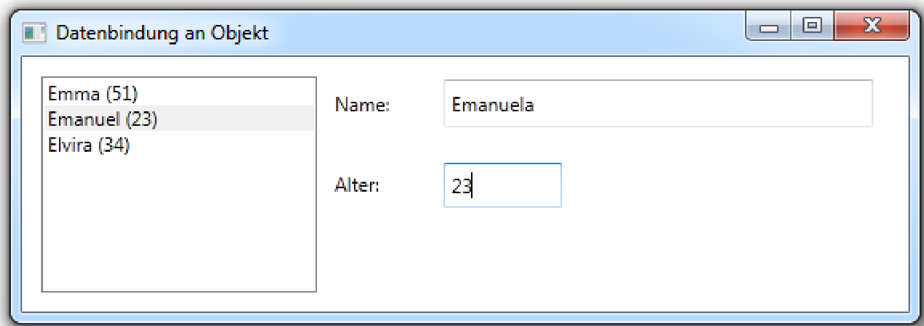
Über die **Mode**-Eigenschaft eines **Binding**-Objekts lässt sich die Arbeitsweise der Datenbindung einstellen:

- **TwoWay**
  Abgleich in beide Richtungen
- **OneWay**
  Datenfluss nur von der Quelle zum Ziel
- **OneTime**
  Datenfluss nur von der Quelle zum Ziel, pro Start des Programms nur einmal
- **OneWayToSource**
  Datenfluss nur vom Ziel zur Quelle

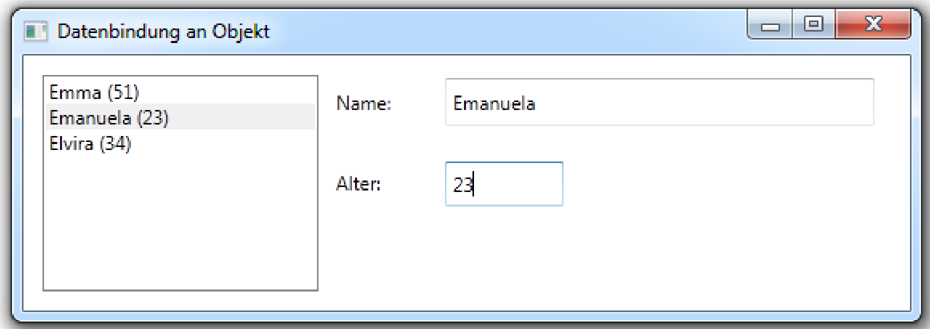
Erhält im Beispiel das **Binding**-Objekt zur Namens - **TextBox** den Bindungsmodus **OneWay**,

```
<Binding Path="Name" Mode="OneWay"/>
```

fließen Textänderungen *nicht* zurück zur Quelle:



Bei einer Zweiwegbindung, die beim **TextBox**-Steuerelement voreingestellt ist, wird der geänderte Name nach dem Fokuswechsel zum Quellobjekt übertragen und anschließend vom **ListBox**-Objekt angezeigt:

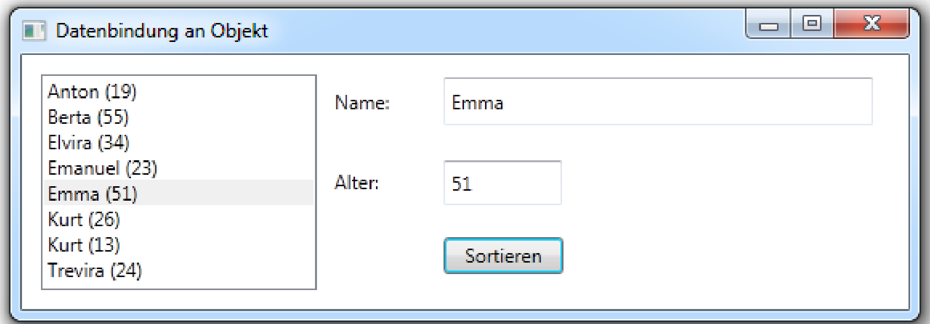


15.5 View-Objekt zu einer Kollektion

Zwischen einem Kollektionsquellobjekt und dem Ziel sitzt ein **View**-Objekt, das die Schnittstelle **System.ComponentModel.ICollectionView** erfüllt und einige Transformationen beherrscht (z.B. Sortieren oder Filtern der Listenelemente). Es übernimmt die Daten der Quelle, transformiert sie und liefert sie beim Datenbindungsziel ab. Bei unserem Kollektionsobjekt aus der generischen Klasse **List**`<Person>` ist ein Objekt der Klasse **System.Windows.Data.ListCollectionView** im Spiel.

15.5.1 Sortieren

Das **View**-Objekt soll in der **Click**-Behandlungsmethode zu einem neu ins Programm aufgenommenen **Button**-Steuerelement



dazu verwendet werden, die Liste der Personen zu sortieren. Hier ist die Code-Behind – Datei zum Hauptfenster zu sehen:

```
using System;
 . . .
using System.ComponentModel;

namespace DatenbindungObjekt {

 public partial class MainWindow : Window {

  Personen personen;
  ICollectionView view;
```

```
    public MainWindow() {
     InitializeComponent();
     personen = (Personen)this.FindResource("Gruppe");
     view = CollectionViewSource.GetDefaultView(personen);
    }

    private void btnSort_Click(object sender, RoutedEventArgs e) {
     if (view.SortDescriptions.Count == 0) {
      view.SortDescriptions.Add(new SortDescription("Name", ListSortDirection.Ascending));
      view.SortDescriptions.Add(
         new SortDescription("Alter", ListSortDirection.Descending));
     } else
      view.SortDescriptions.Clear();
    }
   }
  }
```

Eine Referenz zum Kollektionsobjekt mit der Personenliste erhalten wir über die **Framework-Element**-Methode **FindResource**(). Sie liefert eine Rückgabe vom Typ **Object**, so dass eine explizite Typumwandlung nötig wird. Über die statische **CollectionViewSource**-Methode **GetDefault-View**() wird eine Referenz zum **View**-Objekt beschafft, das aktuell zwischen dem Kollektionsobjekt und Datenbindungszielen vermittelt.

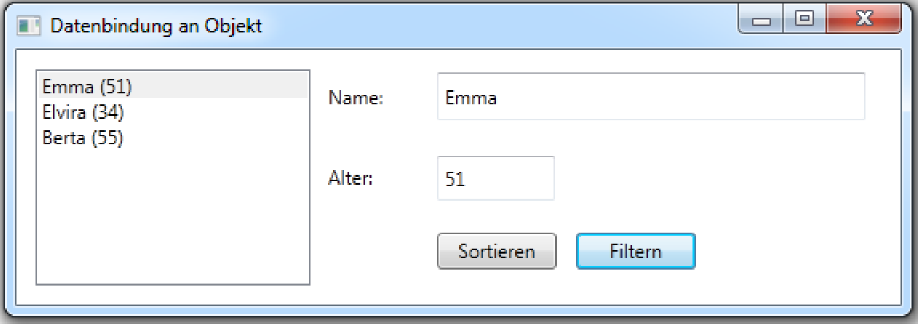
Das **View**-Objekt enthält eine Liste von **SortDescription**-Objekten, welche die nacheinander anzuwendenden Sortierkriterien enthalten. Im Beispiel wird eine leere Liste mit Sortierkriterien so erweitert, dass die Elemente ...

- zunächst aufsteigend nach dem Namen
- und bei namensgleichen Elementen absteigend nach dem Alter sortiert werden.

Sind bereits Sortierkriterien vorhanden, werden diese gelöscht.

15.5.2 Filtern

In der **Click**-Behandlungsmethode zu einem neu ins Programm aufgenommenen **Button**-Objekt



soll das **View**-Objekt zur Kollektion dazu verwendet werden, die Liste auf Personen über 30 zu beschränken:

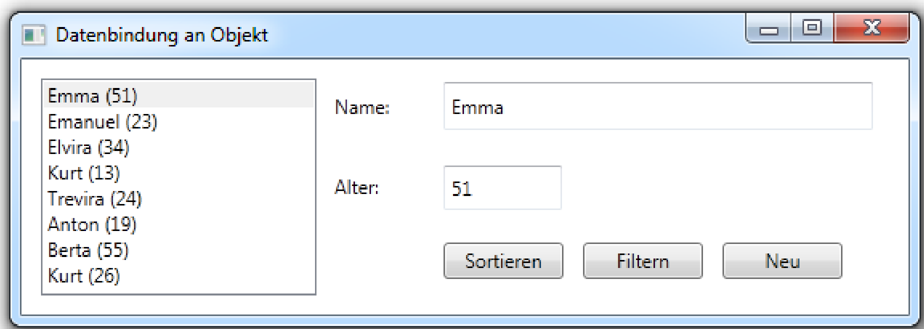
```
private void btnFilter_Click(object sender, RoutedEventArgs e) {
   if( view.Filter == null )
      view.Filter = delegate(object item) {
           return ((Person)item).Alter > 30;
        };
   else
      view.Filter = null;
}
```

Dazu wird der **Filter**-Eigenschaft des **View**-Objekts ein Delegatenobjekt zugewiesen, das auf eine anonyme Methode (siehe Abschnitt 9.3.1.4) mit einem Parameter vom Typ **Object** und einem

Rückgabewert vom Typ **bool** zeigt. Diese Methode wird für jedes Listenelement aufgerufen und entscheidet darüber, ob ein Element im Datenbindungsziel erscheint (Rückgabe **true**) oder nicht (Rückgabe **false**).

15.6 Listenelemente ergänzen oder entfernen

Nach unseren positiven Erfahrungen mit der WPF-Datenbindungstechnik ist es zu erwarten, dass die Veränderung einer Kollektion sich in einem angebundenen **ListBox**-Steuerelement sofort niederschlägt. Um diese optimistische Erwartung zu prüfen, erweitern wir unser Beispielprogramm um ein **Button**-Objekt,



dessen **Click**-Behandlungsmethode eine neue Person in das Objekt der Klasse **List**<`Person`> aufnimmt:

```
private void btnNeu_Click(object sender, RoutedEventArgs e) {
   personen.Add(new Person{Name= "Ortlieb", Alter = 22});
}
```

Bei einem Test stellen wir jedoch enttäuscht im **ListBox**-Steuerelement *keine* Reaktion auf die Listenerweiterung fest.

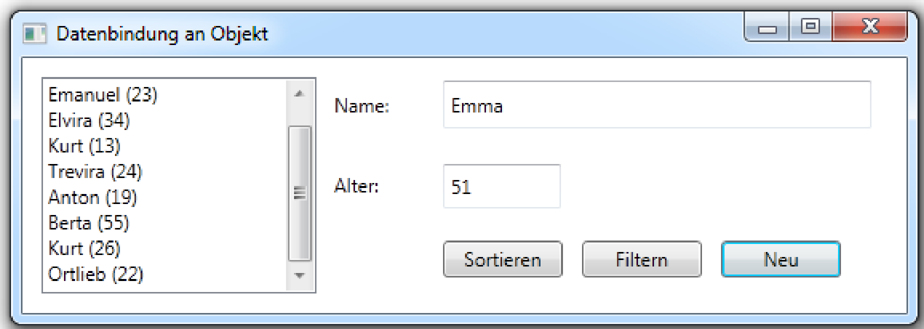
Was fehlt, ist die Kompetenz der Klasse **List**<`Person`>, eine Veränderung der Kollektion zu melden. Für die gewünschte Funktionalität muss eine Kollektionsklasse das Interface **INotifyCollectionChanged** im Namensraum **System.Collections.Specialized** erfüllen. Statt in einer **List<>**–Ableitung die erforderlichen Methoden zu implementieren, verwenden wir (wie übrigens auch schon in Abschnitt 9.6.3) die entsprechend ausgestattete FCL-Klasse **ObservableCollection<>** im Namensraum **System.Collections.ObjectModel**. Wir ersetzen also die Klassendefinition

```
class Personen : List<Person> { }
```

durch die Alternative:

```
class Personen : ObservableCollection<Person> { }
```

Nun erfährt das **ListBox**-Steuerelement von einer Listenerweiterung:

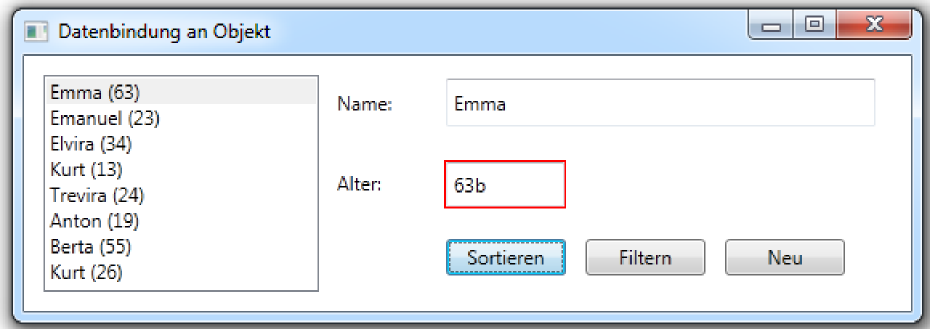


15.7 Validierung von Eingabedaten

Bei einer bidirektionalen Datenbindung besteht potentiell die Aufgabe, die vom Datenbindungsziel (Steuerelement) zur Quelle fließenden Daten zu validieren, damit keine ungültigen Objekte oder Missverständnisse beim Benutzer entstehen. Das Verhindern ungültiger Objekte obliegt den Autoren der Klassen im Quellbereich. Bei unserer Klasse `Person` ist dafür gesorgt, dass Altersangaben auf das Intervall [0, 130] beschränkt bleiben:

```
public int Alter {
   get { return alter; }
   set {
      if (alter == value)
         return;
      if (0 <= value  &&  value <= 130) {
         alter = value;
         ValueChanged("Alter");
      }
   }
}
```

Tritt bei der WPF-internen Verarbeitung ein Ausnahmefehler aus (z.B. wegen einer nicht numerisch interpretierbaren Zeichenfolge), erhält der Benutzer durch die rote Umrahmung des betroffenen Steuerelements einen dezenten Hinweis auf das Problem, z.B.:



Demgegenüber bleibt der Versuch, eine zu kleine oder zu große Zahl einzutragen, folgenlos (keine Wertänderung und auch kein Hinweis). Damit auch bei einer irregulären Zahl der warnende rote Rahmen erscheint, müssen wir ein zur Überprüfung geeignetes Objekt erstellen und in die Liste der benutzerdefinierten Validierungsregeln zum **Binding**-Objekt aufnehmen. Wir definieren eine von **ValidationRule** abgeleitete Klasse namens `AgeRangeRule` und überschreiben die Methode **Validate**():

```
using System;
using System.Windows.Controls;

namespace DatenbindungObjekt {
   public class AgeRangeRule : ValidationRule {
      public override ValidationResult Validate(object value,
            System.Globalization.CultureInfo cultureInfo) {
         int number;
         if (!int.TryParse((String)value, out number)) {
            return new ValidationResult(false,"Keine Zahl!");
         }
         if (number < 0 || number > 130)
            return new ValidationResult(false, "Zahl < 0 oder > 130");
         return ValidationResult.ValidResult;
      }
   }
}
```

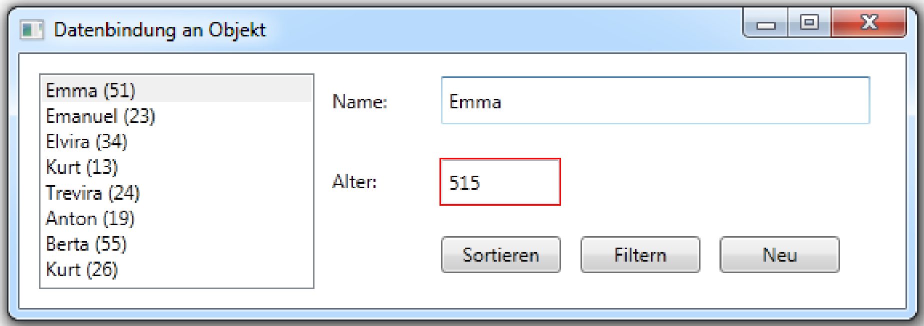
Das **Binding**-Objekt zum betroffenen **TextBox**-Steuerelement erhält als zusätzliche Validierungsregel ein Objekt der Klasse `AgeRangeRule`:

```
<TextBox.Text>
    <Binding Path="Alter">
        <Binding.ValidationRules>
            <dbo:AgeRangeRule />
        </Binding.ValidationRules>
    </Binding>
</TextBox.Text>
```

Damit das Namensraumpräfix verstanden wird, benötigt die XAML-Datei zur Hauptfensterklasse eine entsprechende Namensraumdefinition:

```
xmlns:dbo="clr-namespace:DatenbindungObjekt"
```

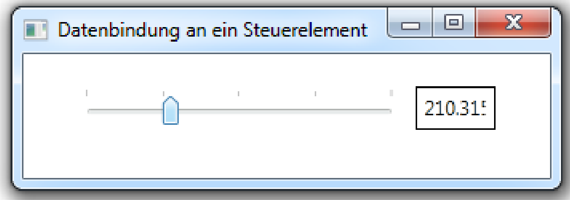
Nun reagiert das Programm auch bei einer gescheiterten Quellaktualisierung wegen Bereichsüberschreitung mit einem optischen Hinweis:



Bei Sells & Griffiths (2007, S. 189ff) finden sich weitere Empfehlungen zur benutzerfreundlichen Reaktion auf ungültige Daten, z.B. durch Ausgabe einer informativen Meldungen unter Verwendung der im **ValidationResult**-Objekt (siehe oben) enthaltenen Fehlerbeschreibung, die im Beispielprogramm ungenutzt bleibt.

15.8 Datenbindung zwischen zwei Steuerelementen

In den bisherigen Beispielen wurden gewöhnliche CLR-Objekte als Datenquellen verwendet; im folgenden Programm spielt ein Steuerelement diese Rolle. Wir verwenden erstmals ein Schieberegler-Steuerelement aus der Klasse **Slider**, und ergänzen ein **Label**-Objekt, das für die präzise numerische Anzeige der Reglerstellung sorgt:



Dazu wird der **Label**-Eigenschaft **Content** im XAML-Code zum Hauptfenster ein **Binding**-Objekt zugewiesen, das die Datenquelle und den Namen der zu verbindenden Eigenschaft kennt:

```
<Window x:Class="DatenbindungSteuerelement.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Datenbindung an ein Steuerelement" Height="115" Width="345">
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
       <Slider Height="23" Margin="10,0,10,10" Name="slider1" Width="200"
               Minimum="100" Maximum="500" TickPlacement="TopLeft" TickFrequency="100" />
       <Label Content="{Binding ElementName=slider1, Path=Value}"
              Width="50" Margin="0, 0, 10, 10" BorderThickness="1" BorderBrush="Black"
              VerticalAlignment="Center" HorizontalContentAlignment="Center"  />
    </StackPanel>
</Window>
```

Beim **Slider**-Objekt werden die folgenden speziellen Eigenschaften mit Werten versorgt:

- **Minimum**, **Maximum**
  Die Eigenschaften **Minimum** und **Maximum** legen den Regelbereich fest.
- **TickPlacement**
  Über die Eigenschaft **TickPlacement** wählt man die Position der Skala.
- **TickFrequency**
  Die Eigenschaft **TickFrequency** regelt den Abstand zwischen zwei Markierungen auf der Skala.

Bei der Anzeige im Label stört eine übergroße und abgeschnittene Anzeige von Dezimalstellen. Um für eine formatierte Ausgabe mit genau zwei Dezimalstellen zu sorgen, verwenden wir die **Binding**-Eigenschaft **StringFormat**:

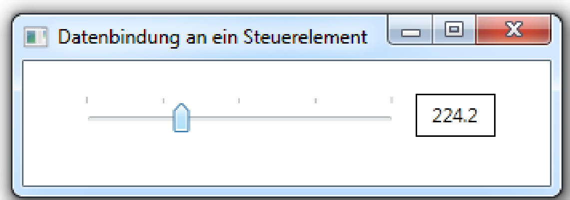
```
{Binding ElementName=slider1, Path=Value, StringFormat={}{0:F1}}">
```

Wie schon aus Abschnitt 3.2.2 bekannt, ist die Formatierungszeichenfolge durch geschweifte Klammern zu begrenzen. Im Kontext mit der XAML-Attributsyntax für Markuperweiterungen (siehe Abschnitt 9.4.2.7) muss durch ein vorangestelltes leeres Klammernpaar {} die besondere Bedeutung der geschweiften Klammern angekündigt werden.

Die **StringFormat**-Eigenschaft ist allerdings nur bei reinen Textausgaben wirksam, während wir bisher der **Label**-Eigenschaft **Content** das **Binding**-Objekt zugewiesen haben. Wir ersetzen daher das **Content**-Attribut im **Label**-Element durch ein Inhaltselement vom Typ **TextBlock** und weisen der **Text**-Eigenschaft dieses Elementes das **Binding**-Objekt zu:

```
<Label Width="50" Margin="0, 0, 10, 10" BorderThickness="1" BorderBrush="Black"
       VerticalAlignment="Center" HorizontalContentAlignment="Center" >
    <TextBlock Text="{Binding ElementName=slider1, Path=Value,
                StringFormat={}{0:F1}}">
    </TextBlock>
</Label>
```

Am Ergebnis

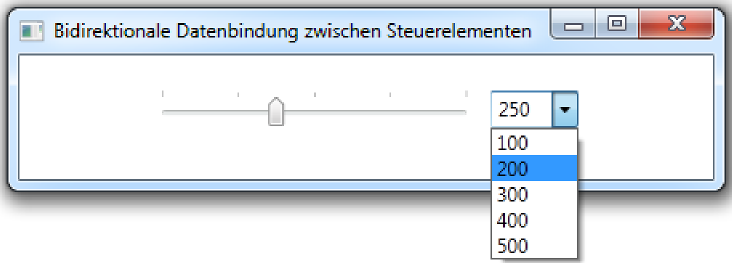


ist nur noch ein Punkt zu bemängeln: WPF verwendet leider das falsche Dezimaltrennzeichen. Offenbar wird zur Formatierung generell die Ländereinstellung **en-US** an Stelle der Voreinstellung des lokalen Betriebssystems verwendet. Man kann ärgerlichen Fehler in .NET 4.0 durch den folgenden Methodenaufruf in einer Behandlungsmethode für das **Application**-Ereignis **Startup** beheben:[1]

```
private void Application_Startup(object sender, StartupEventArgs e) {
 FrameworkElement.LanguageProperty.OverrideMetadata(
    typeof(FrameworkElement),
    new FrameworkPropertyMetadata(
     System.Windows.Markup.XmlLanguage.GetLanguage(
       System.Globalization.CultureInfo.CurrentCulture.IetfLanguageTag)));
}
```

[1] Der Tipp stammt von der Webseite:
http://stackoverflow.com/questions/520115/stringformat-localization-problem/520334#520334

Mit einer **ComboBox** an Stelle eines Labels lässt sich die bidirektionale Kopplung von zwei Steuerelementen demonstrieren. Der vom Benutzer gewählte oder eingetragene **ComboBox**-Wert wird sofort zum **Slider**-Objekt übertragen:



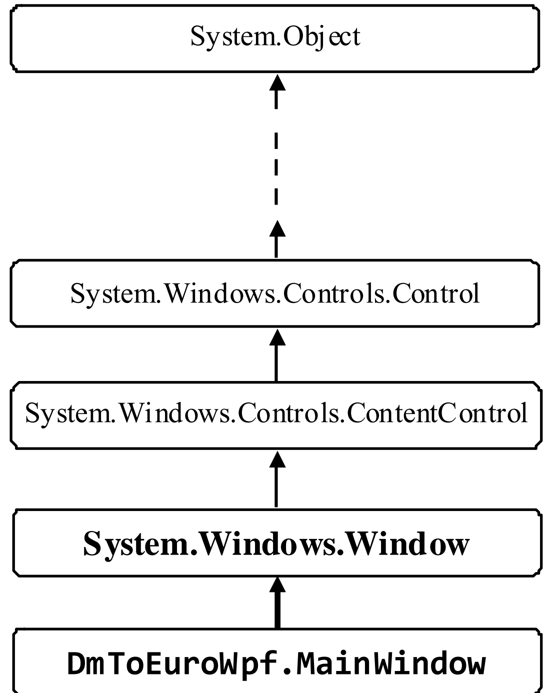
Den XAML-Code zu diesem Beispiel sollen Sie als Übungsaufgabe erstellen.

15.9 Übungsaufgaben zu Kapitel 15

1) Erstellen Sie den XAML-Code zum zweiten Beispiel in Abschnitt 15.8 (Kopplung von **Slider** und **ComboBox**).

16 Fenster und Dialoge

Wir haben bei allen WPF-Anwendungen die von **System.Windows.Controls.ContentControl** abstammende Klasse **System.Windows.Window** als Basisklasse für unsere Hauptfensterklasse verwendet, z.B. schon beim DM–Euro – Konverter in Abschnitt 4.1.1.2:



Allerdings taugt die Klasse **Window** nicht nur für Top-Level – Fenster, sondern auch für Dialogfenster, die bei einem meist zeitlich befristeten Auftritt für spezielle Aufgaben verwendet werden (z.B. Drucken, Anzeige und Änderung von Einstellungen).

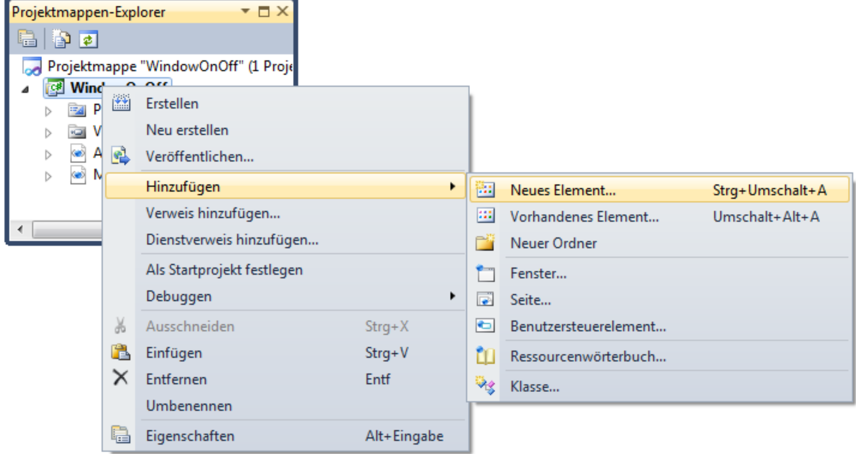
Im Vergleich zu ihrer Basisklasse **ContentControl** leistet die Klasse **Window** wesentliche Beiträge bei der Randgestaltung, z.B. bei der Titelzeile und den dortigen Bedienelementen (Icon mit Systemmenü, Titel, Schaltflächen zum Minimieren, Maximieren, Schließen). Mit den zuständigen Eigenschaften **Icon**, **Title** und **WindowStyle** haben wir uns schon beschäftigt (siehe Abschnitte 9.2.2 und 9.5.10).

16.1 Fenster erstellen und anzeigen

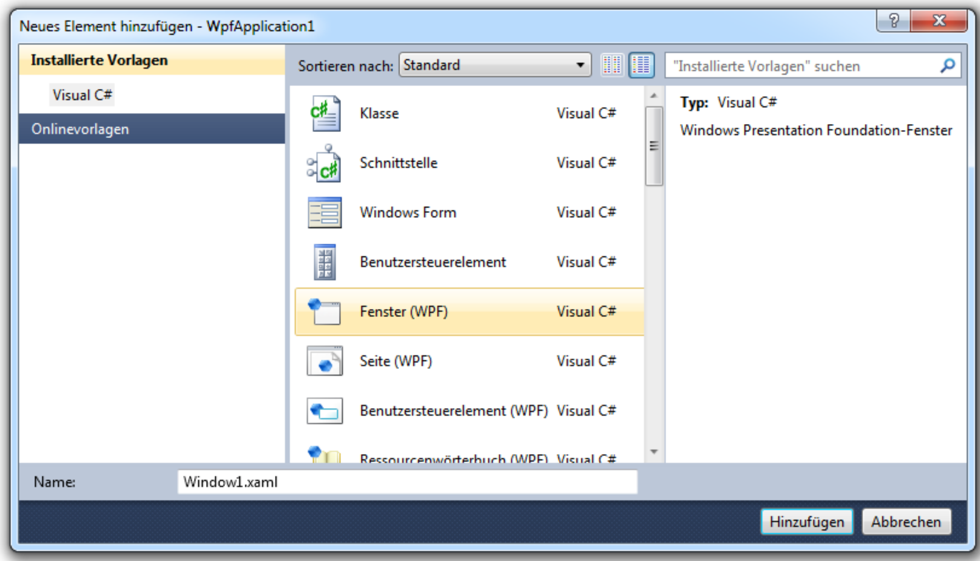
Wenn Sie mit dem Visual Studio eine neue WPF-Anwendung erstellen, wird beim Hauptfenster für das Definieren der Klasse sowie für das Instantiieren, Initialisieren und Anzeigen eines Fensterobjekts automatisch gesorgt. Wir haben in Kapitel 9 einige neugierige Blicke hinter die WPF-Kulissen gewagt.

16.1.1 Unterstützung durch das Visual Studio

Um ein zusätzliches Fenster (Klasse und Objekt) mit Hilfe der Entwicklungsumgebung zu erstellen, öffnet man im **Projektmappen-Explorer** per Maus-Rechtsklick das Kontextmenü zum *Projekt* (nicht zur *Projektmappe*), und fügt ein **neues Element** hinzu:



Entscheiden Sie sich für ein **Fenster (WPF)**:



Nach dem **Hinzufügen** erscheint in der Editorzone zum neuen Fenster der XAML-Code und die Fenstervorschau.

Das Hauptfenster wird beim Start der Anwendung automatisch angezeigt, weil aus seiner XAML-Datei ein **Uri**-Objekt entsteht, das der **StartupUri**-Eigenschaft des Anwendungsobjekts als Wert zugewiesen wird:

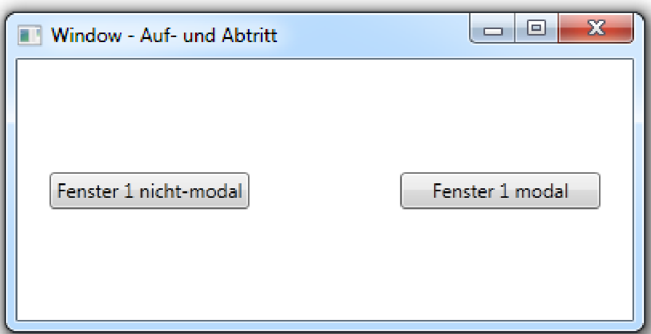
```
public partial class App : System.Windows.Application {
    [System.Diagnostics.DebuggerNonUserCodeAttribute()]
    public void InitializeComponent() {

        #line 4 "..\..\..\App.xaml"
        this.StartupUri = new System.Uri("MainWindow.xaml", System.UriKind.Relative);

        #line default
        #line hidden
    }

    [System.STAThreadAttribute()]
    [System.Diagnostics.DebuggerNonUserCodeAttribute()]
    public static void Main() {
        WindowOnOff.App app = new WindowOnOff.App();
        app.InitializeComponent();
        app.Run();
    }
}
```

16.1.2 Modaler und nichtmodaler Auftritt

Für den Auftritt weitere **Window**-Objekte sorgt der Programmierer und hat dabei zwei Einsatzarten zur Verfügung:

- **Modales Fenster**
  Die meisten Dialogfenster sind vom *modalen* Typ, so dass die restlichen Fenster der Anwendung bis zum Schließen des Dialogs keine Benutzereingaben entgegen nehmen. Einem modalen Dialog ist also die ungeteilte Aufmerksamkeit der Benutzer gewiss.
- **Nicht-modales Fenster**
  Viele Dialogfenster sollen (oder müssen) längere Zeit oder gar ständig offen bleiben und dürfen dabei die Funktionalität des Hauptfensters nicht behindern. Häufig anzutreffende Beispiele sind die Suchdialoge von Editoren.

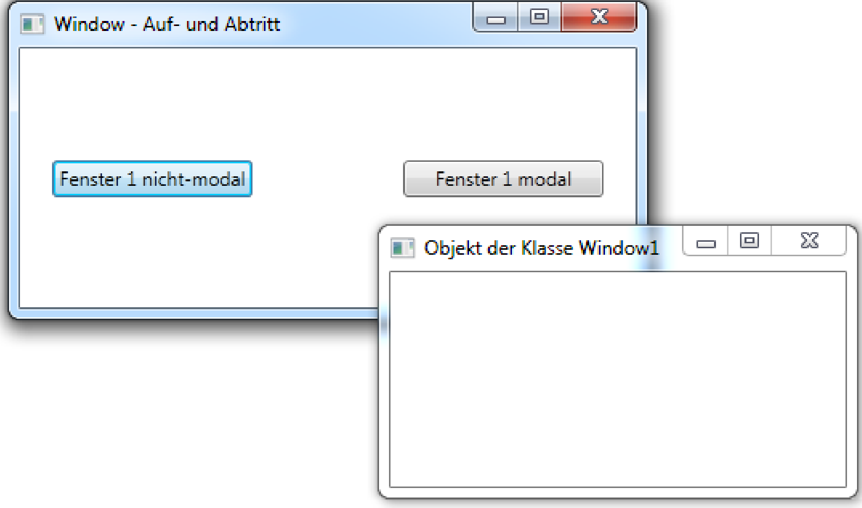
Das folgende Beispielprogramm bietet im Hauptfenster zwei Schalter, die den modalen oder nicht-modalen Auftritt eines sekundären Fensters ermöglichen:



In der **Click**-Behandlungsmethode zum linken Schalter wird ein Fenster der Klasse `Window1` erzeugt und mit der **Show**()-Methode zum nicht-modalen (engl.: *modeless*) Auftritt aufgefordert:

```
private void btnWindowNon_Click(object sender, RoutedEventArgs e) {
   Window1 window = new Window1();
   window.Owner = this;
   window.Show();
}
```

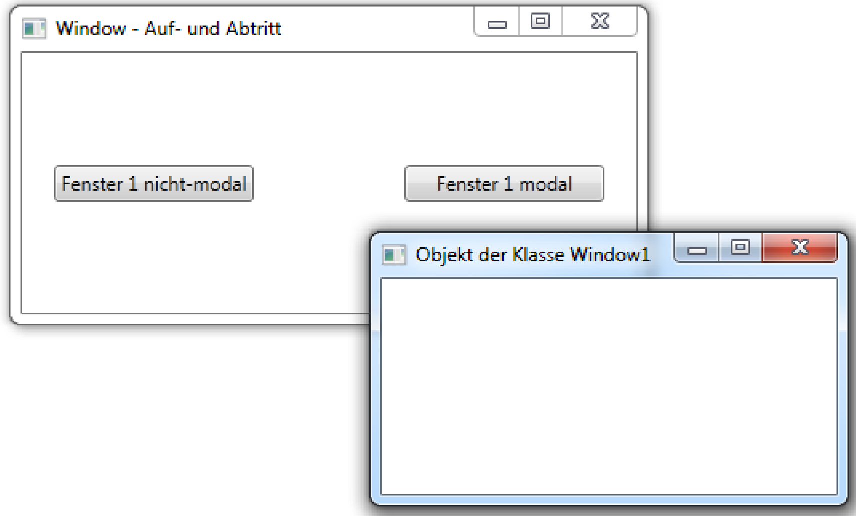
Der **Show**()-Methodenaufruf endet sofort, und das sekundäre Fenster behindert nicht die Verwendbarkeit des Hauptfensters, mal abgesehen von seiner Tendenz, sich in den Vordergrund zu drängen:



In der **Click**-Behandlungsmethode zum *rechten* Schalter im Hauptfenster des Beispielprogramms wird ein Fenster der Klasse `Window1` erzeugt und mit der **ShowDialog**()-Methode zum modalen Auftritt aufgefordert:

```
private void btnWindowMod_Click(object sender, RoutedEventArgs e) {
   Window1 window = new Window1();
   window.ShowDialog();
}
```

Der **ShowDialog**()-Aufruf kehrt erst dann zurück, wenn das sekundäre Fenster geschlossen wird, und so lange bleibt das Hauptfenster unerreichbar:



16.1.3 Besitzverhältnisse

Die Benutzung einer Mehrfensteranwendung wird erleichtert, wenn untergeordnete Fenster über ihre Eigenschaft **Owner** ein übergeordnetes Fenster als Inhaber erhalten, denn:

- Wird das besitzende Fenster minimiert oder reaktiviert, folgen ihm alle in seinem Besitz befindlichen Fenster automatisch. In der Taskleiste besitzt ein abhängiges Fenster keinen
- Wird ein Fenster geschlossen, erleiden alle in seinem Besitz befindlichen Fenster dasselbe Schicksal.
- Ein Kindfenster liegt in der Z-Anordnung per Voreinstellung über dem Besitzer.

Um den Besitzer zu ernennen, haben wir in der **Click**-Behandlungsmethode zum linken Schalter nach dem Erzeugen des Fensterobjekts seine **Owner**-Eigenschaft auf das handelnde Hauptfenster gesetzt:

```
window.Owner = this;
```

16.1.4 Fenstergröße

Die Größe eines **Window**-Objekts hat viele Determinanten:

- Über die **Window**-Eigenschaft **SizeToContent** wird entschieden, ob die enthaltenen Komponenten über die Größe (mit-)bestimmen sollen. Mögliche Werte (vom Enumerationstyp **System.Windows.SizeToContent**):
    - **Manual**
      Bei diesem voreingestellten **SizeToContent**-Wert findet keine Orientierung am Inhalt statt.

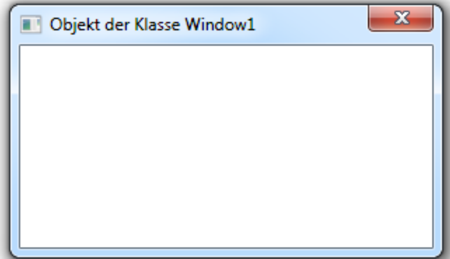
- **Width**, **Height**, **WidthAndHeight**.
  Breite, Höhe oder beide Ausdehnungen orientieren sich am Inhalt des Fensters, wobei allerdings die **Window**-Eigenschaften **MinWidth**, **MaxWidth**, **MinHeight** und **MaxHeight** (siehe unten) respektiert werden.

- Die **Window**-Eigenschaften **Width** und **Height**
  Diese Eigenschaften legen die Ausdehnung eines Fensters (in geräteunabhängigen Pixeln von 1/96 Zoll) fest, wenn keine Orientierung am Inhalt stattfindet und keine Begrenzungen überschritten werden.
- Die **Window**-Eigenschaften **MinWidth**, **MaxWidth**, **MinHeight** und **MaxHeight** begrenzen die Ausdehnung effektiv.

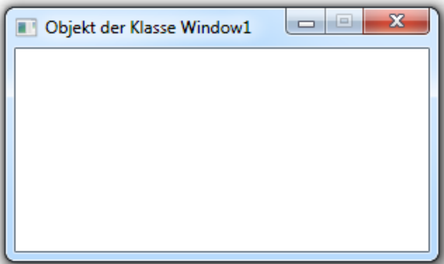
Über die wie auch immer zustande gekommene aktuelle Ausdehnung eines Fensters informieren die beiden schreibgeschützten Eigenschaften **ActualWidth** und **ActualHeight**.

Ob der Benutzer die Fenstergröße ändern kann, regelt die **Window**-Eigenschaft **ResizeMode** mit den folgenden Werten

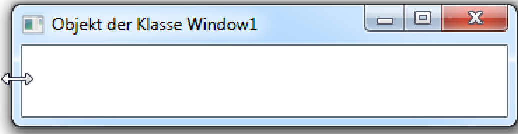
- **NoResize**
  Der Benutzer kann die Fenstergröße nicht ändern. In der Titelzeile sind keine Schaltflächen zum Minimieren oder Maximieren vorhanden:



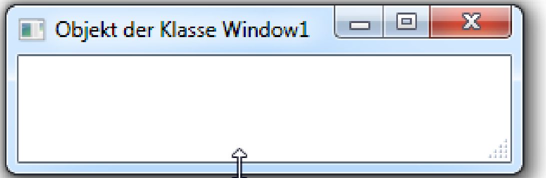
- **CanMinimize**
  Der Benutzer hat keinen Einfluss auf die Fenstergröße, kann das Fenster aber in die Taskleiste dirigieren. Der Schalter zum Maximieren ist sichtbar, aber passiv:

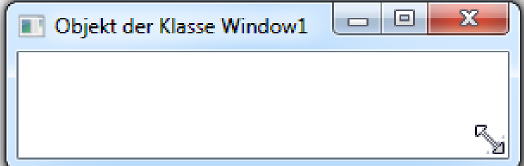


- **CanResize**
  Beim diesem voreingestellten **ResizeMode** hat der Benutzer die volle Kontrolle über die Fenstergröße und kann an jedem Fensterrand zupfen:



- **CanResizeWithGrip**
  Beim diesem **ResizeMode** sind alle Bedienmöglichkeiten der Voreinstellung **CanResize** vorhanden. Zusätzlich bietet die rechte untere Ecke eine erweiterte Griffzone:





16.1.5 Zustand eines Fensters

Über die **Window**-Eigenschaft **WindowState** mit Werten vom Typ der gleichnamigen Enumeration aus dem Namensraum **System.Windows** lassen sich folgende Zustände eines Fensters fest- oder einstellen:

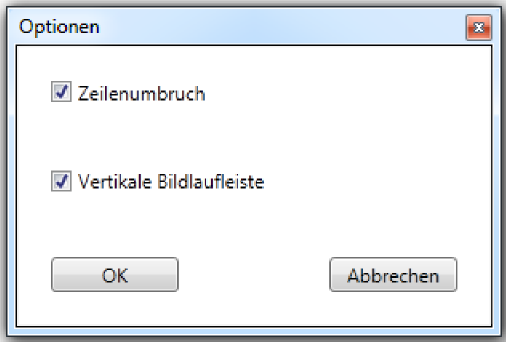
- **Normal**
- **Minimized**
  Das Fenster kann über die Taskleiste reaktiviert werden.
- **Maximized**
  Das Fenster belegt die gesamte Bildschirmfläche (mit Ausnahme der Taskleiste).

Über die **Window**-Eigenschaft **RestoreBounds** vom Typ **System.Windows.Rect** erfährt man Position und Größe eines Fensters vor dem Minimieren oder Maximieren.

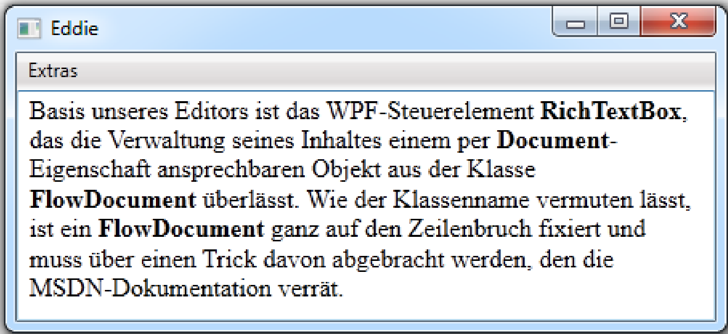
16.2 Modale Dialogfenster

16.2.1 Definition einer Klasse für ein modales Dialogfenster

In der Regel definiert man zu einem Dialogfenster eine eigene Klasse, welche direkt oder indirekt von **Form** abstammt. Wir erstellen als Beispiel die Klasse `OptionsDialog`, die ein Optionen-Dialogfenster



für einen einfachen Editor realisiert:



Folgende Einstellungen des Editors können im Optionen-Dialog modifiziert werden:

- Zeilenumbruch
  Basis unseres Editors ist das WPF-Steuerelement **RichTextBox**, das die Verwaltung seines Inhaltes einem per **Document**-Eigenschaft ansprechbaren Objekt aus der Klasse **Flow-Document** überlässt. Wie der Klassenname vermuten lässt, ist ein **FlowDocument** ganz auf den Zeilenbruch fixiert und muss über einen Trick davon abgebracht werden, den die MSDN-Dokumentation verrät:[1]

  > Text always wraps in a **RichTextBox**. If you do not want text to wrap then set the **PageWidth** on the **FlowDocument** to be larger than the width of the **RichTextBox**. However, once the page width is reached the text still wraps.

  Bei einem **RichTextBox**-Steuerelement mit dem Namen `eddie` wird nach diesem Vorschlag folgendermaßen der Zeilenumbruch abgeschaltet:

  ```
  eddie.Document.PageWidth = 5000;
  ```

  Um den rechten Textrand ohne Verbreitern des Fensters erreichen zu können, sollte man einen horizontalen Rollbalken aktivieren:

  ```
  eddie.HorizontalScrollBarVisibility = ScrollBarVisibility.Visible;
  ```

- Vertikale Bildlaufleiste

Im XAML-Code zur Klasse `OptionsDialog` sorgen wir für ein Dialogbox-adäquates Erscheinungsbild und Verhalten:

```
<Window x:Class="Eddie.OptionsDialog"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="Optionen" Height="197" Width="300"
      WindowStyle="ToolWindow" ResizeMode="NoResize" WindowStartupLocation="CenterOwner">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="192*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <CheckBox Content="Zeilenumbruch" HorizontalAlignment="Left" Margin="20"
                  Name="chkWordWrap" VerticalAlignment="Top" />
        <CheckBox Content="Vertikale Bildlaufleiste" HorizontalAlignment="Left"
                  Margin="20" Name="chkVertSB" VerticalAlignment="Bottom" />
        <Button Name="btnOK" Content="OK" Width="80" Grid.Row="1" Margin="20"
                HorizontalAlignment="Left" IsDefault="True" />
        <Button Name="btnCancel" Content="Abbrechen" Width="80" Grid.Row="1" Margin="20"
                HorizontalAlignment="Right" IsCancel="True" />
    </Grid>
</Window>
```

Den angemessenen Rahmen erhalten `OptionsDialog`-Objekte über den Wert **ToolWindow** für die Eigenschaft **WindowStyle**. Mit dem Wert **NoResize** für die Eigenschaft **ResizeMode** verhindern wir eine Änderung der Fenstergröße durch den Benutzer.

Damit das Dialogfenster zentriert über dem Hauptfenster auftritt, setzen wir die Eigenschaft **WindowStartupLocation** auf den Wert **CenterOwner**. Außerdem muss das Dialogfenster durch einen passenden Wert seiner Eigenschaft **Owner** über den Besitzer informiert werden, z.B. unmittelbar nach dem Erstellen durch das handelnde Objekt:

```
OptionsDialog od = new OptionsDialog();
od.Owner = this;
```

Über die **Button**-Attribute **IsDefault** bzw. **IsCancel** sorgt man dafür dass per **Enter**- bzw. **Esc**-Taste das **Click**-Ereignis zum OK- bzw. Abbrechen-Schalter ausgelöst werden kann.

---

[1] Siehe: http://msdn.microsoft.com/de-de/library/system.windows.controls.richtextbox%28VS.85%29.aspx

Ansonsten bietet der XAML-Code zur Klasse `OptionsDialog` die mittlerweile vertrauten Steuer-element-Gestaltungsarbeiten.

16.2.2 Datenaustausch mit dem aufrufenden Fenster

Für den Datenaustausch mit einer nutzenden Klasse erhält unsere `OptionsDialog`–Klasse zu jeder Einstellmöglichkeit eine öffentliche Eigenschaft mit Lese- und Schreibzugriff, z.B.:

```
public bool WordWrap {
   get {return (bool) chkWordWrap.IsChecked;}
   set {chkWordWrap.IsChecked = value;}
}
```

Zu Beginn des nächsten Abschnitts ist zu sehen, wie diese Eigenschaften vor dem Öffnen des Dialogs initialisiert und nach Beenden des Dialogs ausgelesen werden.

Um den Optionsdialog im Hauptfenstermenü anzubieten, wird das **Extras**-Untermenü um ein Item erweitert:

```
<MenuItem Header="Extras">
    <MenuItem Header="Optionen..." Name="miOptions" Click="miOptions_Click" />
</MenuItem>
```

Mit drei Punkten am Ende einer Menüitem-Beschriftung kündigt man den Auftritt eines Dialogfensters an. Die zugehörige **Click**-Behandlungsmethode mit dem Aufruf des Optionsdialogs wird im nächsten Abschnitt vorgestellt.

16.2.3 Auftritt und Abgang

Ein modaler Dialog wird über die **Window**-Methode **ShowDialog()** angezeigt, z.B. in der **Click**-Behandlungsmethode des zuständigen Menüitems in unserem Editor:

```
private void miOptions_Click(object sender, RoutedEventArgs e) {
   OptionsDialog od = new OptionsDialog();
   od.Owner = this;
   od.WordWrap = wordWrap;
   od.VertSB = vertSB;
   if (od.ShowDialog() == true) {
      wordWrap = od.WordWrap;
      vertSB = od.VertSB;
   }
   if (wordWrap) {
      eddie.HorizontalScrollBarVisibility = ScrollBarVisibility.Disabled;
      eddie.Document.PageWidth = Double.NaN;
   } else {
      eddie.HorizontalScrollBarVisibility = ScrollBarVisibility.Visible;
      eddie.Document.PageWidth = 5000;
   }
   if (vertSB)
      eddie.VerticalScrollBarVisibility = ScrollBarVisibility.Auto;
   else
      eddie.VerticalScrollBarVisibility = ScrollBarVisibility.Disabled;
}
```

In der Methode `miOptions_Click()` des Beispielprogramms werden nur nach Beendigung des Optionen-Dialogfensters mit dem Rückgabewert **true** die dortigen Eigenschaften ins Anwendungsfenster übernommen.

Die Methode **ShowDialog()** gibt die Kontrolle über den aktuellen Thread erst beim Beenden des Dialogfensters an den Aufrufer zurück und berichtet dann per Rückgabewert vom Typ **bool?** (**nullable<bool>**, vgl. Abschnitt 7.3) darüber, welches Verhalten des Benutzers zum Schließen des Fensters geführt hat. Über den **ShowDialog()**-Rückgabewert erfahren wir den Wert der **DialogResult**–*Eigenschaft* des Dialogfenster-Objekts. Diese **Window**-Eigenschaft hat beim Auftritt

des Fensters den Wert **null** und wird bei jedem **Window**-Ereignis überprüft. Die Beobachtung eines von **null** verschiedenen Wertes (also **true** oder **false**) hat folgende Konsequenzen:

- Das Dialogfenster wird geschlossen.
- Die Methode **ShowDialog()** endet und liefert den **DialogResult**–Wert zurück.

Für die terminierende Wertzuweisung sorgt man z.B. in der **Click**-Behandlungsmethode zum OK-Schalter:

```
private void btnOK_Click(object sender, RoutedEventArgs e) {
   DialogResult = true;
   Close();
}
```

Eine analoge **Click**-Behandlungsmethode für die Abbrechen - Schaltfläche

```
private void btnCancel_Click(object sender, RoutedEventArgs e) {
   DialogResult = false;
   Close();
}
```

ist in unserem Beispiel überflüssig, weil wir bei diesem Schalter die **Button**-Eigenschaft **IsCancel** auf den Wert **true** gesetzt haben, damit die **Esc**-Taste das **Click**-Ereignis der Schaltfläche auslöst. Diese Einstellung hat aber auch die automatische **false**-Zuweisung an die **Window**-Eigenschaft **DialogResult** zur Folge.

Klickt der Benutzer auf die Schließen-Schaltfläche der Dialogfenster-Titelzeile, erhält **Dialog-Result** den Wert **false**, so dass **ShowDialog()** mit diesem Rückgabewert endet.

Das vollständige Projekt finden Sie im Ordner

…\BspUeb\WPF\Fenster\EddieModal

16.2.4 Übernehmen Sie keinen Aberglauben

Auch bei modalen Dialogen kann man den Benutzern den Luxus bieten, Einstellungsänderungen auf das Hauptfenster anzuwenden, ohne den Dialog beenden zu müssen. Somit erspart man dem Benutzer, bei Feinjustierungen den Dialog wiederholt aufrufen zu müssen. In der Regel bietet man diese Direktaktivierung über eine mit **Übernehmen** (engl. *Apply*) beschriftete Schaltfläche an. Bei Benutzern ist der Aberglaube verbreitet, dass ein vorhandener **Übernehmen**-Schalter auf jeden Fall vor dem Quittieren eines Dialogfensters mit **OK** betätigt werden muss, damit die vorgenommenen Einstellungen wirksam werden. Weil die abergläubische Sequenz **Übernehmen > OK** stets erfolgreich ist, wird sie lernpsychologisch zementiert. Hoffentlich kommen möglichst wenige Programmierer auf die Idee, Dialogfenster-Effekte tatsächlich vom Schalter **Übernehmen** abhängig zu machen.

Technisch gesehen stellt sich die interessante Frage, wie aus dem *modalen*, noch nicht beendeten Dialog Informationen an das aufrufende Fenster übertragen und dort verarbeitet werden sollen. Wie Sie sich vermutlich bereits gedacht haben, wird dieses Kommunikationsproblem per Ereignis-Technik gelöst:

- Man definiert in der Dialogfenster-Klasse ein öffentliches Ereignis, z.B. mit dem Namen `Apply`:

  ```
  public event EventHandler Apply;
  ```

- Auf dem Dialogfenster wird der zusätzliche Schalter **Übernehmen** eingebaut, z.B.:

  ```
  <Button Name="btnApply" Content="Übernehmen" Width="80" Grid.Row="1"
        Margin="0,20,20,20" HorizontalAlignment="Right" Click="btnApply_Click" />
  ```

- Im **Click**-Handler zum Schalter **Übernehmen** löst man das Ereignis `Apply` aus, d.h. man ruft alle bei diesem Ereignis registrierten Behandlungsmethoden (falls vorhanden) auf, z.B.:

```
private void btnApply_Click(object sender, RoutedEventArgs e) {
   if (Apply != null)
      Apply(this, EventArgs.Empty);
}
```

- In der Hauptfensterklasse definiert man eine Methode, welche den `Apply`-Delegatentyp **EventHandler** erfüllt und nach einem Klick auf den Dialogfenster-Schalter **Übernehmen** ausgeführt werden soll, z.B.:

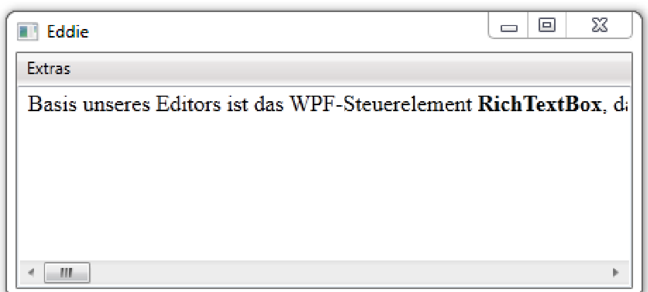
```
private void optionsDialog_Apply(object sender, EventArgs e) {
wordWrap = (sender as OptionsDialog).WordWrap;
vertSB = (sender as OptionsDialog).VertSB;
activateSettings();
}
```

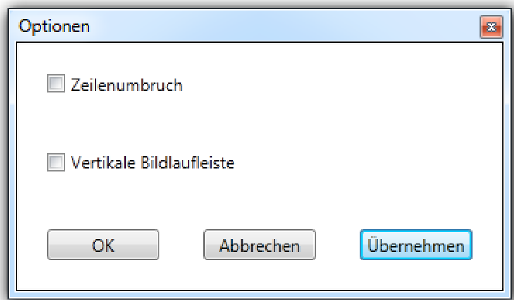
  Hier übernimmt man aus den öffentlichen Eigenschaften des Dialogfelds alle Einstellungen, die sofort umgesetzt werden sollen.

- Diese Methode wird beim `Apply`-Ereignis des Dialogfenster-Objekts registriert, z.B.:

```
od.Apply += new EventHandler(optionsDialog_Apply);
```

Benutzer unseres Editors können nun die Betriebsart ohne Zeilenumbruch inspizieren, ohne den Optionen-Dialog verlassen zu müssen:





Beim Verlassen des Dialogs mit **Abbrechen** sollen nach den Design-Empfehlungen der Firma Microsoft (2010b, S. 569) alle *noch nicht übernommenen* Einstellungsänderungen verworfen werden. Die bereits per **Übernehmen** in Kraft gesetzten Einstellungsänderungen sollen aber erhalten bleiben.

Das vollständige Projekt finden Sie im Ordner

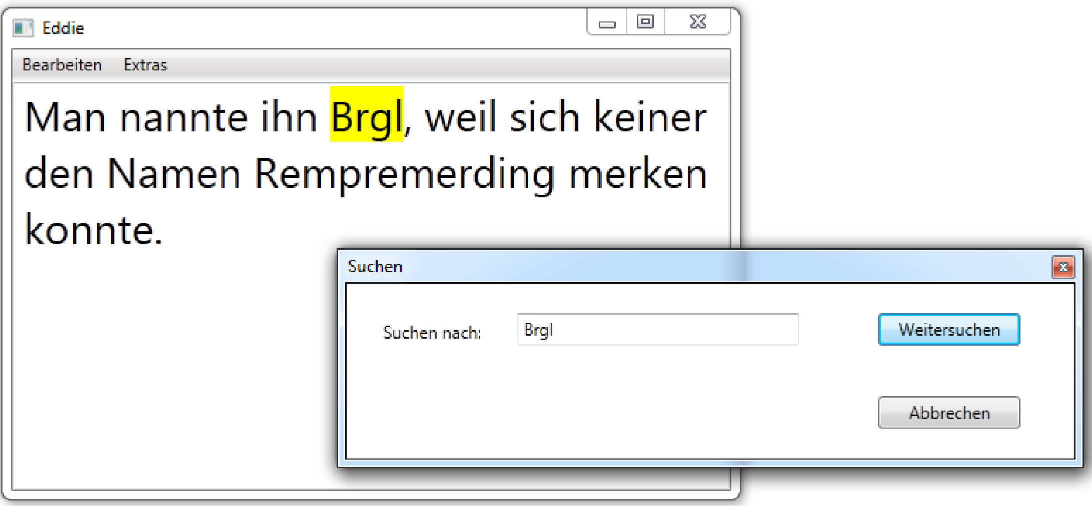
…\BspUeb\WPF\Fenster\EddieApply

16.3 Nicht-modale Dialogfenster

Viele Dialogfenster sollen (oder müssen) längere Zeit oder gar ständig offen bleiben und dürfen dabei die Funktionalität des Hauptfensters nicht behindern. Häufig anzutreffende Beispiele sind die Suchdialoge von Editoren und vergleichbaren Programmen (z.B. Entwicklungsumgebungen, E-Mail-Klienten). Man kann einem solchen Fenster fast beliebige Funktionalitäten zuordnen, so dass man eventuell nicht mehr von einem *Dialogfenster* sondern allgemeiner von einem *sekundären Fenster* reden sollte (Microsoft 2010b, S. 474).

16.3.1 Konfiguration

Der in Abschnitt 16.2 vorgestellte Editor auf **RichTextBox**-Basis soll nun mit einem einfachen Suchdialog ausgestattet werden:



Wir erstellen mit Hilfe der Entwicklungsumgebung die von **Window** abstammende Klasse `SearchDialog` mit der folgenden XAML-Deklaration:

```
<Window x:Class="Eddie.SearchDialog"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Suchen" Height="146" Width="520" ResizeMode="NoResize"
        WindowStyle="ToolWindow" WindowStartupLocation="CenterOwner">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <Button Content="Weitersuchen" Grid.Column="1" Height="23"
                HorizontalAlignment="Center" Margin="20"
                Name="btnNext" VerticalAlignment="Top" Width="101"
                IsDefault="True" Click="btnNext_Click" />
        <Button Content="Abbrechen" Grid.Column="1" Height="23"
                HorizontalAlignment="Center" Margin="20"
                Name="btnCancel" VerticalAlignment="Bottom" Width="101"
                IsCancel="True" Click="btnCancel_Click" />
        <StackPanel Orientation="Horizontal">
            <Label Height="23" HorizontalAlignment="Left" Margin="20,20,0,0"
                   Content="Suchen nach:" VerticalAlignment="Top" />
            <TextBox Height="23" HorizontalAlignment="Right" Margin="20"
                     Name="tbSearchText" VerticalAlignment="Top" Width="200" />
        </StackPanel>
    </Grid>
</Window>
```

Wir führen analog zum Optionen-Dialog (siehe Abschnitt 16.2) folgende Konfigurationsarbeiten aus:

- Die **Window**-Eigenschaft **WindowStyle** erhält den Wert **ToolWindow**.
- Die **Window**-Eigenschaft **ResizeMode** erhält den Wert **NoResize**.
- Die **Window**-Eigenschaft **WindowStartupLocation** erhält den Wert **CenterOwner**, damit der Suchdialog seine Bildschirmposition am initiierenden Fenster orientiert.
- Beim Schalter zum Weitersuchen wird die **Button**-Eigenschaft **IsDefault** auf den Wert **true** gesetzt, damit sein **Click**-Ereignis von der **Enter**-Taste ausgelöst werden kann, sofern kein anderer Schalter den Eingabefokus besitzt.
- Beim Schalter zum Abbrechen wird die **Button**-Eigenschaft **IsCancel** auf den Wert **true** gesetzt, damit sein **Click**-Ereignis von der **Esc**-Taste ausgelöst werden kann.

Wie bei den meisten nicht-modalen Dialogen fehlt auch in unserem Beispiel ein **OK**-Schalter. Stattdessen ist ein Schalter zum Starten der Suche vorhanden. Zum Beenden des Dialogs wird der

Schalter **Abbrechen** angeboten, der aber (anders als beim modalen Dialog) mit einer **Click**-Behandlungsmethode ausgestattet werden muss (siehe unten)

Bei einem nicht-modalen Dialogfenster ist es in der Regel erwünscht, dass es sich stets *über* dem Anwendungsfenster befindet, also von diesem nicht abgedeckt werden kann. Außerdem sollte das Dialogfenster beim Minimieren des Anwendungsfensters ebenfalls vom Desktop verschwinden. Um diese Verhaltensweisen anzufordern, setzen wir die **Owner**-Eigenschaft des Dialogs auf das Hauptfenster, was zwischen Objektkreation und Anzeige geschehen kann:

```
SearchDialog sd = new SearchDialog();
sd.Owner = this;
sd.Show();
```

Um den Suchdialog im Hauptfenstermenü anzubieten, wird das Item **Bearbeiten > Suchen** in die Hauptfensterdeklaration aufgenommen:

```
<MenuItem Header="Bearbeiten">
    <MenuItem Header="Suchen..." Name="miSearch" Click="miSearch_Click" />
</MenuItem>
```

16.3.2 Auftritt und Abgang

Anstelle der zur Anzeige von *modalen* Dialogen erforderlichen **Window**-Methode **ShowDialog()**, welche die Kontrolle erst beim Beenden des Dialogs zurückgibt, wird zur Anzeige eines nicht-modalen Dialogfensters die Methode **Show()** verwendet, z.B. in der **Click**-Behandlungsmethode zum Editor-Menüitem **Bearbeiten > Suchen**:

```
private void miSearch_Click(object sender, RoutedEventArgs e) {
   SearchDialog sd = new SearchDialog();
   sd.Owner = this;
   sd.FindNext += this.searchDialog_FindNext;
   sd.Show();
}
```

Sie macht das Dialogfenster sichtbar und kehrt dann sofort zurück.

Die zum Beenden von *modalen* Dialogen wichtige **Window**-Eigenschaft **DialogResult** (siehe Abschnitt 16.2.3) spielt bei nicht-modalen Dialogen *keine* Rolle. Letztere werden häufig über die **Window**-Methode **Close()** im Rahmen einer Ereignisbehandlungsmethode beendet, z.B.:

```
private void btnCancel_Click(object sender, RoutedEventArgs e) {
   Close();
}
```

Auch bei einem Mausklick auf die Schließen-Schaltfläche in der Titelzeile eines nicht-modalen Dialogfensters wird per Voreinstellung die **Close()**-Methode ausgeführt.

16.3.3 Kommunikation mit dem aufrufenden Fenster

Während ein modales Dialogfenster nur bei Anwesenheit einer **Übernehmen**-Schaltfläche vor seinem Ende Information zum übergeordneten Fenster übertragen muss, ist bei einem nicht-modalen Dialog auf jeden Fall ein solcher Informationskanal erforderlich, der meist über öffentliche Ereignisse und Eigenschaften der Dialogfenster-Klasse realisiert wird. In Abschnitt 16.3.1 war die **Click**-Behandlungsmethode zum Editor-Menüitem **Bearbeiten > Suchen** zu sehen. Dort wird ein `SearchDialog`-Objekt erzeugt und die Editor-Methode `searchDialog_FindNext()` beim `FindNext`-Ereignis des Suchdialogs registriert:

```
sd.FindNext += this.searchDialog_FindNext;
```

Mit dem Ereignis `FindNext` soll ein `SearchDialog`-Objekt alle Interessenten darüber informieren, dass der Benutzer eine Suchzeichenfolge mit dem Schalter **Weitersuchen** quittiert hat. Folglich muss die Klasse `SearchDialog` das Ereignis `FindNext` definieren und in der **Click**-Behandlungsmethode der **Weitersuchen**-Schaltfläche auslösen:

```
public event EventHandler FindNext;
    .   .   .
private void btnNext_Click(object sender, RoutedEventArgs e) {
   if (tbSearchText.Text.Length > 0 && FindNext != null)
      FindNext(this, EventArgs.Empty);
}
```

Damit der Editor (oder eine andere nutzende Klasse) die Suchzeichenfolge in Erfahrung bringen und bei anderer Gelegenheit eventuell auch setzen kann, stellt die Klasse `SearchDialog` eine öffentliche Eigenschaft zur Verfügung:

```
public string SearchText {
   get { return tbSearchText.Text;}
   set {tbSearchText.Text = value;}
}
```

In seiner eben schon erwähnten Ereignisbehandlungsmethode `searchDialog_FindNext()` ermittelt der Editor das nächste Vorkommen der Suchzeichenfolge und markiert ggf. die Trefferstelle durch eine gelbe Hintergrundfarbe:

```
private void searchDialog_FindNext(object sender, EventArgs e) {
   searchText = ((SearchDialog)sender).SearchText;
   if (tr != null)
      tr.ApplyPropertyValue(TextElement.BackgroundProperty,
                            new SolidColorBrush(Colors.White));
   FindAndReplaceManager frm = new FindAndReplaceManager(eddie.Document);
   TextPointer tp = eddie.CaretPosition;
   tr = frm.GetTextRangeFromPosition(ref tp, searchText, FindOptions.None);
   if (tr != null) {
      tr.ApplyPropertyValue(TextElement.BackgroundProperty,
                            new SolidColorBrush(Colors.Yellow));
      eddie.CaretPosition = tp;
   } else
      eddie.CaretPosition = tp.DocumentStart;
}
```

Die eigentliche Sucharbeit wird von der Klasse **FindAndReplaceManager** verrichtet, die Marco Zhou auf seiner WPF-FAQ – Seite veröffentlichet hat.[1] Bedauerlicherweise bietet die WPF-Klasse **RichTextBox** im Unterschied zum WinForms-Gegenstück noch *keine* direkte Unterstützung beim Suchen und Ersetzen.

Das vollständige Projekt finden Sie im Ordner

…\BspUeb\WPF\Fenster\EddieSearch

[1] http://social.msdn.microsoft.com/forums/en-US/wpf/thread/a2988ae8-e7b8-4a62-a34f-b851aaf13886#search_text

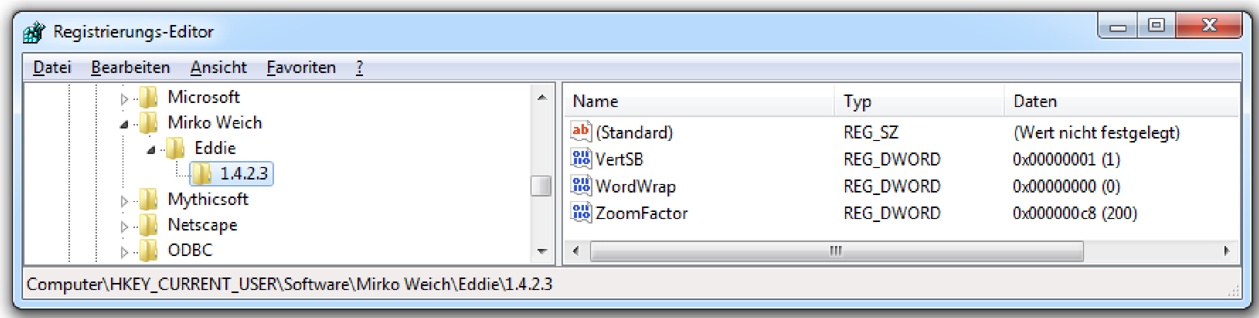
17 Anwendungs- und Benutzereinstellungen

Zum Verwalten von maschinen-, anwendungs- und benutzerbezogenen Einstellungen wurden in den ersten Jahren der Windows-Entwicklung hauptsächlich **ini**-Dateien verwendet, z.B.

```
[Path Diagram]
pen=0,1,0;0,1,0;0,1,0;0,1,0;0,1,12632256;0,1,10485760;0,1,32896;0,1,32768
brush=12632256;8421504;65280;8421376;12632256;12632256;12632256;12632256
varbar=32768,12632256
[Project]
memory=1024
toolbar=1,1,1,0,1,0,1,0
[Data]
format=8,2,2
```

Sie sind bis heute in Windows-Installationen zahlreich anzutreffen, sollten aber bei neuen .NET - Anwendungen nicht mehr eingesetzt werden.[1]

Mit Windows 95 bzw. NT wurde die universelle Registrierungsdatenbank (engl. *Registry*) eingeführt, die bis heute in der Windows-Systemtechnik eine wichtige Rolle spielt. Hier könnte auch unser Editor seine Einstellungen speichern:



Im .NET - Framework wird die Registrierungsdatenbank tendenziell gemieden. Man verwendet zur Einstellungsverwaltung wieder einzelne Dateien, nun aber im XML-Format (***E**xtended **M**arkup **L**anguage*, siehe Abschnitt 9.4.1) . Hier ist eine gekürzte Variante einer XML-Konfigurationsdatei zu unserem Editor zu sehen:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <sectionGroup name="userSettings" type=" . . . " >
            <section name="Eddie.Properties.Settings" type=" . . . "
                     allowExeDefinition="MachineToLocalUser" requirePermission="false" />
        </sectionGroup>
        <sectionGroup name="applicationSettings" type=" . . . " >
            <section name="Eddie.Properties.Settings" type=" . . . "
                     requirePermission="false" />
        </sectionGroup>
    </configSections>
    <userSettings>
        <Eddie.Properties.Settings>
            <setting name="WordWrap" serializeAs="String">
                <value>True</value>
            </setting>
            . . .
        </Eddie.Properties.Settings>
    </userSettings>
```

[1] Im **Windows**-Ordner einer Installation von Windows 7 (64 Bit) fanden sich immerhin 654 Exemplare.

```
    <applicationSettings>
        <Eddie.Properties.Settings>
            <setting name="AppName" serializeAs="String">
                <value>Äddi</value>
            </setting>
        </Eddie.Properties.Settings>
    </applicationSettings>
</configuration>
```

Im Kapitel 17 werden die beiden aktuellen Techniken zur Einstellungsverwaltung (XML-Konfigurationsdateien, Registrierungsdatenbank) bei unserem Editor-Projekt eingesetzt.

17.1 Konfigurationsdateien im XML-Format

Das .NET - Framework kennt ab Version 2.0 XML-Konfigurationsdateien für …

- Anwendungen
  Hier sind wiederum anwendungsgenerelle und benutzerspezifische Konfigurationsdateien zu unterscheiden. In einer anwendungsgenerellen Konfigurationsdatei können sich aber auch benutzerbezogene Einstellungen befinden, die als Voreinstellungen für neue Benutzer dienen. Bei den benutzerspezifischen Konfigurationsdateien ist noch zwischen lokalen und wandernden Exemplaren zu unterscheiden (siehe unten).
- Bibliotheken im Global Assembly Cache (GAC)
  Über die so genannten *Herausgeberrichtliniendateien* informiert z.B. Kühnel (2010, Abschnitt 15.4.1).
- den lokalen PC
  Die für den gesamten Rechner zuständige, von der .NET - Version abhängige Datei **machine.config** befindet sich im Ordner der betroffenen .NET - Version, z.B. (.NET 4, 64 Bit) in:

  C:\Windows\Microsoft.NET\Framework64\v4.0.30319\Config

  Die .NET - Versionen 3.0 und 3.5 legen keinen eigenen **Config**-Ordner an, verwenden also offenbar die Konfigurationsdateien der Version 2.0.

Wir beschränken uns auf Anwendungskonfigurationsdateien.

17.1.1 Aufbau einer .NET - Anwendungskonfigurationsdatei

Beim aktuellen, mit .NET 2.0 eingeführten Konfigurationssystem besitzen die zu einem Programm definierten Einstellungen

- einen **Namen**
- einen **Datentyp**
  Es sind beliebige .NET - Datentypen erlaubt.
- einen **Bezug** (Benutzer oder Anwendung)
  Anwendungsgenerelle Einstellungen können *vom Programm* nur gelesen werden. Änderungen erfordern das Bearbeiten der Konfigurationsdatei. Bei benutzerbezogenen Einstellungen (z.B. Fensterposition beim letzten Schließen) sind Lese- und Schreibzugriffe durch das Programm erlaubt.
- einen **Wert**

Die Anwendungskonfigurationsdateien verwenden das XML-Format (*Extensible Markup Language*), speichern also hierarchisch strukturierte **Elemente** in (meist) lesbarer Form.

Auf die XML-Einleitungszeile mit Versionsangabe folgt als äußerstes Element (auch als *Wurzelelement* bezeichnet) einer .NET - Konfigurationsdatei das **<configuration>** - Element:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  . . .
</configuration>
```

Wie das Beispiel **<configuration>** zeigt, besteht ein XML-Element *mit* Inhalt (z.B. mit untergeordneten Elementen) aus

- Anfangskennung
- Inhalt
- Endkennung

Als Bestandteile seiner Anfangskennung kann ein Element beliebig viele **Attribute** enthalten, die aus Name-Wert - Paaren bestehen. Die **<setting>** - Elemente der .NET - Konfigurationsdateien enthalten die Attribute **name** und **serializeAs** für den Namen und die Serialisierungsmethode einer Einstellung, z.B.:

```
<setting name="WordWrap" serializeAs="String">
  <value>False</value>
</setting>
```

Ein Element *ohne* untergeordnete Elemente (oder sonstige Inhalte) kann sich auf die Anfangskennung beschränken, die dann mit einem Schrägstrich zu enden hat, z.B. beim **<section>** - Element einer .NET - Konfigurationsdatei:

```
<section name="Eddie.Properties.Settings"
  type="System.Configuration.ClientSettingsSection, System, Version=4.0.0.0,
  Culture=neutral, PublicKeyToken=b77a5c561934e089"
  allowExeDefinition="MachineToLocalUser" requirePermission="false" />
```

Eine anwendungsgenerelle Konfigurationsdatei enthält in der Regel auch benutzerbezogene Einstellungen (als Voreinstellungen für neue Benutzer) und hat damit den folgenden Aufbau:

- Das Wurzelelement **<configuration>** enthält:
  - das Element **<configSections>** mit einer Inhaltsangabe der Datei
  - das Element **<userSettings>** für die Sektionsgruppe mit den benutzerbezogenen Einstellungen
  - das Element **<applicationSettings>** für die Sektionsgruppe mit den anwendungsbezogenen Einstellungen
- Das Element **<configSections>** enthält für die beiden Sektionsgruppen mit den benutzerspezifischen bzw. anwendungsgenerellen Einstellungen jeweils ein **<sectionGroup>** - Element:

```
<configSections>

  <sectionGroup name="userSettings"
                type="System.Configuration.UserSettingsGroup, ..." >
    . . .
  </sectionGroup>

  <sectionGroup name="applicationSettings"
                type="System.Configuration.ApplicationSettingsGroup, ..." >
    . . .
  </sectionGroup>

</configSections>
```

- Das **<sectionGroup>** - Element zu einer Sektionsgruppe, z.B. mit den benutzerbezogenen Einstellungen

```
<sectionGroup name="userSettings"
  type="System.Configuration.UserSettingsGroup, System, Version=2.0.0.0,
        Culture=neutral, PublicKeyToken=b77a5c561934e089" >
   <section name="Eddie.Properties.Settings" ... />
</sectionGroup>
```

  enthält Attribute für den Namen (im Beispiel: `userSettings`) und den Typ (im Beispiel: `System.Configuration.UserSettingsGroup`) sowie für jeden enthaltenen Abschnitt ein **<section>** - Element.

- Ein **<section>** - Element benennt für eine Sektion von Einstellungen über das **name**-Attribut die zuständige Klasse, z.B.:

```
<section name="Eddie.Properties.Settings"
 type="System.Configuration.ClientSettingsSection, System, Version=4.0.0.0,
       Culture=neutral, PublicKeyToken=b77a5c561934e089"
 allowExeDefinition="MachineToLocalUser" requirePermission="false" />
```

  Das **allowExeDefinition**–Attribut entscheidet darüber, in welchen Konfigurationsdateien die Einstellungen der Sektion definiert werden dürfen. Mögliche Werte:

  - **MachineOnly**
    Nur **Machine.config**
  - **MachineToApplication** (= Voreinstellung)
    Machine.config oder Konfigurationsdatei zur Exe-Datei
  - **MachineToRoamingUser**
    Machine.config, Konfigurationsdatei zur Exe-Datei, oder benutzerbezogene Konfigurationsdatei im wandernden Windows-Profil.
  - **MachineToLocalUser**
    Machine.config, Konfigurationsdatei zur Exe-Datei, benutzerbezogene Konfigurationsdatei im lokalen Windows-Profil oder benutzerbezogene Konfigurationsdatei im wandernden Windows-Profil.

