

B. Baltes-Götz

Einführung in das Programmieren mit C# 3.0

2010.04.09

Herausgeber: Universitäts-Rechenzentrum Trier
 Universitätsring 15
 D-54286 Trier
 WWW: <http://www.ur.t.uni-trier.de/>
 E-Mail: urt@uni-trier.de
 Tel.: (0651) 201-3417, Fax.: (0651) 3921

Leiter: Dr. Peter Leinen

Autor: Bernhard Baltes-Götz (E-Mail: baltes@uni-trier.de)

Copyright © 2009; URT

Vorwort

Dieses Manuskript entstand als Begleitlektüre zum C# - Einführungskurs, den das Universitäts-Rechenzentrum Trier (URT) über zwei Semester (WiSe 2008/2009 - SoSe 2009) 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 European Computer Manufacturers Association (ECMA) standardisierte Programmiersprache, die auf den Vorbildern Java und C++ 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 (Linux, Mac-OS-X, UNIX) portiert. Dementsprechend ist auf dem EDV-Arbeitsmarkt ein großes Interesse an der .NET-Plattform und an der hier bevorzugten Programmiersprache C# festzustellen.

Das Manuskript behandelt wesentliche Konzepte und Methoden der objektorientierten Softwareentwicklung (z.B. Klassen, Vererbung, Polymorphie, Interfaces), berücksichtigt aber auch viele Standardthemen der Programmierpraxis (z.B. Dateizugriff, Netzwerk, Multithreading, grafische Benutzeroberflächen, Grafikausgabe, Datenbankprogrammierung, Druckausgabe etc.).

Voraussetzungen bei den Leser(innen)

- **EDV-Allgemeinbildung**
Dass die Leser(innen) wenigstens durchschnittliche Erfahrungen bei der *Anwendung* von Computerprogrammen haben sollten, versteht sich von selbst.
- **Programmierkenntnisse** werden *nicht* vorausgesetzt.
- **Motivation**
Generell 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.

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 2008 Express Edition* oder mit *Sharp Develop 3.x*. Das ebenfalls erforderliche *.NET – Framework* wird von der Visual Studio Express Edition bei Bedarf automatisch installiert, ist aber mittlerweile auf praktisch jedem Windows-Rechner als Basis zahlloser Anwendungen und seit Windows Vista als Bestandteil des Betriebssystems vorhanden. Die erwähnte Software ist kostenlos verfügbar. Hinweise zur Installation und zur Verwendung folgen in Kapitel 2 des Manuskripts.

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 Texts, so dass einige Fehler und Mängel verblieben sein dürften. Entsprechende Hinweise an die E-Mail-Adresse

baltes@uni-trier.de

werden dankbar entgegen genommen.

Inhaltsverzeichnis

VORWORT	III
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 Assemblies und Metadaten	17
1.2.4.1 Typ-Metadaten	18
1.2.4.2 Das Manifest eines Assemblies	18
1.2.4.3 Multidatei-Assemblies	18
1.2.4.4 Private und allgemeine Assemblies	19
1.2.4.5 Vergleich mit der COM-Technologie	19
1.2.5 CLR und JIT-Compiler	20
1.2.6 Namensräume und FCL	21
1.2.7 Zusammenfassung zu Abschnitt 1.2	23
1.3 Übungsaufgaben zu Kapitel 1	24
2 WERKZEUGE ZUM ENTWICKELN VON C# - PROGRAMMEN	27
2.1 C# - Entwicklung mit Texteditor und Kommandozeilen-Compiler	27
2.1.1 Editieren	27
2.1.2 Übersetzen in MSIL	30
2.1.3 Ausführen	32
2.1.4 Programmfehler beheben	32
2.2 Entwicklungsumgebungen und andere Programmierhilfen	34
2.2.1 Microsoft Visual Studio 2008 Professional	34
2.2.1.1 Terminal-Server – Umgebung	34
2.2.1.2 Konfiguration beim ersten Start	35
2.2.1.3 Eine erste GUI-Anwendung	37
2.2.2 Microsoft Visual C# 2008 Express Edition	41
2.2.2.1 Installation	41
2.2.2.2 Ein erstes Konsolen-Projekt	44
2.2.2.3 Document Explorer	47
2.2.2.4 Referenzen und Ausgabetyt setzen	48
2.2.3 SharpDevelop	51
2.3 Übungsaufgaben zu Kapitel 2	56

3	ELEMENTARE SPRACHELEMENTE	57
3.1	Einstieg	57
3.1.1	Aufbau von einfachen C# - Programmen	57
3.1.2	Syntaxdiagramme	58
3.1.2.1	Klassendefinition	58
3.1.2.2	Methodendefinition	60
3.1.2.3	Eigenschaftsdefinition	60
3.1.3	Hinweise zur Gestaltung des Quellcodes	61
3.1.4	Kommentare	62
3.1.5	Namen	63
3.1.6	Übungsaufgaben zu Abschnitt 3.1	64
3.2	Ausgabe bei Konsolenanwendungen	65
3.2.1	Ausgabe einer (zusammengesetzten) Zeichenfolge	65
3.2.2	Formatierte Ausgabe	66
3.2.3	Übungsaufgaben zu Abschnitt 3.2	67
3.3	Variablen und Datentypen	67
3.3.1	Wert- und Referenztypen	69
3.3.2	Klassifikation der Variablen nach Zuordnung	71
3.3.3	Elementare Datentypen	71
3.3.4	Vertiefung: Darstellung rationaler Zahlen im Arbeitsspeicher des Computers	73
3.3.4.1	Binäre Gleitkommadarstellung	73
3.3.4.2	Dezimale Gleitkommadarstellung	75
3.3.5	Variablendeklaration, Initialisierung und Wertzuweisung	77
3.3.6	Blöcke und Deklarationsbereiche für lokale Variablen	78
3.3.7	Konstanten	80
3.3.8	Literale	81
3.3.8.1	Ganzzahlliterale	81
3.3.8.2	Gleitkommalliterale	82
3.3.8.3	bool-Literale	83
3.3.8.4	char-Literale	84
3.3.8.5	Zeichenkettenliterale	84
3.3.9	Übungsaufgaben zu Abschnitt 3.3	85
3.4	Einfache Techniken für Benutzereingaben	86
3.4.1	Via Konsole	86
3.4.2	Via InputBox	88
3.5	Operatoren und Ausdrücke	90
3.5.1	Arithmetische Operatoren	91
3.5.2	Methodenaufrufe	93
3.5.3	Vergleichsoperatoren	94
3.5.4	Vertiefung: Gleitkommawerte vergleichen	95
3.5.5	Logische Operatoren	96
3.5.6	Vertiefung: Bitorientierte Operatoren	98
3.5.7	Typumwandlung (Casting) bei elementaren Datentypen	99
3.5.8	Zuweisungsoperatoren	101
3.5.9	Konditionaloperator	103
3.5.10	Auswertungsreihenfolge	104
3.5.11	Übungsaufgaben zu Abschnitt 3.5	105
3.6	Über- und Unterlauf bei numerischen Variablen	107
3.6.1	Überlauf bei Ganzzahltypen	107
3.6.2	Unendliche und undefinierte Werte bei den Typen float und double	109
3.6.3	Überlauf beim Typ decimal	111
3.6.4	Unterlauf bei den Gleitkommatypen	111

3.7	Anweisungen (zur Ablaufsteuerung)	112
3.7.1	Überblick	112
3.7.2	Bedingte Anweisung und Verzweigung	113
3.7.2.1	if-Anweisung	113
3.7.2.2	if-else - Anweisung	114
3.7.2.3	switch-Anweisung	117
3.7.3	Wiederholungsanweisungen	122
3.7.3.1	Zählergesteuerte Schleife (for)	123
3.7.3.2	Iterieren über die Elemente einer Kollektion (foreach)	124
3.7.3.3	Bedingungsabhängige Schleifen	125
3.7.3.4	Endlosschleifen	127
3.7.3.5	Schleifen(durchgänge) vorzeitig beenden	127
3.7.4	Übungsaufgaben zu Abschnitt 3.7	129
4	KLASSEN UND OBJEKTE	133
4.1	Überblick, historische Wurzeln, Beispiel	133
4.1.1	Einige Kernideen und Vorzüge der OOP	133
4.1.1.1	Datenkapselung und Modularisierung	133
4.1.1.2	Vererbung	135
4.1.1.3	Realitätsnahe Modellierung	136
4.1.2	Strukturierte Programmierung und OOP	136
4.1.3	Auf-Bruch zu echter Klasse	137
4.2	Instanzvariablen	140
4.2.1	Deklaration mit Wahl der Schutzstufe	141
4.2.2	Konstante und schreibgeschützte Instanzvariablen	142
4.2.3	Sichtbarkeitsbereich, Existenz und Ablage im Hauptspeicher	143
4.2.4	Initialisierung	144
4.2.5	Zugriff in klasseneigenen und fremden Methoden	145
4.3	Instanzmethoden	145
4.3.1	Methodendefinition	146
4.3.1.1	Modifikatoren	148
4.3.1.2	Rückgabewerte und return-Anweisung	148
4.3.1.3	Formalparameter	149
4.3.1.4	Methodenrumpf	152
4.3.2	Methodenaufruf und Aktualparameter	153
4.3.3	Methoden überladen	154
4.4	Objekte	155
4.4.1	Referenzvariablen deklarieren	155
4.4.2	Objekte erzeugen	156
4.4.3	Objekte initialisieren über Konstruktoren	157
4.4.4	Abräumen überflüssiger Objekte durch den Garbage Collector	160
4.4.5	Objektreferenzen verwenden	161
4.4.5.1	Objektreferenzen als Wertparameter	161
4.4.5.2	Rückgabewerte mit Referenztyp	162
4.4.5.3	this als Referenz auf das aktuelle Objekt	162
4.5	Eigenschaften	163

4.6	Statische Member und Klassen	165
4.6.1	Statische Felder und Eigenschaften	165
4.6.2	Wiederholung zur Kategorisierung von Variablen	166
4.6.3	Statische Methoden	167
4.6.4	Statische Konstruktoren	168
4.6.5	Statische Klassen	169
4.7	Vertiefungen zum Thema Methoden	169
4.7.1	Rekursive Methoden	169
4.7.2	Operatoren überladen	171
4.8	Aggregieren und innere Klassen	172
4.8.1	Aggregation	172
4.8.2	Innere Klassen	175
4.9	Verfügbarkeit von Klassen und Klassenbestandteilen	176
4.10	Bruchrechnungsprogramm mit WinForms-Benutzerschnittstelle	177
4.10.1	Projekt anlegen mit Vorlage <i>Windows Forms - Anwendung</i>	177
4.10.2	Steuerelemente aus der Toolbox übernehmen	178
4.10.3	Steuerelemente gestalten	179
4.10.4	Eigenschaften der Steuerelemente ändern	181
4.10.5	Der Quellcode-Generator	182
4.10.6	Assembly mit der Bruch-Klasse einbinden	183
4.10.7	Ereignisbehandlungsmethoden anlegen	184
4.11	Übungsaufgaben zu Kapitel 4	185
5	WEITERE .NETTE TYPEN	191
5.1	Strukturen	191
5.1.1	Vergleich von Klassen und Strukturen	192
5.1.2	Zur Eignung von Strukturen im Bruchrechnungs-Projekt	195
5.1.3	Strukturen im Common Type System der .NET – Plattform	197
5.2	Boxing und Unboxing	198
5.3	Arrays	200
5.3.1	Array-Referenzvariablen deklarieren	201
5.3.2	Array-Objekte erzeugen	202
5.3.3	Arrays benutzen	203
5.3.4	Beispiel: Beurteilung des .NET - Pseudozufallszahlengenerators	204
5.3.5	Initialisierungslisten	206
5.3.6	Objekte als Array-Elemente	207
5.3.7	Mehrdimensionale Arrays	207
5.3.7.1	Rechteckige Arrays	207
5.3.7.2	Mehrdimensionale Arrays mit unterschiedlich großen Elementen	208
5.3.8	Die Kollektionsklasse ArrayList	209
5.4	Klassen für Zeichenketten	211
5.4.1	Die Klasse String für konstante Zeichenketten	211
5.4.1.1	String als WORM - Klasse	211
5.4.1.2	Interner String-Pool	212
5.4.1.3	Methoden für String-Objekte	214
5.4.2	Die Klasse StringBuilder für veränderliche Zeichenketten	217
5.5	Enumerationen	218
5.6	Indexer	220

5.7	Übungsaufgaben zu Kapitel 5	223
6	VERERBUNG UND POLYMORPHIE	227
6.1	Das Common Type System (CTS) des .NET – Frameworks	228
6.2	Definition einer abgeleiteten Klasse	229
6.3	base-Konstruktoren und Initialisierungs-Sequenzen	230
6.4	Der Zugriffsmodifikator protected	232
6.5	Erbstücke durch spezialisierte Varianten verdecken	233
6.5.1	Geerbte Methoden verdecken	233
6.5.2	Geerbte Felder verdecken	235
6.6	Verwaltung von Objekten über Basisklassenreferenzen	236
6.7	Polymorphie (Methoden überschreiben)	238
6.8	Abstrakte Methoden und Klassen	240
6.9	Versiegelte Methoden und Klassen	242
6.10	Übungsaufgaben zu Kapitel 6	242
7	GENERISCHE TYPEN UND METHODEN	245
7.1	Motive für die Einführung generischer Klassen in .NET 2.0	245
7.2	Generische Klassen	246
7.2.1	Definition	247
7.2.2	Restringierte Typformalparameter	248
7.2.3	Generische Klassen und Vererbung	250
7.3	Nullable<T> als Beispiel für generische Strukturen	251
7.4	Generische Methoden	253
7.5	Übungsaufgaben zu Kapitel 7	254
8	INTERFACES	255
8.1	Interfaces definieren	257
8.2	Interfaces implementieren	258
8.3	Interfaces als Referenzdatentypen	263
8.4	Explizite Interface-Implementationen	264
8.5	Übungsaufgaben zu Kapitel 8	265

9	EINSTIEG IN DIE WINFORMS - PROGRAMMIERUNG	267
9.1	Ein Projektrahmen zum Üben	269
9.2	Elementare Klassen im Windows Forms - Framework	270
9.2.1	Anwendungsfenster und die Klasse Form	271
9.2.2	Windows-Nachrichten und die Klasse Application	274
9.3	Delegaten und Ereignisse	276
9.3.1	Delegaten	276
9.3.1.1	Delegatentypen definieren	277
9.3.1.2	Delegatenobjekte erzeugen und aufrufen	278
9.3.1.3	Delegatenobjekte kombinieren	279
9.3.1.4	Anonyme Methoden	280
9.3.1.5	Generische Delegaten	281
9.3.2	Ereignisse	282
9.3.2.1	Behandlungsmethoden registrieren	283
9.3.2.2	Ereignisse anbieten	286
9.4	Steuerelemente	288
9.4.1	Einsatz von Steuerelementen am Beispiel des Labels	289
9.4.1.1	Steuerelemente erzeugen und konfigurieren	290
9.4.1.2	Ab in den Container	291
9.4.1.3	Ereignisbehandlungsmethoden	292
9.4.2	Befehlsschalter	293
9.4.2.1	Beispiel	293
9.4.2.2	Eingabefokus und Standardschaltfläche	295
9.4.2.3	Bitmaps auf Schaltflächen	297
9.4.3	Textfelder	298
9.4.4	Spezielle Layout-Techniken	300
9.4.4.1	Z-Anordnung	300
9.4.4.2	Verankern	301
9.4.4.3	Andocken	303
9.4.4.4	TableLayoutPanel	304
9.5	Ereignisse und On-Methoden	306
9.6	WinForms - RAD	308
9.6.1	Projekt anlegen mit Vorlage <i>Windows Forms - Anwendung</i>	308
9.6.2	Steuerelemente aus der Toolbox übernehmen	309
9.6.3	Eigenschaften der Steuerelemente ändern	310
9.6.4	Der Quellcode-Generator	311
9.6.5	Ereignisbehandlungsmethoden anlegen	313
9.6.6	Quellcode-Umgestaltung (Refactoring)	315
9.7	Zusammenfassung zu Abschnitt 9	317
9.8	Übungsaufgaben zu Kapitel 9	318

10	AUSNAHMEBEHANDLUNG	319
10.1	Unbehandelte Ausnahmen	319
10.2	Ausnahmen abfangen	321
10.2.1	Die try-catch-finally - Anweisung	321
10.2.1.1	Ausnahmebehandlung per catch-Block	322
10.2.1.2	finally	324
10.2.2	Programmablauf bei der Ausnahmebehandlung	326
10.2.2.1	Beispiel	326
10.2.2.2	Komplexe Fälle	328
10.3	Ausnahmeobjekte im Vergleich mit der traditionellen Fehlerbehandlung	328
10.4	Ausnahme-Klassen im .NET - Framework	330
10.5	Ausnahmen werfen (throw)	332
10.6	Ausnahmen definieren	333
10.7	Übungsaufgaben zu Kapitel 10	336
11	ATTRIBUTE	339
11.1	Attribute vergeben	339
11.2	Attribute per Reflexion auswerten	341
11.3	Attribute definieren	344
11.4	Attribute für Assemblies und Module	344
11.5	Eine Auswahl nützlicher FCL-Attribute	346
11.5.1	Bitfelder per FlagsAttribute	346
11.5.2	Unions per StructLayoutAttribute und FieldOffsetAttribute	347
11.6	Übungsaufgaben zu Kapitel 11	348
12	EIN-/AUSGABE ÜBER DATENSTRÖME	349
12.1	Datenströme aus Bytes	349
12.1.1	Das Grundprinzip	349
12.1.2	Wichtige Methoden und Eigenschaften der Basisklasse Stream	350
12.1.3	Schließen von Datenströmen	351
12.1.4	Ausnahmen abfangen	353
12.1.5	FileStream	353
12.1.5.1	Öffnungsmodus	354
12.1.5.2	Zugriffsmöglichkeiten für den eigenen Prozess	354
12.1.5.3	Optionen für gemeinsamen Zugriff	355
12.2	Verarbeitung von Daten mit höherem Typ	355
12.2.1	Schreiben und Lesen im Binärformat	356
12.2.2	Schreiben und Lesen im Textformat	358
12.2.3	Serialisieren von Objekten	362

12.3	Verwaltung von Dateien und Verzeichnissen	367
12.3.1	Dateiverwaltung	367
12.3.2	Ordnerverwaltung	369
12.3.3	Überwachung von Ordnern	370
12.4	Übungsaufgaben zu Kapitel 12	371
13	THREADS	373
13.1	Threads erzeugen	374
13.2	Threads synchronisieren	378
13.2.1	Die lock-Anweisung	379
13.2.2	Die Klasse Monitor	380
13.2.3	Koordination per Wait() und Pulse()	381
13.3	Threads stoppen	383
13.4	Thread-Lebensläufe	384
13.4.1	Scheduling und Prioritäten	384
13.4.2	Zustände von Threads	385
13.5	Deadlock	386
13.6	Treadpool und APM	387
13.6.1	Die ThreadPool-Methode QueueUserWorkItem()	387
13.6.2	Die Delegatenmethoden BeginInvoke() und EndInvoke()	388
13.7	Timer	391
13.8	Übungsaufgaben zu Kapitel 13	395
14	NETZWERKPROGRAMMIERUNG	397
14.1	Wichtige Konzepte der Netzwerktechnologie	397
14.1.1	Das OSI-Modell	398
14.1.2	Zur Funktionsweise von Protokollstapeln	401
14.1.3	Optionen zur Netzwerkprogrammierung in C#	402
14.2	Internet - Ressourcen per Request/Response – Modell nutzen	402
14.2.1	Statische Webinhalte anfordern	402
14.2.2	Datei-Download per HTTP - Protokoll	405
14.2.3	Dynamische erstellte Webseiten per GET oder POST anfordern	407
14.2.3.1	Überblick	407
14.2.3.2	Arbeitsablauf	407
14.2.3.3	GET	409
14.2.3.4	POST	411
14.3	IP-Adressen bzw. Host-Namen ermitteln	412
14.4	Socket-Programmierung	413
14.4.1	TCP-Server	414
14.4.2	TCP-Klient	416
14.4.3	Simultane Bedienung mehrerer Klienten	417
14.5	Übungsaufgaben zu Kapitel 14	419

15	GRAFIKAUSGABE MIT DEM GDI+	423
15.1	Die Klasse Graphics	423
15.2	GDI-Ressourcen schonen	425
15.3	Fensterrenovierungsmethoden	425
15.3.1	PaintEventHandler	426
15.3.2	OnPaint() überschreiben	427
15.3.3	Fensteraktualisierung per Programm anfordern	428
15.3.4	ResizeRedraw	429
15.4	Positionen und Größen	430
15.4.1	Standardkoordinatensystem	430
15.4.2	Point und Size	431
15.4.3	Rectangle	433
15.5	Farben und Zeichenwerkzeuge	434
15.5.1	Farben	434
15.5.2	Pen	436
15.5.3	Brush	438
15.6	Linien, Kurven und Flächen	440
15.6.1	Einfache Linien und Figuren	440
15.6.2	Splines	443
15.6.3	Flächen	445
15.7	Text malen	447
15.7.1	Schriftarten und -familien	448
15.7.2	Standarddialog zur Schriftauswahl	451
15.7.3	Zeichen setzen	453
15.7.4	Maß halten	456
15.7.4.1	Schrifthöhe und Zeilenabstand	456
15.7.4.2	Textbreite	458
15.8	Transformationen	460
15.8.1	Auflösungserscheinungen	461
15.8.2	Metrische Maßeinheiten (Seitentransformation)	462
15.8.3	Welttransformationen	464
15.9	Rastergrafik	465
15.10	Pfade, Zeichen- und Anzeigebereiche	469
15.10.1	Grafikpfade	469
15.10.2	Zeichenbereiche	473
15.10.3	Anzeigebereiche	475
15.11	Übungsaufgaben zu Kapitel 15	476
16	WEITERE STANDARDKOMPONENTEN FÜR WINFORMS-ANWENDUNGEN	479
16.1	Kontrollkästchen und Optionsfelder	479
16.2	Listen- und Kombinationsfelder	482
16.3	UpDown-Regler	485
16.4	Rollbalken	488
16.5	Ein RTF-Editor auf RichTextBox-Basis	490

16.6	Timer-Komponente	491
17	MENÜS	493
17.1	Wichtige Klassen und Begriffe	493
17.2	Menüitems erzeugen und konfigurieren	495
17.3	Symbole für Menüitems	498
17.4	Verfügbarkeit von Menüitems dynamisch anpassen	500
17.5	Kontextmenü	501
17.6	Unterstützung durch die Entwicklungsumgebungen	502
17.7	Die Menütechnik aus .NET 1	506
17.7.1	Wichtige Klassen und Begriffe	507
17.7.2	Menüitems erzeugen	508
17.7.3	Kontextmenü	511
17.8	Übungsaufgaben zu Kapitel 17	512
18	DIALOGFENSTER	513
18.1	Modale Dialogfenster	513
18.1.1	Konfiguration	514
18.1.2	Datenaustausch mit dem aufrufenden Formular	516
18.1.3	Auftritt und Abgang	517
18.1.4	Übernehmen Sie keinen Aberglauben	518
18.2	Nicht-modale Dialogfenster	520
18.2.1	Konfiguration	520
18.2.2	Auftritt und Abgang	522
18.2.3	Kommunikation mit dem Anwendungsfenster	523
18.3	Standarddialoge	525
18.3.1	FontDialog	525
18.3.2	ColorDialog	527
18.3.3	SaveFileDialog	528
18.3.4	OpenFileDialog	530
18.4	Übungsaufgaben zu Kapitel 18	532
19	ANWENDUNGS- UND BENUTZEREINSTELLUNGEN	533
19.1	Konfigurationsdateien im XML-Format	534
19.1.1	Aufbau einer .NET - Anwendungskonfigurationsdatei	534
19.1.2	Ablage im Dateisystem	536
19.1.3	Die Klasse ApplicationSettingsBase	538
19.1.4	Lesen und Speichern von Einstellungen	540
19.1.5	Bindung von Einstellungen an Eigenschaften von Komponenten	542
19.1.6	Validierung von Einstellungen	542
19.1.7	Unterstützung durch die Entwicklungsumgebungen	543
19.1.8	Weitere Optionen der .NET - Konfigurationsdateien	548
19.2	Windows-Registrierungsdatenbank	548
19.3	Übungsaufgaben zu Kapitel 19	552

20	SYMBOL- UND STATUSLEISTE	553
20.1	Symbolleisten	553
20.1.1	Do-It-Yourself - Erstellung einer Symbolleiste	554
20.1.2	Bewegliche Leisten und ToolStripContainer	557
20.1.3	Unterstützung durch die Entwicklungsumgebungen	558
20.2	Statusleiste	561
20.3	Übungsaufgaben zu Kapitel 20	565
21	DRUCKEN	567
21.1	Wichtige Datentypen	567
21.1.1	Die Klasse PrinterSettings zur Verwaltung von Druckereinstellungen	567
21.1.2	Die Klasse PageSettings zur Verwaltung von Seiteneigenschaften	569
21.1.3	Die Klasse PrintDocument zur Verwaltung von Druckaufträgen	570
21.1.3.1	Eigenschaften und Methoden	571
21.1.3.2	Ereignisse	573
21.1.3.3	Beispiel	574
21.1.4	Der Standarddialog Drucken	576
21.2	Druckfunktionalität für den Editor auf RichTextBox-Basis	578
21.2.1	Beteiligte Objekte	579
21.2.2	BeginPrint - Ereignisbehandlung	581
21.2.3	PrintPage - Ereignisbehandlung	581
21.2.3.1	PrintPageEventArgs-Eigenschaften	581
21.2.3.2	Ausgaberechteck ermitteln	582
21.2.3.3	Textausgabe mit DrawString()	585
21.3	Weitere Standarddialoge zur Unterstützung der Druckausgabe	587
21.3.1	Seitenansicht	587
21.3.2	Seiteneinrichtung	589
21.4	Übungsaufgaben zu Kapitel 21	591
22	ZWISCHENABLAGE, ZIEHEN UND ABLEGEN	593
22.1	Verbesserte RTF-Unterstützung im Editorprojekt	593
22.2	Windows-Zwischenablagenformate	594
22.3	Ein Zwischenablagen-Inspektor	595
22.4	Daten in die Zwischenablage schreiben	597
22.4.1	Bequeme Clipboard-Methoden für Routineaufgaben	597
22.4.2	Die Clipboard-Methode SetDataObject() für spezielle Aufgaben	600
22.5	Daten aus der Zwischenablage abrufen	602
22.5.1	Bequeme Clipboard-Methoden für Routineaufgaben	602
22.5.2	Die Clipboard-Methode GetDataObject()	604
22.6	Ziehen und Ablegen	605
22.6.1	Die Rolle der Ziellanwendung	606
22.6.2	Die Rolle der Quellanwendung	609
22.6.3	Drag & Drop - Funktionalität für unseren Editor	611
22.7	Übungsaufgaben zu Kapitel 22	612

23	RESSOURCEN	613
23.1	Ressourcendateien und -generatoren	613
23.1.1	RESX-Datei	613
23.1.2	TXT-Datei für Zeichenfolge-Ressourcen	615
23.1.3	RESOURCES-Datei	615
23.2	Ressourcen verwenden	616
23.3	Ressourcen-Verwaltung mit der Visual C# 2008 Express Edition	617
23.4	Übungsaufgaben zu Kapitel 23	620
24	INTEROPERABILITÄT MIT SOFTWARE-ALTLASTEN	621
24.1	Funktionen in nativen DLLs nutzen	621
24.1.1	DLLs importieren	621
24.1.2	Parameter-Marshaling	622
24.1.3	Windows-Fehlercode ermitteln	624
24.2	Kooperation mit der COM-Technologie	624
24.2.1	Einige Begriffe und Ideen aus der COM-Technologie	624
24.2.1.1	COM-Server	625
24.2.1.2	COM-Schnittstellen und -Typbibliotheken	625
24.2.1.3	Aufruf einer COM-Methode	626
24.2.1.4	Frühe und späte Bindung	627
24.2.1.5	Registrierung	627
24.2.2	COM-Server in .NET - Anwendungen nutzen	628
24.2.2.1	Proxy-Assembly mit Runtime Callable Wrapper	628
24.2.2.2	COM-Server mit C++ erstellen	629
24.2.2.3	COM-Server in C# verwenden	636
25	VARIABLE ANSICHTEN	643
25.1	Aufteilungssteuerelemente	643
25.1.1	Splitter	643
25.1.2	SplitContainer	646
25.2	Baumansicht	648
25.2.1	Konstruktion	648
25.2.2	Datenerfassung	650
25.2.3	Verzeichnisbaum	651
25.2.4	Bilder für die Knoten	652
25.2.5	Ereignisse	654
25.2.6	Explorer Light	655
25.3	Listen und Tabellen	657
25.3.1	Konstruktion	658
25.3.2	Datenerfassung	660
25.3.3	Ereignisse	661
25.3.4	Dateisystem-Verzeichniseinträge auflisten	661
25.3.5	Bilder für die Listenitems	664
25.3.6	Spezielle Sortierung	665
25.3.7	Explorer Midrange	666

26	DATENBANKPROGRAMMIERUNG MIT ADO.NET	669
26.1	Relationale Datenbanken	671
26.1.1	Tabellen	671
26.1.2	Beziehungen zwischen Tabellen	672
26.2	SQL	673
26.2.1	Überblick	673
26.2.2	Spalten abrufen	674
26.2.3	Fälle auswählen über die WHERE-Klausel	675
26.2.4	Daten aus mehreren Tabellen zusammenführen	675
26.2.5	Abfrageergebnis sortieren	676
26.2.6	Auswertungsfunktionen	676
26.2.7	Daten aggregieren	676
26.3	Microsoft SQL Server 2008 Express Edition	676
26.3.1	Eigenschaften	676
26.3.2	Installation	677
26.3.2.1	SQL Server 2008 Express with Tools	677
26.3.2.2	Beispieldatenbank Northwind	684
26.3.3	Konfiguration	685
26.3.3.1	Rechteverwaltung	685
26.3.3.2	Netzwerkzugriff via TCP/IP erlauben	688
26.4	ADO.NET	690
26.4.1	Überblick	690
26.4.1.1	Verbindungsloses versus verbindungsorientiertes Arbeiten	690
26.4.1.2	Provider	691
26.4.1.3	ADO.NET - Namensräume	692
26.4.2	Die Connection-Klassen	692
26.4.2.1	Verbindungszeichenfolge	692
26.4.2.2	Eigenschaften, Ereignisse und Methoden	696
26.4.3	Die Command-Klassen	697
26.4.3.1	Eigenschaften und Methoden	697
26.4.3.2	Parametrisierte Abfragen	698
26.4.4	Die DataAdapter-Klassen	699
26.4.5	DataSet und andere Provider-unabhängige Klassen	701
26.4.5.1	Abfrageergebnisse und Schema-Informationen aus einer Datenbank übernehmen	701
26.4.5.2	DataTable-Objekte modifizieren	703
26.4.5.3	Beziehungen zwischen Tabellen vereinbaren	708
26.4.6	Datenbank-Update	709
26.4.7	Zusammenspiel der ADO.NET - Klassen beim verbindungslosen Arbeiten	711
26.4.8	Einsatz des DataReaders	713
26.5	Datenbankunterstützung durch die Entwicklungsumgebungen	713
26.5.1	Datenquelle	713
26.5.2	Anwendungsbeispiel	715
26.5.3	Datenbanken mit der Entwicklungsumgebung bearbeiten	723
26.5.3.1	Tabellen anzeigen und bearbeiten	724
26.5.3.2	Datenbankdiagramm erstellen	725
26.5.3.3	Beziehungen definieren	726

27	LINQ	729
27.1	Erweiterungen der Programmiersprache C#	729
27.1.1	Implizite Typzuweisung bei lokalen Variablen	729
27.1.2	Instanz- und Listeninitialisierer	729
27.1.3	Anonyme Klassen	730
27.1.4	Lambda-Ausdrücke	730
27.1.5	Erweiterungsmethoden	731
27.2	LINQ to Objects	732
27.2.1	Die from-Klausel zur Definition einer Datenquelle	732
27.2.2	Die Projektion des Elementtyps der Quelle auf den Elementtyp der Abfrage	733
27.2.3	Formulierung eines Auswahlkriteriums	734
27.2.4	Weitere LINQ-Operationen	734
27.3	LINQ to SQL	734
27.3.1	Elementare LINQ-to-SQL - Klassen und O/R - Designer	735
27.3.2	Datenbankbearbeitung via LINQ to SQL	738
27.3.2.1	Werte ändern	738
27.3.2.2	Neue Entitäten (Tabellenzeilen) anlegen	739
27.3.2.3	Entitäten (Tabellenzeilen) löschen	739
28	ANHANG	741
28.1	Operatorentabelle	741
28.2	Lösungsvorschläge zu den Übungsaufgaben	742
	Kapitel 1 (Einleitung)	742
	Kapitel 2 (Werkzeuge zum Entwickeln von C# - Programmen)	744
	Kapitel 3 (Elementare Sprachelemente)	745
	Abschnitt 3.1 (Einstieg)	745
	Abschnitt 3.2 (Ausgabe bei Konsolenanwendungen)	745
	Abschnitt 3.3 (Variablen und Datentypen)	745
	Abschnitt 3.5 (Operatoren und Ausdrücke)	746
	Abschnitt 3.7 (Anweisungen)	747
	Kapitel 4 (Klassen und Objekte)	749
	Kapitel 5 (Weitere .NET-Typen)	750
	Kapitel 6 (Vererbung und Polymorphie)	751
	Kapitel 7 (Generische Typen und Methoden)	752
	Kapitel 8 (Schnittstellen)	753
	Kapitel 9 (Einstieg in die WinForms - Programmierung)	753
	Kapitel 10 (Ausnahmebehandlung)	754
	Kapitel 11 (Attribute)	754
	Kapitel 12 (Ein-/Ausgabe über Datenströme)	754
	Kapitel 13 (Threads)	755
	Kapitel 14 (Netzwerkprogrammierung)	755
	Kapitel 15 (Grafikausgabe mit dem GDI+)	755
	Kapitel 17 (Menüs)	756
	Kapitel 18 (Dialogfenster)	757
	Kapitel 19 (Anwendungs- und Benutzereinstellungen)	757
	Kapitel 20 (Symbol- und Statusleiste)	757
	Kapitel 21 (Drucken)	757
	Kapitel 22 (Zwischenablage, Ziehen und Ablegen)	757
	Kapitel 23 (Ressourcen)	757

Inhaltsverzeichnis	XIX
LITERATUR	759
STICHWORTREGISTER	761

1 Einleitung

Im ersten Kapitel geht zunächst es um die Denk- und Arbeitsweise (leicht übertrieben: die *Weltanschauung*) der objektorientierten Programmierung. Danach werden die .NET - Plattform und ihre wohl wichtigste Programmiersprache C# als Software-Technologie 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 Bildern, Klängen 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. Speicher) 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, Fahrzeit, 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 allerdings 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 **Klassen** und deren Kommunikation untereinander. 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 Merkmale gehören zu den *Objekten* bzw. *Instanzen* der Klasse (z.B. Zähler und Nenner eines Bruchs), manche gehören zur Klasse selbst (z.B. Anzahl der bereits erzeugten Brüche). Im letztlich entstehenden Programm landet jedes Merkmal in einer so genannten *Variablen*. Dies ist ein benannter Speicherplatz, der Werte eines bestimmten Typs (z.B. Zahlen, Zei-

chen) 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 kann man sich 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).