  In Abschnitt 17.1.3 wird sich zeigen, dass für jede Sektion von Einstellungen eine von **ApplicationSettingsBase** abgeleitete Klasse zuständig ist. Weil eine solche Klasse benutzer- *und* anwendungsbezogene Einstellungen verwalten kann, taucht in der benutzerbezogenen und in der anwendungsbezogenen Sektionsgruppe eine von ihr verwaltete Sektion auf.

- Die Elemente **<userSettings>** und **<applicationSettings>** zu den beiden Sektionsgruppen haben einen analogen Aufbau. Sie enthalten für jede Sektion ein Element, das den Namen der zuständigen Klasse trägt, in unserem Fall ein Element namens **<Eddie.Properties.Settings>**, z.B.:

```
<userSettings>
  <Eddie.Properties.Settings>
    .  .  .
  </Eddie.Properties.Settings>
</userSettings>
```

- Das benutzer- bzw. anwendungsbezogene **<Eddie.Properties.Settings>** - Element enthält für jede Einstellung ein **<setting>** - Element, z.B.:

```
<Eddie.Properties.Settings>
    <setting name="WordWrap" serializeAs="String">
        <value>True</value>
    </setting>
     . . .
</Eddie.Properties.Settings>
```

- Ein **<setting>** - Element enthält
    - Attribute für den Namen und die Serialisierungsmethode einer Einstellung
    - ein **<value>** - Element für den Wert der Einstellung

Eine benutzerspezifische Konfigurationsdatei beschränkt sich auf das Element **<userSettings>**:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <userSettings>
    <Eddie.Properties.Settings>
      <setting name="WordWrap" serializeAs="String">
        <value>False</value>
      </setting>
      <setting name="VertSB" serializeAs="String">
        <value>True</value>
      </setting>
      <setting name="ZoomFactor" serializeAs="String">
        <value>200</value>
      </setting>
    </Eddie.Properties.Settings>
  </userSettings>
</configuration>
```

17.1.2 Ablage im Dateisystem

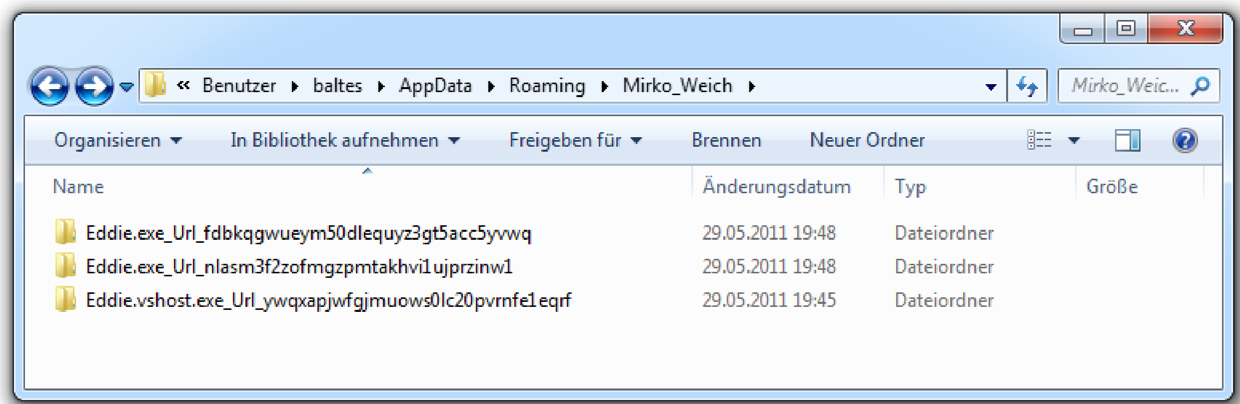
Anwendungsgenerelle Einstellungen werden in einer Datei verwaltet, die sich im selben Ordner befindet wird das Exe-Assembly und dessen Namen um den Suffix „**.config**" erweitert übernimmt. Bei einem Exe-Assembly mit dem Namen **prog.exe** heißt die Anwendungskonfigurationsdatei also **prog.exe.config**. Beachten Sie, dass beim Anwendungsstart aus der Entwicklungsumgebung das Exe-Assembly **prog.vshost.exe** und die Anwendungskonfigurationsdatei **prog.vshost.exe.config** zum Einsatz kommen.

Die benutzerbezogenen Einstellungen werden auf zwei Dateien mit dem identischen Namen **user.config** verteilt, die sich in verschiedenen Ordnern befinden:

- Per Voreinstellung landen benutzerspezifische Einstellungen unter Windows 7 im Ordner:
  <%USERPROFILE%>\Appdata\Local
  <Hersteller>\<Assembly-Name>_<Evidenz-Typ>_<Evidenz-Hash>\<Version>
- Wird eine Einstellung jedoch über das folgenden CLR-Attribut
  ```
  System.Configuration.SettingsManageabilityAttribute(
     System.Configuration.SettingsManageability.Roaming)]
  ```
  dem wandernden Windows-Benutzerprofil zugeordnet (siehe unten), landet sie unter Windows 7 im Ordner:
  <%USERPROFILE%>\Appdata\Roaming
  <Hersteller>\<Assembly-Name>_<Evidenz-Typ>_<Evidenz-Hash>\<Version>

Bei einem Standort mit zahlreichen vernetzten Windows-Rechnern (z.B. in einer Firma oder Behörde) verwendet man in der Regel eine zentralisierte Benutzerverwaltung durch mehrere Server, die als *Domänen-Controller* bezeichnet werden. Verwendet ein Domänen-Benutzer ein wanderndes (engl.: *roaming*) Windows-Benutzerprofil, befinden sich wichtige Dateien des Windows-Profils auf einem Server und werden bei jeder Anmeldung via Netzwerk zum lokalen Rechner kopiert. Von dieser Wanderbewegung sind die Dateien im Zweig **...\Appdata\Local** aber ausgenommen. Speziell bei Benutzern, die sich regelmäßig auf verschiedenen Rechnern in einer Domäne anmelden, ist es aber wichtig, dass sie für ein Programm auf allen Rechnern identische Einstellungen vorfinden. Wie man dafür sorgt, erfahren Sie später. Zunächst werden die weiteren Bestandteile im Pfad zu den benutzerspezifischen Konfigurationsdateien erläutert.

Bei nicht signierten Anwendungen (unser Fall) wird der Evidenz-Typ **URL** angegeben, und in den Evidenz-Hash geht der Installationsordner des Assemblies ein, so dass beim Starten derselben Anwendung aus verschiedenen Ordnern unterschiedliche Konfigurationsdateipfade entstehen, z.B. unter Windows 7:



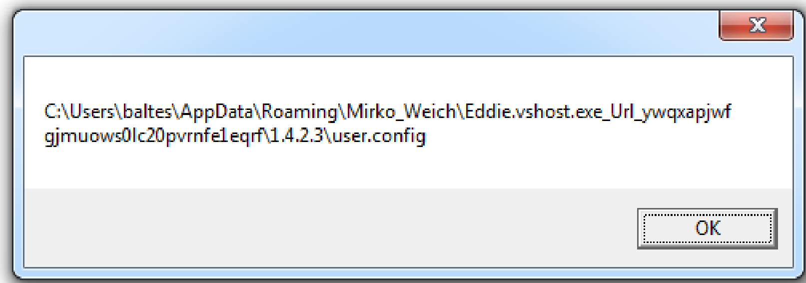
Im Beispiel lautet der Assembly-Name **Eddie.exe**; Hersteller und Version wurden in den folgenden Attribut-Deklarationen festgelegt:

```
[assembly: AssemblyCompany("Mirko Weich")]
[assembly: AssemblyVersion("1.4.2.3")]
```

Zur Modifikation dieser Attribute öffnen man die Datei **AssemblyInfo.cs** über den Projektmappenexplorer(vgl. Abschnitt 11.4)Durch den vom Installationsordner abhängigen Evidenz-Hash will Microsoft verhindern, dass zwei zufällig gleichnamige Firmen Programme mit zufällig gleichem Namen erstellen, welche dieselbe Datei **user.config** verwenden, wenn auch noch die Programmversionen unglücklicherweise übereinstimmen. Beim Anwendungsstart aus der Entwicklungsumgebung resultiert ein Konfigurationsdateipfad mit **vshost** im Namen (siehe Beispiel). Wer die garantiert kollisionsfreien Ablageorte verwirrend und benutzerunfreundlich findet, muss einen eigenen **SettingsProvider** erstellen (siehe Abschnitt 17.1.5).

Im Programm kann man die Konfigurationsdateien des aktuellen Benutzers über die Klasse **ConfigurationManager** in Erfahrung bringen.[1] Hier wird über den **OpenExeConfiguration**-Parameter **ConfigurationUserLevel.PerUserRoaming** die wandernde Version angefordert:

```
Configuration config = ConfigurationManager.OpenExeConfiguration(
ConfigurationUserLevel.PerUserRoaming);
MessageBox.Show(config.FilePath);
```



Eine benutzerspezifische Einstellungsdatei wird vom zuständigen Objekt in der **Save()**-Methode angelegt oder verändert, sofern eine Einstellungsänderung stattgefunden hat. Demgegenüber wird eine anwendungsgenerelle Konfigurationsdatei von der .NET - Konfigurationsverwaltung nur gele-

[1] Bei Verwendung der Klasse **ConfigurationManager** benötigt der Compiler (das Projekt) einen Verweis auf das Bibliotheks-Assembly **System.Configuration.dll**.

sen. Sie muss vom Software-Hersteller ausgeliefert werden, wobei man aber die Erstellung der Entwicklungsumgebung überlassen kann (siehe Abschnitt 17.1.4). Wer als Administrator für eine .NET - Software zuständig ist, muss allerdings zur Anpassung von Einstellungen die anwendungsbezogene Konfigurationsdatei per Editor öffnen und ändern, falls nicht zum Softwarepaket auch ein spezielles Konfigurationsprogramm gehört.

Um die terminologisch aufwändige und sprachlich umständliche Rede von anwendungs- bzw. benutzerbezogenen Anwendungseinstellungen zu vermeiden, sprechen wir anschließend bevorzugt über die Dateien **prog.exe.config** bzw. **user.config**.

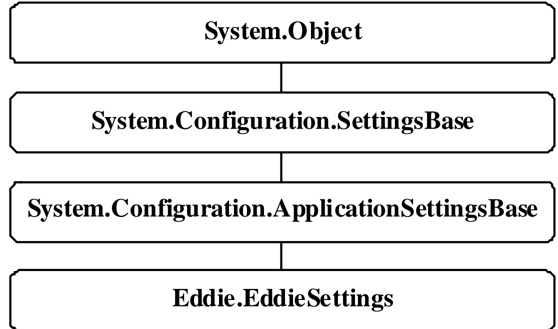
Die Datei **prog.exe.config** enthält

- das Element bzw. die Sektionsgruppe **<userSettings>**
  Hier befinden sich die Voreinstellungen für Benutzer, die (noch) keine Konfigurationsdatei **user.config** haben.
- das Element bzw. die Sektionsgruppe **<applicationSettings>**
  Hier befinden sich anwendungsgenerelle Einstellungen.

Die Datei **user.config** enthält das Element bzw. die Sektionsgruppe **<userSettings>** mit benutzerbezogenen Einstellungen. In Abhängigkeit von der Roaming-Deklaration einzelner Einstellungen, existieren eventuell zwei Dateien mit benutzerspezifischen Einstellungen.

17.1.3 Die Klasse ApplicationSettingsBase

Die FCL-Klassen zur Verwaltung von Anwendungseinstellungen befinden sich im Namensraum **System.Configuration**. Für jede Anwendung ist zur Einstellungsverwaltung (mindestens) eine Klasse aus **ApplicationSettingsBase** abzuleiten, die bei unserem Editorprojekt den Namen `EddieSettings` erhalten soll:



Ein **ApplicationSettingsBase**-Objekt enthält eine per Indexer (mit **String**-Parameter) ansprechbare Kollektion mit den Einstellungen einer Konfigurationssektion, die in der benutzer- und in der anwendungsbezogenen Sektionsgruppe der Konfigurationsdatei auftauchen darf (vgl. Abschnitt 17.1.1). In unserem Beispiel enthalten die beiden Sektionsgruppen jeweils nur eine Sektion, und wir kommen mit einem einzigen **ApplicationSettingsBase**-Objekt aus.

In der anwendungsspezifisch aus **ApplicationSettingsBase** abgeleiteten Einstellungsklasse wird für jede Einstellung (also für jedes **<setting>** - Element in den Konfigurationsdateien) eine öffentliche Eigenschaft definiert, um einen einfachen Zugriff auf die Einstellungen zu ermöglichen. In unserem Editorprojekt sollen die Einstellungen des Optionsdialogs (vgl. Abschnitt 16.2) abgespeichert werden. Außerdem soll der Administrator einer Installation die Möglichkeit erhalten, die Titelzeile der Anwendung beliebig festzulegen, so dass benutzerbezogene und anwendungsgenerelle Einstellungen vorliegen. Um den folgenden Quellcode vom Visual Studio erstellen zu lassen, sind nur wenige Angaben erforderlich (siehe Abschnitt 17.1.4):

```
namespace Eddie.Properties {
    [global::System.Runtime.CompilerServices.CompilerGeneratedAttribute()]
    [global::System.CodeDom.Compiler.GeneratedCodeAttribute(
       "Microsoft.VisualStudio.Editors.SettingsDesigner.SettingsSingleFileGenerator", "10.0.0.0")]
    internal sealed partial class Settings : global::System.Configuration.ApplicationSettingsBase {

        private static Settings defaultInstance =
          ((Settings)(global::System.Configuration.ApplicationSettingsBase.Synchronized(new Settings())));

        public static Settings Default {
            get {
                return defaultInstance;
            }
        }

        [global::System.Configuration.UserScopedSettingAttribute()]
        [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
        [global::System.Configuration.DefaultSettingValueAttribute("True")]
        [global::System.Configuration.SettingsManageabilityAttribute(
            global::System.Configuration.SettingsManageability.Roaming)]
        public bool WordWrap {
            get {
                return ((bool)(this["WordWrap"]));
            }
            set {
                this["WordWrap"] = value;
            }
        }

         . . .

        [global::System.Configuration.ApplicationScopedSettingAttribute()]
        [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
        [global::System.Configuration.DefaultSettingValueAttribute("Äddi")]
        public string AppName {
            get {
                return ((string)(this["AppName"]));
            }
        }
    }
}
```

Zunächst ist anzumerken, dass vom Assistenten alle Klassennamen voll qualifiziert geschrieben werden, wobei das Präfix

global::

eine im globalen Namensraum beginnende Namensauflösung vorschreibt. So wird z.B. verhindert, dass eine im lokalen Projekt vorhandene Definition für den Bezeichner `System` die FCL-Namensraumbezeichnung **System** verdeckt.

In der Eigenschaft zu einer Einstellung wird per Indexer mit **String**-Parameter auf den Wert zugegriffen, wobei der definierte Rückgabetyp **Object** eine explizite Typumwandlung erfordert, z.B.:

```
return ((bool)(this["WordWrap"]));
```

Um den Bezug einer Einstellung bzw. Eigenschaft festzulegen (Benutzer vs. Anwendung), wird eines von den beiden folgenden Attributen angeheftet:

- **ApplicationScopedSettingAttribute**
- **UserScopedSettingAttribute**

In der Regel wird außerdem über ein Attribut der Klasse **DefaultSettingValueAttribute** ein Voreinstellungswert festgelegt, z.B.:

```
[global::System.Configuration.DefaultSettingValueAttribute("True")]
```

Die anwendungsbezogenen Voreinstellungswerte der Einstellungsklasse kommen zum Einsatz, wenn die Datei **prog.exe.config** fehlt. Die benutzerbezogenen Voreinstellungswerte der Einstellungsklasse kommen zum Einsatz, wenn *beide* Konfigurationsdateien fehlen (**prog.exe.config**, **user.config**).

Die Eigenschaften der Einstellungsklasse erlauben den lesenden und bei Benutzerbezug auch den schreibenden Zugriff auf die Einstellungen, die in der **get**- bzw. **set**-Implementation per Indexer über ihren Namen angesprochen werden.

Zu den Extra-Kompetenzen der Klasse **ApplicationSettingsBase** im Vergleich zu ihrer Basisklasse **SettingsBase** gehören:

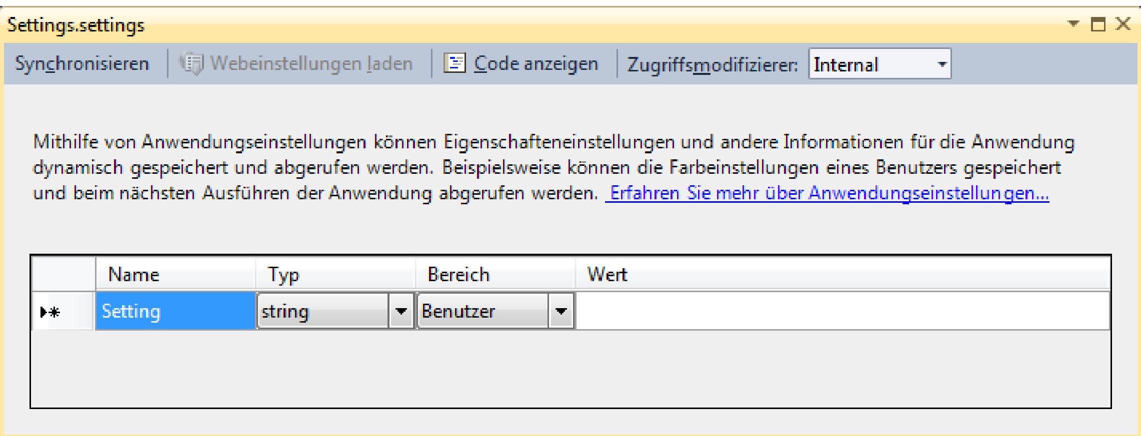
- Restaurierung der benutzerbezogenen Voreinstellungswerte über die Methode **Reset()**
- Unterstützung beim Versions-Update
- Validierung von Einstellungen vor der Änderung oder vor dem Speichern (siehe Abschnitt 17.1.6)

17.1.4 Unterstützung durch das Visual Studio

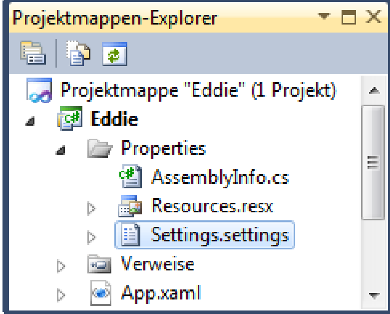
Microsofts Entwicklungsumgebungen unterstützen die Einstellungsverwaltung, indem sie …

- ein Formular zur bequemen Definition von Einstellungen bieten,
- die gem. Abschnitt 17.1.3 erforderliche **ApplicationSettingsBase**-Ableitung automatisch erstellen.

Um die Einstellungsunterstützung der Entwicklungsumgebung kennen zu lernen, wiederholen wir die Einrichtung einer Einstellungsverwaltung für das Editorprojekt. Öffnen Sie das Einstellungs-formular



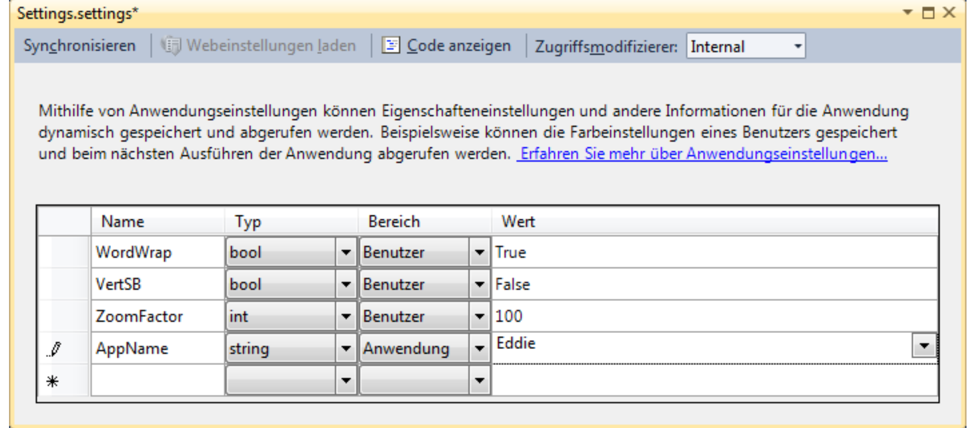
per Doppelklick auf den Eintrag **Properties > Settings.settings** im Projektmappen-Explorer:



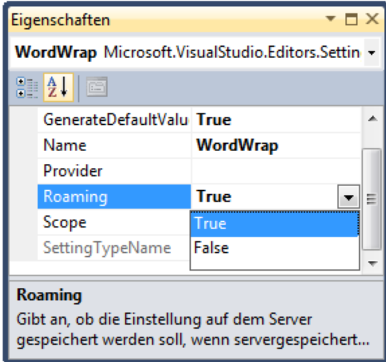
Alternativ ist das Formular auch über den folgenden Menübefehl zu erreichen:

Projekt > Eigenschaften > Einstellungen

Wir legen nun für die geplanten Einstellungen Namen, Datentyp, Bezug und Wert fest:



Für die meist empfehlenswerte Zuordnung einer benutzerspezifischen Einstellung zur wandernden Konfigurationsdatei user.config (vgl. Abschnitt 17.1.2) sorgt man bequem per Eigenschaftsdialog, z.B.:



Die Entwicklungsumgebung legt aufgrund unserer Einstellungsdefinitionen im Projektordner die Datei **app.config** mit folgendem Inhalt hat:[1]

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <sectionGroup name="userSettings" type=" . . . " >
            <section name="Eddie.Properties.Settings" type=" . . . "
                     allowExeDefinition="MachineToLocalUser" requirePermission="false" />
        </sectionGroup>
        <sectionGroup name="applicationSettings" type=" . . . " >
            <section name="Eddie.Properties.Settings" type=" . . . "
                     requirePermission="false" />
        </sectionGroup>
    </configSections>
    <userSettings>
        <Eddie.Properties.Settings>
            <setting name="WordWrap" serializeAs="String">
                <value>True</value>
            </setting>
            <setting name="VertSB" serializeAs="String">
                <value>False</value>
            </setting>
            <setting name="ZoomFactor" serializeAs="String">
                <value>100</value>
            </setting>
        </Eddie.Properties.Settings>
    </userSettings>
```

[1] Achtung: Sollten im Projekt manuell erzeugte Dateien **\bin\debug\prog.exe.config** oder **\bin\debug\prog.vs-host.exe.config** vorhanden sein, werden diese bei jedem Übersetzen durch die Datei **app.config** überschrieben!

```
        <applicationSettings>
            <Eddie.Properties.Settings>
                <setting name="AppName" serializeAs="String">
                    <value>Eddie</value>
                </setting>
            </Eddie.Properties.Settings>
        </applicationSettings>
    </configuration>
```

Außerdem legt unsere Entwicklungsumgebung zur Verwaltung der Einstellungen die von **ApplicationSettingsBase** abgeleitete Klasse **Settings** an. Der Quellcode landet in der Projektdatei **…\Properties\Settings.Designer.cs** und war auszugsweise schon in Abschnitt 17.1.3 zu sehen.

Über die statische **Settings**-Eigenschaft **Default** wird ein mit Hilfe der **ApplicationSettingsBase**-Methode **Synchronized()** erstelltes Objekt einer **Thread**-sicheren Hüllenklasse zu **Settings** angeboten. Bei Verwendung des **Default**-Objekts werden in Multithreading-Anwendungen Inkonsistenzen bei der Einstellungsverwaltung verhindert. Beim Lesen und Schreiben von Einstellungen sollten Sie dieses **Default**-Objekt verwenden, statt selbst ein **Settings**-Objekt per **new**-Operator zu erzeugen, was prinzipiell möglich wäre.

17.1.5 Lesen und Speichern von Einstellungen

Für die Übernahme von Einstellungswerten in Programmvariablen eignet sich eine Behandlungsmethode zum **Window**-Ereignis **Loaded**, z.B.:

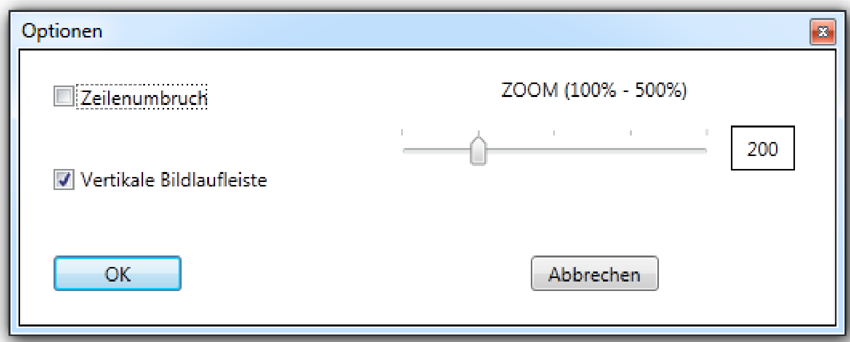
```
private void Window_Loaded(object sender, RoutedEventArgs e) {
   Keyboard.Focus(eddie);
   wordWrap = Settings.Default.WordWrap;
   vertSB = Settings.Default.VertSB;
   zoomFactor = Settings.Default.ZoomFactor;
   activateUserSettings();
}
```

Damit die Einstellungen wirksam werden, müssen Sie den betroffenen Programmvariablen bzw. Steuerelementeigenschaften zugewiesen werden. Wir verwenden für die benutzerspezifischen Einstellungen `WordWrap`, `VertSB` und `ZoomFactor` jeweils eine Instanzvariable der Fensterklasse, und die folgende Methode nimmt die Einstellungen in Betrieb:

```
void activateUserSettings() {
   if (wordWrap) {
      eddie.HorizontalScrollBarVisibility = ScrollBarVisibility.Disabled;
      eddie.Document.PageWidth = Double.NaN;
   } else {
      eddie.HorizontalScrollBarVisibility = ScrollBarVisibility.Visible;
      eddie.Document.PageWidth = 5000;
   }
   if (vertSB)
      eddie.VerticalScrollBarVisibility = ScrollBarVisibility.Auto;
   else
      eddie.VerticalScrollBarVisibility = ScrollBarVisibility.Disabled;
   eddie.LayoutTransform = new ScaleTransform(zoomFactor / 100, zoomFactor / 100);
}
```

Wie die anwendungsgenerelle Einstellung `AppName` die Hauptfenstertitelzeile beeinflusst, erfahren Sie in Abschnitt 17.1.7.

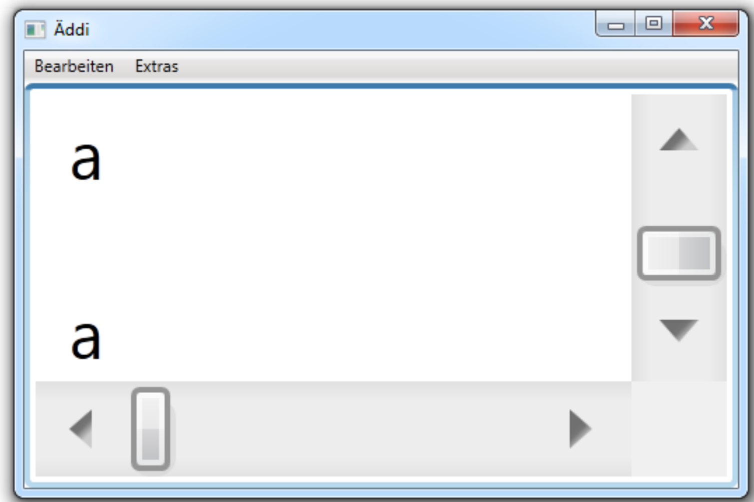
Um die Einstellungsverwaltung etwas realistischer zu machen, hat der Editor einen Zoomfaktor erhalten, der per **Slider**-Steuerelement (vgl. Abschnitt 15.8) einstellbar ist:



Wenn der **RichTextBox** - Eigenschaft **LayoutTransform** ein **ScaleTransform**-Objekt zur gleichmäßigen Streckung in X- und Y-Richtung zugewiesen wird,

```
eddie.LayoutTransform = new ScaleTransform(zoomFactor / 100, zoomFactor / 100);
```

vergrößern sich mit der Schrift leider auch die Rollbalken:



Mit den diversen Vorschlägen zur Korrektur dieses Fehlverhaltens soll die aktuelle Behandlung der Konfigurationsverwaltung aber nicht belastet werden.

Das Konfigurationssystem liest beim Programmstart die Einstellungen aus den **config**-Dateien, reagiert aber nicht auf Veränderungen dieser Dateien während der Laufzeit. Bei lang laufenden Programmen (z.B. als Windows-Dienst tätig) kann es daher sinnvoll sein, die Einstellungen über die **ApplicationSettingsBase**-Methode **Reload()** neu zu laden und anschließend die aktuellen Werte in die Programmvariablen zu übernehmen.

Über eine **ApplicationSettingsBase**-Ableitung lassen sich nur *benutzerbezogene* Einstellungen speichern. Dies stellt aber keine Einschränkung dar, weil Programme aus Sicherheitsgründen in der Regel mit einfachen Benutzerrechten ausgeführt werden, so dass ohnehin keine Schreibrechte für die Datei **prog.exe.config** bestehen. Zum Sichern der benutzerbezogenen Einstellungen eignet sich eine Behandlungsmethode zum Formularereignis **Closing**. Hier befördern wir benutzerbezogene Einstellungen in die korrespondierenden Eigenschaften des Einstellungsverwaltungs-Objekts und verlangen dann per **Save()**-Aufruf das Sichern in die Datei **user.config**:

```
private void Window_Closing(object sender, System.ComponentModel.CancelEventArgs e) {
   Settings.Default.WordWrap = wordWrap;
   Settings.Default.VertSB = vertSB;
   Settings.Default.ZoomFactor = zoomFactor;
   Settings.Default.Save();
}
```

Die von **ApplicationSettingsBase** abgeleitete Einstellungsklasse kümmert sich nicht selbst um das Speichern und Laden von Einstellungen, sondern verwendet für diese Aufgaben einen **Settings-Provider**, wobei wir den voreingestellten **LocalFileSettingsProvider** verwenden. Man kann eine eigene **SettingsProvider**-Klasse erstellen und dabei alternative Ablageorte für die Konfigurationen wählen (z.B. andere Dateien bzw. Ordner, SQL-Datenbanken).

17.1.6 Validierung von Einstellungen

Zur Validierung von Einstellungen beim Laden, Verändern und Sichern bietet die Klasse **ApplicationSettingsBase** vier Ereignisse an, so dass registrierte Behandlungsmethoden bei ungeeigneten Einstellungswerten (z.B. Geburtsdatum in der Zukunft) eingreifen können:

Ereignis	Beschreibung
SettingsLoaded	Das Ereignis tritt auf beim Laden der vom **ApplicationSettingsBase**-Objekt verwalteten Einstellungs-Sektion. Hier lässt sich also der Programmstart mit ungeeigneten Einstellungswerten verhindern. Das Ereignis ist vor allem dann relevant, wenn die Übernahme der Einstellungswerte in Programmvariablen per Data Binding automatisiert wurde (siehe Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**).
SettingChanging	Das Ereignis tritt *vor* der Veränderung einer einzelnen Einstellung auf und ermöglicht das Blockieren der Wertzuweisung. Wenn die Einstellungsdateien ausschließlich vom Programm selbst verändert werden, kann man sich bei der Validierung auf das **SettingChanging**-Ereignis beschränken.
PropertyChanged	Das Ereignis tritt *nach* der Veränderung einer einzelnen Einstellung auf und ist für eine aufwändige, asynchron auszuführende Validierung gedacht.
SettingsSaving	Das Ereignis tritt vor dem Speichern der vom **ApplicationSettingsBase**-Objekt verwalteten Einstellungs-Sektion auf. Hier lässt sich das Speichern von ungeeigneten Einstellungswerten verhindern.

Durch die folgenden (mäßig sinnvollen) Ergänzungen im Editorprojekt könnte man verhindern, dass ein **RichTextBox-ZoomFactor** größer 3 als neuer Wert der Einstellung `ZoomFactor` übernommen wird:

```
Settings.Default.SettingChanging +=
new SettingChangingEventHandler(SettingsDefault_SettingChanging);
   .  .  .
void SettingsDefault_SettingChanging(object sender, SettingChangingEventArgs e) {
   if (e.SettingName.Equals("ZoomFactor")) {
      if ((int)e.NewValue > 300) {
         e.Cancel = true;
         MessageBox.Show("Gespeichert wird der Zoom-Faktor " +
                  Settings.Default.ZoomFactor);
      } else
         MessageBox.Show("Gespeichert wird der Zoom-Faktor " + e.NewValue);
   }
}
```

Größere Zoom-Faktoren könnten weiter mit Gültigkeit für die aktuelle Sitzung per Optionsdialog gewählt, aber nicht mehr in die Datei **user.config** gespeichert werden.

17.1.7 Einstellungen per WPF-Datenbindung übernehmen

Wer bei einem WPF-Projekt eine Steuerelementeigenschaft ohne direkte Verwendung von C# - Quellcode mit einer Anwendungseinstellung verbinden will, kann die WPF-Datenbindungstechnik in der XAML-Deklaration der Fensterklasse verwenden, sofern die zu versorgende Eigenschaft von der Klasse **DependencyProperty** abstammt (vgl. Abschnitt 15.1). Wir demonstrieren dieses Ver-

fahren bei der **Window**-Eigenschaft **Title**, die mit der Anwendungseinstellung `AppName` verbunden werden soll. In der folgenden XAML-Deklaration wird zunächst der CLR-Namensraum `Eddie.Properties` mit dem Präfix `ep` deklariert:

```
<Window x:Class="Eddie.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:ep="clr-namespace:Eddie.Properties"
        Height="350" Width="525" Loaded="Window_Loaded" Closing="Window_Closing"
        Title="{Binding Source={x:Static ep:Settings.Default}, Path=AppName}">
 . . .
</Window>
```

Dann wird der **Title**-Eigenschaft über die Attributsyntax für Markuperweiterungen (siehe Abschnitt 9.4.2.7) ein **Binding**-Objekt (siehe Kapitel 15) zugewiesen, das als Datenquelle das über die statische Eigenschaft **Settings.Default** ansprechbare **Settings**-Objekt verwendet.

Das vollständige Projekt zu der in Abschnitt 17.1 vorgestellten Editorvariante mit Einstellungsverwaltung über XML-Dateien finden Sie im Ordner

...\BspUeb\Anwendungs- und Benutzereinstellungen\XML\EddieConfig

17.2 Lese- und Schreibzugriffe auf die Windows-Registrierungsdatenbank

17.2.1 Elementare Eigenschaften der Registry

Unter Windows landen viele vom Betriebssystem und von Anwendungsprogrammen verwaltete Einstellungen in einer recht umfangreichen, hierarchisch organisierten Registrierungsdatenbank (engl. *Registry*). Sie enthält sowohl benutzer- als auch installationsbezogene Daten in so genannten *Keys* (dt. *Schlüsseln*), die aufgrund der baumartigen Registry-Struktur mit den Pfaden eines Dateisystems vergleichbar sind. Im folgenden Registry-Key könnte z.B. unser Editor die vom Benutzer per Optionsdialog vorgenommenen Einstellungen speichern, damit sie das Programmende überdauern und beim nächsten Einsatz wieder in Kraft gesetzt werden können:

HKEY_CURRENT_USER\Software\Mirko Weich\Eddie\<Version>

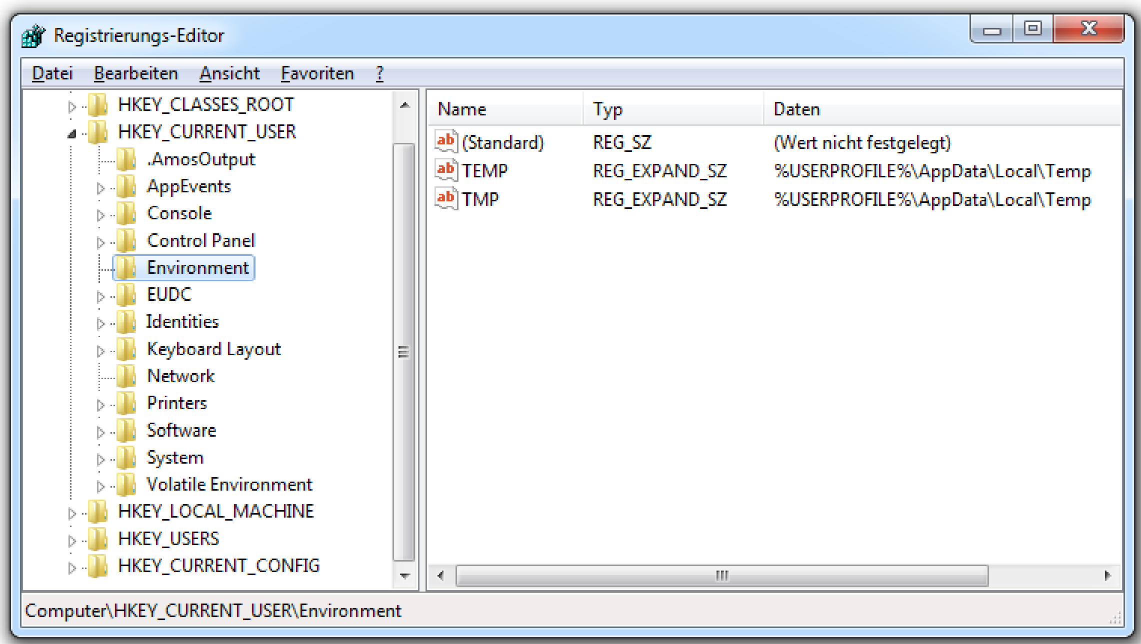
Zur Verwaltung der Einstellungen von .NET - Programmen sind eindeutig die im Abschnitt 17.2 beschriebenen Konfigurationsdateien zu bevorzugen. Das Beherrschen von Registry-Zugriffen ist aber auch für .NET - Anwendungen relevant. Indem die Einstellungsverwaltung unseres Editors nun per Registrierungsdatenbank erneut realisiert wird, können Sie die Vor- und Nachteile beider Techniken vergleichen.

In den Registry-Schlüsseln befinden sich so genannte *Values* (dt.: *Werte*) von unterschiedlichem Typ (z.B. Zeichenfolge, DWORD[1]), welche die eigentlichen Daten enthalten. Im Editor-Beispiel werden wir folgende Werte einsetzen:

Name	Typ	mögliche Daten	
WordWrap	REG_DWORD	0 1	Zeilenumbruch ist aus Zeilenumbruch ist an
VertSB	REG_DWORD	0 1	Zeilenumbruch ist aus Zeilenumbruch ist an
ZoomFactor	REG_DWORD	100 - 500	

Zur Inspektion und Modifikation der Registry kann das Windows-Administrationswerkzeug **Regedit.exe** verwendet werden:

[1] Vorzeichenfreie ganze Zahl mit 32 Bit Speicherplatz



Bei manuellen Registry-Änderungen sind Sachkenntnis und Sorgfalt erforderlich, weil Bedienungsfehler weit reichende Konsequenzen haben können. Eine zum NTFS-Dateisystem analoge Rechteverwaltung verhindert, dass Registry-Einträge durch unberechtigte Benutzer verändert werden.

Von den fünf Hauptzweigen der Registry (siehe obiges **Regedit**-Bildschirmphoto) sind für Anwendungsprogrammierer in der Regel nur **HKEY_CURRENT_USER** (mit Informationen zum angemeldeten Benutzer) und **HKEY_LOCAL_MASCHINE** (mit Daten zum System und zu den installierten Programmen) von Interesse. Wir beschränken uns im Editor-Projekt auf den Schlüssel **HKEY_CURRENT_USER**, wo der angemeldete Benutzer in der Regel uneingeschränkte Zugriffsrechte zum Schreiben und Lesen von Einstellungen besitzt. Für die (z.B. bei der Installation eines Programms sinnvollen) Schreibzugriffe auf den Zweig **HKEY_LOCAL_MASCHINE** werden administrative Benutzerrechte benötigt.

17.2.2 Registry-Bearbeitung in .NET – Programmen

Im .NET-Framework werden Registrierungs-Schlüssel durch Objekte der Klasse **RegistryKey** aus dem Namensraum **Microsoft.Win32** dargestellt, die u.a. folgende Instanzmethoden zur Verwaltung von Unterschlüsseln bietet:

- **public RegistryKey OpenSubKey(string *name*);**
  Öffnet einen vorhandenen Unterschlüssel
- **public RegistryKey CreateSubKey(string *name*);**
  Erzeugt einen Unterschlüssel
- **public void DeleteSubKey(string *name*);**
  Löscht einen Unterschlüssel
- **public void DeleteSubKeyTree(string *name*);**
  Löscht einen kompletten Teilbaum

Zur Lesen und Verändern von Werten beherrscht ein **RegistryKey**-Objekt u.a. die folgende Methoden:

- **public object GetValue(string *name*);**
  Liefert die Daten im Wert mit dem angegebenen Namen oder **null**, wenn der Wert nicht vorhanden ist
- **public void SetValue(string *name*, object *daten*);**
  Schreibt eine Zeichenfolge in den Wert mit dem angegebenen Namen
- **public void SetValue(string *name*, object *daten*, RegistryValueKind *wertSorte*);**
  Schreibt in den Wert mit dem angegebenen Namen unter Verwendung des angegebenen Registry-Wertetyps
- **public void DeleteValue(string *name*);**
  Löscht den Wert mit dem angegebenen Namen

Einen Ansprechpartner für derartige Aufträge verschafft uns die Klasse **RegistryKey** allerdings *nicht*, weil sie weder Konstruktoren noch statische Methoden zum Erzeugen von **RegistryKey**-Objekten besitzt. Die Lücke wird von der Klasse **Registry** geschlossen, die über öffentliche und statische Felder **RegistryKey**-Objekte zu den Stammknoten der Registrierungsdatenbank zugänglich macht, z.B.:

Registrierungsdatenbank-Schlüssel	Registry-Klassenvariable
HKEY_LOCAL_MASCHINE	LocalMaschine
HKEY_CURRENT_USER	CurrentUser

In unserem Editor-Projekt wird die neue Funktionalität durch Erweiterungen der Code-Behind -Datei **MainWindow.xaml.cs** realisiert. Zunächst wird der Namensraum **Microsoft.Win32** importiert:

```
using Microsoft.Win32;
```

Weil dieser Namensraum im stets bekannten Bibliotheks-Assembly **mscorlib.dll** implementiert ist, benötigt das Projekt keinen zusätzlichen Verweis.

Nach dem Namensraumimport definieren wir Assembly-Attribute zur Verwendung in Registry-Namen (vgl. Abschnitt 11.4):

```
[assembly: AssemblyCompany("Mirko Weich")]
[assembly: AssemblyProduct("Eddie")]
[assembly: AssemblyVersion("1.4.2.3")]
```

Die mehrfach benötigten Namen für den HKEY_CURRENT_USER-Unterschlüssel und die dort angesiedelten Werte verwalten wir in **String**-Instanzvariablen der Klasse `MainWindow`,

```
string strUserKey, strWordWrap, strVertSB, strZoomFactor;
```

die im Konstruktor über Reflexionstechniken (vgl. Abschnitt 11.2) mit Werten versorgt werden:

```
Assembly ass = System.Reflection.Assembly.GetExecutingAssembly();
AssemblyCompanyAttribute compName =
   (AssemblyCompanyAttribute) Attribute.GetCustomAttribute(ass, typeof(AssemblyCompanyAttribute));
AssemblyProductAttribute prodName =
   (AssemblyProductAttribute) Attribute.GetCustomAttribute(ass, typeof(AssemblyProductAttribute));
strUserKey = "Software\\" + compName.Company + "\\" + prodName.Product + "\\" + ass.GetName().Version;
strWordWrap = "WordWrap";
strVertSB = "VertSB";
strZoomFactor = "ZoomFactor";
```

17.2.2.1 In die Registry schreiben

Zum Sichern von Einstellungen in die Registry eignet sich eine Behandlungsmethode zum **Window**-Ereignis **Closing**:

```
private void Window_Closing(object sender, System.ComponentModel.CancelEventArgs e) {
   RegistryKey userKey = null;
   try {
      userKey = Registry.CurrentUser.OpenSubKey(strUserKey, true);
      if (userKey == null)
         userKey = Registry.CurrentUser.CreateSubKey(strUserKey);
      userKey.SetValue(strWordWrap, (wordWrap == true ? 1 : 0));
      userKey.SetValue(strVertSB, (vertSB == true ? 1 : 0));
      userKey.SetValue(strZoomFactor, zoomFactor);
   } catch {
      MessageBox.Show("Fehler beim Speichern der Einstellungen", Title,
         MessageBoxButton.OK, MessageBoxImage.Error);
   } finally {
      if (userKey != null)
         userKey.Close();
   }
}
```

Es wird versucht, den Registry-Key unseres Editors zum Schreiben zu öffnen. Ist der Schlüssel noch nicht vorhanden, liefert die Methode **OpenSubKey()** das Referenzliteral **null** zurück, ohne eine Ausnahme zu werfen. In diesem Fall wird der Schlüssel über die Methode **CreateSubKey()** erstellt.

Erhält die verwendete **SetValue()**-Überladung neben dem Namen des betroffenen Werts eine Zeichenfolge oder einen **int**-Wert als Aktualparameter, verwendet sie automatisch den korrekten Registry-Datentyp (Zeichenfolge bzw. DWORD). Um alternative Registry-Datentypen zu bearbeiten (z.B. Binärwert, erweiterbare Zeichenfolge), verwendet man die **SetValue()**-Überladung mit dem Parameter **RegistryValueKind**.

Ein **Regedit**-Bildschirmphoto mit dem von unserem Editor angelegten Registry-Key war schon am Anfang von Kapitel 17 zu sehen.

17.2.2.2 Aus der Registry lesen

Zum Lesen der Einstellungen eignet sich eine Behandlungsmethode zum **Window**-Ereignis **Loaded**:

```
private void Window_Loaded(object sender, RoutedEventArgs e) {
   Keyboard.Focus(eddie);
   RegistryKey userKey = Registry.CurrentUser.OpenSubKey(strUserKey);
   if (userKey != null) {
      try {
         int ww = (int) userKey.GetValue(strWordWrap);
         switch (ww) {
            case 0: wordWrap = false; break;
            case 1: wordWrap = true; break;
         }
         int vsb = (int) userKey.GetValue(strVertSB);
         switch (vsb) {
            case 0: vertSB = false; break;
            case 1: vertSB = true; break;
         }
         int zf = (int) userKey.GetValue(strZoomFactor);
         if (zf >= 100 && zf <= 500)
            zoomFactor = zf;
      } catch {
         MessageBox.Show("Fehler beim Laden der Einstellungen", Title,
            MessageBoxButton.OK, MessageBoxImage.Error);
      }
      userKey.Close();
      activateUserSettings();
   }
}
```

Weil **GetValue()** eine Rückgabe vom Typ **object** liefert, ist eine explizite Typwandlung erforderlich.

Das vollständige Projekt zur eben vorgestellten Editorvariante mit Einstellungsverwaltung per Registrierungsdatenbank finden Sie im Ordner

…\BspUeb\Anwendungs- und Benutzereinstellungen\Registry\EddieReg

17.3 Übungsaufgaben zu Kapitel 17

1) Erweitern Sie unseren Editor ausgehend vom Entwicklungsstand in

…\BspUeb\Anwendungs- und Benutzereinstellungen\XML\EddieConfigVS

so, dass auch die Position und die Größe des Anwendungsfensters beim Beenden in der benutzerspezifischen Einstellungsdatei **user.config** gesichert und beim Starten von dort gelesen werden.

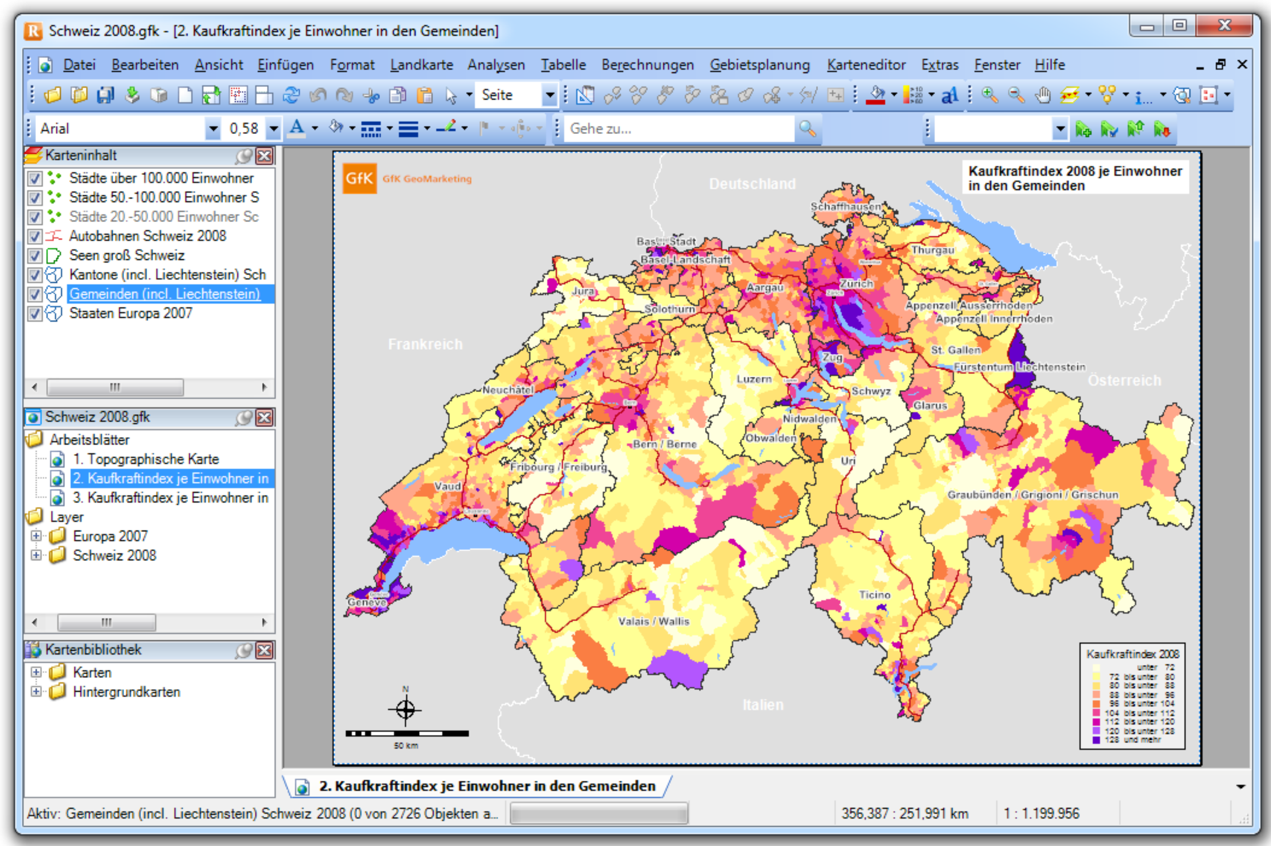
2) Erweitern Sie die Editorvariante mit Registry - basierter Einstellungsverwaltung ausgehend vom Entwicklungsstand in

…\BspUeb\Anwendungs- und Benutzereinstellungen\Registry\EddieReg

so, dass auch die Position und die Größe des Anwendungsfensters beim Beenden in der Registry gesichert und beim Starten von dort gelesen werden.

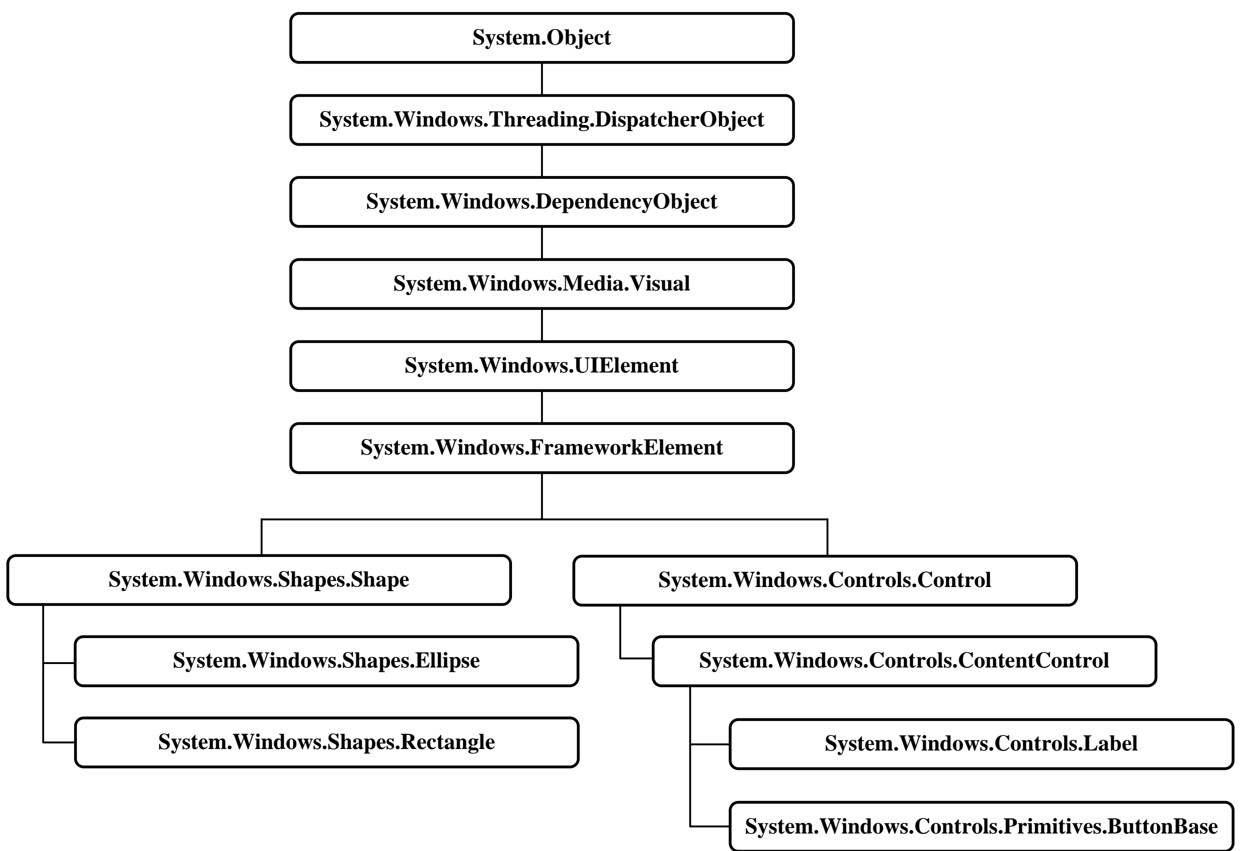
18 2D-Grafik

Zwar bieten die aus **System.Windows.Forms.Control** abstammenden Klassen zahllose Optionen zur Gestaltung von ergonomischen und ansehnlichen Programmoberflächen, doch sind nicht nur Grafikprogramme darauf angewiesen, Fenster individueller zu gestalten, als es durch Verwendung von vorgefundenen Steuerelementen und Modifikation der zugehörigen Eigenschaften möglich ist. Hier ist als Beispiel das kürzlich in .NET - Technik entwickelte Geographische Informationssystem *RegioGraph* der Firma *GfK Geomarketing* (Version 11) zu sehen, das offenbar von anspruchsvoller Grafikprogrammierung lebt:



Mehr Gestaltungsfreiheit benötigen auch Entwickler, die eigene Steuerelemente mit individueller Optik erstellen möchten (z.B. runde Befehlsschalter).

Eine der auffälligeren Neuerungen in der WPF gegenüber dem WinForms-Framework besteht darin, dass Grafikelemente (z.B. modelliert durch die Klassen **Ellipse**, **Rectangle**) in Bezug auf die Layout- und Ereignislogik mit Steuerelementen (z.B. modelliert durch die Klassen **Button**, **TextBox**) gleichbehandelt werden. Einzig die bei vielen Steuerelementen mögliche Aufnahme von untergeordneten Elementen bleibt den Grafikelementen versagt. Im langen *gemeinsamen* Stammbaum von Grafik- und Steuerelementen kommt der große Umfang an Gemeinsamkeiten zum Ausdruck:



Wegen ihrer Integration in den Baum der visuellen WPF-Elemente verursachen die **Shape**-Objekte einen Aufwand, der bei einer sehr großen Zahl zum Performanz-Problem werden kann. Daher steht in der WPF auch eine weniger aufwändige Vektorgrafik-Technologie zur Verfügung, die der traditionellen Grafik-Programmierung unter Windows über Grafikkontexte und deren Ausgabemethoden weitgehend entspricht (z.B. **DrawEllipse**(), siehe Sells & Griffiths, S. 463ff). Nach Charles Petzold (2008) sind die **Shape**-Klassen in folgenden Situationen klar zu bevorzugen:

- Es werden nur wenige Grafikelemente benötigt.
- Die Grafikelemente müssen auf Eingabeereignisse (von Maus, Tastatur oder Touchpad stammend) reagieren.
- Die Grafikelemente sollen individuellen Transformationen (z.B. Skalierung, Drehung) unterworfen werden.

Wir werden in diesem Kapitel ausschließlich mit den **Shape**-Ableitungen arbeiten.

18.1 Vektorgrafik mit Objekten aus der Shape-Klassenhierarchie

Mit den diversen Spezialisierungen der abstrakten Basisklasse **Shape** im Namensraum **System.Windows.Shapes** erhalten Sie Grafikelemente, die sich im XAML-Code wie im C# - Code analog zu Steuerelementen verwenden lassen und so eine bemerkenswert einfache Erstellung von Vektorgrafiken ermöglichen. Auch das Visual Studio unterstützt Grafikelemente in vollem Umfang, so dass man sie etwa aus der Toolbox in ein Fenster übernehmen und per WPF-Designer gestalten kann.

Im folgenden Fenster werden ein Button- und ein Ellipse-Objekt gemeinsam von einem **DockPanel**-Layoutcontainer verwaltet:

Im XAML-Code werden Ausdehnung und Positionierung der Ellipse über die vertrauten Eigenschaften bzw. Attribute geregelt:

```
<Window x:Class="ShapeEinstieg.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Shape-Einstieg" Height="191" Width="366">
    <DockPanel>
        <Button DockPanel.Dock="Top" Content="Farbe wechseln" Click="Button_Click" />
        <Ellipse Name="ellipse1" Height="100" Width="200" HorizontalAlignment="Center"
                 VerticalAlignment="Center" Stroke="Black" Fill="Red"/>
    </DockPanel>
</Window>
```

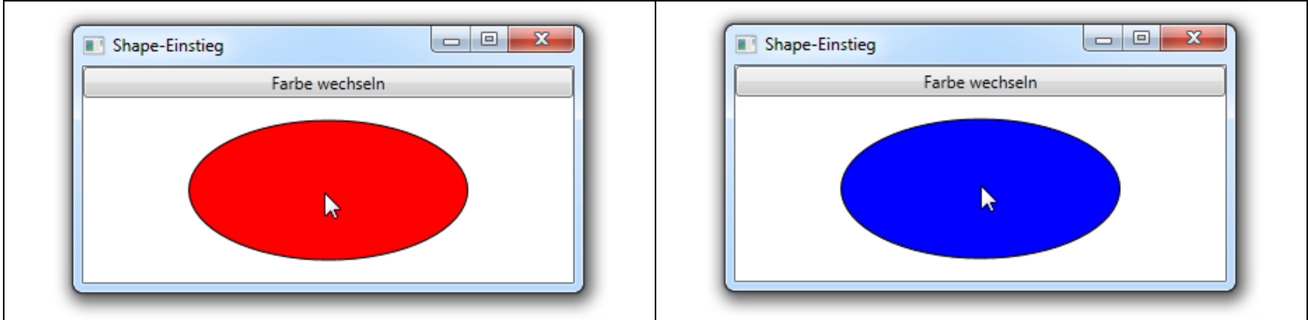
Über die gleich noch im Detail zu behandelnden Attribute **Stroke** und **Fill** lassen sich Rahmen und Fläche der Ellipse gestalten. In der **Click**-Ereignisbehandlungsmethode zum **Button**-Objekt wird zwischen einer roten und einer blauen Ellipsenfüllung umgeschaltet:

```
private void Button_Click(object sender, RoutedEventArgs e) {
   if (ellipse1.Fill == Brushes.Red)
      ellipse1.Fill = Brushes.Blue;
   else
      ellipse1.Fill = Brushes.Red;
}
```

Man benötigt aber keinesfalls die Schaltfläche, um für einen ereignisgesteuerten Farbwechsel bei der Ellipse zu sorgen. Zwar ist in der Klasse **Ellipse** kein **Click**-Ereignis definiert, doch ist z.B. das Ereignis **MouseDown** vorhanden. Mit dieser Deklaration

```
<Ellipse Name="ellipse1" Height="100" Width="200" HorizontalAlignment="Center"
  VerticalAlignment="Center" Stroke="Black" Fill="Red" MouseDown="ellipse1_MouseDown" />
```

und der entsprechenden Implementation lässt sich der Farbwechsel auch per Mausklick auf die Ellipse anfordern:



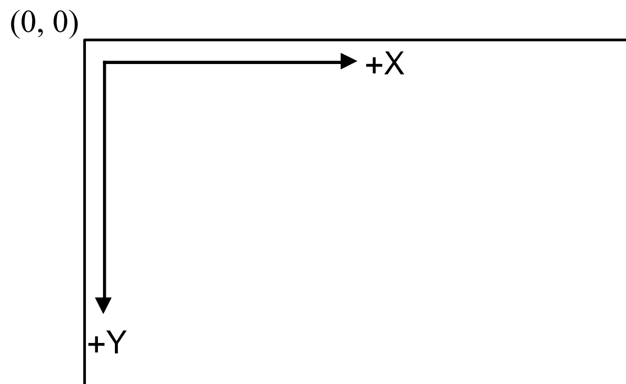
In Abschnitt 18.1.2 werden Sie weitere Erfahrungen mit dem kompetenten und selbständigen Verhalten von WPF- Grafikelementen machen.

18.1.1 Canvas-Layoutcontainer und das metrische WPF-Koordinatensystem

Zwar sorgt die WPF-Layoutlogik mit automatischer Berechnung von Positionen und Ausdehnungen durch intelligente Layoutcontainer oft für flexible und ergonomische Benutzeroberflächen, doch sind bei einer aus mehreren Elementen bestehenden Grafik konstante Positionen und Größen erforderlich. Daher setzt man Grafiken oft in einen Layoutcontainer vom Typ **Canvas**, weil diese Klasse auf jede Layoutlogik verzichtet, so dass im folgenden Beispiel die **Ellipse**-Attribute **HorizontalAlignment** und **VerticalAlignment** ohne Effekt bleiben:

```
<Canvas>
  <Ellipse Name="ellipse1" Height="100"
    Width="200" HorizontalAlignment="Center"
    VerticalAlignment="Center"
    Stroke="Black" Fill="Red"/>
</Canvas>
```

Bei den vom Entwickler zu bestimmenden Positionen und Größen liegt ein zweidimensionales Koordinatensystem mit dem Ursprung (0, 0) in der linken oberen Ecke des umgebenden Containers zu Grunde. Die **X**-Werte wachsen nach rechts und die **Y**-Werte nach unten:



Alle Orts- und Entfernungsangaben erfolgen geräteunabhängig mit einer Einheit von 1/96 Zoll (≈ 0,265 mm). Weil man unter Windows bei einem Bildschirm traditionell eine Auflösung von 96 dpi (*dots per inch*) annimmt, ist dort ein Pixel gerade 1/96 Zoll breit und hoch. Daher bezeichnet man die WPF-Maßeinheit meist als *Pixel*, obwohl z.B. eine 96 Pixel lange Linie auf jedem Bildschirm und Drucker unabhängig von der Auflösung eine Länge von 2,54 cm besitzen sollte. In der Praxis wird diese Erwartung oft nicht erfüllt, weil weder Windows noch die WPF die exakte Auflösung eines Bildschirms kennen.

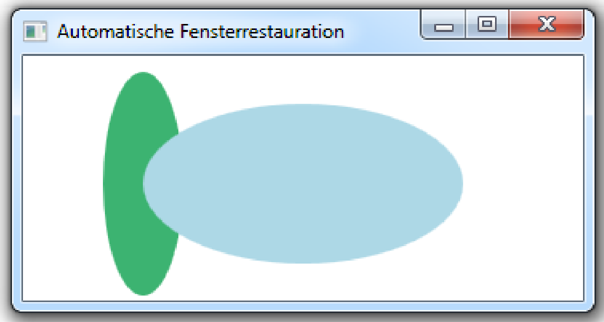
18.1.2 Automatische Fensterrestauration

Für Kenner der früheren Windows- bzw. .NET - APIs zur Grafik-Programmierung (siehe z.B. Baltes-Götz 2010b, Kap. 15) ist es vermutlich sehr beeindruckend, WPF- Grafikelemente als kompetente, selbständig agierende Objekte zu erleben. Während in WinForms-Programmen Grafik- und Steuerelementen sehr verschiedene Programmiertechniken benötigen, benehmen sich in WPF-Programmen die **Shape**-Objekte genauso intelligent wie die **Control**-Objekte:

- Beim Verschieben von **Shape**-Objekten müssen Sie sich *nicht* um die Beseitigung von Spuren am alten Ort kümmern.
- Bei einer Vergrößerung der sichtbaren Fläche eines WPF- Grafikelements (z.B. wegen der Rückkehr des Fensters aus der Windows-Taskleiste oder beim Verschwinden einer vorherigen Abdeckung durch ein anderes Fenster) findet eine Oberflächenrestauration ohne Beteiligung des Programmierers statt. Hinter dieser Restaurationsautomatik steckt WPF-Logik,

wobei durchaus nicht nur Bitmap-Daten gespeichert und restauriert werden (siehe Sells & Griffiths 2007, S. 399). In einer WinForms-Anwendung müssen demgegenüber in einer solchen Situation die beteiligten Fenster-Objekte über ihre **OnPaint**()-Methode für die Renovierung sorgen, also ihre Oberfläche neu zeichnen.

Zur Demonstration der bequemen und sorglosen WPF-Grafikprogrammierung dient eine simple Anwendung mit einer Ellipse, die per Mausklick nach links oder rechts bewegt werden kann und dabei eine andere Ellipse überquert:



Im XAML-Code wird werden die beiden Ellipsen in einem **Canvas**-Layoutcontainer über die angefügten Eigenschaften **Canvas.Top** und **Canvas.Left** positioniert:

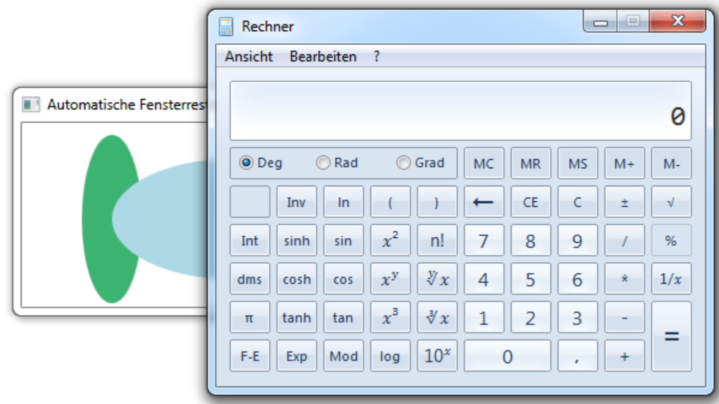
```
<Window x:Class=" AutoWindowRetauration.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Automatische Fensterrestauration" Height="191" Width="366">
    <Canvas Name="canvas">
        <Ellipse Name="ellipse2" Height="140" Width="50" Canvas.Top="10" Canvas.Left="50"
                 Fill="MediumSeaGreen" />
        <Ellipse Name="ellipse1" Height="100" Width="200" Canvas.Top="30" Canvas.Left="0"
                 Fill="LightBlue" MouseDown="ellipse1_MouseDown" />
    </Canvas>
</Window>
```

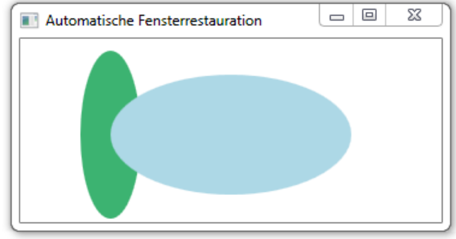
Bei der oberen Ellipse wird per **MouseDown**-Ereignisbehandlung für Beweglichkeit gesorgt:

```
private void ellipse1_MouseDown(object sender, MouseButtonEventArgs e) {
  if (e.LeftButton == MouseButtonState.Pressed) {
   double pos = Canvas.GetLeft(ellipse1);
   if (pos > 0)
    Canvas.SetLeft(ellipse1, pos / 2);
  }
  if (e.RightButton == MouseButtonState.Pressed) {
   double pos = Canvas.GetLeft(ellipse1);
   if (pos + ellipse1.ActualWidth < canvas.ActualWidth)
    Canvas.SetLeft(ellipse1,pos+(canvas.ActualWidth-(pos+ellipse1.ActualWidth))/2);
  }
}
```

Während sich der Programmierer ausschließlich um die gewünschte Neupositionierung der oberen Ellipse kümmern muss, sorgt die WPF für das korrekte Zeichnen der freigelegten unteren Ellipse.

Wenn die gesamte Anwendung im passiven Zustand nach einer vorheriger (partiellen Abdeckung) sichtbar wir, sorgt die WPF (ohne Beteiligung einer anwendungseigenen Renovierungsmethode) für eine perfekte Oberfläche:





18.1.3 Spezielle Eigenschaften von Shape-Objekten

In diesem Abschnitt werden Eigenschaften vorgestellt, die **Shape**-Objekte im Vergleich zu Steuerelementen zusätzlich benötigen. Modelliert ein **Shape**-Objekt eine Figur mit einem umschlossenen inneren, ist über den Wert der Eigenschaft **Fill** ein Objekt der Klasse **Brush** (dt.: *Pinsel*) zu wählen, das die Oberflächengestaltung übernimmt. Im einfachsten Fall sorgt ein Objekt der Klasse **SolidColorBrush** für eine homogene Färbung, z.B. aufgrund der folgenden XAML-Syntax aus obigen Beispielen:

```
Fill="Red"
```

Wir werden uns in Abschnitt 18.1.5 ausführlich mit der Klasse **Brush** beschäftigen. Beim voreingestellten **Fill**-Wert **null** ist das Innere der Figur transparent.

Für die Rahmengestaltung ist ein Objekt der Klasse **Pen** (dt.: *Stift*) zuständig. Es kümmert sich z.B. um die Breite und den Stil der Randlinie. Für den Farbauftrag verwendet die Klasse **Pen** ein **Brush**-Objekt. Ein **Shape**-Objekt besitzt keine Eigenschaft zur Verwaltung einer **Pen**-Referenz, sondern kapselt das **Pen**-Objekt und macht die **Pen**-Eigenschaften über korrespondierende **Shape**-Eigenschaften zugänglich:

Eigenschaft	Beschreibung
Stroke	Diese Eigenschaft ist vom Typ **Brush** und legt den Pinsel zu Randbemalung fest, z.B.: `Stroke="Black"` Beim voreingestellten Wert **null** ist kein Rand vorhanden.
StrokeThickness	Diese **double**-Eigenschaft legt die Breite der Umrisslinie fest, z.B.: `StrokeThickness="10"`
StrokeDashArray	Mit dieser Auflistung von **double**-Werten definiert man einen Linienstil, z.B.: `StrokeDashArray="2.5,0.5,1,0.5"` Ergebnis:

Eigenschaft	Beschreibung
StrokeDashCap	Mit dieser Eigenschaft vom Aufzählungstyp **PenLineCap** wählt man für einen Linienstil die Form der Segmentbegrenzungen, z.B.: `StrokeDashCap="Round"` Ergebnis:
StrokeDashOffset	Diese Eigenschaft legt fest, wo ein Strichmuster startet, z.B.: `StrokeDashOffset="0"` `StrokeDashOffset="7"`

StrokeStartLineCap, StrokeEndLineCap	Mit diesen Eigenschaften vom Aufzählungstyp **PenLineCap** wählt man für eine Linie Start- bzw. Endgestaltung, z.B.:
	`<Line X1="20" Y1="20" X2="300" Y2="100" Stroke="Blue"`
	`StrokeThickness="20"`
	`StrokeStartLineCap="Round" StrokeEndLineCap="Triangle"/>`
	Ergebnis:

StrokeLineJoin	Diese Eigenschaft vom Aufzählungstyp **PenLineJoin** legt fest, wie Liniensegmente verbunden werden sollen. Ohne Angabe
	`<Polyline Points="10,20 100,20 200,100" Stroke="Blue"`
	`StrokeThickness="20" />`
	erhält man die Voreinstellung **Miter**:
	Die gerundete Version
	`<Polyline Points="10,20 100,20 200,100" Stroke="Blue"`
	`StrokeThickness="20" StrokeLineJoin="Round"/>`
	sieht so aus:

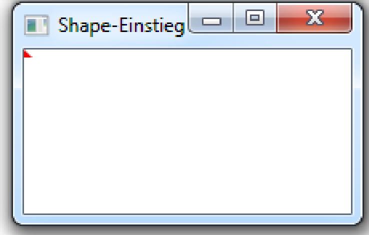
Eigenschaft	Beschreibung
StrokeMiterLimit	Mit dieser **double**-Eigenschaft kann man das Verhältnis aus Gehrungslänge und Linienstärke begrenzen. Ein eventuell unerwünschtes Verhältnis größer Eins resultiert, wenn zwei Linien in spitzem Winkel aufeinander treffen, z.B.: So `<Polyline Points="10,20 100,20 50,100" Stroke="Blue" StrokeThickness="20" StrokeMiterLimit="1"/>` lässt sich der maximale Gehrungsquotient ein Eins begrenzen:

Mit der **Stretch**-Eigenschaft legt man fest, wie ein **Shape**-Objekt den verfügbaren Platz (z.B. per **Height**- und **Width**-Eigenschaft angefordert oder von einem intelligenten Layoutcontainer wie **Grid** zugewiesen) nutzt. Es sind vier Werte möglich:

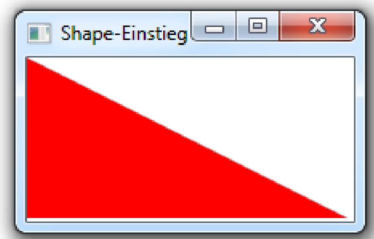
- **None**
  Keine Anpassung an den verfügbaren Platz. Im folgenden Beispiel mit einem winzigen Polygon mit hohem Raumangebot

```
<Window x:Class="ShapeEinstieg.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Shape-Einstieg" Height="140" Width="220">
    <Canvas>
        <Polygon Points="0,0 0,5 5,5" Width="200" Height="100" Fill="Red"
                 Stretch="None" />
    </Canvas>
</Window>
```

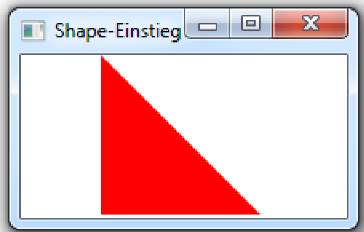
  bleibt die Figur winzig:



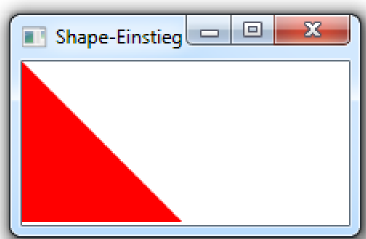
- **Fill**
  Mit dem **Stretch**-Wert **Fill** ermuntert man das Polygon die verfügbare Fläche (`Width="200" Height="100"`) komplett zu nutzen, wobei eine nichtproportionale Streckung erfolgt:



- **Uniform**
  Mit dem **Stretch**-Wert **Uniform** ermuntert man das Polygon die verfügbare Fläche komplett zu nutzen, wobei eine proportionale Streckung erfolgt:



- **UniformToFill**
  Mit dem **Stretch**-Wert **UniformToFill** ermuntert man das Polygon die verfügbare Fläche komplett zu nutzen, wobei eine proportionale Streckung erfolgt. Im Beispiel (deklarierte Breite: 5, verfügbare Breite: 200) werden beide Seiten mit dem Faktor 40 gestreckt. Weil die verfügbare Höhe (100) nicht ausreicht, wird die Figur abgeschnitten:



18.1.4 Konkrete Shape-Ableitungen

Mit den (versiegelten) **Shape**-Ableitungen **Line**, **Rectangle**, **Ellipse**, **Polyline**, **Polygon** und **Path** lassen sich die meisten Aufgaben aus dem Bereich der Vektorgrafik bewältigen.

18.1.4.1 Line

Bei einem **Line**-Objekt legt man den Start- bzw. Endpunkt über die Eigenschaften **X1**/**Y1** bzw. **X2**/**Y2** fest, z.B.:

```
<Canvas>
    <Line X1="10" Y1="10" X2="180" Y2="80" StrokeThickness="5" Stroke="Blue"/>
</Canvas>
```

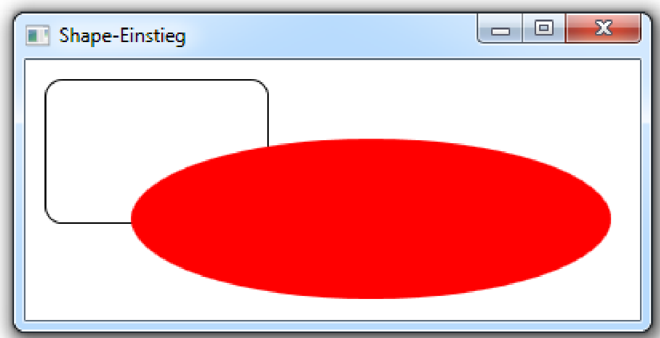
Als **Stretch**-Wert ist **None** voreingestellt.

18.1.4.2 Rectangle und Ellipse

Bei den Klassen **Rectangle** und **Ellipse** legt man die Größe über die Eigenschaften **Width** und **Height** fest. Zum Positionieren auf einer **Canvas**-Zeichenfläche eignen sich die angefügten Eigenschaften **Canvas.Top** und **Canvas.Left**, z.B.:

```
<Canvas>
    <Rectangle Height="90" Width="140" Canvas.Left="12" Canvas.Top="12"  Stroke="Black"
               RadiusX="10" RadiusY="10"/>
    <Ellipse Height="100" Width="300" Canvas.Left="66" Canvas.Top="49" Fill="Red" />
</Canvas>
```

Über die **Rectangle**-Eigenschaften **RadiusX** und **RadiusY** realisiert man abgerundete Ecken:



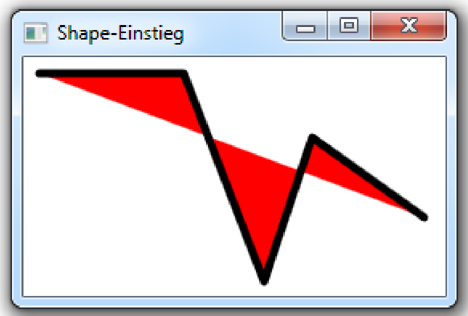
Als **Stretch**-Wert ist **Fill** voreingestellt.

18.1.4.3 Polyline und Polygon

Bei Objekten der Klassen **Polyline** und **Polygon** definiert man einen Linienzug über ein Objekt vom Typ **PointCollection**, das der **Points**-Eigenschaft zugewiesen wird. In der XAML-Deklaration lässt sich das Kollektionsobjekt über eine Zeichenfolge mit intuitiver Syntax definieren, z.B.:

```
<Canvas>
    <Polyline Points="10,10 100,10 150,140 180,50 250,100"
             Stroke="Black" StrokeThickness="5" Fill="Red"
             StrokeStartLineCap="Round" StrokeEndLineCap="Round" StrokeLineJoin="Round"/>
</Canvas>
```

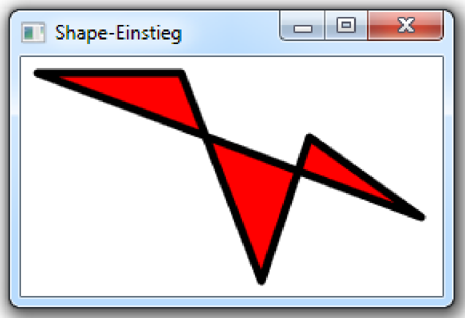
Auch eine Polylinie lässt sich füllen:



Macht man aus der Polylinie ein Polygon,

```
<Polygon Points="10,10 100,10 150,140 180,50 250,100" . . . />
```

wird die Figur automatisch geschlossen:



18.1.4.4 Path

Mit der Klasse **Path** lassen sich auch komplexe Figuren realisieren. Ihre Instanzeigenschaft **Data** zeigt auf ein Objekt aus der Hierarchie zur abstrakten Basisklasse **Geometry** im Namensraum **System.Windows.Media**, das den Umriss der zu zeichnenden Figur beschreibt. Es sind die folgenden Ableitungen vorhanden:

Geometry-Ableitung	Beschreibung
LineGeometry, **RectangleGeometry**, **EllipseGeometry**	Ein Objekt beschreibt eine Linie, ein Rechteck oder eine Ellipse. Auf den ersten Blick erscheinen die Klasse **RectangleGeometry** und **EllipseGeometry** überflüssig, weil mit den **Shape**-Ableitungen **Rectangle** und **Ellipse** bequeme Hüllenklassen zur Verfügung stehen. Gleich lernen Sie aber eine sinnvolle direkte Verwendung bei der Definition einer Geometriegruppe kennen.

GeometryGroup	Aus mehreren **Geometry**-Objekten lässt sich ein **GeometryGroup**-Objekt erzeugen und der **Data**-Eigenschaft eines **Path**-Objekts zuweisen, z.B.:
	`<Path Fill="Goldenrod" Stroke="Black" Name="gg">`
	`  <Path.Data>`
	`    <GeometryGroup>`
	`      <EllipseGeometry Center="140, 110" RadiusX="100" RadiusY="30" />`
	`      <EllipseGeometry Center="80, 110" RadiusX="15" RadiusY="15" />`
	`      <EllipseGeometry Center="140, 110" RadiusX="25" RadiusY="25" />`
	`      <EllipseGeometry Center="200, 110" RadiusX="15" RadiusY="15" />`
	`    </GeometryGroup>`
	`  </Path.Data>`
	`</Path>`
	Es resultiert ein komplexes **Path**-Objekt,
	das z.B. auf Befehl eine Rotation ausführen kann.
	GeometryGroup

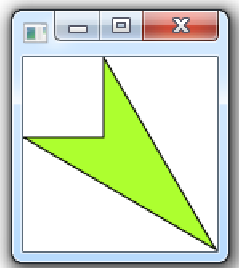
Geometry-Ableitung	Beschreibung
CombinedGeometry	Ein Objekt aus dieser Klasse kombiniert zwei Geometry-Objekte, wobei z.B. eine Vereinigung oder ein Durchschnitt gebildet wird.
PathGeometry	Ein Objekt aus dieser Klasse beschreibt eine komplex aufgebaute Figur (siehe unten).
StreamGeometry	Diese Klasse bietet dieselbe Flexibilität wie die Klasse **PathGeometry**. Man erzielt eine bessere Effizienz, kann die Objekte aber nicht mehr verändern.

Ein **PathGeometry**-Objekt definiert eine komplexe Gestalt, die aus mehreren **PathFigure**-Objekten bestehen kann, gesammelt in einem Objekt der Klasse **PathFigureCollection**, auf das die **PathGeometry**-Eigenschaft **Figures** zeigt. Ein einzelnes **PathFigure**-Objekt enthält:

- einen Startpunkt vom Strukturtyp **Point**, ansprechbar über die Eigenschaft **StartPoint**,
- eine Liste von Segmenten, auf welche die Eigenschaft **Segments** zeigt.

Die Eigenschaft **IsClosed** entscheidet darüber, ob die Figur abgeschlossen werden soll.

Zur Demonstration erstellen wir einen speziellen Pfeil, der in einer Windrose Verwendung finden könnte:



Die XAML-Deklaration:

```
<Canvas>
    <Path Fill="GreenYellow" Stroke="Black">
        <Path.Data>
            <PathGeometry>
                <PathGeometry.Figures>
                    <PathFigure StartPoint="120,120" IsClosed="True">
                        <PathFigure.Segments>
                            <LineSegment Point="50,0" />
                            <LineSegment Point="50,50" />
                            <LineSegment Point="0,50" />
                        </PathFigure.Segments>
                    </PathFigure>
                </PathGeometry.Figures>
            </PathGeometry>
        </Path.Data>
    </Path>
</Canvas>
```

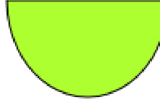
Für die Segmente eines **PathFigure**-Objekts stehen folgende Ableitungen der abstrakten Klasse **PathSegment** zur Verfügung:

PathSegment-Ableitung	Beschreibung
LineSegement	Ein Objekt aus dieser Klasse repräsentiert eine gerade Linie zwischen zwei Punkten einer Pfadfigur.
PolyLineSegment	Ein Objekt aus dieser Klasse repräsentiert eine Polylinie als Segment in einer Pfadfigur.
ArcSegment	Ein Objekt aus dieser Klasse repräsentiert einen Teil aus einem Ellipsenumriss als Segment in einer Pfadfigur. Mit den folgenden **ArcSegment**-Eigenschaften wird der gewünschte Ellipsenausschnitt eindeutig bestimmt: **Point** Endpunkt des Bogens **Size** die beiden Radien der Ellipse **SweepDirection** Bogen im Uhrzeigersinn oder umgekehrt **IsLargeArc** Bogen größer als 180°?
BezierSegment	Ein Objekt aus dieser Klasse repräsentiert einen *kubischen Bézier-Spline* als Segment in einer Pfadfigur (siehe unten). Ein kubischer Bézier-Spline wird durch den Startpunkt, den Endpunkt sowie zwei Kontrollpunkte definiert (siehe unten).
PolyBezierSegment	Ein Objekt aus dieser Klasse repräsentiert eine Serie hintereinander gehängter *Bézier-Splines* als Segment in einer Pfadfigur (siehe unten).
QuadraticBezierSegment, **PolyQuadraticBezierSegment**	Quadratische Bézier-Splines besitzen im Unterschied zur kubischen Variante nur einen Kontrollpunkt.

Als Beispiel für die Verwendung der Klasse **ArcSegment** erstellen wir mit dem folgenden XAML-Code

```
<Path Fill="GreenYellow" Stroke="Black">
  <Path.Data>
    <PathGeometry>
      <PathGeometry.Figures>
        <PathFigure StartPoint="20,20" IsClosed="True">
          <PathFigure.Segments>
            <LineSegment Point="120,20" />
            <ArcSegment Point="20,20" Size="50,60" SweepDirection="Clockwise"
                        IsLargeArc="True" />
          </PathFigure.Segments>
        </PathFigure>
      </PathGeometry.Figures>
    </PathGeometry>
  </Path.Data>
</Path>
```

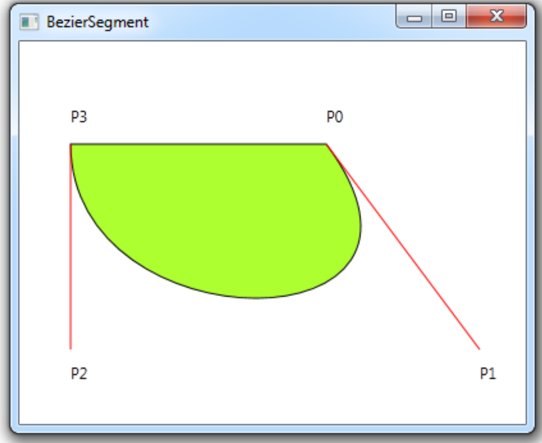
ein halbes Ei:



Bei einem *Bézier-Spline* sind vier Punkte im Spiel. Ausgehend vom Punkt P0 wird die Kurve zunächst vom Kontrollpunkt P1, dann vom Kontrollpunkt P2 angezogen, um schließlich im Punkt P3 zu enden, z.B.:

```
<Canvas>
  <Path Fill="GreenYellow" Stroke="Black">
    <Path.Data>
        <PathGeometry>
          <PathGeometry.Figures>
           <PathFigure StartPoint="40,80" IsClosed="True">
            <PathFigure.Segments>
              <LineSegment Point="240,80" />
              <BezierSegment Point1="360,240" Point2="40,240" Point3="40,80"/>
            </PathFigure.Segments>
          </PathFigure>
        </PathGeometry.Figures>
      </PathGeometry>
    </Path.Data>
  </Path>
</Canvas>
```

Im XAML-Code wurden der Übersichtlichkeit halber die Deklarationen zum Zeichen der Punkte und Hilfslinien weggelassen:



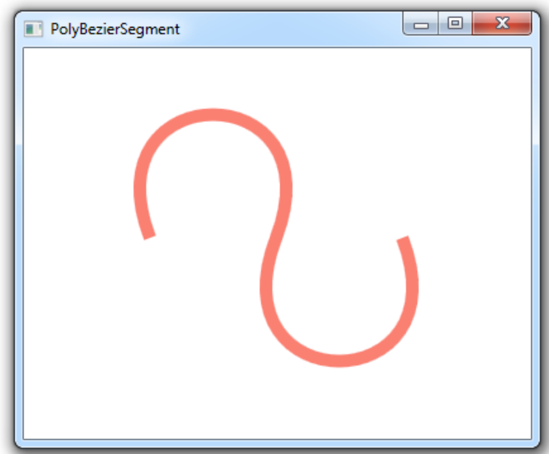
Ein **PolyBezierSegment**-Objekt enthält eine Sequenz von Bézier-Splines, z.B.:

```
<PolyBezierSegment Points="50,20 250,20 200,150 150,280 350,280 300,150" />
```

Damit die Übergänge glatt verlaufen, müssen für zwei benachbarte Splines folgende Punkte auf einer Geraden liegen:

- zweiter Kontrollpunkt des ersten Splines
- Grenzpunkt
- erster Kontrollpunkt des zweiten Splines

Im Beispiel ist diese Bedingung erfüllt:

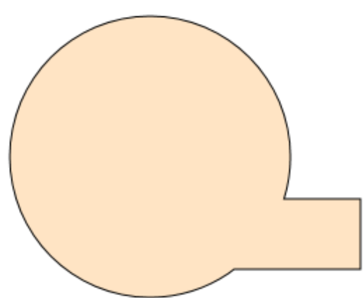


Bei einem **Path**-Objekt mit Gestaltdefinition per **GeometryGroup**-Objekt (siehe oben) bleiben alle Umrisslinien vorhanden und definieren gemeinsam das Innere der Figur. Demgegenüber entsteht bei der Gestaltdefinition per **CombinedGeometry**-Objekt *eine* neue Umrisslinie nach verschiedenen Prinzipien, wählbar über die Eigenschaft **GeometryCombineMode**:

- **Union**
  Die beiden **Geometry**-Objekte werden vereinigt, z.B. bei dieser

```
<CombinedGeometry GeometryCombineMode="Union">
    <CombinedGeometry.Geometry1>
        <EllipseGeometry Center="150,200" RadiusX="100" RadiusY="100" />
    </CombinedGeometry.Geometry1>
    <CombinedGeometry.Geometry2>
        <RectangleGeometry Rect="200,230,100,50"/>
    </CombinedGeometry.Geometry2>
</CombinedGeometry>
```

  Schiedsrichterpfeife:



- **Intersect**
  Nur die Schnittmenge aus beiden **Geometry**-Objekte verbleibt, z.B. bei dieser

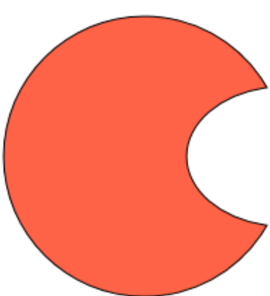
```
<CombinedGeometry GeometryCombineMode="Intersect">
    <CombinedGeometry.Geometry1>
        <EllipseGeometry Center="200,200" RadiusX="180" RadiusY="180" />
    </CombinedGeometry.Geometry1>
    <CombinedGeometry.Geometry2>
        <RectangleGeometry Rect="10,100,400,200"/>
    </CombinedGeometry.Geometry2>
</CombinedGeometry>
```

  abgeflachten Orange:

- **Xor**
  Aus der Vereinigung der beiden **Geometry**-Objekte wird die Schnittmenge entfernt.
- **Exclude**
  Geometrie 2 entfernt die Schnittmenge aus Geometrie 1, z.B. bei dieser

```
<CombinedGeometry GeometryCombineMode="Exclude">
    <CombinedGeometry.Geometry1>
        <EllipseGeometry Center="150,200" RadiusX="100" RadiusY="100" />
    </CombinedGeometry.Geometry1>
    <CombinedGeometry.Geometry2>
        <EllipseGeometry Center="250,200" RadiusX="70" RadiusY="50" />
    </CombinedGeometry.Geometry2>
</CombinedGeometry>
```

angebissenen Tomate:



18.1.5 Pinsel

In diesem Abschnitt beschränken wir uns auf das Auftragen von Farbe. Später werden noch **Brush**-Objekte vorgestellt, die Bitmap- oder Vektorgrafiken auf Oberflächen aufbringen.

18.1.5.1 SolidColorBrush

Bisher haben wir uns auf monochrome Pinsel aus der Klasse **SolidColorBrush** beschränkt, indem wir der **Fill**-Eigenschaft eines **Shape**-Objekts einen Farbnamen zugewiesen haben, z.B.:

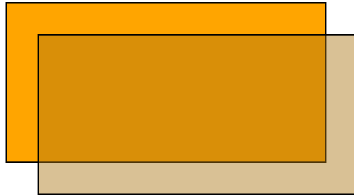
```
<Rectangle Fill="Orange" Height="100" Width="200" Margin="20,20" Stroke="Black" />
```

Das hier verwendete Objekt ist im C# - Quellcode über statische Eigenschaft **Orange** der Klasse **Brushes** ansprechbar.

Anders als die Bezeichnung *Solid Color* (dt.: *Volltonfarbe*) vermuten lässt, erlaubt die Klasse **SolidColorBrush** auch Transparenzeffekte, z.B.:

```
<Rectangle Fill="Orange" Height="100" Width="200" Margin="20,20" Stroke="Black" />
<Rectangle Fill="#72AA7412" Height="100" Width="200" Margin="40,40" Stroke="Black" />
```

Für das obere Rechteck wird eine numerische, vierkanalige Farbdefinition verwendet, wobei die erste Hexadezimalzahl für den Alphakanal steht:



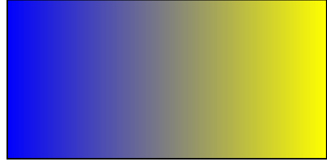
Die Grenzwerte 0 bzw. FF stehen für komplette Transparenz bzw. Deckung.

18.1.5.2 LinearGradientBrush

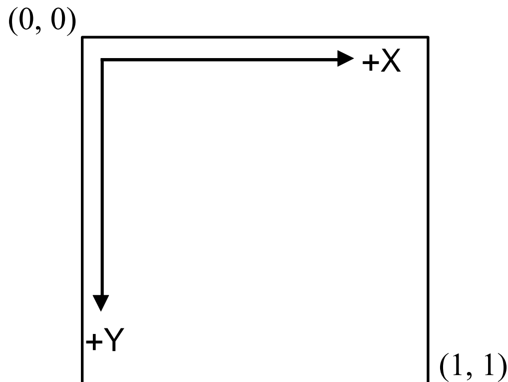
Zur Definition eines Farbverlaufs ist mehr Aufwand erforderlich als bei einem monochromen Pinsel, den das Ergebnis aber belohnt. Im folgenden XAML-Code wird per Eigenschaftselement ein **LinearGradientBrush**-Objekt mit horizontalem Farbverlauf von Blau bis Gelb deklariert und zugewiesen:

```
<Rectangle Height="100" Width="200" Margin="20" Stroke="Black" >
    <Rectangle.Fill>
        <LinearGradientBrush StartPoint="0,0" EndPoint="1,0">
            <GradientStop Color="Blue" Offset="0" />
            <GradientStop Color="Yellow" Offset="1" />
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

Das Ergebnis:



Den Verlaufsvektor definiert man über zwei Punkte im folgenden Koordinatensystem:



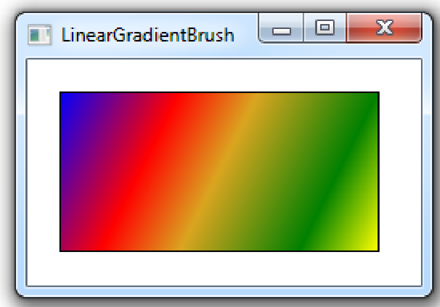
Statt die beiden Farben bzw. **GradientStop**-Objekte wie im Beispiel an den Start- und den Endpunkt des Verlaufsvektors zu setzen, kann man sie über die **GradientStop**-Eigenschaft auf die Strecke und sogar (durch Werte außerhalb des Intervalls [0, 1]) in den Extrapolationsbereich setzen, z.B.:

```
<GradientStop Color="Blue" Offset="-0.5" />
```

Statt sich auf zwei **GradientStop**-Objekte zu beschränken, kann man es noch bunter treiben und mehrere Zwischenfarben deklarieren, z.B.:

```
<Rectangle.Fill>
    <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
        <GradientStop Color="Blue" Offset="0" />
        <GradientStop Color="Red" Offset="0.3" />
        <GradientStop Color="Goldenrod" Offset="0.5" />
        <GradientStop Color="Green" Offset="0.8" />
        <GradientStop Color="Yellow" Offset="1" />
    </LinearGradientBrush>
</Rectangle.Fill>
```

Hier kommt ein diagonaler Farbverlaufsvektor zum Einsatz:

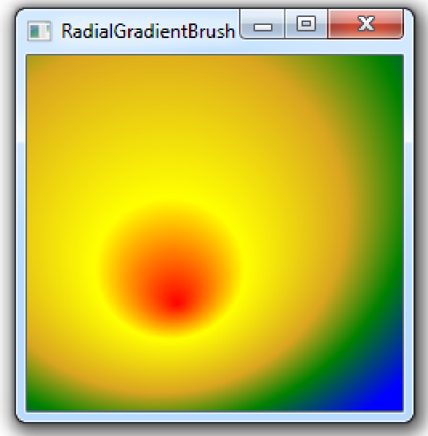


18.1.5.3 RadialGradientBrush

Während die Objekte der Klasse **LinearGradientBrush** mit einem Farbverlaufsvektor arbeiten, verläuft die Farbe bei einem **RadialGradientBrush**-Objekt von einem Startpunkt aus in alle Richtungen auf eine elliptische Grenzlinie zu. Auf den Verlaufsgradienten kann man zwei oder mehrere Farben per Offset-Eigenschaft positionieren, z.B.:

```
<Rectangle >
    <Rectangle.Fill>
        <RadialGradientBrush Center="0.3,0.2" RadiusX="1" RadiusY="1"
                             GradientOrigin="0.4,0.7">
            <GradientStop Color="Red" Offset="0" />
            <GradientStop Color="Yellow" Offset="0.2" />
            <GradientStop Color="Goldenrod" Offset="0.5" />
            <GradientStop Color="Green" Offset="0.7" />
            <GradientStop Color="Blue" Offset="1" />
        </RadialGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

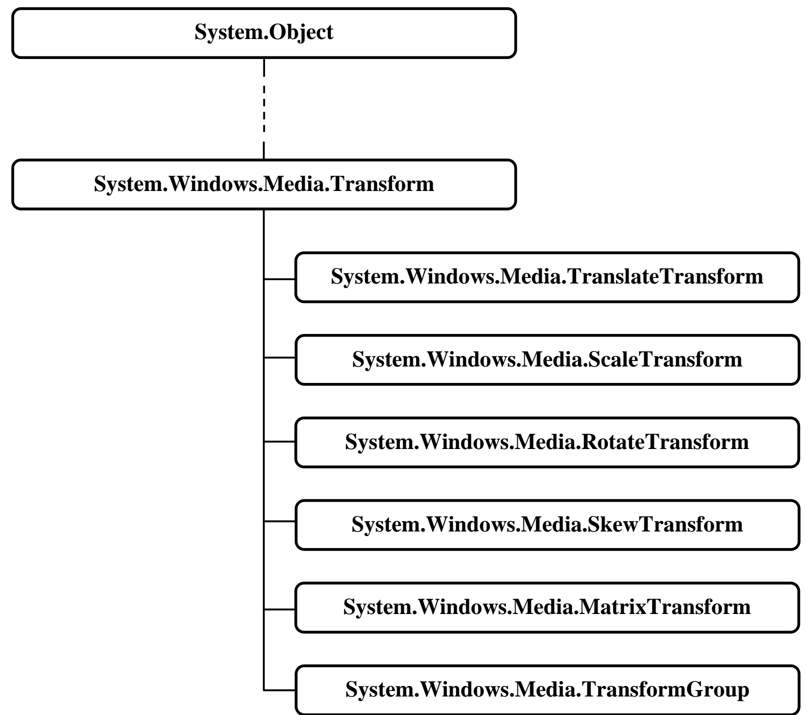
Über die **RadialGradientBrush**-Eigenschaften **Center**, **RadiusX** und **RadiusY** wird die Ellipse definiert, an deren Umrisslinie die Terminalfarbe erreicht wird. Die Eigenschaft **GradientOrigin** legt den Startpunkt für den Farbverlauf fest.



18.2 Transformationen

Auf Steuer- und Grafikelemente lassen sich Transformationen wie Verschiebungen, Größenänderungen oder Rotationen anwenden, was z.B. bei Animationen sinnvoll ist. Alle unterstützten Transformationen arbeiten *affin*, d.h. die Punkte einer Geraden liegen nach der Abbildung wieder auf einer Geraden. In der WPF sind Transformationen auf elementarer Ebene implementiert und für alle Objekte in der **UIElement** - Klassenhierarchie möglich (Petzold 2010, S. 167).

Mit der Durchführung einer Transformation beauftragt man ein Objekt aus der Hierarchie zur abstrakten Basisklasse **System.Windows.Media.Transform**:



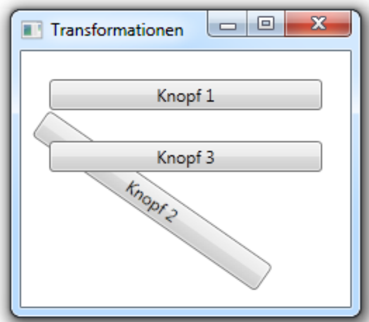
18.2.1 RenderTransform vs. LayoutTransform

Es ist zu differenzieren zwischen Transformationen *mit* bzw. *ohne* Berücksichtigung bei der Größen- und Positionsberechnung durch die WPF-Layoutlogik:

- Um eine Transformation *ohne* Berücksichtigung durch das Layoutsystem anzufordern, wird das **Transform**-Objekt der Eigenschaft **RenderTransform** zugewiesen, die bereits in der Klasse **UIElement** vorhanden ist. Im folgenden XAML-Code

```
<Window x:Class="LayoutTransformNamespace.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Transformationen" Height="220" Width="250">
    <StackPanel Margin="20">
        <Button Content="Knopf 1" />
        <Button Content="Knopf 2">
            <Button.RenderTransform>
                <RotateTransform  Angle="35" />
            </Button.RenderTransform>
        </Button>
        <Button Content="Knopf 3" />
    </StackPanel>
</Window>
```

  wird in einem vertikalen **StackPanel**-Layoutcontainer von drei **Button**-Objekten das mittlere unmittelbar vor dem Rendern (Pixelausgabe auf dem Bildschirm), insbesondere *nach* der Berechnung von Größen und Positionen durch das Layoutsystem gedreht, was in speziellen Situationen als originell empfunden werden mag:

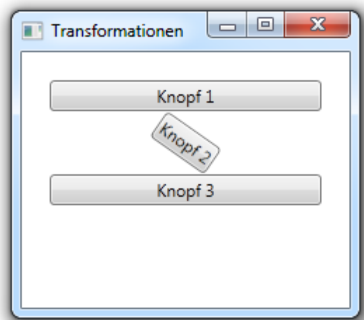


Gleich folgt ein Beispiel mit einem triftigeren Grund, eine Transformation bei der Layout-Berechnung unberücksichtigt zu lassen.

- Um eine Transformation *mit* Berücksichtigung durch das Layoutsystem anzufordern, wird das **Transform**-Objekt der Eigenschaft **LayoutTransform** zugewiesen, die ab der Klasse **FrameworkElement** vorhanden ist. Ändert man das obige Beispiel dementsprechend,

```
<Button.LayoutTransform>
    <RotateTransform  Angle="35" />
</Button.LayoutTransform>
```

kann die Drehung bei der Berechnung von Größen und Positionen durch das Layoutsystem berücksichtigt werden:



18.2.2 TranslateTransform

Ein Objekt der Klasse **TranslateTransform** sich nur auf die Position eines Grafik- oder Steuerelements aus und bleibt es als Referenzziel der Eigenschaft **LayoutTransform** wirkungslos, wenn ein intelligenter Layoutcontainer am Werk ist. Wie das folgende Beispiel nach Petzold (2010, S. 169) zeigt, kann man in dieser Situation über die **UIElement**-Eigenschaft **RenderTransform** eine Verschiebung vornehmen, die *nicht* von der WPF-Layoutlogik kassiert wird:

```
<Window x:Class="TranslateTransformNamespace.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="TranslateTransform" Height="139" Width="895" Background="Black">
    <Grid>
        <TextBlock Text="Thanks to Charles P." Foreground="White" FontSize="96"
                   HorizontalAlignment="Center" VerticalAlignment="Center" />
        <TextBlock Text="Thanks to Charles P." Foreground="Black" FontSize="96"
                   HorizontalAlignment="Center" VerticalAlignment="Center">
            <TextBlock.RenderTransform>
                <TranslateTransform X="1" Y="2" />
            </TextBlock.RenderTransform>
        </TextBlock>
    </Grid>
</Window>
```

Hier wird vor einem schwarzen Hintergrund ein Text in Weiß und dann (über die Eigenschaften **X** und **Y** des **TranslateTransform**-Objekts) leicht verschoben nochmals in Schwarz ausgegeben, um einen **Prägungs-Effekt** (engl.: *emboss*) zu erzielen:

18.2.3 ScaleTransform

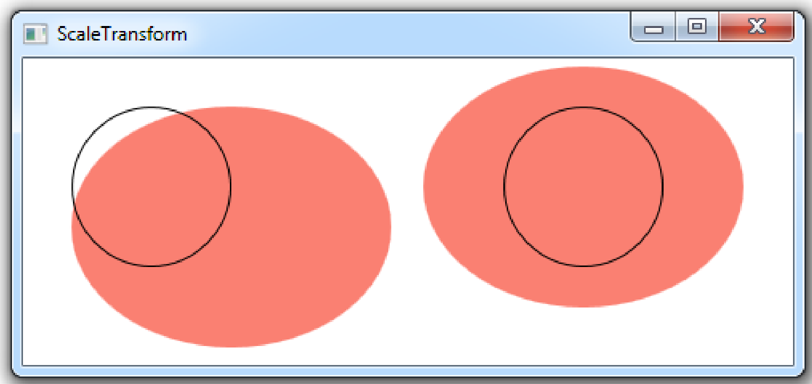
Ein Objekt der Klasse **ScaleTransform** bewerkstelligt eine Größenänderung in X- und/oder Y-Richtung, gesteuert über die als Faktoren zu verstehenden Wert der Eigenschaften **ScaleX** und **ScaleY**.

Per Voreinstellung (**CenterX**-Wert 0) ändert jeder Punkt der Figur seinen Abstand vom linken Rand der Figur um den Faktor **ScaleX**, so dass auch das Zentrum der Figur wandert. Eine analoge Aussage gilt für die Eigenschaften **CenterY** und **ScaleY**. Soll bei einer Größenzunahme das Zentrum der Figur ortsstabil bleiben, setzt man die Eigenschaften **ScaleX** und **ScaleY**. auf die relativen Koordinaten des Zentrums (innerhalb der Figur). Nun wird für jeden Punkt der Figur sein Abstand vom Zentrum in X- bzw. Y-Richtung um den Faktor **ScaleX** bzw. **ScaleY** geändert, wobei das Zentrum selbst offenbar am alten Ort verharrt.

Im folgenden Beispiel wird der linke Kreis mit dem voreingestellten Streckungszentrum (0,0) in eine 2- bzw. 1,5-fach größere Ellipse transformiert; der rechte Kreis erlebt dasselbe Größenwachstum hingegen mit seinem Mittelpunkt als Streckungszentrum:

```
<Canvas>
    <Ellipse Canvas.Left="30" Canvas.Top="30" Height="100" Width="100" Fill="Salmon">
        <Ellipse.RenderTransform>
            <ScaleTransform ScaleX="2" ScaleY="1.5" />
        </Ellipse.RenderTransform>
    </Ellipse>
    <Ellipse Canvas.Left="30" Canvas.Top="30" Height="100" Width="100" Stroke="Black">
    </Ellipse>

    <Ellipse Canvas.Left="300" Canvas.Top="30" Height="100" Width="100" Fill="Salmon">
        <Ellipse.RenderTransform>
            <ScaleTransform CenterX="50" CenterY="50" ScaleX="2" ScaleY="1.5"  />
        </Ellipse.RenderTransform>
    </Ellipse>
    <Ellipse Canvas.Left="300" Canvas.Top="30" Height="100" Width="100" Stroke="Black">
    </Ellipse>
</Canvas>
```

Zur Verdeutlichung der unterschiedlichen Transformationen sind die Ausgangsfiguren als Umrisslinie eingezeichnet:



Solange Sie mit der **UIElement**-Eigenschaft **RenderTransform** arbeiten wollen (also *nicht* mit der **FrameworkElement**-Eigenschaft **LayoutTransform**), können Sie sich die Mühe ersparen, z.B. die Zentrumskoordinaten der zu skalierenden Figur zu ermitteln: Lassen Sie die **ScaleTransform**-Eigenschaften **CenterX** und **CenterY** unverändert, und verwenden Sie stattdessen die **UIElement**-Eigenschaft **RenderTransformOrigin**, die relativ zu verstehende Koordinaten aufnimmt, so dass z.B. mit dem Paar (0.5, 0.5) das Zentrum angesprochen wird:

```
<Ellipse RenderTransformOrigin="0.5 0.5"
         Canvas.Left="300" Canvas.Top="30" Height="100" Width="100" Fill="Salmon">
```

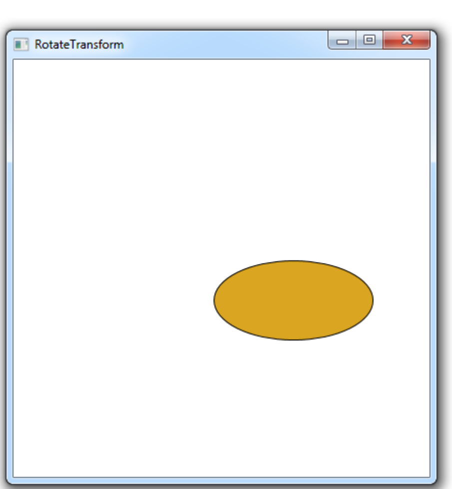
18.2.4 RotateTransform

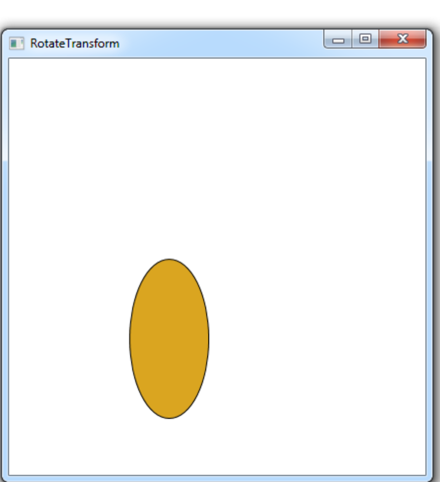
Ein Objekt der Klasse **RotateTransform** dient dazu, ein Grafik- oder ein Steuerelement zu drehen, wobei als wesentliche Eigenschaften der Drehwinkel (**Angle**) und die Koordinaten des Drehpunkts (**CenterX**, **CenterY**) zur Verfügung stehen. Hier wird eine Ellipse gedreht:

```
<Canvas>
    <Ellipse Fill="Goldenrod" Stroke="Black" Height="80" Width="160"
             Canvas.Left="200" Canvas.Top="200">
        <Ellipse.RenderTransform>
            <RotateTransform x:Name="rt" Angle="90"/>
        </Ellipse.RenderTransform>
    </Ellipse>
</Canvas>
```

Wie die Positionen für die Drehwinkel 0° und 90° zeigen,

0° 90°





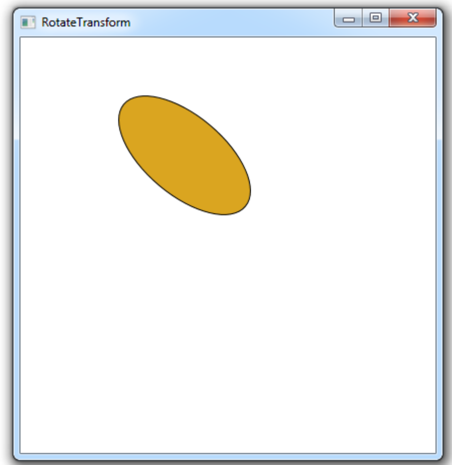
orientiert sich die WPF bei der Interpretation des Winkels am Alltagsmenschen und nicht am Mathematiker:

- statt Bogenmaß (von 0 bis 2π) ist eine Angabe von 0 bis 360 Grad erwünscht,
- es wird nicht im mathematisch positiven Sinn, sondern im Uhrzeigersinn gedreht.

Im Beispiel wurde das **RotationTransform**-Objekt benannt, um in einer **MouseDown**-Ereignisbehandlungsmethode den Drehwinkel ändern zu können:

```
private void Window_MouseDown(object sender, MouseButtonEventArgs e) {
   rt.Angle += 10;
}
```

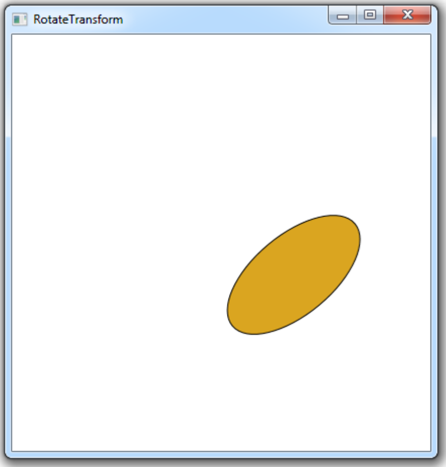
Mit jedem Mausklick erhöht sich der Rotationswinkel um 10°, z.B.:



Verlegt man den Drehpunkt von seiner Voreinstellung (0, 0), also oben links, in das Zentrum der Figur, im Beispiel (80, 40),

```
<RotateTransform x:Name="rt" Angle="0" CenterX="80" CenterY="40"/>
```

dann dreht sich die Figur um ihr Zentrum, ohne zu wandern:



Statt dem **RotateTransform**-Objekt Drehfigur-bezogene absolute Koordinaten anzugeben, kann man wie bei der Skalierungstransformation (vgl. Abschnitt 18.2.3) die Eigenschaft **RenderTransformOrigin** der Figur über relative Koordinaten zentrieren, z.B.:

```
<Ellipse Fill="Goldenrod" Stroke="Black" Height="80" Width="160"
         Canvas.Left="200" Canvas.Top="200" RenderTransformOrigin="0.5 0.5">
    <Ellipse.RenderTransform>
        <RotateTransform x:Name="rt" Angle="0"/>
    </Ellipse.RenderTransform>
</Ellipse>
```

Bei einer „nonorthononalen“ Drehung von Steuerelementen (Drehwinkel nicht restfrei durch 90 teilbar) unterlässt das WPF-Layoutsystem eine Streckung auf die maximale Höhe bzw. Breite, weil sich dabei der Winkel ändern würde (siehe Beispiel in Abschnitt 18.2.1).

18.3 Rastergrafiken

Es werden folgende Rastergrafikdateiformate unterstützt:

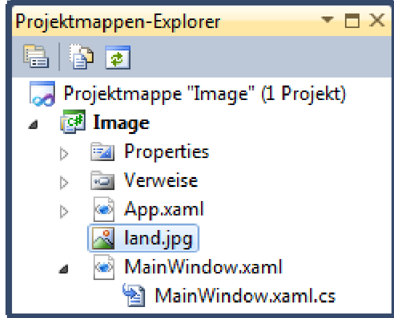
- BMP (Bitmap)
- GIF (Graphics Interchange Format)
- ICO (Windows Symbol-Dateien)
- JPEG (Joint Photographic Experts Group)
- JXR, HDP, WDP (JPEG XR, HD Photo, Windows Media Photo)
- PNG (Portable Network Graphics)
- TIFF (Tag Image File Format)

18.3.1 Die Klasse Image

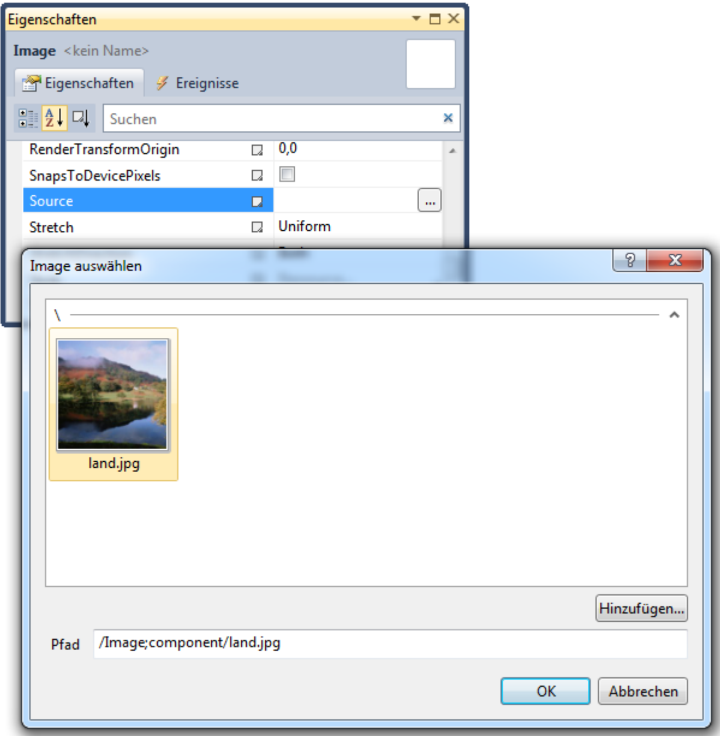
Von den diversen WPF-Optionen zur Nutzung von Rastergrafiken ist die von **FrameworkElement** abgeleitete Klasse **Image** am einfachsten zu verwenden, deren Objekte mit Rechteckform in den Baum der visuellen WPF-Elemente eingefügt werden.

Liegt die Bildquelle als Datei, sollte diese Datei zunächst in das Visual Studio - Projekt aufgenommen werden:

- Kopieren Sie die Bilddatei in den Projektordner.
- Nehmen Sie die Bilddatei in das Projekt auf (Item **Hinzufügen > Vorhandenes Element > Bilddateien** aus dem Kontextmenü zum Projekt). Anschließend wird die Datei im Projektmappen-Explorer angezeigt, z.B.:



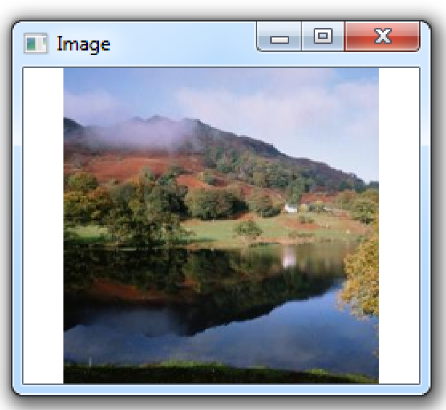
Diese Datei kann nun bequem der **Source**-Eigenschaft (vom Typ.: **ImageSource** aus dem Namensraum **System.Windows.Media**) eines per XAML-Code aufgenommenen **Image**-Objekts per Eigenschaftsfenster zugewiesen werden:

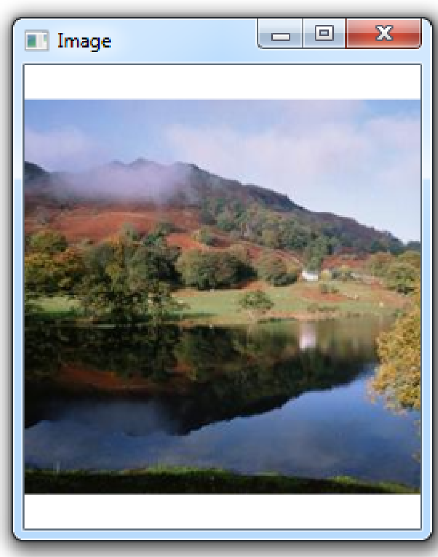


Es resultiert der XAML-Code:

```
<Window x:Class="ImageNamespace.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Image" Height="300" Width="350">
    <Grid>
        <Image Source="/Image;component/land.jpg" />
    </Grid>
</Window>
```

Wie die Klasse **Shape** besitzt auch die Klasse **Image** eine **Stretch**-Eigenschaft (siehe Abschnitt 18.1.3), wobei der Wert **Uniform** voreingestellt ist. Ist die WPF-Layoutlogik am Werk (z.B. im **Grid**-Layoutcontainer), dann nimmt ein **Image**-Objekt die bei unverändertem Seitenverhältnis größtmögliche Fläche ein, z.B.:





Mit dem alternativen **Stretch**-Wert **Fill** erfolgt die Größenanpassung ohne Rücksicht auf das Seitenverhältnis, z.B.:

Während beim XAML-Code dank automatischer Typkonvertierung die Angabe eine Datei als Wert der **Source**-Eigenschaft zum Instantiieren eines **Image**-Objekts genügt, muss im C# - Quellcode explizit ein **ImageSource**-Objekt erstellt werden. Für diese abstrakte Basisklasse ist u.a. die konkrete Ableitung **BitmapSource** verfügbar, z.B.:

```
private void Window_MouseDown(object sender, MouseButtonEventArgs e) {
   bild.Source = new BitmapImage(new Uri(
      "http://www.uni-trier.de/fileadmin/templates/homepagebilder/bild_30.png"));
}
```

18.3.2 Rastergrafiken zum Tapezieren

Wir erweitern unser Pinselarsenal aus Abschnitt 18.1.5 um ein Werkzeug, das eine Bitmapquelle zum Tapezieren von beliebigen, nicht unbedingt rechteckigen Figuren verwendet. Im folgenden XAML-Code dient ein **ImageBrush**-Objekt dazu, eine Ellipse bildfüllend und verzerrungsfrei (**Stretch**-Wert **UniformToFill**, vgl. Abschnitt 18.1.3) mit einem Photo zu verzieren:

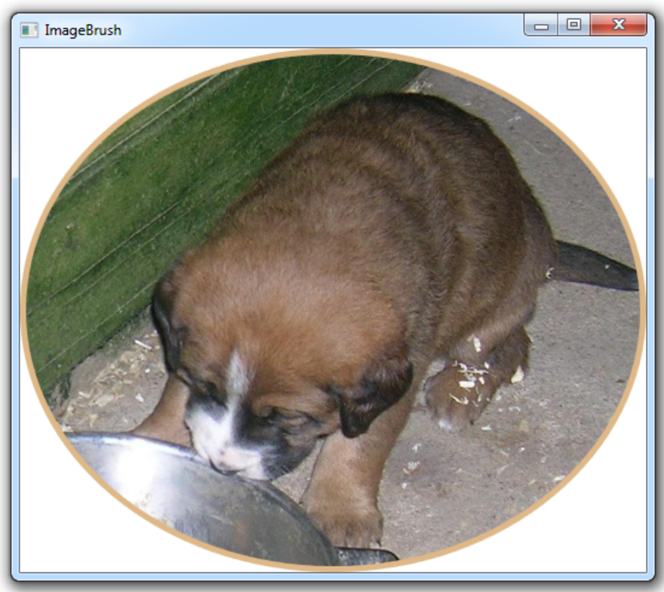
```
<Grid>
    <Ellipse Stroke="BurlyWood" StrokeThickness="5">
        <Ellipse.Fill>
            <ImageBrush ImageSource="/ImageBrush;component/EmmaBodo.jpg"
                        Stretch="UniformToFill" />
        </Ellipse.Fill>
    </Ellipse>
</Grid>
```

Bei der voreingestellten zentrierten Platzierung des Photos wird der nette braune Hund an den Rand gedrängt:

Um Abhilfe zu schaffen, kann man per **Viewbox**-Eigenschaft einen geeigneten Ausschnitt wählen, wobei per Voreinstellung relative Koordinaten zu verwenden sind, z.B.:

```
<ImageBrush ImageSource="/ImageBrush;component/EmmaBodo.jpg" Stretch="UniformToFill"
            Viewbox="0.55,0.0, 0.48,0.64" />
```

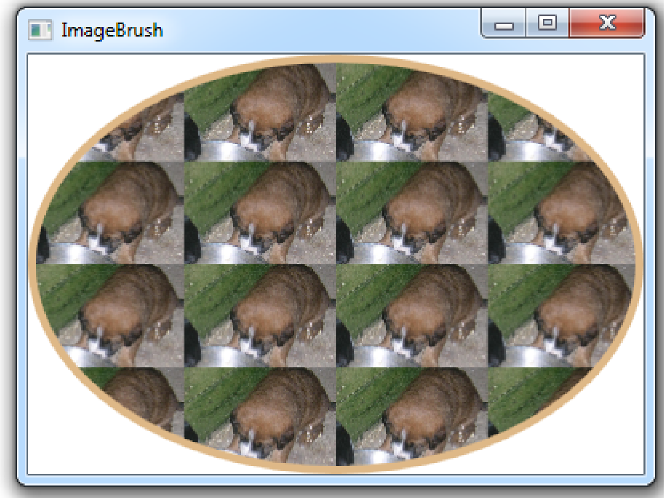
Nun ist der rechte Hund besser zu sehen:



Statt ein Bild oder einen Bildausschnitt auf die „komplette Tapetenrolle“ zu verteilen, kann man per **Viewport**-Eigenschaft einen Ausgabebereich festlegen, wobei per Voreinstellung relative Koordinaten zu verwenden sind, z.B.:

```
<ImageBrush ImageSource="/ImageBrush;component/EmmaBodo.jpg" Stretch="UniformToFill"
            Viewbox="0.55,0.0, 0.4,0.64"
            Viewport="0.0,0.0, 0.25,0.25" TileMode="Tile" />
```

In der Regel wird man die restliche Tapetenfläche nicht leer (transparent) lassen, sondern durch Bildwiederholungen füllen. Dazu wählt man als **TileMode** an Stelle der Voreinstellung **None** z.B. die Alternative **Tile**, im Beispiel mit folgendem Ergebnis:



19 Text darstellen und drucken

Wir behandeln zunächst die Klassen für eine flexible Bildschirmausgabe von Fließtexten und beschäftigen uns dann mit der Druckausgabe mit Hilfe des XPS-Seitenformats (*XML Paper Specification*).

19.1 Die Klasse TextBlock für kurze Fließtexte

Für kurze Textausgaben im Rahmen einer Bedienoberfläche bietet sich die einfach verwendbare Klasse **TextBlock** an, die wie schon mehr eingesetzt haben. Man kann der **Text**-Eigenschaft ein **String**-Objekt zuweisen und hat dabei keine Möglichkeit, einzelne Textpassagen unterschiedlich zu formatieren.

Eine erheblich flexiblere Alternative besteht darin, mit einem **InlineCollection**-Objekt zu arbeiten, das der **TextBlock**-Eigenschaft **Inlines** zugewiesen wird und beliebig viele **Inline**-Elemente aufnehmen darf. Die von **TextElement** abstammende abstrakte Klasse **Inline** besitzt diverse Konkretisierungen zur Darstellung und Formatierung von Fließtext, z.B.:

- **Run**
  Ein **Run**-Objekt enthält ein (über die **Text**-Eigenschaft ansprechbares) **String**-Objekt als Fließtextbestandteil. Hier
  ```
  <TextBlock.Inlines>
      Text kann z.B. <Bold>fett</Bold> oder <Italic>kursiv</Italic> gesetzt werden.
  </TextBlock.Inlines>
  ```
  folgen aufeinander:
  - ein **Run**-Objekt („`Text kann z.B. `“)
  - ein **Bold**-Objekt (siehe unten)
  - ein **Run**-Objekt („` oder `“)
  - ein **Italic**-Objekt (siehe unten)
  - ein **Run**-Objekt („` gesetzt werden.`“).
- **Bold**
  Ein **Bold**-Objekt beinhaltet ein (über die **Inlines**-Eigenschaft ansprechbares) **InlineCollection**-Objekt und stellt den enthaltenen Text **fett** dar, z.B.:
  ```
  <Bold>fett</Bold>
  ```
- **Italic**
  Ein **Italic**-Objekt beinhaltet ein (über die **Inlines**-Eigenschaft ansprechbares) **InlineCollection**-Objekt und stellt den enthaltenen Text *kursiv* dar, z.B.:
  ```
  <Italic>kursiv</Italic>
  ```
- **LineBreak**
  Mit einem **LineBreak**-Element erzwingt man einen Zeilenumbruch, z.B.:
  ```
  <LineBreak/>
  ```

Weitere **TextBlock**-Eigenschaften

- **TextWrapping**
  Mit dieser Eigenschaft vom Enumerationstyp **TextWrapping** wird der Zeilenumbruch im **TextBlock** geregelt. Mögliche Werte:
  - **NoWrap**
    Kein Zeilenumbruch
  - **Wrap**
    Bei Überschreitung der Blockbreite findet ein Zeilenbruch statt, zur Not auch innerhalb eines Wortes.
  - **WrapWithOverflow**
    Bei Überschreitung der Blockbreite findet nach Möglichkeit ein Zeilenbruch statt, aber nicht innerhalb eines Wortes.

- **TextAlignment**
  Mit dieser Eigenschaft vom Enumerationstyp **TextAlignment** wird die Textausrichtung im **TextBlock** geregelt. Mögliche Werte: **Left**, **Right**, **Center**, **Justify** (Blocksatz).

Ein **TextBlock** kann durchaus mehrere Zeilen umfassen, doch für längere Texte sollte man ein **FlowDocument**-Objekt verwenden.

19.2 Schrifteigenschaften

Die Klasse **TextBlock**, aber auch die Klassen **Run**, **Bold** oder **Italic** etc. für die **Inline**-Elemente, welche die oberste Textebene in einem **TextBlock** bilden, kennen u.a. die folgenden Schrifteigenschaften:

- **FontFamily**
  Um ein Objekt der Klasse **FontFamily** zu erstellen, kann man u.a. einen Familiennamen angeben, z.B.:
  ```
  <TextBlock FontFamily="Arial" . . .>
  <Bold FontFamily="Consolas">fett</Bold>
  ```
- **FontSize**
  Es ist ein **double**-Wert anzugeben, wobei geräteunabhängige Pixel (1/96 Zoll) als Maßeinheit dienen, z.B.:
  ```
  <TextBlock FontFamily="Arial" FontSize="16" . . .>
  ```
  Im XAML-Code kann die Maßeinheit explizit angegeben werden, wobei u.a. folgende Alternativen erlaubt sind:
  - `px` (1/96 Zoll)
  - `pt` (1/72 Zoll = 96/72 `px`)
    Der traditionellen Größenangabe 12pt entspricht also in der bevorzugten WPF-Maßeinheit die Größe 16px.
- **FontStyle**
  Mögliche Werte:
  - **Normal**
  - **Italic**
    Spricht den kursiven Schriftschnitt an, der in einer Schriftartenfamilie eventuell vorhanden ist
  - **Oblique**
    Die Zeichen des normalen Schriftschnitts werden schräg gestellt.
- **FontWeight**
  Über statische Eigenschaften der Klasse **FontWeights** werden verschiedene Schriftbreiten angefordert: **Thin**, **ExtraLight**, **UltraLight**, **Light**, **Normal**, **Regular**, **Medium**, **DemiBold**, **SemiBold**, **Bold**, **ExtraBold**, **UltraBold**, **Black**, **Heavy**, **ExtraBlack**, **UltraBlack**. Den Eigenschaften entsprechen teilweise identische Breiten (z.B. bei **Normal** und **Regular**).
- **Foreground**
  Wie z.B. schon in Abschnitt 18.2.2 zu sehen war, lässt sich die Schriftfarbe über die Eigenschaft **Foreground** einstellen, z.B.:
  ```
  Background="Black"
  ```

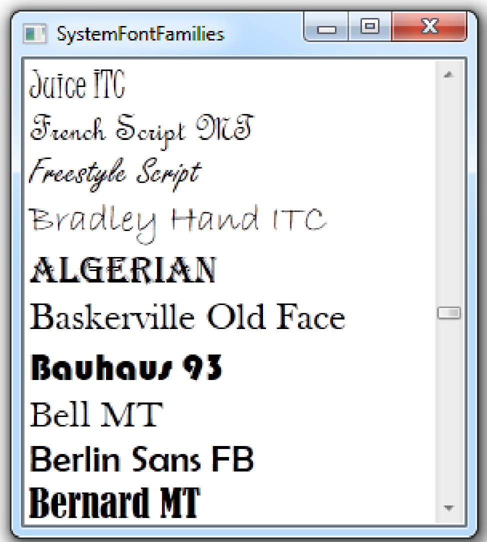
Über die Eigenschaft **SystemFontFamilies** der Klasse **Fonts** erhält man eine Liste mit den auf einem Rechner vorhandenen Schriftartfamilien. Um diese Schriftarten inspizieren zu können, erstellen wir eine simple Anwendung mit der XAML-Fensterdeklaration

```
<Window x:Class="SystemFontfamiliesNamespace.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="SystemFontFamilies" Height="191" Width="300" Loaded="Window_Loaded">
    <Grid>
        <ListBox Name="liste"/>
    </Grid>
</Window>
```

und der folgenden Behandlungsmethode zum Fensterereignis **Loaded**

```
private void Window_Loaded(object sender, RoutedEventArgs e) {
   foreach (FontFamily ff in Fonts.SystemFontFamilies) {
      TextBlock tb = new TextBlock();
      tb.Text = ff.ToString();
      tb.FontFamily = ff;
      tb.FontSize = 24;
      liste.Items.Add(tb);
   }
}
```

Wir sehen jeweils den normalen Schriftschnitt in der Größe 24px:



19.3 Die Klasse FlowDocument für Fließtext mit Block-Elementen

Während ein **TextBlock**-Objekt nur **Inline**-Elemente (wie **Run**, **Bold** und **Italic**, vgl. Abschnitt 19.1) aufnehmen kann, unterstützt die Klasse **FlowDocument** komplexe Texte auch Blockelemente wie z.B. Absätze, Abschnitte, Listen und Tabellen. Zur Bearbeitung von **FlowDocument**-Texten bietet die WPF mit der Komponente **RichTextBox** eine noch ausbaufähige Lösung, der z.B. eine Suchfunktion fehlt (vgl. Abschnitt 16). Demgegenüber stehen zur *Anzeige* von **FlowDocument**-Objekten gut ausgestattete Komponenten (inkl. Suche) zur Verfügung, von denen die Klasse **FlowDocumentReader** den größten Komfort bietet. Während ein **TextBlock** seinen Inhalt verwaltet *und* anzeigt, beschränkt sich ein **FlowDocument**-Objekt auf die Textverwaltung, so dass man zur Anzeige eine Komponente wie den **FlowDocumentReader** benötigt.

Ein **FlowDocument** kann **Block**-Objekte aus den folgenden Klassen aufnehmen, die von der gemeinsamen Basisklasse **Block** im Namensraum **System.Windows.Documents** abstammen:

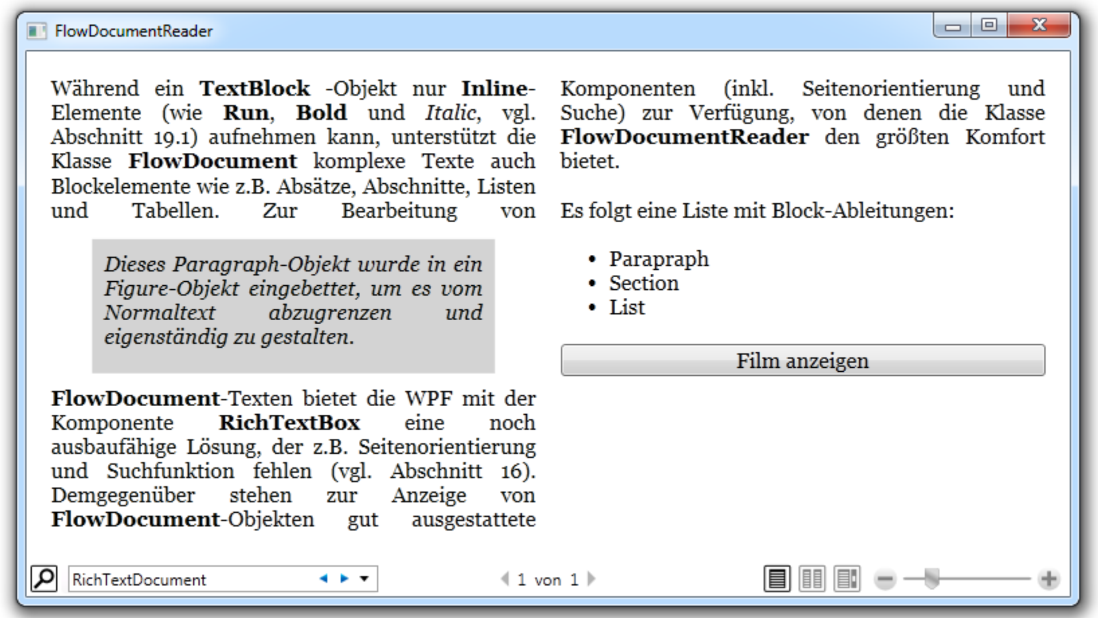
- **Paragraph**
  Ein **Paragraph** nimmt Fließtext mit **Inline**-Elementen auf (vgl. Abschnitt 19.1).
- **Section**
  In einem Abschnitt fasst man mehrere, gemeinsam zu formatierende Blöcke zusammen.

- **List**
  Diese Klasse unterstützt Auflistungen, z.B.:
  ```
  <List>
      <ListItem> <Paragraph>Parapraph</Paragraph> </ListItem>
      <ListItem> <Paragraph>Section</Paragraph> </ListItem>
      <ListItem> <Paragraph>List</Paragraph> </ListItem>
  </List>
  ```
- **Table**
  Diese Klasse unterstützt Tabellen.
- **BlockUIContainer**
  Ein Block von diesem Typ kann Steuerelemente aufnehmen, z.B.:
  ```
  <BlockUIContainer>
      <Button Content="Film anzeigen"/>
  </BlockUIContainer>
  ```

Hier wird ein **FlowDocument** der Inhaltseigenschaft **Document** einer **FlowDocumentReader**-Komponente zugewiesen:

```
<Grid>
    <FlowDocumentReader>
        <FlowDocument>
            <Paragraph>
              Während ein <Bold>TextBlock</Bold> -Objekt nur <Bold>Inline</Bold>-Elemente
                . . .
            </Paragraph>
            <Paragraph>Es folgt eine Liste mit Block-Ableitungen:</Paragraph>
            <List>
                <ListItem> <Paragraph>Parapraph</Paragraph> </ListItem>
                <ListItem> <Paragraph>Section</Paragraph> </ListItem>
                <ListItem> <Paragraph>List</Paragraph> </ListItem>
            </List>
            <BlockUIContainer>
                <Button Content="Film anzeigen"/>
            </BlockUIContainer>
        </FlowDocument>
    </FlowDocumentReader>
</Grid>
```

Ein **FlowDocumentReader** besitzt etliche Bedienelemente, z.B. ein Suchfeld (unten links):



19.4 XPS-basierte Druckausgabe

Über das XPS-Dokumentenformat (*XML Paper Specification*) steht die volle WPF-Darstellungsflexibilität und -qualität auch bei der Druckausgabe zur Verfügung (Sells & Griffiths 2007, Kap. 15). XPS ist ...

- ein Dateiformat für druckfertige Dokumente
  Eine XPS-Datei ist ein **ZIP-Archiv**, das XML-formatierten Text sowie Binärbestandteile wie Abbildungen und eingebettete Schriftarten (OpenType oder TrueType) enthält. Zur Organisation der XPS-Inhalte werden (wie bei den DOCX-Dateien von MS Word ab Version 2007) die **Open Packaging Conventions** (OPC) aus den von Microsoft vorgeschlagenen und von der ECMA[1] akzeptierten **Office Open XML** - Standards für Office-Dokumente verwendet.
- ein Dateiformat für die Zwischenspeicherung von Druckausgaben unter Windows
- eine XML-basierte Seitenbeschreibungssprache

Dank XPS-Technik ist es möglich, auf dem Weg vom Dokument zum Drucker den Wechsel zwischen verschiedenen Formaten zu vermeiden, z.B.

- RTF als Dokumentenformat
- EMF für Druckausgabedateien (engl.: *spool files*)
  Das in Windows über viele Jahre verwendete EMF-Format für Druckausgabedateien unterstützt nicht alle WPF-Darstellungsoptionen (Sells & Griffiths 2007, S. 522).
- PCL als Seitenbeschreibungssprache für den Drucker

Die WPF-Druckausgabe basiert auf der XPS-Technologie, wobei XPS-Inhalte für traditionelle Druckertreiber automatisch in das EMF-Druckausgabeformat übersetzt werden.[2] Wie aufgrund der obigen Ausführungen zu erwarten, kommt bei der Druckausgabe dasselbe API zum Einsatz wie bei der Erstellung von XPS-Dokumenten. Erfreulicherweise ist die Druckausgabe in der WPF einfacher zu realisieren als in früheren Windows-APIs (z.B. WinForms).

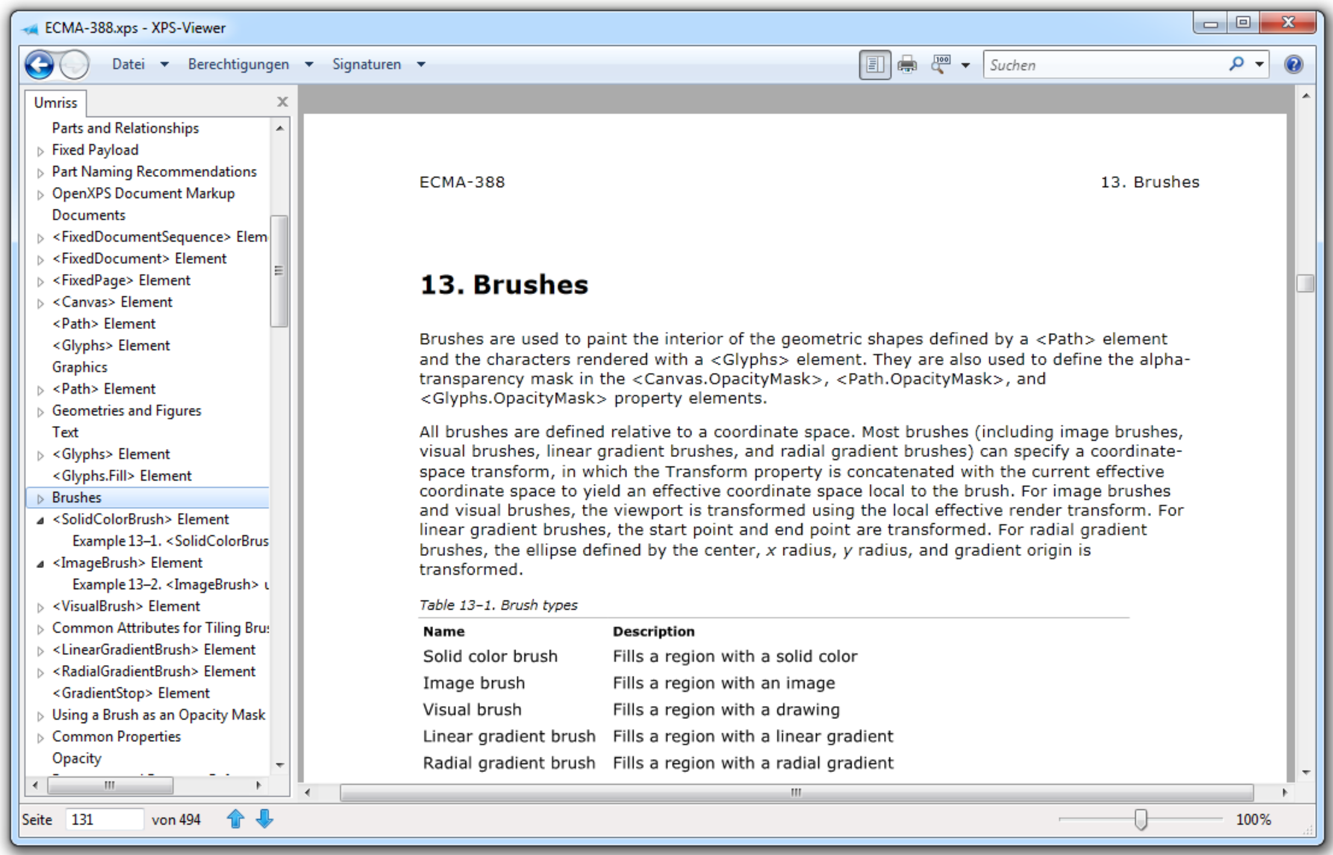
Das von Microsoft mit Windows Vista bzw. NET 3.0 eingeführte XPS-Format wurde 2009 von der ECMA unter dem Namen **OpenXPS** standardisiert.[3] In der Dokumentation (ECMA 2009) treffen Sie auf viele Begriffe aus der WPF-Technik (z.B. **Canvas**, **Brush**, **Graph**). Außerdem ist die frei verfügbare OpenXPS-Dokumentation ein sehr gutes Beispiel für eine XPS-Datei. Auf jedem Rechner mit einer .NET - Installation ab Version 3.0 ist mit dem **XPS-Viewer** ein zur Anzeige von XPS-Dokumenten geeignetes Programm vorhanden:

---

[1] ECMA stand früher für *European Computer Manufacturers Association*. Mittlerweile taucht diese Bedeutung auf der Homepage http://www.ecma-international.org/ allerdings *nicht* mehr auf.

[2] Webseite: http://msdn.microsoft.com/en-us/library/ms742418.aspx

[3] Webseite: http://www.ecma-international.org/publications/standards/Ecma-388.htm



19.4.1 Aufbau eines XPS-Dokuments und zugehörige WPF-Klassen

Bei der Präsentation von Fließtexten am Bildschirm stand die Klasse **FlowDocument** aus dem Namensraum **System.Windows.Documents** im Vordergrund. Aus demselben Namensraum stammt die Klasse **FixedDocument**, die bei der Druckausgabe eine zentrale Rolle spielt. Weile eine XPS-Datei mehrere Dokumente enthalten kann, steht die Klasse **FixedDocumentSequence** zur Verfügung. Die einzelnen Seiten eines Dokuments werden durch Objekte der Klasse **FixedPage** repräsentiert.

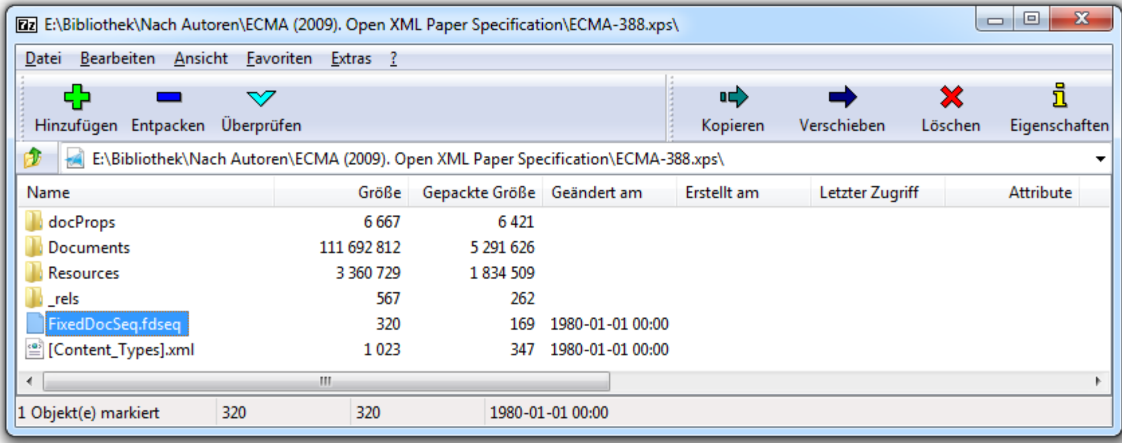
Alle Klassennamen starten mit dem Hinweis auf den fixierten Zustand einer XPS-Datei, die nicht mehr verändert, sondern formatgetreu ausgedruckt werden (analog zur PDF-Datei).

In der oben angesprochenen OpenXPS - Dokumentation (ECMA 2009) finden sich direkte Entsprechungen zu den drei **Fixed**-Klassen:

- Der WPF-Klasse **FixedDocumentSequence** entspricht die Datei **FixedDocSeq.fdseq** auf der Wurzelebene des Archivs. Es zeigt sich, dass im Beispiel nur *ein* Dokument vorhanden ist:

  ```
  <FixedDocumentSequence xmlns="http://schemas.microsoft.com/xps/2005/06">
      <DocumentReference Source="/Documents/1/FixedDoc.fdoc"/>
  </FixedDocumentSequence>
  ```

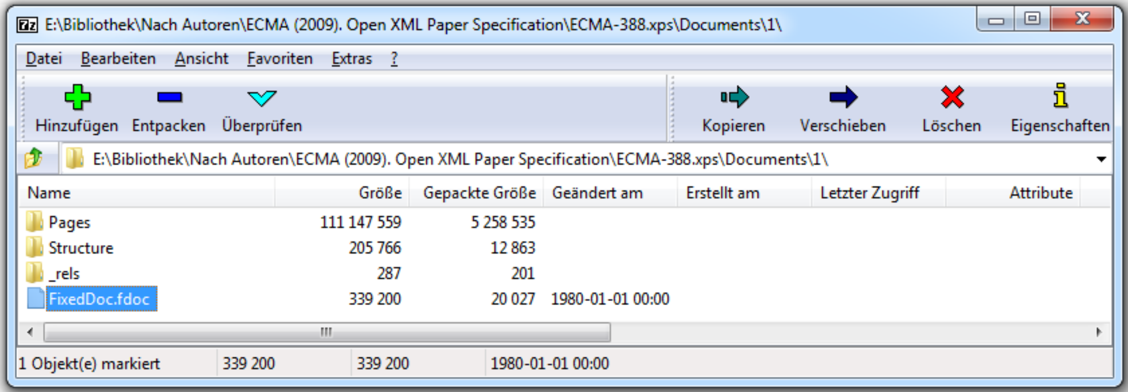
  Dies ist vermutlich der am häufigsten anzutreffende Fall. Der Dateiname **FixedDocSeq.fdseq** ist übrigens *nicht* universell gültig. In welcher Datei eines XPS-Archivs die Dokumentensequenz deklariert ist, erfährt man über die stets vorhandene Datei **_rels\.rels** (Sells & Griffiths 2007, S. 525). Hier ist das mit dem Freeware-Packer **7-Zip** geöffnete Archiv **ECMA-388.xps** zu sehen:



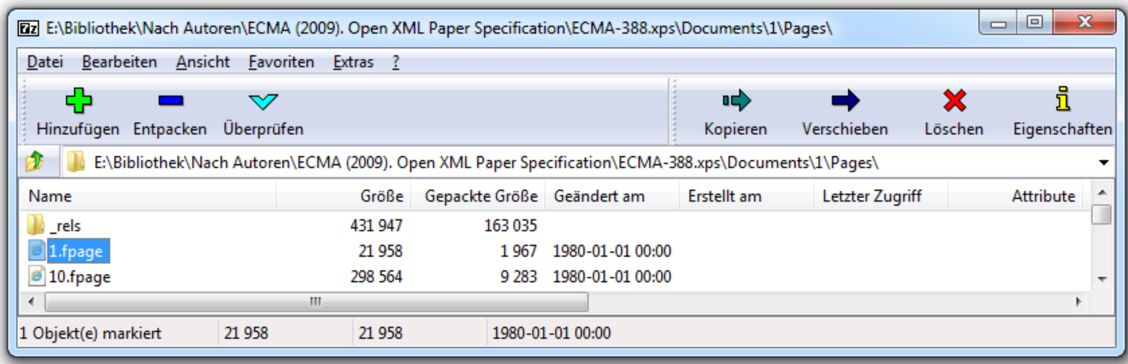
- Im Beispiel befinden sich die Daten zum einzigen XPS-Dokument im Ordner **/Documents/1/**, wobei man die Datei **FixedDoc.fdoc**

```
<FixedDocument xmlns="http://schemas.microsoft.com/xps/2005/06">
    <PageContent Source="Pages/1.fpage">
        <PageContent.LinkTargets>
            <LinkTarget Name="PG_1_LNK_2278"/>
        </PageContent.LinkTargets>
    </PageContent>
    <PageContent Source="Pages/2.fpage">
    </PageContent>
    . . .
</FixedDocument>
```

  mit der WPF-Klasse **FixedDocument** assoziieren kann. Hier ist der Ordner zum Dokument zu sehen:



- Im Archivordner **/Documents/1/Pages** enthält das Beispiel für jede Seite des Dokuments eine Datei mit dem Namen ***n*.fpage** (*n* steht für die Seitennummer), die von einem Objekt der WPF-Klasse **FixedPage** repräsentiert werden kann. Hier ist die XML-Datei mit den Elementen der ersten Seite zu finden:



Ein Blick in die Datei zur ersten Seite der Datei **ECMA-388.xps** zeigt eine XML-Deklaration, die als Bestandteile neben altbekannten WPF-Elementen (**Canvas**, **Path**) das Element (bzw. die Klasse) **Glyph** aufführt:

```
<FixedPage xmlns="http://schemas.microsoft.com/xps/2005/06" Width="816"
    Height="1056" xml:lang="en-US">
    <Canvas>
        <Canvas.RenderTransform>
            <MatrixTransform Matrix="1.333333333,0,0,1.333333333,0,0"/>
        </Canvas.RenderTransform>
        . . .
        <Glyphs BidiLevel="0" Fill="#FF000000"
                FontUri="/Resources/43AA7BBA-4360-585B-084D-5444E14D9954.odttf"
                FontRenderingEmSize="48" StyleSimulations="None" OriginX="194.3"
                OriginY="168.26" UnicodeString="Open XML "
                Indices=",87;,72;,68.5;,73.25;,36.25;,78.5;,96.75;,65.75;">
        </Glyphs>
        . . .
        <Path Name="PG_1_LNK_2278" Data="M 191.96,72 L 193.88,72 " Stroke="#00000000"
              StrokeThickness="1">
        </Path>
    </Canvas>
</FixedPage>
```

19.4.2 FixedDocument und FixedPage

Für eine mehrseitige Druckausgabe ist ein Objekt der Klasse **FixedDocument** zu erstellen:

```
FixedDocument fdoc = new FixedDocument();
```

Für jede Seite benötigen wir ein Objekt der Klasse **FixedPage**, z.B.:

```
FixedPage fpage = new FixedPage();
```

Allerdings kann ein **FixedPage**-Objekt nicht direkt in die **FixedDocument**-Seitenkollektion aufgenommen werden, sondern muss dazu in ein **PageContent**-Objekt verpackt werden:

```
PageContent fpageContent = new PageContent();
```

Zum Eintüten des **FixedPage**-Objekts ist die Methode **IAddChild()** zu verwenden, die von der Klasse **PageContent** über eine explizite Schnittstellenimplementierung realisiert ist (vgl. Abschnitt 8.4). Daher ist beim Aufruf der Schnittstellenname voranzustellen, z.B.:

```
((System.Windows.Markup.IAddChild)fpageContent).AddChild(fpage);
```

Zur Aufnahme des **PageContent**-Objekts in die Seitenkollektion des **FixedDocument**-Objekts dient der erwartete **Add()**-Aufruf:

```
fdoc.Pages.Add(fpageContent);
```

Nun machen wir uns daran, die erste Seite mit Inhalt zu füllen. Gemäß XPS-Spezifikation sind dabei nur drei Typen erlaubt:

- **Canvas**
  Dieser Layoutcontainer (vgl. 9.7.6) dient zur Gruppierung von Elementen, die einer gemeinsamen Formatierung (z.B. Hintergrundfarbe) oder Transformation unterzogen werden sollen.
- **Glyph**
  Diese Klasse repräsentiert fest formatierten und positionierten Text.
- **Path**
  Mit dieser Klasse lassen sich beliebige zweidimensionale Figuren darstellen (siehe Abschnitt 18.1.4.4).