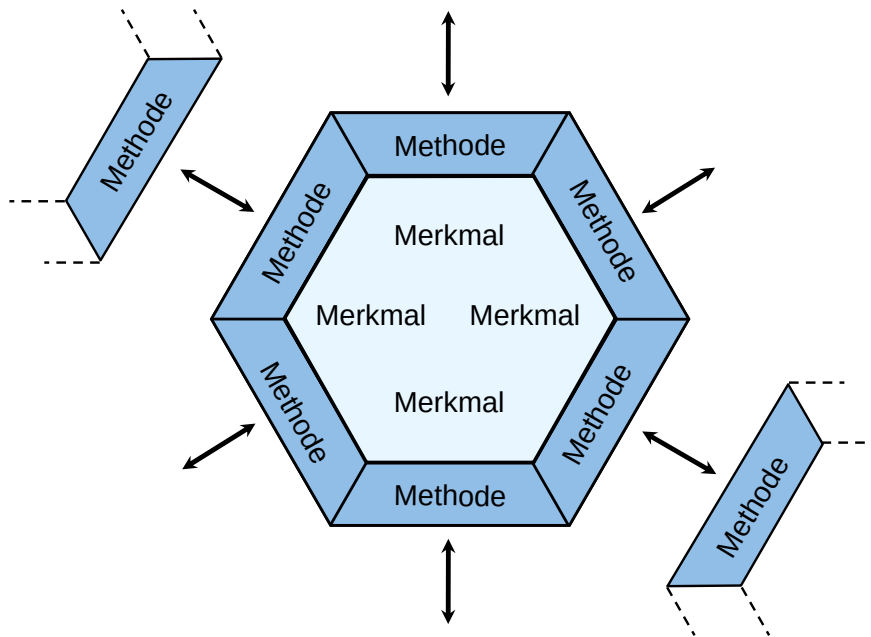
Dass Zähler und Nenner die zentralen **Merkmale** eines Bruchs sind, bedarf keiner Begründung. Sie werden in der Klassendefinition durch ganzzahlige Felder (C# - Datentyp **int**) repräsentiert:

- `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 Zustände (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. Außerdem wird aus später zu erläuternden Gründen die Produktivität der Softwareentwicklung durch die Datenkapselung gefördert.

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 Gegenstandsbereiches 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 Objekten aus 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 *Lesezugriff* 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 Setzen 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 – Laufzeitbibliothek (siehe Abschnitt 1.2.6), von denen alle C# - Programmierer regen Gebrauch machen. 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 verwendet 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 *Zugriffsmethoden* 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 *Zugriffsmethoden*, 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 Beispiel wohl trotz den angesprochenen didaktischen Probleme gerechtfertigt.

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

- `Nenner` (Eigenschaft zu `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.
- `Zaehler` (Eigenschaft zu `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 bezeichnet die in einer Klasse definierten Bestandteile auch als **Member**. Unsere `Bruch`-Klasse enthält folgende Member:

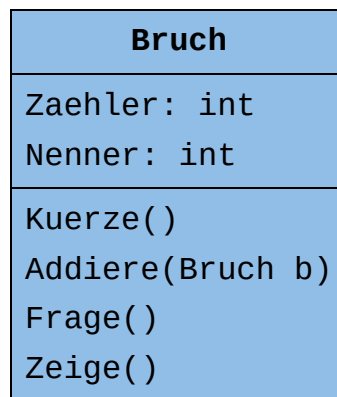
- **Felder**
`zaehler`, `nenner`
- **Eigenschaften** (Hier handelt es sich letztlich um Paare von Methoden.)
`Zaehler`, `Nenner`
- **Methoden**
`Kuerze()`, `Addiere()`, `Frage()` und `Zeige()`

Von kommunizierenden Objekten mit Handlungskompetenzen zu sprechen, mag als übertriebener Anthropomorphismus (als Vermenschlichung) erscheinen. Bei der Ausführung von Methoden sind Objekte 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 Abschnitten 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 *Zugriffsmethoden*, 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):



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-Entwicklerwerkzeuge Teile des Quellcodes automatisch aus dem Modell erzeugen können.¹

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;
        }
    }
}
```

¹ Microsofts Entwicklerwerkzeuge haben in den letzten Jahren wenig UML- oder sonstige Modellierungsunterstützung geboten, doch gibt es aktuell (in 2008) deutliche Hinweise auf eine UML-Unterstützung in der kommenden Version der Premium-Variante *Visual Studio Team Systems*.

```

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;
    } else
        nenner = 1;
}

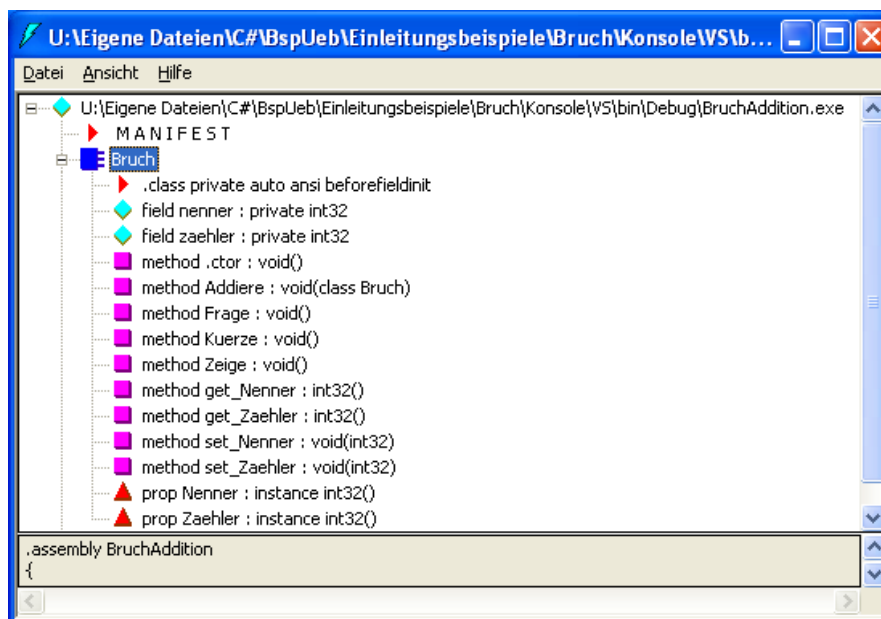
public void Addiere(Bruch b) {
    zaehler = zaehler*b.nenner + b.zaehler*nenner;
    nenner = nenner*b.nenner;
    Kuerze();
}

public void Frage() {
    Console.Write("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. Demgegenüber werden die Eigenschaften und Methoden über den Modifikator **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 im kostenlos verfügbaren *Windows-SDK* der Firma Microsoft (siehe Abschnitt 2.2.3) enthaltene 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 Gegenstandsbereichs* 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:

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

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

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

Folglich ist t auch ein Teiler von v :

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

Weil die Paare (u, v) und $(u, u - v)$ 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 immer ein größter gemeinsamer Teiler, der eventuell gleich Eins ist.

Dieses Ergebnis wird in `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	Ein- und Ausgabe
<pre>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(); } }</pre>	<pre>1. Bruch Zaehler: 20 Nenner : 84 5 ---- 21 2. Bruch Zaehler: 12 Nenner : 36 1 ---- 3 Summe 4 ---- 7</pre>

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\Einleitungsbeispiele\Bruch\Konsole\VS\bin\Debug\BruchAddi...
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 die Kompetenzen der Klasse **Console** aus der Standardbibliothek und rufen ihre Klassenmethode **WriteLine()** auf.

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

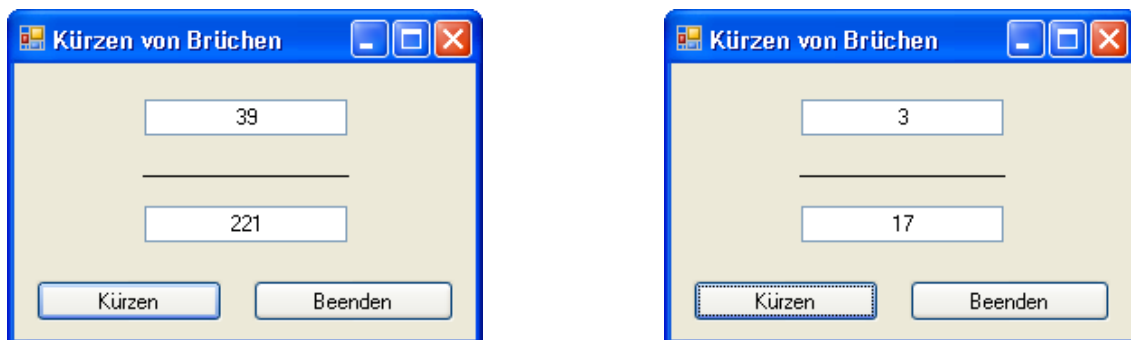
- Die Klasse **Bruch** enthält den Bauplan für die wesentlichen Akteure im Aufgabenbereich. Dort alle Merkmale und Handlungskompetenzen von Brüchen zu konzentrieren, hat folgende Vorteile:
 - Die Klasse kann in verschiedenen Programmen eingesetzt werden (Wiederverwendbarkeit). Dies fällt vor allem deshalb so leicht, weil die Objekte Handlungskompetenzen (Methoden) besitzen **und** alle erforderlichen Instanzvariablen mitbringen. Wir müssen bei der Definition dieser Klasse ihre allgemeine Verfügbarkeit explizit mit dem Zugriffsmodifikator **public** genehmigen. Per Voreinstellung ist eine Klasse nur intern (in der eigenen Übersetzungseinheit) verfügbar.

- o 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.
- 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 Namenserverweiterung **.cs**. Das .NET - Framework erlaubt aber auch Quellcodedateien mit mehreren Klassen und beliebigem Namen.

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\\Klassen und Objekte\\Bruch\\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 bzw. 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

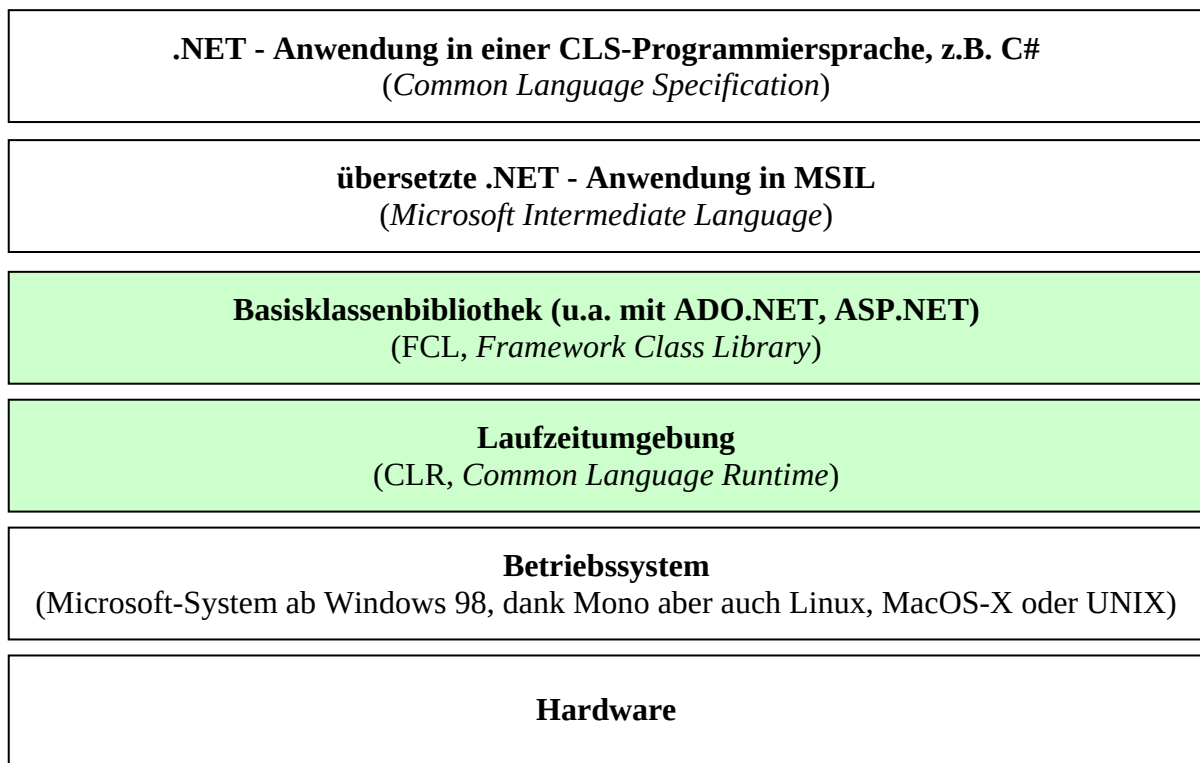
einer CPU aus der x86-Familie wird z.B. der Wert Vier in das EAX-Register (ein Speicherort im Prozessor) geschrieben. Die CPU holt sich einen Maschinenbefehl nach dem anderen aus dem Hauptspeicher und führt ihn aus, heutzutage immerhin mehrere Milliarden Befehle pro Sekunde (*Instructions Per Second, 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 im .NET – Kontext verfügbaren 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), welche die bisherige Windows-Programmierung über das Win32-API oder die COM-Technologie (*Common Objekt Model*) modernisieren und insbesondere vereinfachen soll. Außerdem soll eine Konkurrenz zur Java-Plattform der Firma Sun geschaffen werden, die bei vielen IT-Firmen und Entwicklern hohes Ansehen genießt.

Als Orientierung für die nächsten Abschnitte kann die folgende, stark vereinfachte Darstellung des .NET - Frameworks dienen:



Seit der Windows-Version *Vista* ist das .NET – Framework fester Systembestandteil. Bei älteren Versionen (ab Windows 98) kann ein bei Microsoft kostenlos verfügbares .NET Framework - Paket (meist als *Redistributable* bezeichnet) nachinstalliert werden (zur Beschaffung und Installation siehe Abschnitt 1.2.2). Dieses Paket darf auch 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 (Beta)	2009

Wir werden im Kurs lange Zeit mit der Version 2.0 auskommen, die derzeit (2009) als Standard gelten kann, aber z.B. bei der Datenbankprogrammierung auch die neuen Möglichkeiten der Version 3.5 nutzen (*Language Integrated Query*, LINQ). Die mit Windows Vista ausgelieferte .NET - Version 3.0 hat als Neuerung vor allem das *Windows Presentations Framework* (WPF) gebracht. Dieses Framework für multimedial ambitionierte Bedienoberflächen wird im Kurs keine Rolle spielen (vgl. Abschnitt 9).

Das .NET-Framework und die Programmiersprache C# besitzen eine eigenständige Versionierung, wobei die beiden Zeitreihen aber stark korrelieren, wie die folgende Tabelle (mit Prognose der näheren Zukunft) aus Schwichtenberg (2009a) zeigt:

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

.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# 2008 Express Edition* (siehe Abschnitt 2.2.1) landet das .NET - Framework (in der Version 3.5) 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) doch fehlt, kann man es kostenlos (z.B. via Internet bei Microsoft¹) beschaffen und nachinstallieren. Neben der für die meisten Klienten-rechner relevanten 32-Bit – Version (Plattform **x86**) unterstützt Microsoft auch die Plattformen **x64** (64-Bit – Windows auf einem Standard-PC mit 64-Bit - CPU) und **ia64** (64-Bit – Windows auf einem Itanium-System).

Das Paket zur Framework-Version 2.0 für die x86-Plattform

Microsoft .NET Framework 2.0 Redistributable (x86)

benötigt einen Rechner mit ...

- Windows 98 (SE), Windows ME, Windows 2000 (mit SP 3), Windows XP (mit SP 2) oder Windows Server 2003.
- ca. 280 MB freiem Festplattenspeicher

Mit *Service Pack 1* ist das .NET - Framework 2.0 anspruchsvoller:

- Windows 2000 (mit SP 4), Windows XP (mit SP 2) oder Windows Server 2003.
- ca. 500 MB freiem Festplattenspeicher

Als Installationsordner dient bei .NET 2.0 (ohne Änderungsmöglichkeit):

%SystemRoot%\Microsoft.NET\Framework\v2.0.50727

Es ist sinnvoll, auch das deutsche Sprachpaket zu installieren:

Microsoft .NET Framework 2.0 Language Pack Deutsch (x86)

Zu der (in Windows 7 bereits enthaltenen) Framework-Version 3.5

Microsoft .NET Framework 3.5

bietet Microsoft einen Online-Installer, der die benötigten Installationsbestandteile während der Installation ermittelt und von einem Server lädt, sowie einen (deutlich größeren) Offline-Installer, der kompatible Systeme ohne Internetzugriff versorgen kann. Beide erwarten einen Zielrechner mit ...

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.

- Windows XP, Windows Vista, Windows Server 2003, Windows Server 2008
- ca. 200 - 600 MB freiem Festplattenspeicher (abhängig von den bereits installierten .NET - Versionen)

In der Version 3.5 ist die Bezeichnung *Redistributable* gestrichen worden. Außerdem ist kein separates Sprachpaket mehr erforderlich. Zudem werden freundlicherweise ältere Framework-Versionen gleich mit installiert. Dies ist sinnvoll, weil ältere .NET - Programme zur Vermeidung von Versionskonflikten stets von einer CLR auf ihrem eigenen Versionsstand ausgeführt werden. Als Installationsordner dient bei .NET 3.5 (ohne Änderungsmöglichkeit):

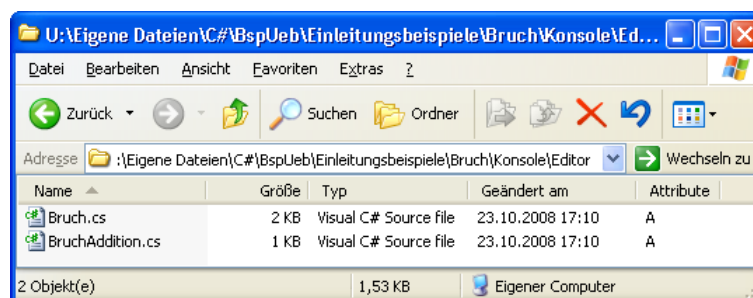
%SystemRoot%\Microsoft.NET\Framework\v3.5

Der Ordner zur .NET - Version 3.5 ist deutlich kleiner als der Ordner zu .NET 2.0, weil Microsoft bei unverändert weiterverwendbaren Dateien einer älteren Version auf ein Kopieren verzichtet.

1.2.3 C#-Compiler und MSIL

Den (z.B. mit einem beliebigen Editor verfassten) C# - Quellcode übersetzt der C# - **Compiler** im .NET - Framework 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 er sich gut für die Übertragung über Netzwerke. Die Übersetzung des Zwischencodes in die Maschinsprache einer konkreten CPU geschieht *just-in-time* bei der Ausführung des Programms durch die CLR (siehe Abschnitt 1.2.5).

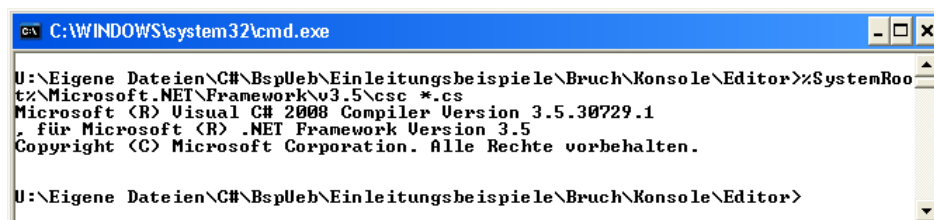
Befinden sich die beiden Quellcodedateien **Bruch.cs** und **BruchAddition.cs** im aktuellen Verzeichnis



eines Konsolenfensters, dann kann ihre Übersetzung durch den C# - Compiler **csc.exe** der .NET-Version 3.5 mit dem folgenden Kommando

%SystemRoot%\Microsoft.NET\Framework\v3.5\csc *.cs

veranlasst werden:



Weil sich der Ordner

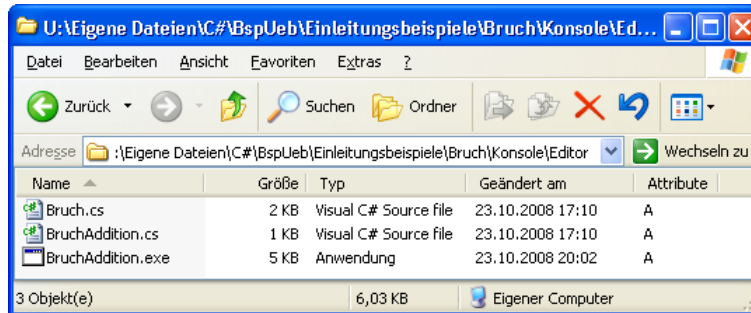
%SystemRoot%\Microsoft.NET\Framework\v3.5

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 Namensendung **.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**



ist u.a. der IL-Code der beiden Klassen 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()
.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)
.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. In gewissen Grenzen (siehe unten) können verschiedene Programmierer bei der Erstellung von Klassen für ein gemeinsames Projekt jeweils die individuell bevorzugte Programmiersprache verwenden. In der Java-Welt arbeiten alle Entwickler hingegen mit derselben Programmiersprache, was man als Vor- oder Nachteil betrachten kann.

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

Mittlerweile sind für viele Programmiersprachen MSIL-Compiler verfügbar, etliche werden sogar mit dem .NET - Framework - Paket ausgeliefert (z.B. **csc.exe** für C#). 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). Microsoft (2007a) hat unter dem Namen **Common Language Specification** (CLS) 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.

C# kann der großen Konkurrenz als die bevorzugte .NET - Programmiersprache gelten, weil schließlich das Framework selbst überwiegend in C# entwickelt wurde.

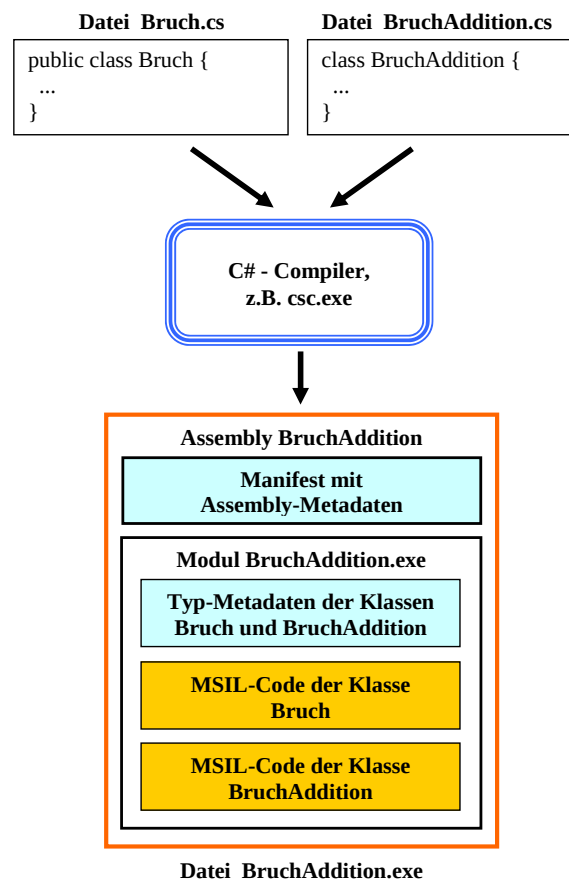
1.2.4 Assemblies und Metadaten

Die von einem .NET - Compiler (z.B. **csc.exe** bei C#) bei einem Aufruf erzeugten Binärdateien (mit MSIL-Code) werden als **Assemblies** bezeichnet und haben die Namenserweiterung:

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

Man übergibt dem Compiler im Allgemeinen *mehrere* Quellcodedateien mit jeweils einer Klassendefinition (siehe obiges Beispiel) 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.

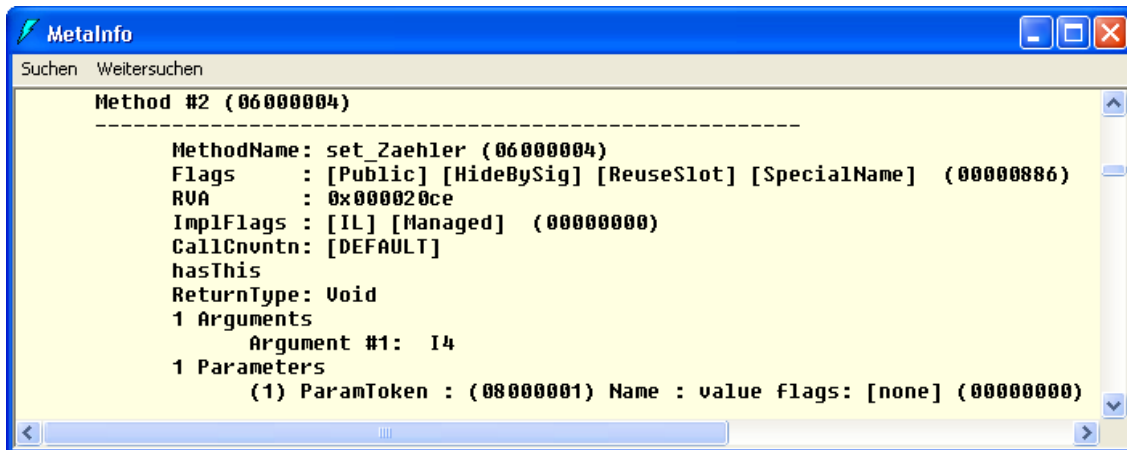
Die folgende Abbildung (nach Mössenböck 2003, S. 6) fasst wesentliche Informationen über Quellcode, C#-Compiler, MSIL-Code und Assemblies anhand des Bruchadditionsbeispiels zusammen und zeigt mit den anschließend zu beschriebenen Metadaten weitere wichtige Bestandteile eines .NET - Assemblies:



1.2.4.1 Typ-Metadaten

Neben dem IL-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



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

1.2.4.2 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.
- 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.