Glücklicherweise kann die Klasse **FixedPage** die elementaren und unhandlichen **Glyph**-Objekte durch automatische Konvertierung von bequemen Objekten (z.B. aus der Klasse **TextBlock**) erstellen. Hier wird ein **Canvas**-Objekt erzeugt, das eine Hintergrundfarbe erhält und ein **TextBlock**-Objekt aufnimmt:

```
Canvas canPage = new Canvas();
canPage.Background = Brushes.Bisque;
TextBlock textBlock = new TextBlock();
canPage.Children.Add(textBlock);
```

Das **TextBlock**-Objekt erhält seinen Inhalt von einem **TextBox**-Steuerelement und Schriftmerkmale über seine Eigenschaften **FontFamily** und **FontSize**:

```
textBlock.Text = textBox.Text;
textBlock.FontFamily = new FontFamily("Arial");
textBlock.FontSize = 32;
```

Analog lassen sich weitere **FixedPage**-Objekte erstellen, befüllen und in das **FixedDocument** aufnehmen.

Per Voreinstellung misst ein **FixedPage**-Objekt 816 × 1056 geräteunabhängige Pixel (jeweils 1/96 Zoll), was gerade 8,5 × 11 Zoll (engl. *inch*) ergibt. Diese Abmessungen landen in den **FixedPage**-Eigenschaften **Width** und **Height**.

Daneben kann ein **FixedPage**-Objekt in seiner Eigenschaft **ContentBox** (vom Typ **Rect** mit den Eigenschaften **X** und **Y**) den tatsächlich bedruckbaren Bereich bereithalten zur aufmerksamen Beachtung durch die Programmierer. Gleich ist zu sehen, wie man für den zu verwendenden Drucker die Papiergröße sowie die Position und die Größe des Druckbereichs in Erfahrung bringt.

19.4.3 XpsDocumentWriter

Für die Ausgabe von XPS-Inhalten (durch einen Drucker oder in eine XPS-Datei) ist die Klasse **XpsDocumentWriter** zuständig. Um ein Objekt zu erhalten, das einen vom Benutzer gewählten Drucker beliefern wird, rufen wir auf Empfehlung von Sells & Griffiths (2007, S. 534) die statische Methode **CreateXpsDocumentWriter**() der Klasse **PrintQueue** auf:

```
PrintDocumentImageableArea imageableArea = null;
XpsDocumentWriter fdocWriter = PrintQueue.CreateXpsDocumentWriter(ref imageableArea);
```

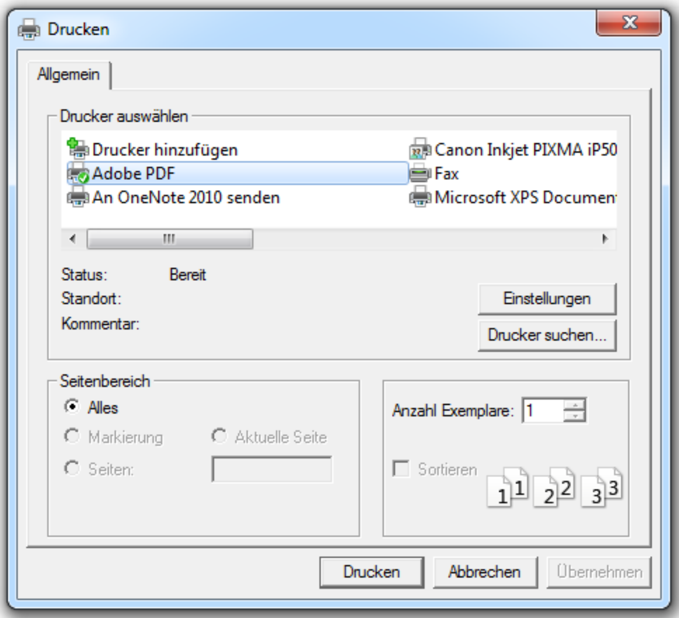
Über den Referenzparameter erhalten wir ein **PrintDocumentImageableArea**-Objekt, das die Papiergröße sowie die Größe und die Position des Druckbereichs kennt.

Voraussetzungen für das beschriebene Verfahren:

- Das Projekt benötigt einen Verweis auf das Assembly **System.Printing.dll**.
- Der Einfachheit halber sollten die Namensräume **System.Printing** und **System.Windows.Xps** importiert werden:

  ```
  using System.Printing;
  using System.Windows.Xps;
  ```

Der Benutzer kann einen Drucker per Standarddialog wählen:



Entscheidet er sich für das **Abbrechen**, liefert **CreateXpsDocumentWriter()** den Wert **null** zurück, und wir beenden die zum Drucken aufgerufene Methode:

```
if (fdocWriter == null)
  return;
```

Ein **XpsDocumentWriter** kann über seine **Write()**-Überladungen Objekte der Klassen **FixedPage**, **FixedDocument** oder **FixedDocumentSequence** ausgeben. Wir erstellen ein **FixedPage**-Objekt und verwenden für seine Eigenschaften **Width** und **Height** die Papierabmessungen des Druckers:

```
FixedPage fpage = new FixedPage();
fpage.Width = imageableArea.MediaSizeWidth;
fpage.Height = imageableArea.MediaSizeHeight;
```

Zunächst erstellen wir ein **Canvas**-Objekt mit Hintergrundfarbe **Bisque**, passender Größe und einem umlaufenden Rand von einem Zoll:

```
double rand = 96;
Canvas canPage = new Canvas();
canPage.Background = Brushes.Bisque;
canPage.Width = fpage.Width - 2 * rand;
canPage.Height = fpage.Height - 2 * rand;
canPage.Margin = new Thickness(rand);
```

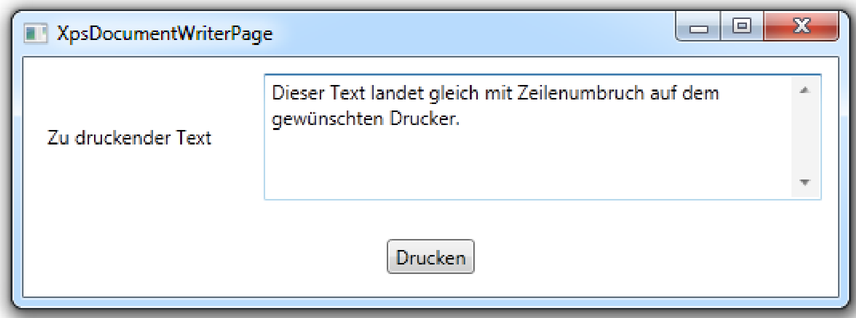
Auf die **Canvas**-Oberfläche setzen wir ein **TextBlock**-Objekt mit passender Größe und automatischem Zeilenumbruch:

```
TextBlock textBlock = new TextBlock();
textBlock.Width = canPage.Width;
textBlock.Height = canPage.Height;
textBlock.TextWrapping = TextWrapping.Wrap;
textBlock.Text = textBox.Text;
textBlock.FontFamily = new FontFamily("Arial");
textBlock.FontSize = 32;
canPage.Children.Add(textBlock);
```

Nun setzen wir den Textblock auf den **Canvas**-Container und den Container auf die Seite:

```
canPage.Children.Add(textBlock);
fpage.Children.Add(canPage);
```

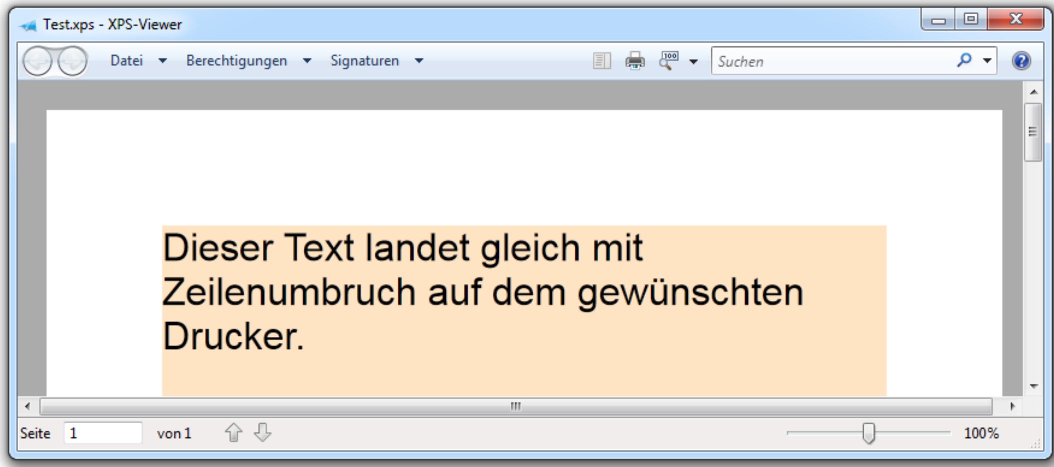
Den **TextBlock**-Inhalt darf der Benutzer über ein mehrzeiliges **TextBox**-Steuerelement festlegen:



Über den folgenden **Write**()-Aufruf wird ein **FixedPage**-Objekt zum Drucker befördert:

```
fdocWriter.Write(fpage);
```

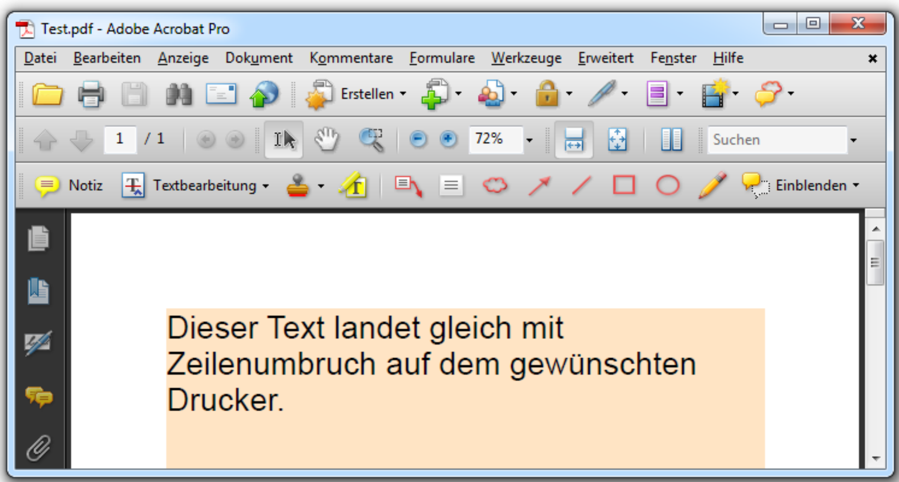
Verwendet man zur „Druckausgabe“ den (XPS-fähigen!) Pseudo-Druckertreiber Microsoft **XPS Document Writer** resultiert eine gültige XPS-Datei, die von XPS-Viewer so angezeigt wird:



Bei Verwendung eines anderen Druckertreibers (mit konventioneller Windows-Technik) habe ich allerdings beim Drucken eines **FixedPage**-Objekts nur eine leere Seite erhalten. Fügt man hingegen das **FixedPage**-Objekt gemäß Beschreibung in Abschnitt 19.4.2 in ein **FixedDocument** ein,

```
FixedDocument fdoc = new FixedDocument();
PageContent fpageContent = new PageContent();
((System.Windows.Markup.IAddChild)fpageContent).AddChild(fpage);
fdoc.Pages.Add(fpageContent);
fdocWriter.Write(fdoc);
```

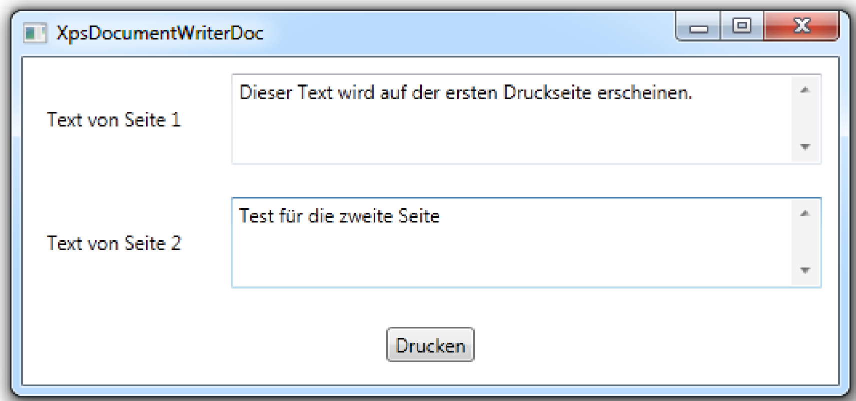
klappt das Drucken mit jedem Treiber, z.B.:



Für eine *mehrseitige* Druckausgabe per **XpsDocumentWriter** fügt man entsprechend viele **FixedPage**-Objekte mit **PageContent**-Verpackung in ein **FixedDocumet**-Objekt ein, das dann mit einer passenden **Write()**-Überladung gedruckt wird. Die folgende Methode

```
private void buttonPrint_Click(object sender, RoutedEventArgs e) {
   double rand = 96;
   PrintDocumentImageableArea imageableArea = null;
   XpsDocumentWriter fdocWriter = PrintQueue.CreateXpsDocumentWriter(ref imageableArea);
   if (fdocWriter == null)
      return;
   FixedDocument fdoc = new FixedDocument();
   fdoc.Pages.Add(PreparePage(textBox1, imageableArea.MediaSizeWidth,
                  imageableArea.MediaSizeHeight, rand));
   fdoc.Pages.Add(PreparePage(textBox2, imageableArea.MediaSizeWidth,
                  imageableArea.MediaSizeHeight, rand));
   fdocWriter.Write(fdoc);
}
```

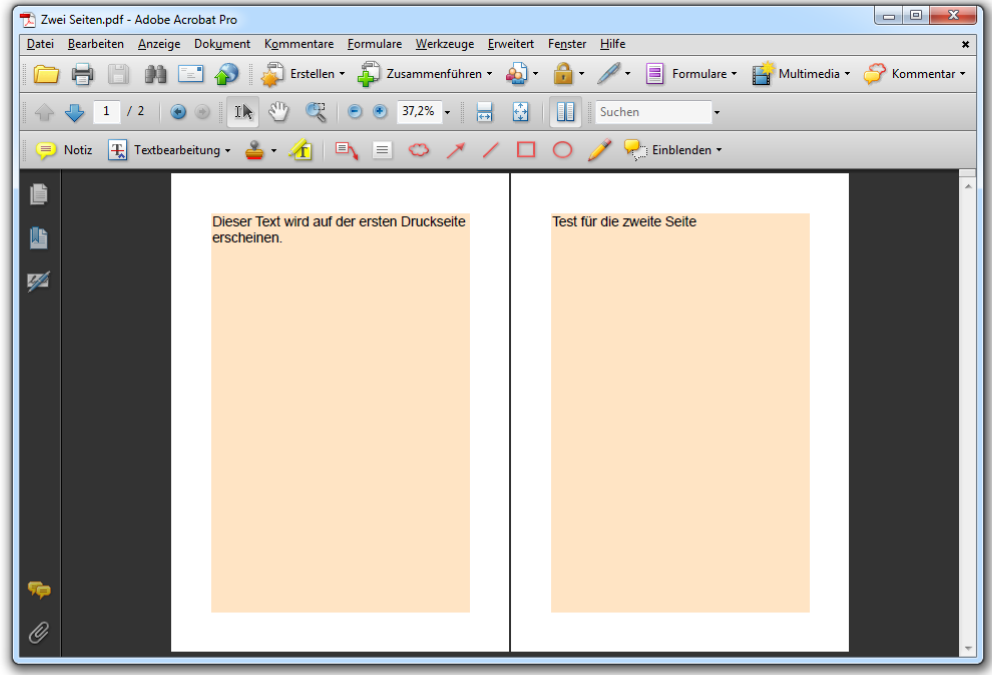
druckt auf zwei Seiten die Inhalte von zwei **TextBox**-Steuerelementen:



Den Aufbau der **PageContent**-Objekte erledigt die folgende Methode:

```
PageContent PreparePage(TextBox tb, double width, double height, double rand) {
   FixedPage fpage = new FixedPage();
   fpage.Width = width;
   fpage.Height = height;
   PageContent fpageContent = new PageContent();
   ((System.Windows.Markup.IAddChild)fpageContent).AddChild(fpage);
   Canvas canPage = new Canvas();
   canPage.Background = Brushes.Bisque;
   canPage.Width = fpage.Width - 2 * rand;
   canPage.Height = fpage.Height - 2 * rand;
   canPage.Margin = new Thickness(rand);
   TextBlock textBlock1 = new TextBlock();
   textBlock1.Width = canPage.Width;
   textBlock1.Height = canPage.Height;
   textBlock1.TextWrapping = TextWrapping.Wrap;
   textBlock1.Text = tb.Text;
   textBlock1.FontFamily = new FontFamily("Arial");
   textBlock1.FontSize = 32;
   canPage.Children.Add(textBlock1);
   fpage.Children.Add(canPage);
   return fpageContent;
}
```

Das Ergebnis:



19.4.4 Automatische Paginierung bei Fließtext

Erfreulicherweise kann man per **XpsDocumentWriter** auch ein **FlowDocument** (vgl. Abschnitt 19.3) drucken, z.B. den von einem **RichTextBox**-Steuerelement (vgl. Abschnitt 16) verwalteten Fließtext. Im Unterschied zu einem **FixedDocument** mit mehreren **FixedPage**-Objekten kennt ein **FlowDocument** (vgl. Abschnitt 19.3) keine Seitenstruktur. Allerdings implementiert die Klasse **FlowDocument** das Interface **IDocumentPaginatorSource** und stellt daher über die Eigenschaft **DocumentPaginator** ein Objekt zur Verfügung, das den Inhalt auf einzelne Seiten verteilen kann. Andererseits kann ein **XpsDocumentWriter** bei einigen **Write()**-Überladungen mit einem solchen Paginierer kooperieren. In der folgenden Methode

```
private void buttonPrint_Click(object sender, RoutedEventArgs e) {
   PrintDocumentImageableArea imageableArea = null;
   XpsDocumentWriter writer = PrintQueue.CreateXpsDocumentWriter(ref imageableArea);
   if (writer == null)
      return;

   Run run = new Run(textBox.Text);
   run.FontFamily = new FontFamily("Arial");
   run.FontSize = 16;
   FlowDocument flowDoc = new FlowDocument();
   flowDoc.Blocks.Add(new Paragraph(run));
   flowDoc.ColumnWidth = imageableArea.ExtentWidth;

   DocumentPaginator paginator = (flowDoc as IDocumentPaginatorSource).DocumentPaginator;
   paginator.PageSize = new Size(imageableArea.ExtentWidth, imageableArea.ExtentHeight);

   writer.Write(paginator);
}
```

wird ein **Flowdocument** erstellt, das aus einem Block vom Typ **Paragraph** besteht, der ein **Inline**-Element vom Typ **Run** enthält (vgl. Abschnitte 19.1 und 19.3). Weil die Eigenschaft **Document-Paginator** durch explizite Schnittstellenimplementierung realisiert ist (vgl. Abschnitt 8.4), muss sie über den Interface-Typ angesprochen werden.

```
DocumentPaginator paginator = (flowDoc as IDocumentPaginatorSource).DocumentPaginator;
```

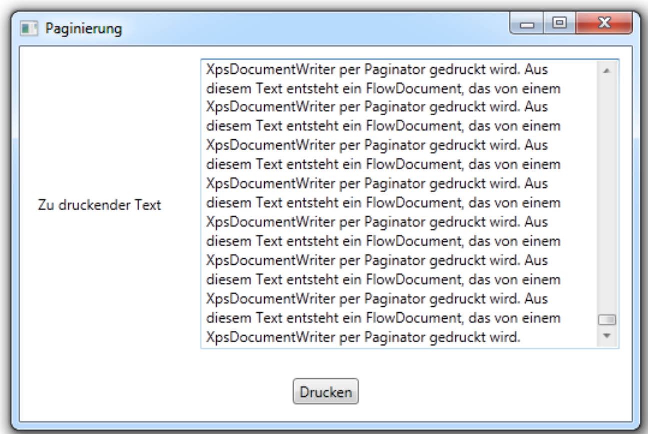
Per Voreinstellung verwendet der Paginierer eine Seitengröße von 8,5 × 11 Zoll. Daher wird im Beispiel seiner **PageSize**-Eigenschaft die physikalische Seitengröße des vom Benutzer gewählten Druckers zugewiesen:

```
paginator.PageSize = new Size(imageableArea.ExtentWidth, imageableArea.ExtentHeight);
```

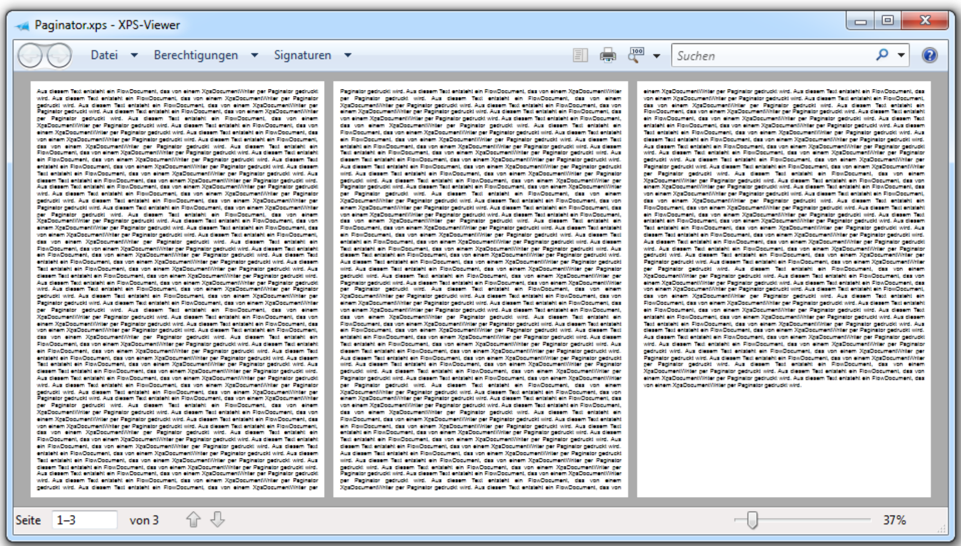
Um den voreingestellte 2-Spalten-Druck zu verhindern, erhält die **FlowDocument**-Eigenschaft **ColumnWidth** die Breite des Druckbereichs als Wert:

```
flowDoc.ColumnWidth = imageableArea.ExtentWidth;
```

Der vom Benutzer in ein **TextBox**-Steuerelement eingetragene Fließtext



landet beim Druck mit dem Pseudo-Druckertreiber Microsoft **XPS Document Writer** in einer 3-seitigen XPS-Datei, die von XPS-Viewer so angezeigt wird:



Mit etwas Aufwand lassen sich Kopf- und Fußbereiche (z.B. mit automatischer Seitennummerierung) nachrüsten, indem man die Seitengröße für den Paginierer reduziert und ein **FixedDocument** erstellt, das den paginierten **FlowDocument**-Inhalt mit ergänzten Kopf- und Fußbereichen enthält (siehe Sells & Griffiths 2007, S. 539ff).

19.4.5 Asynchrones Drucken

Bei einer längeren Druckausgabe können Sie es den Benutzern Ihres Programms nicht zumuten, auf die Fertigstellung des Druckauftrags zu warten. Dies lässt sich mit Hilfe der asynchronen Ausgabemethoden der Klasse **XpsDocumentWriter** leicht vermeiden, z.B.:

```
writer.WriteAsync(paginator);
```

Zur Information über den Verlauf und das ende des Druckauftrags bietet die Klasse **XpsDocumentWriter** diverse Ereignisse, z.B. **WritingCompleted**.

19.4.6 Mehr Kontrolle über die Druckkonfiguration

19.4.6.1 PrintQueue

Die Klasse **PrintQueue** repräsentiert einen Drucker und die die von ihm zu verarbeitende Druckjobwarteschlange. Wir haben bisher die statische **PrintQueue**-Methode **CreateXpsDocumentWriter()** kennen gelernt, die uns ein **XpsDocumentWriter**-Objekt verschafft, das mit dem vom Benutzer gewählten Drucker (also mit dem zugehörigen **PrintQueue**-Objekt) verbunden ist.

Soll kein Druckauswahlstandarddialog angezeigt, sondern der Standarddrucker des angemeldeten Benutzers verwendet werden, bringt man das zugehörige **PrintQueue**-Objekt so in Erfahrung:

```
LocalPrintServer localPC = new LocalPrintServer();
PrintQueue defPrintQueue = localPC.DefaultPrintQueue;
```

Um zu einem bereits bekannten **PrintQueue**-Objekt einen **XpsDocumentWriter** anzufordern, verwendet man eine **CreateXpsDocumentWriter()**-Überladung mit **PrintQueue**-Parameter, z.B.:

```
XpsDocumentWriter xpsDW = PrintQueue.CreateXpsDocumentWriter(defPrintQueue);
```

Über die **PrintQueue**-Methode **GetPrintCapabilities()** erhält man ein **PrintCapabilities**-Objekt, das diverse Merkmale des verbundenen Druckers kennt, z.B. den Druckbereich:

```
PageImageableArea imagArea = defPrintQueue.GetPrintCapabilities().PageImageableArea;
```

Über zahlreiche **PrintQueue**-Eigenschaften lassen sich aktuelle Betriebszustände des Druckers abfragen, z.B. **HasToner**, **IsPaperJammed**.

19.4.6.2 PrintServer

Die Klasse **PrintServer** repräsentiert einen Rechner im Netzwerk, der Druckdienste anbietet. Es steht u.a. ein Konstruktor mit **String**-Parameter für den Namen des Rechners zur Verfügung. z.B.:

```
PrintServer wgPrintServer = new PrintServer(@"\\PrintServer");
```

Über die **PrintServer**-Methode **GetPrintQueues()** erhält man eine Kollektion der angeschlossenen Drucker. Ist der Name einer Druckerwarteschlange bekannt, kann man über die Methode **GetPrintQueue()** das zugehörige **PrintQueue**-Objekt anfordern.

Wir haben in Abschnitt 19.4.6.1 schon die **PrintServer**-Ableitung **LocalPrintServer** kennen gelernt, deren Eigenschaft **DefaultPrintQueue** auf den voreingestellten Drucker des angemeldeten Benutzers zeigt:

```
LocalPrintServer localPC = new LocalPrintServer();
```

19.4.6.3 PrintDialog

Ruft man die statische **PrintQueue**-Methode **CreateXpsDocumentWriter()** ohne **PrintQueue**-Parameter auf, erscheint automatisch der Druckjob-Standarddialog. Hier kann der Benutzer nicht nur einen Drucker wählen, sondern eventuell auch andere Eigenschaften des Druckauftrags festlegen (z.B. den Seitenbereich). Durch die explizite Verwendung eines **PrintDialog**-Objekts gewinnen Sie die Möglichkeit, vor dem Erscheinen des Dialogs Voreinstellungen festzulegen.

Nachdem ein **PrintDialog** vom Benutzer mit **OK** quittiert worden ist, kann man über die **PrintDialog**-Eigenschaften **PrintQueue** bzw. **PrintTicket** den gewählten Drucker bzw. die gewählten Druckjob-Eigenschaften in Erfahrung bringen, z.B.:

```
pq = printDialog.PrintQueue;
```

Über die **PrintDialog**-Eigenschaften **PrintableAreaWidth** und **PrintableAreaHeight** ermittelt man zum verbundenen Drucker die Breite und Höhe des Druckbereichs, z.B.:

```
FixedPage fpage = new FixedPage();
fpage.Width = printDialog.PrintableAreaWidth;
fpage.Height = printDialog.PrintableAreaHeight;
```

Verwendet man stets dasselbe **PrintDialog**-Objekt, trifft der Benutzer bei wiederholten Aufrufen immer auf die zuletzt verwendeten Einstellungen.

Über seine **PrintDocument**-Methode kann der **PrintDialog** auch die Druckausgabe übernehmen, wobei im ersten Parameter ein Paginator zur Druckquelle (vgl. Abschnitt 19.4.4) und im zweiten Parameter ein Name für den Druckjob anzugeben ist. Im folgenden Code-Segment wird ein Objekt der Klasse **FixedDocument** ausgegeben, die über einen Paginator verfügt:

```
FixedDocument fdoc = new FixedDocument();
PageContent fpageContent = new PageContent();
((System.Windows.Markup.IAddChild)fpageContent).AddChild(fpage);
fdoc.Pages.Add(fpageContent);
printDialog.PrintDocument(fdoc.DocumentPaginator, "Name des Druck-Jobs");
```

20 Datenbankprogrammierung mit ADO.NET

Die Verwaltung großer Datenbestände (z.B. Personaldaten einer Firma, Bestellungen eines Webshops) ist eine häufige und oft unternehmenswichtige Aufgabe für Softwaresysteme. In diesem Kapitel steigen wir in die Datenbankprogrammierung in C# ein und behandeln dabei folgende Themen:

- Grundlagen der relationalen Datenbanktechnik (inkl. SQL)
- Microsofts Datenbankverwaltungsprogramme
- Traditionelle Datenbankprogrammierung mit dem ADO.NET - Framework
- Typisierte DataSets

In Kapitel 21 über die Abfragetechnik LINQ (*Language Integrated Query*) und in Kapitel 22 über das Entity Framework, das als ORM-Lösung (*Object-Relational Mapping*) die Kluft zwischen objektorientierter Programmierung und relationaler Datenbanktechnik überbrückt, werden moderne Entwicklungen in der Datenbankprogrammierung vorgestellt, die speziell für komplexe Projekte empfehlenswert sind.

Auf angehende Datenbankspezialisten wartet auch danach noch eine große Zahl wichtiger Themen und eine entsprechend reichhaltige Literatur (siehe z.B. Doberenz & Gewinnus 2010).

20.1 Datenbankmanagementsysteme

Für den Begriff *Datenbank* schlägt Ebner (2000, S. 21) folgende Definition vor:

> *Eine Datenbank ist eine Sammlung von nicht-redundanten Daten, die von mehreren Anwendungen benutzt werden kann.*

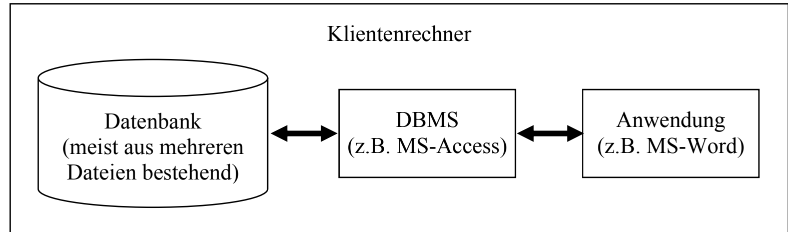
Redundanz würde sich z.B. in der Datensammlung eines Versandhauses schnell einstellen, wenn man für jede Bestellung einen Datensatz anlegen und dabei z.B. auch die Adresse des Bestellers einbeziehen würde. Sobald von einem Kunden mehrere Bestellungen vorlägen, wäre z.B. seine Adresse mehrfach vorhanden. Neben dem unsinnigen Erfassungsaufwand und der Platzverschwendung hat die Redundanz einen weiteren Nachteil: die Gefahr **inkonsistenter** Daten.

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 das konsistente, effiziente und sichere Speichern der Daten verantwortlich und ermöglicht simultane Zugriffe durch mehrere Anwendungsprogramme, sofern deren Benutzer über entsprechende Rechte verfügen. Bekannte Beispiele für diese Softwaregattung sind:

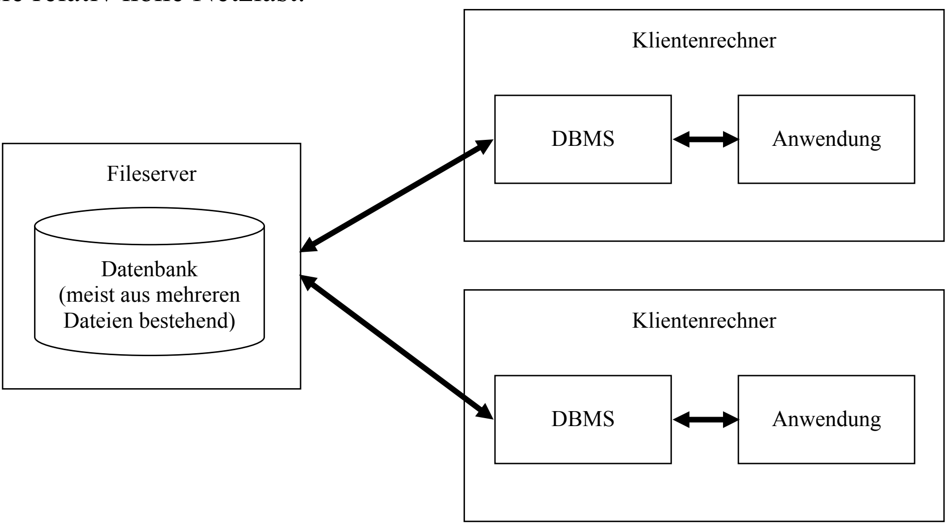
- Microsoft SQL Server (mit Express- und Compact- Edition)
- MySQL
- Oracle Database
- DB2 von IBM usw.

Auch bei Datenbeständen, die voraussichtlich nur von einer einzigen Anwendung benutzt werden sollen, kann es sinnvoll (bequem und sicher) sein, Verwaltung und Abruf von Daten einem DBMS zu überlassen. Man kann drei Datenbankeinsatzszenarien unterscheiden, wobei sich aber für den Datenbankzugriff durch ein Anwendungsprogramm in C# praktisch keine Unterschiede ergeben:

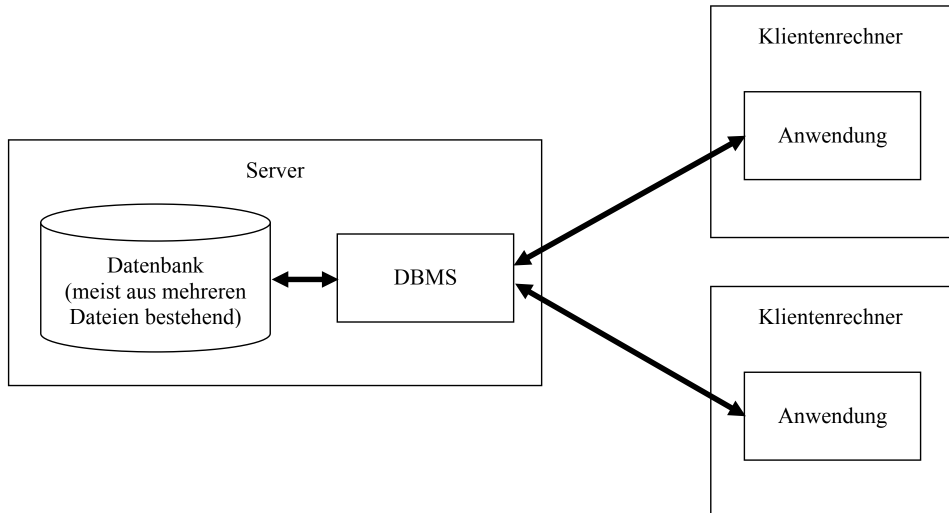
- **Einzelbenutzerdatenbank**
  Anwendungsprogramm und DBMS laufen auf demselben Rechner, z.B. bei einer Seriendrucklösung bestehend aus den MS-Office - Komponenten Access und Word.



- **Mehrbenutzerdatenbank per Fileserver**
  Hier laufen Anwendungsprogramm und DBMS auf demselben Rechner, während sich die Datenbank auf einem zentralen Fileserver befindet und damit für mehrere Rechner zugänglich ist. Wenn mehrere DBMS-Programme auf dieselbe Datenbank zugreifen, müssen sie ihre Tätigkeit über eine Sperrdatei (engl. *lock file*) koordinieren, wobei es leicht zu Störungen kommen kann. Auch andere Hilfsdateien (z.B. mit Indizes zur Beschleunigung der Suche) werden fehleranfällig von vielen Programmen benutzt. Ungünstig an der Fileserver-Lösung ist auch die relativ hohe Netzlast.



- **Mehrbenutzerdatenbank per Client-Server - Lösung**
  Es ist nur *ein* DBMS im Spiel, das sich wie die Datenbank auf einem Server befindet (nicht unbedingt auf demselben). Das DBMS bedient und koordiniert die Zugriffe durch Anwendungsprogramme auf Klientenrechnern, so dass perfekte Datenbankintegrität und minimaler Netzwerkverkehr resultieren.



Wir werden in diesem Kurs zwei kostenlos verfügbare Varianten von Microsofts SQL-Server einsetzen:

- **Microsoft SQL Server Express**
  Diese Software erlaubt Client-Server - Lösungen mit wenigen (für uns irrelevanten) Einschränkungen gegenüber der kostenpflichtigen Variante. Sie ist als Installationsoption im Visual Studio 2010 Express enthalten, wobei wir allerdings die separate Installation einer aktuelleren Version mit Hilfsprogrammen bevorzugen.
- **Microsoft SQL Server Compact**
  Wenn nur eine eingebettete Einzelbenutzerdatenbank benötigt wird, ist diese schlanke Software zu empfehlen (Plattenspeicherbedarf: 2 MB). Sie wird automatisch zusammen mit dem Visual Studio 2010 Express installiert.

Das .NET – Framework enthält eine Bibliothek namens **ADO.NET** (*ActiveX Data Objects .NET*) mit Klassen zur Kooperation mit DBMS-Software und zur Verwendung von anderen Datenquellen (z.B. XML-Dateien), die ältere Microsoft-Datenbankzugriffstechniken (DAO, ADO) ablösen soll. Als Vorteile gegenüber dem direkten Vorgänger ADO (*ActiveX Data Objects*) sind vor allem zu nennen:

- ADO wurde für COM-basierte Anwendungen und die zugehörige Datenzugriffschnittstelle OLEDB entworfen. Demgegenüber ist ADO.NET komplett in das .NET-Framework integriert.
- Verbesserte Unterstützung der *verbindungslosen* Arbeitsweise über die Klasse **DataSet** (siehe unten).
- XML ist als alternatives Datenformat nahtlos integriert.

20.2 Relationale Datenbanken

Heute arbeitet praktisch jedes DBMS mit der **relationalen Datenbankstruktur**, die sich gegenüber alternativen Bauformen (z.B. hierarchische Datenbank, Netzwerkdatenbank) weitgehend durchgesetzt hat und von der neueren objektorientierten Datenbankstruktur noch nicht ernsthaft bedrängt wird. Es sind zahlreiche relationale RDBMS-Produkte (relationale Datenbankmanagementsysteme) mit einem hohen Reifegrad verfügbar. Erfreulicherweise ist der Zugriff auf relationale Datenbanken über die Abfragesprache SQL (siehe Abschnitt 20.3) weitgehend standardisiert. Von praktischer Bedeutung ist auch, dass unsere Entwicklungsumgebung Visual Studio mit Microsofts RDBMS-Produkten exzellent kooperiert.

20.2.1 Tabellen

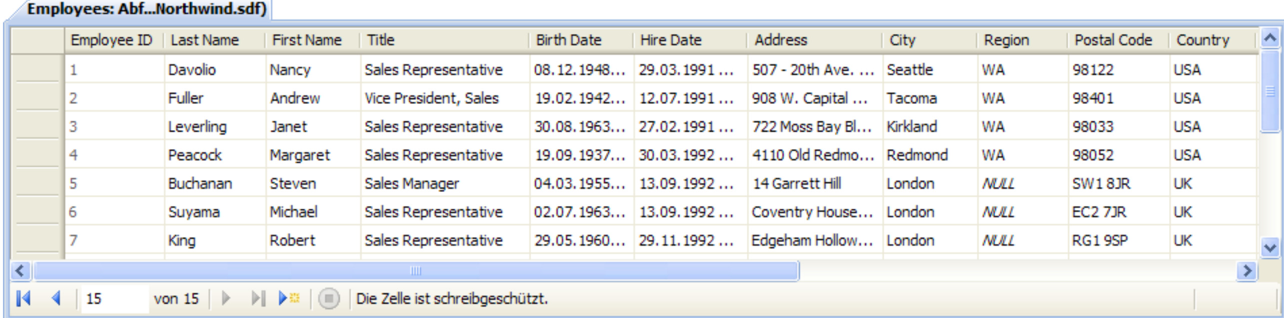
Bei einer relationalen Datenbank sind die Daten in **Tabellen**[1] angeordnet, wobei die **Zeilen** auch als **Datensätze** (engl: *records*), **Tupel** oder **Fälle** und die **Spalten** auch als **Felder**, **Attribute**, **Merkmale** oder **Variablen** bezeichnet werden:

[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 EDV-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 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).

- Jede Tabelle enthält Zeilen eines bestimmten Typs (z.B. Kunden, Artikel, Lieferanten, Reklamationen). Welche Tabellen benötigt werden, hängt von 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. INT, DATE).
- Eine Tabelle besitzt in der Regel einen **Primärschlüssel** (engl.: *primary key*) mit 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 sind Primärschlüssel mit *numerischem* Datentyp aus Performanzgründen zu bevorzugen. Man kann mit einem RDBMS vereinbaren, die Werte einer Primärschlüsselspalte ausgehend von einem Startwert automatisch zu erhöhen (engl.: *auto increment*).
- Man kann auch für beliebige *andere* Spalten die eben für Primärschlüssel genannten **Restriktionen** (engl.: *constraints*) formulieren:
    - Eindeutigkeit der Werte
    - Verbot fehlender Werte
- Für einen Primärschlüssel ist ein **Index** vorhanden, welcher die Suche nach bestimmten Werten dank Sortierung beschleunigt. Zusätzlich können in einer Tabelle zu weiteren Spalten Indizes angelegt werden. Weil für Aufbau und Pflege der Indizes ein gewisser Aufwand erforderlich ist, sollte man sich auf die zur Beschleunigung von Suchanfragen erforderlichen Indizes beschränken.

Wir betrachten in Kapitel 20 die von Microsoft angebotene Beispieldatenbank zu einer erfundenen Firma namens **Northwind** (siehe Abschnitt 20.4.2.2 zum Bezug und zur Installation der Datenbank). In der **Northwind**-Datenbank befindet sich u.a. eine Tabelle mit dem Personal der Firma:

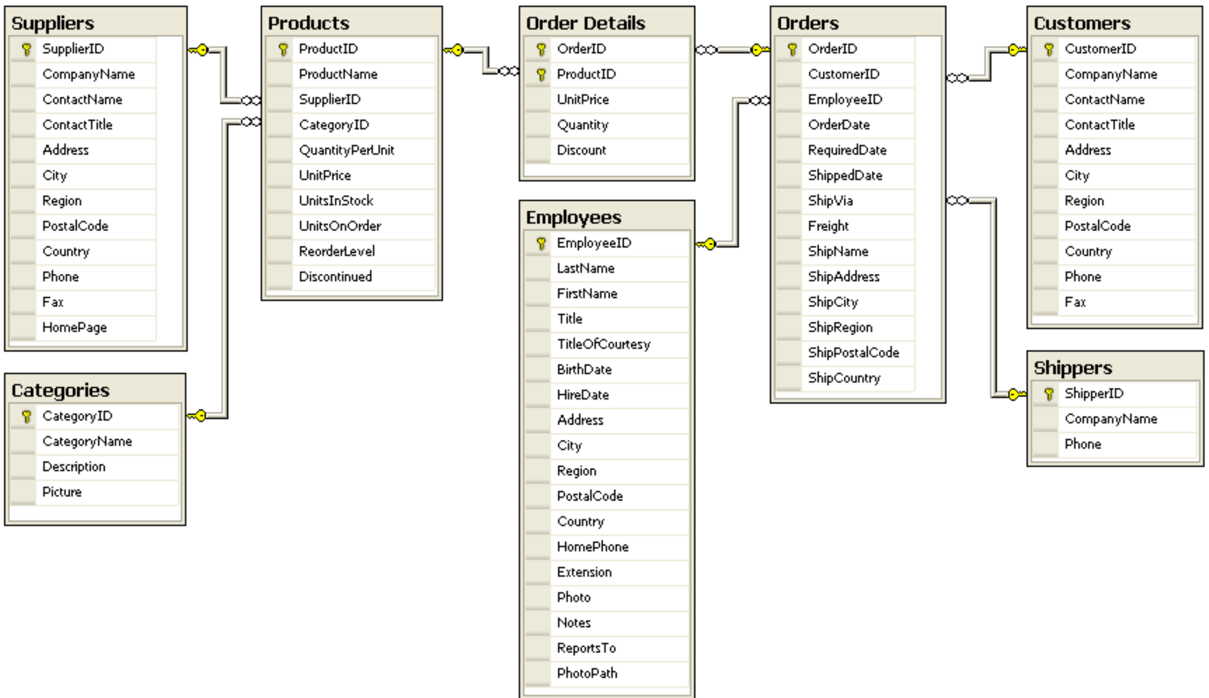


Employees: Abf...Northwind.sdf)

Employee ID	Last Name	First Name	Title	Birth Date	Hire Date	Address	City	Region	Postal Code	Country
1	Davolio	Nancy	Sales Representative	08.12.1948...	29.03.1991 ...	507 - 20th Ave. ...	Seattle	WA	98122	USA
2	Fuller	Andrew	Vice President, Sales	19.02.1942...	12.07.1991 ...	908 W. Capital ...	Tacoma	WA	98401	USA
3	Leverling	Janet	Sales Representative	30.08.1963...	27.02.1991 ...	722 Moss Bay Bl...	Kirkland	WA	98033	USA
4	Peacock	Margaret	Sales Representative	19.09.1937...	30.03.1992 ...	4110 Old Redmo...	Redmond	WA	98052	USA
5	Buchanan	Steven	Sales Manager	04.03.1955...	13.09.1992 ...	14 Garrett Hill	London	NULL	SW1 8JR	UK
6	Suyama	Michael	Sales Representative	02.07.1963...	13.09.1992 ...	Coventry House...	London	NULL	EC2 7JR	UK
7	King	Robert	Sales Representative	29.05.1960...	29.11.1992 ...	Edgeham Hollow...	London	NULL	RG1 9SP	UK

15 von 15 Die Zelle ist schreibgeschützt.

Das folgende Datenbankdiagramm zeigt die wichtigsten **Northwind**-Tabellen mit den zugehörigen Feldern und den Primärschlüsseln (einfach oder zusammengesetzt):

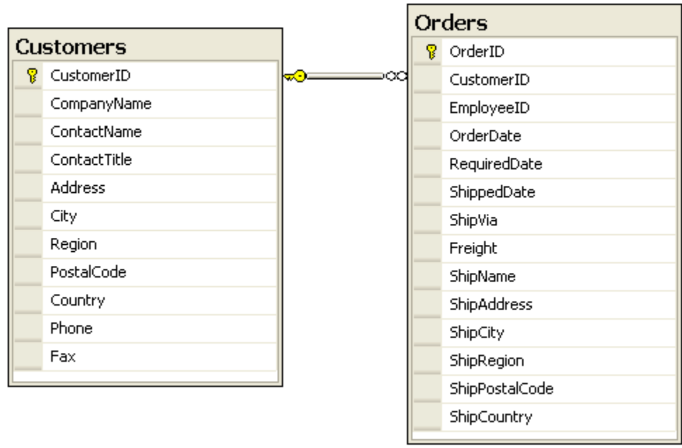


Ein redundanzfreier Datenbankentwurf erfordert bei den meisten Aufgabenstellungen das Verteilen der Daten auf *mehrere* Tabellen. Man spricht hier vom *Normalisieren* einer Datenbank. Damit das RDBMS Inkonsistenzen verhindern kann (z.B. einen Fall der **Orders**-Tabelle mit einer Kundennummer, die in der **Customers**-Tabelle nicht vorhanden ist), müssen Beziehungen zwischen den Tabellen definiert werden (siehe Abschnitt 20.2.2).

Die eben zur Illustration eingesetzten Bildschirmphotos stammen von der Visual C# 2010 Express Edition. Wie unsere Entwicklungsumgebung als SQL Server - Frontend genutzt werden kann, erfahren Sie in Abschnitt 20.7.

20.2.2 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 so genannten **Master-Detail** – 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 Datenhaltung gewährleistet ist.

Aus der Master-Tabelle wirkt ein Primärschlüssel bei der Beziehung mit. Die zur Verknüpfung herangezogene Spalte der Details-Tabelle wird auch als *Fremdschlüssel* bezeichnet. Sie muss denselben Datentyp haben wie der zugehörige Primärschlüssel, aber nicht unbedingt denselben Namen.

In der Beispieldatenbank sind zahlreiche Master-Detail - Beziehungen vorhanden. Bei Betrachtung in umgekehrter Richtung wird daraus jeweils eine Detail-Master - bzw. (n:1) – Beziehung.

Aufgrund der im Datenbankentwurf vereinbarten Beziehungen sorgt ein DBMS 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 **Schema** einer relationalen Datenbank beinhaltet Informationen über

- die Tabellen
  Für jede Tabelle sind die Spalten (Felder) mit Namen und Datentyp zu definieren sowie Restriktionen festzulegen (z.B. Eindeutigkeitsforderung für die Werte einer Spalte).
- die Beziehungen zwischen den Tabellen

20.3 SQL

Ein RDBMS interagiert mit anderen Softwaresystemen über die **Structured Query Language** (SQL). Microsofts SQL-Server 2008 R2 bietet unter der Bezeichnung **T-SQL** eine Variante, die im Vergleich zum ISO-Standard SQL-92 aus dem Jahr 1992 zusätzlich so genannte *Stored Procedures* und *Transaktionen* unterstützt. Mit solchen Datenbank-Spezialthemen werden wir uns aber nicht beschäftigen.

20.3.1 Überblick

Ein RDBMS interagiert mit anderen Programmen über die **Structured Query Language** (SQL). Wie der Name *SQL* nahe legt, ist die *Abfrage* von Informationen klarer Einsatzschwerpunkt, doch deckt der Sprachumfang alle Aufgaben der Datenverwaltung ab, wobei sich folgende Teilmengen unterscheiden lassen:

- **DDL** (**D**ata **D**efinition **L**anguage)
  Mit den DDL-Befehlen (z.B. **CREATE TABLE**, **CREATE INDEX**, **DROP TABLE**) kann man das Schema einer Datenbank definieren oder verändern.
- **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 zusammenführt werden. Oft spricht man beim **SELECT**-Befehl von einer *Auswahlabfrage* im Unterschied zu den anschließend behandelten *Aktionsabfragen*.

- **DML** (**D**ata **M**anipulation **L**anguage)
  Mit den DML-Befehlen werden Datenbanktabellen verändert:
  - **INSERT**
    Hängt einen Datensatz am Ende einer Tabelle an
  - **DELETE**
    Löscht einen Datensatz
  - **UPDATE**
    Ändert die Werte von Fällen

  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 20.5.6).

Auf der Webseite

http://msdn.microsoft.com/de-de/library/ms365303.aspx

bietet Microsoft eine T-SQL - Referenz und Lernprogramme an.

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 nahe liegenden Gründen bei C# - Code 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 aufeinander folgende Punkte wird zum Ausdruck gebracht, dass eine Liste beliebig verlängert werden darf.

Im Unterschied zu C# ist in SQL die Groß/Klein-Schreibung irrelevant. Es hat sich aber eingebürgert, die Schlüsselwörter groß zu schreiben.

20.3.2 Spalten abrufen

Will man per **SELECT**-Befehl *alle* Spalten einer Tabelle abrufen, 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
```

20.3.3 Fälle auswählen über die WHERE-Klausel

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 Zeichenkettenliterale durch *einfache* Anführungszeichen zu begrenzen. Treten in *Namen* (z.B. von Tabellen oder Spalten) Leerzeichen auf, sind diese Namen durch *doppelte* Anführungszeichen zu begrenzen, 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 folgende 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 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')
```

20.3.4 Daten aus mehreren Tabellen zusammenführen

Sobald die **FROM**-Klausel *mehrere* Tabellen enthält, muss den Namen der abgerufenen Spalten ein Tabellenname vorangestellt werden, z.B.:

```
SELECT Customers.CompanyName, Orders.OrderDate
  FROM Customers, Orders
  WHERE Customers.CustomerID = Orders.CustomerID
```

Der bei Beteiligung mehrerer Tabellen anfallende Schreibaufwand lässt sich durch Abkürzungen begrenzen:

```
SELECT C.CompanyName, B.OrderDate FROM Customers C, Orders B
 WHERE C.CustomerID = B.CustomerID
```

Ohne WHERE-Klausel werden 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. Obige WHERE-Klausel legt fest, 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. Zu den beiden in einer Master-Detail - Beziehung stehen Tabellen erhalten wir also ein Abfrageergebnis, das alle Detail-Zeilen mit zusätzlichen Master-Daten enthält.

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
```

20.3.5 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
```

20.3.6 Auswertungsfunktionen

Bei den Spaltendefinitionen einer **SELECT**-Anweisung stehen auch einige Auswertungsfunktionen zur Verfügung. Im folgenden Beispiel wird der mittlere Preis aller Fälle in der **Northwind**-Tabelle **Products** festgestellt:

```
SELECT AVG(UnitPrice) FROM Products
```

Wichtige SQL-Auswertungsfunktionen:

Funktion	Beschreibung
COUNT(*col*)	Zählt die 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 Spalte *col*
MAX(*col*)	Ermittelt das Maximum in 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).

20.3.7 Daten gruppieren

Über die Klausel **GROUP BY** kann man Untergruppen bilden, für die sich obige Funktionen auswerten lassen. In unserem Beispiel kann man etwa über die Variable **Country** aggregieren, um die Kundenzahlen in den einzelnen Herkunftsländern festzustellen:

```
SELECT Country, COUNT(*) FROM Customers GROUP BY Country
```

20.4 Microsoft SQL Server 2008 R2 Express Edition

20.4.1 Eigenschaften

Microsofts SQL Server 2008 R2 Express Edition ist ein kostenloses, voll funktionstüchtiges, zeitlich unbefristet verwendbares Datenbankmanagementsystem (DBMS). Im Vergleich zur ebenfalls kostenlosen Microsofts SQL Server Compact Edition und zur kostenpflichtigen Standard Edition von Microsofts SQL Server 2008 R2 bestehen folgende Unterschiede:

	Standard Edition	**Express Edition**	**Compact Edition**
Client-Server - Betrieb	Ja	Ja	Nein
Maximale Datenbankgröße	524 PB[110]	10 GB	4 GB
Maximal nutzbarer Hauptspeicher	64 GB	1 GB	unbegrenzt
Maximale Zahl nutzbarer CPUs	4	1	unbegrenzt
Empfohlene Verwendung	Professionelle Client-Server-Lösungen	Einfache Client-Server-Lösungen	Eingebettete Lösungen

Mehr Details (insbesondere zu den Unterschieden zwischen den *verschiedenen* kostenpflichtigen Varianten) finden Sie auf der folgenden Webseite:

http://www.microsoft.com/sqlserver/2008/en/us/editions.aspx

Bei allen Varianten von Microsofts SQL-Server haben wir von der guten Integration mit dem Visual Studio (siehe Abschnitt 20.7) u.a. folgende Vorteile:

- bequeme Datenbankprogrammierung
- Ablage von kompletten .NET - Objekten in einer Datenbank

Einer Verwendung der Express- oder Compact-Edition vom Microsofts SQL Server in Ihren eigenen Programmen stehen keine lizenzrechtlichen Probleme im Wege, weil Microsoft eine Verbreitung zusammen mit Ihren Anwendungen erlaubt. Im Fall der Express Edition ist dazu allerdings eine Registrierung über die folgende Webseite erforderlich:

http://www.microsoft.com/sqlserver/2008/en/us/express/redistregister.aspx

20.4.2 Installation

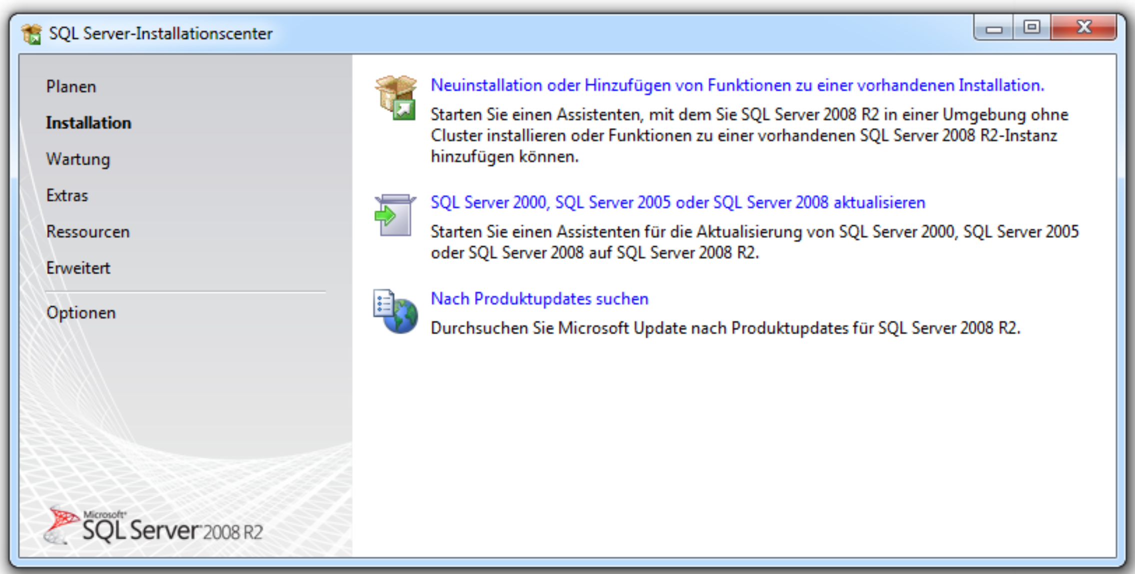
20.4.2.1 SQL Server 2008 R2 Express with Tools

Bei der Installation unserer Entwicklungsumgebung Visual C# 2010 Express Edition (siehe Abschnitt 2.2.2.1) haben wir bewusst auf die Option verzichtet, den *SQL Server 2008 Express Edition* gleich mit auf die Festplatte zu befördern, weil der integrierten Variante die nützliche graphische Bedienoberfläche *SQL Server 2008 Management Studio Express* fehlt. Daher installieren wir nun das von Microsoft kostenlos zum Download angebotene Paket *SQL Server 2008 R2 Express with Tools*, das neben dem DBMS auch das Management Studio enthält.

Der *Microsoft SQL Server Compact 3.5*, den wir in diesem Kapitel ebenfalls verwenden, ist übrigens automatisch zusammen mit der *Visual C# 2010 Express Edition* installiert worden.

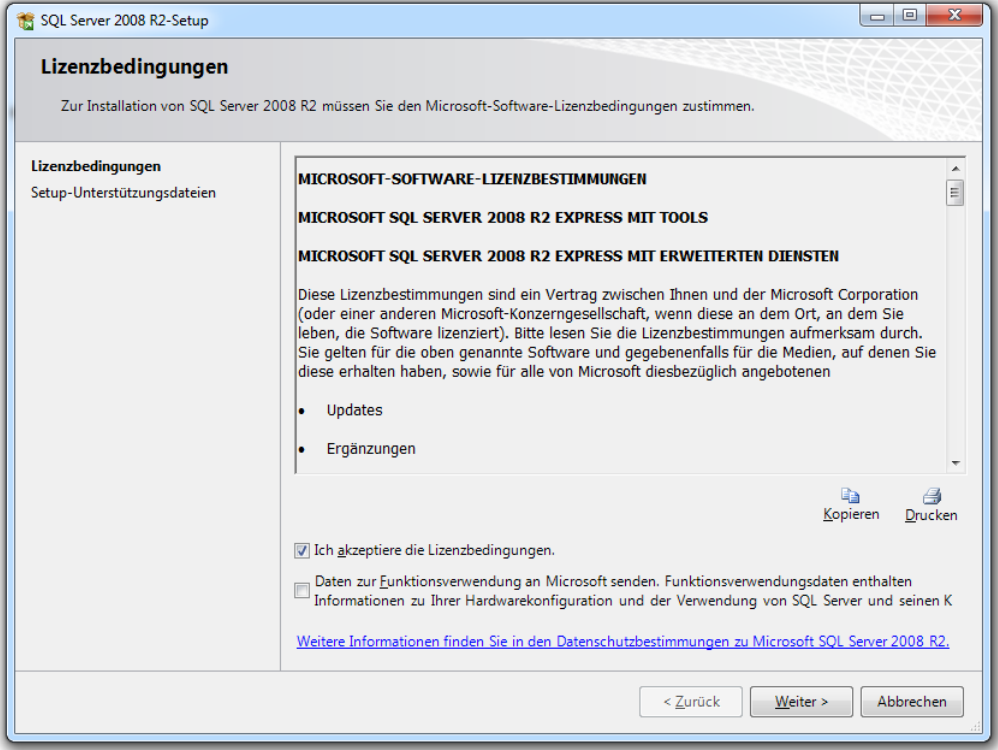
Nach einem Doppelklick auf die herunter geladene Datei (z.B. **SQLEXPRWT_x64_DEU.exe** für die 64-Bit-Variante) startet das **Installationscenter**:

[110] Bei Verwendung der dezimalen Notation ergeben 1000 Gigabyte ein Terabyte (Abkürzung: TB) und 1000 Terabyte ein Petabyte (Abkürzung: PB).

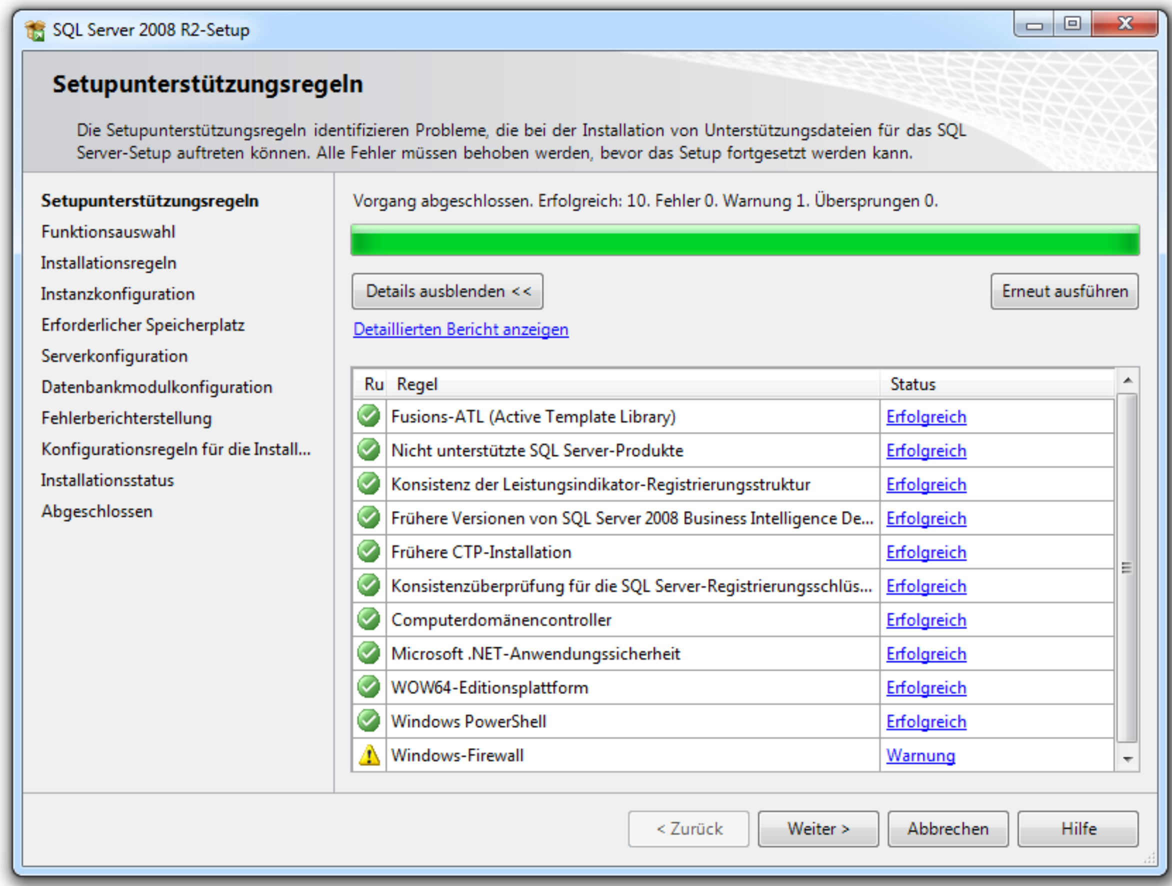


Vermutlich werden auch Sie eine **Neuinstallation** vornehmen wollen.

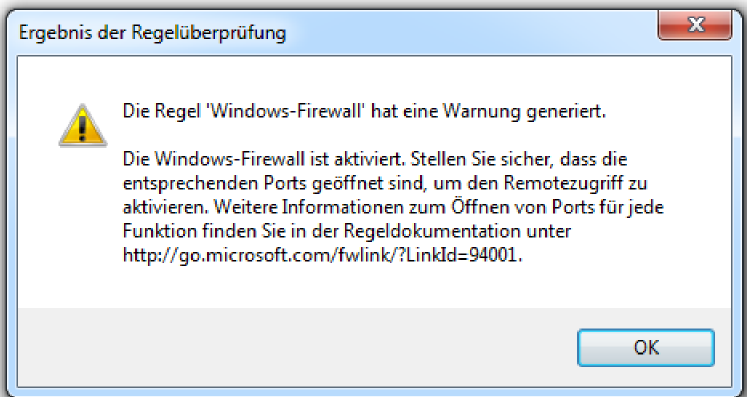
Nachdem Sie die Lizenzbedingungen akzeptiert haben,



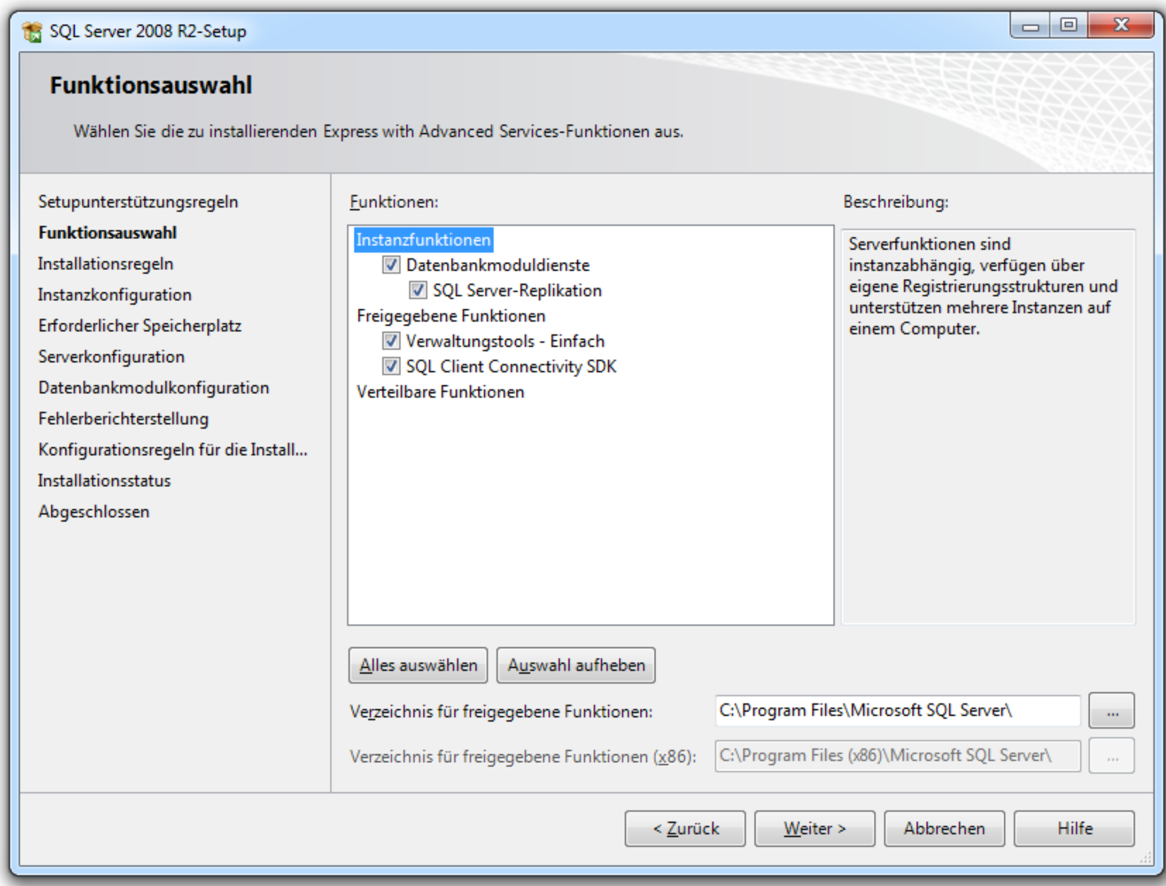
werden **Setup-Unterstützungsdateien** installiert, sofern alle **Setupunterstützungsregeln** erfüllt sind:



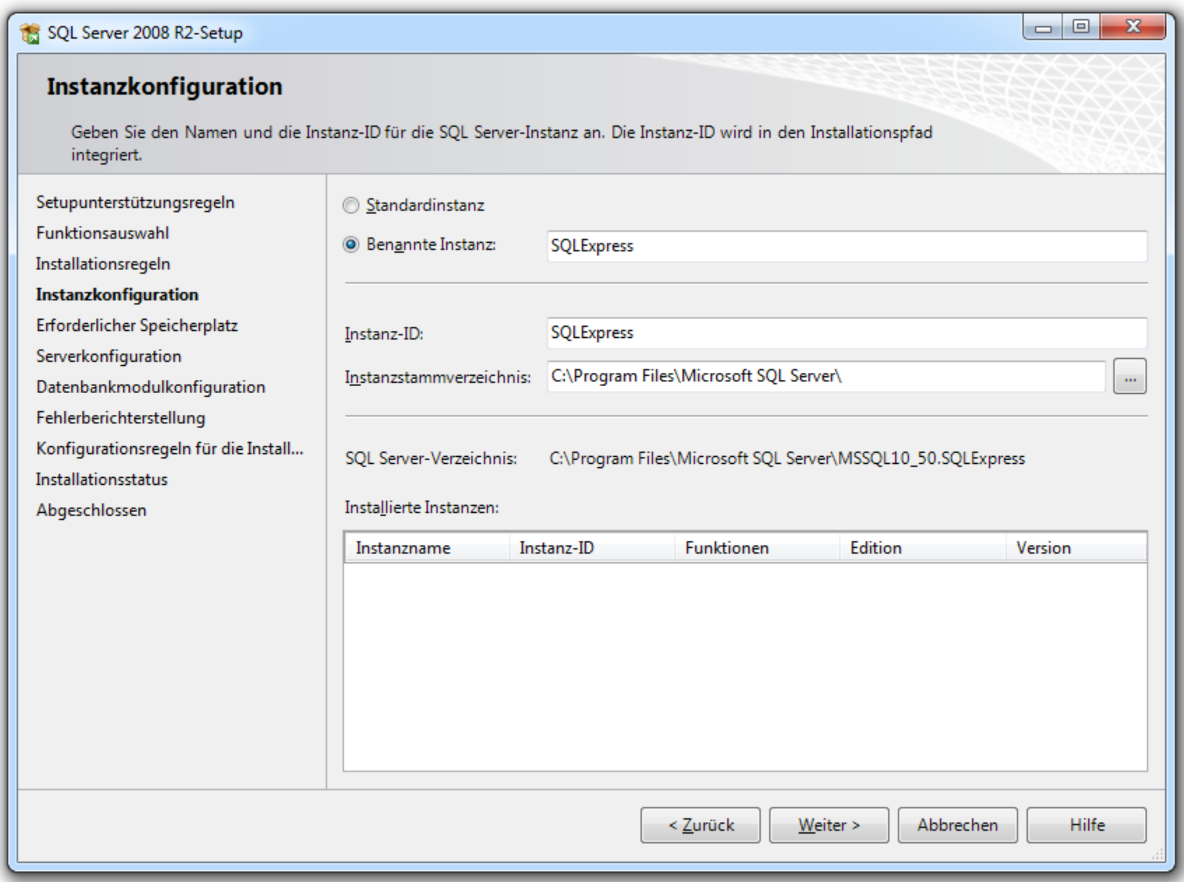
Auf einem korrekt gesicherten Rechner erscheint ein Ausrufezeichen neben der Regel **Windows-Firewall**, das nach einem Klick auf den Link **Warnung** so erläutert wird:



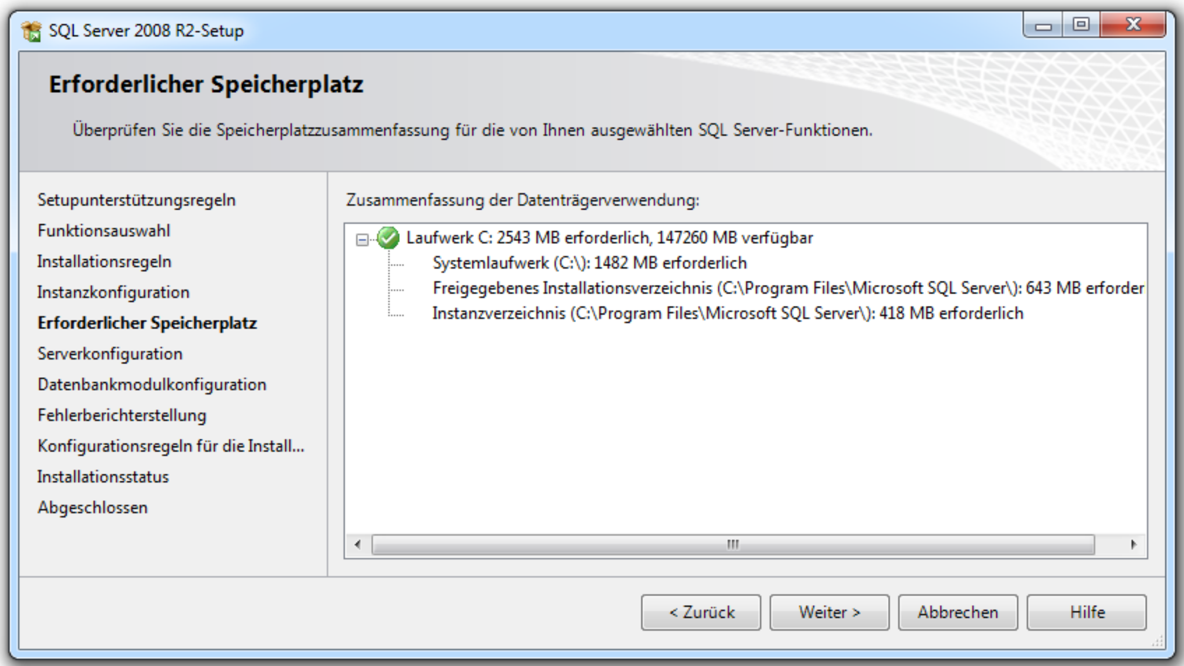
Wir machen **weiter**, und wählen im Dialog **Funktionswahl** alle Optionen:



Im Dialog **Instanzkonfiguration** übernehmen wir die Voreinstellungen:



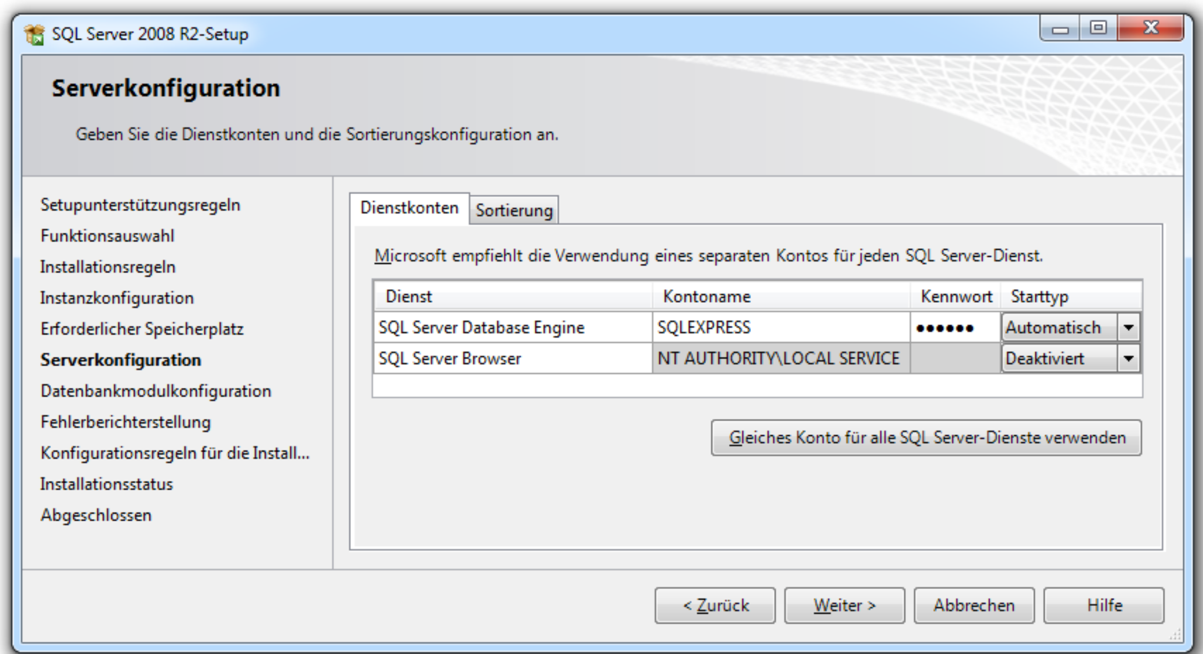
Der SQL Server 2008 R2 Express macht sich auf der Festplatte ganz schön breit:



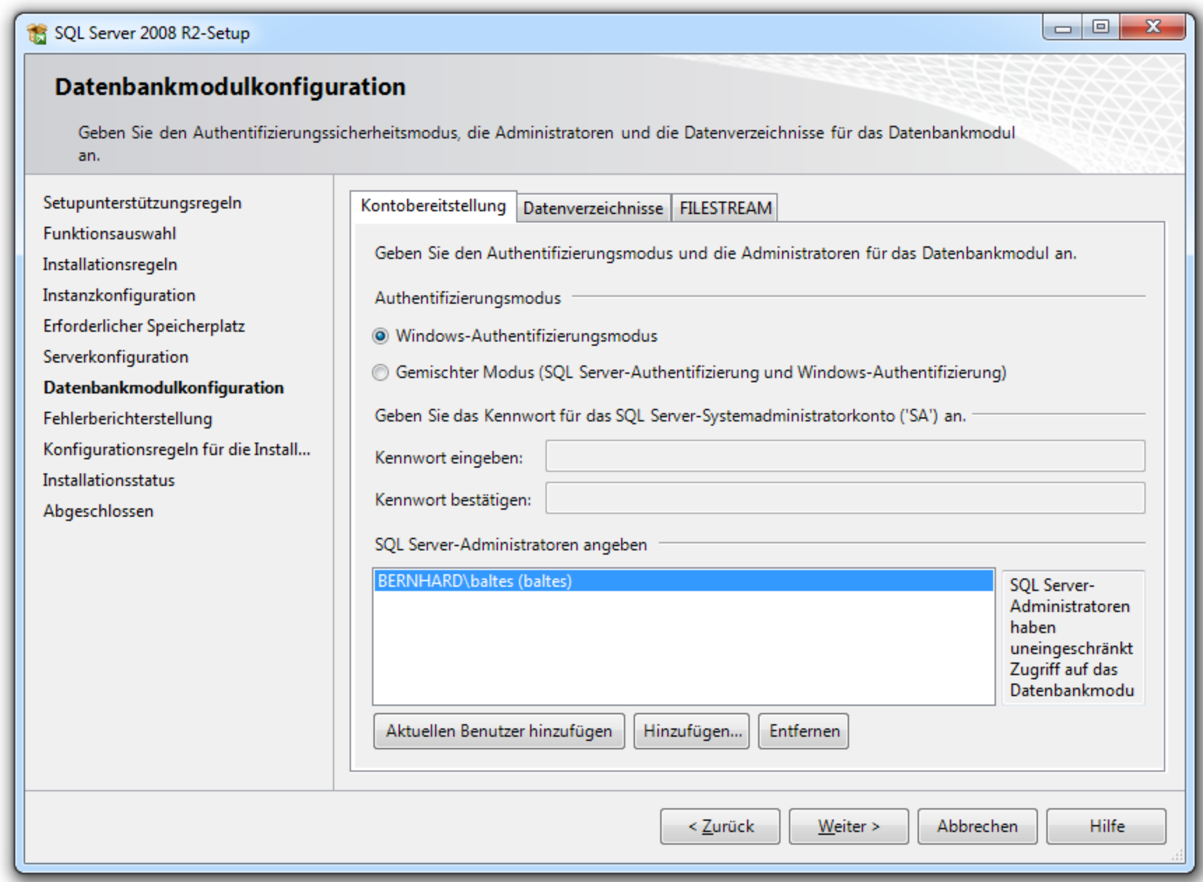
Im Dialog **Serverkonfiguration** legt man den **Kontonamen** und den **Starttyp** für die **SQL Server Database Engine** fest. Der Datenbankdienst sollte unter einem Konto ausgeführt werden, das nur die benötigten Rechte besitzt. Diesbezügliche Empfehlungen von Microsoft sind auf der folgenden Webseite zu finden:

http://msdn.microsoft.com/de-de/library/ms143504%28SQL.105%29.aspx

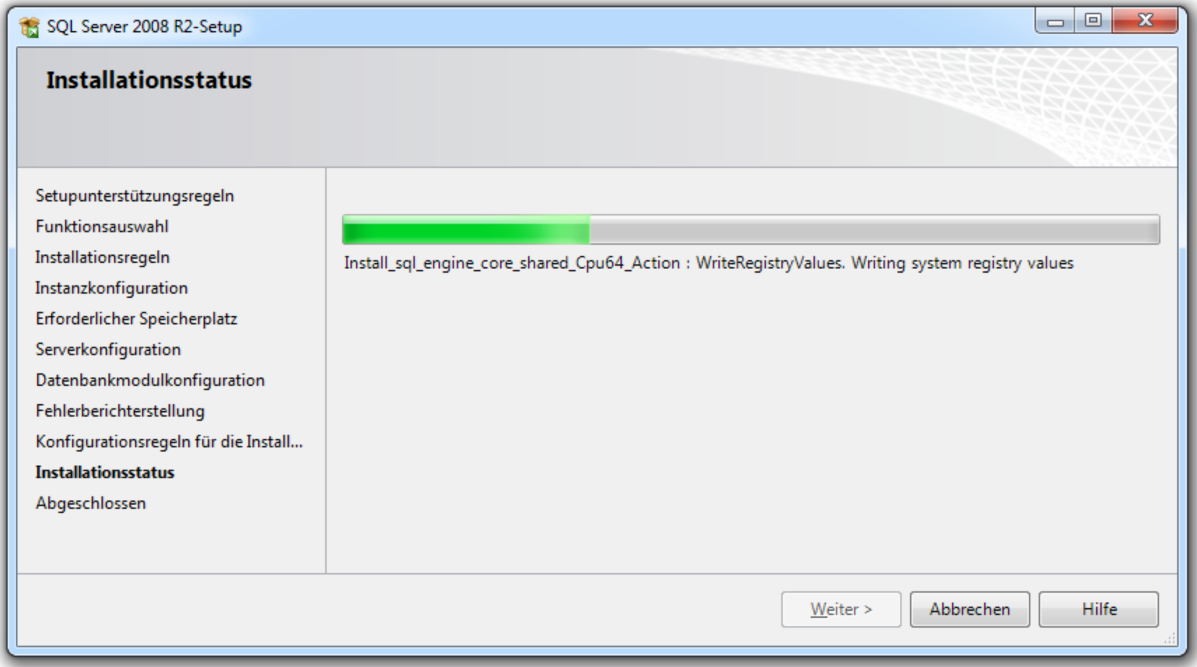
Hier wird ein lokales, zuvor angelegtes Konto mit dem Namen `SQLEXPRESS` verwendet:



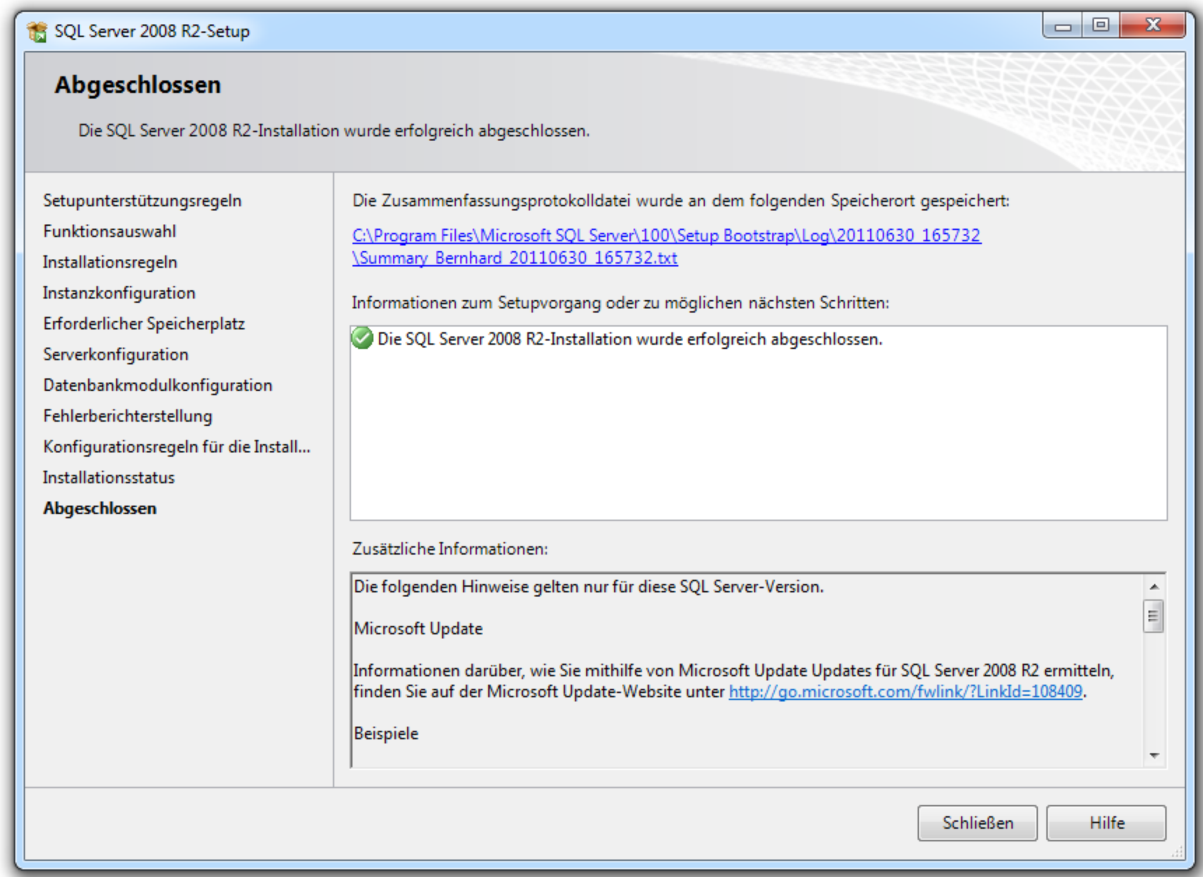
Im Dialog **Datenbankmodulkonfiguration** ist der **Authentifizierungsmodus** zu wählen und (mindestens) ein **Server-Administrator** einzutragen:



Wir verzichten auf die **Fehlerberichterstellung** und beobachten geduldig (viele Minuten lang) die Fortschritte beim **Installationsstatus**:



Was lange währt, wird endlich gut:



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)

Folglich ist der SQL-Server auch ohne expliziten Start durch einen Benutzer dienstbereit, zunächst aber nur für lokal angemeldete Benutzer, weil per Voreinstellung keine Netzzugriffe erlaubt (siehe Abschnitt 20.4.3.2).

20.4.2.2 Beispieldatenbank Northwind

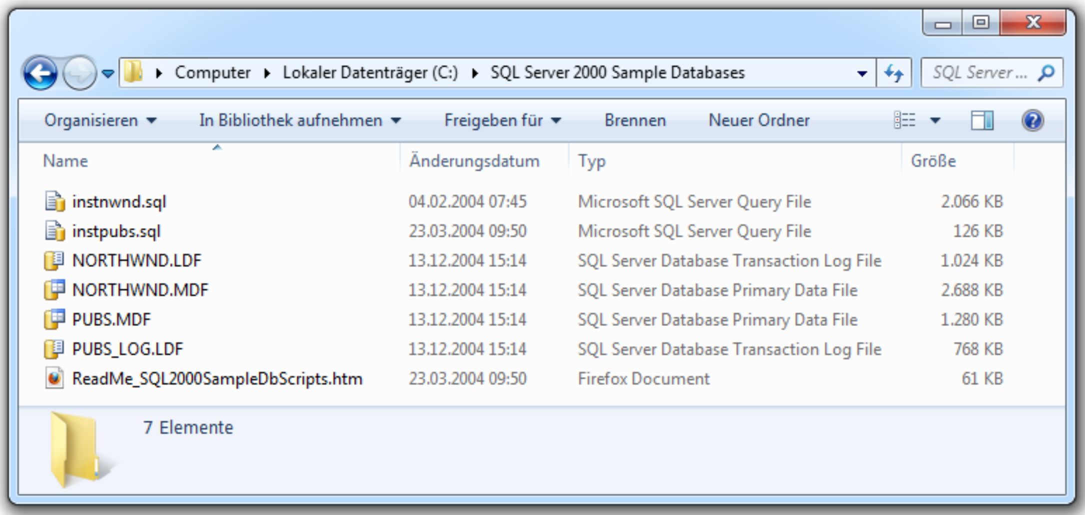
Für seine SQL-Server mit Client-Server - Fähigkeit (das sind alle Varianten mit Ausnahme der Compact Edition) bietet Microsoft die Beispieldatenbanken **Northwind** und **Pubs** auf der Webseite

http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=23654

als Installationspaket **SQL2000SampleDb.msi** an. Bei der (z.B. im Windows-Explorer per Doppelklick auf das MSI-Paket gestarteten) Installation landen die Datenbanken im neu angelegten Ordner

C:\SQL Server 2000 Sample Databases

Neben der eigentlichen Datenbankdatei (Namenserweiterung: **mdf**) gehört zu einer MS-SQL - Datenbank noch eine Log-Datei (Namenserweiterung: **ldf**). Die **Northwind**-Datenbank besteht aus den Dateien **NORTHWND.MDF** und **NORTHWND.LDF**:



Wir verwenden ausschließlich die **Northwind**-Datenbank, wobei aber auch die Compact Edition des SQL-Servers zum Einsatz kommen soll. Eine angepasste Version der Datenbank wird freundlicherweise zusammen mit Compact Edition ausgeliefert und ist nach einer Standardinstallation unter Windows 7 (64 Bit) hier zu finden:

C:\Program Files (x86)\Microsoft SQL Server Compact Edition\v3.5\Samples\Northwind.sdf

Offenbar verwenden die beiden SQL-Server verschiedene Dateiformate, was in unterschiedlichen Dateinamensendungen zum Ausdruck kommt.

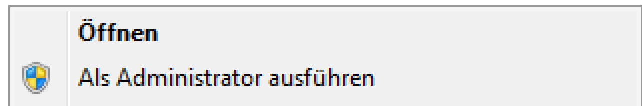
Für einen Datenbankzugriff durch die Compact Edition des SQL-Servers benötigt man lediglich die Datenbankdatei. Ähnlich verwaltungsarm geht es bei einer sogenannten *Benutzerinstanz* der Express Edition zu (siehe Abschnitt 20.5.2.1.1).

Bevor eine Datenbank im *Client-Server - Modus* angesprochen werden kann, muss sie dem SQL Server bekannt gemacht werden, was auf unterschiedliche Weise geschehen kann (siehe Readme-Datei zu den Beispieldatenbanken **Northwind** und **Pubs**). Wir verwenden das eben zusammen mit der SQL Server 2008 R2 Express Edition installierte **SQL Server Management Studio** (siehe Abschnitt 20.4.2.1), um die Beispieldatenbank **Northwind** in die Verwaltung des SQL Servers zu integrieren:

- Starten Sie das Verwaltungswerkzeug über den Link

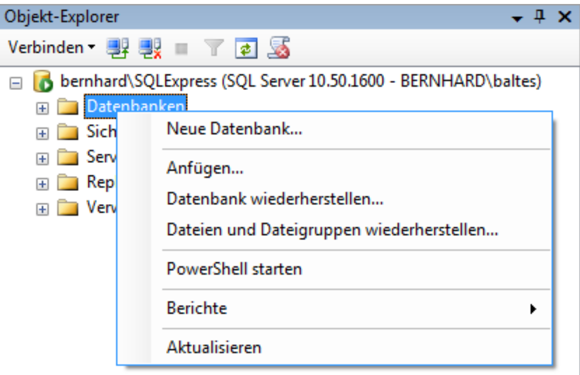
  SQL Server Management Studio

  in der Programmgruppe **Microsoft SQL Server 2008** mit Windows-Administratorrechten. Auf meinem Rechner unter Windows 7-64 hat jedenfalls das Hinzufügen der Datenbank **Northwind** nur geklappt, nachdem ich das Management Studio als Administrator gestartet hatte (über das entsprechende Item im Kontextmenü zum Startlink):

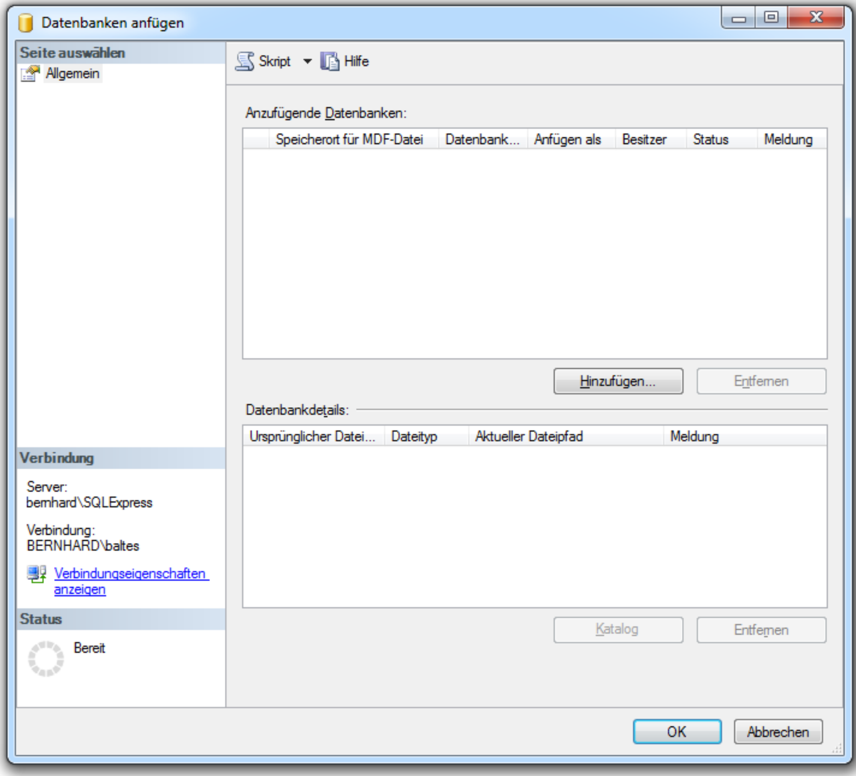
  

- Stellen Sie im folgenden Dialog mit dem Konto eines SQL-Server-Administrators (siehe Datenbankmodulkonfiguration in Abschnitt 20.4.2.1) die Verbindung zum Server her:

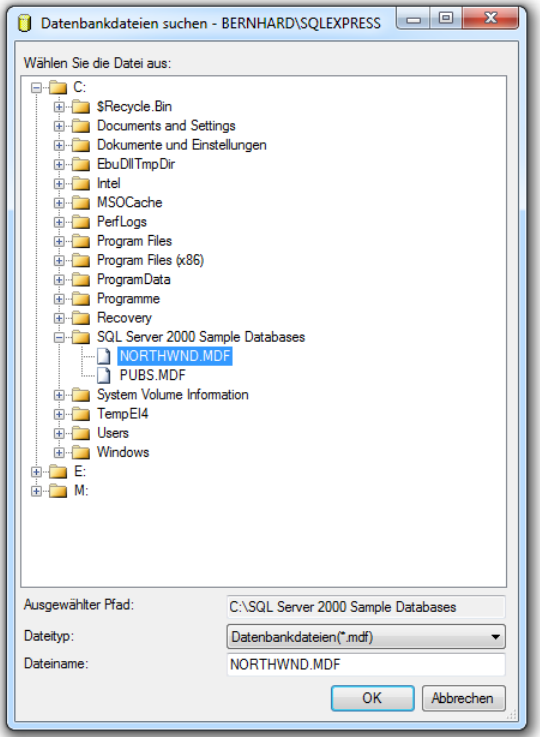
- Öffnen Sie im **Objekt-Explorer** das Kontextmenü zum Eintrag **Datenbanken**, und wählen Sie das Menüitem **Anfügen**.



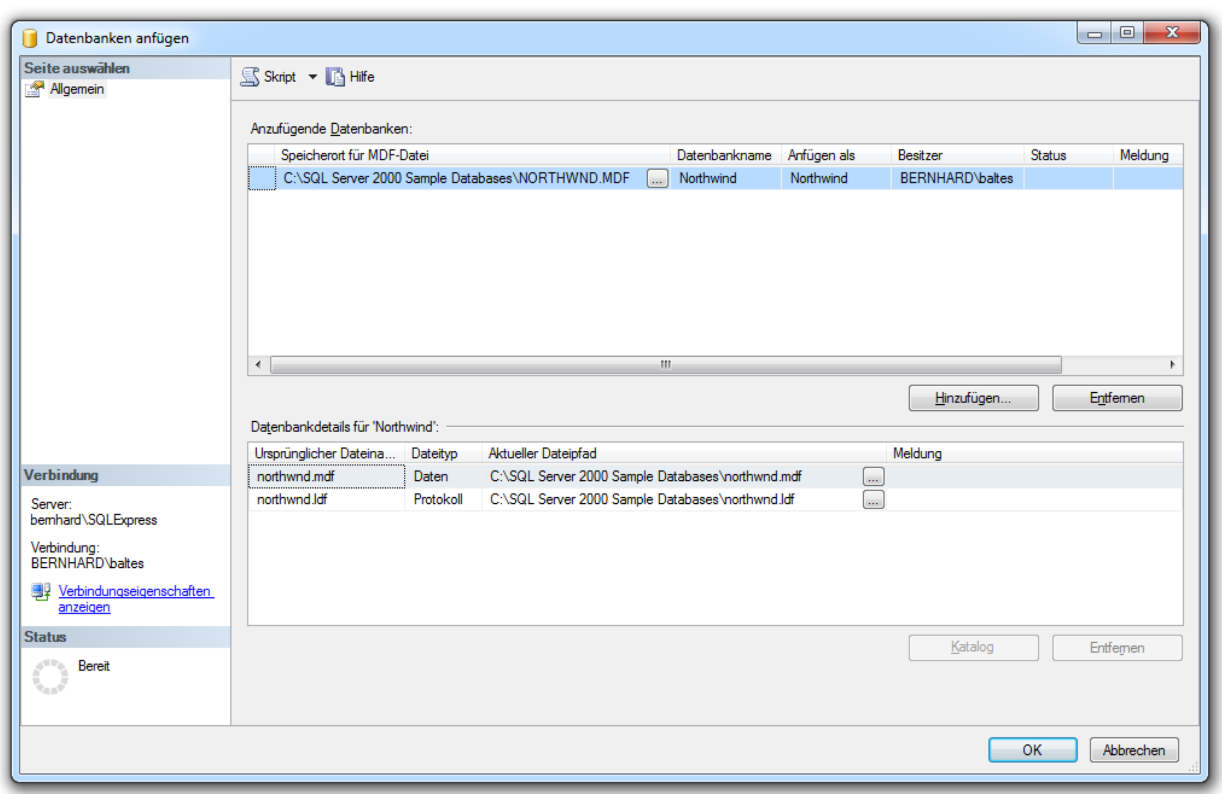
- Es erscheint das Fenster **Datenbanken anfügen**:



- Klicken Sie auf **Hinzufügen**, wählen Sie die **mdf**-Datei zur **Northwind**-Datenbank,

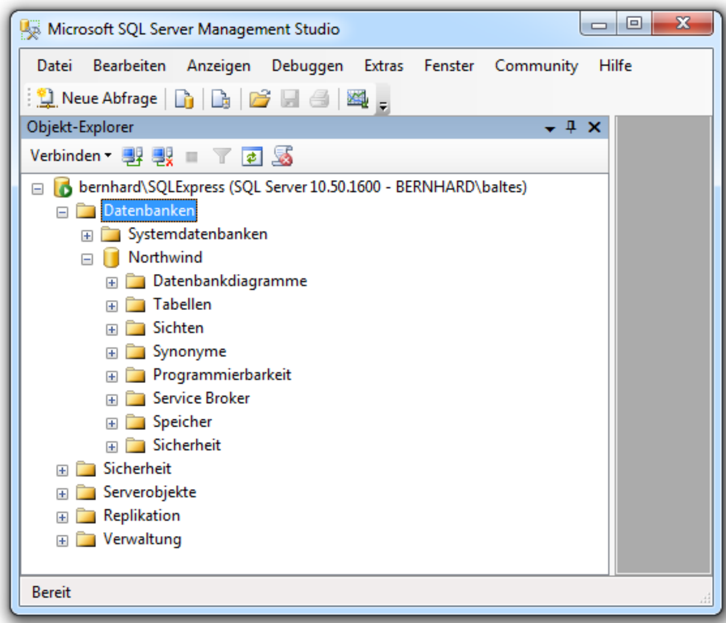


und quittieren Sie im Fenster **Datenbanken anfügen**



mit **OK**.

Anschließend finden Sie die Beispieldatenbank im SQL Server Management Studio:



Der *logische Datenbankname* (hier: **Northwind**) wird per Voreinstellung aus der **mdf**-Datei übernommen und kann bei Bedarf im Management Studio geändert werden.

20.4.3 Konfiguration

20.4.3.1 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 eindeutig die Nutzung der Windows-Konten. Jedoch unterstützt der SQL-Server auch interne Konten, die er unabhängig vom Betriebssystem oder einer Domänenautorität selbst verwaltet.
- **Autorisierung**
  Welche Rechte haben die einzelnen Konten?

Bei der SQL Server *Compact Edition* ist keine Rechteverwaltung erforderlich.

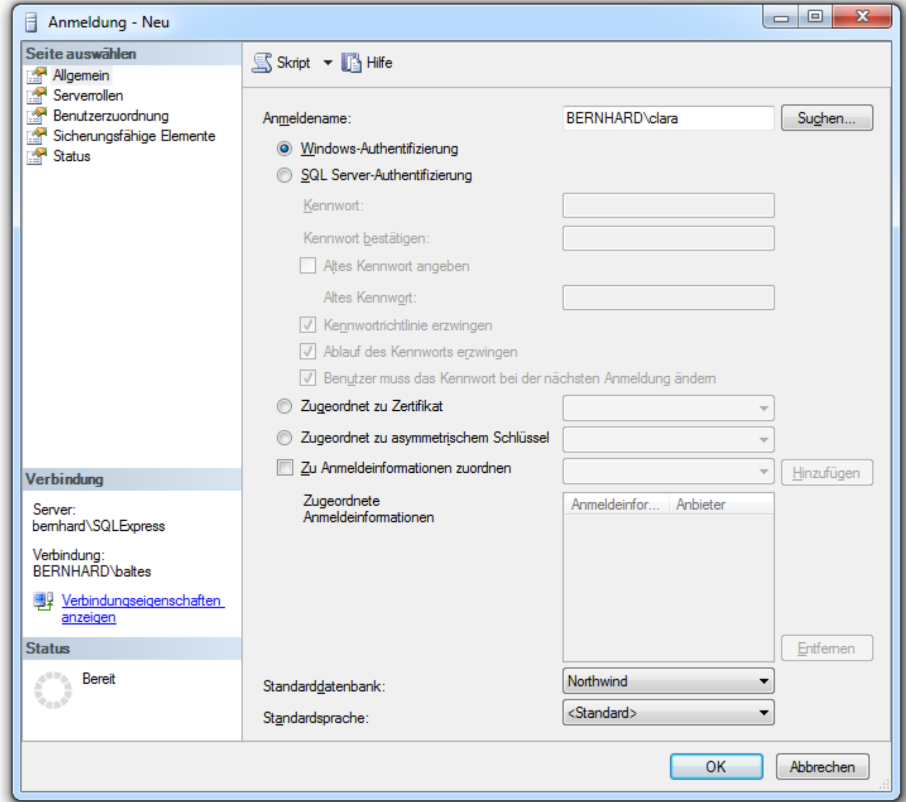
Die SQL Server 2008 R2 *Express Edition* bietet die Option, für den angemeldeten Windows-Benutzer eine so genannte *Benutzerinstanz* des SQL Servers dynamisch zu erzeugen und eine private Datenbank dynamisch einzubinden (vgl. Abschnitt 20.5.2.1). Bei dieser Einsatzart ist *keine* Authentifizierung des Benutzers erforderlich.

20.4.3.1.1 Anmeldung beim SQL-Server für ein Windows-Konto erlauben

Gehen Sie folgendermaßen vor, um einem Windows-Konto den Zugriff auf den SQL Server zu erlauben:

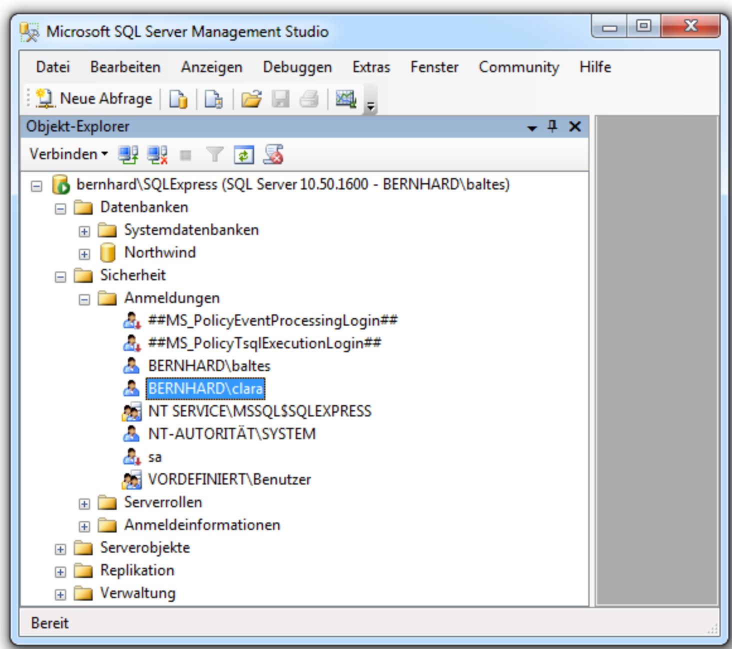
- Starten Sie das **SQL Server Management Studio** mit Windows-Administratorrechten, und stellen Sie die Verbindung zum SQL Server mit dem Konto eines SQL-Server-Administrators her.
- Öffnen Sie im Objekt-Explorer aus dem Kontextmenü zum Knoten **Sicherheit > Anmeldungen** das Item **Neue Anmeldung**. 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:



- Wählen Sie für den Benutzer eine **Standarddatenbank**.

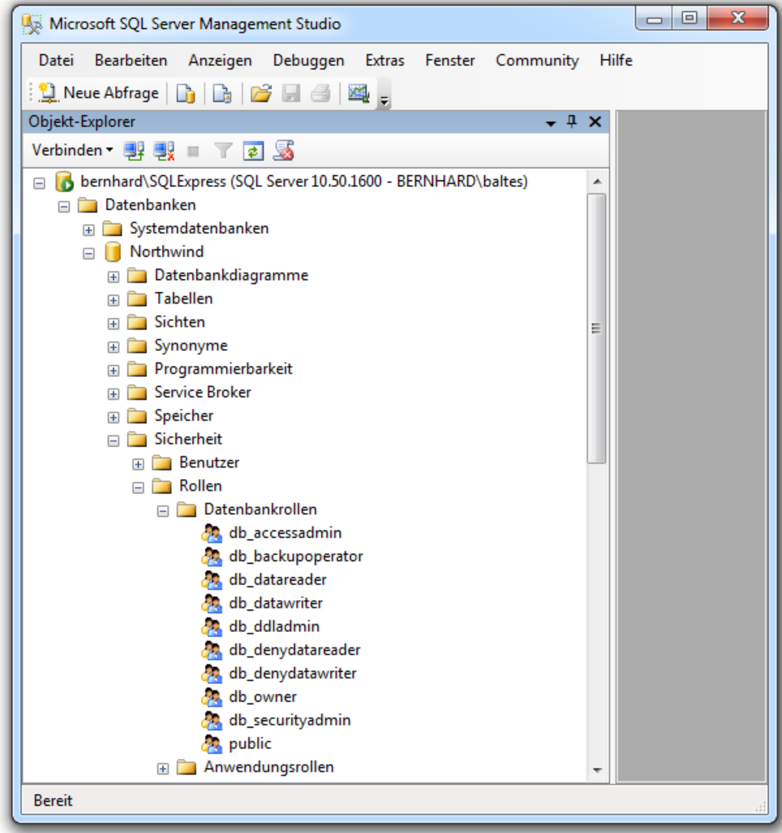
Der neue Benutzer wird im Ordner **Sicherheit > Anmeldungen** des Objekt-Explorers aufgelistet:



20.4.3.1.2 Autorisierung durch Rollen

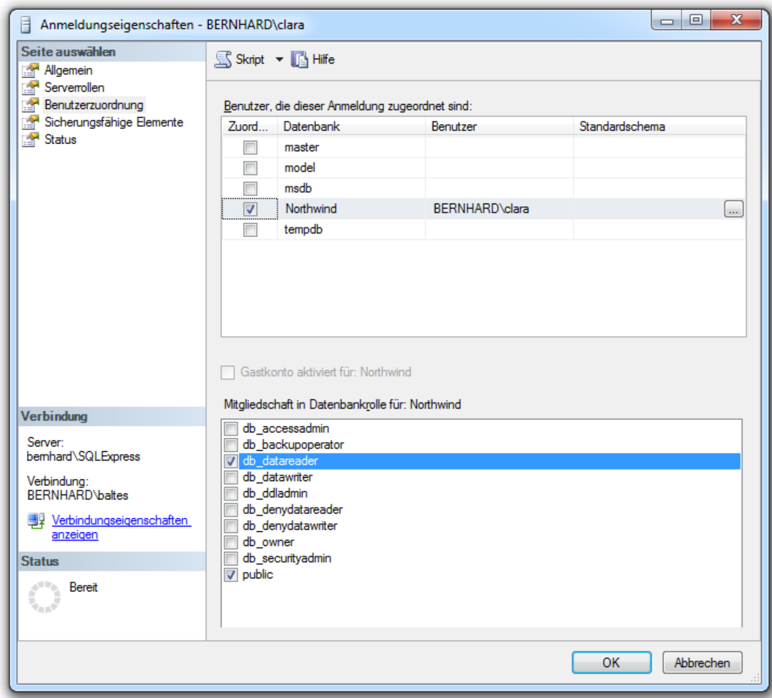
Zur Verwaltung von Benutzerrechten verwendet der SQL Server **Rollen**. Es gibt …

- **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**. Zur Rolle **sysadmin** gehören z.B. die Administratoren des SQL Servers.
- **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 oben eingetragenen Benutzer zu autorisieren, öffnen wir im **Objekt-Explorer** des Management Studios über **Sicherheit > Anmeldungen** sein Kontextmenü und starten mit der Option **Eigenschaften** den Dialog **Anmeldungseigenschaften**. Wir verzichten auf die Vergabe von **Serverrollen**, öffnen die Seite **Benutzerzuordnung** und vergeben für die Datenbank **Northwind** die Rolle **db_datareader**:



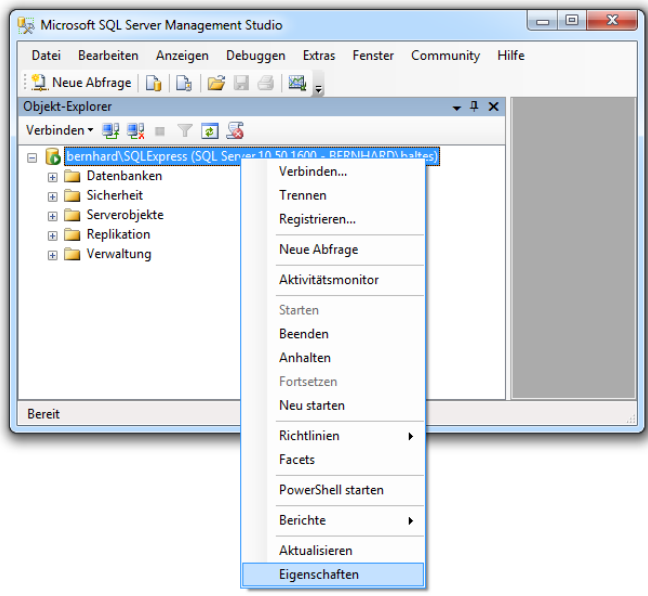
Damit kann der Benutzer nur lesend auf die Datenbank zugreifen. Mit der Rolle **public**, die einem Benutzer nicht entzogen werden kann, sind keine nennenswerten Rechte verbunden.

20.4.3.1.3 SQL Server - Authentifizierung erlauben

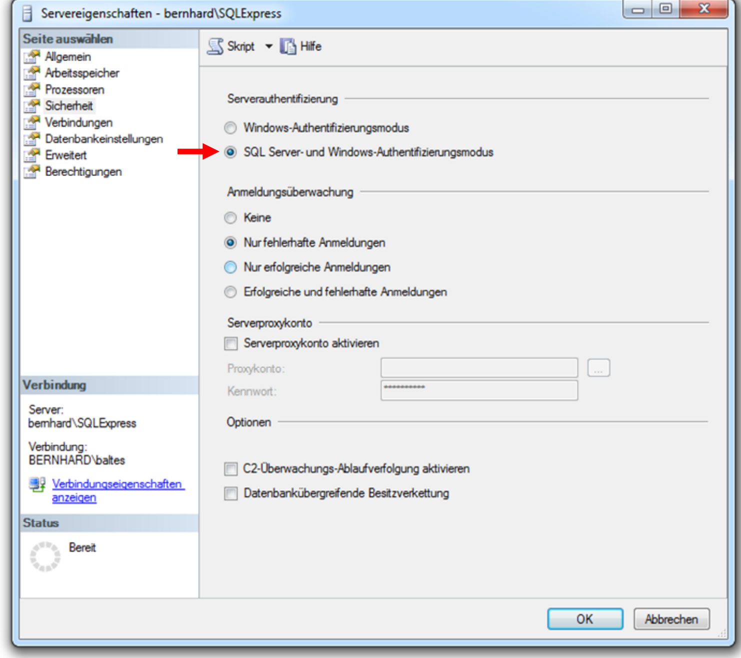
In einer Netzwerkumgebung *ohne* Windows - Domänencontroller ist man bei der Windows - Authentifizierung auf lokale Benutzerkonten (auf dem Rechner mit dem SQL Server) beschränkt. Damit ein Benutzer von einem anderen Rechner aus via Netzwerk auf den SQL Server zugreifen kann, muss auf dem Klienten-PC 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 (siehe Abschnitt 20.4.2.1) unterstützt der SQL Server 2008 R2 Express nur die Windows - Authentifizierung. Im Management Studio lässt sich jedoch die SQL-Server - Authentifizierung nachträglich zuschalten:

- Wählen Sie im 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**,



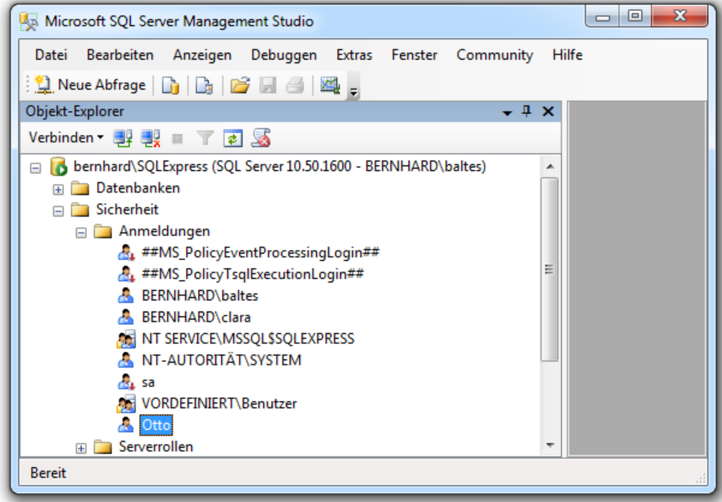
und quittieren Sie mit **OK**.

- Anschließend muss der SQL-Server neu gestartet werden (siehe Abschnitt 20.4.3.2).

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, entfernen Sie Markierung beim Kontrollkästchen **Kennwortrichtlinie erzwingen** und wählen Sie eine **Standarddatenbank**.

Der neue Benutzer erscheint im Ordner **Sicherheit > Anmeldungen** des **Objekt-Explorers**, z.B.:



Bei der Autorisierung über Rollenvergabe gibt es keine Unterschiede zwischen SQL-Server - und Windows - Konten.

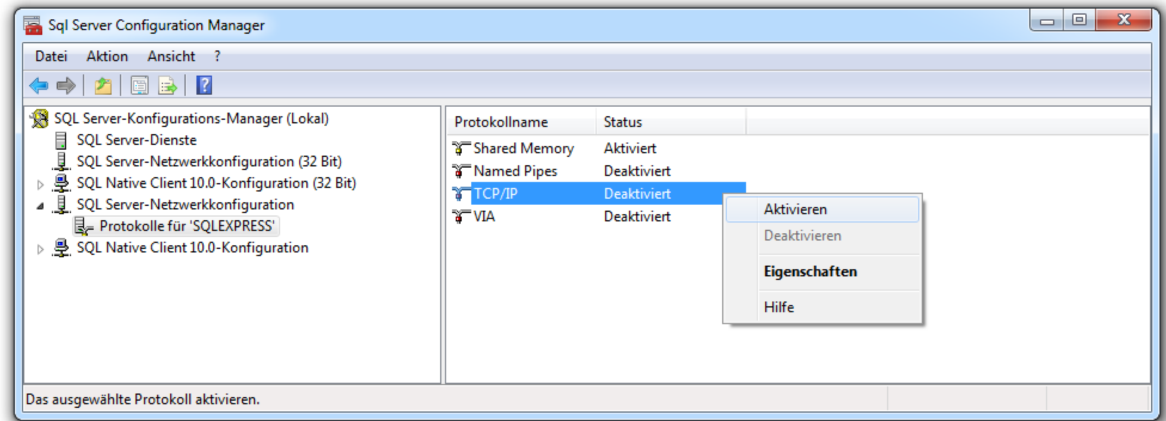
20.4.3.2 Netzwerkzugriff via TCP/IP erlauben

Per Voreinstellung steht der SQL-Server nur lokal angemeldeten Benutzern zur Verfügung, was für die Beispielprogramme in Kapitel 20 meist genügt. Trotzdem soll an dieser Stelle erläutert werden, wie man den Netzwerkzugriff via TCP/IP (vgl. Kapitel 14) auf den SQL-Server erlaubt. Dazu sind drei Maßnahmen erforderlich:

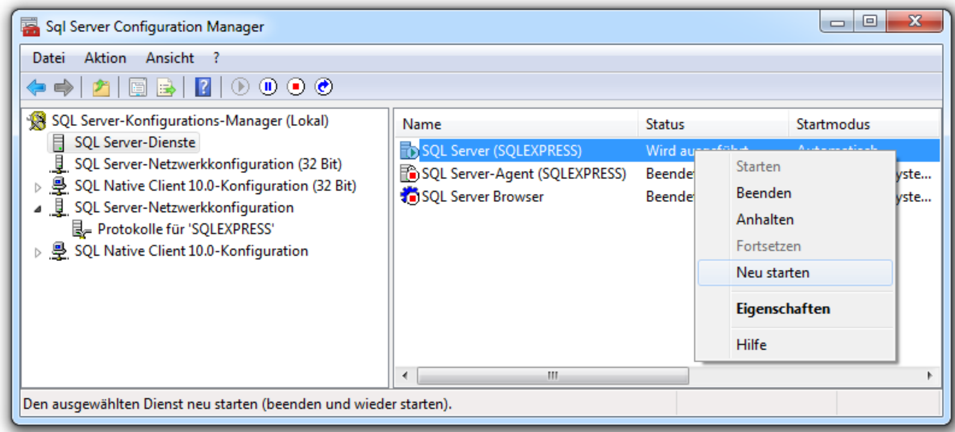
a) Netzwerkprotokoll TCP/IP aktivieren

Arbeitsschritte:

- Starten Sie den **SQL Server Configuration Manager** über seinen Link im Unterordner **Konfigurationstools** der Programmgruppe zum SQL-Server 2008.
- 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 den Knoten **SQL Server-Dienste**, und starten Sie per Kontextmenü den Windows-Dienst **SQL Server (SQLEXPRESS)** neu:

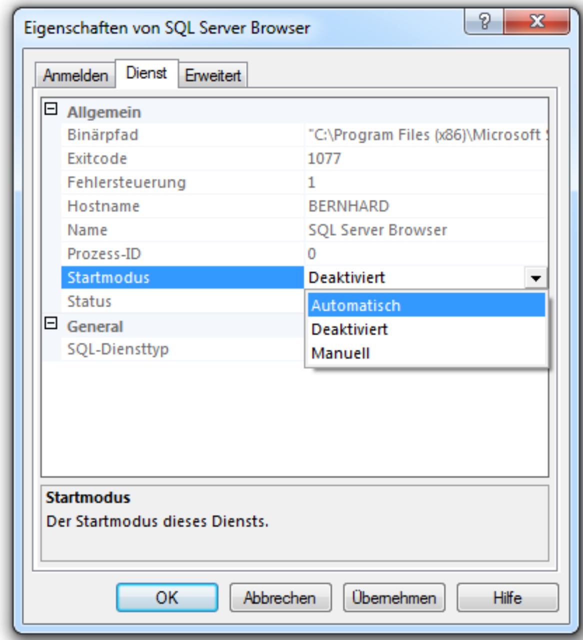


b) SQL Server Browser Dienst starten

Wenn der SQL-Server ohne Angabe einer Portnummer in der Verbindungszeichenfolge (vgl. Abschnitt 20.5.2) über Netz ansprechbar sein soll, 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 SQL-Server 2008.

- 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:



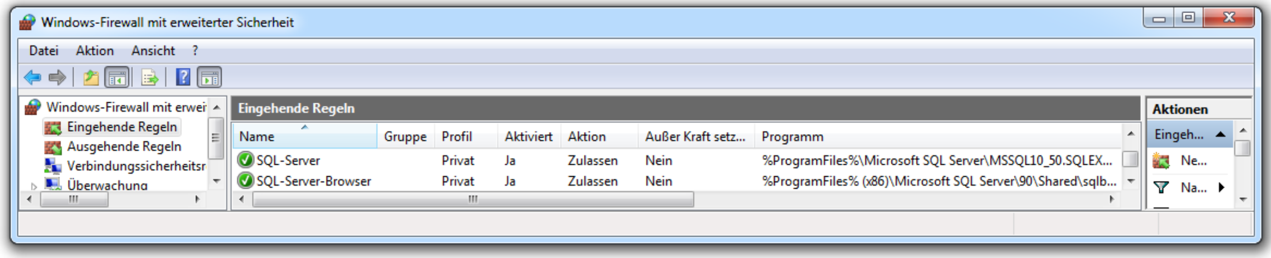
Der Browser-Dienst hört den UDP-Port 1434 ab.

c) Firewall-Ausnahmen definieren

Für den SQL-Server - Dienst und für den SQL-Server-Browser - Dienst muss jeweils eine Firewall-Ausnahme eingetragen werden, was z.B. unter Windows 7 so geschehen kann:

- Starten Sie das Systemsteuerungs-Applet zur Verwaltung der Windows-Firewall über **Start > Systemsteuerung > Windows-Firewall**, klicken Sie auf den Link **Erweiterte Einstellungen**.
- Klicken Sie im Dialog **Windows-Firewall mit erweiterter Sicherheit** auf **Neue Regel**.
- Wählen Sie im Fenster des **Assistenten für neue eingehende Regeln** den Regeltyp **Programm**.
- Machen Sie **weiter**, und wählen Sie über den Schalter **Durchsuchen** das für den SQL-Server - Dienst zuständige Programm, z.B.:
  C:\Program Files\Microsoft SQL Server\MSSQL10_50.SQLEXPRESS\MSSQL\Binn\sqlservr.exe
- Machen Sie **weiter**, und entscheiden Sie im nächsten Dialog, ob Sie beliebige oder nur geschützte Verbindungen zulassen wollen.
- Machen Sie **weiter**, und regeln Sie im nächsten Dialog den Gültigkeitsbereich der neuen Regel (Firmennetzwerk, Privatnetzwerk, öffentliches Netzwerk).
- Machen Sie **weiter**, und vereinbaren Sie einen Namen für die neue Regel.
- Erstellen Sie analog eine Ausnahme für das Programm
  C:\Program Files (x86)\Microsoft SQL Server\90\Shared\sqlbrowser.exe
  zum SQL-Server-Browser - Dienst.

Hier sind die neuen Regeln im Dialog **Windows-Firewall mit erweiterter Sicherheit** zu sehen:



20.5 ADO.NET

20.5.1 Überblick

20.5.1.1 Verbindungsloses versus verbindungsorientiertes Arbeiten

Moderne Softwarearchitekturen bevorzugen beim Datenbankzugriff das *verbindungslose* Arbeiten, wobei die relevanten Daten vom DBMS bezogen, in einem lokalen **DataSet**-Objekt gespeichert, bearbeitet und ggf. anschließend wieder zur Datenbank zurück übertragen werden.

Beim Durchsuchen sehr großer Datenbestände ist es eventuell weniger sinnvoll, mit einer lokalen Kopie zu arbeiten. Für solche Anwendungen bieten die ADO.NET – Provider (siehe Abschnitt 20.5.1.2) 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.

20.5.1.2 Provider

Für den Datenbankzugriff wird ein so genannter *Provider* benötigt, worunter eine Sammlung von .NET - Klassen zu verstehen ist, die gemeinsam den Zugriff auf eine bestimmte Datenbanktechnologie unterstützen. Das .NET-Framework enthält Provider für die folgenden Datenbankmanagementsysteme bzw. Datenbankschnittstellen:

- **SqlClient**
  Dieser Provider unterstützt SQL-Server und ist speziell optimiert für Microsofts SQL Server (inkl. Express Edition).
- **SqlServerCe**
  Dieser Provider unterstützt die Compact Edition von Microsofts SQL Server.
- **OracleClient**
  Dieser Provider unterstützt die Datenbanktechnik der Firma Oracle. Weil Microsoft die Weiterentwicklung eingestellt hat,[111] sollte man wohl besser auf den von Oracle kostenlos angebotenen Provider setzten.[112]
- **OleDb**
  Es können alle Datenbanken mit OLEDB-Schnittstelle angesprochen werden, wobei aber die Performanz unter der implizit erforderlichen NET-COM - Interoperabilität leidet.
- **Odbc**
  Für diesen Vorgänger der OLEDB-Technik gelten die dortigen Bemerkungen analog.

Weitere Provider existieren z.B. für das Datenbankmanagementsystem DB2 der Firma IBM sowie für die frei verfügbaren Lösungen MySQL und PostgreSQL.

[111] Siehe Webseite: http://blogs.msdn.com/b/adonet/archive/2009/06/15/system-data-oracleclient-update.aspx
[112] Siehe Webseite: http://www.oracle.com/technology/tech/windows/odpnet/index.html

Zum Provider **SqlClient**, den wir diesem Kurs ausschließlich einsetzen werden, gehören die folgenden Klassen aus dem Namensraum **System.Data.SqlClient**:

- **SqlConnection**
  Erlaubt das Konfigurieren und Öffnen einer Verbindung zum SQL Server (siehe Abschnitt 20.5.2)
- **SqlCommand**
  Repräsentiert ein SQL-Kommando (siehe Abschnitt 20.5.3)
- **SqlDataAdapter**
  Diese Klasse vermittelt zwischen der Datenbankverbindung und dem lokalen **DataSet**-Objekt (siehe Abschnitt 20.5.4).
- **SqlDataReader**
  Erlaubt ein lesendes, sequentielles Durchlaufen der angeschlossenen Datenbank (siehe Abschnitt 20.5.8)
- **SqlParameter**
  Diese Klasse repräsentiert Parameter zur Übergabe an eine gespeicherte Prozedur im DBMS und wird im Manuskript nicht behandelt.
- **SqlTransaction**
  Diese Klasse stellt eine SQL-Transaktion dar, die vom DBMS vorgenommen werden soll, und wird im Manuskript nicht behandelt.

Weil die Klassen eines Providers jeweils eine bestimmte Schnittstelle zu implementieren haben (z.B. **IDbConnection** bei **SqlConnection**), ist der Wechsel zu einem anderen Provider kein Problem und erfordert nur unwesentliche Änderungen im Quellcode.

20.5.1.3 ADO.NET - Namensräume

Anhand der ADO.NET - Namensräume kann man die Funktionalität des Frameworks gut überblicken. Besonders zu erwähnen sind:

- **System.Data**
  In diesem Namensraum befinden sich provider*un*abhängige Klassen, meist beschäftigt mit der klientenseitigen, verbindungslosen Verarbeitung von Daten:
  - **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.
  - **DataColumn, DataColumnCollection**
    Ein **DataColumn**-Objekt repräsentiert eine Tabellenspalte und enthält z.B. in seiner **DataType**-Eigenschaft den Typ der enthaltenen Daten.
  - **DataRow, DataRowCollection**
    Ein **DataRow**-Objekt repräsentiert eine Tabellenzeile.
  - **DataRelation, DataRelationCollection**
    Ein **DataRelation**-Objekt repräsentiert die Beziehung zwischen zwei **DataTable**-Objekten (vgl. Abschnitt 20.2.2).
  - **Constraint, ConstraintCollection**
    Ein **Constraint**-Objekt repräsentiert eine Regel zur Sicherung der Datenintegrität. Spezialisierte Unterklassen sind: **ForeignKeyConstraint** und **UniqueConstraint**.

  In diesem Namensraum sind keine Klassen für die Verbindung zu einem DBMS vorhanden.

- **System.Data.SqlClient**, **System.Data.SqlServerCe**, **System.Data.OracleClient**, **System.Data.OleDb**, **System.Data.Odbc**
  Die Klassen der Provider für bestimmte Datenbanktechnologien (siehe oben) befinden sich jeweils in einem eigenen Unternamensraum.

In einem .NET - Programm mit Datenbankzugriff sollte man den Namensraum **System.Data** sowie den Namensraum zum eingesetzten Provider importieren.

20.5.2 Die Connection-Klassen

Für die Verbindung zu einer Datenbank auf einem Datenbankserver ist beim Provider **SqlClient** die Klasse **SqlConnection** aus dem Namensraum **System.Data.SqlClient** zuständig. Bei einer eingebetteten Lösung mit der Microsoft SQL-Server Compact Edition (Provider **SqlServerCe**) kommt die Klasse **SqlCeConnection** aus dem Namensraum **System.Data.SqlServerCe** zum Einsatz. Andere Provider bieten analog benannte Klassen.

Wir beschränken uns anschließend auf die Verbindung zu einem SQL-Server der Firma Microsoft, wobei die Express Edition im Mittelpunkt steht, aber auch die Compact Edition behandelt wird.

20.5.2.1 Verbindungszeichenfolgen für Microsofts SQL-Server

Um die Verbindung zu einer bestimmten Datenbank herstellen zu können, benötigt ein **Connection**-Objekt etliche Informationen, die als so genannte **Verbindungszeichenfolge** dem Konstruktor übergeben oder der Eigenschaft **ConnectionString** zugewiesen werden können, z.B.:

```
SqlConnection dbConnection = new SqlConnection();
dbConnection.ConnectionString = @"Data Source=.\SQLEXPRESS;"+
  @"AttachDbFilename=E:\Data\NORTHWND.MDF;"+
  "Integrated Security=true;User Instance=true";
```

Die Verbindungszeichenfolge besteht aus „Parameter=Wert“ – Paaren, die jeweils durch ein Semikolon getrennt werden. In Abhängigkeit vom Provider und sonstigen Bedingungen kommen zahlreiche Varianten in Frage.[113]

20.5.2.1.1 Client-Server - fähige Versionen (inkl. Express Edition)

20.5.2.1.1.1 Parameter

Bei einem **SqlConnection**-Objekt zur Verbindung mit einem SQL-Server enthält der **ConnectionString** in der Regel folgende Parameter:

a) SQL-Server

Zur Bezeichnung des Parameters kann man beliebig 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**
  Verwenden Sie für den Wert die Syntax
  Servername\Instanzname

---

[113] Es sind sogar spezialisierte Internet-Angebote mit Vorschlägen zur Bildung von korrekten Verbindungszeichenfolgen entstanden, z.B. http://www.connectionstrings.com/

Den lokalen Rechner spricht man mit einem Punkt, mit **localhost** oder mit **(lokal)** an, wobei die FCL-Dokumentation zur letztgenannten Variante rät. Über den Instanznamen kann man mehrere Installationen des SQL-Servers auf demselben Rechner unterscheiden (vgl. Misner 2007, S. 185). Die SQL Server 2008 R2 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**
  Ist der SQL-Server einem anderen Rechner über das TCP/IP - Protokoll (an einem bestimmten Port) erreichbar (vgl. Abschnitt 20.4.3.2 zur Freigabe des Zugriffs), eignet sich diese Syntax:

  Server=tcp:*Servername**Instanzname*[, *portnumber*]

  Die Portnummer darf fehlen, wenn auf dem Server der SQL Browser Dienst läuft und den UDP-Port 1434 abhört (siehe Abschnitt 20.4.3.2).

  Beispiel:

  ```
  Server=tcp:VMWin764\SQLEXPRESS
  ```

b) Datenbank

Die zu verwendende Datenbank kann über ihren logischen Namen (beim SQL-Server) oder über ihren Dateinamen (bei Benutzerinstanzen der SQL Server Express Edition, siehe Punkt d) angesprochen werden:

- **Logischer Datenbankname**
  In der Regel wird eine Datenbank über ihren logischen Namen angesprochen, unter dem sie beim SQL-Server geführt wird. Dieser kann sich vom Dateinamen unterscheiden. Zur Bezeichnung des Parameters kann man beliebig zwischen den folgenden synonymen Schlüsselwörtern wählen:

  Database, **Initial Catalog**

  Beispiel:

  ```
  Database=Northwind
  ```

- **AttachDbFilename (bei Benutzerinstanzen)**
  Bei dynamisch erzeugten SQL Server - Benutzerinstanzen (siehe unten) gibt man den **mdf**-Dateinamen als Wert zum Parameter **AttachDbFilename** an, z.B.:

  ```
  AttachDbFilename=|DataDirectory|\NORTHWND.MDF
  ```

  Die Datenbankdatei kann über das mit Makro **DataDirectory** *relativ* adressiert werden. Das Makro zeigt bei einer Windows-Anwendung per Voreinstellung auf den Ordner mit dem Exe-Assembly. Mit dem folgenden Methodenaufruf kann ein alternatives Verzeichnis eingestellt werden:

  ```
  AppDomain.CurrentDomain.SetData("DataDirectory",@"E:\Data");
  ```

  Dabei wird eine Eigenschaft der so genannten *Anwendungsdomäne* geändert. Mit diesem wichtigen Begriff der .NET - Technologie haben wir uns leider aus Zeitgründen nicht beschäftigen können (siehe z.B. Richter 2006, S. 533ff).

  Es ist zu beachten, dass Microsofts SQL-Server aus Gründen der Performanz und Datenintegrität nur Datenbankdateien auf einem lokalen Laufwerk oder auf einer Netzwerk-Ressource mit SAN-Technik (*Storage Area Network*) unterstützen. Datenbankdateien auf einer gewöhnlichen Netzfreigabe (z.B. auf einem NAS-Server (*Network Attached Storage*)) werden also *nicht* akzeptiert.[114]

[114] Diese Regel mit dem gelegentlich empfohlenen Service-Startparameter -T1807 außer Kraft zu setzen, ist mir nicht gelungen.

c) Benutzer - Authentifizierung

Wie Sie schon aus Abschnitt 20.4.3.1 wissen, unterstützen Microsofts SQL Server 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**, z.B.:

```
Integrated Security=true
```

Bei der SQL Server - Authentifizierung sind die Schlüsselwörter **User** (alias **User ID**) und **Password** zu verwenden, z.B.:

```
User ID=Otto;Password=xxx
```

d) Benutzerinstanzen der Microsoft SQL Server 2008 R2 Express Edition

Die SQL Server 2008 *Express Edition* bietet die Option, für den angemeldeten Windows-Benutzer eine so genannte *Benutzerinstanz* des SQL Servers dynamisch zu erzeugen und eine Datenbankdatei dynamisch einzubinden. Der ansonsten nur mit Windows-Standardrechten ausgestattete Benutzer kann als Besitzer der Datenbankdatei agieren, sofern er Lese- und Schreibrechte für die beiden Dateien zur Datenbank (Namenserweiterung **mdf** und **ldf**) besitzt. Weder muss die Datenbank beim SQL-Server angemeldet werden (vgl. Abschnitt 20.4.2.2), noch ist eine Konfiguration von Zugriffsrechten im Sinn von Abschnitt 20.4.3.1 vorzunehmen. Folglich ist eine Anwendung mit so implementiertem Datenbankzugriff leicht auf einen anderen Rechner zu übertragen. Nötigenfalls liefert man die SQL Server 2008 Express Edition gleich mit, was Microsoft explizit erlaubt. Um mit einer Benutzerinstanz zu arbeiten, setzt man den Parameter **User Instance** auf den Wert **true**:

```
User Instance=true
```

und wählt über den Parameter **AttachDbFilename** eine Datenbankdatei (siehe oben). Außerdem ist die Windows - Authentifizierung anzugeben:

```
Integrated Security=true
```

Beim Einsatz einer Benutzerinstanz werden die zur Verwaltung des SQL - Servers nötigen Dateien im Windows-Profilordner des angemeldeten Benutzers angelegt, z.B. unter Windows 7 in

C:\Users\baltes\AppData\Local\Microsoft\Microsoft SQL Server Data\SQLEXPRESS

Benutzerinstanzen der SQL Server 2008 R2 Express Edition eignen sich gut für Datenbanken, die eng mit einer Anwendung verbunden und nur von *einer* Person benutzt werden sollen. Das Anwendungsprogramm, der SQL Server und die Datenbankdatei befinden sich auf dem Rechner, an dem der Benutzer angemeldet ist. In dieser Lage sollte allerdings geprüft werden, ob als DBMS nicht auch die sehr schlanke *SQL Server 2008 Compact Edition* ausreicht (Plattenspeicherbedarf: nur 2 MB).

e) Zugriffsoptionen

Über den Parameter **Connect Timeout** legt man fest, wie lange auf eine erfolgreiche Verbindung zum SQL Server gewartete werden soll (Voreinstellung: 15 Sekunden), z.B.:

```
Connection Timeout=3
```

Beim Datenbankzugriff über Netzwerk und großem Transportvolumen kann es sich lohnen, über den Parameter **Packet Size** die Größe der Netzwerkpakete zu erhöhen (Voreinstellung: 8192 Bytes). Über weitere Parameter informiert z.B. die FCL-Dokumentation zur **ConnectionString**-Eigenschaft der Klasse **SqlConnection**.

20.5.2.1.1.2 Weitere Hinweise

Weitere Hinweise zur Verbindungszeichenfolge:

- Die Groß-/Kleinschreibung ist irrelevant.
- Der **ConnectionString** darf im Programm nur bei geschlossener Datenbankverbindung geändert werden, was bei verbindungsloser Arbeitsweise meist der Fall sein sollte.
- Ist zur Authentifizierung ein SQL-Benutzername samt Passwort erforderlich, sollten diese Daten *nicht* im Quellcode erscheinen, sondern beim Benutzer abgefragt werden.

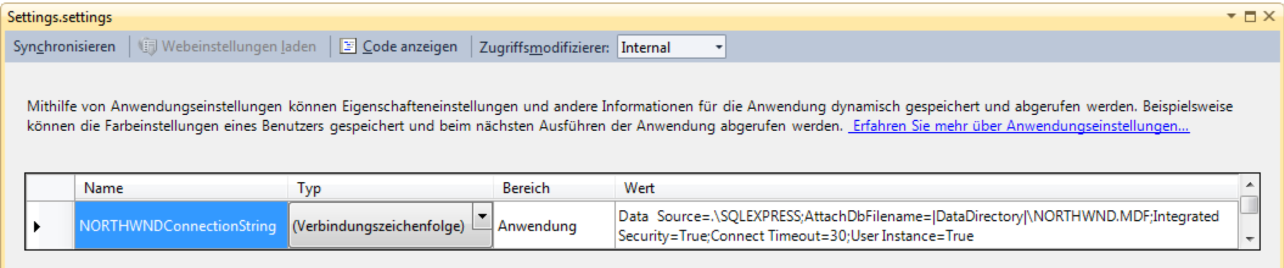
Am Anfang des Abschnitts 20.5.2.1 war eine komplette Verbindungszeichenfolge für den Zugriff auf eine lokale SQL Server - Benutzerinstanz zu sehen. Hier folgt nun ein Beispiel für den Netzwerkzugriff auf einen SQL Server bei Verwendung der Windows - Authentifizierung:

```
dbConnection.ConnectionString = @"Server=tcp:VMWin764\SQLEXPRESS;"+
   "Database=Northwind;Integrated Security=true";
```

Soll die Datenbankverbindung einer Anwendung flexibel konfigurierbar sein, eignet sich der Quellcode nicht zur Ablage der Verbindungszeichenfolge. Man bringt sie besser in einer .NET - Konfigurationsdatei unter (vgl. Abschnitt 17.1). Dieses Exemplar

```
app.config
 version="1.0" encoding="utf-8" ?>
iguration>
configSections>
/configSections>
connectionStrings>
  <add name="Datenquelle.Properties.Settings.NORTHWNDConnectionString"
      connectionString="Data Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\NORTHWND.MDF;Integrated Security=True;Connect Timeout=30;User Instance=True"
      providerName="System.Data.SqlClient" />
/connectionStrings>
figuration>
100 %
```

wurde von der Visual Studio 2010 Express Edition zu dem im Abschnitt 20.6.2 vorzustellenden Projekt erstellt. Die Verbindungszeichenfolge landet im **<connectionStrings>**-Element der Konfigurationsdatei und kann bequem über das Einstellungsformular



modifiziert werden, das z.B. per Doppelklick auf den Eintrag **Properties > Settings.settings** im Projektmappen-Explorer erreichbar ist. Die Entwicklungsumgebung verwaltet im Projektordner die Konfigurationsdatei **app.config** und überträgt ihren Inhalt bei jedem Erstellen des Projekts (ohne Nachfrage überschreibend) in die eigentliche Anwendungskonfigurationsdatei, die sich die im Ordner mit dem Exe-Assembly befindet. Bei einem Projekt mit dem Namen **Datenquelle** hat die Anwendungskonfigurationsdatei den Namen **Datenquelle.exe.config**. Für den Programmstart im Rahmen der Entwicklungsumgebung findet sich im selben Ordner außerdem die Datei **Datenquelle.vshost.exe.config**, die ebenfalls bei jedem Erstellen des Projekts durch die Datei **app.config** überschrieben wird.

Auf Techniken zur *verschlüsselten* Ablage von Verbindungszeichenfolge können wir hier aus Zeitgründen nicht eingehen.

20.5.2.1.2 Compact Edition

Bei der Microsoft SQL Server 2008 Compact Edition kann die Verbindungszeichenfolge besonders einfach ausfallen und sich z.B. auf die Angabe des Datenbankdateinamens beschränken:

```
dbConnection.ConnectionString = @"Data Source=E:\Data\Northwind.sdf";
```

20.5.2.2 Eigenschaften, Ereignisse und Methoden

Die Eigenschaften der **Connection**-Klassen dienen teilweise (je nach ADO.NET - Provider) dazu, einen bequemen Lesezugriff auf Parameter der Verbindungszeichenfolge zu bieten. Bei der Klasse **SqlConnection** können auf diese Weise z.B. **DataSource**, **Database** und **ConnectionTimeout** abgefragt werden. Über die Eigenschaft **State** vom Typ **ConnectionState**, die nahe liegender Weise ebenfalls nur einen lesenden Zugriff erlaubt, ist von einem **SqlConnection**-Objekt u.a. zu erfahren, ob die Verbindung zur Datenquelle momentan offen oder geschlossen ist.

Über die Ereignisse **InfoMessage** bzw. **StateChanged** informiert ein **Connection**-Objekt über Probleme mit dem SQL Server bzw. Änderungen im Verbindungsstatus.

Wenn Sie eine Datenbankverbindung explizit öffnen (per **Open**()-Aufruf), müssen Sie diese Verbindung möglichst frühzeitig wieder schließen (per **Close**()-Aufruf). Beim expliziten Einsatz einer **DataReader**-Klasse (z.B. **SqlDatatReader**, vgl. Abschnitt 20.5.8) ist das explizite Öffnen unumgänglich. Beim verbindungslosen Arbeiten mit der Klasse **SqlDataAdapter** (siehe Abschnitt 20.5.4) ist das explizite Öffnen und Schließen der Datenbankverbindung hingegen überflüssig. So ist z.B. bei den folgenden Anweisungen zum Füllen eines **DataSet**-Objekts

```
dbAdapter = new SqlDataAdapter();
dbAdapter.SelectCommand = selCommand;
ds = new DataSet();
try {
   dbConnection.Open();
   dbAdapter.Fill(ds);
} finally {
   dbConnection.Close();
}
```

etlicher Aufwand überflüssig. Denselben Effekt erreicht man auch so:

```
dbAdapter = new SqlDataAdapter();
dbAdapter.SelectCommand = selCommand;
ds = new DataSet();
dbAdapter.Fill(ds);
```

Bei einer *Sequenz* von Methodenaufrufen mit Verbindungsautomatismus kann es aber performanter sein, die Datenbankverbindung vorher explizit zu öffnen und so ein automatisches zwischenzeitliches Schließen zu vermeiden.

20.5.3 Die Command-Klassen

Für SQL-Anweisungen an ein RDBMS (und auch für die im Kurs nicht behandelten gespeicherten Prozeduren) ist beim ADO.NET - Provider **SqlClient** die Klasse **SqlCommand** aus dem Namensraum **System.Data.SqlClient** zuständig. Andere Provider bieten analog benannte Klassen.

20.5.3.1 Eigenschaften und Methoden

Die wichtigsten Eigenschaften eines **SqlCommand**-Objekts:

- **Connection**
  Diese Eigenschaft ermittelt oder setzt das **SqlConnection**-Objekt, das die Verbindung zu einer Datenbank auf einem Datenbankserver darstellt.
- **CommandText**
  Diese Eigenschaft ermittelt oder setzt die SQL-Anweisung (oder die gespeicherte Prozedur), die ausgeführt werden soll.
- **Parameters**
  Diese Eigenschaft zeigt auf eine Kollektion mit dem Datentyp **SqlParameterCollection** und wird in Abschnitt 20.5.3.2 über parametrisierte Abfragen behandelt.

Beispiel:

```
SqlConnection dbConnection = new SqlConnection();
SqlCommand selCommand;
  . . .
dbConnection.ConnectionString = @"Data Source=(local)\SQLEXPRESS;" +
  @"AttachDbFilename=E:\Data\NORTHWND.MDF;" +
  "Integrated Security=True;User Instance=True";
selCommand=new SqlCommand("SELECT EmployeeID,FirstName,LastName FROM Employees",
                          dbConnection);
```

Command-Objekte tun meist als Helfer eines **DataAdapter**-Objekts (siehe Abschnitt 20.5.4) ihren Dienst, doch bieten sie auch Methoden zur direkten Ausführung des enthaltenen Kommandos:

- **ExecuteReader()**
  Diese Methode sendet den **CommandText** an die **Connection** und liefert als Rückgabewert einen **DataReader**, der beim ausschließlich lesenden Zugriff auf eine große Datenbank als Alternative zur verbindungslosen Arbeitsweise (unter Verwendung einer lokalen Kopie) in Frage kommt (siehe Abschnitt 20.5.8).
- **ExecuteNonQuery()**
  Mit dieser Methode lässt man DDL-Befehle (wie **CREATE TABLE**) oder DML-Befehle (**UPDATE**, **INSERT**, **DELETE**) ausführen.
- **ExecuteScalar()**
  Diese Methode eignet sich zur Ausführung von **SELECT**-Kommandos, die einen einzelnen Wert zurückliefern, z.B. den mittleren Preis aller Produkte:

  ```
  SELECT AVG(UnitPrice) FROM Products
  ```

  Von einem „breiteren“ oder „längeren“ Abfrageergebnis liefert **ExecuteScalar()** nur die erste Spalte der ersten Zeile. Weil der Rückgabewert von Typ **Object** ist, muss er in der Regel einer expliziten Typanpassung unterworfen werden.
  Beispiel:

  ```
  SqlCommand selComm=new SqlCommand("SELECT AVG(UnitPrice) FROM Products",
                                    dbConnection);
  dbConnection.Open();
  Console.WriteLine(Convert.ToDouble(selComm.ExecuteScalar()));
  dbConnection.Close();
  ```

20.5.3.2 Parametrisierte Abfragen

Wenn sich bei wiederholt benötigten Abfragen nur einzelne Bestandteile ändern, definierte man Parameter und ändert deren Werte, statt das komplette Kommando jeweils neu zu erzeugen. Im folgenden Programm

```
using System;
using System.Data;
using System.Data.SqlClient;

class CommandDemo {
   SqlConnection dbConnection = new SqlConnection();
   SqlCommand selAvgSupp;

   CommandDemo() {
      dbConnection.ConnectionString = @"Data Source=.\SQLEXPRESS; " +
         "AttachDbFilename=E:\Data\NORTHWND.MDF;" +
         "Integrated Security=True;User Instance=True";
      selAvgSupp = new SqlCommand(
         "SELECT AVG(UnitPrice) FROM Products WHERE SupplierID = @SuppID",
         dbConnection);
      selAvgSupp.Parameters.Add("@SuppID", SqlDbType.Int);
   }

void AvgPriceSupplier() {
   Console.Write("Nummer das Anbieters: ");
   int supp = Convert.ToInt32(Console.ReadLine());
   selAvgSupp.Parameters["@SuppID"].Value = supp;
   try {
      dbConnection.Open();
      Console.WriteLine("Mittlerer Preis aller Produkte von Anbieter " + supp +
                        ": " + Convert.ToDouble(selAvgSupp.ExecuteScalar()));
   } finally {
      dbConnection.Close();
   }
}
   static void Main() {
      CommandDemo cd = new CommandDemo();
      cd.AvgPriceSupplier();
   }
}
```

entscheidet der Anwender, für welchen Lieferanten der Firma **Northwind** der mittlere Produktpreis ermittelt werden soll. Im **CommandText** steht an Stelle einer festen Liefentennummer der Parameter `@SuppID`:

```
selAvgSupp = new SqlCommand(
   "SELECT AVG(UnitPrice) FROM Products WHERE SupplierID = @SuppID",
   dbConnection);
```

Das **SqlCommand**-Objekt wird über den Parameter informiert, wobei Name und Datentyp anzugeben sind:

```
selAvgSupp.Parameters.Add("@SuppID", 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.

Bei der Syntax für parametrisierte Abfragen unterscheiden sich die Provider in ADO.NET (siehe z.B. die FCL-Dokumentation zur **Parameters**-Eigenschaft der Klasse **OleDbCommand**).

20.5.4 Die DataAdapter-Klassen

20.5.4.1 Zuständigkeiten

Beim verbindungslosen Arbeiten spielt ein **DataAdapter**-Objekt die zentrale Rolle:

- Es führt mit Hilfe von **Command**-Objekten **SELECT**- und DML-Abfragen durch und vermittelt dabei zwischen den lokal (in einem **DataSet**-Objekt) zwischengespeicherten Daten und der Datenbank.
- Es öffnet bei Bedarf automatisch eine Verbindung zum Datenbankserver und schließt sie ebenso 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.

Für die Verbindung zu den beteiligten **Command**-Objekten sorgen die folgenden **DataAdapter**-Eigenschaften

- **SelectCommand**
- **UpdateCommand**
- **InsertCommand**
- **DeleteCommand**

20.5.4.2 Datentransfer von der Datenbank zum DataSet-Objekt

Beim Aufruf der **DataAdapter**-Methode **Fill()**, die das zu füllende **DataSet**-Objekt 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
```

Die Microsoft SQL Server Compact Edition kann solche Kommandoserien übrigens *nicht* verarbeiten.