1.2.4.3 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 zunächst gefahrlos ignorieren dürfen. Es besteht aus mehreren **Moduldateien** mit Zwischencode und zugehörigen Metadaten. Das Manifest des gesamten Assemblies steckt

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

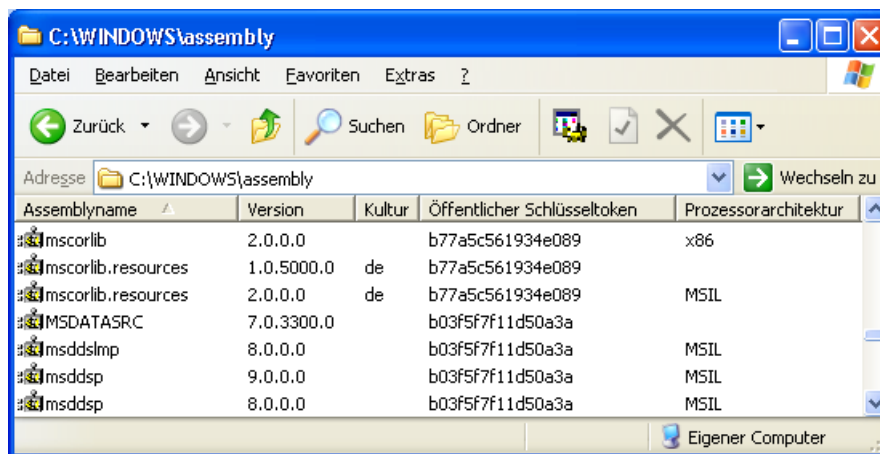
entweder in *einer* Moduldatei oder in einer separaten Datei. Diese Architektur hat z.B. dann Vorteile, wenn ein Klient über eine langsame Netzverbindung auf ein Assembly zugreift, weil sich der Transport auf die tatsächlich benötigten Module beschränken kann. Ohne die Aufteilung in Module müsste das gesamte Assembly transportiert werden. Für Moduldateien ohne eigenes Manifest wird meist die Namenserverweiterung **.netmodule** verwendet.

Wie die obige Abbildung zeigt, ist genau genommen auch beim Einzeldatei-Assembly ein Modul im Spiel. Dieses Einzelstück trägt denselben Namen wie das Assembly und enthält den MSIL-Code sowie die Metadaten seiner Typen.

1.2.4.4 Private und allgemeine Assemblies

Ein Assembly benötigt in der Regel weitere Assemblies, deren Typen in den eigenen Methoden 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.:



Wie das Beispiel **msddsp** zeigt, können im GAC gleichnamige Assemblies mit unterschiedlichen Versionsständen existieren. Eine nähere Beschäftigung mit dem GAC ist im ersten C# - Lernjahr nicht unbedingt erforderlich.

1.2.4.5 Vergleich mit der COM-Technologie

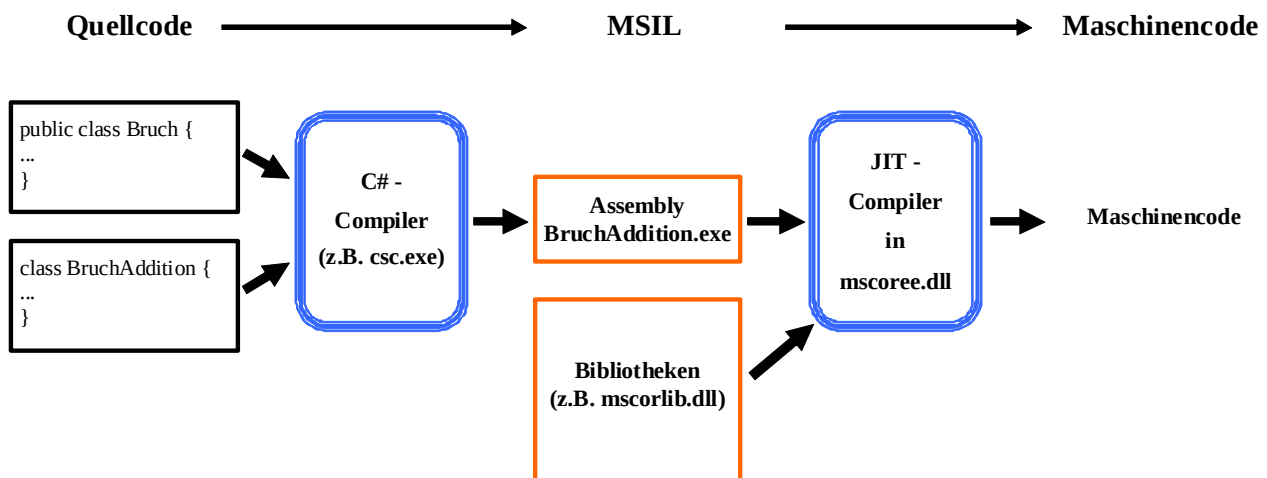
Die Metadaten der .NET -Technologie sorgen dafür, dass die Klassen eines Assemblies unproblematisch von beliebigen .NET - Programmen genutzt werden können. Oben wurde schon erwähnt, dass mit dem .NET – Framework die ältere COM-Architektur (*Component Object Model*) abgelöst werden soll, die in den 90er Jahren des letzten Jahrhunderts geschaffen wurde, um sprachübergreifende Interoperabilität von Software-Komponenten zu ermöglichen, und die auch heute noch von großer Bedeutung ist. 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 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 kein Registry-Eintrag erforderlich.

Weitere Informationen zu den .NET - Metadaten finden Sie z.B. bei Richter (2006, S. 62ff).

1.2.5 CLR und JIT-Compiler

Bislang haben Sie erfahren, dass aus dem C# - Quellcode durch einen Compiler (z.B. **csc.exe**) der sprach- und maschinenunabhängige MSIL-Code 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, wobei auch noch die Namen wichtiger Dateien des .NET – Frameworks vermerkt sind:



Sorgen um mangelnde Performanz aufgrund der indirekten Übersetzung sind übrigens unbegründet. Eine Untersuchung von Schäpers & Huttary (2003) ergab, dass die Zwischencode-Sprachen Java und C# bei diversen Benchmarks sehr gut mit den „echten Compiler-Sprachen“ C++ und Delphi mithalten können.

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:

- **Gewährleisten der Code Access Security (CAS)**
Beim Laden eines Assemblies stellt die CLR seine Zugriffsrechte (z.B. auf ein Dateisystem) aufgrund verschiedener Beweise (z.B. Herkunft, vorhandene digitale Signatur) fest und verhindert ein Überschreiten dieser Rechte. Während ein klassisches Windows-Programm ebenso viele Rechte hat wie der angemeldete *Benutzer*, orientiert sich die CAS an Merkmalen des jeweiligen Assemblies und bietet weitaus differenziertere Steuerungsmöglichkeiten.
- **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.

1.2.6 Namensräume und FCL

Damit Programmierer nicht das Rad (und ähnliche Dinge) ständig 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 – Programmiersprachen 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.

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.Text;

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

Bei kleinen Beispielprogrammen sind Namensräume jedoch überflüssig, und wir werden meist der Einfachheit 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, was zu unerwünschten Ergebnissen führen kann. 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**-Variablenamen 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**-Variablenamen 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.

Der geschickte Umgang mit Namensräumen zur Erleichterung der Softwareentwicklung ist in erster Linie Sache der .NET - Compiler. Die CLR entnimmt den Assemblies die vollständigen Typbezeichner und zerlegt diese *nicht* in ein Namensraumpräfix und den „eigentlichen“ Typnamen.

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.Drawing.Drawing2D

Klassen und andere Datentypen zur Unterstützung der zweidimensionalen Grafikausgabe.

Einen ersten Eindruck vom Leistungsvermögen der FCL vermittelt die folgende Auswahl ihrer Namensräume. Diese Auflistung ist für Programmierereinsteiger 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 mathematische Berechnungen oder Konvertierungen. 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.Drawing	... enthält Klassen für die Grafikausgabe mit GDI+ (erweitertes Windows Graphics Device Interface).
System.IO	... enthält Klassen für die Ein-/Ausgabebehandlung im Datenstrom-Paradigma.
System.Net	... enthält Klassen für die Netzwerk-Programmierung.
System.Reflection	... ermöglicht es u.a., zur Laufzeit Informationen über Klassen abzufragen oder neue Methoden zu erzeugen. Dabei werden die Metadaten in den .NET – Assemblies genutzt.
System.Security	... enthält Klassen, die sich z.B. mit Berechtigungen und Verschlüsselungs-Techniken beschäftigen.
System.Threading	... unterstützt parallele Ausführungsfäden.
System.Web	... unterstützt die Entwicklung von Internet-Anwendungen (inkl. ASP.NET).
System.Windows.Forms	... 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.

Microsoft stellt kostenlos eine umfangreiche FCL-Dokumentation und mit dem *Document Explorer* zudem ein komfortables Werkzeug zur Nutzung der Informationsfülle zur Verfügung (siehe Abschnitt 2).

1.2.7 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.
- **Betriebssystemunabhängigkeit**
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.
- **Schutz vor schädlicher und fehlerhafter Software**
Die CLR verwaltet die fein granulierten CAS-Rechte (*Code Access Security*) und verhindert die Ausführung von fehlerhaftem Zwischencode

- **Sehr breites Anwendungsspektrum**

Es kann Software für praktisch jeden Einsatzzweck zur Verwendung auf einem Arbeitsplatzrechner, auf einem Webserver oder auf einem PDA (Personal Digital Assistant) entstehen.

Wir haben im Abschnitt 1.2 u.a. folgende Begriffe kennen gelernt:

- **MSIL**

.NET – Compiler übersetzen den Quellcode in die *Microsoft Intermediate Language*.

- **Assembly**

Beim Übersetzen von (im Allgemeinen mehreren Quellen) entsteht ein Assembly. Diese ist kleinste Einheit von .NET – Software bei der ...

- o Weitergabe,
- o Versionierung,
- o Zuweisung von Sicherheitsbeweisen (evidence)

Ein Assembly kann beliebig viele Klassen implementieren. Ist eine Startklasse (mit Methode **Main()**) vorhanden, handelt es sich um ein Programm (Namenserweiterung **.exe**), anderenfalls um eine Bibliothek (Namenserweiterung **.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 Sicherheit (CAS), Stabilität (Verifikation) und Speicherverwaltung (Garbage Collection).

- **Namensraum**

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 seine Aufträge 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.

Anschließend arbeiten wir mit einer zeitgemäßen integrierten Entwicklungsumgebung, wobei Sie zwischen den folgenden drei Produkten wählen können:

- **Microsoft Visual Studio 2008 Professional**
Auf den Pool-PCs werden wir die Professional-Version von Microsofts kommerzieller Entwicklungsumgebung einsetzen. Dieses Produkt hat seit vielen Jahren einen sehr großen Marktanteil bei der Softwareentwicklung für Windows und wird von Fremdfirmen sehr gut durch Plugins für diverse Zwecke unterstützt.
- **Microsoft Visual C# 2008 Express Edition**
Dieser kostenlose Visual Studio - Ableger ist vermutlich auf Ihrem Privat-PC eine sehr gute Wahl. Bei den für uns relevanten Aufgaben gibt es keine wesentlichen Bedienungsunterschiede zwischen der *Visual C# 2008 Express Edition* und dem *Visual Studio 2008 Professional*.
- **SharpDevelop 2.2 oder 3.0**
Dies ist eine gute Alternative für ältere Rechner.

Alle drei Produkte sind bei der Projektverwaltung kompatibel und bieten u.a. folgende Leistungen:

- Guter Editor (z.B. mit Codevervollständigung, Syntaxhervorhebung)
- Graphischer Fenster-Designer
- Verschiedene Assistenten, z.B. zur Datenbankankündigung

Mit Ausnahme der Entwicklungsumgebung SharpDevelop stammt die im Kurs verwendete Software von der Firma Microsoft und kann (abgesehen vom Visual Studio 2008 Professional) über folgende Web-Adresse kostenlos bezogen werden:¹

<http://www.microsoft.com/downloads/>

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!");
    }
}
```

¹ Auf die Angabe von spezifischen Adressen wird verzichtet, weil sie teilweise kurzlebig und außerdem wegen der zahlreichen Parameter recht unübersichtlich sind.

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 durchaus als „pseudo-objektorientiert“ 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 aufgerufen, 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 1.2.6). Diese für C# - Programme typische Vorgehensweise soll auch im Hallo-Beispiel vorgeführt werden, obwohl sie hier den Schreibaufwand sogar vergrößert.
- Nach dem Schlüsselwort **class** folgt der frei wählbare Klassenname¹. Hier ist wie bei allen Bezeichnungen zu beachten, dass C# streng zwischen Groß- und Kleinbuchstaben unterscheidet.
- Dem Kopf der Klassendefinition folgt der mit geschweiften Klammern eingerahmte Rumpf.
- Weil die `Hallo`-Klasse startfähig sein soll, muss sie eine Methode namens `Main()` besitzen. Diese wird beim Programmstart 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:²

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

² 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 2001) ist nicht nur die Methode `Main()`, sondern auch die Klasse als **public** definiert, z.B.:

o **static**

Mit diesem Modifikator wird **Main()** als **Klassenmethode** gekennzeichnet. Im Unterschied zu den *Instanzmethoden* 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).

o **void**

Im Beispiel erhält die Methode **Main()** den Typ **void**, weil sie keinen Rückgabewert liefert.

- 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 im Hallo-Beispiel) eben eine leere. Wir werden uns später ausführlich mit Methodenparametern beschäftigen.
- Dem Kopf einer Methodendefinition folgt der mit geschweiften Klammern eingerahmte Rumpf mit Variablendeklarationen und 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 Programmtteile 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

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

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

```
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 ausschließlich von der CLR 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.

2.1.2 Übersetzen in MSIL

Öffnen Sie ein Konsolenfenster, und wechseln Sie in das Verzeichnis mit dem neu erstellten Quellprogramm **Hallo.cs**. Lassen Sie das Programm vom C#-Compiler **csc.exe** aus dem .NET Framework übersetzen, z.B.:

```
C:\WINDOWS\Microsoft.NET\Framework\v3.5\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.:



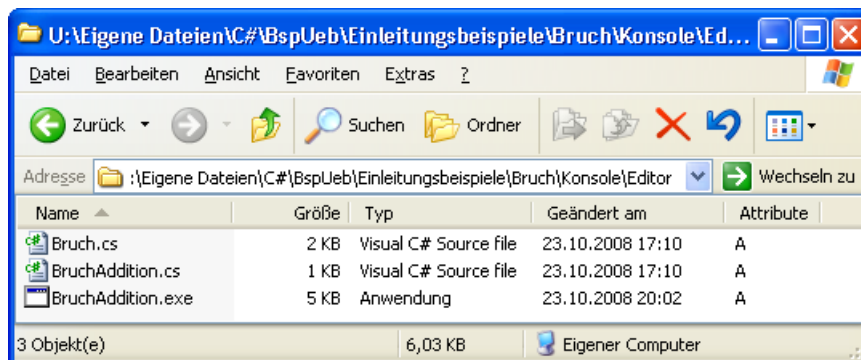
Sind *mehrere* Quellcodedateien in ein Assembly zu übersetzen, gibt man sie beim **csc**-Aufruf hintereinander an, z.B.

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

Wie Sie bereits wissen, 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 **Ausgabety**p 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 sorgen würde.
Beispiel:
`csc /target:winexe Bruch.cs BruchKürzen.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 (referenzierten) Klassen implementieren, z.B.:

```
csc /reference:microsoft.visualbasic.dll Prog.cs
```

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

Man kann den Compiler auch per **Response-Datei** mit Referenzen und sonstigen Befehlszeilenoptionen versorgen. Die Datei

%SystemRoot%\Microsoft.NET\Framework\Version\csc.rsp

ist Teil der Framework-Installation und wird automatisch vom Compiler ausgewertet. Beim .NET-Framework 3.5 enthält sie folgende Referenzen auf häufig benötigte Assemblies:

```
# 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.Vsa.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 Compiler mit der Befehlszeilenoption

```
/noconfig
```

und verhindern damit die Auswertung der voreingestellten Antwortdatei. Folglich wird (zeitsparend!) nur noch das zentrale Bibliotheks-Assembly **mscorlib.dll** automatisch durchsucht. Zur Verwaltung der in einem Projekt zusätzlich benötigten Referenzen bieten die Entwicklungsumgebungen bequeme Bedienelemente.

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 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).

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:

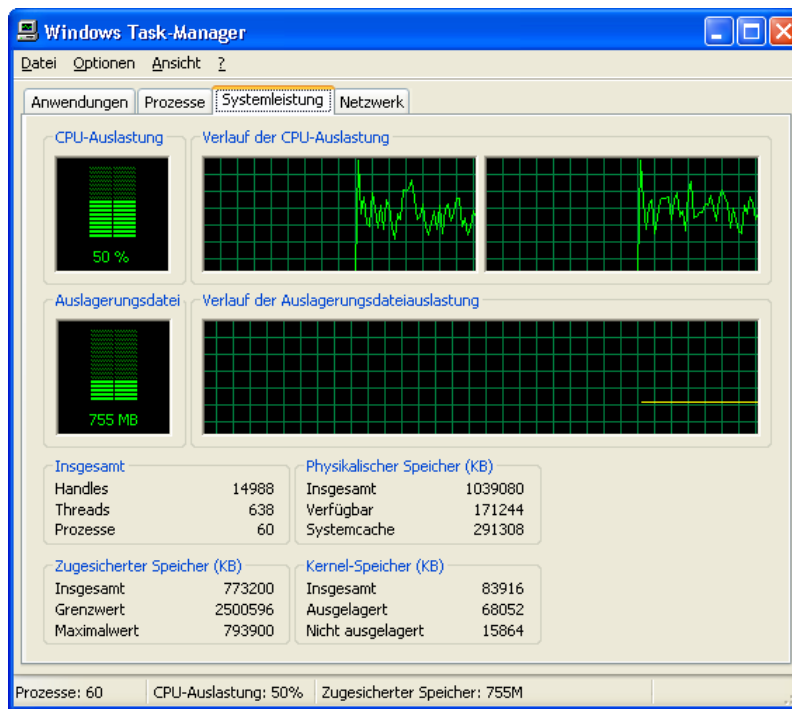
```
Hallo.cs(4,9): error CS0103: Der Name console ist im aktuellen
Kontext nicht
    vorhanden.
```

Der Compiler hat die fehlerhafte Stelle sehr gut lokalisiert: Datei **Hallo.cs**, Zeile 4, Spalte 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 aber ein unerwünschtes Verhalten: Weil die Methode `Kuerze()` in eine Endlosschleife gerät, „hängt“ das Programm und verbraucht dabei reichlich Rechenzeit, wie der Windows-Taskmanager (auf einem Rechner mit Single-Core - Hyper-Threading - CPU) belegt:



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

```

C:\WINDOWS\system32\cmd.exe
U:\Eigene Dateien\C#\BspUeb\Einleitungsbeispiele\Bruch\Konsole\Editor>BruchAddition
1. Bruch
Zaehler: 1
Nenner : 0
^C
U:\Eigene Dateien\C#\BspUeb\Einleitungsbeispiele\Bruch\Konsole\Editor>

```

2.2 Entwicklungsumgebungen und andere Programmierhilfen

Auf die Dauer ist das Hantieren mit Notepad und Kommandozeile beim Entwickeln von .NET - Software keine ernsthafte Option. Anschließend werden drei integrierte Entwicklungsumgebungen für die bequeme und rationelle Softwareentwicklung vorgestellt.

2.2.1 Microsoft Visual Studio 2008 Professional

2.2.1.1 Terminal-Server – Umgebung

Das Visual Studio 2008 Professional steht an der Universität Trier auf einem Terminalserver zur Verfügung, wobei die Verbindung zum Server auf den Pool-PCs folgendermaßen hergestellt wird:

**Start > Alle Programme > Programmentwicklung >
Microsoft Visual Studio > Visual Studio 2008 Professional**

Melden Sie sich beim Terminalserver mit Ihrem normalen URT-Konto an.

Im voreingestellten Vollbildmodus der Terminalserververbindung befindet man sich in einem neuen, abgeschlossenen Windows-Dialog, so dass sich z.B. die Tastenkombination:

Alt+Tabulator

zum Wechseln zwischen aktiven Programmen sich auf den *Terminalserver* bezieht.

Auf dem Terminalserver startet das Visual Studio automatisch. Beim Verlassen der Entwicklungsumgebung wird die Verbindung zum Terminalserver automatisch beendet.

Wer ohne Beendigung der Terminalserver Sitzung zwischendurch mit Programmen des lokalen Windows-Systems arbeiten möchte, kann folgendermaßen vorgehen:

- Am oberen Bildschirmrand befindet sich ein gelbes Bedienelement. Eventuell müssen Sie den Bildschirmrand mit der Maus berühren, damit das Bedienelement erscheint.
- Hier finden sich die gewohnten Symbole, um den Vollbildmodus zu verlassen oder die Terminal-Sitzung die Taskleiste (der lokalen Windows-Sitzung) zu schicken.

Der Terminalserver arbeitet mit der englischen Version des Betriebssystems Windows Server 2008, das zur selben Generation gehört wie das Desktop-System Windows Vista. Auf folgende Weise können Sie das **GUI-Design** auf dem Terminalserver dem Vista-Standard annähern (englische Anzeigesprache vorausgesetzt):

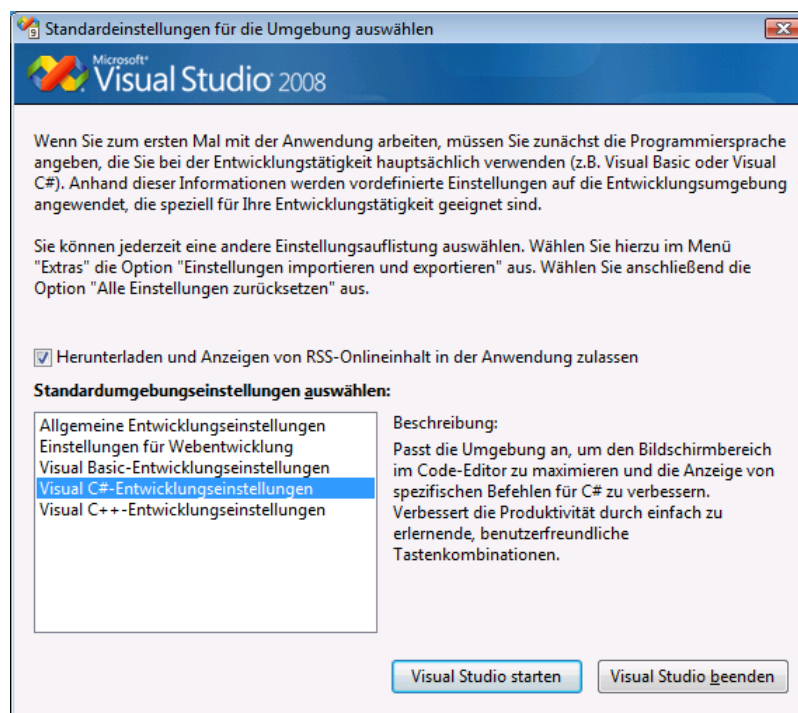
Control Panel > Personalize > Theme > Theme = Windows Vista

Gehen Sie folgendermaßen vor, um die voreingestellte **Anzeigesprache Englisch** durch *Deutsch* zu ersetzen:

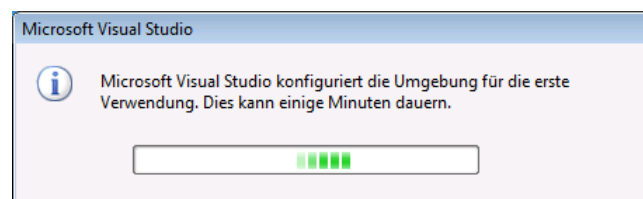
Control Panel > Clock, Language, and Region > Change Display language > Choose a display language = Deutsch

2.2.1.2 Konfiguration beim ersten Start

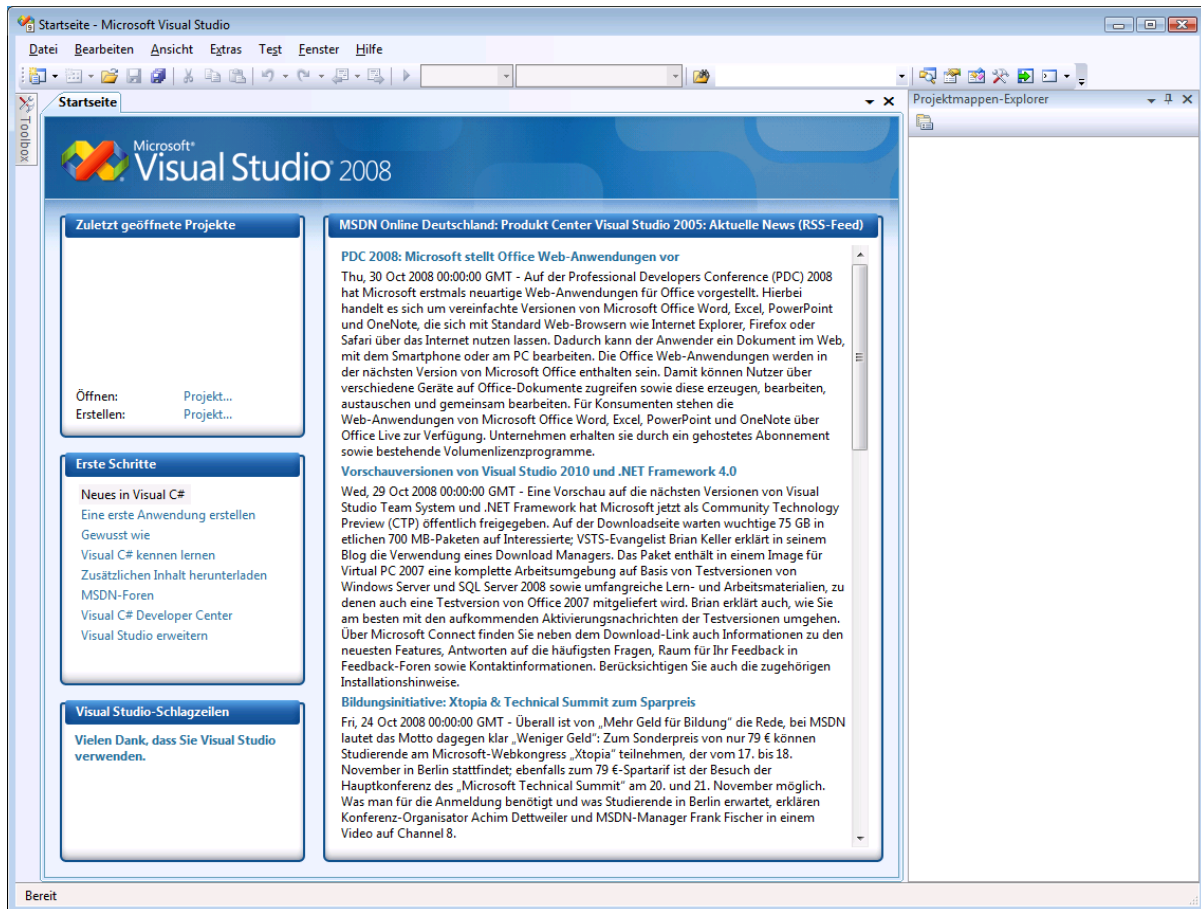
Beim ersten Start der Entwicklungsumgebung sind einige Einstellungen vorzunehmen. Wählen Sie zunächst in der folgenden Dialogbox die **Visual C# - Entwicklungseinstellungen**:



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



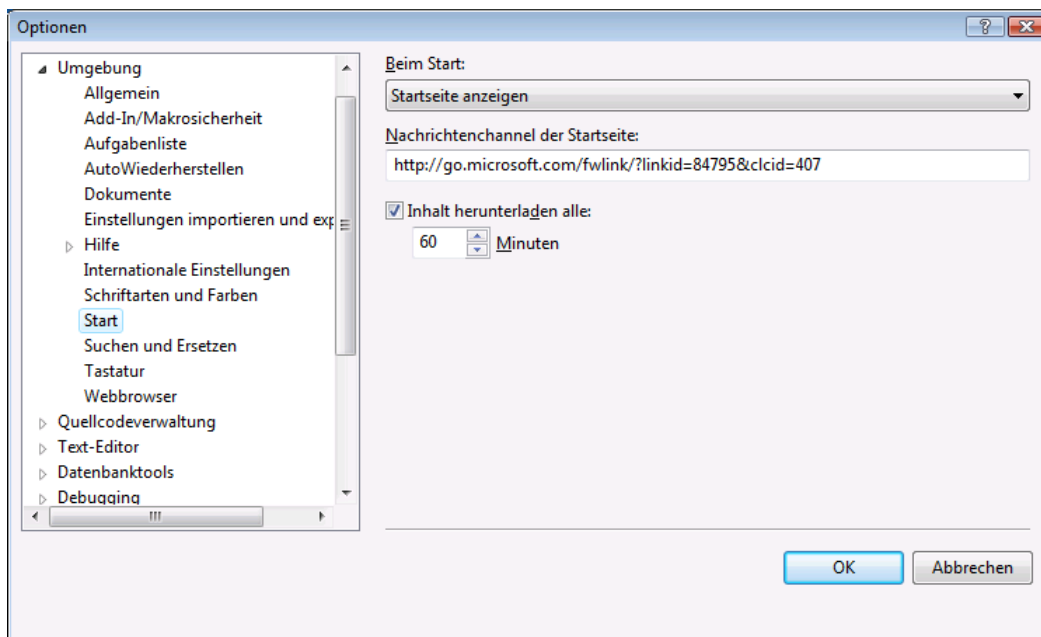
Auf der Startseite präsentiert das Visual Studio Neuigkeiten für Entwickler, falls Sie im obigen Konfigurationsdialog das **Herunterladen und Anzeigen von RSS-Onlineinhalten** zugelassen haben:



Das Startverhalten des Visual Studios lässt sich jederzeit nach

Extras > Optionen > Umgebung

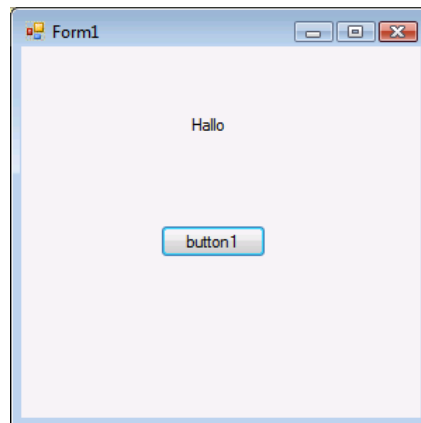
im folgenden Dialog beeinflussen:



2.2.1.3 Eine erste GUI-Anwendung

Dieser Abschnitt bietet 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.