Die im folgenden Beispiel verwendete **Fill()** - Überladung erwartet eine Referenz auf das zu füllenden **DataSet**-Objekt und den gewünschten programminternen Tabellennamen:

```
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 gewinnen.

Sollte (wie im obigen Beispiel) das beteiligte **SelectCommand**-Objekt *mehrere* Tabellen liefern, würden diese die Namen `Employees`, `Employees1`, etc. erhalten. Ein **DataAdapter** bietet über seine Eigenschaft **TableMappings** eine Möglichkeit zur individuellen Benennung *mehrerer* **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);
```

die Tabellennamen `Employees` und `Customers`.

Ist der Primärschlüssel einer Tabelle bekannt (z.B. aufgrund eines **FillSchema()**-Aufrufs, siehe unten), dann überschreibt die **Fill()**-Methode ggf. in der Tabelle 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");
```

20.5.4.3 Datentransfer vom DataSet-Objekt zur Datenbank

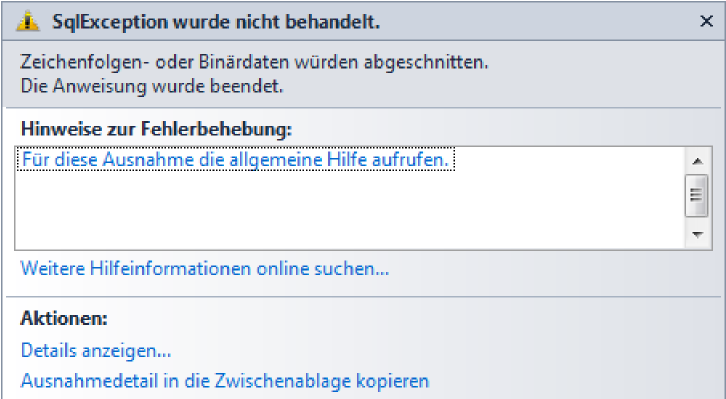
Beim Aufruf der **DataAdapter**-Methode **Update()** werden die DML-Befehle in **UpdateCommand**, **InsertCommand** und **DeleteCommand** ausgeführt, um die am lokalen **DataSet**-Objekt vorgenommenen Änderungen zur Datenbank zu übertragen, z.B.:

```
dbAdapter.Update(ds);
```

In Abschnitt 20.5.6 ist zu erfahren, wie die DML-Befehle per **CommandBuilder** entstehen, und was bei **DataSet**-Objekten mit mehreren Tabellen zu beachten ist.

20.5.4.4 Schematransfer von der Datenbank zum DataSet-Objekt

Die in einer Datenbank definierten Restriktionen (z.B. Länge einer Zeichenfolge, **Null**-Verbot) und Primärschlüsseldefinitionen werden durch einen **Fill()**-Aufruf *nicht* zum lokalen **DataSet**-Objekt übertragen, was bei einem ausschließlich *lesenden* Datenbankzugriff auch nicht erforderlich ist. Wird ein **DataTable**-Objekt allerdings inkompatibel verändert und anschließend via **Update()** zur Datenbank übertragen, kommt es zu einem Laufzeitfehler, z.B.:



Um das Schema der Datenbank mit den Gültigkeitsregeln und Primärschlüsseldefinitionen zu den Tabellen in ein lokales **DataSet** zu übertragen, kann man die **DataAdapter**-Methode **FillSchema()** verwenden. Im folgenden Code-Segment wird ein einzelnes **DataTable**-Objekt per **Fill()**-Aufruf angelegt, gefüllt und benannt. Im anschließenden **FillSchema()**-Aufruf wird das **DataTable**-Objekt mit Schemainformationen versorgt:

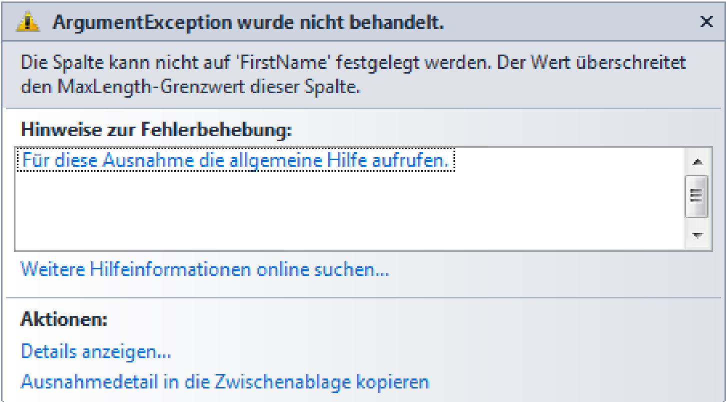
```
SqlConnection dbConnection = new SqlConnection();
SqlDataAdapter dbAdapter = new SqlDataAdapter();
DataSet ds = new DataSet();
DataTable dt;
  . . .
dbAdapter.SelectCommand =
  new SqlCommand("SELECT EmployeeID,FirstName,LastName FROM Employees",dbConnection);
  . . .
```

```
dbAdapter.Fill(ds, "Employees");
dt = ds.Tables["Employees"];
dbAdapter.FillSchema(dt, SchemaType.Source);
```

Im Beispiel benötigt und kennt der **SqlDataAdapter** keine **TableMappings**-Einträge, so dass der zweite **FillSchema**()-Parameter (vom Datentyp **SchemaType**) den Wert **SchemaType.Source** erhalten hat. Anderenfalls ist der Wert **SchemaType.Mapped** zu verwenden.

Existiert im **DataSet** noch kein **DataTable**-Objekt mit dem angegebenen Namen, dann legt **FillSchema()** diese Tabelle an. Zur Aufnahme der Schemainformationen werden nötigenfalls in der Tabelle **DataColumn**-Objekte erzeugt und konfiguriert (über die Eigenschaften **MaxLength**, **AllowDBNull** etc.). 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 zeitaufwendige Datenbankverbindung erkannt:



Statt die **DataAdapter**-Methode **FillSchema**() aufzurufen, kann man die **DataAdapter**-Eigenschaft **MissingSchemaAction** auf den Wert **AddWithKey** der Enumeration **MissingSchemaAction** setzen, z.B.:

```
dbAdapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;
```

Daraufhin findet bei jedem zugehörigen **Fill**()-Aufruf ein Abgleich zwischen den Schemainformationen in der Datenbank und im lokalen **DataSet** statt, was bei wiederholten **Fill()**-Aufrufen überflüssigen Aufwand verursacht. Microsoft empfiehlt,[115] …

- die **FillSchema**()-Methode zu verwenden, wenn ein mehrfaches Lesen von Datenbankinhalten durch **Fill**()-Aufrufe zu erwarten ist,
- die **MissingSchemaAction** auf den Wert **AddWithKey** zu setzen, wenn die **Fill**()-Methode nur einmal pro **DataSet** aufgerufen wird.

Erfreulich ist bei den **DataAdapter**-Methoden **Fill**(), **FillSchema**() und **Update**() die Fähigkeit, bei Bedarf eine Datenbankverbindung automatisch zu öffnen und zu schließen (vgl. Abschnitt 20.5.2.2).

20.5.5 DataSet und andere Provider-unabhängige Klassen

In diesem Abschnitt werden wichtige Klassen aus dem Provider-unabhängigen Namensraum **System.Data** behandelt. Sie speichern beim verbindungslosen Arbeiten lokale Kopien der angeforderten Datenbanktabellen samt Restriktionen und Beziehungen.

[115] Siehe Webseite http://support.microsoft.com/?scid=kb%3Ben-us%3B310128&x=10&y=7

20.5.5.1 Abfrageergebnisse und Schema-Informationen aus einer Datenbank übernehmen

Im folgenden Code-Segment werden aus den **Northwind**-Tabellen **Customers** und **Orders** jeweils einige Spalten in ein Zweitabellen - **DataSet**-Objekt übernommen:

```
SqlConnection dbConnection;
SqlCommand selCommandCust, selCommandOrd;
SqlDataAdapter dbAdapterOrd, dbAdapterCust;
DataSet ds = new DataSet();
DataTable dtCustomers, dtOrders;
  . . .
dbConnection = new SqlConnection(@"Data Source=(local)\SQLEXPRESS;" +
  @"AttachDbFilename=E:\Data\NORTHWND.MDF;" +
  "Integrated Security=True;User Instance=True");

selCommandCust = new SqlCommand(
  "SELECT CustomerID, CompanyName, Country FROM Customers;", dbConnection);
selCommandOrd = new SqlCommand(
  "SELECT OrderID, CustomerID, ShipCountry FROM Orders", dbConnection);

dbAdapterCust = new SqlDataAdapter(selCommandCust);
dbAdapterOrd = new SqlDataAdapter(selCommandOrd);

dbAdapterCust.MissingSchemaAction = MissingSchemaAction.AddWithKey;
dbAdapterOrd.MissingSchemaAction = MissingSchemaAction.AddWithKey;

try {
  dbConnection.Open();
  dbAdapterCust.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 einzelnen Tabellen (**DataTable**-Objekten) des **DataSets**. Im Beispielprogramm werden der Bequemlichkeit halber **DataTable**-Referenzvariable zu den beiden Tabellen angelegt.

Im Beispiel kommen *zwei* **SqlDataAdapter**-Objekte zum Einsatz, damit die zu Datenbankänderung erforderlichen Aktionskommandos (**UPDATE**, **INSERT**, **DELETE**) automatisch aus den **SE-LECT**-Kommandos für die beiden beteiligten Datenbanktabellen erzeugt werden können (siehe Abschnitt 20.5.6).

Mit dem folgenden Methodenaufruf

```
Console.WriteLine(" MaxLength(CompanyName)      = " +
  dtCustomers.Columns["CompanyName"].MaxLength +
  "\n Primärschlüssel(Customers) = " + dtCustomers.PrimaryKey[0].ColumnName + "\n");
```

kann man sich davon überzeugen, dass durch die Anweisung

```
dbAdapterCust.MissingSchemaAction = MissingSchemaAction.AddWithKey;
```

wichtige Merkmale der Datenbanktabelle **Customers** in das zugehörige **DataTable**-Objekt 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 Code-Segment 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:

```
Constraint1   System.Data.UniqueConstraint
  Spalte zur UniqueConstraint: CustomerID
```

Leider kann ADO.NET die Beziehungsstruktur einer Datenbank *nicht* (z.B. per **FillSchema**()) automatisch übernehmen, so dass **ForeignKeyConstraint**-Objekte (z.B. zum **DataTable**-Objekt `Orders`) erst erscheinen, nachdem im Programm zum **DataSet** `ds` eine entsprechende **DataRelation** angelegt worden ist (siehe Abschnitt 20.5.5.3).

Die **Rows**- bzw. **Columns**-Eigenschaft eines **DataTable**-Objekts zeigt auf das **DataRowCollection**- bzw. auf das **DataColumnCollection**-Objekt mit den Zeilen bzw. Spalten der Tabelle. Per **Count**-Eigenschaft lässt sich die jeweilige Anzahl feststellen. Weil die beiden Kollektionen über einen Indexer verfügen, kann man mit einer bequemen Indexsyntax auf die Elemente zugreifen, was z.B. in der folgenden Methode zur Konsolenausgabe von Daten aus der `Orders`-Tabelle geschieht:

```
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 den Spaltennamen (Eigenschaft **ColumnName**).

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**) abgebildet. Auf der Webseite

http://msdn.microsoft.com/de-de/library/cc716729.aspx

findet sich eine umfangreiche Tabelle mit *SQL Server-Datentypmappings (ADO.NET)*.

20.5.5.2 DataTable-Objekte modifizieren

Änderungen an den (lokalen) **DataTable**-Objekten werden später über die **DataAdapter**-Methode **Update()** (siehe Abschnitt 20.5.6) zur Datenbank übertragen.

20.5.5.2.1 Werte in vorhandnen Zeilen ändern

Um einzelne Werte einer **DataTable**-Zeile (eines **DataRow**-Objekts) zu ändern, kann man die zugehörigen Zellen per Indexer-Zugriff ansprechen, wobei die Spalten alternativ über ihren Namen oder einen nullbasierten numerischen Index angesprochen werden können, z.B.:

```
DataRow dr = dtCustomers.Rows[0];
dr["Country"] = "Lummerland";
dr[0] = "TOMSP";
```

Zur Identifikation einer Zeile kann bei einer Tabelle mit 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 leer bleiben oder 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 ansprechen, z.B.:

```
dr.ItemArray = new Object[] {null, "Alfreds Fuderkiste", "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
```

Hatte ein **DataRow**-Objekt den Ausgangszustand **Unchanged** (siehe unten), gelangt es durch eine Änderung von Werten in den Zustand **Modified** und durch einen anschließenden Aufruf der **DataSet**-, **DataTable**- oder **DataRow**-Methode **AcceptChanges()** (siehe unten) wieder zurück in den Zustand **Unchanged**.

20.5.5.2.2 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 ohne Anspruch auf Vollständigkeit wichtige Ereignisse beschrieben:

Ereignis	**Beschreibung**
ColumChanging	Das Ereignis tritt ein, nachdem 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 ggf. über die **DataRow**-Methode **SetColumnError()** einen später auszuwertenden Fehlerindikator (siehe Abschnitt 20.5.5.2.3) setzen.
ColumChanged	Das Ereignis tritt ein, sobald der Wert einer Zelle erfolgreich geändert worden ist.

Ereignis	Beschreibung
RowChanging	Das Ereignis tritt ein, wenn bei einer Zeile ein Zellenwert oder die **RowState**-Eigenschaft geändert werden soll. Einer registrierten Methode wird mit dem Ereignisbeschreibungsobjekt vom Typ **DataRowChangeEventArgs** auch eine Referenz zur betroffenen Zeile übergeben, so dass ggf. über die **DataRow**-Methode **SetColumnError()** ein später auszuwertender Fehlerindikator (siehe Abschnitt 20.5.5.2.3) gesetzt werden kann.
RowChanged	Das Ereignis tritt ein, nachdem bei einer Zeile ein Zellenwert oder die **RowState**-Eigenschaft erfolgreich geändert werden 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 folgenden Ereignismethoden beschränken sich auf diagnostische Ausgaben:

```
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]);
}
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]);
}
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);
}
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);
}
```

Sie 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
RowChanging (ID: ALFKI), Action: Change, RowState: Unchanged
RowChanged (ID: ALFKI), Action: Change, RowState: Modified
ColumnChanging, Spalte: Country
  vorgeschl. Wert: Hummerland, akt. Wert: Germany
ColumnChanged,  Spalte: Country
  vorgeschl. Wert: Hummerland, akt. Wert: Hummerland
RowChanging (ID: ALFKI), Action: Change, RowState: Modified
RowChanged (ID: ALFKI), Action: Change, RowState: Modified
```

Es zeigt sich u.a.:

- Für jede Wertveränderungen erhält man vier Ereignisse in folgender 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**. Im Zusammenhang mit dem **RowChanged**-Ereignis wechselt der Zeilenstatus auf **Modified**. Mit den möglichen **RowState**-Werten einer Zeile werden wir uns in Abschnitt 20.5.5.2.6 noch 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. im anschließend beschriebenen Bearbeitungsmodus für Datenzeilen realisieren lässt.

20.5.5.2.3 Der Bearbeitungsmodus für DataRow-Objekte

Die auf eine Einleitung durch die **DataRow**-Methode **BeginEdit()** folgenden Wertänderungen 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 == false)
   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** den Wert **true** besitzt, z.B. aufgrund eines **SetColumnError()**-Aufrufs in einer Validierungsmethode:

```
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");
}
```

Der Bearbeitungsmodus desaktiviert **RowChanging**- und **RowChanged**-Ereignisse sowie die Kontrolle auf verletzte Eindeutigkeitsrestriktionen bis zum **EndEdit()**-Aufruf. Geänderte Spaltenwerte 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**.

20.5.5.2.4 DataRow-Objekt hinzufügen

Um eine neue Tabellenzeile anzulegen, …

- erstellt man mit der **DataTable**-Methode **NewRow()** eine Zeile mit dem Schema der Tabelle,
- versorgt man die Pflichtfelder (**NULL**-Verbot) mit Werten,
- nimmt man die Zeile 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**, gelangt durch den **Add()**-Aufruf in den Zustand **Added** und durch einen Aufruf der **DataSet**-, **DataTable**- oder **DataRow**-Methode **AcceptChanges()** (siehe unten) in den Zustand **Unchanged**.

Bei einem späteren Datenbank-Update (siehe Abschnitt 20.5.6) erhalten die unversorgten Felder der neuen Zeile den Wert **NULL**, sofern dieser erlaubt ist.

20.5.5.2.5 DataRow-Objekt zum Löschen vormerken und entfernen

Um eine **DataTable**-Zeile im Zustand (d.h. mit dem **RowState**-Wert, siehe unten) **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 beseitigt, sondern durch den **RowState**-Wert **Deleted** (siehe unten) zum Entfernen gekennzeichnet, so dass beim späteren Datenbank-Update die zugehörige Zeile der Datenbanktabelle (falls vorhanden) gelöscht wird. Der Vollständigkeit halber soll noch erwähnt werden, dass ein **DataTable**-Objekt im Zustand **Added** per **Delete()** in den Zustand **Detached** versetzt wird.

Durch die **DataSet**-, **DataTable**- oder **DataRow**-Methode **AcceptChanges()** werden **DataRow**-Objekte im Zustand **Deleted** aus der Tabelle entfernt (in den Zustand **Detached** versetzt), wobei die Daten verloren gehen.

Denselben Effekt wie die **Delete()** - **AcceptChanges()** - Sequenz hat die **DataRowCollection**-Methode **Remove()**, z.B.:

```
dtCustomers.Rows.Remove(drem);
```

Beim Datenbank-Update (siehe unten) haben **DataRow**-Objekte im Zustand **Detached** *keinen* Effekt, sodass korrespondierende Tabellenzeilen der Datenbank unbehelligt bleiben.

20.5.5.2.6 Zeilenstatus und Zeilenversion

Um eine flexible und sichere Bearbeitung von **DataTable**-Objekten (insbesondere im Zusammenhang mit komplexen Softwaresystemen) zu ermöglichen, verwaltet ADO.NET für jede **DataTable**-Zeile ihren 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 **DataAdapter**-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 zurück 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 Tabelle entfernt (per **Remove()** oder durch eine **Delete()** - **AcceptChanges()** - Sequenz).
Deleted	Die zuvor im Zustand **Unchanged** 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** zurückversetzt 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:

<table>
<tr><th>DataRowVersion</th><th>Beschreibung</th></tr>
<tr><td>Current</td><td>Diese Version enthält die aktuellen Werte einer Tabellenzeile.</td></tr>
<tr><td>Original</td><td>Diese Version enthält die ursprünglichen Werte, genauer: die aktuellen Werte aus dem letzten Unchanged-Zustand der Zeile.</td></tr>
<tr><td>Proposed</td><td>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 verfügbar (deren Zustand ist: Detached).</td></tr>
<tr><td>Default</td><td>Die Standardversion hängt vom RowState-Wert ab:
<table>
<tr><th>DataRowState</th><th>Default-Version ist gleich</th></tr>
<tr><td>Unchanged</td><td>Original</td></tr>
<tr><td>Modified</td><td>Current</td></tr>
<tr><td>Added</td><td>Current</td></tr>
<tr><td>Detached</td><td>Proposed</td></tr>
<tr><td>Deleted</td><td>Current</td></tr>
</table>
</td></tr>
</table>

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 auf die Werte
Modified	**Unchanged**	Die **Original**-Version wird durch die **Current**-Version überschrieben.
Added	**Unchanged**	Die **Current**-Version wird in die **Original**-Version kopiert.
Deleted	**Detached**	Die Daten der Zeile werden gelöscht.

  Durch einen **AcceptChanges()**-Aufruf wird nötigenfalls der Bearbeitungsmodus für **DataRow**-Objekte über einen impliziten **EndEdit()**-Aufruf abgeschlossen (siehe Abschnitt 20.5.5.2.3).

- **RejectChanges()** wirkt sich folgendermaßen auf **DataRow**-Objekte aus:

RowState vorher	**RowState** nachher	Wirkung auf die Werte
Modified	**Unchanged**	Die **Current**-Version wird durch die **Original**-Version überschrieben.
Added	**Detached**	Die Daten der Zeile werden gelöscht.
Deleted	**Unchanged**	Die **Original**-Version wird in die **Current**-Version kopiert.

20.5.5.3 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 20.2.2). Von einer beteiligten Tabelle aus sind die **DataRelation**-Objekte über die **ChildRelations**- bzw. über die **ParentRelations**-Eigenschaft ansprechbar.

Leider kann ADO.NET die Beziehungsstruktur einer Datenbank **nicht** (z.B. per **FillSchema()**) automatisch übernehmen. Wird das in den Beispielen von Abschnitt 20.5.5 beteiligte **DataSet**-Objekt `ds` (mit den Tabellen `Customers` und `Orders`) nach der Anzahl seiner **DataRelation**-Objekte befragt,

```
Console.WriteLine("ds.Relations.Count: " + ds.Relations.Count);
```

resultiert das enttäuschende 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** aus der **Customers**-Tabelle und der gleichnamigen Child-Spalte aus der **Orders**-Tabelle:

```
DataColumn parent = dtCustomers.Columns["CustomerID"];
DataColumn child = dtOrders.Columns["CustomerID"];
ds.Relations.Add(new DataRelation("CustomersOrders", parent, child));
```

Zu einem **DataRelation**-Objekt erstellt ADO.NET automatisch **Constraint**-Objekte für die betroffenen Tabellen. Mit den folgenden Anweisungen:

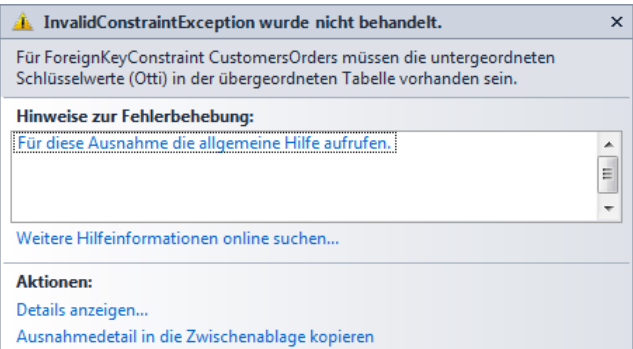
```
DataRelation co = ds.Relations["CustomersOrders"];
Console.WriteLine("\nParentKeyConstraint zur Beziehung \"CustomersOrders\":"+
   "\n Tabelle: " + co.ParentKeyConstraint.Table.TableName +
   "\n Feld:    " + co.ParentKeyConstraint.Columns[0].ColumnName +
   "\n Typ:     " + co.ParentKeyConstraint.GetType());
```

```
Console.WriteLine("\nChildKeyConstraint zur Beziehung \"CustomersOrders\":"+
   "\n Tabelle: " + co.ChildKeyConstraint.Table.TableName +
   "\n Feld:    " + co.ChildKeyConstraint.Columns[0].ColumnName +
   "\n Typ:     " + co.ChildKeyConstraint.GetType());
```

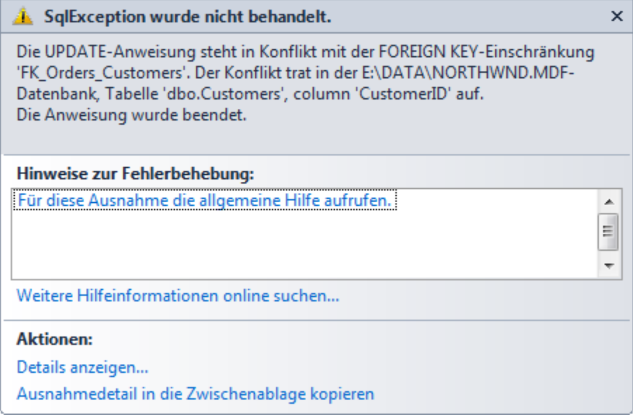
werden Informationen zu den Gültigkeitsregeln anfordern, die aus der Beziehung `CustomersOrders` resultieren:

```
ParentKeyConstraint zur Beziehung "CustomersOrders":
 Tabelle: Customers
 Feld:    CustomerID
 Typ:     System.Data.UniqueConstraint

ChildKeyConstraint zur Beziehung "CustomersOrders":
 Tabelle: Orders
 Feld:    CustomerID
 Typ:     System.Data.ForeignKeyConstraint
```

Sind solche Gültigkeitsregeln lokal (im Programm) bekannt, können Verstöße ohne kostenintensiven Datenbankzugriff per **InvalidConstraintException** erkannt und behandelt werden, z.B.



Anderenfalls muss man den SQL-Server belästigen und Wartezeiten in Kauf nehmen:



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 aufgetretene **ForeignKeyConstraint**-Objekt zur **Constraints**-Kollektion der **Orders**-Tabelle.

Über die **DataRows**-Methode **GetChildRows()** ermittelt man die zu einer Tabellenzeile aufgrund einer Master-Details - Beziehung gehörigen Detailzeilen, z.B.:

```
DataRow[] crs;
  . .
crs = dtCustomers.Rows[i].GetChildRows(ds.Relations["CustomersOrders"]);
```

20.5.6 Datenbank-Update

Für das Aktualisieren der Datenbank durch Übertragung des aktuellen **DataSet**-Zustands ist die **DataAdapter**-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 DML-Kommando (vgl. Abschnitt 20.3) an das RDBMS geschickt:

- ein **UPDATE**-Kommando für geänderte Zeilen
- ein **INSERT**-Kommando für ergänzte Zeilen
- ein **DELETE**-Kommando für geänderte Zeilen

Man kann die beteiligten SQL-Kommandos selbst erstellen und verpackt in ein **SqlCommand**-Objekt der **UpdateCommand**-, **InsertCommand-** bzw. **DeleteCommand-**Eigenschaft des **DataAdapters** zuweisen. Bequemer ist es jedoch, zum Adapter einen passenden **CommandBuilder** zu erzeugen, der die SQL-Aktionskommandos automatisch produziert:

```
new SqlCommandBuilder(dbAdapterCust);
```

Damit der **CommandBuilder** seine Arbeit verrichten kann, benötigt er Schlüsselspalteninformationen.

Die **CommandBuilder**-Kreationen basieren auf dem **SelectCommand**-Objekt des **DataAdapter**s, wobei eine wichtige Einschränkung zu beachten ist: Enthält das **SelectCommand**-Objekt eine *Sequenz* von **SELECT**-Kommandos (z.B. zum Erstellen von mehreren **DataTable**-Objekten, siehe Abschnitt 20.5.4), dann beachtet der **CommandBuilder** nur das *erste* **SELECT**-Kommando. Dementsprechend werden nur Änderungen an der korrespondierenden *ersten* Tabelle von der **DataAdapter**-Methode **Update()** an das DBMS übertragen. Auf den **CommandBuilder** zu verzichten und die Aktionskommandos selbst zu erstellen, kann sehr aufwändig sein. Eine sinnvolle Alternative besteht darin, für jede Datenbanktabelle ein separates Trio aus **DataAdapter**, **SelectCommand** und **CommandBuilder** zu verwenden, z.B.:

```
SqlCommand selCommandCus, selCommandOrd;
SqlDataAdapter dbAdapterCus, dbAdapterOrd;
  . . .
selCommandCus = new SqlCommand(
  "SELECT CustomerID, CompanyName, Country FROM Customers", dbConnection);
selCommandOrd = new SqlCommand(
  "SELECT OrderID, CustomerID, ShipCountry FROM Orders", dbConnection);
dbAdapterCus = new SqlDataAdapter(selCommandCus);
dbAdapterOrd = new SqlDataAdapter(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:

```
void UpdateData() {
   try {
      dbConnection.Open();
      dbAdapterCust.Update(dtCustomers);
      dbAdapterCust.Update(dtOrders);
   } finally {
      dbConnection.Close();
   }
}
```

Analog zur **Fill()**-Methode stellt auch die **Update()**-Methode bei Bedarf kurzeitig eine Datenbankverbindung her (vgl. Abschnitt 20.5.2.2). Bei einer *Sequenz* von Methodenaufrufen mit Verbindungsautomatismus kann es aber performanter sein, die Datenbankverbindung vorher explizit zu öffnen und so ein automatisches zwischenzeitliches Schließen zu vermeiden.

Am Ende eines **Update()**-Aufrufs wird automatisch eine **AcceptChanges()**-Nachricht an den passenden lokalen Datenspeicher geschickt, um **RowState**-Anpassungen zu veranlassen (siehe Abschnitt 20.5.5.2.6). Weil eine Tabelle unmittelbar nach einem **AcceptChanges()**-Aufruf nur noch Zeilen im Zustand **Unchanged** enthält, ist in dieser Situation ein **Update()**-Aufruf wirkungslos.

20.5.7 Zusammenspiel der ADO.NET - Klassen beim verbindungslosen Arbeiten

Im bisherigen Verlauf von Abschnitt 20.5 haben Sie viele ADO.NET - Klassen vorgestellt, wobei deren Verwendung jeweils durch Code-Segmente illustriert wurde. Jetzt soll endlich ein komplettes Beispielprogramm vorgeführt werden.

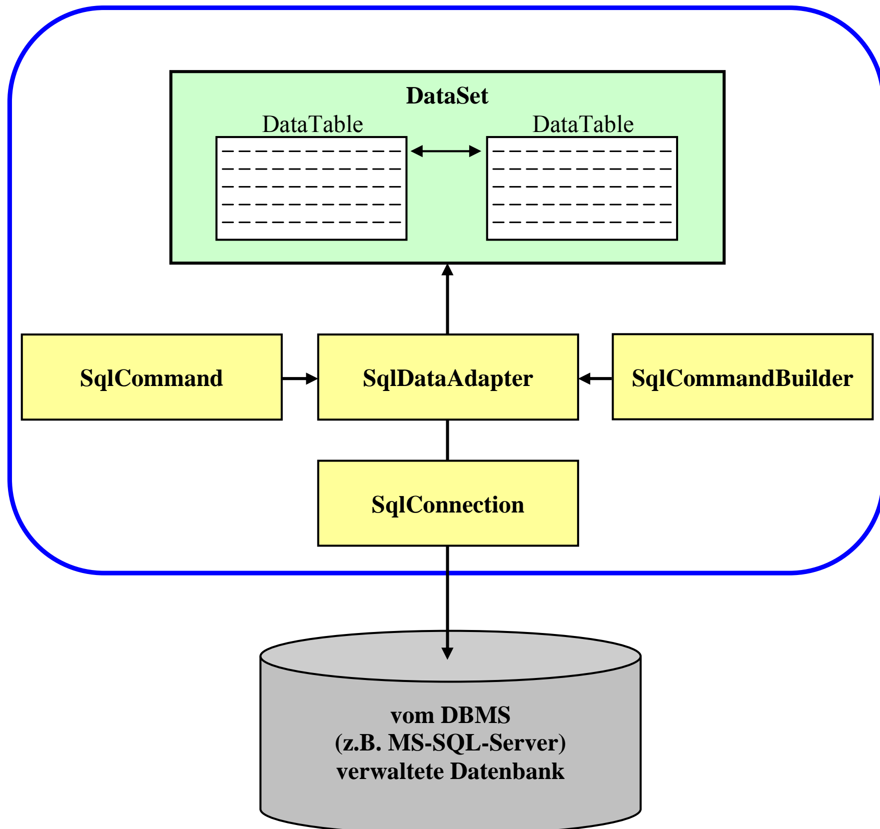
Im folgenden Programm werden im Sinne der verbindungslosen Datenbankbearbeitung aus der von einer SQL Server Express - Benutzerinstanz verwalteten Datenbank **Northwnd.mdf** die beiden Tabellen **Customers** und **Orders** in **DataTable**-Objekte eingelesen, die zu einem gemeinsamen **DataSet**-Objekt gehören. Dabei kommen die Klassen **SqlConnection**, **SqlDataAdapter**, **SqlCommand** und **SqlCommandBuilder** des **SqlClient** - Providers zum Einsatz:

```
using System;
using System.Data;
using System.Data.SqlClient;

class Verbindungslos {
 SqlConnection dbConnection = new SqlConnection();
 SqlCommand selCommandCus, selCommandOrd;
 SqlDataAdapter dbAdapterCus, dbAdapterOrd;
 DataSet ds = new DataSet();
 DataTable dtCustomers, dtOrders;

 Verbindungslos() {
  dbConnection.ConnectionString = @"Data Source=(local)\SQLEXPRESS;" +
   @"AttachDbFilename=E:\Data\NORTHWND.MDF;" +
   "Integrated Security=True;User Instance=True";
  selCommandCus = new SqlCommand(
   "SELECT CustomerID, CompanyName, Country FROM Customers", dbConnection);
  selCommandOrd = new SqlCommand(
   "SELECT OrderID, CustomerID, ShipCountry FROM Orders", dbConnection);
  dbAdapterCus = new SqlDataAdapter(selCommandCus);
  dbAdapterOrd = new SqlDataAdapter(selCommandOrd);

  // DML-Kommandos für beide DataAdapter automatisch erstellen lassen
  new SqlCommandBuilder(dbAdapterCus);
  new SqlCommandBuilder(dbAdapterOrd);
```

```
    // 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"];
   }

   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]);
   }

   void ChangeData() {
    dtCustomers.Rows[0]["Country"] = "Lummerland";
    dtOrders.Rows[0]["CustomerID"] = "TOMSP";
   }

   void UpdateData() {
    try {
     dbConnection.Open();
     dbAdapterCus.Update(dtCustomers);
     dbAdapterOrd.Update(dtOrders);
    }
    finally {
     dbConnection.Close();
    }
   }

   static void Main() {
    Verbindungslos dsd = new Verbindungslos();
    Console.WriteLine("Ausgangszustand:\n");
    dsd.PrintData();
    Console.ReadLine();
    dsd.ChangeData();
    Console.WriteLine("Nach der Änderung:\n");
    dsd.PrintData();
    dsd.UpdateData();
    Console.ReadLine();
   }
  }
```

Zum Übersetzen dieses Programms benötigt der Compiler Verweise auf die folgenden Bibliotheks-Assemblies: **System.dll**, **System.Data.dll**, **System.Data.SqlXml.dll** und **System.XML.dll**.

In der folgenden Abbildung wird die Kooperation der beteiligten Klassen skizziert:



20.5.8 Einsatz des DataReaders

Ist ausschließlich ein sequentiell-lesender Zugriff auf eine große Datenbank gefragt, dann kommt an Stelle der verbindungslosen Arbeitsweise mit einer lokalen Kopie der relevanten Datenbankinhalte der Einsatz einer Provider-spezifischen **DataReader**-Ableitung in Frage. Das folgende Beispielprogramm verwendet ein Objekt der zum Provider **SqlClient** gehörenden Klasse **SqlDataReader** dazu, aus der **Customers**-Tabelle der **Northwind**-Datenbank alle Firmennamen aufzulisten:

```
using System;
using System.Data.SqlClient;

class DataReaderDemo {
 static void Main() {
  SqlConnection dbConnection = new SqlConnection();
  dbConnection.ConnectionString =
   @"Data Source=.\SQLEXPRESS;AttachDbFilename=E:\Data\NORTHWND.MDF;" +
   @"Integrated Security=True;User Instance=True";

  SqlCommand command = new SqlCommand("SELECT CompanyName FROM Customers",dbConnection);
  SqlDataReader reader = null;

  try {
   dbConnection.Open();
   reader = command.ExecuteReader(System.Data.CommandBehavior.CloseConnection);
   while (reader.Read()) {
    Console.WriteLine(reader["CompanyName"]);
   }
  } finally {
   reader.Close();
  }
 }
}
```

Zur Klasse **SqlDataReader** existiert kein öffentlicher Konstruktor. Im Beispiel erzeugt die **SqlCommand**-Methode **ExecuteReader**() den **SqlDataReader** und liefert eine Referenz zurück. Der Parameter **CommandBehavior.CloseConnection** bewirkt, dass beim Schließen des **SqlDataReaders** auch die benutzte Datenbankverbindung geschlossen wird.

Ein **Read**()-Aufruf an den **SqlDataReader** beschafft die nächste Datenzeile und signalisiert ggf. mit dem Rückgabewert **false**, dass keine weiteren Zeilen verfügbar sind.

Solange ein **SqlDataReader** geöffnet ist, verwendet er das zugehörige **Connection**-Objekt exklusiv. Es wird erst durch explizites Schließen des **DataReaders** für eine andere Verwendung frei.

Wurde der **SqlDataReader** von einem parameterlosen **ExecuteReader**()-Aufruf geliefert,

```
reader = command.ExecuteReader();
```

dann bewirkt der **Close**()-Aufruf an den **SqlDataReader** *kein* Schließen der Datenbankverbindung. Ist das automatische Schließen der Verbindung gewünscht, verwendet man die im Beispiel vorgeführte **ExecuteReader**()-Überladung.

20.6 Typisierte DataSets

Bei der in Abschnitt 20.5 beschriebenen Datenbankentwicklung mit dem ADO.NET - Framework besteht Verbesserungsbedarf:

- Im Quellcode werden viele Datenbankelemente (z.B. Tabellen, Spalten) über Zeichenfolgen angesprochen, z.B.:
  ```
  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, so dass es leicht zu Laufzeitfehlern kommt.
- Es besteht ein Paradigmenbruch zwischen objektorientierter Programmierung einerseits und der Bearbeitung von Tabellenzeilen andererseits, den die angelsächsische Literatur oft als *impedance mismatch* bezeichnet.

Das erstgenannte Problem hat Microsoft mit dem so genannten **typisierten DataSet** behoben, wobei durch einen Assistenten der Entwicklungsumgebung (den **DataSet-Designer**) oder durch das Konsolenprogramm **xsd.exe**[116] zu einer Datenbankabfrage spezifische Klassen definiert werden, die in enger Beziehung zu den (untypisierten) ADO.NET - Klassen für das verbindungslose Arbeiten stehen, die Sie in Abschnitt 20.5 kennen gelernt haben.

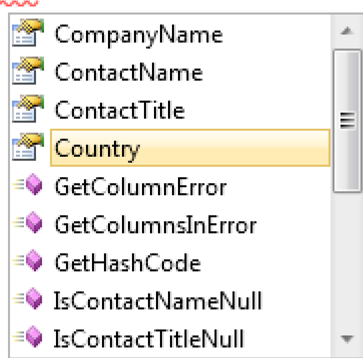
Beim Zugriff auf ein Datenbankfeld ist nun eine dem Compiler bekannte C# - Eigenschaft im Spiel, z.B.:

```
nORTHWNDDataSet.Customers[0].Country = "Lummerland";
```

IntelliSense hilft beim bequemen und tippfehlerfreien Kodieren:

[116] Bei der Installation der Visual C# 2010 Express Edition unter Windows 7 (64 Bit) landet das Hilfsprogramm im Ordner

%ProgramFiles(x86)%\Microsoft SDKs\Windows\v7.0A\bin



Für das zweite Problem (*impedance mismatch*) hat Microsoft nach langer Wartezeit mit dem *Entity Framework* (siehe Kapitel 22) eine attraktive ORM-Lösung (Object Relational Mapping) vorgelegt. Man kann also das typisierte DataSet als mittlerweile überholte Übergangslösung betrachten. Weil es lange Zeit die beste Option zur Datenbankentwicklung mit Microsofts Entwicklungswerkzeugen war, ist es in zahllosen Projekten anzutreffen und soll daher im Kurs behandelt werden.

20.6.1 Zu einer Datenbankabfrage automatisch definierte Klassen

Beim Einsatz eines typisierten DataSets werden vom gewählten Werkzeug (**DataSet-Designer** oder Konsolenprogramm **xsd.exe**) zu einer Datenbankabfrage etliche Klassen definiert:

- eine typisierte **DataSet**-Klasse
  Eine typisierte **DataSet**-Klasse wird von der Entwicklungsumgebung passend zu den benötigten Bestandteilen (z.B. Tabellen) einer Datenbank definiert, wobei die Klasse **DataSet** als Basisklasse dient, z.B.:

```
public partial class NORTHWNDDataSet : System.Data.DataSet {

 private CustomersDataTable tableCustomers;
 private OrdersDataTable tableOrders;
 private global::System.Data.DataRelation relationFK_Orders_Customers;
 . . .
}
```

  Es werden datenbankspezifische 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 ein dem Compiler bekanntes Symbol angesprochen werden. Tippfehler im Tabellennamen werden per IntelliSense minimiert bzw. vom Compiler reklamiert, statt einen Laufzeitfehler zu verursachen.

- typisierte **DataTable**- und **DataRow**-Klassen
  Die eben vorgeführte Eigenschaft `Orders` ist vom Typ `OrdersDataTable`, der speziell zur **Orders**-Tabelle der Datenbank vom Visual Studio als innere Klasse von `NORTHWNDDataSet` 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 dient eine projektspezifische **DataRow**-Ableitung, z.B.:

```
public partial class OrdersRow : System.Data.DataRow {
   . . .
}
```

Über die `OrdersDataTable`-Eigenschaft **Rows** ist eine Kollektion vom Typ **DataRowCollection** mit Objekten vom Typ `OrdersRow` ansprechbar.

Außerdem besitzt die Klasse `OrdersDataTable` für jede Spalte der zugehörigen Tabelle eine Eigenschaft mit dem Datentyp **System.Data.DataColumn**, und das referenzierte Objekt enthält die Schemainformationen der Tabellenspalte, z.B.:

```
this.columnOrderID.AutoIncrement = true;
this.columnOrderID.AllowDBNull = false;
this.columnOrderID.ReadOnly = true;
this.columnOrderID.Unique = true;
this.columnCustomerID.MaxLength = 5;
```

Durch diese Integration von Schemainformationen aus der Datenbank kann der Compiler etliche Fehler verhindert, was die Entwicklung beschleunigt und den Anwendern Ärger erspart.

- eine typisierte **TableAdapter**-Klasse zu jeder anwendungsspezifischen **DataTable**-Klasse, z.B.:

```
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;
   }
 }
 . . .
}
```

Der **TableAdapter** kann als typisierter, für eine bestimmte Tabelle zuständiger **DataAdapter** betrachtet werden und ist somit für das Befüllen der zugehörigen **DataTable** und das Datenbank-Update zuständig, z.B.:

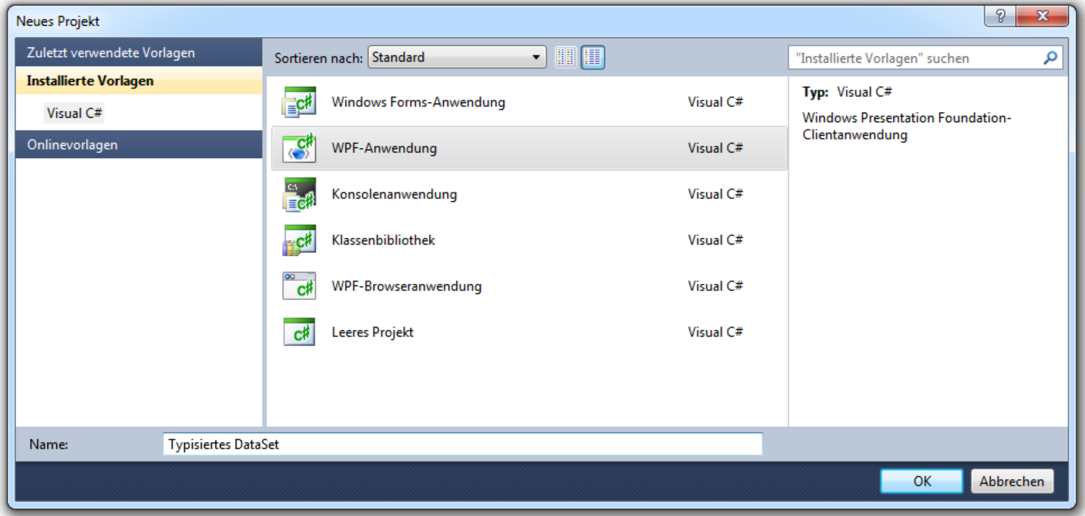
```
public virtual int Fill(NORTHWNDDataSet.OrdersDataTable dataTable) {
   this.Adapter.SelectCommand = this.CommandCollection[0];
   if ((this.ClearBeforeFill == true)) {
      dataTable.Clear();
   }
   int returnValue = this.Adapter.Fill(dataTable);
   return returnValue;
}
```

Eine Datenbank-Datenquelle im Sinn unserer Entwicklungsumgebung enthält also mindestens eine typisierte **DataSet**-Klasse und eine typisierte **DataTable**-Ableitung mit zugehöriger **TableAdapter**-Klasse.

Durch die zur Entwurfszeit erforderlichen Definitionen ist das Konzept der Datenquelle statisch. Werden *dynamisch* definierte Abfragen benötigt, müssen die Klassen **DataSet** und **DataAdapter** verwendet werden.

20.6.2 Anwendungsbeispiel

Wir erstellen eine WPF-Anwendung mit Datenbankzugriff über ein typisiertes DataSet. Legen Sie dazu ein neues Projekt auf Basis der Vorlage **WPF-Anwendung** an, z.B.:



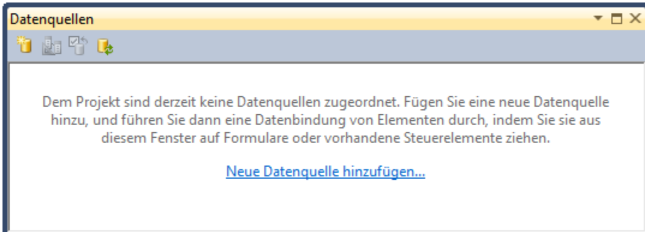
20.6.2.1 Datenquelle einrichten

Wir verwenden einen leistungsstarken Assistenten, der sich um die Verbindung zur Datenbank kümmert (z.B. die Verbindungszeichenfolge erstellt) und auch die in Abschnitt 20.6 beschriebenen anwendungsspezifischen (typisierten) Datenbankklassen erstellt.

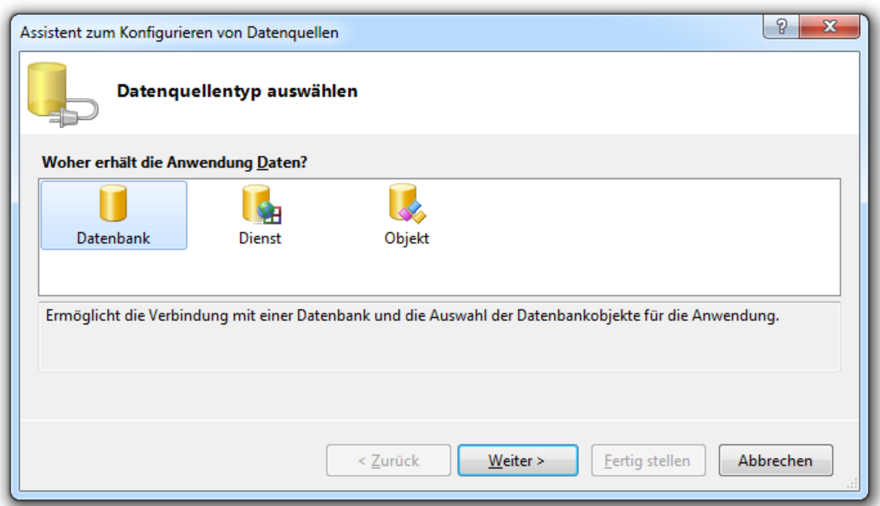
- Öffnen Sie nötigenfalls mit dem Menübefehl

  Daten > Datenquellen anzeigen

  das **Datenquellen**-Fenster. Es enthält einen Link, der zum Ergänzen einer neuen Datenquelle auffordert:

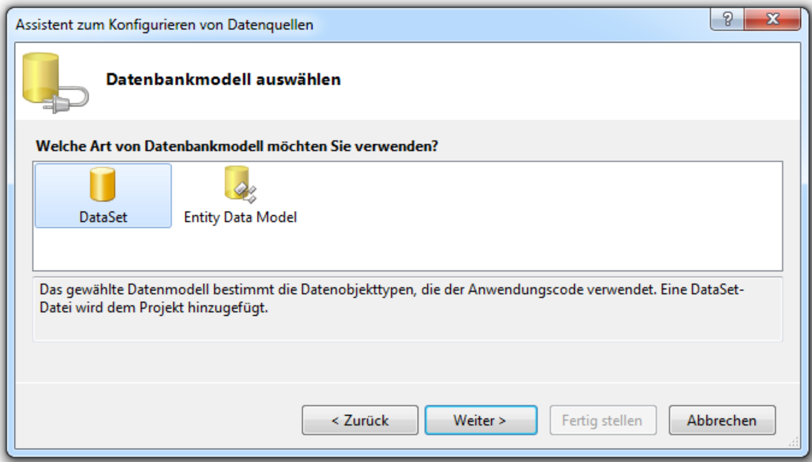


- Nach einem Klick auf diesen Link startet der Datenquellen-Konfigurationsassistent:



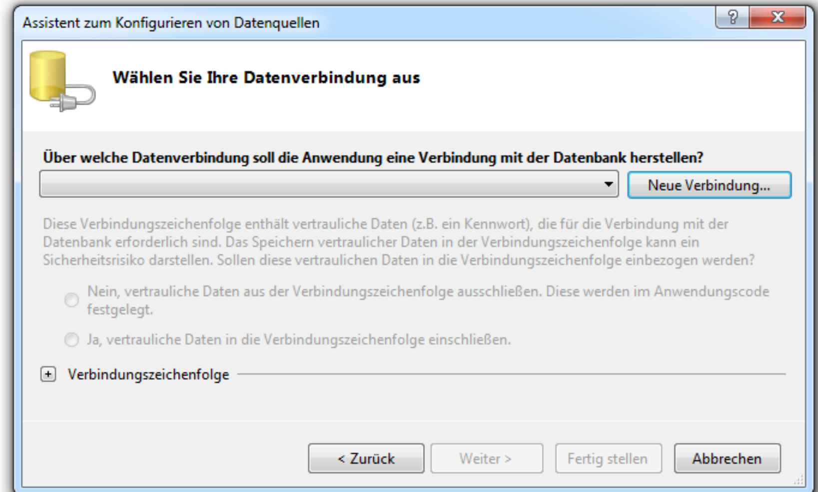
Wählen Sie den Datenquellentyp **Datenbank**. Mit den beiden anderen Optionen können wir uns in diesem Kurs aus Zeitgründen nicht beschäftigen.

- Wählen Sie im nächsten Dialog das Datenbankmodell **DataSet**

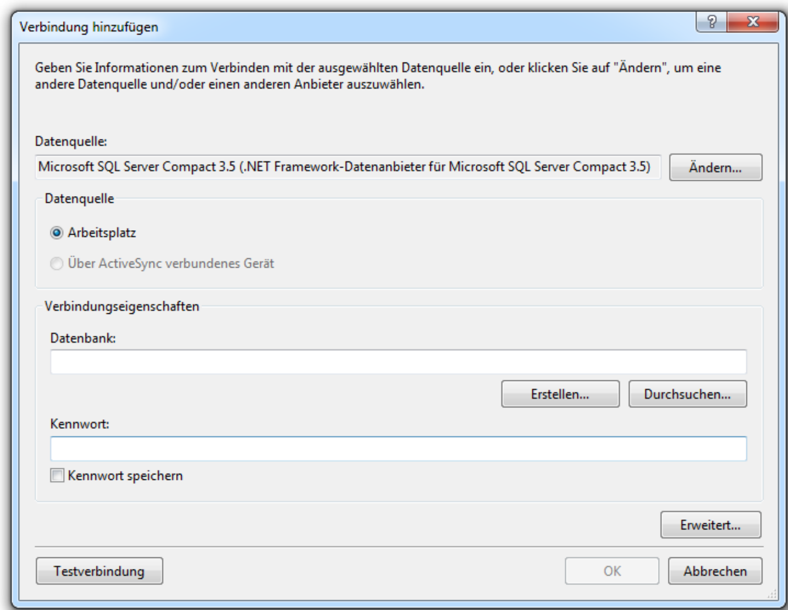


Mit dem **Entity Data Model** werden wir uns in Kapitel 22 geschäftigen.

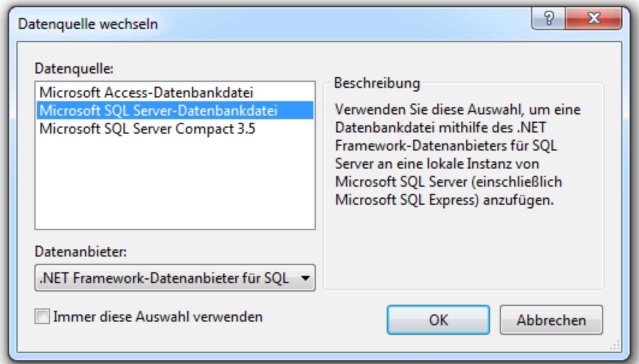
- Klicken Sie im Dialog zur **Auswahl einer Datenverbindung**



auf den Schalter **Neue Verbindung**, um folgende Dialogbox zu erhalten:

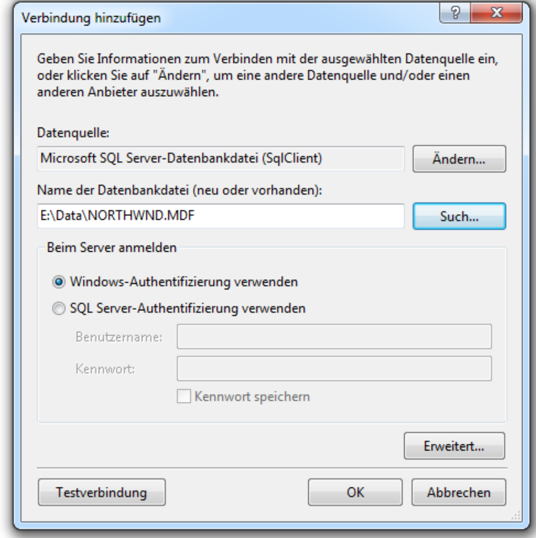


- Nach einem Klick auf den Schalter **Ändern** haben Sie (alle im Kurs vorgeschlagenen Installationen vorausgesetzt) die Wahl zwischen der Express- und der Compact-Edition von Microsofts SQL Server:

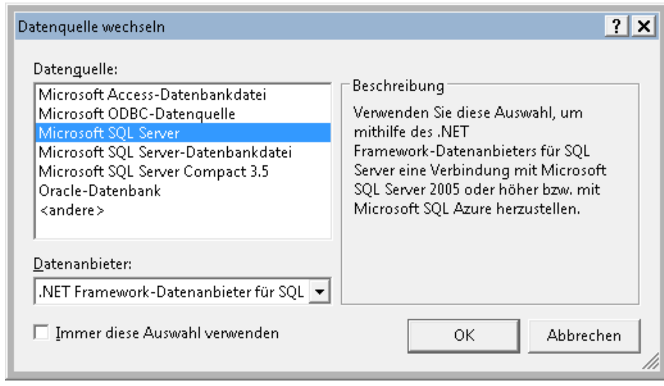


  Wenn *beide* Optionen vorhanden sind, spricht einiges für die leistungsfähigere Express-Edition. Am Zusatz *Datenbankdatei* hinter dem Eintrag *Microsoft SQL Server* ist erkennbar, dass eine Benutzerinstanz des SQL-Servers eingerichtet wird (vgl. Abschnitt 20.5.2.1).[117]

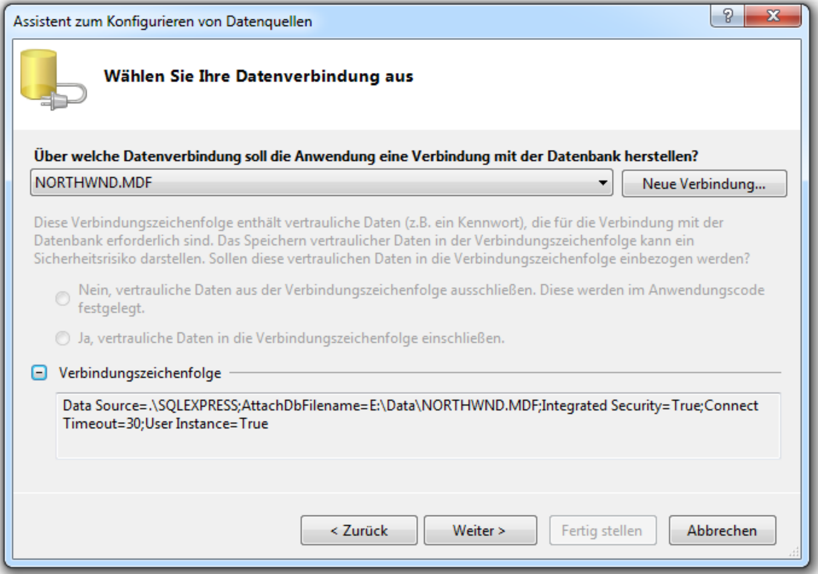
- Wählen Sie im Dialog **Verbindung hinzufügen** die Datenbankdatei **Northwnd.mdf**, und quittieren Sie mit **OK**:



[117] Wenn Sie statt der Visual C# 2010 Express Edition eine kostenpflichtige Version der Entwicklungsumgebung verwenden, stehen im Dialog **Datenquelle wechseln** mehr Optionen zur Verfügung, u.a. die Verbindung mit einem **Microsoft SQL Server** auf einem beliebigen Rechner:

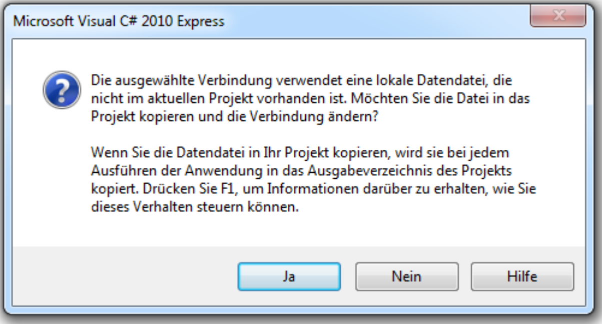


- Nun kann im Dialog zur Auswahl einer **Datenverbindung** die Verbindungszeichenfolge eingesehen werden:



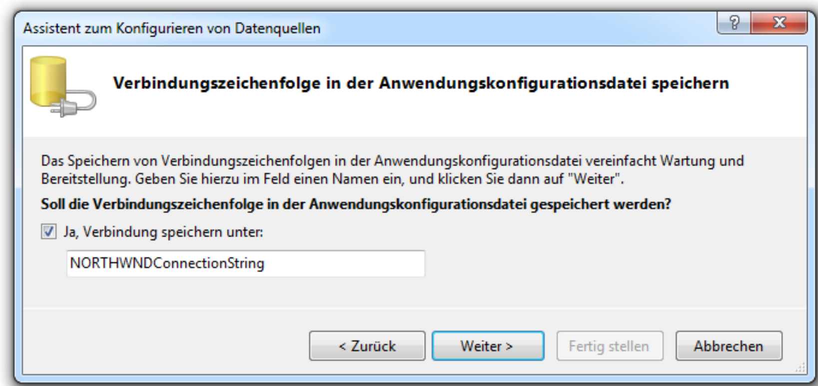
Das Ergebnis entspricht ziemlich genau unseren Erwartungen auf der Basis von Abschnitt 20.5.2.1.

- Wenn Sie **Weiter** machen, erkundigt sich die Entwicklungsumgebung nach Ihren Absichten zur Ablage der Datenbankdatei:

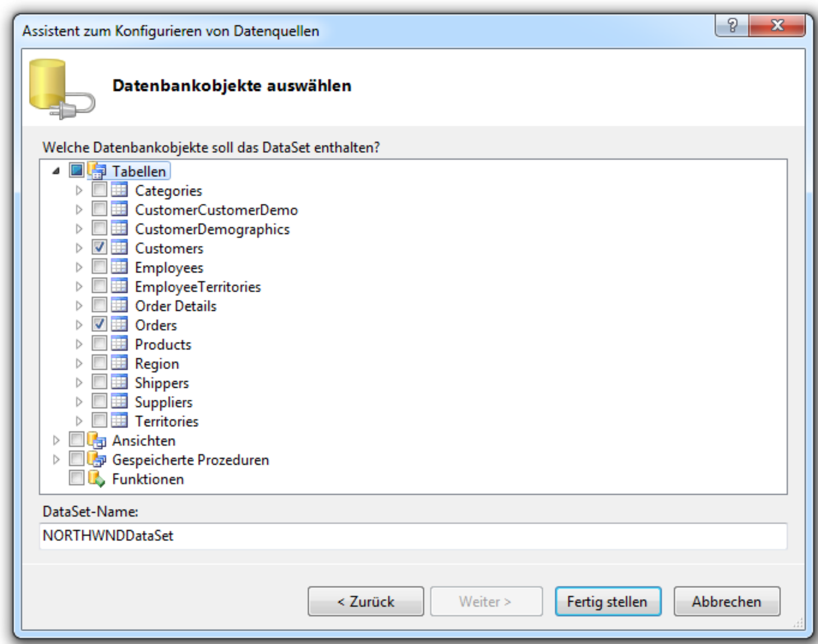


Eine Zustimmung zur projektinternen Kopie vereinfacht die Übertragung des Projekts auf andere Rechner.

- Die nächste Frage bezieht sich auf die in Abschnitt 20.5.2.1 behandelte Option, die Verbindungszeichenfolge in der Anwendungskonfigurationsdatei zu speichern. In der Regel sollten sie das Angebot annehmen:



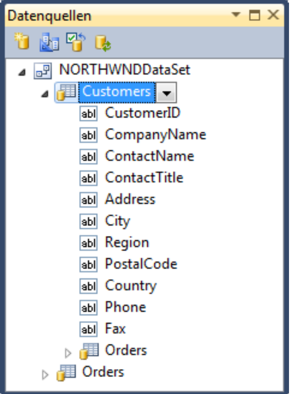
- Schließlich geht es an die Auswahl der benötigten Tabellen aus der **Northwind**-Datenbank:



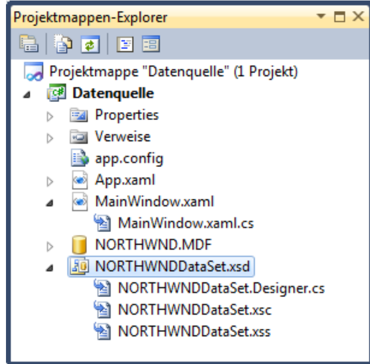
Wählen Sie die Tabellen **Customers** und **Orders**. Von der Möglichkeit, einzelne Tabellenspalten auszuwählen, machen wir keinen Gebrauch.

- Beenden Sie den Assistenten mit einem Klick auf den Schalter **Fertig stellen**.

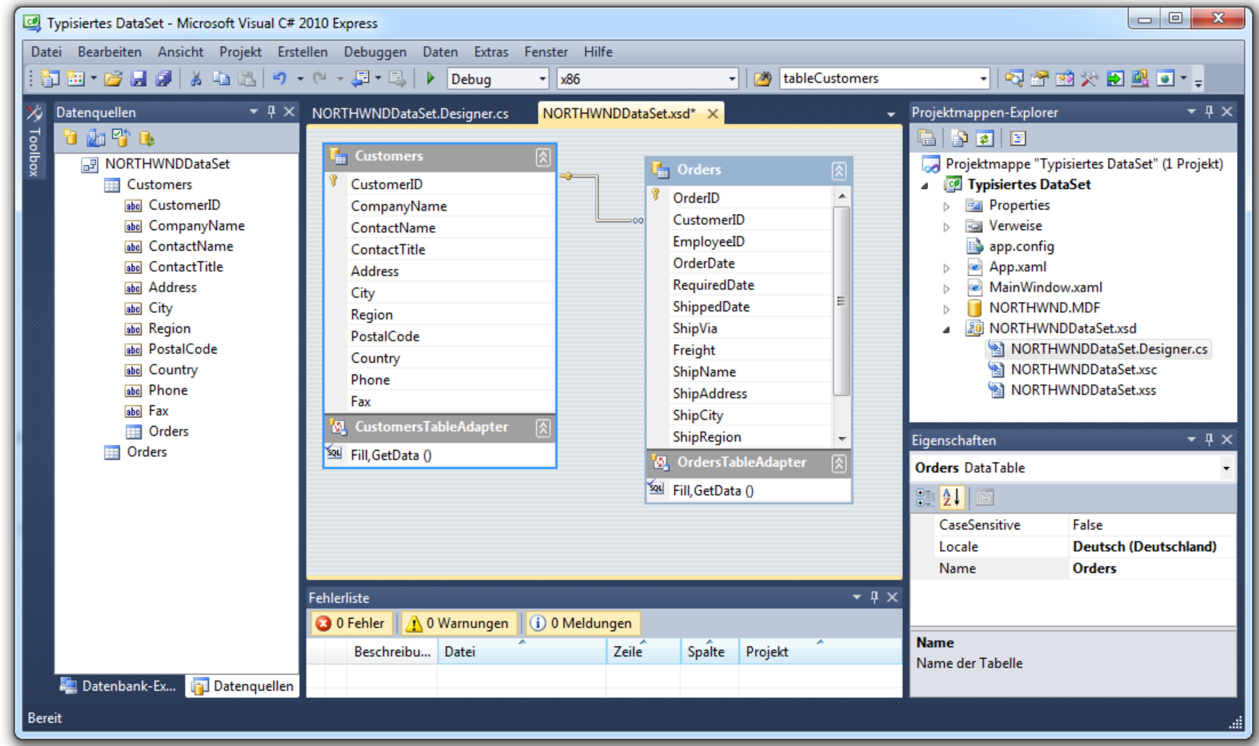
Im **Datenquellen**-Fenster der Entwicklungsumgebung sind die beiden ausgewählten Tabellen zu finden:



Im **Projektmappen-Explorer** zeigt sich nach dem Ergänzen der Datenquelle folgendes Bild:



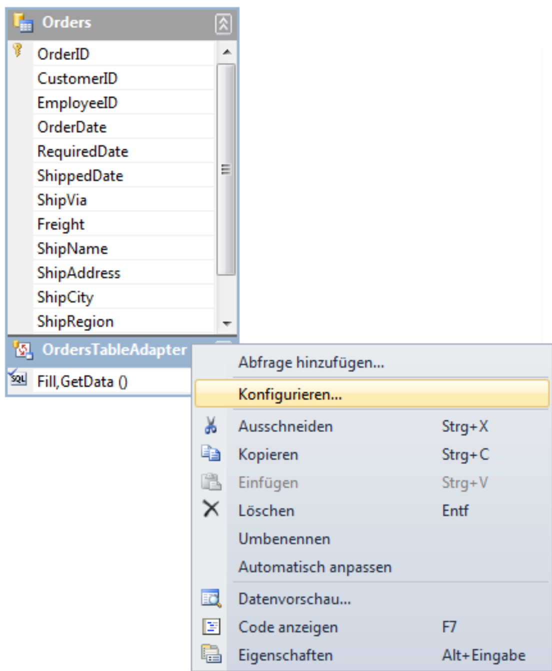
Die Entwicklungsumgebung hat mittlerweile die in Abschnitt 20.6 beschriebenen Klassen für die spezifischen Datenbankbedürfnisse unserer Anwendung angelegt, die nun mit Hilfe des **DataSet-Designers** konfiguriert werden können. Wählen Sie aus dem Kontextmenü zum Datenquellenfenster das Item **DataSet mit Designer bearbeiten**, oder setzen Sie einen Doppelklick auf den Eintrag **NORTHWNDDataSet.xsd** im Projektmappen-Explorer.



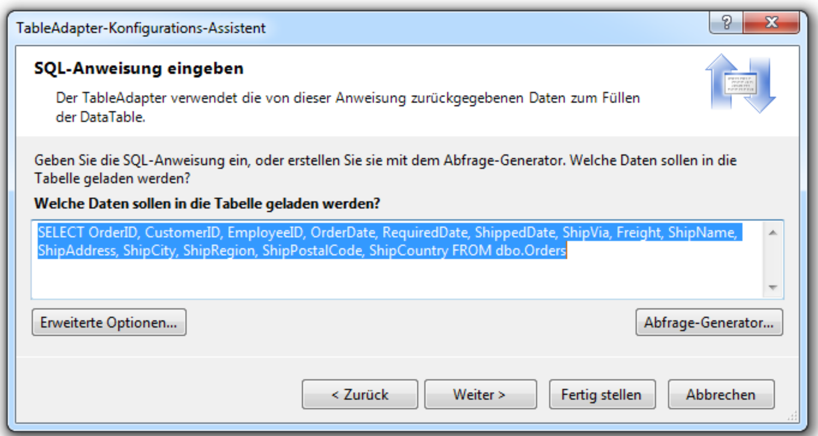
Wir können auf intuitive Weise die Schemadatei **NORTHWNDDataSet.xsd** zum typisierten **DataSet** bearbeiten. Auf den ersten Blick wird klar, dass auch die Master-Details - Beziehung aus der Datenbank übernommen wurde.

20.6.2.2 Abfragen modifizieren

Um eine Abfrage zu modifizieren, startet man den **TableAdapter-Konfigurations-Assistenten** im **DataSet**-Designer über das Item **Konfigurieren** aus dem Kontextmenü zu einem **TableAdapter**, z.B.:

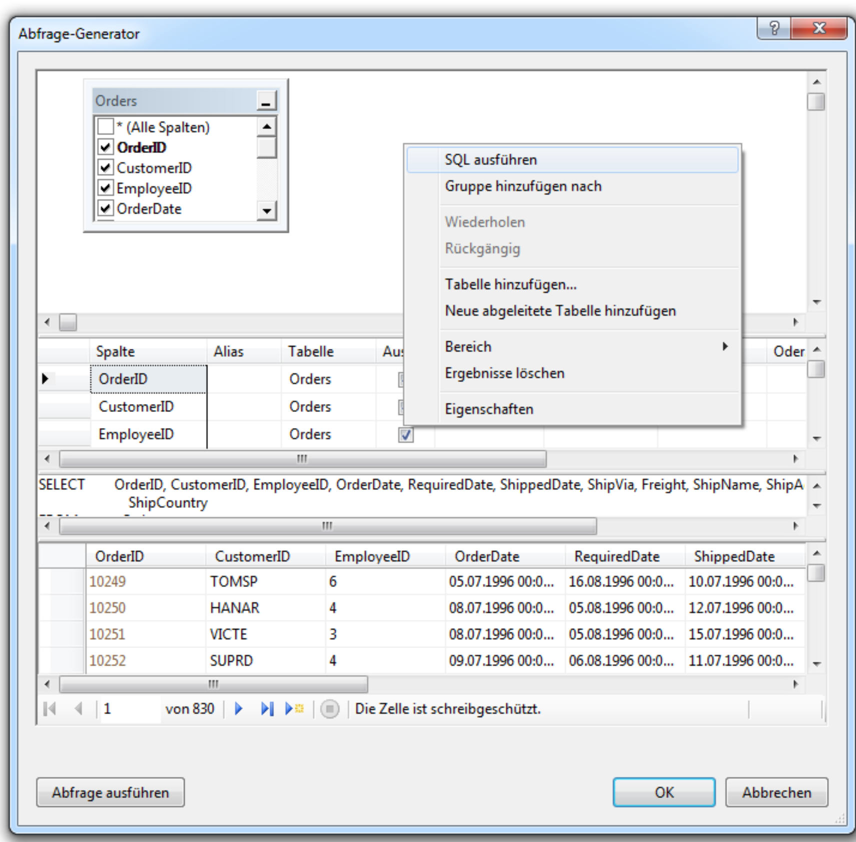


Hier ist das **SELECT**-Kommando zur Auswahlabfrage zu sehen und zu modifizieren:



Es handelt sich um die **CommandText**-Eigenschaft eines **SqlCommand**-Objekts, wie sich durch Inspektion der vom **DataSet**-Designer gepflegten Quellcodedatei **NORTHWND-DataSet.Designer.cs** leicht bestätigen lässt.

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**.

Aus Zeitgründen müssen wir viele Optionen zur Assistenten-gestützten Gestaltung der typisierten Datenbankklassen ignorieren, über die z.B. die folgende Microsoft-Webseite informiert:

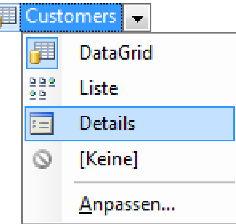
http://msdn.microsoft.com/de-de/library/8bw9ksd6.aspx

20.6.2.3 Datenverbundene Bedienelemente erstellen

Wir versorgen das Fenster unserer Anwendung nun mit datenzentrierten Bedienelementen und verwenden dabei die besonders bequeme *Drag Once* - Technik. Wie das Visual Studio dabei für Da-

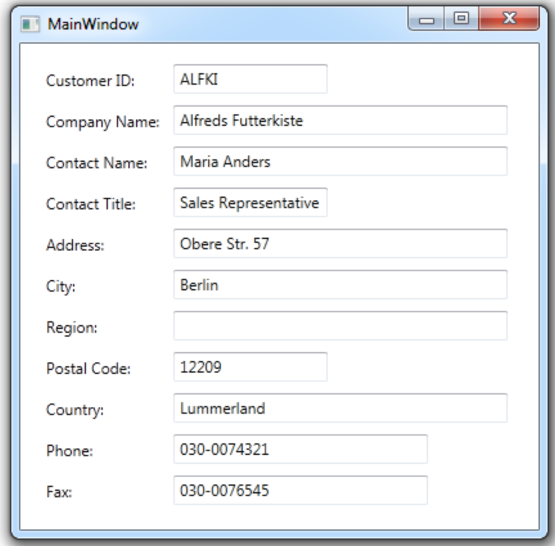
tenbindungen im Sinn von Kapitel 15 sorgt, wird in Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.** erläutert.

Öffnen Sie den Fensterdesigner. Im Datenquellenfenster wird für jede Tabelle und jede Spalte per Symbol angezeigt, welche Steuerelementklasse resultiert, wenn das Datenobjekt auf das Formular gezogen wird. Per DropDown-Menü kann die Voreinstellung abgeändert werden. Wählen Sie für die `Customers`-Tabelle anstelle der Voreinstellung **DataGridView** die Option **Details**,

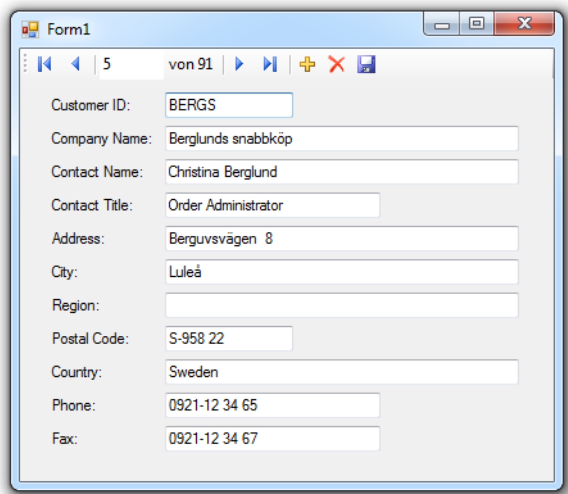


so dass jede Spalte ein eigenes Steuerelement gemäß Typeinstellung im Datenquellenfenster erhält.

Ziehen Sie die **Customers**-Tabelle auf das Fenster, und korrigieren Sie bei Bedarf per Maus die **Width**-Eigenschaftsausprägungen der **TextBox**-Steuerelemente, z.B.:



Leider fehlt in der WPF ein Äquivalent zum WinForms-Steuerelement **BindingNavigator**, das bei WinForms-Projekten mit dem oben geschilderten „Entwicklungsaufwand“ von den Assistenten kostenlos eingebaut wird:



Wer eine solche Symbolleiste in einem WPF-Projekt benötigt, muss sie wohl selber erstellen. Mit den Informationen aus Kapitel 15 über die Realisation der Datenbindung im automatisch erstellten XAML- bzw. C# - Code dürfte das kein großes Problem sein.