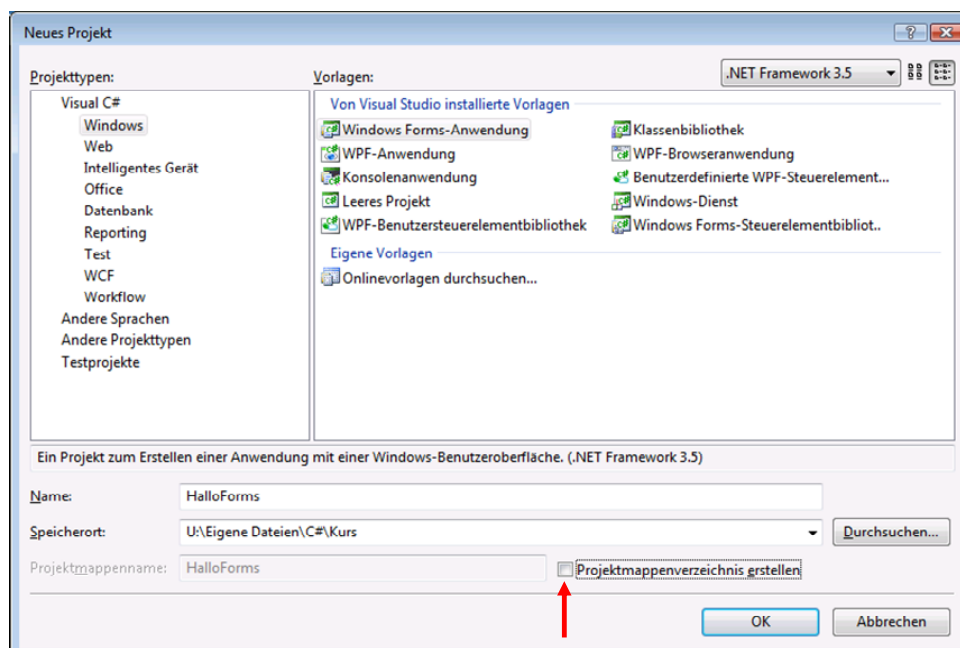
Nun machen wir uns an die Erstellung des folgenden Hallo-Programms mit GUI:



Wir fordern über

Datei > Neu > Projekt

den folgenden Dialog an:

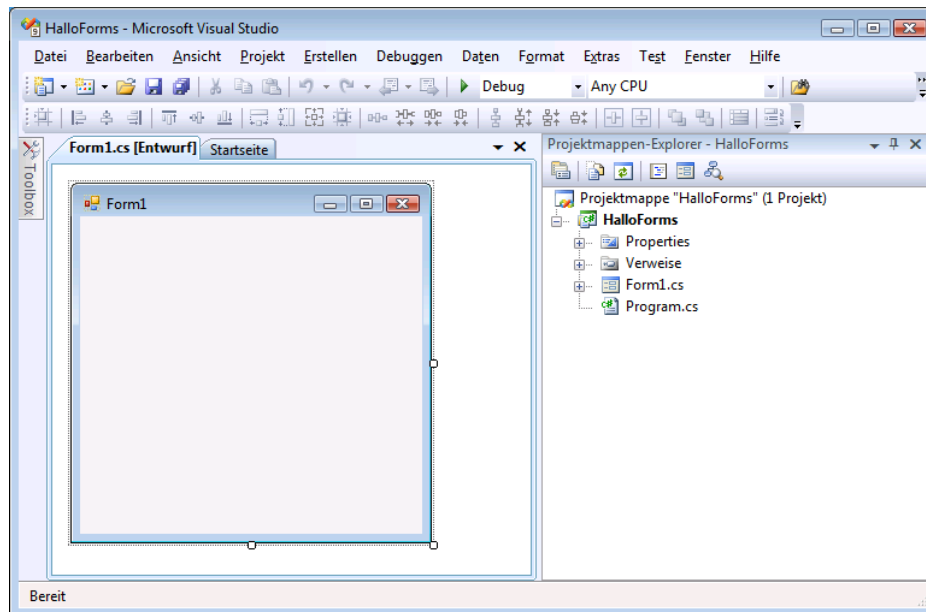


Hier akzeptieren wir die voreingestellte Projektvorlage **Windows-Anwendung**, wählen den Projektnamen **HalloForms**¹ sowie einen geeigneten **Speicherort** für den Projektordner. Allerdings wird kaum eine zusammengehörige *Familie* von Projekten entstehen, die in einem gemeinsamen Ordner versammelt werden sollten. Entfernen Sie daher nötigenfalls die Markierung beim Kontroll-

¹ Das Motiv für den Projektnamen wird Ihnen bald nach der Bekanntschaft mit den Klassen im Namensraum **System.Windows.Forms** plausibel erscheinen.

kästchen **Projektmappenverzeichnis erstellen**. Wir wählen damit eine flachere Projektdatenverwaltung, die für praktisch alle im Kurs entstehende Projekte angemessen ist.

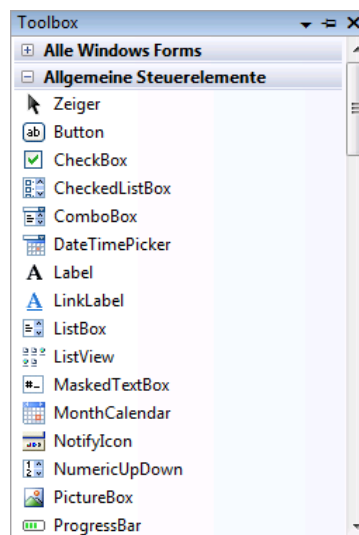
Nach einem Mausklick auf **OK** präsentiert das Visual Studio im **Projektmappen-Explorer** (am rechten Fensterrand) eine Baumansicht zur Projektverwaltung und im Formulardesigner einen Rohling für das Hauptfenster der Anwendung:



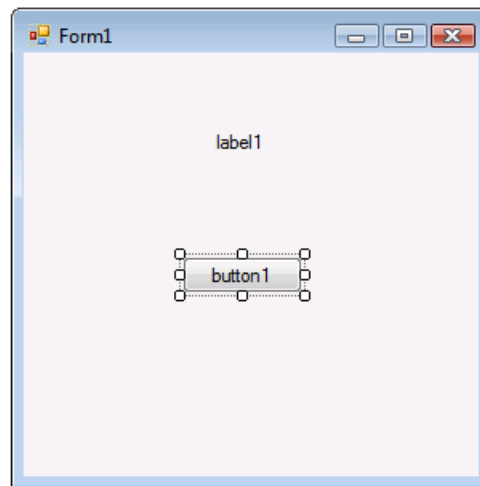
Öffnen Sie das **Toolbox**-Fenster mit dem Menübefehl

Ansicht > Toolbox

oder durch kurzes Verharren des Mauszeigers über der **Toolbox**-Schaltfläche am linken Fensterrand, und erweitern Sie die Liste der **Allgemeinen Steuerelemente**:

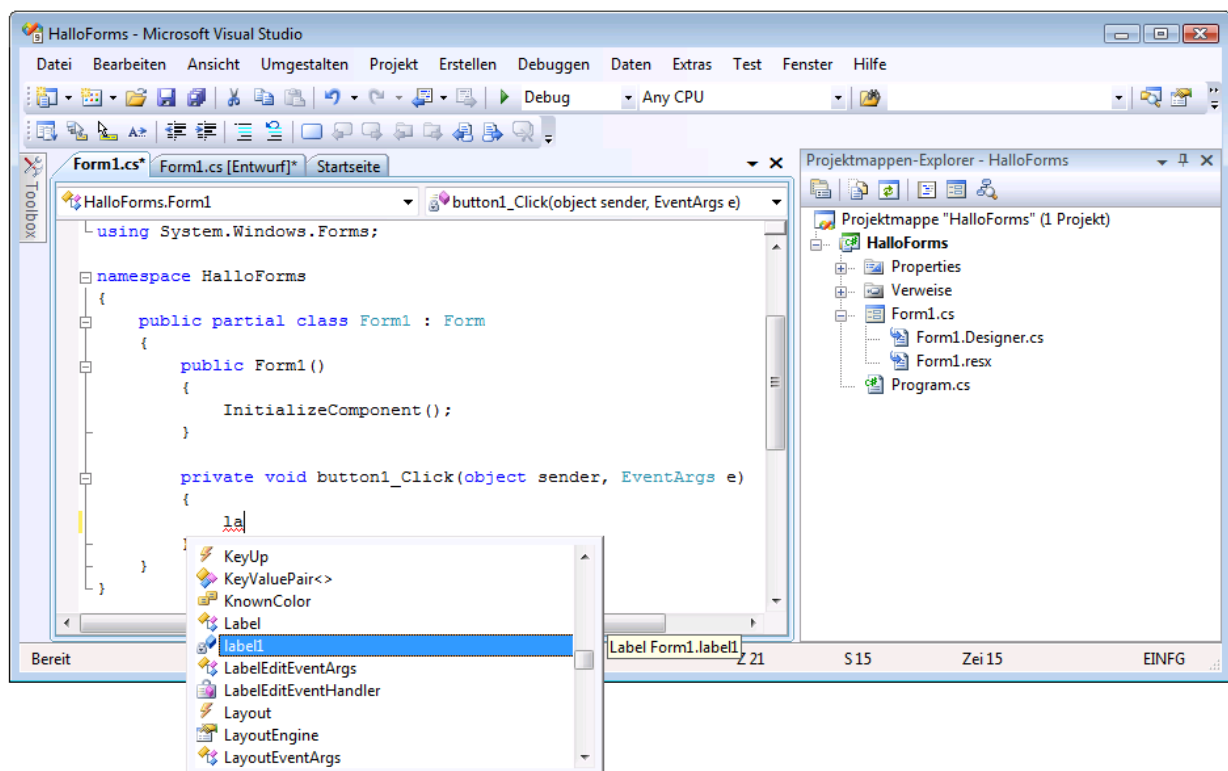


Ziehen Sie ein **Label**-Objekt (die Bezeichnung *Objekt* ist durchaus im Sinn von Abschnitt 1.1 gemeint) und ein **Button**-Objekt auf das Formular, indem Sie einen linken Mausklick auf das jeweilige Objekt setzen, die Maus dann mit gedrückter Taste zum Ziel bewegen und dort die Taste wieder loslassen. Das Ergebnis sollte ungefähr so aussehen:



Wenn 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.

Sobald Sie damit beginnen, im Methodenrumpf eine Nachricht an das **Label**-Objekt mit der Bitte um **Text**-Änderung zu verfassen, errahnt das Visual Studio Ihre Absicht und bietet mögliche Fortsetzungen Ihrer Anweisung an:



Akzeptieren Sie den Vorschlag 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 **Label**-Objekt soll aufgefordert werden, seine **Text**-Eigenschaft auf den Wert *Hallo* zu setzen. Wählen Sie diese Eigenschaft aus der Liste (z.B. per Doppelklick), und vervollständigen Sie die Anweisung:

```
private void button1_Click(object sender, EventArgs e)
{
    label1.Text = "Hallo";
}
```


Der Quellcode-Editor unserer Entwicklungsumgebung bietet außerdem ...

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

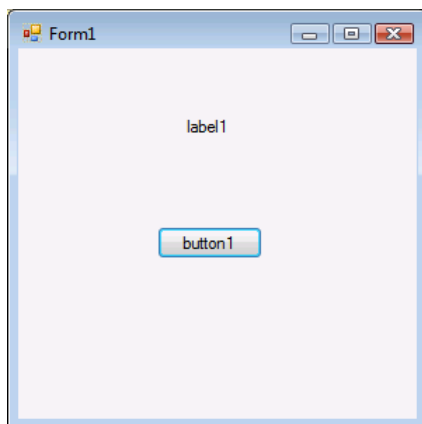
```
private void button1_Click(object sender, EventArgs e)
{
    label1.Text = "Hallo"
}
```

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

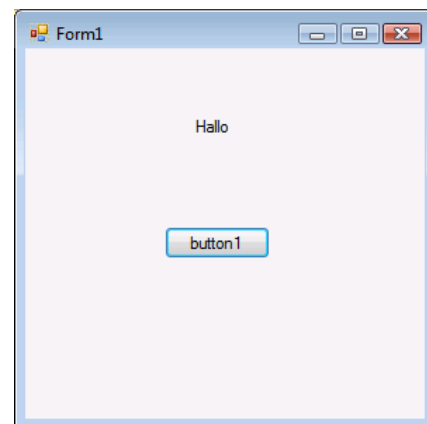
Debuggen > Debugging starten

veranlasst man das Übersetzen und die Ausführung der Anwendung:

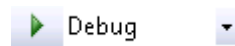
vor dem Mausklick auf den Schalter



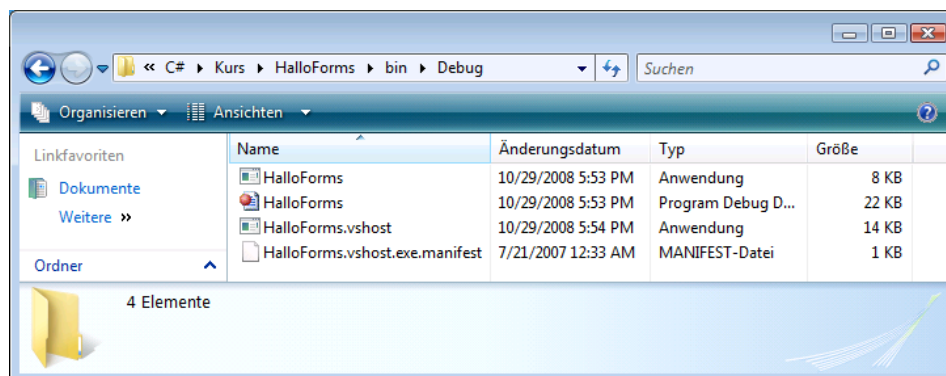
nach dem Mausklick auf den Schalter



Microsofts Entwicklungsumgebungen unterscheiden beim Übersetzen (bzw. Erstellen) die Konfigurationen **Debug** und **Release** (und ggf. noch weiteren benutzerdefinierten Konfigurationen), wobei die Auswahl per Standard-Symbolleiste erfolgt:



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 voreingestellten Debug-Konfiguration werden zusätzliche Ausgaben erzeugt, die eine Quellcode-basierte Fehlersuche erlauben. Weil auch in unserem Beispiel die Debug-Konfiguration eingestellt ist, landet das Assembly **HalloForms.exe** im Projekt-Unterverzeichnis ...**\bin\Debug**:



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

Erstellen > Projektmappe erstellen

Trotz der guten Erfahrungen mit der GUI-Programmierung werden wir uns die Grundbegriffe der Programmierung im Rahmen von möglichst einfachen Konsolenprojekten erarbeiten und auf die Assistenten im Visual Studio vorläufig verzichten. Im Manuskript werden ab jetzt die meisten Projekte mit der Visual C# 2008 Express Edition erstellt, deren Installation Thema des nächsten Abschnitts ist.

2.2.2 Microsoft Visual C# 2008 Express Edition

Die Firma Microsoft stellt mit der Visual C# 2008 Express Edition eine kostenlose Einstiegsversion seiner Entwicklungsumgebung Visual Studio 2008 zur Verfügung, die für unsere Kurszwecke sehr gut geeignet ist¹. 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 *mehrsprachig* sind, 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.

Weil im Programmieralltag unbedingt eine Dokumentation der .NET - Standardbibliothek (FCL) benötigt wird, sollte nach Möglichkeit zusätzlich zur *Visual C# 2008 Express Edition* auch die ebenfalls frei verfügbare *MSDN 2008 Express Edition* installiert werden. Ein weiteres kostenloses Angebot von Microsoft ist die *MSDN Library für Visual Studio 2008*. Dieses Paket dokumentiert neben dem .NET - Framework auch die Win32- und die COM-Programmierung und geht über den üblichen Informationsbedarf eines C# - Programmierers hinaus. Ist eine flotte Internetverbindung vorhanden, kann man bei geringer Komforteinschränkung auf eine lokale MSDN-Installation verzichten und die Online-Variante nutzen (<http://msdn.microsoft.com/de-de/library/default.aspx>). Die Abkürzung *MSDN* steht übrigens für das *Microsoft Developer Network*. Unter diesem Titel bietet die Firma Microsoft zahlreiche kostenlose oder auch kommerzielle Informationspakete für Software-Entwickler.

2.2.2.1 Installation

Voraussetzungen:

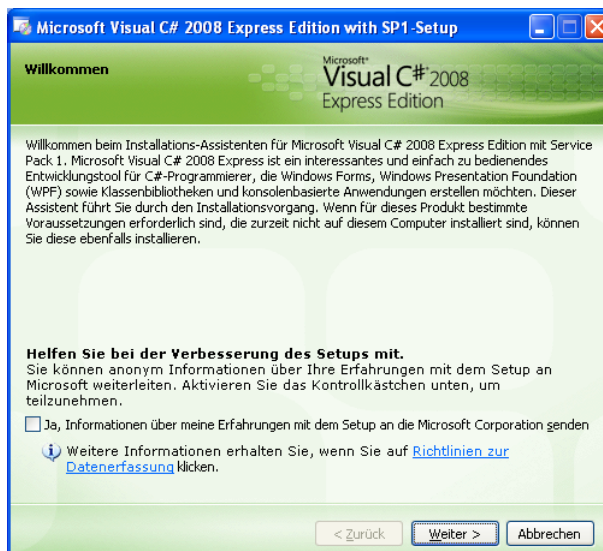
- Windows XP ab SP 2, Windows Vista, Windows Server 2003 SP 2, Windows Server 2008, Windows 7
- Prozessor mit 1 GHz Taktfrequenz (1,6 GHz empfohlen)
- 192 MB RAM (384 empfohlen)
- Die folgenden Angaben zum Platzbedarf auf der Festplatte gelten für einen PC mit Windows XP/Vista (32-Bit) und bereits installiertem .NET - Framework 2.0:

Produkt	Platzbedarf
Visual Studio 2008 Express Edition (ohne Option SQL-Server 2008 Express Edition)	ca. 800 MB
MSDN 2008 Express Edition	ca. 800 MB

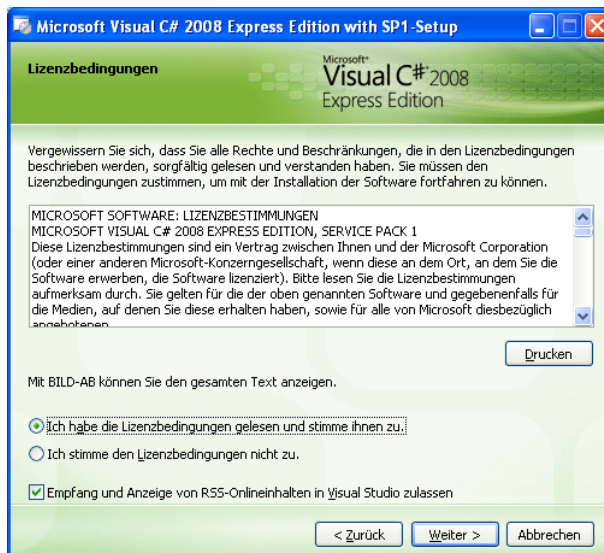
Die Installation der *Visual C# 2008 Express Edition* verläuft einfach und zuverlässig:

- Installationsprogramm **setup.exe** starten
- Entscheiden Sie selbst, ob Ihre Installationserfahrungen an Microsoft übermittelt werden sollen:

¹ Auf der Webseite werden <http://www.microsoft.com/germany/Express/download/>.



- Stimmen Sie der Lizenzvereinbarung zu:



Außerdem ist festzulegen, ob das Visual Studio Neuigkeiten anzeigen darf (z.B. auf der Startseite).

- Wählen Sie die gewünschten Zusatzoptionen:



Bei *Silverlight* handelt es sich um Microsofts Alternative zu Flash.

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). Mit der *Visual C# 2008 Express Edition* wird generell der für kleinere Datenbankaufgaben durchaus geeignete *SQL Server Compact 3.5* installiert. Wir werden bei der Behandlung der Datenbankprogrammierung beide SQL - Server verwenden.

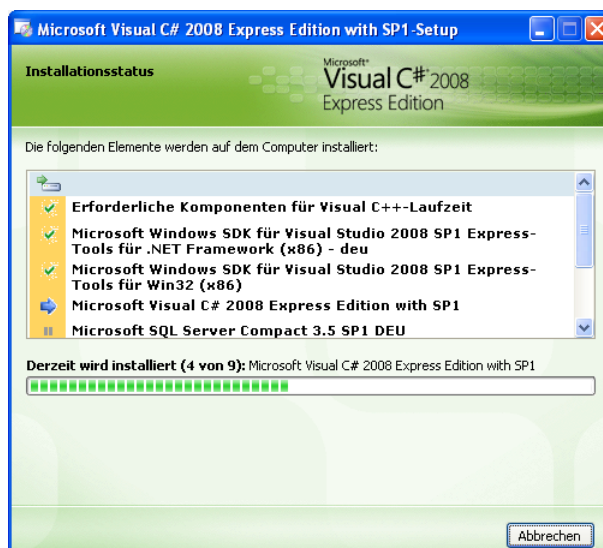
Wird die *SQL Server 2008 Express Edition* (durch Wahl in obigem Dialog) gemeinsam mit der *Visual C# 2008 Express Edition* installiert, fehlt anschließend die nützliche graphische Bedienoberfläche *SQL Server 2008 Management Studio Express*. Bei einer separaten Installation des (ebenfalls kostenlos bei Microsoft verfügbaren) Pakets *SQL Server 2008 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# 2008 Express Edition* zu installieren, sondern bei Bedarf später das Paket *SQL Server 2008 Express with Tools* zu installieren. Diese Empfehlung wird häufig, sobald das *SQL Server 2008 Management Studio Express* separat verfügbar ist. Mit der *Visual C# 2008 Express Edition* werden generell auch das .NET -Framework 3.5 sowie das Windows-SDK 6.0a installiert.

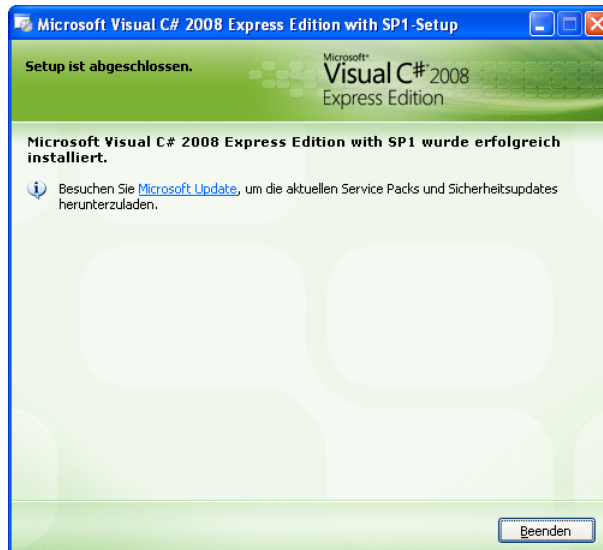
- Eventuell wollen Sie den Zielordner ändern:



- Nach einem Mausklick auf **Installieren** ist etwas Geduld gefragt:

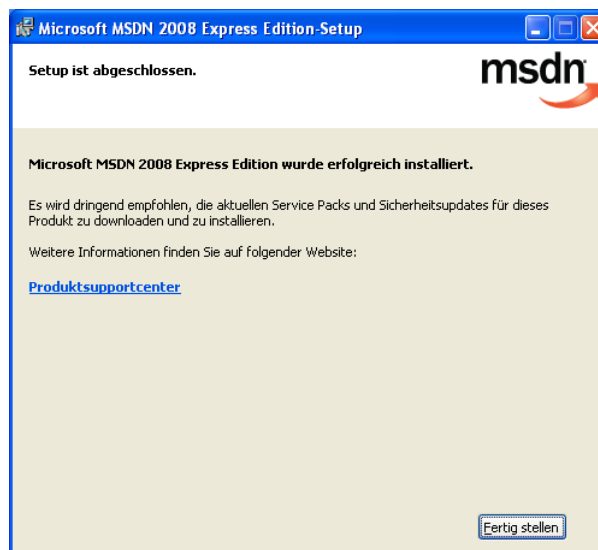


- Fertig:



Anschließend sollten Sie noch die *MSDN 2008 Express Edition* installieren:

- Starten Sie das Installationsprogramm **msdnixp.exe** und quittieren Sie den **Willkommen-**Dialog mit **Weiter**.
- Stimmen Sie der Lizenzvereinbarung zu.
- Nach einiger Zeit sollte die Erfolgsmeldung erscheinen:



2.2.2.2 Ein erstes Konsolen-Projekt

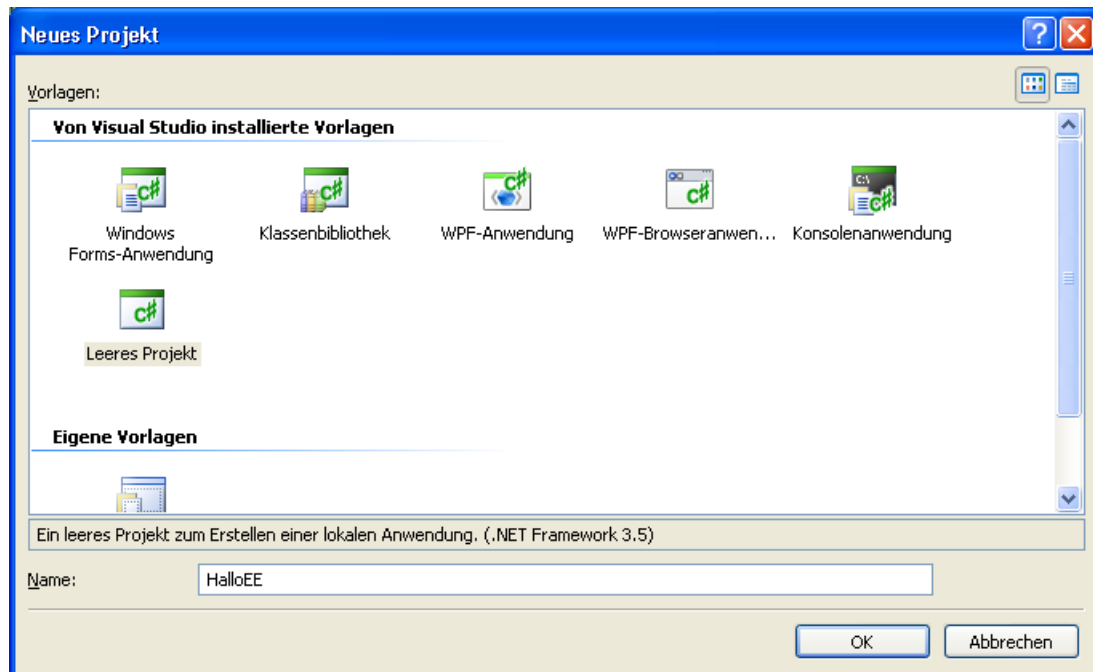
Beim ersten Start benötigt Visual C# einige Zeit zum Aufbau seiner Arbeitsumgebung



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

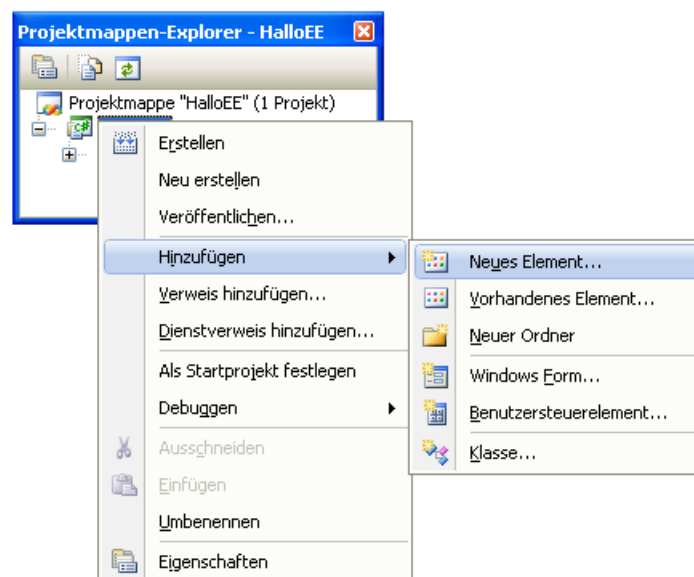
Datei > Neu > Projekt

den Dialog für neue Projekte:

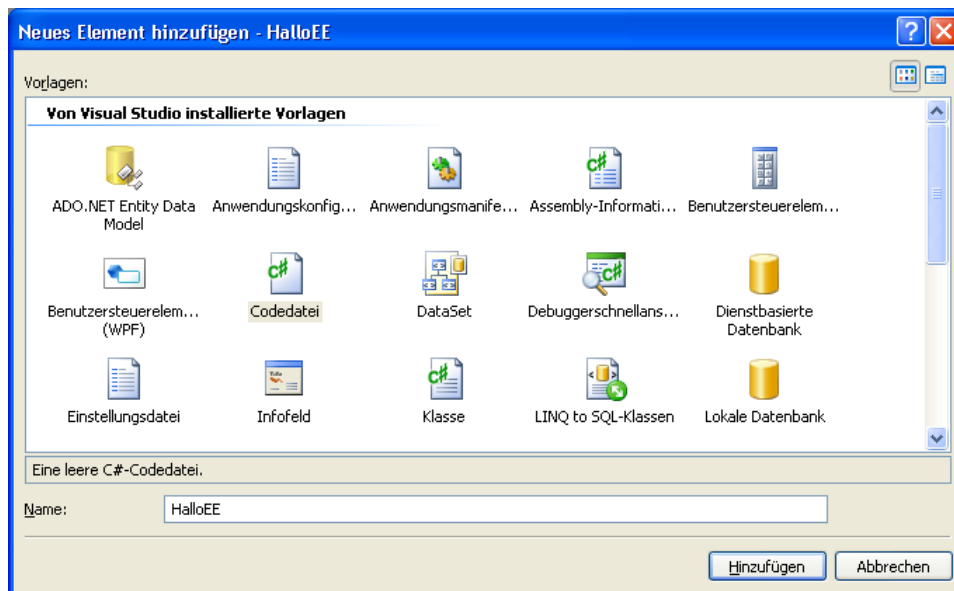


Wählen Sie die Vorlage **Leeres Projekt** und einen Projektnamen.

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:

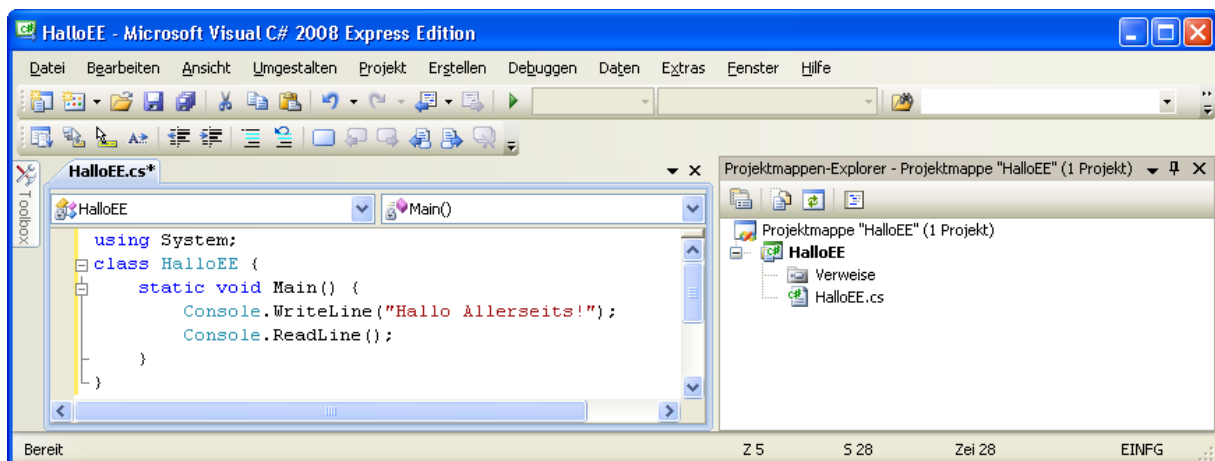


Entscheiden Sie sich für eine **Codedatei** mit geeignetem Namen:



Nach dem **Hinzufügen** präsentiert die Entwicklungsumgebung ein Projekt ohne eine einzige Zeile Assistenten-Quellcode.