Bei unserer Anwendung besteht noch sehr viel Planungs- und Entwicklungsbedarf, z.B. zur Integration der **Orders** – Tabelle. Immerhin konnten etliche Routinearbeiten der Datenbankprogrammierung mit Assistentenhilfe bequem erledigt werden. Insbesondere sind Datenbindungen im Sinn von Kapitel 15 für die Steuerelemente hergestellt. In der XAML-Deklaration zum Hauptfenster wird der **Grid**-Layoutcontainer als übergeordnetes Element im WPF-Baum über sein Attribut **DataContext** mit einem Objekt der Klasse **CollectionViewSource** verbunden:

```
<Grid DataContext="{StaticResource customersViewSource}" HorizontalAlignment="Left" . . >
  . .
</Grid>
```

Dieses Objekt vermittelt im Sinn von Abschnitt 15.5 zwischen einem Kollektionsobjekt mit den Daten (= Datenbindungsquelle) und den Steuerelementen der Bedienoberfläche (= Datenbindungsziele) und erlaubt Transformation wie das Sortieren, Filtern oder Gruppieren ohne Änderung der Datenquelle. Es ist als statische Ressource zum Hauptfenster deklariert:

```
<Window.Resources>
    <my:NORTHWNDDataSet x:Key="nORTHWNDDataSet" />
    <CollectionViewSource x:Key="customersViewSource"
        Source="{Binding Path=Customers, Source={StaticResource nORTHWNDDataSet}}" />
</Window.Resources>
```

Genau genommen spielt das **CollectionViewSource**-Objekt nur eine Vermittlerrolle und ermöglichst es, per XAML-Code das eigentliche **CollectionView**-Objekt zu konfigurieren, das die Transformationen zwischen Quelle und Ziel der Datenbindung ausführt. Über die **CollectionViewSource**-Eigenschaft **Source** wird die Datenquelle festgelegt (siehe obigen XAML-Code), und über die Eigenschaft **View** ist das Transformationsobjekt zu ermitteln, was z.B. Code-Behind - Datei **MainWindow.xaml.cs** geschieht:

```
customersViewSource.View.MoveCurrentToFirst();
```

Als Datenbindungsquelle dient im Beispiel ein Objekt der Klasse `CustomersDataTable`, das über die Eigenschaft `Customers` der Klasse `NORTHWNDDataSet` anzusprechen ist. Die beiden Klassen gehören zu dem Datenbank-spezifisch vom Visual Studio definierten typisierten DataSet.

Um die Datenbindung im Anwendungsbeispiel zu komplettieren, wird per XAML-Code jedem Ziel, also jedem **TextBox**-Element des Hauptfensters, ein **Binding**-Objekt mit Quellangabe zugewiesen (vgl. Abschnitt 15.2.2). Im **Binding**-Objekt zur **Text**-Eigenschaft im **TextBox**-Element für die Anzeige und Änderung des Firmennamens wird per **Path**-Attribut die zu verbindende Quellspalte genannt:

```
<TextBox Name="companyNameTextBox"
         Text="{Binding Path=CompanyName, Mode=TwoWay, ValidatesOnExceptions=true,
               NotifyOnValidationError=true}" . . . />
```

Für den Rest sorgt die WPF-Datenbindungstechnik:

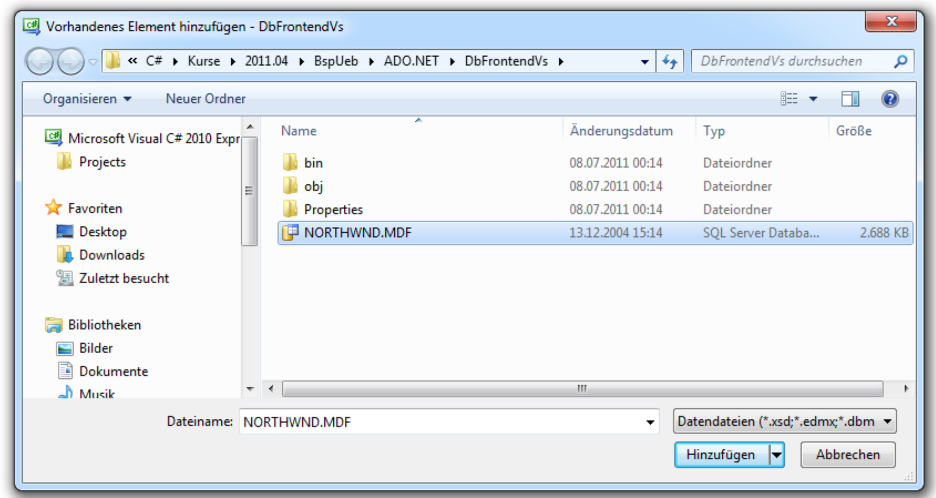
- Es wird im Elementbaum nach oben ein Quellobjekt gesucht, das der **DataContext**-Eigenschaft eines Steuerelements zugewiesen worden ist.
- Ist das Quellobjekt eine Kollektion, wird deren aktuelles Element ermittelt (im Beispiel: der aktuelle Datensatz). Zum Verschieben des Datensatzzeigers besitzt **CollectionView**-Objekt geeignete Methoden, z.B.
    - **MoveCurrentToFirst()**
      Auf das erste Element positionieren (siehe oben)
    - **MoveCurrentToLast()**
      Auf das letzte Element positionieren

- **MoveCurrentToNext()**
  Auf das nächste Element positionieren
- **MoveCurrentToPrevious()**
  Auf das vorherige Element positionieren

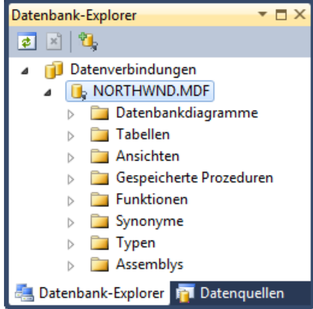
Leider fehlt die Zeit, die praktische Entwicklungsarbeit mit einem typisierten DataSet im erforderlichen Umfang zu erläutern. Ein Grund für die Zurückhaltung besteht darin, dass bei neuen Projekten das in Kapitel 22 vorzustellende Entity Framework gegenüber dem typisierten DataSet zu bevorzugen ist.

20.7 Datenbanken mit der Entwicklungsumgebung bearbeiten

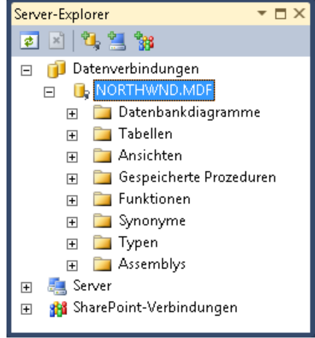
Das Visual Studio 2010 bietet (auch in der Express-Edition) dank der guten Kooperation mit Microsofts SQL-Server einige Möglichkeiten zum Erstellen und Bearbeiten von Datenbanken. Wir verwenden zur Demonstration erneut die **Northwind**-Datenbank. Kopieren Sie daher die Datenbankdatei **NORTHWND.MDF** in den Ordner eines neuen Projekts. Nehmen Sie die Datenbank per Projektmappen-Explorer über das Kontextmenü des Projekts mit **Hinzufügen > Vorhandenes Element** in das Projekt auf:



Brechen Sie den Datenquellen-Konfigurationsassistenten ab. Nach einem Doppelklick auf den Eintrag der Datenbankdatei im Projektmappen-Explorer erscheint der **Datenbank-Explorer**: (bei Visual C# Express)

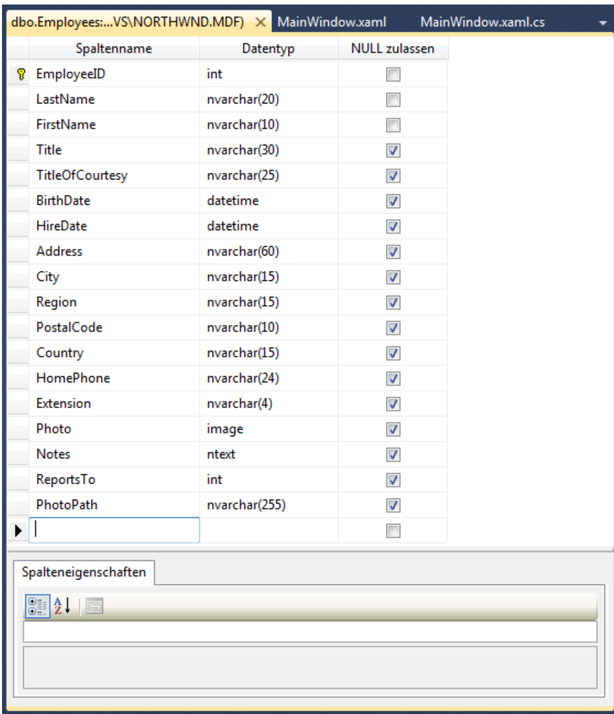


bzw. der **Server-Explorer** (bei einer kommerziellen Visual Studio - Variante):



20.7.1 Tabellen anzeigen und bearbeiten

Erweitern Sie den Zweig **Tabellen** und öffnen Sie per Doppelklick die Tabelle **Employees** zur Bearbeitung in der Designer-Zone der Entwicklungsumgebung:

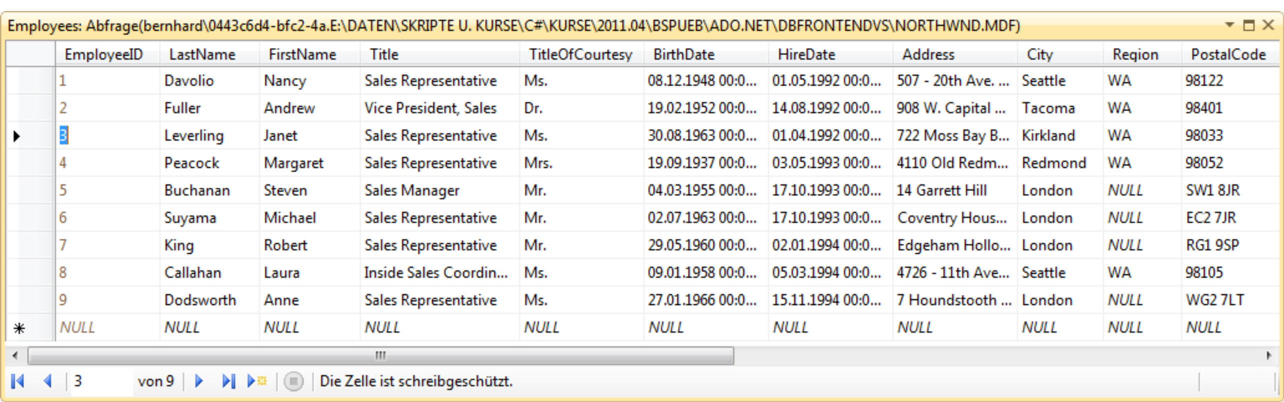


Spaltenname	Datentyp	NULL zulassen
EmployeeID	int	☐
LastName	nvarchar(20)	☐
FirstName	nvarchar(10)	☐
Title	nvarchar(30)	☑
TitleOfCourtesy	nvarchar(25)	☑
BirthDate	datetime	☑
HireDate	datetime	☑
Address	nvarchar(60)	☑
City	nvarchar(15)	☑
Region	nvarchar(15)	☑
PostalCode	nvarchar(10)	☑
Country	nvarchar(15)	☑
HomePhone	nvarchar(24)	☑
Extension	nvarchar(4)	☑
Photo	image	☑
Notes	ntext	☑
ReportsTo	int	☑
PhotoPath	nvarchar(255)	☑
		☐

Jetzt lassen sich z.B. neue Spalten definieren.

Im Kontextmenü zum **Tabellen**-Zweig im Datenbank-Explorer findet sich die Option, eine neue Tabelle anzulegen.

Über die Option **Tabellendaten anzeigen** im Kontextmenü einer vorhandenen Tabelle kann man in der Designer -Zone ihre Daten anzeigen und bearbeiten, z.B.:



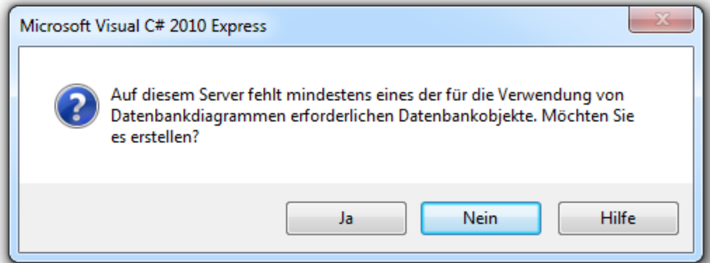
EmployeeID	LastName	FirstName	Title	TitleOfCourtesy	BirthDate	HireDate	Address	City	Region	PostalCode
1	Davolio	Nancy	Sales Representative	Ms.	08.12.1948 00:0...	01.05.1992 00:0...	507 - 20th Ave. ...	Seattle	WA	98122
2	Fuller	Andrew	Vice President, Sales	Dr.	19.02.1952 00:0...	14.08.1992 00:0...	908 W. Capital ...	Tacoma	WA	98401
3	Leverling	Janet	Sales Representative	Ms.	30.08.1963 00:0...	01.04.1992 00:0...	722 Moss Bay B...	Kirkland	WA	98033
4	Peacock	Margaret	Sales Representative	Mrs.	19.09.1937 00:0...	03.05.1993 00:0...	4110 Old Redm...	Redmond	WA	98052
5	Buchanan	Steven	Sales Manager	Mr.	04.03.1955 00:0...	17.10.1993 00:0...	14 Garrett Hill	London	NULL	SW1 8JR
6	Suyama	Michael	Sales Representative	Mr.	02.07.1963 00:0...	17.10.1993 00:0...	Coventry Hous...	London	NULL	EC2 7JR
7	King	Robert	Sales Representative	Mr.	29.05.1960 00:0...	02.01.1994 00:0...	Edgeham Hollo...	London	NULL	RG1 9SP
8	Callahan	Laura	Inside Sales Coordin...	Ms.	09.01.1958 00:0...	05.03.1994 00:0...	4726 - 11th Ave...	Seattle	WA	98105
9	Dodsworth	Anne	Sales Representative	Ms.	27.01.1966 00:0...	15.11.1994 00:0...	7 Houndstooth ...	London	NULL	WG2 7LT
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

20.7.2 Datenbankdiagramm erstellen

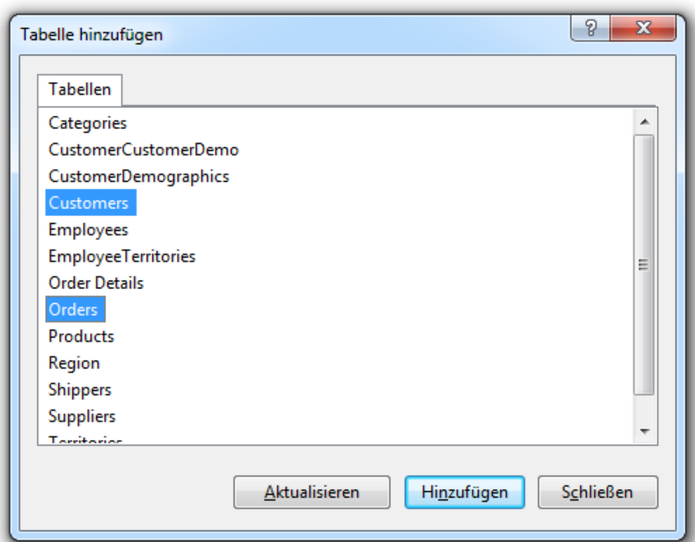
Wenn Sie im **Datenbank**- bzw. **Server-Explorer** den Zweig **Datenbankdiagramme** erweitern, erscheint eventuell der folgende Hinweis auf einen ungültigen DBO-Benutzer:

Machen Sie sich per Mausklick auf den **Ja**-Schalter zum DBO (***d**ata**b**ase **o**wner*) der Datenbank.

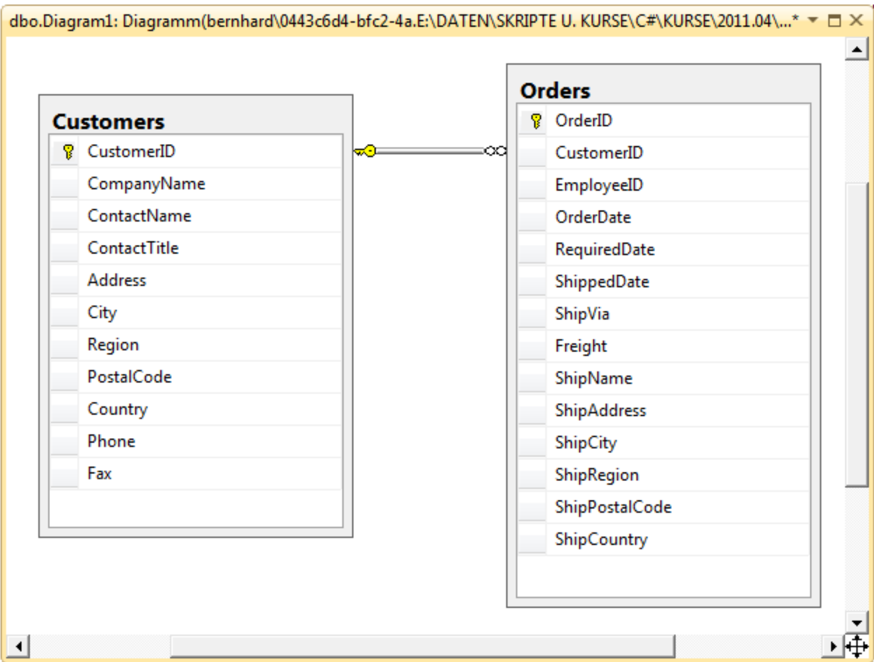
Veranlassen Sie ggf. anschließend das Erstellen fehlender Datenbankobjekte:



Über das Kontextmenü zum Zweig **Datenbankdiagramme** lässt sich nun ein **neues Diagramm hinzufügen**. Es erscheint ein Dialog zur Auswahl der Tabellen, die samt Relationen 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 der Entwicklungsumgebung durch Verschieben von Tabellen und Relationspfeilen gestalten:

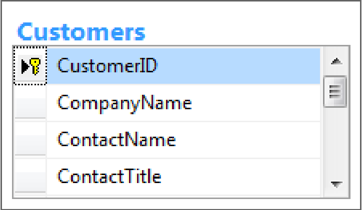


Beim Speichern des Diagramms (per **Strg-S** oder Symbolschalter angefordert) legt man seinen Namen fest.

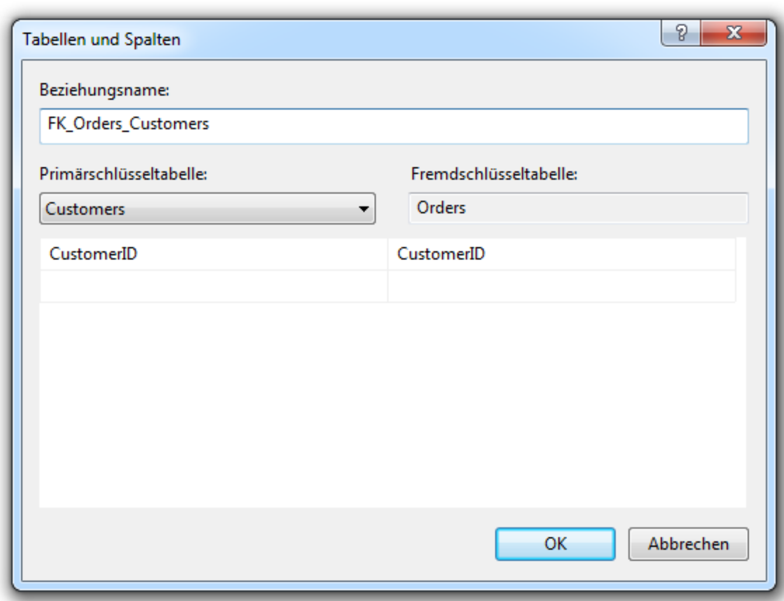
20.7.3 Beziehungen definieren

Über ein Datenbankdiagramm lassen sich Beziehungen definieren:

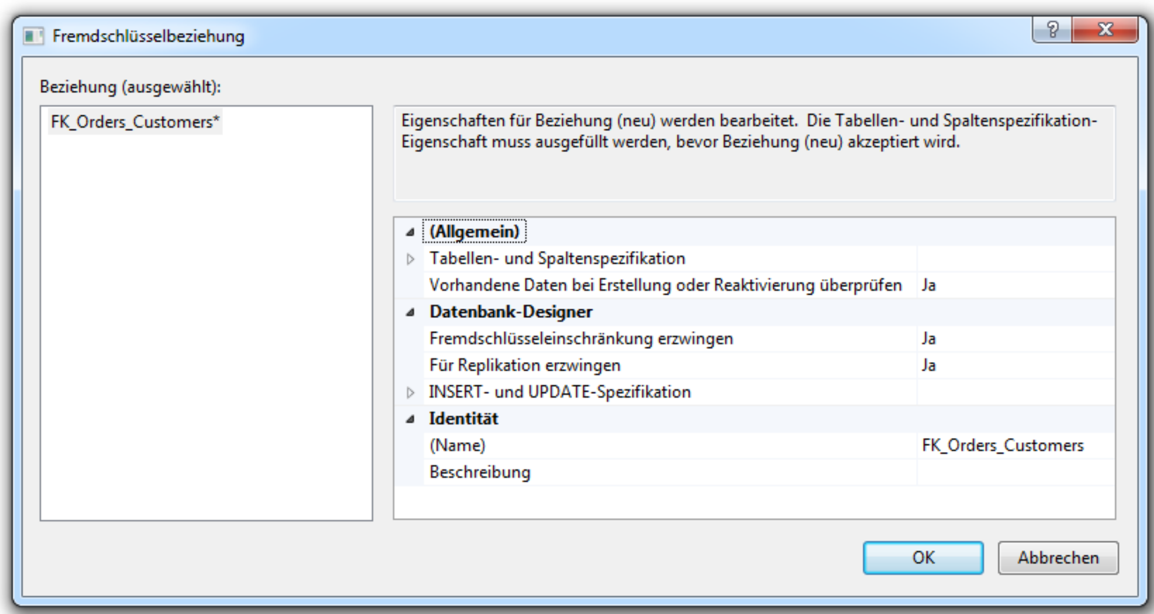
- Primärschlüssel per Klick auf das Schlüsselsymbol markieren, z.B.:



- 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**



sowie in der Dialogbox **Fremdschlüsselbeziehung**



jeweils mit **OK** quittieren.

21 LINQ

Seit C# 3.0 bzw. .NET 3.5 ist eine einheitliche, für diverse Datenquellen geeignete, typsichere, an SQL angelehnte Abfragetechnik namens LINQ (*Language Integrated Query*) verfügbar. Leider ist die Darstellung der Technik in diesem Manuskript noch unausgereift.

21.1 Erweiterungen der Programmiersprache C#

Um LINQ realisieren zu können, waren einige Erweiterungen der Programmiersprache C# erforderlich.

21.1.1 Implizite Typzuweisung bei lokalen Variablen

Bei der Deklaration von *lokalen* Variablen kann die explizite Typangabe durch das Schlüsselwort **var** ersetzt werden, wobei der obligatorische Initialisierungswert den Typ liefert, z.B.:

Quellcode:

```
using System;
class Prog {
   static void Main() {
      var i = 13;
      var d = 3.14;
      Console.WriteLine("Wert: " + i +
         "\nTyp:  "+i.GetType());
      Console.WriteLine("\nWert: " + d +
         "\nTyp:  "+d.GetType());
   }
}
```

Ausgabe:

```
Wert: 13
Typ:  System.Int32

Wert: 3,14
Typ:  System.Double
```

Die strenge Typisierung von C# wird keinesfalls gelockert, so dass sich z.B. auch ein implizit festgelegter Typ nicht mehr ändern lässt.

21.1.2 Instanz- und Listeninitialisierer

Beim Instantiieren kann man *öffentliche* Felder und Eigenschaft mit Werten versorgen, ohne einen speziellen Konstruktor zu benötigen, z.B.:

Quellcode:

```
using System;
class CT {
   public int i;
}
struct ST {
   public int i;
}
class Prog {
   static void Main() {
      CT ct = new CT {i = 13 };
      ST st = new ST {i = 4711};
      Console.WriteLine(ct.i + "\n" + st.i);
   }
}
```

Ausgabe:

```
13
4711
```

Wie sich gleich in Abschnitt 21.1.3 zeigen wird, ist diese Technik speziell für die neuen *anonymen* Typen von Interesse.

Bei Attributen haben wir übrigens unter der Bezeichnung *Namensparameter* eine analoge Option kennen gelernt (vgl. Abschnitt 11.4).

Analog zu den vertrauten Array-Initialisierungslisten (vgl. Abschnitt 5.3.5), die Array-Objekte bei der Kreation mit Werten versorgen, arbeiten die mit C# 3.0 eingeführten Initialisierer für (generische) Listen-Objekte, z.B.:

Quellcode

```
using System;
using System.Collections.Generic;
using System.Collections;

class ListenInitialisierer {
   static void Main() {
      // Generische Liste
      List<int> il = new List<int>() {1, 2, 3};
      foreach (int e in il)
         Console.Write(e + " ");

      // ArrayList-Behälter für Elemente mit beliebigem Typ
      ArrayList ali = new ArrayList() {"abc", 2, 3.14};
      Console.WriteLine();
      foreach (object e in ali)
         Console.Write(e.ToString() + " ");
      Console.ReadLine();
   }
}
```

Ausgabe

```
1 2 3
abc 2 3,14
```

21.1.3 Anonyme Klassen

Wird bei der Deklaration einer lokalen Variablen die explizite Typangabe durch das Schlüsselwort **var** (siehe Abschnitt 21.1.1) ersetzt und die Initialisierung durch einen Objekt- oder Listeninitialisierer (siehe Abschnitt 21.1.2) vorgenommen, dann resultiert ein Objekt einer **anonymen Klasse**. Genau genommen hat die Klasse durchaus einen (vom Compiler vergebenen) Namen, doch darf dieser Name im Quellcode nicht verwendet werden.

Man könnte für die Referenzvariable auch den Datentyp **Object** verwenden, müsste dann aber beim Zugriff auf die Member jeweils eine umständliche Typumwandlung vornehmen.

Als Innenausstattung enthält die anonyme Klasse eine öffentliche **get-only** - Eigenschaft für jedes Listenelement des Initialisierers, so dass im folgenden Beispiel

Quellcode-Segment

```
var a = new {Name="Knut", Alter=53};
Console.WriteLine(a.GetType());
```

Ausgabe

```
<>f__AnonymousType0`2[System.String,System.Int32]
```

eine Klasse mit den Eigenschaften `Name` und `Alter` resultiert, wobei der Compiler die Datentypen (**String** und **Int32**) aus dem zugewiesenen Werten ermittelt. Zu den Eigenschaften gehören private Felder. Außerdem erbt eine anonyme Klasse die Member ihrer Basisklasse **Object**.

An der Stelle eines Name-Wert - Paars kann im Objektinitialisierer auch ein Variablen- oder Parametername der umgebenden Methode stehen, der als Eigenschaftsname für die anonyme Klasse übernommen wird, z.B.:

Quellcode-Segment	Ausgabe

`int Alter = 53;`
`var b = new {Name = "Knut", Alter};`
`Console.WriteLine(b.Alter.GetType());`
	`System.Int32`

Aufgrund ihrer Entstehungsgeschichte können anonyme Klassen über das **Object**-Erbgut hinaus keine Handlungskompetenzen besitzen.

Man verwendet die Objekte anonymer Klassen nur in den erzeugenden Methoden. Solche Objekte über das Ende der erzeugenden Methoden hinaus (durch Export von **Object**-Referenzen) am Leben zu erhalten, ist zwar möglich, aber kaum jemals nützlich.

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 = "Kurt", Alter = 54};
  . . .
b = a;
```

21.1.4 Lambda-Ausdrücke

Mit den Lambda-Ausdrücken bietet C# seit der Version 3.0 eine syntaktisch einfachere (allerdings gewöhnungsbedürftige) Alternative zu den anonymen Methoden (siehe Abschnitt 9.3.1.4), die schon seit C# 2.0 dazu dienen, ohne explizite Methodendefinition ein ausführbares Delegatenobjekt zu erstellen. Hier sind für einen einfachen Delegatentyp

```
delegate int Rest(int a, int b);
```

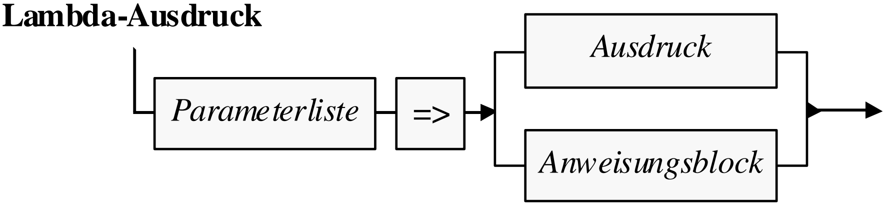
die äquivalenten Realisationen über eine anonyme Methode

```
Rest R = delegate(int a, int b) {
   return Math.Max(a, b) % Math.Min(a, b);
};
```

und einen Lambda-Ausdruck zu sehen:

```
Rest R = (int a, int b) => Math.Max(a, b) % Math.Min(a, b);
```

Um mit der Syntax des Lambda-Ausdrucks vertraut zu werden, verschaffen wir uns zunächst einen Überblick mit Hilfe eines Syntaxdiagramms:



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.:
  ```
  Rest R = (a, b) => Math.Max(a, b) % Math.Min(a, b);
  ```
- Bei einem einzelnen, implizit typisierten Parameter kann man die runden Klammern weglassen, z.B.:
  ```
  delegate bool Krit (int a);
    . . .
  Krit K = a => a % 2 == 0;
  ```

Ist der Lambda-Rumpf ein Anweisungsblock und der Rückgabetyp des zu erfüllenden Delegaten von **void** verschieden, dann muss der Rückgabewert per **return**-Anweisung geliefert werden, z.B.:

```
Krit K = a => {
   double r = Math.Sqrt(a);
   return r % 1 == 0 ? true: false;
};
```

21.1.5 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 umständlich, oder unmöglich ist (z.B. bei Strukturen oder versiegelten Klassen). 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.

Im folgenden Beispiel wird für Instanzen der Struktur **Double** das Potenzieren durch die Erweiterungsmethode `H()` syntaktisch vereinfacht:

<table>
<tr><th>Quellcode-Segment</th><th>Ausgabe</th></tr>
<tr><td>

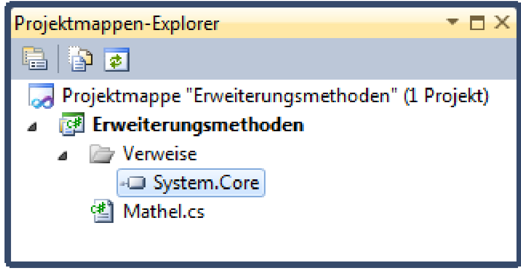
```
using System;
static class Mathe {
   public static double H(this double arg, double expo) {
      return Math.Pow(arg, expo);
   }
}
class Prog {
   static void Main() {
      double pi = 3.14;
      Console.WriteLine(pi.H(2));
   }
}
```

</td><td>

```
9,8596
```

</td></tr>
</table>

Bei Erweiterungsmethoden ist das Attribut **System.Runtime.CompilerServices.ExtensionAttribute** beteiligt. Es darf nicht im Quellcode verwendet werden, muss aber trotzdem für den Compiler auffindbar sein, so dass ein Verweis auf das Bibliotheks-Assembly **System.Core.dll** erforderlich ist, z.B.:



Tritt eine Erweiterungsmethode in Konkurrenz mit einer typeigenen, dann gewinnt letztere.

21.2 LINQ to Objects

In diesem Abschnitt werden wir wichtige LINQ-Abfragetechniken beim Zugriff auf Daten in Kollektions-Objekten bzw. Arrays üben, wobei die in Abschnitt 21.1 eingeführten C# - Sprachergänzungen sowie neue C# - Schlüsselwörter zum Einsatz kommen.

Eine zentrale Rolle bei LINQ-Abfragen spielen diejenigen Erweiterungsmethoden, welche die statische Klasse **Enumerable** im Namensraum **System.Linq** für *alle* Typen realisiert, welche die Schnittstelle **IEnumerable<T>** implementieren (z.B. für beliebige Arrays).

Beim Datenzugriff mit LINQ to Objects können Sie zwischen zwei Syntax-Varianten wählen:

- **Erweiterungsmethoden-Syntax**
  Letztlich übersetzt der Compiler alle LINQ-Datenzugriffe in Aufrufe von Erweiterungsmethoden der Klasse **Enumerable**, und viele LINQ-Optionen erfordern auch die direkte Verwendung dieser Methodenaufrufe. Wer in die objektorientierte Programmierung eingedacht ist und sich mit den Spracherweiterungen in Abschnitt 21.2 vertraut gemacht hat, wird mit Konstruktionen wie im folgenden Beispiel bald keine großen Probleme haben:
  ```
  Customer[] customers = new Customer[7];
  .  .  .
  var cust = customers.
                 Where(c => c.Country == "Mexico").
                 Select(c => new {c.CustomerID, c.Country});
  ```
  Hier werden die **Enumerable**-Erweiterungsmethoden **Where()** und **Select()** (siehe unten) für ein Array-Objekt vom Typ `Customer[]` ausgeführt. An **Where()** wird als Aktualparameter eine per Lambda-Ausdruck realisierte anonyme Methode übergeben, die einen Rückgabewert vom Typ **bool** liefert. Als **Where()**-Rückgabe erhält man ein Kollektionsobjekt vom Typ **IEnumerable**`<Customer>`.

  Select() erhält als Aktualparameter eine per Lambda-Ausdruck realisierte anonyme Methode, die ein Objekt eines anonymen Typs liefert, das eine Auswahl der `Customer`-Eigenschaften besitzt. Als **Select()**-Rückgabe erhält man ein Objekt, das die generische Schnittstelle **IEnumerable<T>** mit Elementen vom anonymen Typ erfüllt und bei Bedarf frische Daten aus der Quelle beschafft (siehe Abschnitt 21.2.2).

  Offenbar realisiert **Where()** einen *Filter*, während **Select()** die verbliebenen Elemente in ein neues Format bringt (*projiziert*).
- **Abfragesyntax**
  Alternativ kann oft eine SQL-ähnliche Syntax verwendet werden, die auf den ersten Blick angenehmer wirkt, z.B.:
  ```
  var cust = from c in customers
                 where c.Country == "Mexico"
                 select new {c.CustomerID, c.Country};
  ```
  Man sollte allerdings wissen, welche Erweiterungsmethoden hinter den neuen Schlüsselwörtern (z.B. **from**, **where**, **select**) stecken, hat also im Vergleich zur reinen Erweiterungsmethoden-Syntax etwas mehr Lernaufwand, bevor man die elegantere Syntax sicher beherrscht. In manchen Situationen kommt man um den direkten Aufruf von Erweiterungsmethoden nicht herum.

Anschließend werden elementare **Enumerable**-Erweiterungsmethoden zusammen mit ihren Abfrage-Entsprechungen dargestellt.

21.2.1 Die from-in - Klausel zur Definition einer Datenquelle

Ein LINQ-Abfrageausdruck beginnt mit einer **from-in** - Klausel mit Angabe der Datenquelle und unterscheidet sich damit auf den ersten Blick von einer SELECT-Abfrage in SQL (vgl. Abschnitt 20.3). Indem zunächst die Datenquelle ausgewählt wird, von der die nachfolgenden Operationen ausgehen, kann die Entwicklungsumgebung mit ihrer IntelliSense-Technik das Erstellen von Abfragesyntax ebenso unterstützen wie das Erstellen von Erweiterungsmethoden-Syntax.

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<T>** implementieren, und die hinter dem Schlüsselwort **from** genannte Elementvariable hat implizit den Typ **T**. Im Beispiel hat die Datenquelle `customers` den Array-Typ `Customer[]`, und `c` ist folglich vom Typ `Customer`.

Implementiert eine Kollektion nur die nicht-generische Schnittstelle **IEnumerable**, muss hinter dem Schlüsselwort **from** 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 Erweiterungsmethoden-Syntax wird keine **from-in** - Entsprechung benötigt, z.B.:

```
var custIds = customers.Select(c => c.CustomerID);
```

21.2.2 Projektion des Elementtyps der Quelle auf den Elementtyp der Abfrage

Mit der **Enumerable**-Erweiterungsmethode **Select<TSource, TResult>()**

> **public static IEnumerable<TResult> Select<TSource, TResult>(**
> **this IEnumerable<TSource>** *quelle*, **Func<TSource, TResult>** *projektor*)

bzw. mit dem korrespondierenden Abfrage-Operator **select** 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 erwartet **Select<TSource, TResult>()** einen Parameter vom Delegatentyp **Func<TSource, TResult>**:

> **public delegate TResult Func<in TSource, out TResult>(TSource** *arg*)

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;
   public String CompanyName;
   public String Country;
}
```

und die Elemente der Abfrage vom Typ **String**. Das projizierende Delegatenobjekt wird über einen Lambda-Ausdruck erstellt.

So sieht die äquivalente Abfragesyntax aus:

```
var custIds = from c in customers select c.CustomerID;
```

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, so dass in der verzögert ausgeführten Abfrage der aktuelle Zustand enthalten ist:

Quellcode-Segment	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, verwendet man eine anonyme Klasse (vgl. Abschnitt 21.1.3). Dieser Erweiterungsmethodenaufruf

```
var cust = customers.Select(c => new {c.CustomerID, c.Country});
```

liefert wie die äquivalente Abfragesyntax

```
var cust = from c in customers
           select new {c.CustomerID, c.Country};
```

Abfrageelemente mit den beiden Eigenschaften `CustomerID` und `Country`.

21.2.3 Filtern

Mit der **Enumerable**-Erweiterungsmethode **Where<TSource>()**

public static IEnumerable<TSource> Where<TSource>(
this IEnumerable<TSource> *quelle*, **Func<TSource, bool>** *bedingung***)**

bzw. mit dem korrespondierenden Abfrage-Operator **where** wird eine Teilmenge der Quelldatensätze ausgewählt. Neben dem **this**-Parameter vom zu erweiternden Typ erwartet die 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 einen booleschen Rückgabewert, welche für jeden Datensatz der Quelle über die Aufnahme in die Ergebnisliste entscheidet.

Dieser Erweiterungsmethodenaufruf

```
var cust = customers
             .Where(c => c.Country == "Mexico")
             .Select(c => new {c.CustomerID, c.Country});
```

beschränkt das Ergebnisliste wie die äquivalente Abfragesyntax

```
var cust = from c in customers
             where c.Country == "Mexico"
             select new {c.CustomerID, c.Country};
```

auf Kunden in Mexiko.

22 ADO.NET Entity Framework

Eine typische Anwendung im Geschäftsbereich benötigt lesenden und schreibenden Zugriff auf eine mehr oder weniger komplexe Datenbank, deren Schema unabhängig von den Anwendung entstanden ist und potentiell aus Datenbank-internen Gründen ohne Rücksicht auf die Anwendung verändert werden könnte. Microsoft hat mit dem Entity Framework (EF) eine auf ADO.NET aufsetzende Technologie geschaffen, die Datenbank-abhängige Anwendungen durch zwei Übersetzungsleistungen unterstützt:

- Abschottung der Programmlogik gegenüber Besonderheiten und eventuellen Modifikationen im Schema der relationalen Datenbank, die zur Speicherung der zu verarbeitenden Daten dient und dabei Datenbank-bezogenen Optimierungen unterworfen ist. Mit dem Entity Data Model (EDM, siehe Abschnitt 22.1) wird eine anwendungsspezifische Sicht auf die Datenbank geschaffen. Das Programm arbeitet mit dem konzeptionellen EDM, das vom Entity Framework beim Lesen und Speichern auf das Schema der relationalen Datenbank abgebildet wird.
- Auch das EDM besteht noch aus Tabellen (und deren Beziehungen), während in einem modernen Software-Entwurf kooperierende Klassen und Objekte die Szene beherrschen. Die im EF enthaltenen Objektdienste (siehe Abschnitt 22.3) ermöglicht es, dass aus einer Datenbankabfrage statt Tabellenzeilen Objekte resultieren. Beim Sichern von veränderten oder neuen Objekten transportiert das EF deren Eigenschaften in die Datenbank. Damit überbrückt das EF die Kluft zwischen objektorientierter Programmierung und relationaler Datenspeicherung, ist also eine ORM-Technologie (*Object-Relational Mapping*).

Unter der Bezeichnung *LINQ to SQL* hat Microsoft noch eine zweite, etwas ältere ORM-Technologie für .NET-Entwickler im Angebot. In Vergleichen schneidet das EF eindeutig besser ab (siehe z.B. Schwichtenberg 2009b), und Microsoft hat sich dafür entschieden, das flexiblere, mächtigere, aber aucch mit einem größeren Lernaufwand verbundene Entity Framework bei der Weiterentwicklung zu bevorzugen, was z.B. die folgenden Stellungnahme belegt:[118]

> Moving forward, Microsoft is committing to supporting both technologies as important parts of the .NET Framework, adding new features that meet customer requirements. We do, however, expect that the bulk of our overall investment will be in the Entity Framework, as this framework is built around the Entity Data Model (EDM). EDM represents a key strategic direction for Microsoft that spans many of our products, including SQL Server, .NET, and Visual Studio.

Beachten Sie bitte, dass diese negative Prognose die ORM-Technik LINQ to SQL betrifft, aber *keinesfalls* die generelle Abfragetechnik LINQ, die auch im Zusammenhang mit dem EF eine wichtige Rolle spielt (siehe Abschnitt 22.3.2).

Traditionell arbeitet das EF in der Betriebsart *Database First*, wobei zu einem bestehenden Datenbankschema ein EDM und zugehörige Klassen entstehen. In der brandneuen Version 4.1 (Stand: April 2011) beherrscht das EF auch die Betriebsart *Code First*, wobei zuerst in C# .NET - Klassen entstehen, die dann auf eine existierende Datenbank abgebildet oder zum Erstellen einer neuen Datenbank verwendet werden.

Das EF stützt sich auf die ADO.NET - Provider, so dass diverse Datenbankmanagementsysteme verwendet werden können. Zwar erfordert das EF eine Erweiterung des ADO.NET - Providers, doch sollte diese mittlerweile in der Regel verfügbar sein (z.B. für IBM DB2, MySQL, Oracle, PostgreSQL).

Wer das EF in der Praxis einsetzen möchte, benötigt über das aktuelle Kapitel hinaus zusätzliche Informationen, die z.B. das Buch von Lerman (2010) bietet.

---

[118] Webseite http://msdn.microsoft.com/en-us/data/bb525059#Q3, abgefragt am 22.07.2011

22.1 Entity Data Model

Anwendungsentwickler treffen oft auf eine komplexe und durch Normalisierung optimierte Datenbank, welche die Ein- und Ausgabedaten des geplanten Programms (z.B. Kunden. Aufträge) permanent und effizient speichert. Bei der Aufgabeanalyse kann es z.B. passieren, dass eine Informationseinheit aus der Sicht des Programms (z.B. die Daten eines Kunden) nicht auf eine einzelne Tabelle abzubilden ist, sondern Daten aus mehreren Tabellen einbezieht. Zum Verteilen der Kundendaten auf mehrere, in einer 1:1 - Relation stehenden, Tabellen können z.B. Sicherheitsargumente (Differenzierung von Zugriffsrechten) geführt haben. Mit einer Datenzugriffstechnik wie ADO.NET müssen dann die Daten der Informationseinheiten über mehr oder weniger komplexe **SELECT**-Kommandos aus der Datenbank importiert werden. Beim Speichern müssen die Daten einer Informationseinheit wieder auf die einzelnen Tabellen der Datenbank verteilt werden. Besonders übel wird es, wenn sich die Administratoren der Datenbank entscheiden, deren Schema zu ändern, weil nun im Programm diverse Anpassungen notwendig werden.

Es ist anzustreben, dem Programm eine Datenstruktur mit problemadäquaten Informationseinheiten anzubieten, um den Programmcode einfach zu halten und gegenüber Änderungen der Datenbank abzuschotten. Genau dies leistet das Entity Framework mit seiner Abbildung der Datenbankschemas (aus Tabellen, Beziehungen und Restriktionen) auf das konzeptionelle Schema (aus Entitäten, Beziehungen und Restriktionen). Unter einer **Entität** ist eine problemadäquate Informationseinheit zu verstehen, also im Wesentlichen eine Zeile aus einer problemadäquaten Tabelle mit passend zusammengestellten Spalten. Wir haben also die Welt aus rechteckigen Tabellen, Beziehungen (z.B. Master-Details) und Restriktionen (z.B. Nullverbot für eine Spalte) nicht verlassen, sondern nur problemadäquat umgestaltet und gegenüber Veränderungen des Datenbankschemas abgeschottet. Mit Hilfe der Object Services im EF (siehe Abschnitt 22.3) lassen sich die Entitäten in Objekte umsetzen, welche eine entsprechende Merkmalsausstattung besitzen, aber keine (über typunabhängige Methoden) hinausgehenden Verhaltenskompetenzen.

Das Entity Framework leistet die problemadäquate Datenabstraktion durch zwei Schemadefinitionen und eine vermittelnde Abbildung, wobei jeweils zur Deklaration ein XML-Dialekt Verwendung findet:

- **Konzeptionelles Modell**
  Hier ist das **Entity Data Model** (EDM) aus problemadäquaten Entitäten und deren Beziehungen angesiedelt. Zum Beschreibung kommt die *Conceptual Schema Definition Language* (CSDL) zum Einsatz. Man kann das EDM als Microsofts Variante des *Entity Relation Models* (ERM) bezeichnen.
- **Speichermodell (Logische Schicht)**
  Das Datenbankschema wird mit der *Store Schema Definition Language* (SSDL) beschrieben. Man spricht von der *logischen Schicht*, weil die reale Datenbankstruktur durch technischen Erfordernisse geprägt sein kann, die im aktuellen Kontext keine Rolle spielen.
- **Zuordnung**
  Die Abbildung zwischen dem Speichermodell und dem konzeptionellem Modell (EDM) wird in der Mapping Specification Language (*MSL*) beschrieben.

Zur Entwurfszeit werden die beiden Modelle und die Zuordnung in *einer* Datei mit der Namenserweiterung **.EDMX** gespeichert, doch zur Laufzeit entstehen daraus drei separate Dateien, die vom EF genutzt werden (Lerman 2010, S. 5)

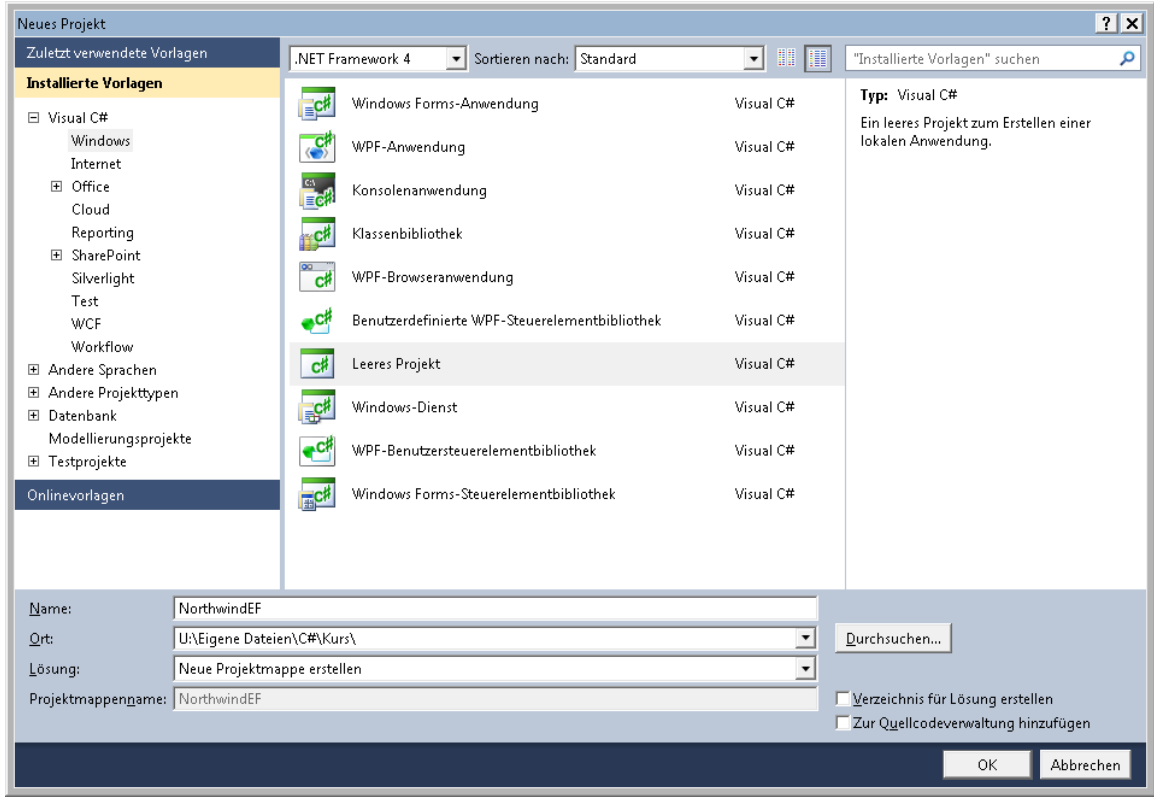
Zum Glück muss man die umfangreichen XML-Deklarationen nicht manuell erstellen, sondern kann einem Assistenten der Entwicklungsumgebung die Arbeit überlassen. U.a. bietet er die Möglichkeit, die Modelle aus einer vorhandenen Datenbank zu generieren, wobei dabei natürlich keine echte Abbildung stattfindet und zu jeder Tabelle eine Entität entsteht. Wir müssen aus Zeitgründen

auf eine Demonstration der Abbildungsleistung des Entity Frameworks verzichten, wollen aber doch einen Blick auf die Deklaration eines konzeptionellen Modells werfen.

22.1.1 EDM aus einer vorhandenen Datenbank generieren lassen

Wir verwenden wieder einmal die Tabellen **Customers** und **Orders** der Datenbank **Northwind** und lassen von dem im Visual Studio enthaltenen Entity Framework - Assistenten daraus ein EDM bzw. eine EDMX-Datei erstellen.

Legen Sie im Visual Studio ein neues Projekt unter Verwendung der Vorlage **Leeres Projekt** an. Wer eine kommerzielle Version der Entwicklungsumgebung verwendet, sollte darauf achten, dass die .NET - Framework - Version 4 eingestellt ist, die beim Entity Framework einige Erweiterungen gegenüber der Version 3.5 bietet, z.B.:



Alternativ lässt sich das Zielframework (auch in der Express Edition) über den Menübefehl

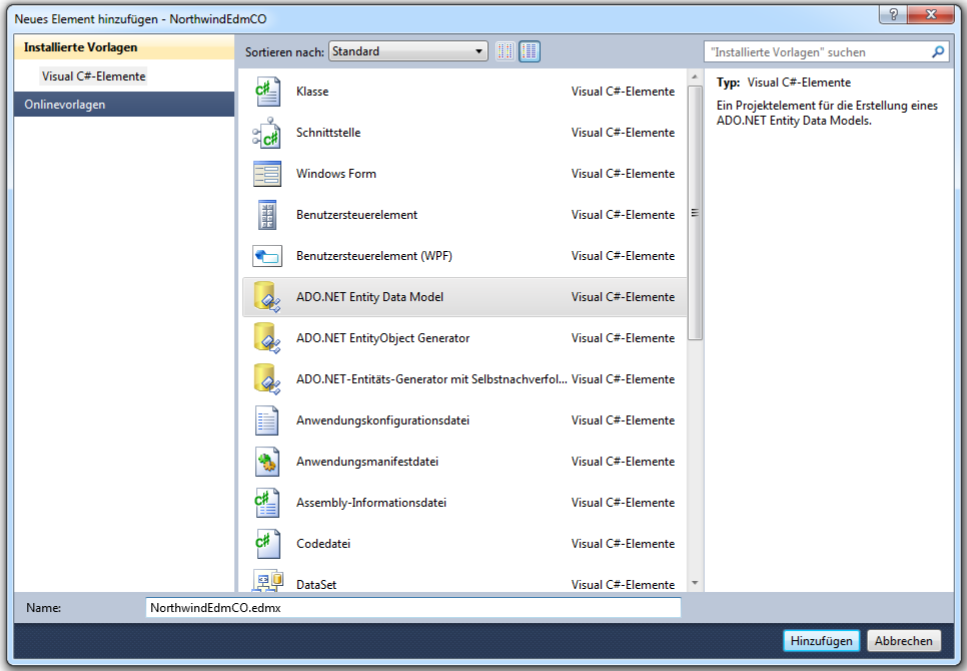
Projekt > Eigenschaften > Anwendung

einstellen.

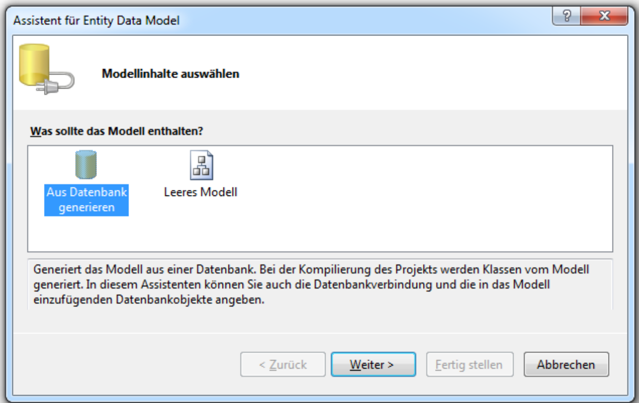
Erweitern Sie das neue Projekt über sein Kontextmenü mit dem Befehl

Hinzufügen > Neues Element

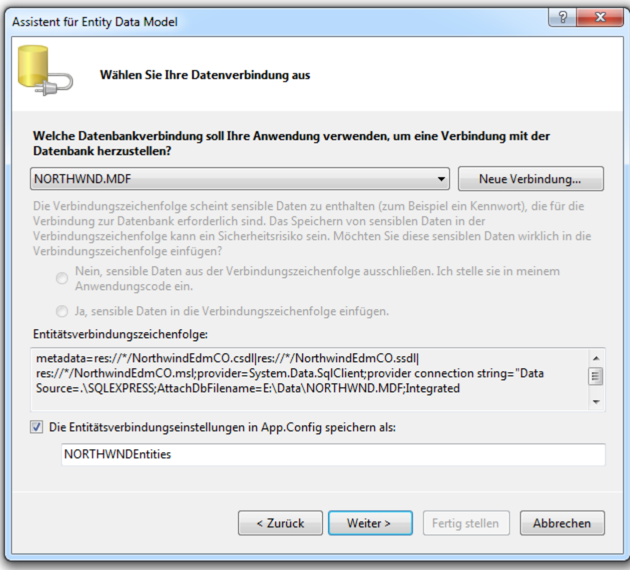
um ein **ADO.NET Entity Data Model**, z.B.:



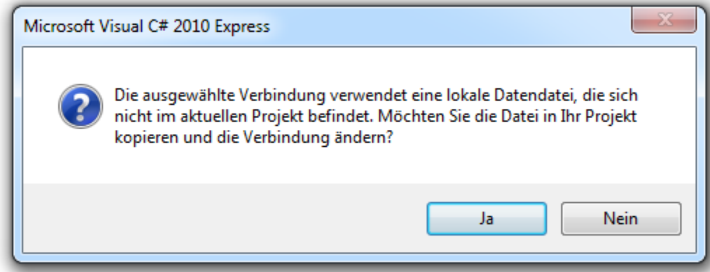
Wählen Sie die Alternative, das EDM **aus einer Datenbank zu generieren**:



Anschließend können Sie eine vorhandene, im Datenbank-Explorer der Entwicklungsumgebung angezeigte Verbindung zur Datenbank **Northwind** nutzen oder eine neue anlegen (vgl. Abschnitt 20.6.2.1), z.B.:



In Abhängigkeit vom Typ der Datenbankverbindung wird nachgefragt, ob die Datenbankdatei in den Projektordner kopiert werden soll:

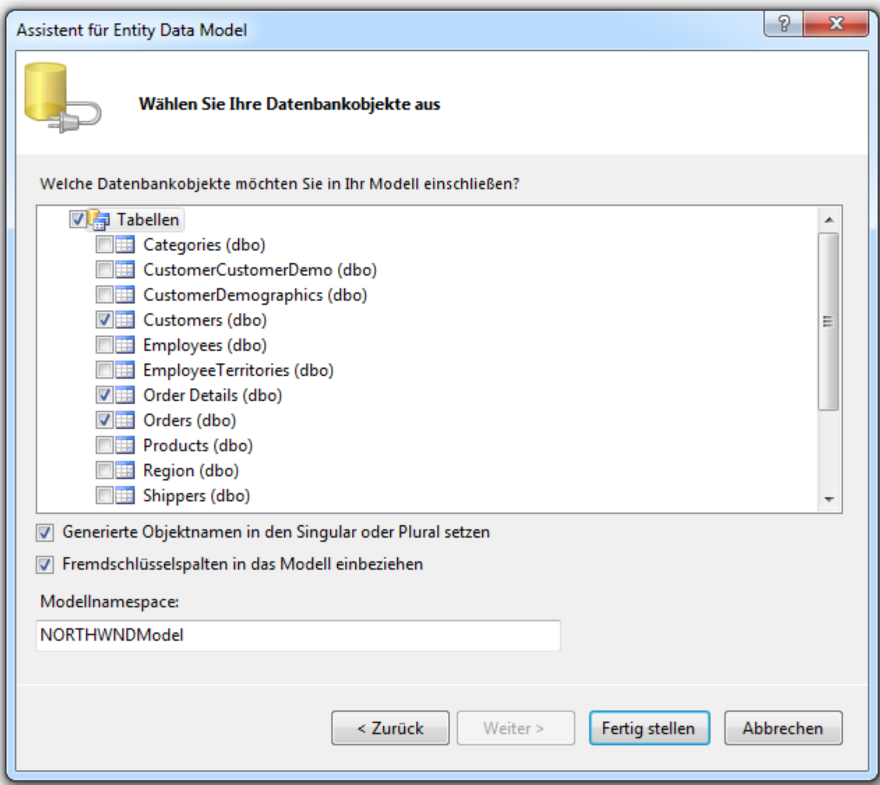


Im Beispiel wird nach einer Zustimmung in der **Entitätszeichenfolge** der feste Pfad zur Datenbankdatei durch eine relative Adressierung über das Makro **DataDirectory** ersetzt (vgl. Abschnitt 20.5.2.1.1.1):

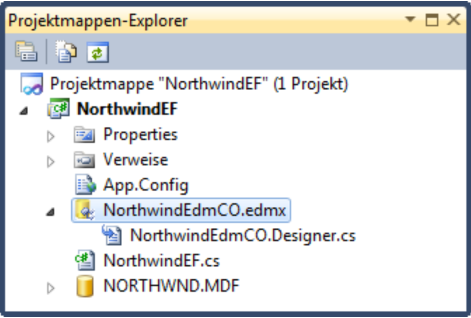
```
metadata=res://*/NorthwindEdmCO.csdl|res://*/NorthwindEdmCO.ssdl|
res://*/NorthwindEdmCO.msl;provider=System.Data.SqlClient;provider connection string="Data
Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\NORTHWND.MDF;Integrated
Security=True;Connect Timeout=30;User Instance=True"
```

Das Makro zeigt bei einer Windows-Anwendung per Voreinstellung auf den Ordner mit dem Exe-Assembly.

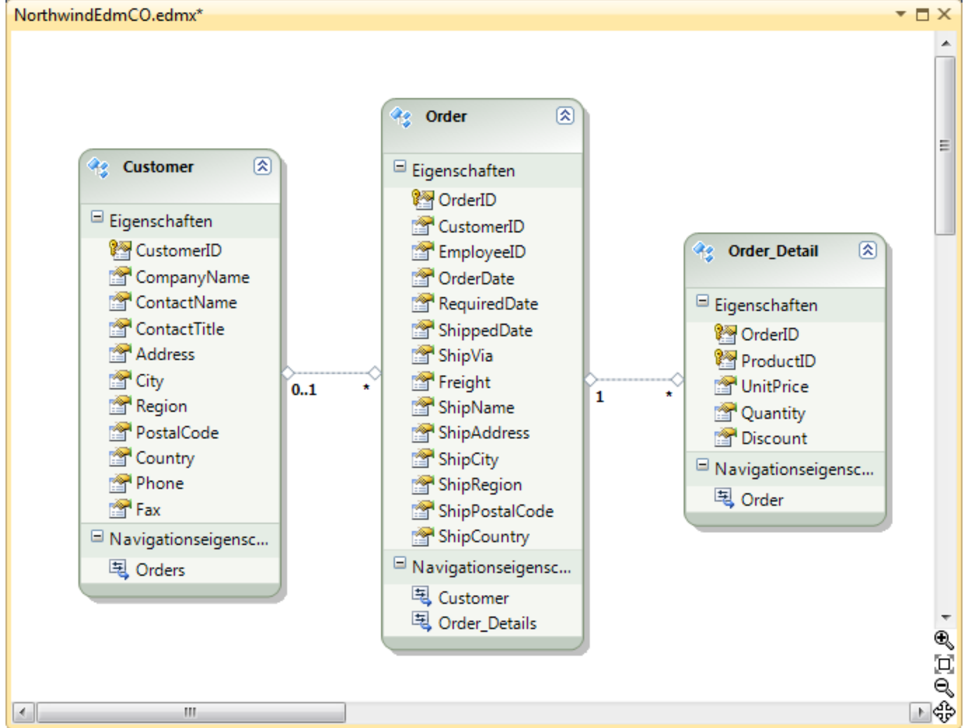
Bei der Auswahl der **Datenbankobjekte** beschränken wir uns auf die Tabellen **Customers**, **Orders** und **Order Details**. Um einer üblichen Praxis entsprechend Entitätsnamen im Singular (also `Customer` und `Order`) zu erhalten (vgl. Lerman 2010, S. 27), markieren wir das Kontrollkästchen **Generierte Objektnamen in den Singular oder Plural setzen**:



Nach einem Klick auf den Schalter **Fertig stellen** erscheinen im Projektmappen-Explorer das EDM und die Datenbankverbindung:

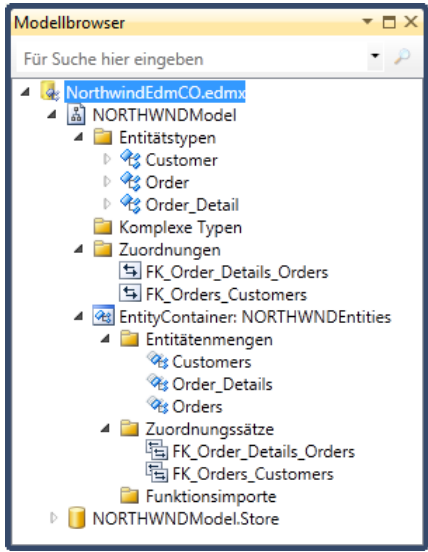


In der Designerzone erscheint ein GUI-Werkzeug zur Modifikation des Modells. Im Beispiel zeigt es zwei Entitäten mit Master-Details - Beziehung, die durch direkte Abbildung der Datenbanktabellen entstanden sind:



Weil diese Ansicht für eine EDMX-Datei voreingestellt ist, kann sie bei Bedarf per Doppelklick auf den Eintrag im Projektmappen-Explorer (im Beispiel: **NorthwindEdmCO.edmx**) aufgerufen werden.

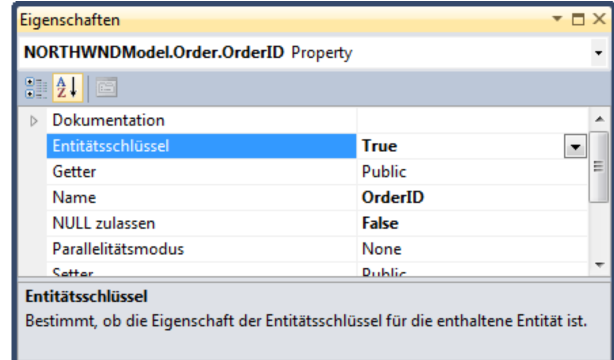
Über das Kontextmenü zum EDM-Designer kann man den **Modellbrowser** anfordern:

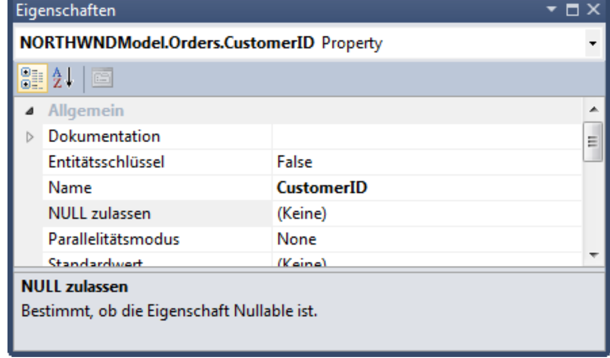


Man sieht alle EDM-Bestandteile und kann sie markieren, um die jeweiligen Eigenschaften zu inspizieren oder zu ändern. Neben dem konzeptionellen Modell (im Beispiel: `NORTHWNDModel`) wird hier auch das Speichermodell (im Beispiel: `NORTHWNDModel.Store`) angezeigt.

22.1.2 Entitäten mit (Navigations-)Eigenschaften und Assoziationen

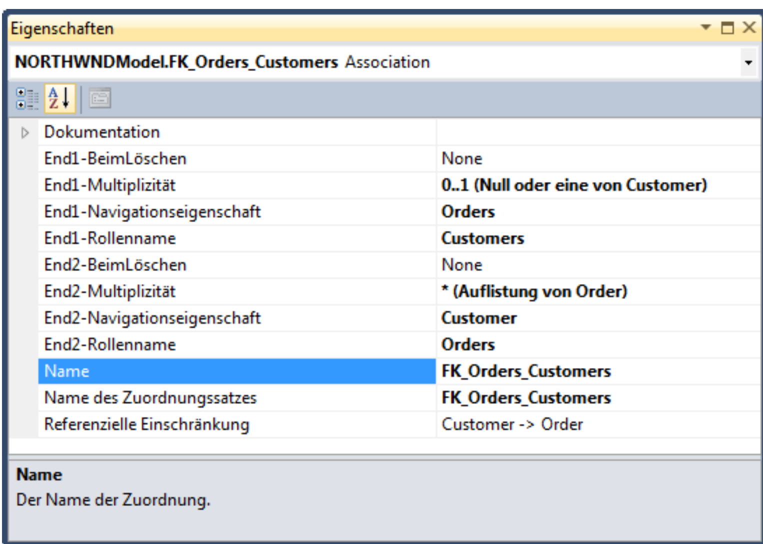
Aus den Datenbankfeldern sind skalare Eigenschaften der Entitäten hervorgegangen, z.B. die Eigenschaften `OrderID` und `CustomerID` bei der Entität `Order`. Aus dem Schema der Datenbank wurden zugehörige Beschreibungsmerkmale übernommen, die per EDM-Designer eingesehen und verändert werden können, z.B. bei den genannten `Order`-Eigenschaften:





Bei Entitäten vom Typ `Order` wird die Identität über die Eigenschaft `OrderID` festgelegt (`Entitätsschlüssel` = **True**).

Auch für die aus dem Datenbankschema abgeleitete Assoziation zwischen den beiden Entitäten `Order` und `Customer` sind (nach dem Markieren der Verbindungslinie) etliche Eigenschaften sicht- und manipulierbar:

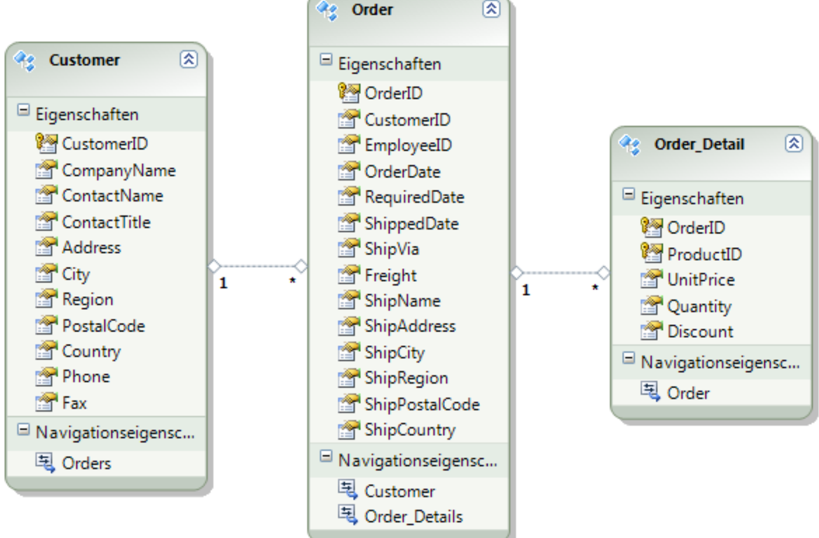


Für die beiden Entitäten als Endpunkte der Assoziation ist jeweils eine **Multiplizität** zu wählen, wobei die folgenden Werte bzw. Symbole zur Verfügung stehen:

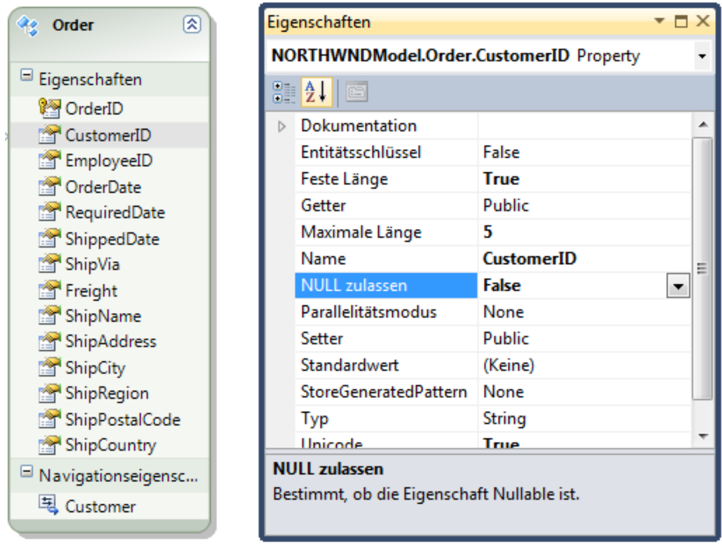
- **0..1** kein oder ein Objekt
- **1** genau ein Objekt
- ***** beliebig viele Objekte

Weil aufgrund des ausgewerteten Datenbankschemas für die `Order`-Eigenschaft `CustomerID` *kein* Nullverbot besteht, besitzt die Assoziation zwischen den beiden Entitäten als **End1-Multiplizität** den Wert **0..1**, so dass Aufträge ohne Kunden akzeptiert werden (eventuell vorgesehen für firmen-

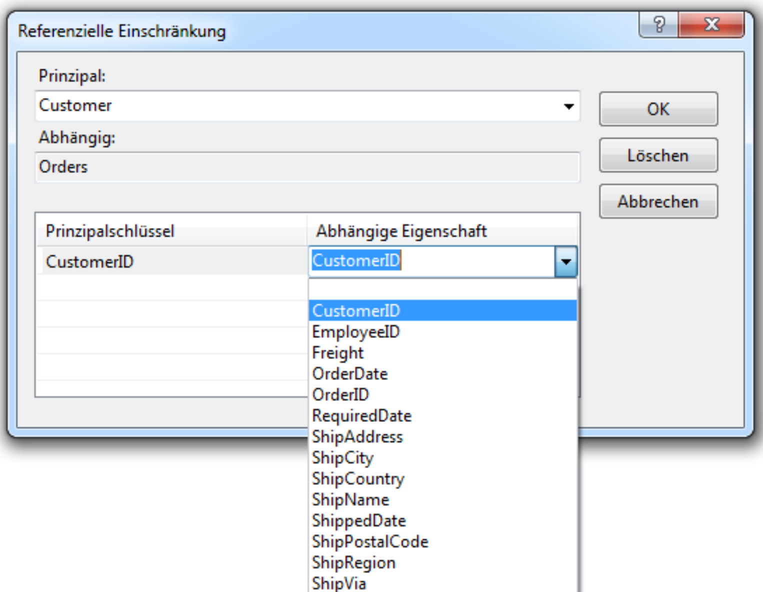
interne Aufträge). Um solche Aufträge zu verhindern, setzen wir die **End1-Multiplizität** der Assoziation per Eigenschaftsfenster auf den Wert **1** mit dem Ergebnis:



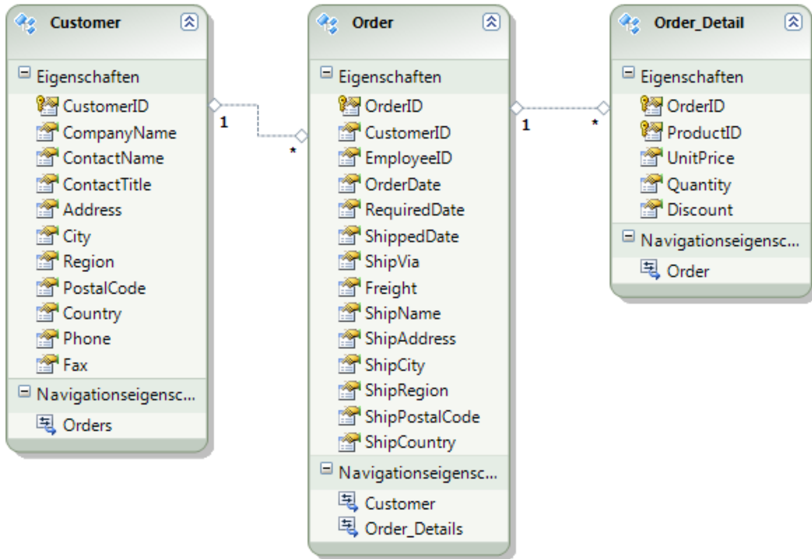
Damit im EDM keine Inkonsistenz entsteht, welche z.B. das automatische Erstellen von Klassen behindert, muss nun für die beteiligte `Order`-Eigenschaft `CustomerID` ein Nullverbot ausgesprochen werden:



Wir haben beim Generieren des EDMs aus der **Northwind**-Datenbank (siehe in Abschnitt 22.1.1) die Voreinstellung des Assistenten beibehalten, Fremdschlüssel einzubeziehen. Folglich spielt der Fremdschlüssel `CustomerID` in der Entität `Order` eine entscheidende Rolle bei der Definition der Assoziation. Über die Assoziationseigenschaft **referentielle Einschränkung** ist der folgende Dialog zur Gestaltung der Fremdschlüssel-Abhängigkeit zugänglich:



Bei einer Beziehungslinie zwischen zwei Entitäten bemüht sich der EDM-Designer übrigens *nicht* um „korrekte" Endpunkte. Wer auf solche Feinheiten Wert legt, muss Hand- bzw. Mausarbeit investieren, z.B.:



Weil zwischen den Entitäten Assoziationen bestehen, besitzen sie neben ihren skalaren Eigenschaften auch **Navigationseigenschaften**. Infolgedessen kann z.B. ein Objekt vom Entitätstyp `Customer` über das Navigationsmerkmal `Orders` auf eine Kollektion mit den zugehörigen `Order`-Objekten zugreifen. Umkehrt bringt ein Objekt vom Entitätstyp `Order` über das Navigationsmerkmal `Customer` den zugehörigen Kunden in Erfahrung.

22.1.3 CSDL-Elemente

Nachdem wir bisher den graphischen EDM-Designer verwendet haben, werfen wir nun einen Blick auf den erzeugten XML-Code. Im Beispiel basierend auf den beiden **Northwind**-Tabellen **Customers** und **Orders** ist eine EDMX-Datei mit der folgenden Deklaration des konzeptionellen Modells erstellt worden:

```
<edmx:ConceptualModels>
  <Schema Namespace="NORTHWNDModel" Alias="Self"
          xmlns:annotation="http://schemas.microsoft.com/ado/2009/02/edm/annotation"
          xmlns="http://schemas.microsoft.com/ado/2008/09/edm">
    <EntityContainer Name="NORTHWNDEntities" annotation:LazyLoadingEnabled="true">
      <EntitySet Name="Customers" EntityType="NORTHWNDModel.Customer" />
      <EntitySet Name="Order_Details" EntityType="NORTHWNDModel.Order_Detail" />
      <EntitySet Name="Orders" EntityType="NORTHWNDModel.Order" />
      <AssociationSet Name="FK_Orders_Customers"
                      Association="NORTHWNDModel.FK_Orders_Customers">
        <End Role="Customers" EntitySet="Customers" />
        <End Role="Orders" EntitySet="Orders" />
      </AssociationSet>
      <AssociationSet Name="FK_Order_Details_Orders"
                      Association="NORTHWNDModel.FK_Order_Details_Orders">
        <End Role="Orders" EntitySet="Orders" />
        <End Role="Order_Details" EntitySet="Order_Details" />
      </AssociationSet>
    </EntityContainer>
    <EntityType Name="Customer">
      <Key>
        <PropertyRef Name="CustomerID" />
      </Key>
      <Property Name="CustomerID" Type="String" Nullable="false" MaxLength="5"
                Unicode="true" FixedLength="true" />
      <Property Name="CompanyName" Type="String" Nullable="false" MaxLength="40"
                Unicode="true" FixedLength="false" />
      . . .
      <Property Name="Fax" Type="String" MaxLength="24" Unicode="true"
                FixedLength="false" />
      <NavigationProperty Name="Orders" Relationship="NORTHWNDModel.FK_Orders_Customers"
                          FromRole="Customers" ToRole="Orders" />
    </EntityType>
    <EntityType Name="Order_Detail">
      . . .
    </EntityType>
    <EntityType Name="Order">
      . . .
    </EntityType>
    <Association Name="FK_Orders_Customers">
      <End Role="Customers" Type="NORTHWNDModel.Customer" Multiplicity="1" />
      <End Role="Orders" Type="NORTHWNDModel.Order" Multiplicity="*" />
      <ReferentialConstraint>
        <Principal Role="Customers">
          <PropertyRef Name="CustomerID" />
        </Principal>
        <Dependent Role="Orders">
          <PropertyRef Name="CustomerID" />
        </Dependent>
      </ReferentialConstraint>
    </Association>
    <Association Name="FK_Order_Details_Orders">
      . . .
    </Association>
  </Schema>
</edmx:ConceptualModels>
```

Um den XML-Code im Editorbereich der Entwicklungsumgebung zu öffnen, wählt man aus dem Kontextmenü des EDMX-Eintrags im Projektmappen-Explorer den Befehl

Öffnen mit > XML (Text)-Editor

Zur Definition eines konzeptionellen Modells stehen folgende CSDL-Elemente zur Verfügung:

- **EntityContainer**
  Der **EntityContainer** nimmt **EntitySets** and **AssociationSets** auf (siehe unten).
- **EntitySet**
  Dies ist ein Container für Entitäten von einem Entitätstyp oder seinen Ableitungen.
- **EntityType**
  Dieses Element deklariert einen Datentyp, gekennzeichnet durch eine Reihe von (Navigations-)Eigenschaften. Außerdem ist ein **Key**-Element enthalten, das die identifizierenden Eigenschaften des Entitäts-Typs deklariert. Im Eigenschaftsfenster des EDM-Designers sind die identifizierenden Eigenschaften am **Entitätsschlüssel**-Wert **True** zu erkennen (siehe Abschnitt 22.1.2).
- **AssociationSet**
  Dies ist ein Container für Assoziationsinstanzen, die jeweils aus einem Paar von zugeordneten Entitäten bestehen.
- **Association**
  Dieses Element definiert eine Assoziation durch die Bedingungen, die ein Paar von zugeordneten Entitäten erfüllen muss.