Wir übernehmen den Quellcode vom Hallo-Beispielprogramm aus Abschnitt 2.1.1 und ergänzen am Ende der **Main()**-Methodendefinition noch einen **ReadLine()**-Methodenaufruf:

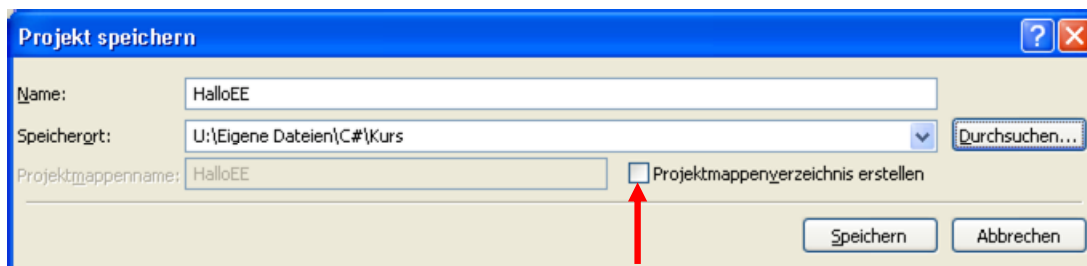


Er wartet auf die **Enter**-Taste und verhindert so, dass die im Rahmen der Entwicklungsumgebung mit dem Schalter  oder der Funktionstaste **F5** gestartete Konsolenanwendung nach ihrer Bildschirmausgabe sofort verschwindet. Außerdem passen wir den Namen der Startklasse an den Namen der Quellcodedatei an.

Fordern Sie über den Schalter  oder den Menübefehl

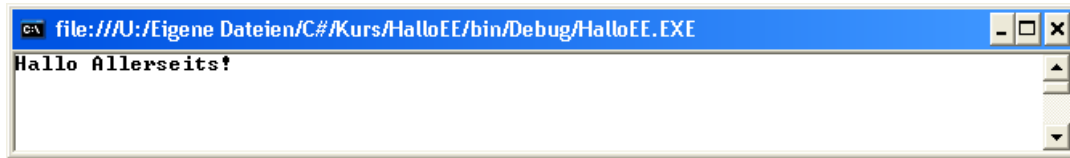
Datei > Alle Speichern

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 Projektdateverwaltung, die für praktisch alle im Kurs entstehende Projekte angemessen ist.

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

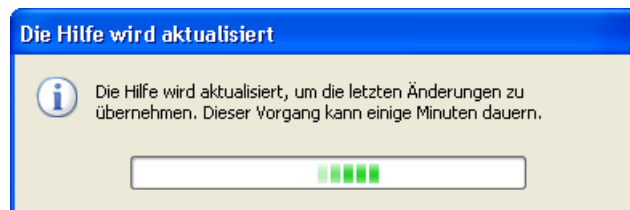


Sobald Sie die **Enter**-Taste drücken, kehrt der **ReadLine()**-Methodenaufruf zurück, sodass die Anwendung endet und das Konsolenfenster verschwindet.

2.2.2.3 Document Explorer

Zur Anzeige der FCL-Dokumentation und sonstiger MSDN-Informationen verwenden wir bevorzugt den zusammen mit dem Visual Studio Professional oder mit Visual C# installierten *Document Explorer*. Er wird über das Hilfemenü der Entwicklungsumgebungen oder auch (beim Visual Studio Professional) direkt über die Gruppe **Microsoft Developer Network** im Programm-Menü gestartet und bietet neben einer detaillierten Referenz zur .NET - Standardbibliothek auch zahlreiche Einführungstexte (Tutorials) und Beispiele.

Beim ersten Start benötigt der Document Explorer einige Zeit zum Aufbau seiner Arbeitsumgebung



Auf dem **Inhalts**-Registerblatt der Navigationszone im linken Teil des Fensters wählt man die Referenzinformationen zur .NET – Standardbibliothek über:

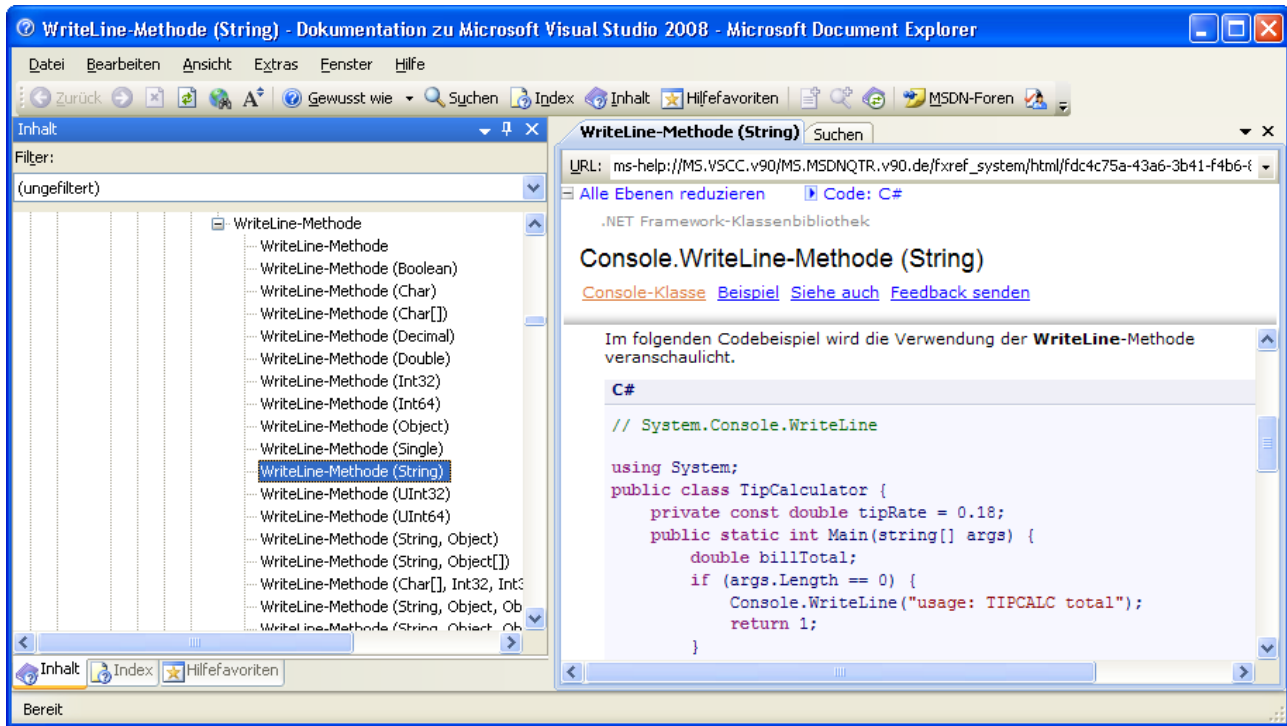
Microsoft MSDN Express Library 2008 > Dokumentation zu .NET Framework SDK > .NET Framework-Klassenbibliothek

Wenn Sie über die Baumansicht einen erwarteten Inhalt (z.B. die FCL-Dokumentation) nicht finden können, ist vermutlich der **Filter** zu eng eingestellt (z.B. auf **Visual C# Express Edition**). Wählen Sie einen alternativen Filter oder die Option (**ungefiltert**).

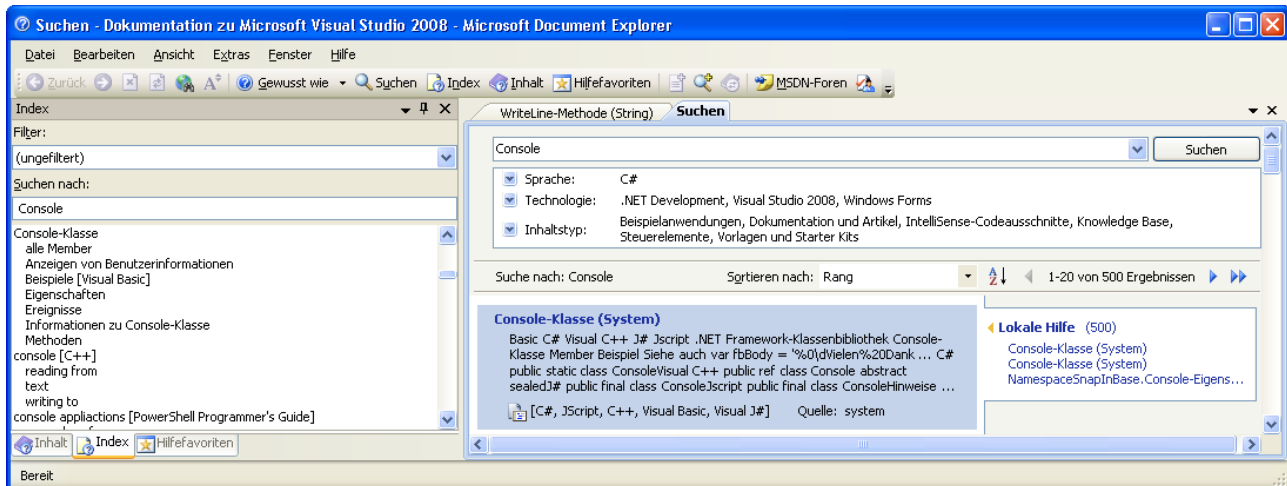
Wer sich z.B. über die in unseren Beispielen oft benutzte Methode **WriteLine()** der Klasse **Console** aus dem Namensraum **System** informieren möchte, bewegt sich ab dem Ausgangspunkt **.NET Framework-Klassenbibliothek** weiter mit den Stationen:

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:



Zur Suche nach speziellen Informationen eignen sich das **Index**-Registerblatt der Navigationszone und die Suchfunktion, z.B.:



Wer über eine Suchanfrage und/oder eine Sequenz von Mausklicks auf Verknüpfungen eine interessante Seite in das Inhaltsfenster des Document Explorers befördert hat und deren Standort im Inhaltsverzeichnis erfahren möchte, kann Inhalts- und Navigationsfenster mit dem Schalter  synchronisieren.

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

2.2.2.4 Referenzen und Ausgabtyp 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.:

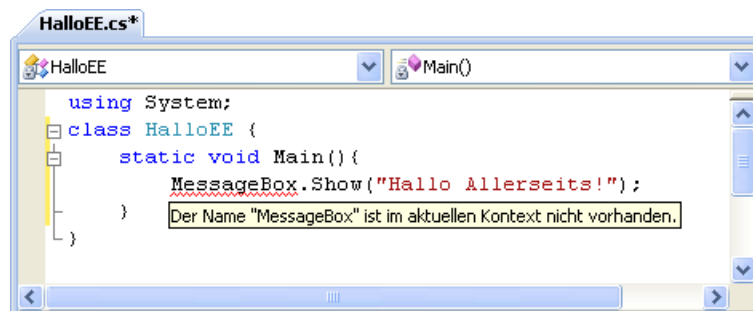
- Ausgabtyp des resultierenden Assemblies
- Referenzen auf Assemblies, die der Compiler nach Klassen durchsuchen soll