22.2 Der Provider EntityClient

Das EF bringt mit dem **EntityClient** einen Datenprovider, der analog zu den ADO.NET - Providern (z.B. **SqlClient**, vgl. Abschnitt 20.5) arbeitet und analoge Klassen enthält (z.B. **EntityCommand**, **EntityConnection**, **EntityDataReader**). Bei der Arbeit mit diesem Provider greift man nicht auf die Tabellen der realen Datenbank zu, sondern auf die Entitäten und Beziehungen im EDM, wobei der **EntityClient** mit Hilfe des zugrunde liegenden ADO-NET- Providers die Entitätszugriffe in Datenbankzugriffe umsetzt.

Der **EntityClient** unterstützt zwei Arbeitsweisen:

- Direkte Verwendung über den providerunabhängigen SQL-Dialekt **Entity SQL** (eSQL)
  Man definiert wie beim Einsatz eines ADO.NET - Providers ein SELECT-Kommando und erhält über die **EntityCommand** - Methode **ExecuteReader**() einen **EntityDataReader**, der analog zu einem **SqlDataReader** (siehe Abschnitt 20.5.8) genutzt werden kann. eSQL enthält (noch) keine MDL-Kommandos, so dass *keine Änderungen* an den Entitäten (und letztlich an den Datenbanktabellen) möglich sind. Dementsprechend bietet der Provider **EntityClient** *kein* Analogon zum **DataAdapter** der ADO.NET - Provider.
- Indirekte Verwendung über die Object Services im EF
  Bei der im Normalfall empfehlenswerten indirekten Verwendung des **EntityClient**-Providers via Object Services (siehe Abschnitt 22.3) werden die vom Provider **EntityClient** gelieferten Tabellenzeilen in Objekte gewandelt. Man kann nicht nur ohne Stilbruch objektorientiert programmieren, sondern auch Änderungen an den Daten vornehmen.

Wenn keine Änderungen an der Datenbank und keine Umsetzung von Datenbankzeilen in Objekte erforderlich ist, kann der Direktzugriff auf den **EntityClient** eine sinnvolle Option sein, seil sich die im Textformat vorliegenden eSQL-Abfragen zur Laufzeit dynamisch generieren lassen. Aufgrund des providerunabhängigen SQL-Dialekts können beim Wechsel des Datenbankmanagementsystems keine Probleme durch verschieden implementierte **SELECT**-Kommandos auftreten. In Programmen mit komplexer Geschäftslogik ist es allerdings sehr zu empfehlen, die Entitäten durch Objekte zu repräsentieren.

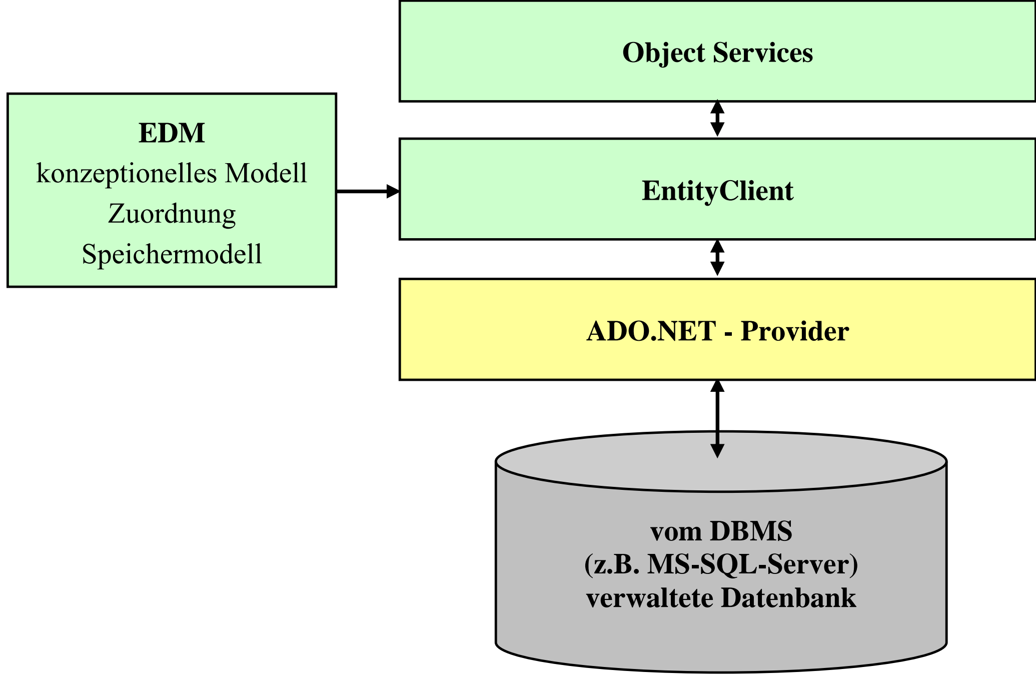
Das EF übernimmt in jedem Fall die Kommunikation mit dem RDBMS, so dass im Quellcode keine lästigen Routinearbeiten wie das Öffnen und Schließen von Datenbankverbindungen erforderlich sind.

22.3 Object Services

Durch die Objektdienste im ADO.NET Entity Framework wird endlich der Paradigmenbruch zwischen objektorientierter Programmierung einerseits und der Bearbeitung von Tabellenzeilen andererseits behoben, den die angelsächsische Literatur oft als *impedance mismatch* bezeichnet. Aus den vom Provider **EntityClient** bereit gestellten Entitäten (Zeilen von problemadäquaten Tabellen) werden Objekte. Wie wir es erwarten dürfen, erzeugen die EF-Werkzeuge (z.B. als Bestandteile im Visual Studio) automatisch die erforderlichen Klassen, welche die konzeptionelle EDM-Schicht in der objektorientierten Welt repräsentieren. Dabei entstehen partielle Klassendefinitionen, so dass Sie Erweiterungen vornehmen können, ohne mit dem Quellcode-Generator zu kollidieren.

Die Objektdienste verfolgen Änderungen bei den betreuten Objekten und schreiben diese nach Auforderung in Kooperation mit dem EF- und ADO-NET - Unterbau über automatisch erzeugte **INSERT**-, **UPDATE**- und **DELETE** - Kommandos in die Datenbank (Lerman 2010, S. 14). Damit aktualisierbare Objekte entstehen, dürfen Sie allerdings beim Erstellen über den **EntityClient** keine Projektion (vgl. Abschnitt 21.2.2) verwenden, weil dabei unvollständige Entitäten resultieren, deren Änderungen nicht gesichert werden können (Lerman 2010, S. 130).

An dieser Stelle soll eine vereinfachte Darstellung der im Manuskript behandelten EF-Architektur präsentiert werden

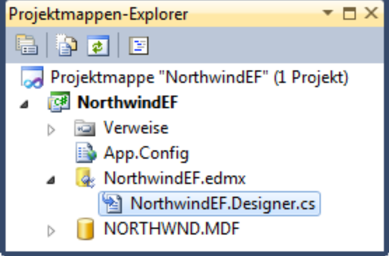


22.3.1 Klassen und Objekte zu einem EDM

Der **EntityContainer** im EDM (siehe Abschnitt 22.1) wird durch eine von **ObjectContext** abgeleitete Klasse repräsentiert (im Beispiel: `NORTHWNDEntities`):

```
public partial class NORTHWNDEntities : ObjectContext {
    . .
}
```

Man findet ihren automatisch erstellten Quellcode in einer Datei, die dem EDM-Eintrag im Projektmappen-Explorer untergeordnet ist:



Weil die Basisklasse **ObjectContext** die **ObjectQuery<T>**-Schnittstelle implementiert, sind Abfragen in eSQL und LINQ möglich. Wir werden aber nur die empfohlene Abfragetechnik LINQ to Entities verwenden (siehe Abschnitt 22.3.2).

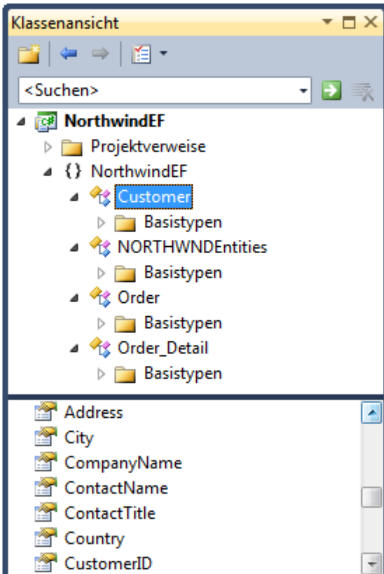
Weiterhin entsteht für jeden Entitätstyp im EDM eine .NET - Klasse, die von der Basisklasse **EntityObject** erbt. Im Beispiel erhalten wir also die Klassen `Customer`, `Order` und :

```
public partial class Customer : EntityObject {
  . . .
}

public partial class Order : EntityObject {
  . . .
}

public partial class Order_Detail : EntityObject EntityObject {
  . . .
}
```

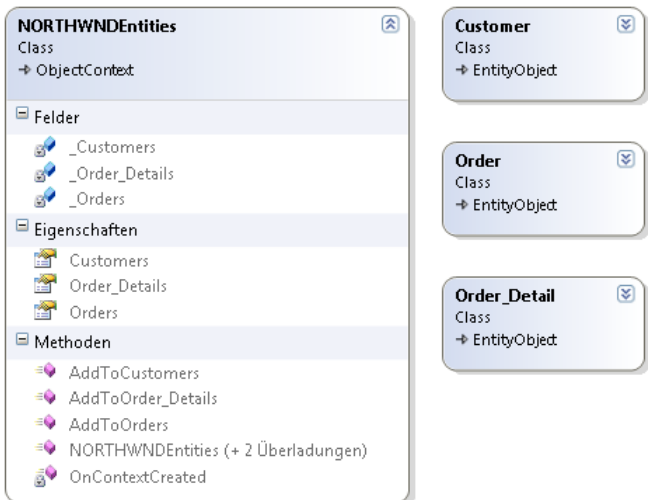
Wird über den Menübefehl

Ansicht > Klassenansicht

die Klassenansicht der Entwicklungsumgebung aktiviert, sind dort die drei erzeugten Klassen mit Basistypen und Innenausstattung zu sehen:



Wer eine kostenpflichtige Version der Entwicklungsumgebung benutzt, kann über das Kontextmenü zum Eintrag des Namensraums `NorthwindEF` in der Klassenansicht das folgende Klassendiagramm anfordern:



Zu jeder **Property** eines Entitätstyps besitzt die zugehörige Klasse ein privates Feld und eine öffentliche Eigenschaft. In der Klasse `Customer` finden sich z.B. zurück gehend auf die **Property** `CustomerID`

```
<Property Name="CustomerID" Type="String" Nullable="false" MaxLength="5"
          Unicode="true" FixedLength="true" />
```

die Eigenschaftsdefinition:

```
[EdmScalarPropertyAttribute(EntityKeyProperty=true, IsNullable=false)]
[DataMemberAttribute()]
public global::System.String CustomerID
{
    get
    {
        return _CustomerID;
    }
    set
    {
        if (_CustomerID != value)
        {
            OnCustomerIDChanging(value);
            ReportPropertyChanging("CustomerID");
            _CustomerID = StructuralObject.SetValidValue(value, false);
            ReportPropertyChanged("CustomerID");
            OnCustomerIDChanged();
        }
    }
}
```

und die Felddeklaration:

```
private global::System.String _CustomerID;
```

Zu jeder Navigationseigenschaft eines Entitätstyps besitzt die zugehörige Klasse eine öffentliche Eigenschaft. In der Klasse `Customer` findet sich z.B. zurückgehend auf die Navigationseigenschaft `Orders`

```
<NavigationProperty Name="Orders" Relationship="NORTHWNDModel.FK_Orders_Customers"
                    FromRole="Customers" ToRole="Orders" />
```

die folgende Eigenschaftsdefinition:

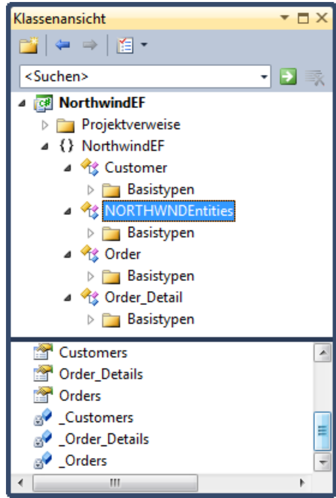
```
[XmlIgnoreAttribute()]
[SoapIgnoreAttribute()]
[DataMemberAttribute()]
[EdmRelationshipNavigationPropertyAttribute("NORTHWNDModel", "FK_Orders_Customers",
                                            "Orders")]
public EntityCollection<Order> Orders
{
    get
    {
        return ((IEntityWithRelationships)this).RelationshipManager.
          GetRelatedCollection<Order>("NORTHWNDModel.FK_Orders_Customers", "Orders");
    }
    set
    {
        if ((value != null))
        {
            ((IEntityWithRelationships)this).RelationshipManager.
                 InitializeRelatedCollection<Order>(
                     "NORTHWNDModel.FK_Orders_Customers", "Orders", value);
        }
    }
}
```

Ein Objekt der Klasse `Customer` liefert über die Eigenschaft `Orders` eine Kollektion mit allen `Order`-Objekten, die ihm zuzurechnen sind. So kann die aus dem Datenbankschema übernommene Beziehung zwischen den Tabellen **Customers** und **Orders** in elegante objektorientierte Programmierung umgesetzt werden.

Ansonsten werden die im EDM definierten Assoziationen im Assistenten-Quellcode jeweils durch ein Assembly-Attribut vom Typ **EdmRelationshipAttribute** berücksichtigt, z.B.:

```
[assembly: EdmRelationshipAttribute("NORTHWNDModel", "FK_Orders_Customers", "Customers",
   System.Data.Metadata.Edm.RelationshipMultiplicity.One, typeof(NorthwndEF.Customers),
   "Orders", System.Data.Metadata.Edm.RelationshipMultiplicity.Many,
   typeof(NorthwndEF.Orders))]
```

Das EF verwaltet für jeden Entitätstyp ein **EntitySet** als Container zur Aufnahme von Objekten, die vom Entitätstyp oder von einem abgeleiteten Typ sind. In diesem Kollektionsobjekt landen die per Abfrage über den Provider **EntityClient** importierten oder im Programm neu angelegte Entity-Objekte. In der vom EF-Werkzeug erstellten **ObjectContext** - Klasse, die den **EntityContainer** zu einem EDM repräsentiert, findet sich zu jedem Entity Set ein privates Objekt vom Typ **ObjectSet<TEntity>** und eine zugehörige öffentliche Eigenschaft. In unserem Beispiel mit der **ObjectContext** - Klasse `NORTHWNDEntities` sind die drei Eigenschaften `Customers`, `Order_Details` und `Orders` mit den zugehörigen Feldern vorhanden:



Hier ist der Quellcode zum **EntitySet** `Customers` zu sehen (aus der Datei **NorthwindEF.Designer.cs**):

```
public ObjectSet<Customer> Customers {
    get {
        if ((_Customers == null)) {
            _Customers = base.CreateObjectSet<Customers>("Customers");
        }
        return _Customers;
    }
}
private ObjectSet<Customers> _Customers;
```

22.3.2 Objekte per LINQ to Entities erstellen und verwenden

Um das Erstellen und Verwenden von EDM-Objekten üben zu können, erweitern wir das in Abschnitt 22.1.1 gestartete Projekt um die leere Quellcodedatei **NorthwindEF.cs** mit dem folgenden Programmrahmen:

```
using System;
using NorthwindEF;
using System.Linq;

class Program {
   static void Main() {
      using (NORTHWNDEntities context = new NORTHWNDEntities()) {
      .  .  .
      }
   }
}
```

Wir erzeugen zunächst einen Objekt-Kontext aus der vom Visual Studio definierten Klasse `NORTHWNDEntities`:

```
NORTHWNDEntities context = new NORTHWNDEntities();
```

Seine Eigenschaft `Customers` ist vom Typ `ObjectSet<Customers>` und damit eine geeignete Datenquelle für eine LINQ-Abfrage. Im folgenden Beispiel wird eine Kollektion mit allen Kunden aus Germany angefordert:

```
var custColl = from c in context.Customers
            where c.Country == "Germany"
            select new {c.CustomerID, c.CompanyName, c.Orders};
```

Die Objekte der Kollektion sind von einem anonymen Typ, der u.a. die Navigationseigenschaft `Orders` enthält, so dass wir sehr elegant (insbesondere ohne komplexes **SELECT**-Kommando) auf die Aufträge eines Kunden zugreifen können, z.B.:

```
foreach (var c in custColl) {
   Console.WriteLine(c.CustomerID + " " + c.CompanyName);
   foreach (var ord in c.Orders)
      Console.WriteLine(" " + ord.OrderDate);
}
```

Weil die Objekte der Kollektion `custColl` von einem anonymen Typ sind, können Änderungen bei diesen Objekten *nicht* durch die EF-Objektdienste zur Datenbank übertragen werden. Soll dies möglich sein, müssen *komplette* Entitätsobjekte erstellt werden. Aus der folgenden Abfrage resultiert genau ein Objekt vom Entitätstyp `Customer`, falls in der Datenbanktabelle **Customers** eine Zeile mit der angeforderten `CustomerID` vorhanden ist:

```
String custID = "WANDK";
Customer cust = null;
try {
   cust = context.Customers.First<Customer>(c => c.CustomerID == custID);
} catch {
   Console.WriteLine("CustomerID " + custID + " nicht vorhanden");
}
```

Die Eigenschaft **EntityState** des neu erzeugten Objekts hat den Wert **Unchanged**, was die folgende Ausgabeanweisung

```
Console.WriteLine(cust.EntityState + " \t " + cust.CustomerID +
                  " \t " + cust.CompanyName);
```

zeigt:

```
Unchanged        WANDK   Die Wandernde Kuh
```

Nach einer Änderung

```
cust.CompanyName = "Die Wabernde Kuh";
```

liefert dieselbe Ausgabeanweisung:

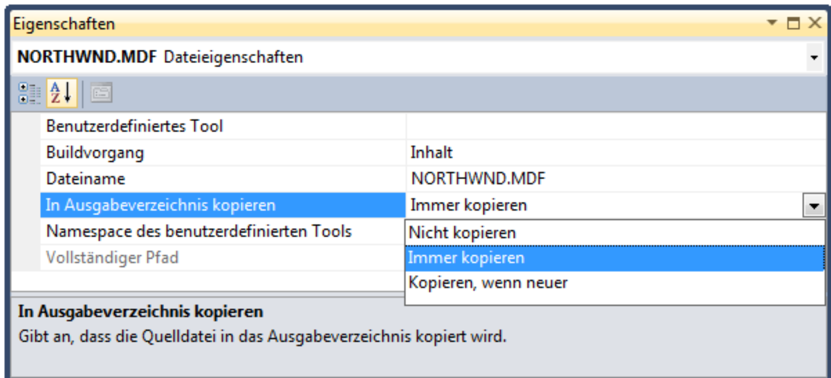
```
Modified         WANDK   Die Wabernde Kuh
```

Mit der Methode **SaveChanges()** wird der Objektkontext beauftragt, die Datenbank zu aktualisieren:

```
context.SaveChanges();
```

Bei unserem Beispielprojekt befindet sich die Datenbankdatei **NORTHWND.MDF** im Projektordner (vgl. Abschnitt 22.1.1). Bei einer solchen Projektorganisation verwaltet die Entwicklungsumgebung *zwei* Versionen der Datenbankdatei:

- Die vom Programm genutzte Version befindet sich (zusammen mit dem Assembly) im Ausgabeverzeichnis des Projekts, bei aktiver Debug-Konfiguration also in **...\bin\Debug**.
- Die auf der Hauptebene des Projektordners befindliche Version der Datenbankdatei wird vom Visual Studio bei Bedarf in das Ausgabeverzeichnis kopiert:
  - wenn sie dort fehlt,
  - wenn das Projekt neu erstellt wird, z.B. nach einer Quellcode-Änderung
    Für diesen Kopieranlass ist eine Einstellung der Entwicklungsumgebung zur Datenbankdatei relevant:



Per Voreinstellung wird **immer kopiert**, d.h. bei jedem Erstellen des Projekts.

In der Regel ist in der Projektentwicklungsphase das automatische Zurücksetzen der Datenbankdatei auf den Ausgangszustand von Vorteil. Wenn es stört, kann es leicht unterbunden werden.

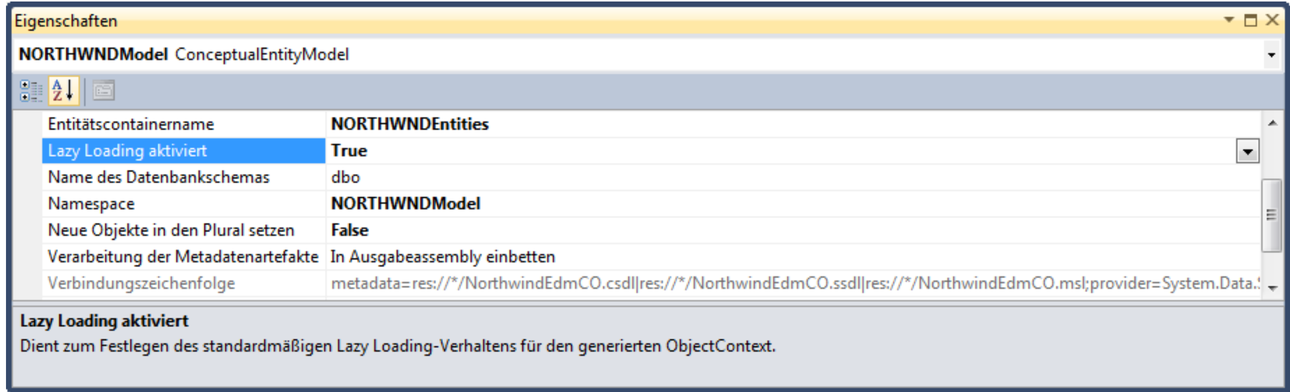
Durch die folgenden Anweisungen werden für das zuvor kreierte `Customer`-Entitätsobjekt aus der per Navigationseigenschaft `Orders` ansprechbaren Kollektion mit den zugehörigen `Order`-Entitätsobjekten alle Aufträge mit einem Datum vor dem 1.1.1998 gelöscht:

```
int i = 0;
int last = cust.Orders.Count - 1;
while (i <= last) {
   if (DateTime.Compare((DateTime)cust.Orders.ElementAt(i).OrderDate,
                        new DateTime(1998, 1, 1)) < 0) {
      cust.Orders.ElementAt(i).Order_Details.Clear();
      context.DeleteObject(cust.Orders.ElementAt(i));
      last--;
   } else
      i++;
}
```

Zu jedem Order-Objekt müssen auch die zugehörigen `Order_Detail`-Objekte gelöscht werden, weil anderenfalls bei der Datenbankaktualisierung per **SaveChanges**() die verletzte referentielle Integrität der Datenbank zu einem Ausnahmefehler führen würde.

22.3.3 Lazy Loading

Sind Entitäten über Assoziationen und zugehörige Navigationseigenschaften verknüpft, sollten beim Zugriff auf eine Navigationseigenschaft eines bereits geladenen Objekts die verknüpften Objekte, sofern sie sich noch nicht im Objektkontext befinden, automatisch geladen werden. Dieses *Lacy Loading* (deutsch: *spitzenmäßiges Laden*) beherrscht das Entity Framework seit der Version 4. Bei einem mit Assistentenhilfe erzeugten EDM (siehe Abschnitt 22.1.1) ist es per Voreinstellung aktiviert:



Daher sind im Beispielprojekt für ein Objekt von Entitätstyp `Customer` nach einem Zugriff auf seine Navigationseigenschaft `Orders`

```
Customer cust = context.Customers.First<Customer>(c => c.CustomerID == "WANDK");
Console.WriteLine("Lacy-Loading: " + context.ContextOptions.LazyLoadingEnabled +
   "\nAufträge des Kunden: " + cust.Orders.Count);
```

auch die zugehörigen Aufträge geladen:

```
Lacy-Loading: True
Aufträge des Kunden: 10
```

Bei abgeschaltetem Lacy Loading

```
context.ContextOptions.LazyLoadingEnabled = false;
```

unterbleibt das automatische Laden:

```
Lacy-Loading: False
Aufträge des Kunden: 0
```

22.4 *Entity Framework im Vergleich zu traditionellen ADO.NET - Techniken*

Zumindest bei komplexeren Projekten mit Persistenzschicht sollte eine ORM-Lösung gegenüber traditionellen ADO.NET - Techniken (z.B. typisiertes DataSet) bevorzugt werden, wobei von den

beiden ORM-Lösungen aus dem Hause Microsoft eindeutig das Entity Framework die besseren Zukunftschancen hat als der Konkurrent LINQ to SQL.

Von den zahlreichen Vorteilen des Entity Frameworks sollen noch einmal genannt werden:

- Das höhere, anwendungsspezifische Abstraktionsniveau erlaubt eine Konzentration auf die Geschäftslogik.
- Weil keine SQL-Abfragen zu formulieren sind, wird die Abhängigkeit vom potentiell inkompatiblen SQL-Dialekt des verwendeten RDBMS vermieden.
- Es ist weniger Datenzugriffscode erforderlich.
- Änderungen im Datenbankschema haben weniger Einfluss auf den Quellcode.

23 Anhang

23.1 Operatorentabelle

In der folgenden Tabelle sind alle im Kurs behandelten Operatoren in absteigender Priorität (von oben nach unten) aufgelistet. Gruppen von Operatoren mit gleicher Priorität sind durch horizontale Linien abgegrenzt. Sie verfügen dann auch über dieselbe Auswertungsrichtung. Die meisten Operatoren links-assoziativ, werden also von links nach rechts ausgewertet. Z.B. wird

```
x - y - z
```

ausgewertet als

```
(x - y) - z
```

Die Zuweisungsoperatoren, der Lambda-Operator und der Konditionaloperator sind jedoch rechts-assoziativ; sie werden also von rechts nach links ausgewertet. Z.B. wird

```
a += b -= c = 3
```

ausgewertet als

```
a += (b -= (c = 3))
```

Operator	Bedeutung
x.*y*	Member-Zugriff
Methode(*Parameter*)	Methoden- oder Delegatenaufruf
[]	Index
x++, *x* --	Postinkrement bzw. -dekrement
new	Objekterzeugung
typeof	Typermittlung
checked	Ganzzahl-Überlaufdiagnose
-*x*	Vorzeichenumkehr
!	Negation
~	Bitweise Negation
++*x*, --*x*	Präinkrement bzw. -dekrement
(*Typ*)	Typumwandlung
*, /	Punktrechnung
%	Modulo (Divisionsrest)
+, -	Strichrechnung

<table>
<tr><th>Operator</th><th>Bedeutung</th></tr>
<tr><td>+</td><td>Stringverkettung</td></tr>
<tr><td><<, >></td><td>Links- bzw. Rechts-Shift</td></tr>
<tr><td>>, <,
>=, <=, is</td><td>Vergleichsoperatoren</td></tr>
<tr><td>==, !=</td><td>Gleichheit, Ungleichheit</td></tr>
<tr><td>&

&</td><td>Bitweises UND

Logisches UND (mit unbedingter Auswertung)</td></tr>
<tr><td>^

^</td><td>Exklusives bitweises ODER

Exklusives logisches ODER</td></tr>
<tr><td>|

|</td><td>Bitweises ODER

Logisches ODER (mit unbedingter Auswertung)</td></tr>
<tr><td>&&</td><td>Logisches UND (mit bedingter Auswertung)</td></tr>
<tr><td>||</td><td>Logisches ODER (mit bedingter Auswertung)</td></tr>
<tr><td>??</td><td>Null-Koaleszenz</td></tr>
<tr><td>? :</td><td>Konditionaloperator</td></tr>
<tr><td>=

+=, -=,
*=, /=,
%=

=></td><td>Wertzuweisung

Wertzuweisung mit Aktualisierung

Lambda-Operator</td></tr>
</table>

23.2 Lösungsvorschläge zu den Übungsaufgaben

Kapitel 1 (Einleitung)

Aufgabe 1

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.

Aufgabe 2

1. **Falsch**
   Plattformunabhängigkeit setzt allerdings voraus, dass ein Assembly für die Platform MSIL übersetzt wurde.
2. **Richtig**
3. **Falsch**
   Die Standardbibliothek (FCL) ist für alle .NET – Sprachen identisch.
4. **Richtig**
   Allerdings müssen die Regeln der CLS (Common Language Specification) eingehalten werden, damit die Interoperabilität garantiert ist.

Aufgabe 3

Bisher wurden als wesentliche Aufgabe der CLR erwähnt:

- Verifikation des MSIL-Codes beim Laden
  So werden technische Defekte abgefangen.
- Überwachung von Code mit beschränkten Rechten (*hosted code*, via Internet bezogen)
- Der JIT-Compiler in der CLR übersetzt den MSIL-Code der Assemblies in Maschinencode.
- Speicherverwaltung (GC, Garbage Collection)

Aufgabe 4

In Benchmark-Tests schneiden .NET – Anwendungen im Vergleich zu traditionellen Windows-Programmen, die ein Compiler direkt in Maschinencode übersetzt hat, gut ab.

Mit dem SDK-Werkzeug **ngen.exe** kann der IL-Code eines Assemblies im Maschinencode übersetzt werden, so dass bei der Ausführung kein JIT-Compiler mehr benötigt wird. Allerdings zeigen sich dabei nur in speziellen Situationen Leistungsverbesserungen.

Aufgabe 5

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.

Z.B. befindet sich die Klasse **Uri** aus dem Namensraum **System**, die zur Modellierung von Netzwerk-Ressourcen dient, im DLL-Assembly **System.dll**. Die ebenfalls zum Namensraum **System** 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 6

MSIL — Ein .NET – Compiler übersetzt den Quellcode nicht in Maschinensprache, sondern in die *Microsoft Intermediate Language* (kurz: MSIL). Diesen Zwischencode übersetzt der JIT-Compiler in der CLR in Maschinencode.

FCL — Die Standardklassenbibliothek der .NET - Plattform wird als *Framework Class Library* bezeichnet. Sie enthält für praktisch alle Routineaufgaben der Programmierung (z.B. graphische Benutzoberflächen, Dateiverarbeitung, Netzwerkprogrammierung) ausgereifte Lösungen.

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 — Diese traditionelle Windows-Komponententechnologie (*Component Object Model*) soll vom .NET – Framework abgelöst werden. Aktuell (2010) spielt COM-Software in der Windows-Welt aber noch eine große Rolle.

Kapitel 2 (Werkzeuge zum Entwickeln von C# - Programmen)

Aufgabe 2

Beim Hallo-Programm lohnt sich die **using**-Direktive für den Namensraum **System** ausnahmsweise nicht, weil das Namensraum-Präfix im Quellcode nur einmal auftritt:

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

Aufgabe 3

- Das Schlüsselwort **using** wird klein geschrieben.
- 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 3 (Elementare Sprachelemente)

Abschnitt 3.1 (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 noch.
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 beginnen.
- `else`
  Schlüsselwörter wie **else** sind als Namen verboten.

Abschnitt 3.2 (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 3.3 (Variablen und Datentypen)

Aufgabe 1

Die Variable `i` ist nur im innersten Block gültig.

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';
```

Abschnitt 3.5 (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 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 Null, 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`, so dass 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 (>, ==) stehen in der Priorität über den logischen Operatoren, so dass 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 Priorität vor dem Operator **&&** ausgewertet.

`la2` erhält den Wert **true**, weil die runden Klammern dafür sorgen, dass der Operator **^** zuletzt ausgewertet wird.

`la3` erhält den Wert **false**, und die Variable `c` bleibt beim Initialisierungswert `'n'`.

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\Exp

Aufgabe 5

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\DM2Euro

Abschnitt 3.7 *(Anweisungen)*

Aufgabe 1

Weil die **else**-Klausel der *zweiten* (nach oben nächstgelegenen) **if**-Anweisung zugeordnet wird, ergibt sich folgender „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.

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, so dass 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\DM2EuroS

Aufgabe 6

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 3.3.5.1)

$$(-1)^0 \cdot 2^{(1023-1023)} \cdot (1+0,0)$$

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

$$1,0 + 2^{-53} \approx 1,0 + 1,1102230246251565 \cdot 10^{-16} \approx 1,0000000000000001110223024625157$$

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

„mühelos“ 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 7

Lösungsvorschläge mit den beiden Algorithmus-Varianten befinden sich in den Ordnern:

...\BspUeb\Elementare Sprachelemente\GGT.Diff
...\BspUeb\Elementare Sprachelemente\GGT.Mod

Kapitel 4 (Klassen und Objekte)

Aufgabe 1

1. **Falsch**
2. **Stimmt**
   Die Methodenüberladung ist erlaubt und üblich.
3. **Falsch**
   Es darf überhaupt kein Rückgabetyp angegeben werden.
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 Klassen, sofern eine entsprechende Referenz vorhanden ist. Der Klassendesigner ist für das sinnvolle Verhalten der Methoden verantwortlich.
5. **Stimmt**
6. **Falsch**

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. Methoden fremder Klassen sollte in der Regel der direkte Zugriff auf das Feld versperrt sein (Datenkapselung).

Aufgabe 3

In der Anweisung

```
return Anzahl;
```

wurde `Anzahl` versehentlich groß geschrieben, so dass sich die **set**-Methode der Eigenschaft `Anzahl` selbst aufruft. Diese ungeplante Rekursion führt zu einem Stack-Überlauf (vgl. Abschnitt 4.7.1).

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\FakulRek

Aufgabe 6

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Einleitung\Bruch\GUI

Aufgabe 7

Begriff	**Pos.**
Definition einer Instanzmethode mit Referenzrückgabe	7
Deklaration lokale Variable	8
Def. einer Instanzmeth. mit Wert-parameter 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 5 (Weitere .NETte Typen)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Weitere .NETte Typen\Strukturen\R2VekS

Es sind nur wenige Änderungen erforderlich:

- Das Schlüsselwort **class** ist durch **struct** zu ersetzen.
- Die Felder von Strukturen dürfen bei der Deklaration nicht initialisiert werden.
- Es darf kein parameterfreier Konstruktor definiert werden.

Aufgabe 2

Dank Autoboxing kann man einer **object**-Variablen auch einen **int**-Wert zuweisen, wobei ein neues Objekt auf dem Heap erstellt wird, 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 5.4.1.2.2).

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Weitere .NETte Typen\Arrays\Lotto

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Weitere .NETte Typen\Arrays\Eratosthenes

Aufgabe 5

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Weitere .NETte Typen\Arrays\FloatMatrix

Aufgabe 6

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Weitere .NETte Typen\Zeichenfolgen\PerZuf

Aufgabe 7

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Weitere .NETte Typen\Zeichenfolgen\StringUtil

Kapitel 6 (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 aufruft (vgl. Abschnitt 6.3).

Aufgabe 2

In der Klasse `Figur` haben `xpos` und `ypos` den voreingestellten Zugriffsschutz **private**. Damit hat die `Kreis`-Klasse keinen direkten Zugriff. 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 der Basisklassenmethode durch eine Unterklassenmethode mit gleichem Namen und identischer 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 die *Basisklassenvariante* zum Einsatz (frühe Bindung).
- Bei überschreibenden Methoden wird die *Unterklassenvariante* benutzt (späte bzw. dynamische Bindung).

Bei klassenbezogenen Methoden kommt das späte Binden bzw. Überschreiben nicht in Frage.

Kapitel 7 (Typgenerizität und Kollektionen)

Aufgabe 1

Sie finden einen Lösungsvorschlag in:

...\BspUeb\Typgenerizität und 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 des **List**<`Person`> - Objekts sogar sortiert werden. Weitere Informationen über Schnittstellen (*Interfaces*) folgen gleich in Abschnitt 8).

Aufgabe 2

Ein typisches Ergebnis (gemessen auf einem Rechner mit Intel-CPU Core i3 550:

```
Zeit in Millisek. für List<int>: 11,7188
Zeit in Millisek. für ArrayList: 63,4765
```

Das zugehörige Programm finden Sie in:

...\BspUeb\Typgenerizität und Kollektionen\Listenwerte

Kapitel 8 (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 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 Konstruktoren, Felder und statische Member verboten.
- **Abstammungsverhältnisse (Typkompatibilitäten)**
  Eine Klasse kann nur *eine* abstrakte Basisklasse besitzen, aber beliebig viele Interfaces implementieren.

Aufgabe 3

Weil Interfaces auch 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.

Kapitel 9 (Einstieg in die GUI-Programmierung mit WPF-Technik)

Aufgabe 1

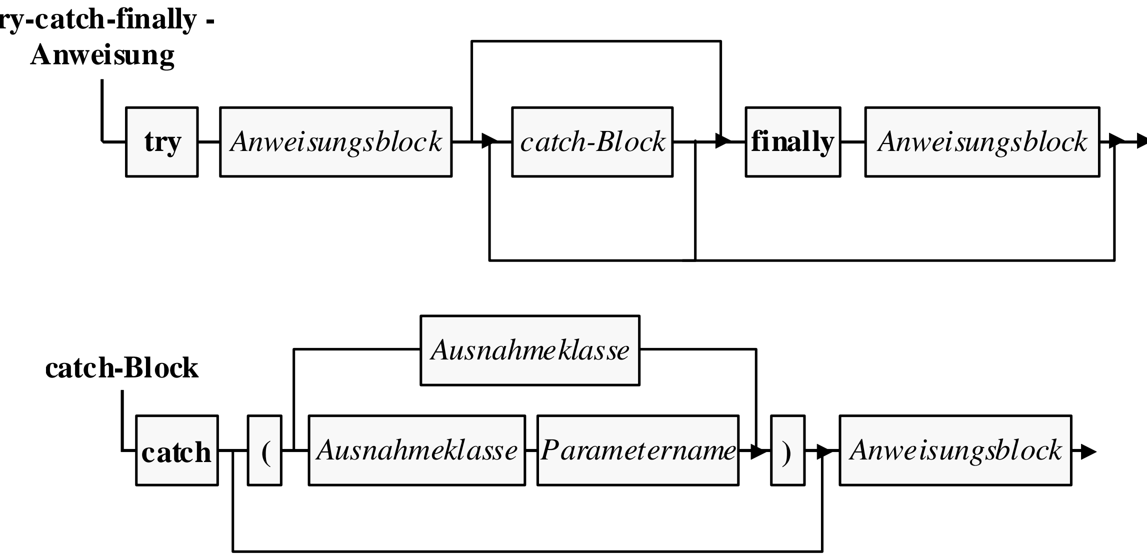
Sie müssen lediglich im XAML-Code zum **DockPanel**-Layoutcontainer die Aufnahmereihenfolge ändern:

```
<DockPanel>
    <Button DockPanel.Dock="Left" Width="50">Left</Button>
    <Button DockPanel.Dock="Right" Width="50">Right</Button>
    <Button DockPanel.Dock="Top">Top</Button>
    <Button DockPanel.Dock="Bottom">Bottom</Button>
    <Button>Rest</Button>
</DockPanel>
```

Kapitel 10 (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 11 (Attribute)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Attribute\NonsenseAttribute

Kapitel 12 (Ein- und Ausgabe über Datenströme)

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 13 (Multithreading)

Aufgabe 1

- Falsch
- Richtig
- Richtig
- Falsch

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Multithreading\CountdownEvent

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Multithreading\DirSize

Kapitel 14 (Netzwerkprogrammierung)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Netzwerk\SimpleBrowser

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Netzwerk\MultiClientEchoServer\DauerSender

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Netzwerk\Chat

Kapitel 15 (Datenbindung)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

…\BspUeb\WPF\Datenbindung\Slider und ComboBox

Kapitel 17 (Anwendungs- und Benutzereinstellungen)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

…\BspUeb\Anwendungs- und Benutzereinstellungen\XML\EddieConfigPlus

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

…\BspUeb\Anwendungs- und Benutzereinstellungen\Registry\EddieRegPlus

Literatur

Albahari, J. (2010). *Threading in C#*. Online-Dokument: http://www.albahari.com/threading/.

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. (2010a). *Einführung in das Programmieren mit Java 6*. Online-Dokument: http://www.uni-trier.de/index.php?id=22787

Baltes-Götz, B. (2010b). *Einführung in das Programmieren mit C# 3.0*. Online-Dokument: http://www.uni-trier.de/index.php?id=22777

Balzert, H. (1999). *Lehrbuch der Objektmodellierung: Analyse und Entwurf*. Heidelberg: Spektrum.

Bayer, J. (2006). *Das C# 2005 Codebook*. München: Addison-Wesley

Doberenz, W. & Gewinnus, T. (2010). *Datenbankprogrammierung mit Visual C# 2010*. Unterschleißheim: Microsoft Press.

Campbell, C., Johnson, R., Miller, A. & Toub, S. (2010). *Parallel Programming with Microsoft .NET*. Microsoft Press. Online verfügbar: http://msdn.microsoft.com/en-us/library/ff963553.aspx

Drayton, P., Albahari, B. & Neward, T. (2003). *C# in a Nutshell*. Beijing: O'Reilly.

Ebner, M. (2000). *Delphi 5 Datenbankprogrammierung*. München: Addison-Wesley.

ECMA (2006). *C# Language Specification* (4th ed.). Online-Dokument: http://www.ecma-international.org/publications/standards/Ecma-334.htm

ECMA (2009). *Open XML Paper Specification*. Online-Dokument: http://www.ecma-international.org/publications/standards/Ecma-388.htm

Eller, F. & Kofler, M. (2005). *Visual C#*. München: Addison-Wesley.

Frischalowski, D. (2007). *Windows Presentation Foundation*. München: Addison Wesley.

Goll, J., Weiß, C. & Rothländer, P. (2000). *Java als erste Programmiersprache*. Stuttgart: Teubner

Gunnerson, E. (2002). *C#* (2. Aufl.). Bonn: Galileo

Krüger, M. (2002). *C# Coding Style Guide*. (Version 0.3). Online-Dokument: http://www.icsharpcode.net/TechNotes/SharpDevelopCodingStyle03.pdf

Kühnel, A. (2010). *Visual C# 2010*. Bonn: Galileo Press. Auch als OpenBook verfügbar: http://www.galileocomputing.de/openbook/visual_csharp/

Lau, O. (2009). Faites vos jeux! Zufallszahlen erzeugen, erkennen und anwenden. *c't Magazin für Computertechnik*. 2009, Heft 2, 172-178.

Lammarsch, J. & Lammarsch, M. (2004). *C#*. Hannover: RRZN.

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.

Liberty, J. (2002). *Programming C#* (2nd ed.). Sebastopol, CA: O'Reilly.

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.ps

Louis, D. & Strasser, S. (2002). *C# in 21 Tagen*. München: Markt + Technik.

Louis, D. & Strasser, S. (2008). *Microsoft Visual C# 2008 - Das Entwicklerbuch*. Unterschleißheim: Microsoft Press Deutschland.

MacBeth, G. S. (2004). *C#. Programmer's Handbook*. New York: Springer.

Michaelis, M. (2010). *Essential C# 4.0*. Upper Saddle River, NJ: Addison-Wesley.

Microsoft Inc. (2010a). *C# Language Specification 4.0*. Online-Dokument: http://www.microsoft.com/downloads/en/details.aspx?FamilyID=DFBF523C-F98C-4804-AFBD-459E846B268E

Microsoft Inc. (2010b). *Windows User Experience Interaction Guidelines for Win 7 and Win Vista*. Online-Dokument: http://www.microsoft.com/downloads/en/details.aspx?FamilyID=-E49820CB-954D-45AE-9CB3-1B9E8EA7FE8C

Misner, S. (2007). *Microsoft SQL Server 2005 Express Edition. Start Now!* Redmond, WA: Microsoft Press.

Mössenböck, H. (2003). *Softwareentwicklung mit C#. Eine kompakter Lehrgang*. Heidelberg: dpunkt.

Mössenböck, H. (2005). *Sprechen Sie Java? Einführung in das systematische Programmieren*. Heidelberg (3. Aufl.). Heidelberg, dpunkt.Verlag.

Müller, N. (2004). *Rechner-Arithmetik*. Online-Dokument: http://www.informatik.uni-trier.de/~mueller/Lehre/2005-arith/arith-2003-folien.pdf

Petzold, C. (2008). *Vektorgrafik und die WPF-Shape-Klasse*. MSDN-Magazin. 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.

Sceppa, D. (2003). *Microsoft ADO.NET – Das Entwicklerhandbuch*. Unterschleißheim: Microsoft Press.

Schwichtenberg, H. (2009a). *C# oder Visual Basic? Die richtige Programmiersprache für .NET-Entwickler*. Online-Dokument: http://www.heise.de/developer/C-oder-Visual-Basic-Die-richtige-Programmiersprache-fuer-NET-Entwickler--/artikel/140794

Schwichtenberg, H. (2009b). *Verwirrung um objekt-relationale Mapper*. Online-Dokument: http://www.heise.de/developer/artikel/Verwirrung-um-objekt-relationale-Mapper-LINQ-to-SQL-oder-ADO-NET-Entity-Framework-227256.html

Schwichtenberg, H. (2010). Kantenglättung. *iX. Magazin für professionelle Informationstechnik*, 2010(3), S. 139-143).

Sells, C & Griffiths, I. (2007). *Programming WPF* (2nd ed.). Sebastopol, CA: O'Reilly.

Spurgeon, C. E. (2000). *Ethernet. The Definitive Guide*. Sebastopol, CA: O'Reilly.

Strey, A. (2003). Computer-Arithmetik. Online-Dokument: http://www.informatik.uni-ulm.de/ni/Lehre/WS02/CA/CompArith.html

Stichwortregister

(

() Operator 110

:

:
  -Operator 25

+

+-Operator 80

A

Abfrage
  parametrisierte 642
Ablaufsteuerung 131
Abort() 469
abstract 271, 293
Abstraktion 1
AcceptChanges() 649, 652, 654
AcceptTcpClient() 511
AccessText 380
ACK-Bit 497
Acos() 219
ActiveX Data Objects 611
Add()
  Dictionary<K,V> 286
  HashSet<T> 285
  ListBox 390
AddHandler() 354
ADO 611
Aggregation 198
Aggregieren 617
Aktionsabfrage 614
Aktualisierungsoperatoren 120
Aktualparameter 168, 172
Algorithmen 8
allowExeDefinition 550
Alt-Tastenbefehl 380
Angefügte Eigenschaften 363
Angefügte Ereignisse 357
Anonyme Klassen 678
Anonyme Methoden 312
ANSI-Code 443
Anweisung
  zusammengesetzte 132
Anweisungen 130
Anweisungsblöcke 131
Anwendungsdomäne 638
Anwendungseinstellungen 548
Anwendungsfenster 304
Anwendungskonfigurationsdatei 548, 640, 666
APM 475
Append()
  StringBuilder 248
Application 206, 304, 306, 336
ApplicationException 412
ApplicationScopedSettingAttribute 554
ApplicationSettingsBase 553
Apply 541
AquireReaderLock() 467
AquireWriterLock() 467
ArgumentOutOfRangeException 410, 415
Arithmetische Operatoren 107
Arithmetischer Ausdruck 107
Array 232, 289
  mehrdimensional 238
ArrayList 240, 277
Array-Parameter 171
ASCII-Code 443
as-Operator 268
Assembler 12
Assembly 15, 17
AssemblyInfo.cs 423
Assembly-Metadaten 19
AssemblyName 424
Assoziativität 121
AsyncCallBack 476
Asynchronous Programming Model 475
AsyncWaitHandle 476
AttachDbFilename 638
attached event 357
attached properties 363
Attribut
  XML 323, 549
Attribute 417
Aufzählungen 249
Ausdrücke 106
Ausdrucksanweisungen 131
Ausgabe
  im Konsolenfenster 79
Ausgabeparameter 170
Ausgabetyp 32
Ausnahme 397
Auswahlabfrage 614
Auswertungsfunktionen 617
Auswertungsreihenfolge 121
Auswertungsrichtung 121
Autoboxing 231
AutoFlush 442

B

Backing Field 189
BAML-Datei 331
base 260, 264
Base Class Library 24
base-Konstruktoren 260, 261
Basisklasse 257
BCL 24
Bearbeitungsmodus
  DataRow 651
Bedingte Anweisung 132
Befehlsschalter 378
Befehlszeilenargumente 139
BeginEdit() 651
BeginGetHostEntry() 510
BeginInvoke() 475
  Control 482
Benutzerinstanz 657, 665
Benutzerinstanz beim SQL-Server Express 639
Bezeichner 77
Beziehung 613
Bézier-Spline 577
Binärdateien 436
binäre
  Operatoren 107
Binäre
  Gleitkommadarstellung 89
BinaryFormatter 445

BinaryReader 437
BinaryWriter 437
Binding 347, 521
BindingNavigator 670
Bitfelder 425
Bitflags 425
BitmapSource 590
Bitorientierte Operatoren 115
Bitweises UND 116
Blasenereignisse 354
Block 94
Blockanweisung 94, 131
BMP 588
bool 88
bool-Literale 100
BorderBrush 45
BorderThickness 45
Boxing 230
break 138
Break 485
break-Anweisung 147
breakpoint 174
Browseranwendungen 301
Bubbling 354
Button 378
ButtonBase 383
byte 87
Bytecode 16

C

C++ 95
Call Back - Routinen 299
Camel Casing 79
Canceled
    Task 490
CancelEdit() 651
CancellationToken 489
CancellationTokenSource 489
CanRead 431
CanSeek 431
CanWrite 431
Capacity 241
case-Marke 138
casting 117
Casting-Operator 118
catch-Block 400
CFF-Explorer 21
CGI 503
Change()
    Timer 480
char 88
char-Literale 100
CheckBox 383
checked
    Anweisung 126
    Compileroption 126
    Operator 126
Children 360
CIL 16
Clear()
    Dictionary<K, V> 287
    HashSet<T> 285
    ListBox 390
Clone() 290
Close() 431
    StreamWriter 442
Closing-Ereignis 558, 562
CLR 13, 23
CLS 13, 17
Code-Behind – Dateien 329
CollectionView 671
CollectionViewSource 671
ColumnSpan
    Grid 365
COM 22, 420
ComboBox 388
CommandBuilder 656
Common Gateway Interface 503
Common Intermediate Language 16
Common Language Runtime 13
Common Language Specification 13, 17
Common Type System 154, 229, 258
    Strukturen 229
CompareTo() 280
    String 244
Compiler 15
Conceptual Schema Definition Language 686
ConnectionString 637
Console 79
const 96, 163
ConstraintCollection 648
Contains()
    HashSet<T> 285
ContainsKey()
    Dictionary<K,V> 286
ContentBox 601
continue-Anweisung 147
controls 299
Convert 103
Copy()
    File 449
Cos() 219
CountdownEvent 467
CountdownSignal 491
CPU 12
Create()
    File 449
    WebRequest 498
CreateDirectory() 450
CreateSubKey() 563
CreateText()
    File 449
csc 31
csc.exe 15
csc.rsp 32
CSDL 686
ctor 181
CTS 154, 229, 258
    Strukturen 229
CultureInfo 455
CurrentThread() 459

D

dangling else 135
DataColumn 646
DataDirectory 638, 689
DataReader 659
DataRelation 654
DataRow 646
DataRowState 652
DataRowVersion 653
DataSet-Designer 667
DataTable 646
DataTemplate 347, 523
Datenbankdiagramm 674
Datenbank-Explorer 672
Datenbankmanagementsystem 609
Datenbindung 347

Datenkapselung 151, 164
Datenstrom 429
Datentyp 84
Datentypen
  elementare 87
DateTime 150
DB2 635
DBMS 609
DBNull 649
Deadlock 472
debug
  -Compiler-Option 409
Debug 174
decimal 87, 91, 99, 113, 129
default
  bei einem Typparameter 280
  switch 138
DefaultProperty 326
Definitionstabellen 19
Deklarationsbereich 94, 95
Deklarative Programmierung 417
delegate
  Schlüsselwort 309
Delegaten 308
  generische 313
Delete()
  Directory 450
  File 449
DeleteValue() 562
DeMorgan 124
Denormalisierte
  Gleitkommadarstellung 90
DependencyObject 303
deprecated 472
Descendents() 345
Destruktor 184
Dialogfenster
  nicht-modal 542
DictionaryEntry 409
Directory 450
DirectoryInfo 450
Direktereignisse 354
Direktive
  using 25
DispatcherObject 303
DispatcherTimer 482
Dispose() 432
DLL-Hölle 19
Dns 509
DNS 509
Dock 366
DockPanel 366
Dokumentationskommentar 77
Dokumentengliederung 213
Domain Name System 509
Domänen-Controller 551
do-Schleife 146
double 87, 89, 112
Double 128
  Epsilon 130
Downcast 268
Download
  Dateien 501
DownloadData() 506
DownloadFile() 501
Drag Once 669
Durchfall 138
dynamic 34, 84
Dynamische Bindung 268

E

EDM 686
EDMX-Datei 686
Eigenschaft
  Syntaxdiagramm 75
Eigenschaften 4, 188
  klassenbezogene 192
Eigenschaftenfenster 211
Einschränkende Konvertierung 118
einstellige
  Operatoren 107
Element
  XML 548
Elementare Datentypen 87
else-Klausel 133
Emboss-Effekt 585
Encoding 443
EndEdit() 651
EndGetHostEntry() 510
EndInvoke() 475
Endlosschleife 147
Enter()
  Monitor 462
Entität 686
Entitätszeichenfolge 689
Entity Framework 685
Entity SQL 695
EntityClient 695
EntityObject 697
EntityState 701
Enum 425
Enumerable 681
Enumerationen 249
Epsilon 130
Eratosthenes 254
Ereignisse 314
ERRORLEVEL 398
Ersetzbarkeitsregel 273
Erweiternde Konvertierungen 117
Erweiterungsmethoden 680
Escape-Schaltfläche 380
Escape-Sequenzen 80, 81, 100
Euklidischer Algorithmus 8, 149
event 319
Exception 397
Exception-Handler 400
ExecuteNonQuery() 642
ExecuteReader() 642
ExecuteScalar() 642
Exists()
  Directory 450
  File 449
Exit 216
Exit() 398
  Klasse Environment 398
  Monitor 462
Exitcode 398
Exklusives logisches ODER 115
Explizite
  Schnittstellenimplementierung 296
Expression Blend 301

F

FCL 13, 24
Felder 2
  klassenbezogene 191
  verdeckte 265
Fensterdesigner 42

FieldOffsetAttribute 426
File 448
FileAccess 435
FileInfo 448
FileMode 435
FileShare 436
FileStream 430, 434
FileSystemWatcher 451
Fill()
  DataAdapter 644
FillSchema() 645
Finalize() 185
finally-Block 400
Find()
  DataRowCollection 649
FindMembers() 421
FixedDocument 600
FixedPage 600
Flache Kopie 290
FlagsAttribute 425
Fließkommazahl 88
float 87, 89, 99, 112
floating point number 88
Flush() 431
  StreamWriter 442
Flussdiagramm 132
Focus() 216, 380
FontFamily 594
FontSize 594
FontStyle 594
FontWeight 594
For() 485
foreach 142
Formalparameter 168
Formatierte Ausgabe
  im Konsolenfenster 80
Formatierung
  von C#-Programmen 75
Formular
  HTML 504
for-Schleife 142
Frame 494
Framework 291
Framework Class Library 13, 24
FrameworkElement 304
Fremdschlüssel 614, 675, 692
Fremdschlüssel-Restriktion 614
FROM-Klausel (SQL) 615
FTP-Protokoll 503
Funktions-Pointer 308

G

GAC 104
Ganzzahlarithmetik 108
Ganzzahlliterale 97
Garbage Collector 184, 431
Generische
  Delegaten 313
  Klassen 279
  Methoden 284
Geschachtelte Klassen 201
GET
  CGI-Parameter 506
GetChildRows() 655
GetCurrentDirectory() 450
GetCustomAttribute() 421
GetCustomAttributes() 422
GetDirectories() 451
GetFiles() 451
GetHostEntry() 509
GetLength()
  Array 239
GetPrintQueue() 607
GetResponse() 499
GetResponseStream 499
GetString() 507
GetType() 154, 230, 259
GetValue()
  RegistryKey 562
GIF 588
Gleitkommaarithmetik 88, 108
Gleitkommadarstellung
  binär 89
Gleitkommaliterale 99
Gleitkommazahl 88
global 25
Global Assembly Cache 19
Globale Variablen 86
goto-Anweisung 138
GradientStop 581
Grid 359
  WPF-Layoutcontainer 360
GROUP BY 617
GUI 299

H

Hallo 29
Haltepunkt 174
Handle()
  AggregateException 488
HasErrors 651
HasVersion() 653
Hauptfenster 304
HDP 588
Heap 86, 160, 180, 454
Height 304
Help Viewer 59
Hilfebibliotheksmanager 60
Hollywood-Prinzip 299
Host-Name 509
HTTP 499, 501
HttpUtility 507
HttpWebRequest 499

I

IAddChild() 600
IClonable 290
ICMP 495
ICO 588
IcoFX 350
ICollectionView 525
IComparable 289, 294
IDictionary 409
IDisposable 432
IDL 22
IEEE-754 89
IEnumerable 345
IEnumerable<T> 681
if-Anweisung 132
IFormatter 445
IL 15
ILDasm 182, 183, 185, 188, 189
Image 588
ImageBrush 590
Implizite Typzuweisung 677
Index

in Datenbanktabellen 612
Indexer 251
IndexOf()
String 245
IndexOutOfRangeException 235, 398
information hiding 151, 164
Initialisierung 84, 93
Initialisierungslisten 237
InitializeComponent() 214, 331, 343
Inkonsistenz 609
InlineCollection 593
Inlining 190, 409
INNER JOIN (SQL) 617
Innere Klassen 201
InnerException 409
INotifyPropertyChanged 520
InputBox() 104
Insert()
StringBuilder 248
Instanzen 223
Instanzinitialisierer 677
Instanzvariablen 86, 159
Integrated Security 639
IntelliSense 47
Interaction 104
Interface 289
Interface Definition Language 22
Intern() 246
Interner String-Pool 246
Internet Control Message Protocol 495
Interrupt() 472
IntersectWith() 286
InvalidConstraintException 655
InvalidOperationException 419
IN-Vergleichsoperator (SQL) 616
Invoke()
Control 482
Parallel 483
IPAddress 509
IP-Adresse 509
IP-Datagramme 495
IPHostEntry 509
IP-Protokoll 495
IPv4 495
IPv6 495
IsBackground 458
IsCancel 380
IsChecked 384
IsDefined() 420, 425
ISerializable 444
IsInterned() 247
IsNaN() 128
IsNegativeInfinity() 128
is-Operator 268
IsPositiveInfinity() 128
ItemArray 649
ItemCollection 389
Items
ItemsControl 390
ItemSource
ListBox 389
ItemsSource 345
ItemTemplate 347, 523

J

jagged arrays 239
Java 16
JIT-Compiler 23
Join() 465
JPEG 588
JXR 588

K

Kapselung 151
Key
Registry 560
Keyboard 216, 380
KeyValuePair<K, V> 287
Klammern 121
Klasse
Syntaxdiagramm 72
Klassen 151
anonyme 678
innere 201
statische 195
Klassenansicht 697
klassenbezogene
Eigenschaften 192
Klassendiagramm 697
Klassenmethode 30
Klassenvariablen 86
Kodierung 439
Kollektionen 144
Kombinationsfeld 388
Kommentar 76
XML 323
Konditionaloperator 121
Konfiguration
Visual Studio 61
Konfigurations-Manager 63
Konsolenfenster
Formatierte Ausgabe 80
Konstanten 96, 163
Konstruktoren 181
statische 194
Kontextbezogen-reservierte Schlüsselwörter 78
Kontrollkästchen 383
Kontrollstrukturen 131
Konvertierung
einschränkende 118
erweiternde 117

L

Label 372
Lambda-Ausdrücke 679
Language Integrated Query 677
LayoutTransform 584
Lazy Loading 702
Leere Anweisung 131
Length 191, 235, 244
Stream 431
lexikographische Priorität 244
LIFO-Stapel 279
LIKE-Vergleichsoperator (SQL) 616
Links-Shift-Operator 116
LINQ 677
LINQ to Entities 700
LINQ to Objects 680
LINQ to SQL 685
LINQ to XML 345
Linux 13
Liskovsches Substitutionsprinzip 273
List<T> 278, 313
ListBox 388
Listenfeld 388
Listeninitialisierer 678

Literale 97
Loaded-Ereignis 216, 349, 380, 563
  bei Window 557
LocalFileSettingsProvider 559
LocalPrintServer 607
lock 460
Logische Operatoren 114
Logisches ODER 115
Logisches UND 114
Lokale Variablen 86
LSP 273

M

MAC-Adresse 494
machine.config 548
MacOS-X 13
Main() 9, 71
managed code 28
Manifest 19
ManualResetEventSlim 467
Markuperweiterungen 328
Maschinencode 12
Master-Detail -
  Beziehung 613
Math-Klasse 110
MaxValue
  Double 128
Mehrfachvererbung 260, 292
Member 5
MemberInfo 420, 421
memory leaks 184
MessageBox 65
Methode
  Syntaxdiagramm 74
Methoden 164
  Aufruf 172
  Definition 165
  generische 284
  Modifikatoren 166
  rekursive 195
  Rückgabewert 167
  statische 193
  Überladen 177
  überschreiben 268
MethodImplAttribute 461
Microsoft Intermediate Language 13, 15
Microsoft Reference Source Code Center 315, 353, 376
Microsoft SQL Server 2008 R2 Express Edition 617
MissingSchemaAction 646
Modellbrowser
  Entity Framework 690
ModifierKeys 425
Modifikatoren
  bei Methoden 166
Modularisierung 152
module 32
Module 19
Modulo 108
Monitor 462
Mono 13, 27
Move()
  File 449
mscorlib.dll 20, 32
MSIL 13, 15
Müllsammler 184
MulticastDelegate 309
Multiplizität 691
Multitasking 453
Multithreading 453
Murphy's Law 397
Mutex 462
MySQL 635

N

Namen 77
  von Klassen 159
Namensparameter 423
Namensräume 24
namespace 24
NameValueCollection 508
NaN 128
NAS 638
Nebeneffekt 107
Nebeneffekte 109, 115
Negation 114
Network Attached Storage 638
NetworkStream 511, 514
Netzwerkprogrammierung 493
new-Modifikator 264
new-Operator 179, 181
NewRow() 652
Next() 236
ngen.exe 23
NonSerialized 444
Normalisieren 613
Normalisierte
  Gleitkommandarstellung 89
Novell 27
null 162, 179
NULL (SQL) 616
Null-Koaleszenz - Operator 284
NullReferenceException 162

O

ObjectContext 696
Objektdiagramm 444
Objektinitialisierer 184
Objektserialisierung 444
Objektvariablen 86
ObservableCollection<> 527
Obsolete
  Attribut 418
ObsoleteAttribute 418
ODBC 635
Office Open XML 597
Öffnungsmodus 435
OLEDB 635
OneWay 524
OpCode 231
OpenSubKey() 563
OpenXPS 597
operator+ 198
Operatoren 106
  Arithmetische 107
  bitorientierte 115
  logische 114
  überladen 197
  vergleichende 111
Operatorentabelle 705
Optionsfeld 383
OracleClient 635
ORDER BY (SQL) 617
ORM 685
OSI-Modell 494
out 171
  -Compiler-Option 32

override 269, 270
Owner 536, 544

P

PageContent 600
Panel 360
PAP 132
Parallel 483
Parametrisierte Abfragen 642
params 171
partial 214, 330
Pascal 155
Pascal Casing 78
PasswordBox 388
Passwörter 388
PathSegment 576
Peek() 443
Pfadname 449
PHP 505, 506
ping 495
PNG 588
Polling 477
Polymorphie 268, 296
Port 495
Portable Network Graphics 350
Position
  Stream 430
Positionsparameter 423
POST
  CGI-Parameter 508
Postinkrement bzw. -dekrement 109
Potenzfunktion 110
Pow() 110
Prägungs-Effekt 585
Präinkrement bzw. -dekrement 108
Preemtives Zeitscheibenverfahren 471
Primärer Thread 458
Primärschlüssel 612, 647
PrimaryKey 647
Primzahlen 148
PrintDialog 607
PrintQueue 607
PrintServer 607
Priorität 121
Prioritäten 470
Priority
  Thread 470
private 161
Privates Assembly 19
Process 349
ProcessorArchitecture 20
Produktivität 3
Programmablaufplan 132
Projektion 682
Projektmappe 42
Projektmappen-Explorer 42
properties 188
Properties 4
PropertyChanged 520
protected 262
Provider 635
Pseudozufallszahlengenerator 235
Puffer
  StreamWriter 442
Pulse()
  Monitor 463
PulseAll() 464
Punktoperator 163, 172
PuTTY 513

Q

QUERY_STRING 506

R

Race Condition 460
RadioButton 383
Random 195, 236
Rank 239
RDBMS 611
Read()
  SqlDataReader 660
  Stream 430
ReadByte() 430
ReaderWriterLock 467
ReadKey() 452
ReadLine() 103
readonly 164
Read-Only
  Eigenschaft 190
ReadTimeout 515
record 155
Redundanz 609
Refaktorierung 57
reference
  -Compiler-Option 32
Referentielle Integrität 614
Referenz 65
Referenzparameter 169
Referenzsemantik 224
Referenztabellen 19
Referenztypen 85
Referenzvariablen 178
Reflection 420
Reflexion 417
Registrierungsdatenbank 560
Registry 547, 560
RegistryKey 561
RejectChanges() 654
Rekursive Methoden 195
Relationale Datenbanken 611
ReleaseReaderLock() 467
ReleaseWriterLock() 467
Reload() 558
Remove ()
  StringBuilder 248
Remove()
  HashSet<T> 285
  ListBox 390
RemoveKey()
  Dictionary<K,V> 287
RenderTransform 583
RenderTransformOrigin 586
Replace ()
  StringBuilder 248
Replace()
  String 245
Reservierte Schlüsselwörter 78
Reset() 465
ResetAbort() 470
Resize()
  Array 234
ResizeMode 378, 537
Response-Datei 32
Restmantisse 89
Resume() 472
return 141, 167
return-Anweisung 138, 167
Returncode 406

RichTextBox 595
roaming profile 551
Rollen
  SQL-Server 630
RotateTransform 586
Round-Robin 471
RoutedEvent 352
RoutedEventArgs 393
Router 495
Routingereignisse 352
RowDefinitions
  Grid 361
RowSpan
  Grid 365
RowState 651, 652
RSS-Feed 334
Rückgabewert 167
Run() 304
Running
  Task 490

S

SAN 638
Sandcastle 77
Save()
  Einstellungen 558
SaveChanges() 701
ScaleTransform 585
Schaltfläche 378
Scheduler 470
Schema einer Datenbank 614, 645
Schleifen 141
Schließen
  von Datenströmen 431
Schnittstelle 289
sealed 274
Seek() 431
SeekOrigin 431
Sekundäre Threads 458
Select() 518, 682, 683
SELECT-Befehl (SQL) 615
SelectedIndex 390
SelectedItem
  ListBox 390
SelectedItems 392
SelectionMode
  ListBox 392
Serialisieren 444
Serializable 444
SerializationException 444
Server-Explorer 672
SetColumn() 364
SetCurrentDirectory() 450
SetLastWriteTime()
  File 449
SetRow() 364
SettingChanging 559
SettingsProvider 552, 559
SetValue()
  RegistryKey 562
Shape 566
Show() 304
ShowDialog() 536, 540
ShowGridLines 365
Sichtbarkeitsbereich 95, 160
Sieb des Eratosthenes 254
Signatur 177, 263
Silverlight 52
Sin() 219
Singlethread-Apartment 420
Skalarprodukt 218
Sleep() 457
Slider 529
Smalltalk 151
SMTP-Server 496
Socket 496, 510
Solaris 13
SolidColorBrush 580
Solution 42
Sort() 289
SortDescription 526
Sortieren 526
Späte Bindung 268
Speicherlöcher 184
Split() 255
SQL 614
  FROM-Klausel 615
  GROUP BY 617
  INNER JOIN-Verbundoperator 617
  IN-Vergleichsoperator 616
  LIKE-Vergleichsoperator 616
  NULL 616
  ORDER BY 617
  SELECT-Befehl 615
  WHERE-Klausel 615
SQL Server 2008 R2 Express Edition 617
SQL Server Browser Dienst 633
SqlClient 635
SqlCommand 641, 657
SqlCommandBuilder 657
SqlConnection 637, 657
SqlDataAdapter 644, 657
SqlDataReader 659
SqlParameter 643
SqlServerCe 635
Sqrt() 218
STA 420
Stabilität 3
Stack 86, 160, 172, 454
  Überlauf 197
Stack Frames 175
StackPanel 320, 367
Standardkonstruktor 181
Start() 349
Startfähige Klasse 71
Startklasse 9
StartNew() 485
StartsWith()
  String 245
StartupUri 333
starvation 471
STAThreadAttribute 419
static 191
Statische
  Felder 191
  Klassen 195
  Konstruktoren 194
  Methoden 193
Statische Methoden
  Verdecken 265
Statusvariable 407
Steuerelemente 299
Storage Area Network 638
Stream 430
streams 429
Stretch 589
String 242, 289
  Methoden 243
StringBuilder 247

StringFormat
  Binding 530
String-Pool 246
Strings
  vergleichen 244
  verketten 243
Strom 429
struct 155, 224
StructLayoutAttribute 426
Structured Query Language 614
Struktogramm 197
Strukturen 223
Strukturiertes Programmieren 155
Substitutionsprinzip 273
Substring()
  String 244
Suspend() 472
switch-Anweisung 137
Synchronisierter Block 461
Synchronized() 557
Syntaxdiagramm 72
System.IO 429

T

TableAdapter 662
TableMappings 644
Tag-Navigator 213
target
  -Compiler-Option 32
Task 485
Task Parallel Library 483
TaskFactory 485
TCP 495
TCP/IP
  SQL-Server 633
TcpClient 514
TCP-Flags 497
TcpListener 510
Terminalserver 36
TextAlignment 594
TextBlock 593
TextBox 45, 519
TextChanged 387
Textdateien 436, 440
Texteingabefeld 386
TextProperty 519
TextReader 440
TextWrapping 387, 593
TextWriter 440
this 163, 173, 183, 187
  Indexer 252
Thread 454
ThreadAbortException 469
ThreadInterruptedException 472
Threadpool 454, 473
ThreadPriority 470
Threads 453
ThreadState 471
throw 410
throw-Anweisung 138
Tiefe Kopie 290
TIFF 588
TileMode 591
Time To Live 495
Timeout
  NetworkStream 514
Timer
  Threading 480
Time-To-Live 497
ToChar() 116
ToInt32() 103
ToLower() 140, 245
ToolTip 393
ToolTipService 393
ToString()
  StringBuilder 249
ToUpper() 245
Transform 582
Transformationen 582
TranslateTransform 584
Transmission Control Protocol 495
Transparentfarbe 382
Trennzeichen 75
TrimToSize() 241
try-catch-finally 399
T-SQL 614
TTL 497
Tunnelereignisse 353
Tunneling 353
Type 154, 259
Typformalparameter 279, 285
Typinferenz 677
Typisierte DataSets 660
Typisiertes DataSet 661
Typkonverter
  XAML 325, 328
Typ-Metadaten 18
Typsicherheit 84
Typtest-Operator 268
Typumwandlung 117

U

Überladen
  von Methoden 177
  von Operatoren 197
Überladung 243, 264
Überlauf 124
Übernehmen 541
Überschreiben von Methoden 268
UDP 496
UIElement 304, 354
UIElementCollection 360
uint 87
ulong 87
UML 5
Umschalter 383
unäre
  Operatoren 107
Unboxing 231
unchecked-Operator 127
undefinierte Werte 128
Unendlich 127
Ungarische Notation 384
Unicode-Escape-Sequenzen 100
Unicode-Zeichensatz 78
Unified Modeling Language 5
Uniform Resource Identifier 498
UniformGrid 368
Union 426
UnionWith() 286
UNIX 13
unmanaged code 20
Unterbrechungspunkt 174
Unterlauf 130
Unterprogrammtechnik 155
Unterstrich 79
Update()
  DataAdapter 645

UploadValues 508
Uri 333
URI 498
URL 498
UrlEncode() 507
URL-Kodierung 505
User Datagram Protocol 496
User Instance 639
UserAgent 499
UserScopedSettingAttribute 554
ushort 87
using
  -Direktive 25
using-
  Anweisung 432
UTF8Encoding 439
UTF8-Kodierung 443

V

ValidationRule 528
Validierung
  von Einstellungen 559
value 188
Value
  Registry 560
ValueType 228
var 677, 678
Variablen 82
  globale 86
  lokale 86
Variablendeklaration 93, 131
Verbindungsloser Datenzugriff 635
Verbindungszeichenfolge 637
Verbundanweisung 94, 131
Verdecken 263
Verdeckte Felder 265
Vererbung 257
Verfügbarkeit 203
Vergleich 111
Vergleichen
  von Strings 244
Vergleichsoperatoren 111
Verifikation 23
Verketten
  von Strings 243
Verkettete Liste 251
versiegelt 274
Versiegelte
  Klassen 274
  Methoden 273
Verweis 65
Viewport 591
virtual 269, 270
Visual 303
Visual C# 2010 Express Edition 51, 126, 204
  Befehlszeilenargumente 141
Vorschauereignisse 354

W

Wahrheitstafeln 114
Wait()
  Monitor 463
WaitAll() 467
WaitAny() 467
WaitCallback 474
WaitHandle 476
WaitingToRun
  Task 490
WaitOne() 465
WaitSleepJoin-Zustand eines Threads 463
Wanderndes
  Benutzerprofil 551
WCF 504
WDP 588
WebBrowser 500, 517
WebClient 501, 508
Webdienste 493
WebResponse 499
Wertparameter 168
Wertsemantik 224
Werttypen 84
Wertzuweisung 94
where
  Typrestriktion 281
WHERE-Klausel (SQL) 615
while-Schleife 145
widgets 299
Width 304
Wiederholungsanweisungen 141
Window 533
Windows Communication Foundation 504
Windows Life ID 54
Windows Presentations Foundation 13
Windows Vista 13
Windows-Registry 560
Windows-SDK 52, 182
WindowState 538
WinFX 300
WM_QUIT 307
WOW64 20
WPF 13, 299
WPF-Browseranwendungen 301
WPF-Designer 42, 204, 335
WPF-Ereignissystem 352
WrapPanel 368
Write() 79, 431
WriteAsync() 606
WriteByte() 431
WriteLine() 79
Write-Only
  Eigenschaft 190
WSDL 493

X

XAML 205, 322
XAML-Auflistungssyntax 327
xamlc.exe 331
XAML-Eigenschaftselement 325
XAML-Inhaltseigenschaft 326
XAML-Instanzelement 325
XAML-Textinhaltssyntax 327
XAML-Typkonverter 325, 328
XDocument 344
XML 322, 448, 547, 548
XML-Kommentar 323
XML-Namensräume 324
xmlns 324
XmlSerializer 448
XPS 597
XpsDocumentWriter 601
XPS-Viewer 597
xsd.exe 660

Z

Zahlenkreis 125
Zeichenketten 242
Zeichenkettenliterale 101
Zeilenumbruch 255
Zeitscheibenverfahren 471
Zoom-Werkzeug 208, 339
Zufallszahlen 195, 235
Zugriffsmethode 188
Zugriffsschutz 151, 203
Zugriffstaste 380
Zusammengesetzte
 Anweisung 132
Zuweisungsoperator 119
Zweierkomplement 125
zweistellige
 Operatoren 107