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

Um dies üben zu können, erstellen wir nun mit der Visual C# 2008 Express Edition ein Hallo-Projekt gemäß Abschnitt 2.2.2.2, ersetzen aber die Textausgabe durch eine Windows-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!");
```

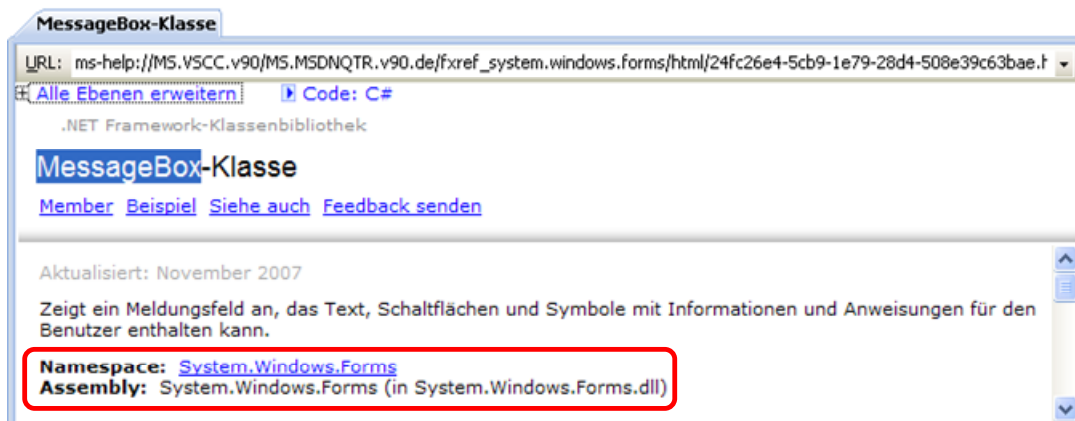
Ohne weitere Ergänzungen des Quellcodes kann die Übersetzung allerdings nicht gelingen, wie die Entwicklungsumgebung vorausblickend mit genauer Fehlerdiagnose mitteilt:



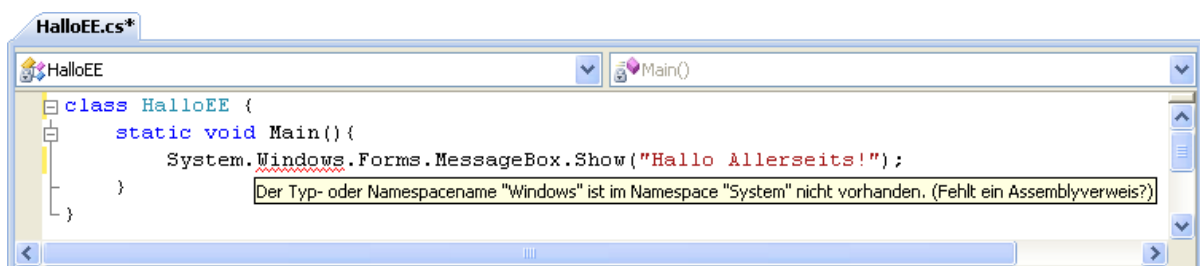
Mit Hilfe der FCL-Dokumentation, die als wesentlicher Bestandteil der *MSDN 2008 Express Edition* installiert wird, ermittelt man leicht, welcher Namensraum anzugeben ist (als Präfix zum Klassennamen oder in einer **using**-Direktive), um dem C# - Compiler die Klasse **MessageBox** bekannt zu machen. Von Visual C# aus erreicht man auf dem folgenden Weg

Hilfe > Suchen=MessageBox > MessageBox-Klasse (System.Windows.Forms)

diese Ausgabe:



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



Visual C# gibt aber mit der Frage nach einem eventuell fehlenden Assembly-Verweis einen guten Hinweis auf das Restproblem: Weil sich das die Klasse **System.Windows.Forms.dll** implementierende Assembly **System.Windows.Forms.dll** nicht in der gültigen Referenzliste des Compilers befindet, wird die Klasse nicht gefunden.

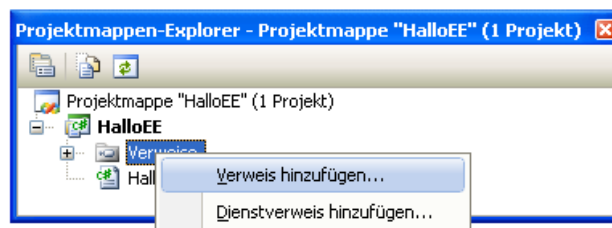
Das Assembly **System.Windows.Forms.dll** ist übrigens in der voreingestellten Antwortdatei **csc.rsp**

%SystemRoot%\Microsoft.NET\Framework\Version\csc.rsp

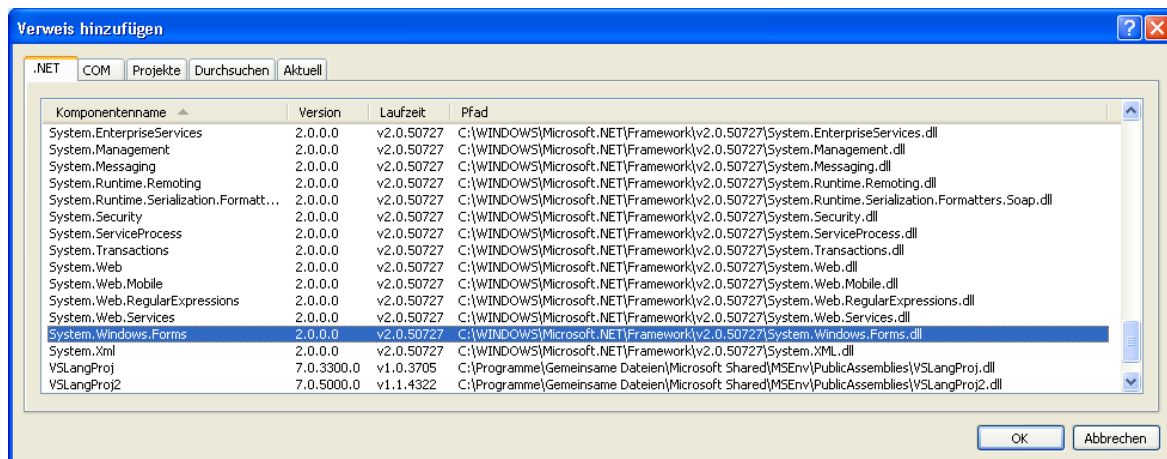
(vgl. Abschnitt 2.1.2) enthalten, so dass der obige Quellcode bei einem Compiler-Aufruf im Konsolenfenster ohne **reference** – Option erfolgreich übersetzt werden kann. Die integrierten Entwicklungsumgebungen rufen den Compiler jedoch mit der Option

/noconfig

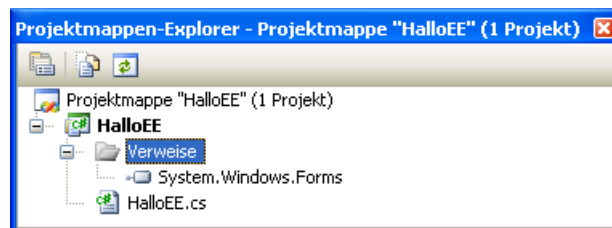
auf, welche die Auswertung der voreingestellten Antwortdatei verhindert, so dass nur noch das zentrale Bibliotheks-Assembly **mscorlib.dll** automatisch durchsucht wird. Folglich benötigt unser Projekt eine Referenz auf das Assembly **System.Windows.Forms.dll**¹. Wählen Sie dazu im **Projektmappen-Explorer** aus dem Kontextmenü zum Eintrag **Verweise** die Option **Verweis hinzufügen**:



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



Anschließend taucht der Verweis im Projektmappen-Explorer auf:



¹ Dies geschieht bei vielen Projektvorlagen der Entwicklungsumgebungen automatisch, nicht jedoch bei der Vorlage **Leeres Projekt**, die wir aus Gründen der Einfachheit und Transparenz derzeit bevorzugen.

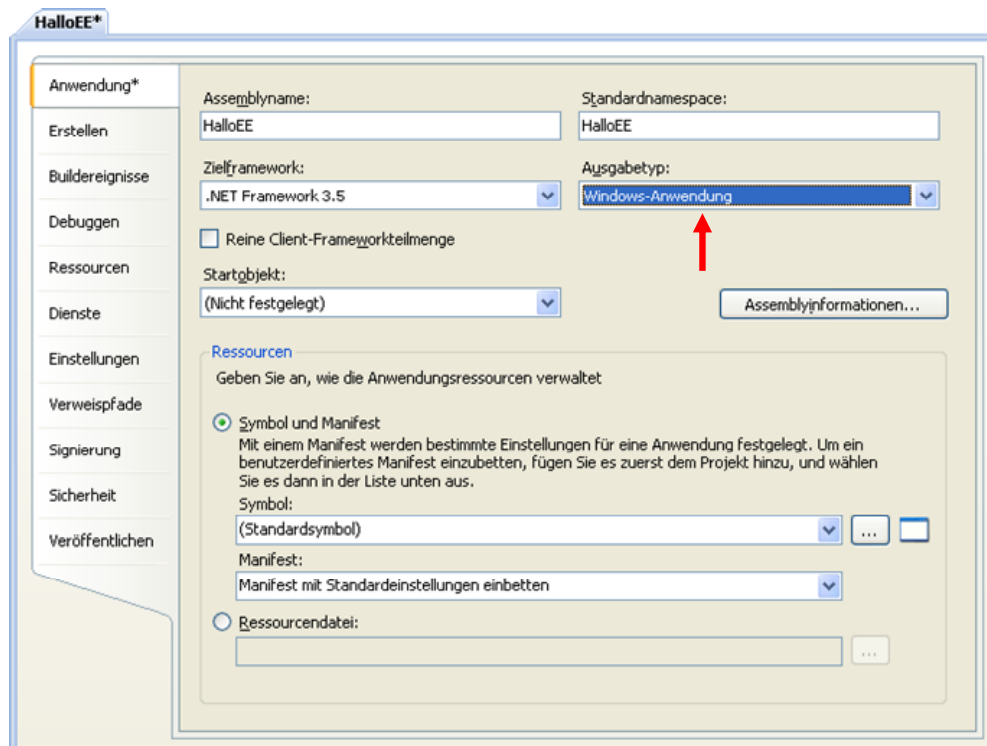
und die Reklamation im Quellcode-Editor verschwindet. Wenn Sie das Programm mit dem Schalter  oder der Funktionstaste **F5** übersetzen und starten, erscheint die gewünschte Messagebox:



Außerdem erscheint aber auch ein leeres Konsolenfenster. Um seinen Auftritt zu verhindern, ersetzt man per Compiler-Option den voreingestellten Ausgabotyp **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# 2008 Express Edition nach:

Projekt > Einstellungen > Anwendung

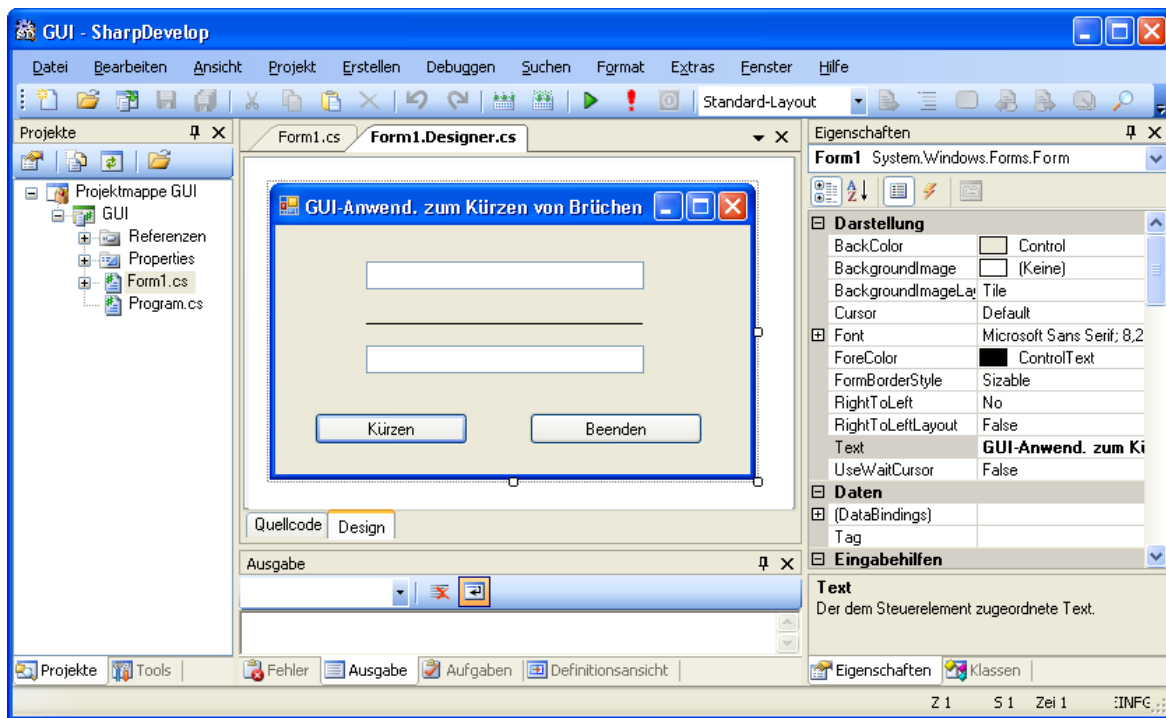
in folgendem Dialogfenster:



Bei einem Programm mit dem Ausgabotyp **winexe (Windows-Anwendung)** gehen alle Konsolenausgaben verloren, so dass man diesen Ausgabotyp mit Bedacht wählen sollte. Weitere Unterschiede zwischen den Ausgabotypen **exe** und **winexe** bestehen nicht.

2.2.3 SharpDevelop

Das schlanke, unproblematisch zu bedienende und recht flink agierende Programm ist selbst in C# entwickelt worden. Hier ist die Version 3.0 (Beta 2) zu sehen:



Über die folgenden Webseite:

<http://www.icsharpcode.net/OpenSource/SD/>

ist nicht nur ein bequemes Installationsprogramm zu beziehen, sondern auch der Quellcode des Open Source – Projekts.

Mit dem .NET Framework 2.0 und SharpDevelop 2.2 lässt sich eine besonders ressourcenschonende C# - Entwicklungsumgebung realisieren. Wer etwas mehr Festplattenspeicher zur Verfügung hat und SharpDevelop gegenüber den Microsoft-Entwicklungsumgebungen bevorzugt, sollte die Version 3.0 folgendermaßen installieren:

- .NET Framework 3.5 (siehe Abschnitt 1.2.2)
- Windows-SDK 6.1 for Windows Server 2008 and .NET Framework 3.5¹
Das Windows-SDK ist *nicht* obligatorisch für den Einsatz von SharpDevelop 3.0, allerdings sehr nützlich, weil es die komplette FCL-Dokumentation (in englischer Sprache) und diverse Hilfsprogramme und enthält (z.B. das schon in Abschnitt 1.1.2 vorgestellte Programm **ILDasm.exe** zur Analyse von Assemblies).

Systemvoraussetzungen für das Windows-SDK 6.1:

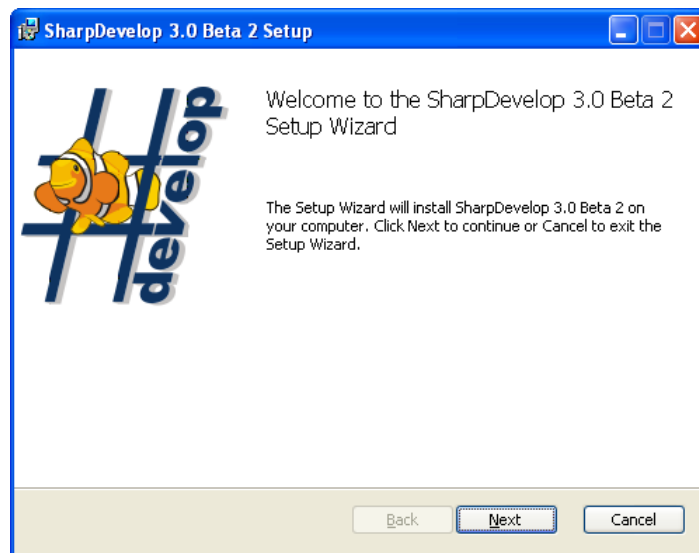
- Windows XP ab SP 2, Windows Vista, Windows Server 2003 SP 2, Windows Server 2008
- .NET Framework 3.5
- ca. 2,5 GB (!) Festplattenspeicher

Das MSDN-Online-Angebot ist möglicherweise eine platzsparende Alternative bei der Beschaffung von FCL-Referenzinformationen (siehe unten).

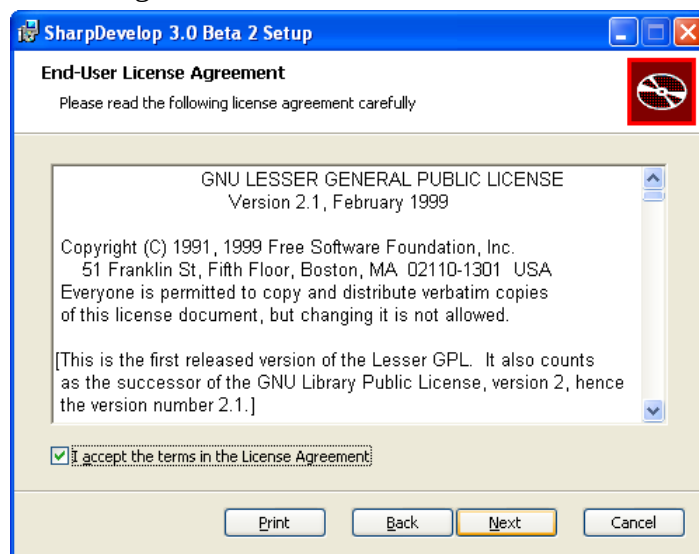
Bei der Installation einer Microsoft-Entwicklungsumgebung landet übrigens das Windows SDK (in der etwas älteren Version 6.0a) generell auf der Festplatte.

- SharpDevelop 3.0
Das Setup-Programm zur Version 3.0 präsentiert nach der Begrüßung

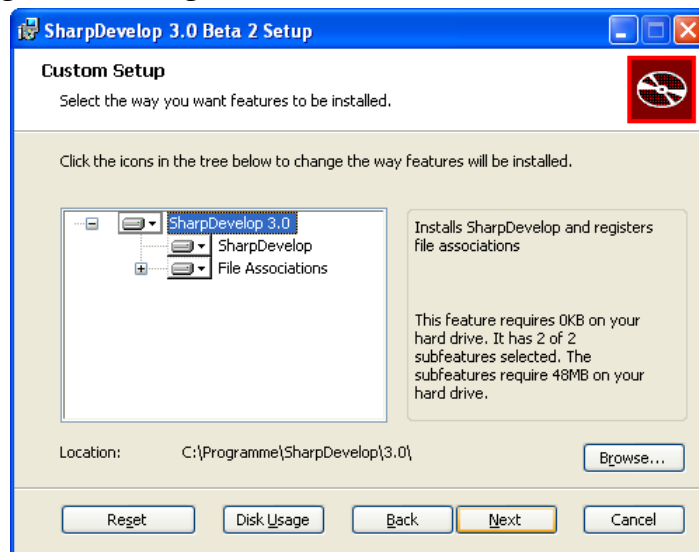
¹ Bezugsquelle: <http://www.microsoft.com/downloads/details.aspx?FamilyId=F26B1AA4-741A-433A-9BE5-FA919850BDBF&displaylang=en>



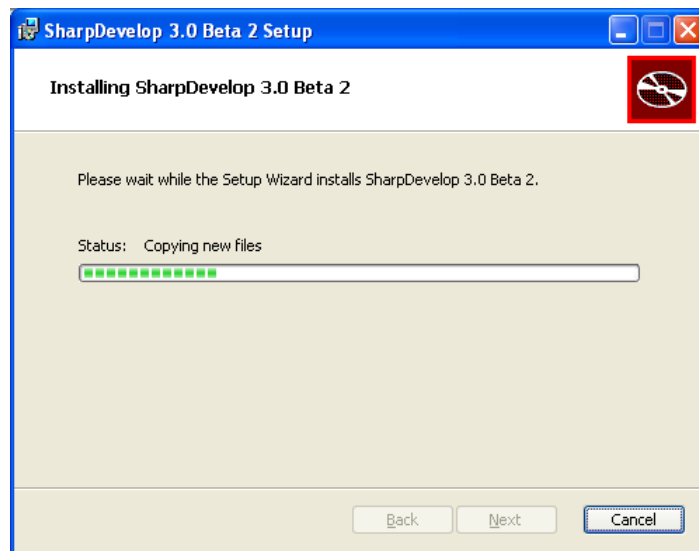
und der Lizenzvereinbarung



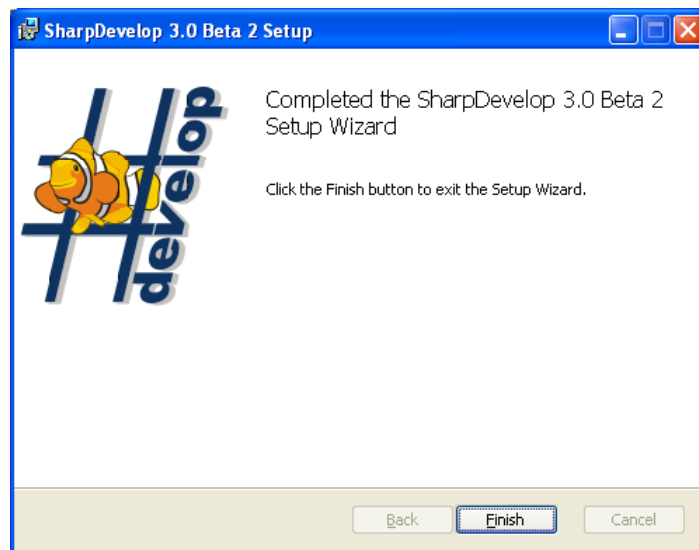
folgenden Konfigurationsdialog:



Die Installation



geht flott über die Bühne:

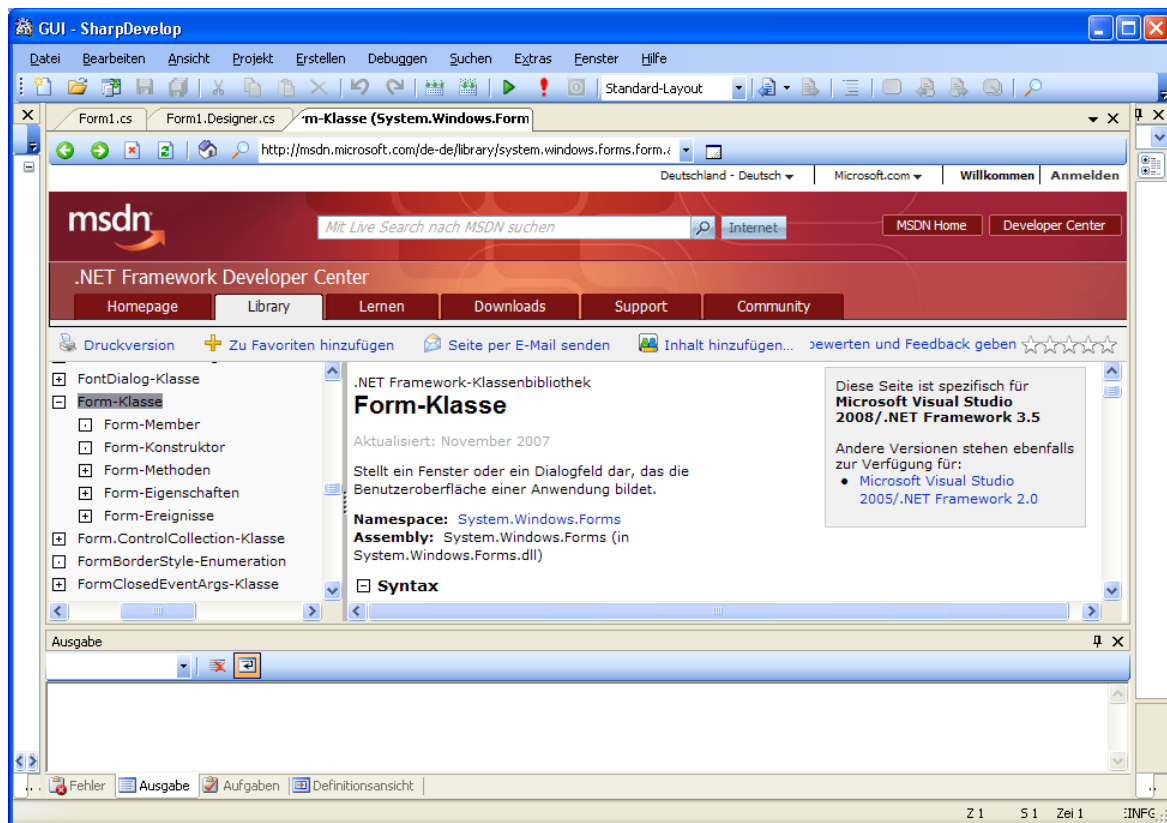


und begnügt sich mit ca. 50 MB Festplattenspeicher.

Leider lässt sich die *MSDN 2008 Express Edition* (siehe Abschnitt 2.2.2.1) nur installieren, wenn Visual C# oder ein anderes Produkt der Express - Serie bereits vorhanden ist. Sie ist damit keine Alternative, wenn die Windows-SDK - Installation wegen des großen Festplattenspeicherbedarfs vermieden werden soll. Mit einer flotten Internetverbindung kann man aber bei geringer Komforteinschränkung die Online-Variante der MSDN-Bibliothek nutzen (<http://msdn.microsoft.com/de-de/library/default.aspx>). Wählt man in SharpDevelop 3.0 den Menübefehl

Hilfe > Kontext-Hilfe

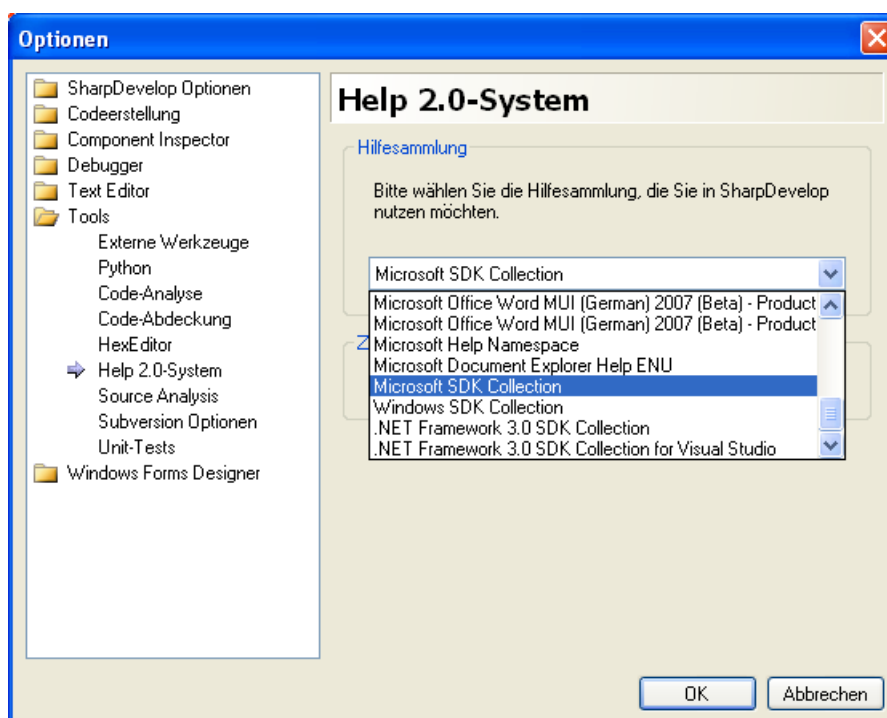
greift die Entwicklungsumgebung sogar spontan auf das MSDN-Online-Angebot zurück, z.B.:



Damit SharpDevelop 3.0 eine lokal installierte FCL-Dokumentation nutzt, müssen Sie eventuell nach

Extras > Optionen > Tools > Help-2.0-System

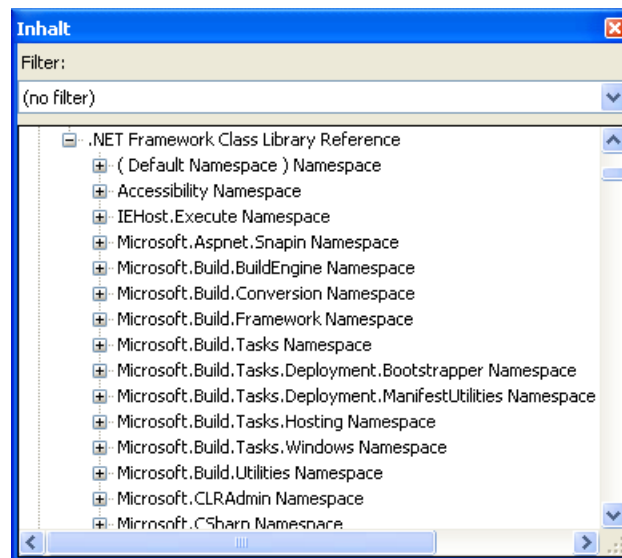
die gewünschte Hilfesammlung auswählen, z.B.:



Anschließend bietet das über

Hilfe > Inhalt

erreichbare **Inhalts**-Fenster die FCL-Dokumentation an:



Weil sich die Projektverwaltung von SharpDevelop 3.0 stark am Visual Studio 2008 bzw. an der Visual C# 2008 Express Edition orientiert, sollte ein Wechsel zwischen den drei Entwicklungsumgebungen keine Schwierigkeiten bereiten.

2.3 Übungsaufgaben zu Kapitel 2

1) Installieren Sie nach Möglichkeit auf Ihrem privaten PC entweder die Visual C# 2008 Express Edition (gemäß Abschnitt 2.2.2.1) oder SharpDevelop 3.0 (gemäß Abschnitt 2.2.3).

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 1.2.6).

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

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

3 Elementare Sprachelemente

In Abschnitt 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 einfachen, aus einem einzigen Assembly bestehenden, 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 Namensweiterung.
- 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 **exe**-Assembly enthält neben dem **MSIL-Code** aber auch Typ- und Assembly-**Metadaten**.

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

Diese Methode wird beim Programmstart vom Laufzeitsystem 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 Rückgabotyp **int** anzugeben. Fehlt eine solche Rückgabe, ist der Pseudorückgabotyp **void** anzugeben. Beim Bruchadditions-Beispiel in Abschnitt 1.1 ist die Klasse **BruchAddition** startfähig.

- 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:

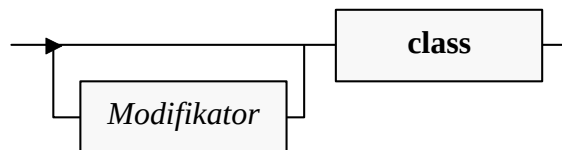
```
using System;
class Prog {
    static void Main() {
        // Platz zum Üben elementarer Sprachelemente
    }
}
```

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 Pfaddiagramm 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.

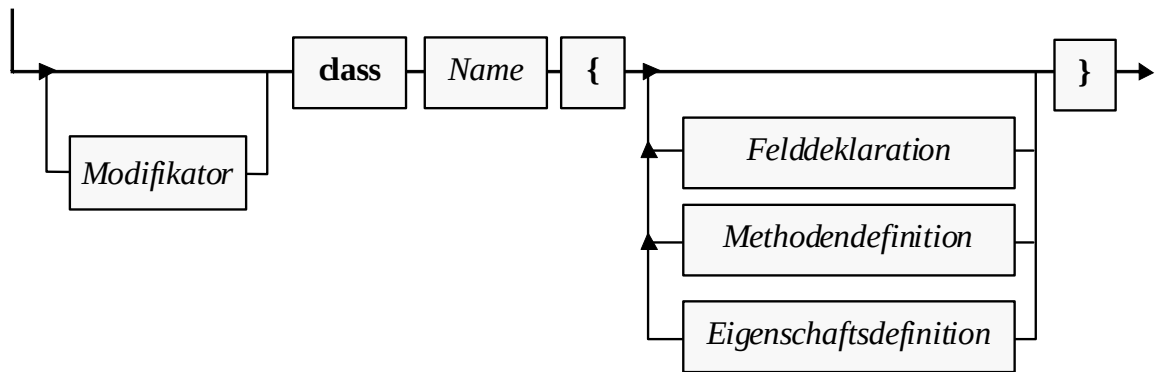
Als Beispiele betrachten wir die Syntaxdiagramme zur Definition von Klassen, Methoden- und Eigenschaften. Aus didaktischen Gründen zeigen die Diagramme nur solche Sprachbestandteile, die bisher in einem Beispiel verwendet oder im Text beschrieben wurden, so dass sie langfristig keinesfalls als Referenz 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).

In diesem Abschnitt 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

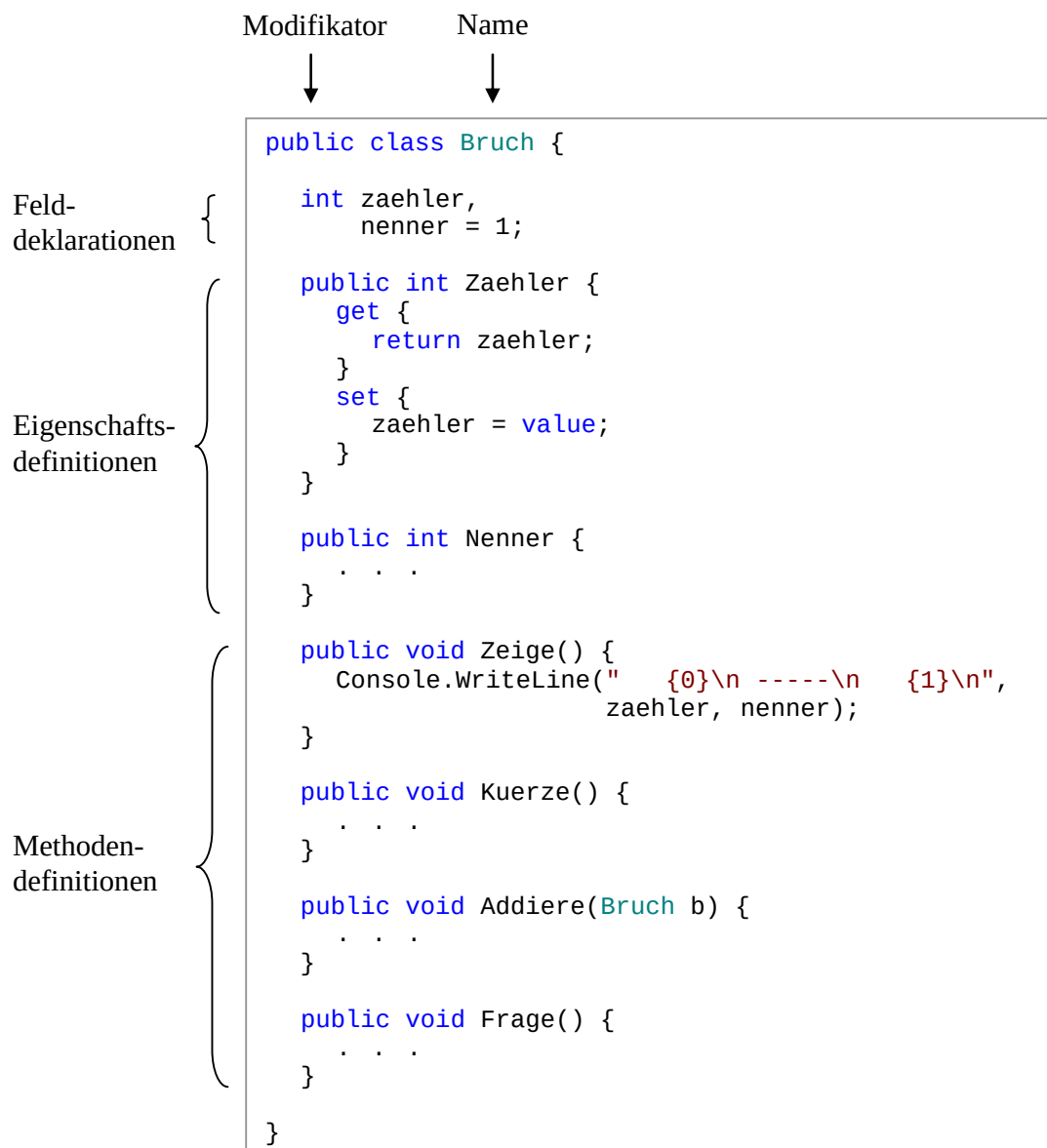
Wir arbeiten vorerst mit dem folgenden, leicht vereinfachten Klassenbegriff:

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.

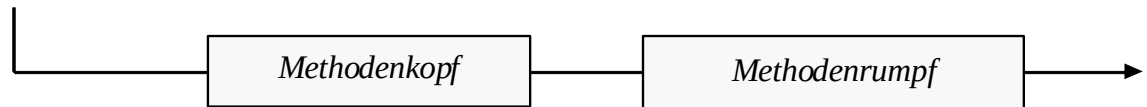
Als Beispiel betrachten wir die im Abschnitt 1.1 vorgestellte Klasse Bruch:



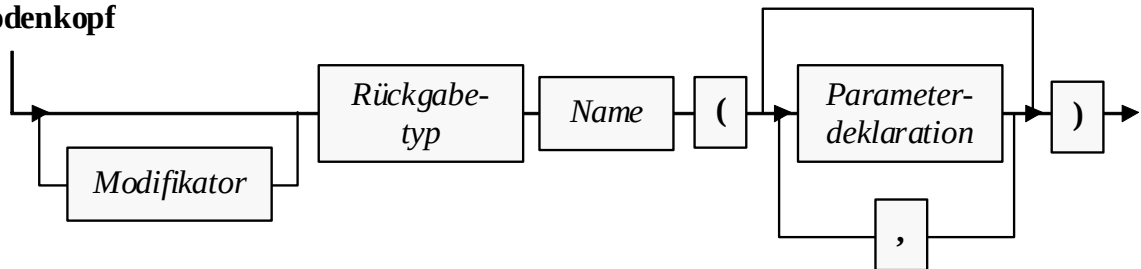
3.1.2.2 Methodendefinition

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

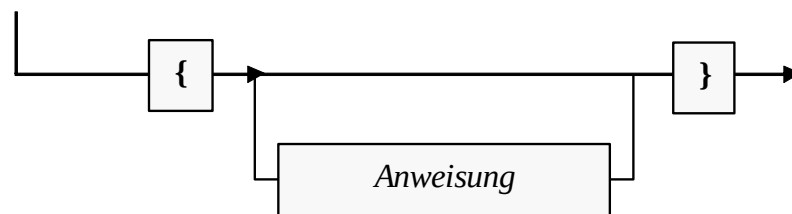
Methodendefinition



Methodenkopf

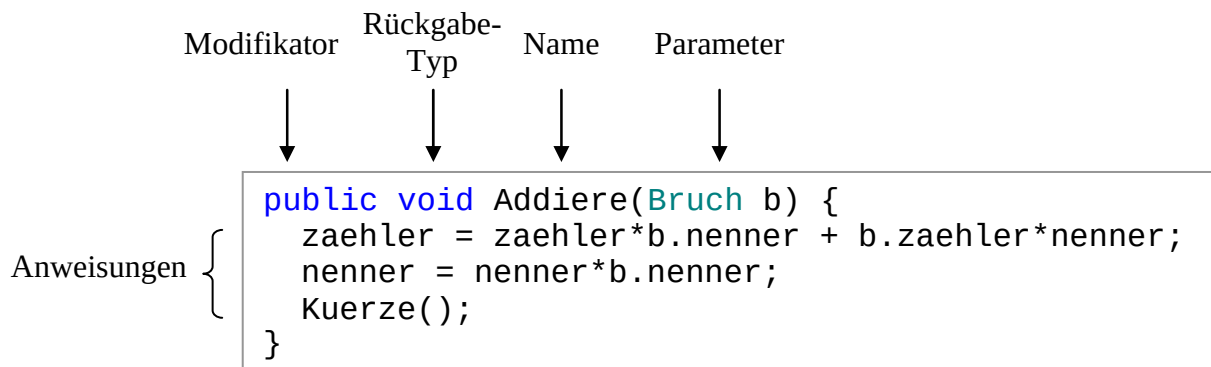


Methodenrumpf



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

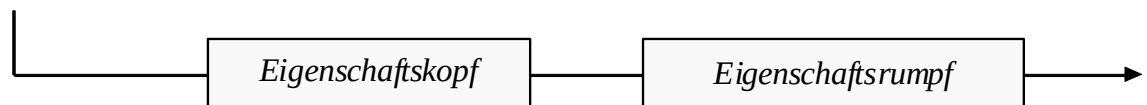
Als Beispiel betrachten wir die Definition der Bruch-Methode `Addiere()`:

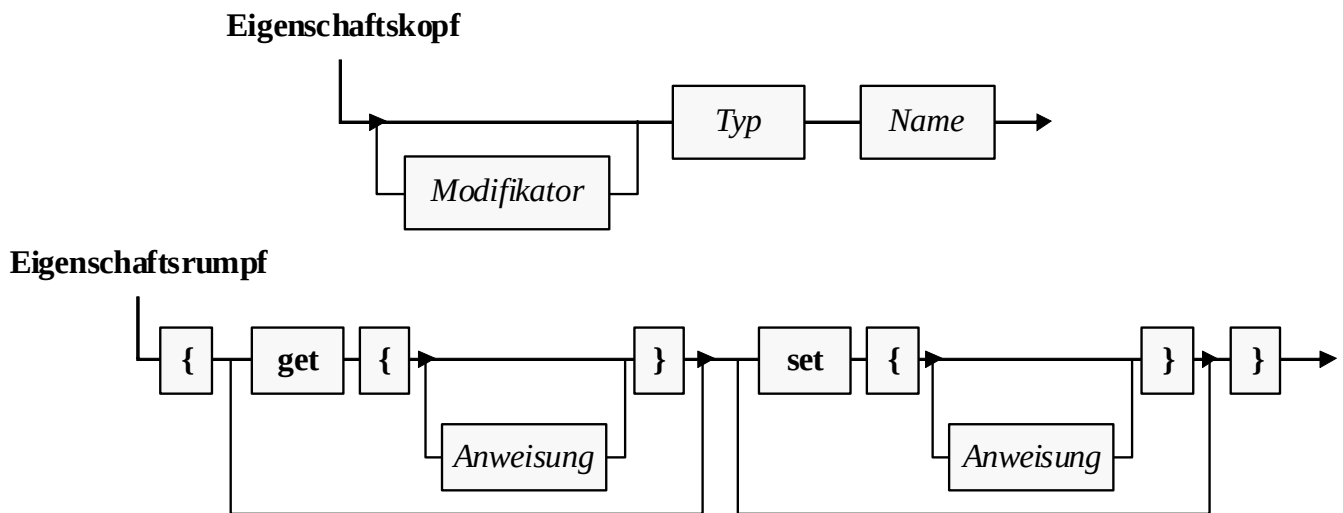


3.1.2.3 Eigenschaftsdefinition

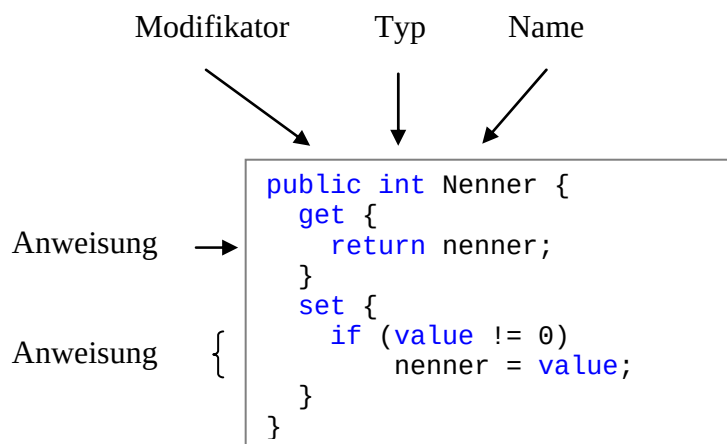
Auch beim Syntaxdiagramm für den Eigenschaftsbegriff gehen wir schrittweise vor:

Eigenschaftsdefinition





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 im Prinzip 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 sehr unerwünscht.
- Zeichen mit festgelegter Bedeutung wie z.B. ";", "(", "+", ">" sind *selbstbegrenzend*, 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# 2008 Express Edition, die via 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.:

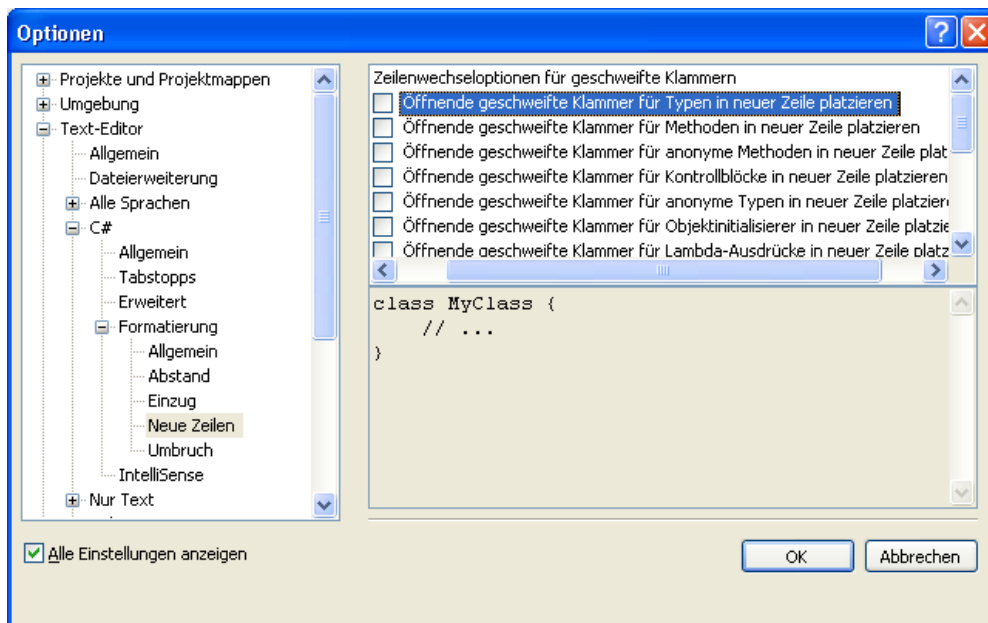
```
class Hallo {
    static void Main() {
        System.Console.WriteLine("Hallo");
    }
}
```

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

Die Visual C# 2008 Express Edition bevorzugt die rechte Variante, kann aber nach

Extras > Optionen

in der folgenden Dialogbox 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, einen Programmteil (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+E+C** veranlasst, werden doppelte Schrägstriche vor jede Zeile gesetzt. Wendet man den Menübefehl

Bearbeiten > Erweitert > Auskommentierung der Auswahl aufheben

bzw. die Tastenkombination **Strg+E+C** auf einen zuvor mit Doppelschrägstrichen auskommentierten Block an, werden die Kommentar-Schrägstriche 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. *NDoc* oder *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 der damit verbundenen Fehlerwahrscheinlichkeit nicht empfehlenswert.

3.1.5 Namen

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

- 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 Wörter** 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				

- 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 3.3.2), Parametern 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 großen Anfangsbuchstaben:

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

Alternativ kann man auch den Unterstrich zur Verbesserung der Lesbarkeit zusammengesetzter Namen verwenden, was der folgende Methodenname demonstriert, den Visual C# im HalloForms-Projekt erstellt hat (siehe Abschnitt 2.2.1.3):

```
private void button1_Click(object sender, EventArgs 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 einerzusammengesetzten 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!");
```

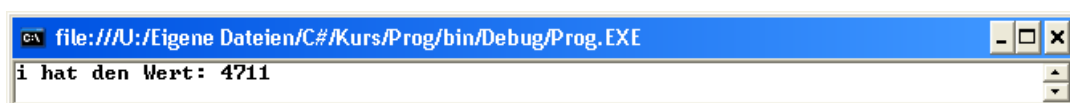
Übergebene Argumente anderen Typs werden vor der Ausgabe automatisch in eine Zeichenfolge konvertiert, z.B. der Wert einer ganzzahligen Variablen (siehe unten):

```
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**

Beispiel:

Quellcode-Fragment	Ausgabe
<pre>int i = 47, j = 11; Console.WriteLine("Werte:\t{0}\n\t{1}", i, j);</pre>	<pre>Werte: 47 11</pre>

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.

3.2.3 Übungsaufgaben zu Abschnitt 3.2

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

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

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 wie 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, l = 4;
```

mit zwei **WriteLine()**-Aufrufen diese Ausgabe produziert:

Rechtsbündig:

```
i = 4711
j =  471
k =   47
l =    4
```

Linksbündig:

```
4711 (i)
471  (j)
47   (k)
4    (l)
```

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 so genannte **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 = 4711;           //schreibender Zugriff auf ivar
        Console.WriteLine(ivar);  //lesender Zugriff auf ivar
    }
}
```

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.:

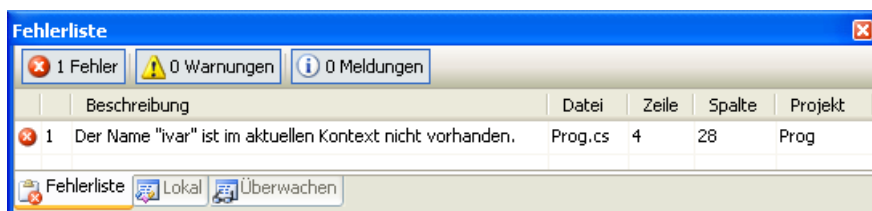
Prog.cs(4,28): error CS0103: Der Name ivar ist im aktuellen Kontext nicht vorhanden.

In der Visual C# 2008 Express Edition informiert schon der Quellcode-Editor über das Problem, z.B.:

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

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 Variablenamen 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.:

Quellcode	Ausgabe
<pre>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</pre>	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. Die Variable erhält auch gleich den **Initialisierungswert** 4711. Auf diese oder andere Weise müssen Sie jeder *lokalen*, d.h. innerhalb einer Methode deklarierten, Variablen einen Wert zuweisen, bevor Sie zum ersten Mal lesend darauf zugreifen (vgl. Abschnitt 3.3.5).

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 Speicheradressen praktisch keine Rolle. Wir werden jedoch zwei wichtige Speicherregionen unterscheiden (*Stack* und *Heap*), weil Performanzgründe die gezielte Auswahl einer bestimmten Region nahelegen können.

3.3.1 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 können:

- **Werttypen**

Die traditionellen Datentypen werden in C# als *Werttypen* (*Value Types*) bezeichnet. Variablen mit einem Werttyp spielen auch in C# eine unverzichtbare Rolle, obwohl sie „nur“ zur Verwaltung bzw. Nutzung ihres Inhalts dienen. 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:

```
int zaehler, nenner = 1;
```

- **Referenztypen**

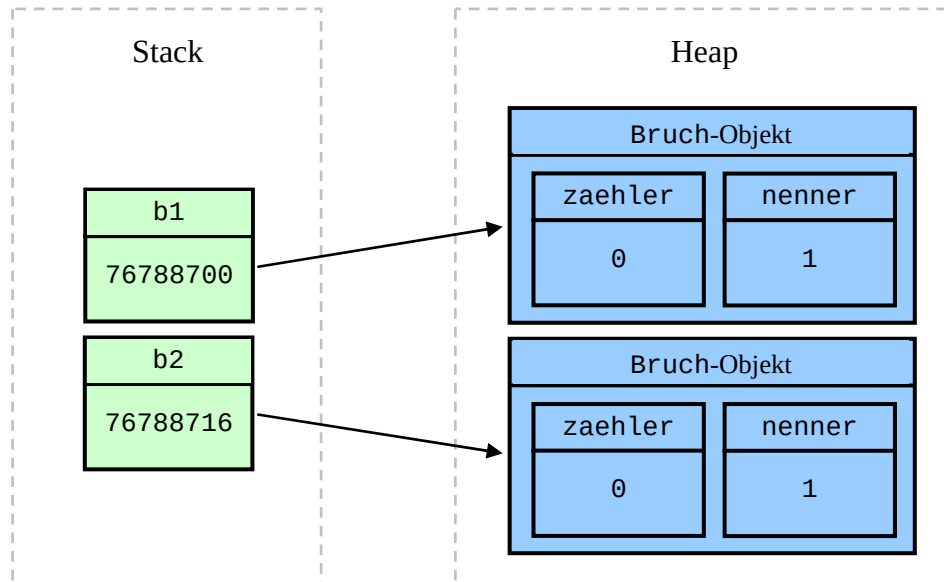
Eine Variable mit Referenztyp dient dazu, die **Speicheradresse eines Objekts** aus einer bestimmten Klasse aufnehmen. Sobald ein solches Objekt erzeugt und seine Speicheradresse der Referenzvariablen zugewiesen worden ist, kann das Objekt über die Referenzvariable angesprochen werden. Von den Variablen 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 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 Variablen (Felder) von beliebigem Typ. So enthält z.B. ein **Bruch**-Objekt die Variablen (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 ermöglichen. 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.

3.3.2 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 deklariert. Ihre Gültigkeit beschränkt sich auf die Methode bzw. auf einen Block innerhalb der Methode (siehe Abschnitt 3.3.6).

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 (synonym: jede *Instanz*) einer Klasse verfügt über einen vollständigen Satz der Instanzvariablen der Klasse. So besitzt z.B. jedes Objekt der Klasse `Bruch` einen `zaehler` und einen `nenner`.

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 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 wurden, 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.

3.3.3 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
<code>sbyte</code>	Diese Variablentypen speichern ganze Zahlen <i>mit</i> Vorzeichen. Beispiel: <code>int zaehler = -7</code>	-128 ... 127	8
<code>short</code>		-32768 ... 32767	16
<code>int</code>		-2147483648 ... 2147483647	32
<code>long</code>		-9223372036854775808 ... 9223372036854775807	64

Typ	Beschreibung	Werte	Bits
byte	Diese Variablentypen speichern ganze Zahlen <i>ohne</i> Vorzeichen. Beispiel: <code>byte alter = 31;</code>	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: <code>float p = 4.2526f;</code> float -Literele (s.u.) benötigen den Suffix f (od. F).	Minimum: $-3.4028235 \cdot 10^{38}$ Maximum: $3.4028235 \cdot 10^{38}$ Kleinster Betrag: $1.4 \cdot 10^{-45}$	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: <code>double p = 1.13445626535898;</code>	Minimum: $-1,7976931348623157 \cdot 10^{308}$ Maximum: $1,7976931348623157 \cdot 10^{308}$ Kleinster Betrag: $4,9 \cdot 10^{-324}$	64 1 für das Vorz., 11 für den Expon., 52 für die Mantisse
decimal	Variablen vom Typ decimal speichern alle Dezimalzahlen mit bis zu 28 Stellen exakt und eignen sich besonders für die Finanzmathematik , wo Rundungsfehler zu vermeiden sind. Beispiel: <code>decimal p = 2344.2554634m;</code> decimal -Literele (s.u.) benötigen den Suffix m (oder M).	Minimum: $-(2^{96}-1)$ Maximum: $2^{96}-1$ Kleinster Betrag: 10^{-28}	128 1 für das Vorz., 5 für den Expon., 96 für die Mantisse, restl. Bits ungenutzt Im Expon. sind nur die Werte 0 bis 28 erlaubt, die negativ interpret. werden.
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: <code>char zeichen = 'j';</code> char – Literale (s.u.) sind mit <i>einfachen</i> Anführungszeichen einzurahmen.	Unicode-Zeichen Tabellen mit allen Unicode-Zeichen sind z.B. auf der Webseite http://www.unicode.org/charts/ des Unicode-Konsortiums verfügbar.	16
bool	Variablen vom Typ bool speichern Wahrheitswerte. Beispiel: <code>bool cond = false;</code>	true, false	1

Als *Gleitkommazahl* (synonym: *Gleitpunkt-* oder *Fließkommazahl*, engl.: *floating point number*) bezeichnet man eine EDV-freundlich notierte rationale Zahl (Dezimalzahl). Während die Mantisse mit einer festen Anzahl von Ziffern für Genauigkeit sorgt, speichert man zusätzlich per Exponentialfaktor die Position des Dezimalkommas, sodass der Mantissenwert anwendungsgerecht für z.B. für Lichtjahre oder Nanometer stehen kann. Weil mit dem Speicheraufwand auch der Wertebereich begrenzt ist, bilden die Gleitkommazahlen natürlich nur eine endliche Teilmenge der rationalen Zahlen (im mathematischen Sinn). 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 rationalen Zahlen im Arbeitsspeicher eines Computers folgen für speziell interessierte Leser in Abschnitt 3.3.4.

3.3.4 Vertiefung: Darstellung rationaler Zahlen 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 Teilnehmer mit Vorkenntnissen, die sich eventuell (z.B. im Abschnitt über elementare Sprachelemente) etwas langweilen.

3.3.4.1 Binäre Gleitkommadarstellung

Bei den binären Gleitkommatypen **float** und **double** werden auch „relativ glatte“ Zahlen nicht unbedingt genau gespeichert, wie das folgende Programm zeigt:

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { double f130 = 1.3f; double f125 = 1.25f; Console.WriteLine("{0,20:f16}", f130); Console.WriteLine("{0,20:f16}", f125); float ff130 = 1.3f; Console.WriteLine("\n{0,20:f16}", ff130); Console.ReadLine(); } }</pre>	<pre>1,29999999523162800 1,25000000000000000 1,30000000000000000</pre>

Die Zahl 1,3 kann im **float**-Format (sieben signifikante Dezimalstellen) *nicht* exakt gespeichert werden. Um dies zu demonstrieren, wird ein **float**-Wert (erzungen 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 zugrunde liegende **IEEE-754** – Format für die Darstellung von rationalen Zahlen zu erklären. Es handelt sich um ein **binäres Gleitkommaformat**, wobei jede Zahl als Produkt aus drei getrennt zu speichernden Faktoren dargestellt wird:

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

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 ihr *dezimaler* Gesamtwert folgendermaßen ergibt:

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

Eine **float**- bzw. **double**-Variable mit dem Vorzeichen v (Null oder Eins), dem Exponenten e und dem dezimalen Mantissenwert m speichert also bei normalisierter Darstellung den Wert:

$$(-1)^v \cdot 2^{e-127} \cdot (1 + m) \quad \text{bzw.} \quad (-1)^v \cdot 2^{e-1023} \cdot (1 + m)$$

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	100000000000000000000000
1,0 = $(-1)^0 \cdot 2^{(127-127)} \cdot (1+0,0)$	0	01111111	000000000000000000000000
1,25 = $(-1)^0 \cdot 2^{(127-127)} \cdot (1+0,25)$	0	01111111	010000000000000000000000
-2,0 = $(-1)^1 \cdot 2^{(128-127)} \cdot (1+0,0)$	1	10000000	000000000000000000000000
2,75 = $(-1)^0 \cdot 2^{(128-127)} \cdot (1+0,25+0,125)$	0	10000000	011000000000000000000000
-3,5 = $(-1)^1 \cdot 2^{(128-127)} \cdot (1+0,5+0,25)$	1	10000000	110000000000000000000000

Nun kommen wir endlich zur Erklärung der eingangs dargestellten Genauigkeitsunterschiede beim Speichern der Zahlen 1,25 und 1,3. Während die Restmantisse

$$\begin{aligned} 0,25 &= 0 \cdot 2^{-1} + 1 \cdot 2^{-2} \\ &= 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} \end{aligned}$$

perfekt dargestellt werden kann, gelingt dies bei der Restmantisse 0,3 nur approximativ:

$$\begin{aligned} 0,3 &= 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} + \dots \\ &= 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} + 0 \cdot \frac{1}{8} + 0 \cdot \frac{1}{16} + 1 \cdot \frac{1}{32} + \dots \end{aligned}$$

Sehr aufmerksame Leser werden sich darüber wundern, wieso die Tabelle mit den elementaren Datentypen in Abschnitt 3.3.3 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:

$$(-1)^v \cdot 2^{-126} \cdot m \quad \text{bzw.} \quad (-1)^v \cdot 2^{-1022} \cdot m$$

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	000000000000000000000000
$-5,877472 \cdot 10^{-39} = (-1)^1 \cdot 2^{-126} \cdot 2^{-1}$	1	00000000	100000000000000000000000
$1,401298 \cdot 10^{-45} = (-1)^0 \cdot 2^{-126} \cdot 2^{-23}$	0	00000000	000000000000000000000001

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**:

```

C:\U:\Eigene Dateien\C#\Kurs\BspUeb\Elementare Sprachelemente\Bits\FloatBits\bin\Debug\Fl...
Bits einer float-Zahl anzeigen
float: -3,5
Bits:
1 12345678 12345678901234567890123
1 10000000 11000000000000000000000000000000
  
```

3.3.4.2 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 Nachkommastellen als Ganzzahl gespeichert, wobei Werte von 0 bis 28 erlaubt sind (Begründung folgt).

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

Eine **decimal**-Variable mit dem Vorzeichen v (0 oder 1), der Mantisse m und dem Exponenten e speichert den Wert:

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

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

$$\overbrace{99999999999999999999999999999999}^{28 \text{ Stellen}} < 2^{96}-1 < \overbrace{99999999999999999999999999999999}^{29 \text{ Stellen}}$$

Folglich kann jede Dezimalzahl mit maximal 28 Stellen (vor oder hinter dem Dezimaltrennzeichen) exakt in einer **decimal**-Variablen gespeichert werden.

Im folgenden Programm wird der (besonders in der Finanzmathematik relevante) Genauigkeitsvorteil des Datentyps **decimal** im Vergleich zu den binären Gleitkommatypen demonstriert:

Quellcode	Ausgabe
<pre>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); } }</pre>	<pre>0,10000004 0,099999999999999996 0,1</pre>

Allerdings hat der Typ **decimal** auch Nachteile im Vergleich zu **float** bzw. **double**

- Kleiner Wertebereich
- Hoher Speicherbedarf
- Hoher Zeitaufwand bei arithmetischen Operationen
- Keine Unterstützung für Sonderfälle wie +/- Unendlich und NaN (Not a Number) (vgl. Abschnitt 3.6)

Bei der Aufgabe,

$$13000000 - \sum_{i=1}^{10000000} 1,3$$

zu berechnen, ergeben sich für die Datentypen **double** und **decimal** folgende Genauigkeits- und Laufzeitunterschiede:

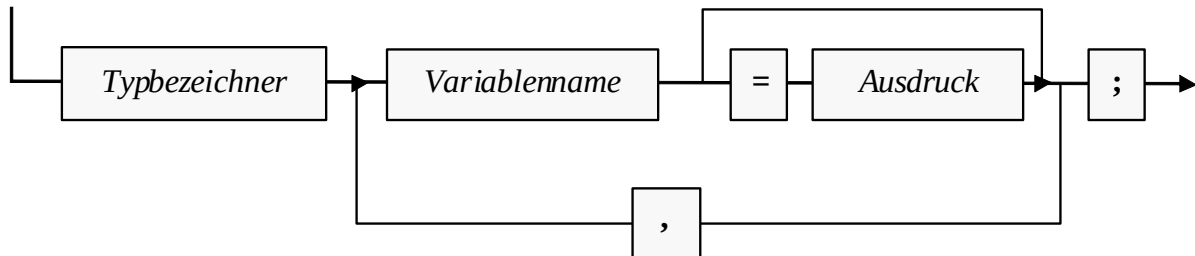
double:
 Abweichung: 0,0017944723367691
 Benöt. Zeit: 109,375 Millisek.

decimal:
 Abweichung: 0,0
 Benöt. Zeit: 1421,875 Millisek.

3.3.5 Variablendeklaration, Initialisierung und Wertzuweisung

In C#-Programmen muss jede Variable vor ihrer ersten Verwendung deklariert¹ werden. Dabei sind auf jeden Fall der Name und der Datentyp anzugeben, wie das Syntaxdiagramm zur **Variablen-deklarationsanweisung** zeigt:

Deklaration einer lokalen Variablen



Als Datentypen kommen in Frage (vgl. Abschnitt 3.3.1):

- 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 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();
```

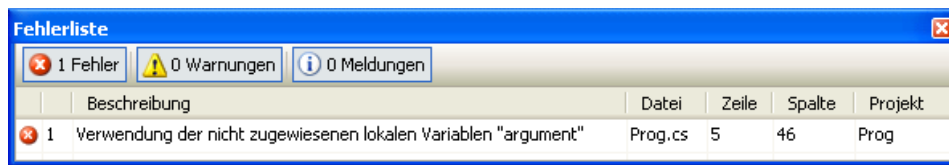
Im zweiten Beispiel wird per **new**-Operator ein **Bruch**-Objekt erzeugt und dessen Adresse in die neue Referenzvariable **b1** geschrieben. Mit der Objektkreation und auch mit der Konstruktion von gültigen *Ausdrücken*, die einen Wert von passendem Datentyp liefern müssen, werden wir uns noch ausführlich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

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

Der Compiler meint dazu:

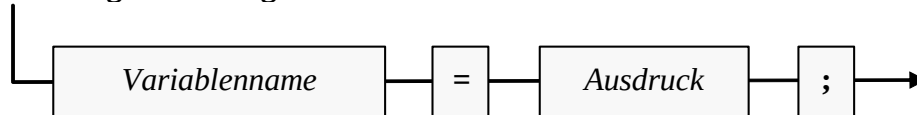
¹ Während in C++ die beiden Begriffe *Deklaration* und *Definition* verschiedene Bedeutungen haben, werden sie im Zusammenhang mit den meisten anderen Programmiersprachen (so auch bei C#) synonym verwendet. In diesem Manuskript wird im Zusammenhang mit Variablen bevorzugt von *Deklarationen*, im Zusammenhang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



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 Programmablauf zu verändern, verwendet man eine **Wertzuweisung**, die zu den einfachsten und am häufigsten benötigten Anweisungen gehört:

Wertzuweisungsanweisung



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

3.3.6 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 Klammern begrenzt:

Block- bzw. Verbundanweisung



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;
}

```

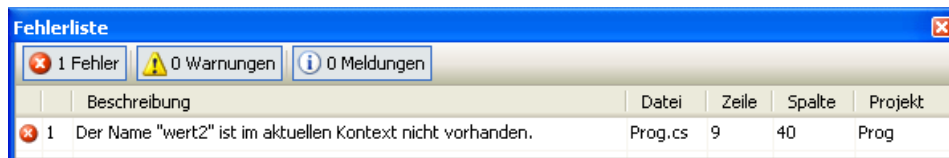
Anweisungsblöcke haben einen wichtigen Effekt auf die Gültigkeit der darin deklarierten Variablen: Eine lokale Variable ist verfügbar von der deklarierenden Zeile bis zur schließenden Klammer des innersten Blockes. Nur in diesem **Deklarations- bzw. Sichtbarkeitsbereich** kann sie über ihren Namen angesprochen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

```

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

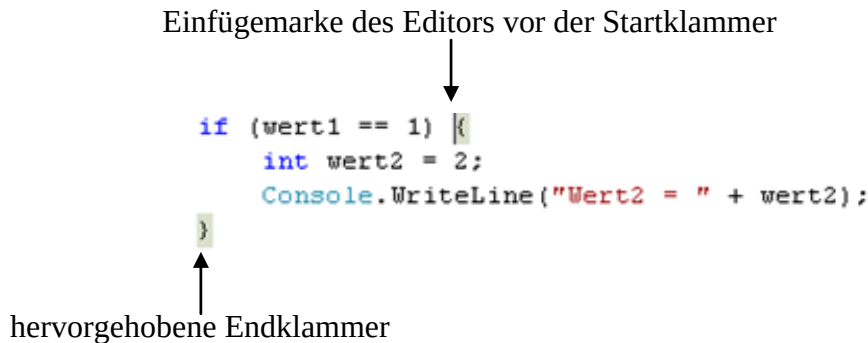
komplett nach rechts eingerückt

und mit

Umschalt + Tab

komplett nach links ausgerückt

werden. Außerdem kann man sich zu einer Blockklammer das Gegenstück anzeigen lassen:



3.3.7 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:

Quellcode	Ausgabe
<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>	Brutto: 33,44

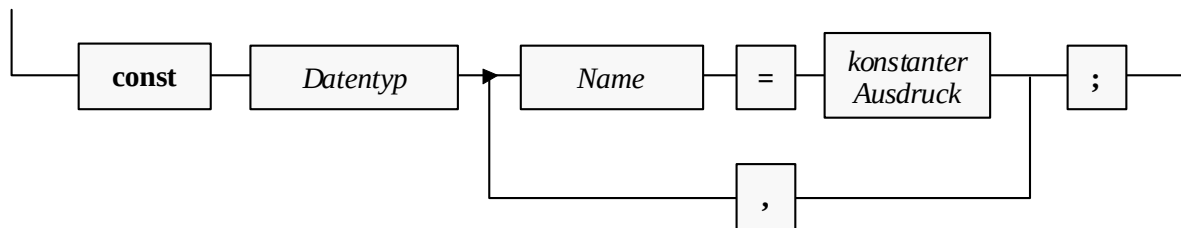
Im Vergleich zu einer Variablen weist eine Konstante folgende Besonderheiten auf:

- Ihre Deklaration beginnt mit dem Modifikator **const** und *muss* eine Initialisierung enthalten, wobei ein *konstanter* Ausdruck zu verwenden ist, der nur Literale (siehe Abschnitt 3.3.8) 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:

Deklaration einer konstanten lokalen Variablen

Neben lokalen Variablen können auch Felder einer Klasse als konstant deklariert werden (siehe Abschnitt 4.2.2).

3.3.8 Literale

Die im Programmcode auftauchenden expliziten Werte bezeichnet man als *Literale*. Wie Sie aus dem Abschnitt 3.3.7 wissen, ist es oft sinnvoll, Literale *innerhalb von Konstanten-Deklarationen* zu verwenden, z.B.:

```
const double MWST = 1.16;
```

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.

3.3.8.1 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:

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

Der Typ 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 wenig geeignet.

- 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, produziert die Entwicklungsumgebung eine Warnung, 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:

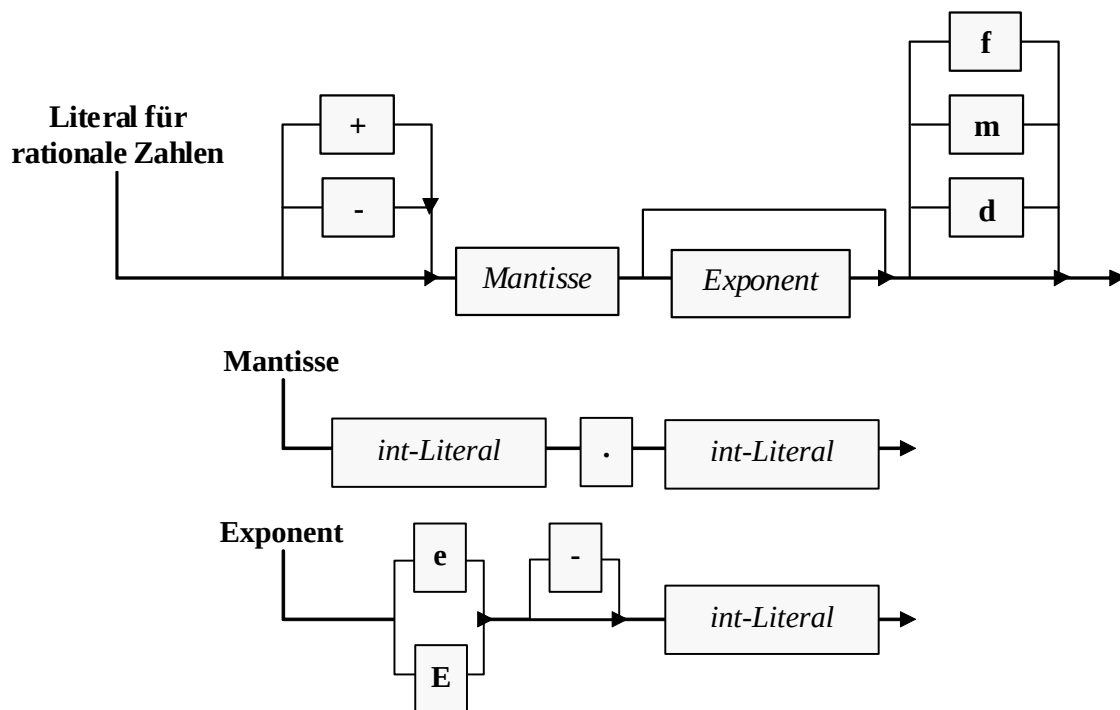
Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { Console.WriteLine(18446744073709551616); } }</pre>	<pre>Prog.cs(4,21): error CS1021: Die integrale Konstante ist zu groß</pre>

3.3.8.2 Gleitkommalliterale

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 Gleitkommalliteralen 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 3.3.8.1 beschriebenen Präfixe (0x, 0X) und Suffixe (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:

```
const float P = 1.25;
```

zu der Fehlermeldung:

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

3.3.8.3 bool-Literale

Als Literale vom Typ **bool** sind nur die beiden reservierten Wörter **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
<pre>using System; class Prog { static void Main() { bool b = false; Console.WriteLine(b); } }</pre>	

3.3.8.4 *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:

- o Ein Steuerzeichen, z.B.:

Neue Zeile	\n
Horizontaler Tabulator	\t
Alarmton	\a

- o Einfaches oder doppeltes Hochkomma sowie der Rückwärts-Schrägstrich:

```
\'  
\"  
\\
```

Beispiel:

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { char rs = '\\'; Console.WriteLine("Inhalt von rs: " + rs); } }</pre>	Inhalt von rs: \

- **Unicode-Escape-Sequenzen**

Eine Unicode-Escape-Sequenz enthält eine Unicode-Zeichennummer (vorzeichenlose Ganzzahl mit 16 Bits, also im Bereich von 0 bis $2^{16}-1 = 65535$) in hexadezimaler, vierstelliger Schreibweise (ggf. links mit Nullen aufgefüllt) nach der Einleitung durch \u oder \x. So lassen sich Zeichen ansprechen, die per Tastatur nicht einzugeben sind.

Beispiel:

```
const char alpha = '\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).

3.3.8.5 *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";
```

Zeichenkettenliterals sind vom Datentyp **string**, und später wird sich herausstellen, dass es sich bei diesem Typ um eine Klasse aus dem Namensraum **System** handelt.¹

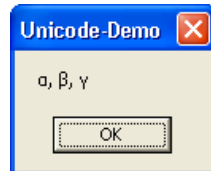
Während ein **char**-Literal stets *genau ein* Zeichen enthält, kann ein Zeichenkettenliteral aus beliebig vielen Zeichen bestehen oder auch leer sein, z.B.:

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

Das folgende Programm enthält einen Aufruf der statischen **Show()**-Methode der Klasse **MessageBox** aus dem Namensraum **System.Windows.Forms** (siehe Abschnitt 2.2.2.4) zur Anzeige eines Zeichenkettenliterals, das drei Unicode-Escape-Sequenzen enthält:

```
using System.Windows.Forms;
class Prog {
    static void Main() {
        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:\Programme\Mapp\bin");
```

3.3.9 Übungsaufgaben zu Abschnitt 3.3

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

Quellcode	Fehlermeldung
<pre>class Prog { static void Main() { { int i = 2; } System.Console.WriteLine(i); } }</pre>	Prog.cs(6,34): error CS0103: Der Name i ist im aktuellen Kontext nicht vorhanden.

2) Beseitigen Sie bitte alle Fehler in folgendem Programm:

¹ Das (klein geschriebene!) Schlüsselwort **string** steht in C# aus Bequemlichkeitsgründen als Aliasname für die Klassenbezeichnung **System.String** zur Verfügung.

```
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:

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

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

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 abgeschlossene 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 in der **int**-Eigenschaft `Zaehler`.

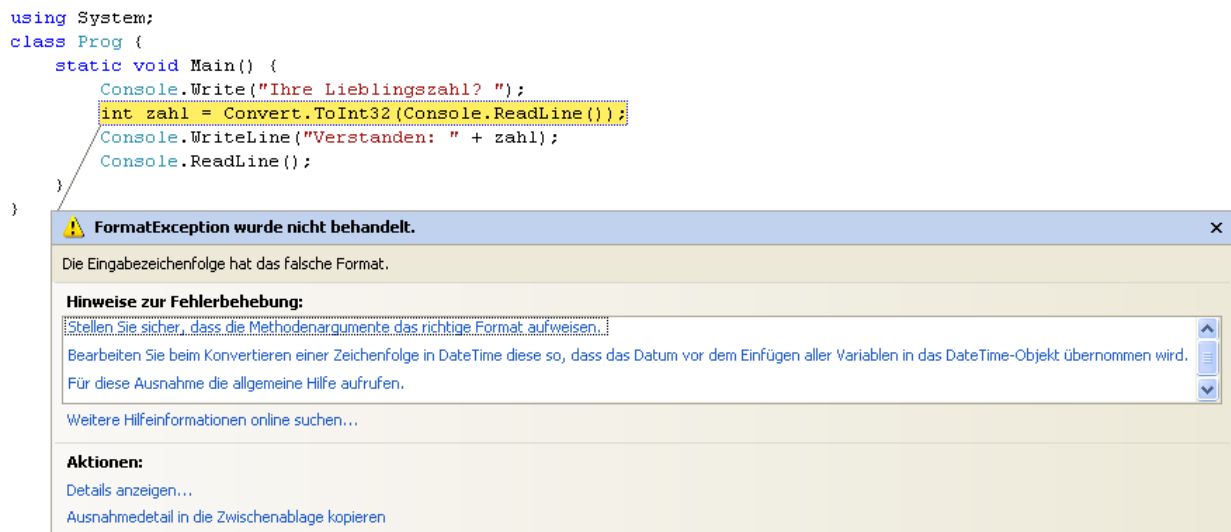
Hier liegt hier eine Verschachtelung zweier Methodenaufrufe vor, die bei Programmierern der kompakten Schreibweise wegen sehr beliebt ist. Ein etwas umständliches, aber für Anfänger leichter verständliches Äquivalent zur obigen Anweisung könnte lauten:

```
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 das Programm mit einem unbehandelten Ausnahmefehler, z.B.:

Quellcode	Ein- und Ausgabe
<pre>using System; class Prog { static void Main() { Console.Write("Ihre Lieblingszahl? "); int zahl = Convert.ToInt32(Console.ReadLine()); Console.WriteLine("Verstanden: " + zahl); } }</pre>	Ihre Lieblingszahl? drei Unbehandelte Ausnahme: System.FormatException: Die Eingabezeichenfolge hat das falsche Format.

Wenn Sie das Programm aus Visual C# heraus mit der Taste **F5** oder mit dem Schalter  (also im so genannten Debug-Modus) gestartet haben, führt der Ausnahmefehler zu folgender Anzeige:



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	Ein- und Ausgabe
<pre>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!"); } } }</pre>	Ihre Lieblingszahl? drei Falsche Eingabe!

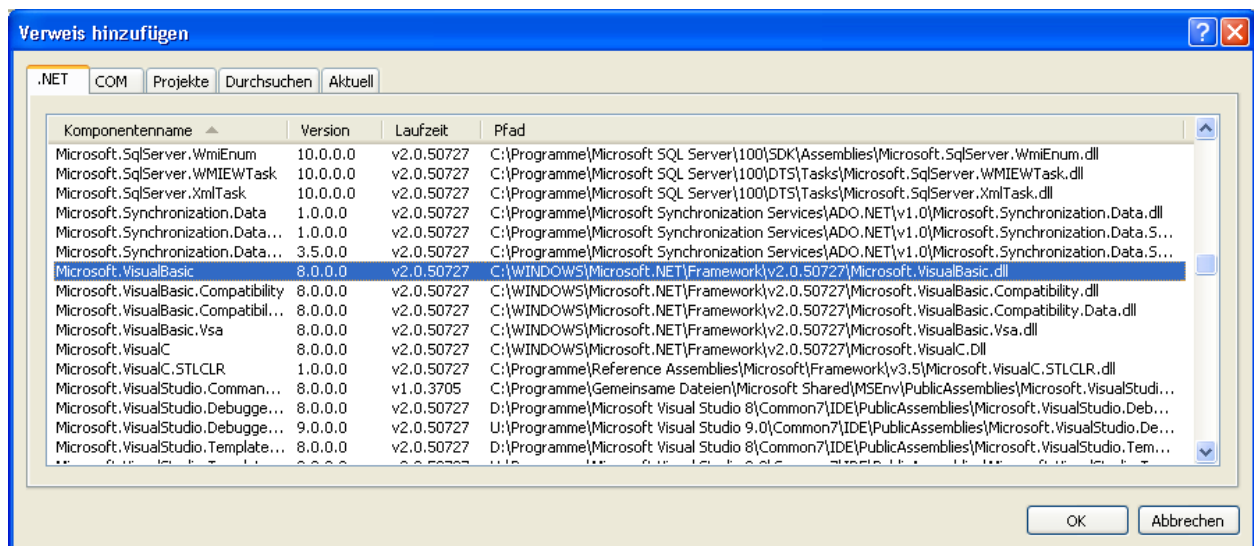
In den Übungs- bzw. Demoprogrammen verwenden wir der Einfachheit halber ungesicherte **ToInt32** – Aufrufe bzw. analoge Varianten für andere Datentypen.

3.4.2 Via InputBox

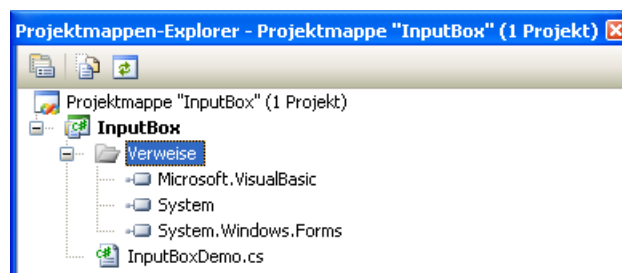
Wer schon jetzt Anwendungen mit grafikorientierter Benutzerinteraktion erstellen möchte, kann statt **ReadLine()** z.B. die statische Methode **InputBox()** der Klasse **Interaction** aus dem Namensraum **Microsoft.VisualBasic** benutzen. Die Klasse **Interaction** ist als Migrationshilfe für Visual Basic 6 - Programmierer gedacht, kann aber natürlich auch in C# - Programmen genutzt werden. Dazu muss dem Compiler das implementierende und im **GAC** (Global Assembly Cache) installierte Assembly **microsoft.visualbasic.dll** per **/reference** – Option bekannt gemacht werden, z.B.:

```
csc /reference:microsoft.visualbasic.dll Prog.cs
```

Wie Sie aus Abschnitt 2.2.2.4 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:

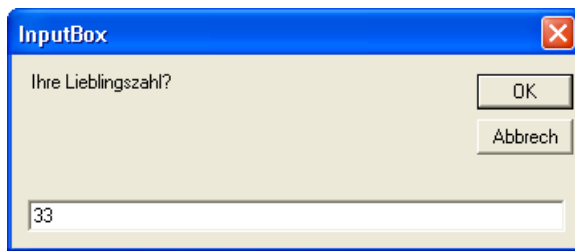


Aus dem Beispielprogramm in Abschnitt 3.4.1 entsteht folgende GUI-Variante:

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

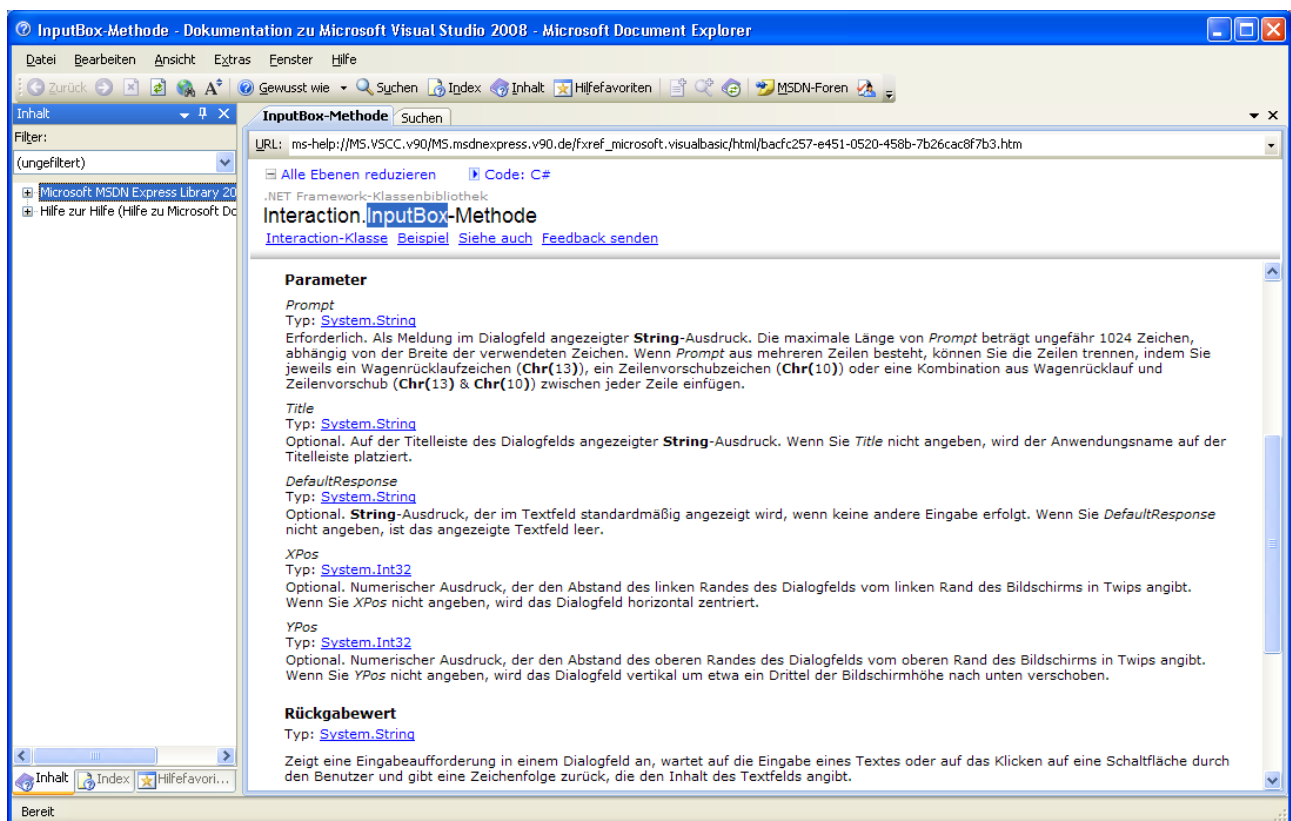
class InputBox {
    static void Main() {
        int zahl = Convert.ToInt32(
            Interaction.InputBox("Ihre Lieblingszahl?", "InputBox", "", -1, -1)
        );
        MessageBox.Show("Verstanden: "+zahl);
    }
}
```

Ein- und Ausgabe werden per Dialogbox erledigt:



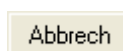
Wie man ein überflüssiges Konsolenfenster unterdrückt, wurde in Abschnitt 2.3 im Rahmen einer Übungsaufgabe gezeigt.

Über die Parameter (Argumente) und den Rückgabewert des **Inputbox()**-Methodenaufrufs kann man sich z.B. über die Hilfefunktion der Visual C# 2008 Express Edition informieren. Eine Suche nach dem Klassennamen **Interaction** führt schnell zum Ziel:



Die Angaben zu *optionalen Parametern* betreffen allerdings nur Visual Basic - Programmierer. In der CLS (*Common Language Specification*) und in C# werden optionale Parameter *nicht* unterstützt.

Zwar sieht die Ein- bzw. Ausgabe per GUI attraktiver aus, jedoch lohnt sich der erhöhte Aufwand bei Demo- bzw. Übungsprogrammen zu speziellen Sprachelementen kaum, so dass wir in der Regel darauf verzichten werden. Beim späteren Entwurf eigener Dialogboxen werden wir die Größen der Bedienelemente sorgfältig entwerfen und den folgenden Schönheitsfehler vermeiden, der offenbar durch den höheren Platzbedarf der deutschen Beschriftung im Vergleich zur englischen Variante entstanden ist:



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:

$$\begin{array}{c} \text{Operator} \\ \downarrow \\ \text{az} = \underbrace{\text{az} - \text{an}}_{\text{Ausdruck}}; \end{array}$$

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. Besteht ein Ausdruck aus einem Methodenaufruf mit dem Pseudorückgabetyt **void**, dann liegt allerdings *kein* Wert vor.

Beispiel: 1.5

Dies ist ein Ausdruck mit dem Typ **double** und dem Wert 1,5.

Mit Hilfe diverser Operatoren entstehen komplexere Ausdrücke, wobei Typ und Wert von den Argumenten und den Operatoren abhängen.

Beispiele: 2 * 1.5

Hier resultiert der **double**-Wert 3,0.

2 > 1.5

Hier resultiert der **boolean**-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 von den Argumenten. Bei Argumenten unterschiedlichen Typs 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.

Es hängt von den Datentypen der Argumente ab, ob die **Ganzzahl**-, oder die **Gleitkommaarithmetik** zum Einsatz kommt. Besonders auffällig sind die Unterschiede im Verhalten des Divisionsoperators, z.B.:

Quellcode	Ausgabe
<pre>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); } }</pre>	<pre>0 0,6666666666666667</pre>

Bei der Ganzzahldivision werden die Stellen nach dem Dezimaltrennzeichen abgeschnitten, was gelegentlich durchaus erwünscht ist. Im Zusammenhang mit dem Über- bzw. Unterlauf 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	
		Programmfragment	Ausgabe
<code>-Var</code>	Vorzeichenumkehr	<code>int i = 2; Console.WriteLine(-i);</code>	-2
<code>Num1 + Num2</code>	Addition	<code>Console.WriteLine(2 + 3);</code>	5
<code>Num1 - Num2</code>	Subtraktion	<code>Console.WriteLine(2.6 - 1.1);</code>	1,5
<code>Num1 * Num2</code>	Multiplikation	<code>Console.WriteLine(4 * 5);</code>	20
<code>Num1 / Num2</code>	Division	<code>Console.WriteLine(8.0 / 5); Console.WriteLine(8 / 5);</code>	1,6 1
<code>Num1 % Num2</code>	Modulo (Divisionsrest) Sei <i>GAD</i> der ganzzahlige Anteil aus dem Ergebnis der Division (<i>Num1</i> / <i>Num2</i>). Dann ist <i>Num1 % Num2</i> def. durch <i>Num1 - GAD · Num2</i>	<code>Console.WriteLine(19 % 5); Console.WriteLine(-19 % 5.4);</code>	4 -2,8
<code>++Var</code> <code>--Var</code>	Präinkrement bzw. -dekrement Als Argument ist nur eine Variable erlaubt. <code>++Var</code> liefert <i>Var</i> + 1 erhöht <i>Var</i> um 1 <code>--Var</code> liefert <i>Var</i> - 1 reduziert <i>Var</i> um 1	<code>int i = 4; double a = 1.2; Console.WriteLine(++i + "\n" + --a);</code>	5 0,2
<code>Var++</code> <code>Var--</code>	Postinkrement bzw. -dekrement Als Argument ist nur eine Variable erlaubt. <code>Var++</code> liefert <i>Var</i> erhöht <i>Var</i> um 1 <code>Var--</code> liefert <i>Var</i> reduziert <i>Var</i> um 1	<code>int i = 4; Console.WriteLine(i++ + "\n" + i);</code>	4 5

Bei den Inkrement- und den Dekrementoperatoren ist zu beachten, dass sie *zwei* Effekte haben:

- Das Argument wird ausgelesen, um den Wert des Ausdrucks zu ermitteln.
- Der Wert des Argumentes 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;
```

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 3.3.3) 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-Rückgabebetyp **void** anzugeben.
- Neben dem Rückgabewert hat ein Methodenaufruf oft weitere Effekte, z.B. auf die Merkmalsausprägungen des handelnden Objekts oder auf die Konsolenausgabe.

In syntaktischer Hinsicht stellen wir fest, dass ein Methodenaufruf einen **Ausdruck** darstellt, wobei seine Rückgabe den Datentyp und den Wert des Ausdrucks bestimmt. Bei passendem Rückgabebetyp darf ein Methodenaufruf auch als Argument für komplexere Ausdrücke oder für 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), wie Sie aus den zahlreichen Einsätzen der Methode **Console.WriteLine()** in unseren Beispielprogrammen bereits wissen.

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 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
<pre>using System; class Prog{ static void Main(){ Console.WriteLine(Math.Pow(2.0, 3.0)); } }</pre>	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 verständliches Äquivalent zum obigen **WriteLine()**-Aufruf könnte z.B. lauten:

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

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, z.B.:

```
if (arg > 0)
    Console.WriteLine(Math.Log(arg));
```

In der folgenden Tabelle mit den von C# unterstützten Vergleichsoperatoren stehen

- *Expr1* und *Expr2* für komparable Ausdrücke
- *Num1* und *Num2* für numerische Ausdrücke

Operator	Bedeutung	Beispiel	
		Programmfragment	Ausgabe
<i>Expr1</i> == <i>Expr2</i>	Gleichheit	<code>Console.WriteLine(2 == 3);</code>	False
<i>Expr1</i> != <i>Expr2</i>	Ungleichheit	<code>Console.WriteLine(2 != 3);</code>	True
<i>Num1</i> > <i>Num2</i>	größer	<code>Console.WriteLine(3 > 2);</code>	True
<i>Num1</i> < <i>Num2</i>	kleiner	<code>Console.WriteLine(3 < 2);</code>	False
<i>Num1</i> >= <i>Num2</i>	größer oder gleich	<code>Console.WriteLine(3 >= 3);</code>	True
<i>Num1</i> <= <i>Num2</i>	kleiner oder gleich	<code>Console.WriteLine(3 <= 2);</code>	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>	False 1

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

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

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

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
<pre>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); } }</pre>	<pre>False True True</pre>

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 3.3.4.1 beschriebenen Genauigkeitsprobleme bei der Speicherung von binären Gleitkommazahlen berücksichtigt, ist das Vergleichsergebnis durchaus nicht überraschend.

Für Anwendungen im Bereich der Finanzmathematik wurde schon in Abschnitt 3.3.4 der dezimale Gleitkommatyp **decimal** vorgeschlagen. Hier sind bei der Speicherung 28 signifikante Dezimalstellen garantiert, und Berechnungen erfolgen mit einer Genauigkeit von bis zu 28 Dezimalstellen. 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 Mantissen-Fehler bei zwei **double**-Variablen mit eigentlich identischem Wert in Abhängigkeit vom Exponenten zu sehr unterschiedlichen Gesamtfehlern führen können. Vom häufig anzutreffenden Vorschlag, $|d_1 - d_2|$ mit einer Schwelle zu vergleichen, ist daher abzuraten. Die-

ses 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 enthalten:

$$d_1 = (t + e_1) 10^k \quad \text{und} \quad d_2 = (t + e_2) 10^k$$

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 Abschätzung:

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

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 3.3.4.1) benötigt wird,
- dass die definierte Indifferenzrelation nicht transitiv ist.

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 Unterschiedsschwelle 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	Negation
<i>La1</i>	<i>!La1</i>
true	false
false	true

Argument 1 <i>La1</i>	Argument 2 <i>La2</i>	Logisches UND <i>La1 && La2</i> <i>La1 & La2</i>	Logisches ODER <i>La1 La2</i> <i>La1 La2</i>	Exklusives ODER <i>La1 ^ La2</i>
true	true	true	true	false
true	false	false	true	true
false	true	false	true	true
false	false	false	false	false

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

Operator	Bedeutung	Beispiel	
		Programmfragment	Ausgabe
<i>!La1</i>	Negation Der Wahrheitswert wird umgekehrt.	<code>bool erg = true; Console.WriteLine(!erg);</code>	False
<i>La1 && La2</i>	Logisches UND (mit bedingter Auswertung) <i>La1 && La2</i> ist genau dann wahr, wenn beide Argumente wahr sind. Ist <i>La1</i> falsch, wird <i>La2</i> nicht ausgewertet.	<code>int i = 3; bool erg = false && i++ > 3; Console.WriteLine(erg + "\n" + i);</code> <code>erg = true && i++ > 3; Console.WriteLine(erg + "\n" + i);</code>	False 3 False 4
<i>La1 & La2</i>	Logisches UND (mit unbedingter Auswertung) <i>La1 & La2</i> ist genau dann wahr, wenn beide Argumente wahr sind. Es werden auf jeden Fall beide Ausdrücke ausgewertet.	<code>int i = 3; bool erg = false & i++ > 3; Console.WriteLine(erg + "\n" + i);</code>	False 4
<i>La1 La2</i>	Logisches ODER (mit bedingter Auswertung) <i>La1 La2</i> ist genau dann wahr, wenn mindestens ein Argument wahr ist. Ist <i>La1</i> wahr, wird <i>La2</i> nicht ausgewertet.	<code>int i = 3; bool erg = true i++ == 3; Console.WriteLine(erg + "\n" + i);</code> <code>erg = false i++ == 3; Console.WriteLine(erg + "\n" + i);</code>	True 3 True 4
<i>La1 La2</i>	Logisches ODER (mit unbedingter Auswertung) <i>La1 La2</i> ist genau dann wahr, wenn mindestens ein Argument wahr ist. Es werden auf jeden Fall beide Ausdrücke ausgewertet.	<code>int i = 3; bool erg = true i++ > 3; Console.WriteLine(erg + "\n" + i);</code>	True 4
<i>La1 ^ La2</i>	Exklusives logisches ODER <i>La1 ^ La2</i> ist genau dann wahr, wenn genau ein Argument wahr ist, wenn also die Argumente verschiedene Wahrheitswerte haben.	<code>Console.WriteLine(true ^ true);</code>	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 105 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). Weil diese Operatoren zwei *integrale* Argumente (z.B. Datentyp **int**) erwarten, kann der Compiler allerdings 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 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 **CBit** 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 Indexvariable `i` mit dem Wert 15, der am Ende jedes Schleifendurchgangs um Eins dekrementiert wird (`i--`). Ob es zum nächsten Schleifendurchgang kommt, hängt von der Fortsetzungsbedingung ab (`i >= 0`):

Quellcode	Ausgabe
<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>	<pre>Zeichen: x Unicode: 00000000001111000</pre>

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 3.3.3), interessieren im Augenblick nur die rechten 16. Bei der Eins erhalten wir:

```
000000000000000001
```

Im 10. Schleifendurchgang ($i = 6$) geht dieses Muster z.B. über in:

```
00000000001000000
```

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 i eine Eins enthält, wenn *beide* Argumentmuster an dieser Stelle eine Eins besitzen. Bei `cbit = 'x'` ist das Unicode-Bitmuster

```
00000000001111000
```

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

```
00000000001000000
```

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 im i -ten Schleifendurchgang genau dann wahr, wenn das 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 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	Ausgabe
<pre>using System; class Prog { static void Main() { long i = 9223372036854775313; double d = i; Console.WriteLine(i); Console.WriteLine("{0,20:f2}", d); } }</pre>	<pre>9223372036854775313 9223372036854780000,00</pre>

Eine Abweichung von 4687 (z.B. Euro oder Meter) kann durchaus Proteste von Kunden oder Schlimmeres zur Folge haben.

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

Quellcode	Ausgabe
<pre>class Prog { static void Main() { System.Console.WriteLine("x/2 \t= " + 'x' / 2); System.Console.WriteLine("x*0,27 \t= " + 'x' * 0.27); } }</pre>	<pre>x/2 = 60 x*0,27 = 32,4</pre>

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**-interpretation 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:

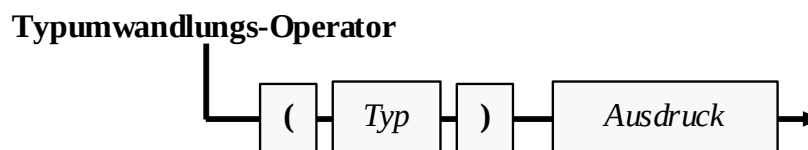
Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { Console.WriteLine((int)'x'); double a = 3.14159; Console.WriteLine((int)a); a = 1.45367e50; Console.WriteLine((int)a); } }</pre>	<pre>120 3 -2147483648</pre>

In der zweiten Ausgabeanweisung des Beispielprogramms wird per Casting-Operation der ganzzahlige Anteil eines **double**-Wertes ermittelt, was im Programmieralltag gelegentlich sinnvoll ist. Wie die dritte Ausgabe zeigt, sind bei einer explizit angeforderten **einschränkenden Konvertierung** kapitale Programmierfehler möglich, wenn die Wertebereiche der beteiligten Variablen bzw. Datentypen nicht beachtet werden. So soll die Explosion der europäischen Weltraumrakete Ariane-5 am 4. Juni 1996 (Schaden: ca. 500 Millionen Dollar)



durch die Konvertierung eines **double**-Werts (mögliches Maximum: $1,7976931348623157 \cdot 10^{308}$) in einen **short**-Wert (mögliches Maximum: $2^{15} - 1 = 32767$) verursacht worden sein.

Die C# - Syntax zur expliziten Typumwandlung:



3.5.8 Zuweisungsoperatoren

Bei den ersten Erläuterungen zur Wertzuweisung (vgl. Abschnitt 3.3.5) 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 kompatibellem 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
<pre> using System; class Prog { static void Main() { int ivar = 13; Console.WriteLine(ivar = 4711); Console.WriteLine(ivar); } } </pre>	<pre> 4711 4711 </pre>

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
<pre>using System; class Prog { static void Main() { int i = 2, j = 4; i = j = j * i; Console.WriteLine(i + "\n" + j); } }</pre>	<pre>8 8</pre>

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.

Wie wir seit Abschnitt 3.3.5 wissen, stellt ein Zuweisungsausdruck bereits eine vollständige **Anweisung** dar, sobald man ein Semikolon dahinter setzt. Dies gilt auch für 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>i</i>
<i>Var</i> += <i>Expr</i>	<i>Var</i> erhält den neuen Wert <i>Var</i> + <i>Expr</i> .	<code>int i = 2; i += 3;</code>	5
<i>Var</i> -= <i>Expr</i>	<i>Var</i> erhält den neuen Wert <i>Var</i> - <i>Expr</i> .	<code>int i = 10, j = 3; i -= j * j;</code>	1
<i>Var</i> *= <i>Expr</i>	<i>Var</i> erhält den neuen Wert <i>Var</i> * <i>Expr</i> .	<code>int i = 2; i *= 5;</code>	10
<i>Var</i> /= <i>Expr</i>	<i>Var</i> erhält den neuen Wert <i>Var</i> / <i>Expr</i> .	<code>int i = 10; i /= 5;</code>	2
<i>Var</i> %= <i>Expr</i>	<i>Var</i> erhält den neuen Wert <i>Var</i> % <i>Expr</i> .	<code>int i = 10; i %= 5;</code>	0

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, 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.

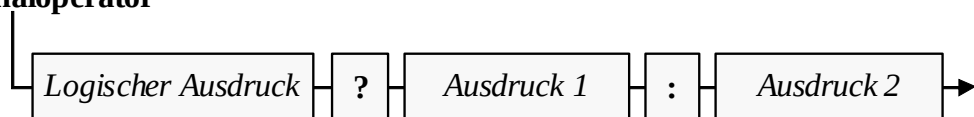
$$i = \begin{cases} i + j & \text{falls } k > 0 \\ i - j & \text{sonst} \end{cases}$$

In C# ist für diese Zuweisung mit Fallunterscheidung nur eine einzige Zeile erforderlich:

Quellcode	Ausgabe
<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>	3

Eine Besonderheit des Konditionaloperators besteht darin, dass er *drei* Argumente verarbeitet, welche durch die Zeichen **?** und **:** getrennt werden:

Konditionaloperator



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 *Auswertungsreihenfolge* nach Möglichkeit gemieden. Nun werden die Regeln vorgestellt, nach denen der C# - Compiler komplexe Ausdrücke mit mehreren Operatoren auswertet:

1) Priorität

Zunächst entscheidet die Priorität der Operatoren (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

2) 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):

- Mit Ausnahme der Zuweisungsoperatoren sind alle binären Operatoren 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

$$x = y = z$$

ausgewertet als

$$x = (y = z)$$

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.:

$$(3 - 2) - 1 = 0 \neq 3 - (2 - 1) = 2$$

3) Klammern

Wenn aus obigen Regeln nicht die gewünschte Auswertungsfolge resultiert, greift man mit runden Klammern steuernd ein. Die Auswertung von (eventuell mehrstufig) eingeklammerten Teilausdrücken erfolgt von innen nach außen.

In der folgenden Tabelle sind die bisher behandelten Operatoren in absteigender Priorität aufgelistet. Gruppen von Operatoren mit gleicher Priorität sind durch fette horizontale Linien voneinander abgegrenzt. In der **Operanden**-Spalte werden die zulässigen Datentypen der Argumentausdrücke mit Hilfe der folgenden Platzhalter beschrieben:

<i>N</i>	Ausdruck mit numerischem Datentyp
<i>I</i>	Ausdruck mit ganzzahligem (integralem) Datentyp
<i>L</i>	logischer Ausdruck
<i>B</i>	Ausdruck mit beliebigem kompatiblen Datentyp
<i>S</i>	String (Zeichenfolge)
<i>V</i>	Variable mit beliebigem kompatiblen Datentyp
<i>V_n</i>	Variable mit numerischem Datentyp

Operator	Bedeutung	Operanden
++, --	Postinkrement bzw. -dekrement	V _n
-	Vorzeichenumkehr	N
!	Negation	L
++, --	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
>, <, >=, <=	Vergleichsoperatoren	N, N
==, !=	Gleichheit, Ungleichheit	B, B
&	Bitweises UND	I, I
&	Logisches UND (mit unbedingter Auswertung)	L, L
^	Exklusives logisches ODER	L, L
	Bitweises ODER	I, I
	Logisches ODER (mit unbedingter Auswertung)	L, L
&&	Logisches UND (mit bedingter Auswertung)	L, L
	Logisches ODER (mit bedingter Auswertung)	L, L
? :	Konditionaloperator	L, B, B
=	Wertzuweisung	V, B
+=, -=, *=, /=, %=	Wertzuweisung mit Aktualisierung	V _n , N

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}\n erg2 = {1}", erg1, erg2);
    }
}
```

3) Welche Wahrheitswerte 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:

$$\begin{aligned} \neg(la1 \ \&\& \ la2) &= \neg la1 \ || \ \neg la2 \\ \neg(la1 \ || \ la2) &= \neg la1 \ \&\& \ \neg la2 \end{aligned}$$

4) Erstellen Sie ein Programm, das den Exponentialfunktionswert e^x 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 Argumentes eine Variante der in Abschnitt 3.4 beschriebenen Technik, wobei die **Convert**-Methode **ToInt32()** zu ersetzen ist durch **ToDouble()**.

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 3.3.3). 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

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.
- Schließlich wird der **long**-Variablen *k* das falsche Ergebnis zugewiesen.

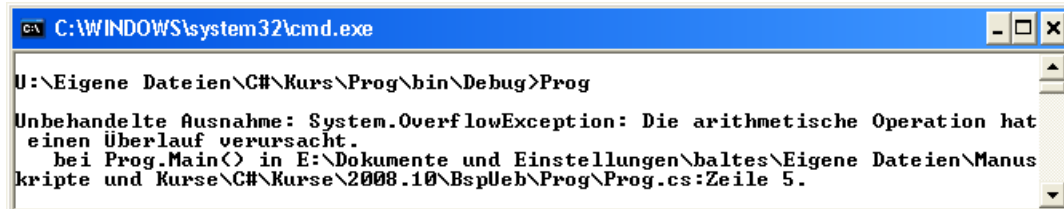
In C# steht im Unterschied zu vielen anderen Programmiersprachen mit dem **checked**-Operator eine Möglichkeit bereit, den Überlauf bei Ganzzahlvariablen abzufangen. Eine gesicherte Variante des ursprünglichen Beispielprogramms

```

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:



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 folgende Möglichkeiten, die Überlaufdiagnose für Ganzzahltypen einzuschalten:

- **checked**-Anweisung

Man kann die Überwachung für einen kompletten Anweisungsblock einschalten.

Beispiel:

```

checked {
    . . .
    k = i + j;
    . . .
}

```

- **checked**-Compileroption

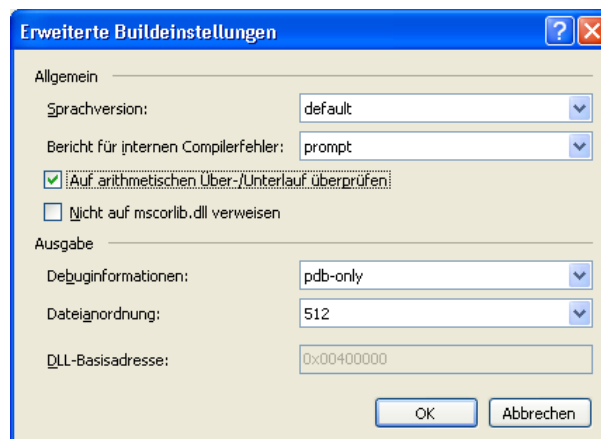
Man kann die Überwachung für einen kompletten Compiler-Lauf einschalten.

Beispiel:

```
csc /checked Prog.cs
```

Für ein Projekt der Visual C# 2008 Express Edition vereinbart man diese Compiler-Option folgendermaßen:

- o Menübefehl **Projekt > Eigenschaften**
- o Registerkarte **Erstellen**
- o Schalter **Erweitert**
- o Kontrollkästchen **Auf arithmetischen Über-/Unterlauf überprüfen**



Der Vollständigkeit halber soll an dieser Stelle noch der **unchecked**-Operator erwähnt werden, mit dem sich die Typüberwachung des Compilers abschalten lässt, was für sehr spezielle Zwecke (vielleicht zum Programmieren von Pseudozufallszahlengeneratoren) durchaus einmal sinnvoll sein kann. Im folgenden Beispiel ist in der ersten Wertzuweisung an die **int**-Variable die Überwachung abgeschaltet, während der Zuweisungsversuch in der nächsten Zeile das normale Verhalten des Compilers demonstriert.

```
int i;
i = unchecked{(int) 2147483648};
i = 2147483648;
```

```
struct System.UInt32
Represents a 32-bit unsigned integer.
```

Fehler:

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

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“

Unendlich – Unendlich

$$\frac{\text{Unendlich}}{\text{Unendlich}}$$

Unendlich · 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 Struktur¹ **Double** aus dem Namensraum **System** den größten Wert in Erfahrung bringt, der in einer **double**-Variablen gespeichert werden kann.

Über die statischen **Double**-Methoden

- **IsPositiveInfinity()**
- **IsNegativeInfinity()**
- **IsNaN()**

lässt sich für eine **double**-Variable prüfen, ob sie einen unendlichen oder undefinierten Wert besitzt, z.B.:

Quellcode	Ausgabe
<pre> using System; class Prog { static void Main() { double d = 0.0 / 0.0; if (Double.IsNaN(d)) Console.WriteLine("0/0 ist NaNsense"); } } </pre>	0/0 ist NaNsense

Die **if**-Anweisung (siehe Abschnitt 3.7.2) sorgt dafür, dass die Ausgabe nur unter einer bestimmten Bedingung (Rückgabewert **true** von **IsNaN()**) erfolgt.

Für besonders neugierige Leser sollen abschließend noch die **float**-Darstellungen der speziellen Gleitkommawerte angegeben werden (vgl. Abschnitt 3.3.4.1):

Wert	float-Darstellung		
	Vorz.	Exponent	Mantisse
+unendlich	0	11111111	000000000000000000000000
-unendlich	1	11111111	000000000000000000000000
NaN	0	11111111	100000000000000000000000

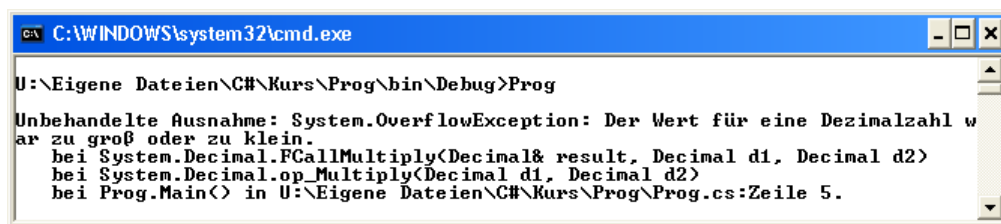
¹ Bei den später noch ausführlich zu behandelnden *Strukturen* handelt es sich um Werttypen mit starker Verwandtschaft zu den Klassen. Insbesondere wird sich zeigen, dass die elementaren Datentypen (z.B. **double**) auf Strukturtypen aus dem Namensraum **System** abgebildet werden (z.B. **Double**).

3.6.3 Überlauf beim Typ decimal

Beim Typ **decimal** wird im Fall eines Überlaufs *nicht* mit dem speziellen Wert Unendlich weiter gearbeitet, sondern eine Ausnahme gemeldet, die unbehandelt zur Beendigung des Programms führt. Im Unterschied zu den Ganzzahltypen 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:



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
<pre>using System; class Prog { static void Main() { double smallD = Double.Epsilon; Console.WriteLine(smallD); smallD /= 10.0; Console.WriteLine(smallD); } }</pre>	<pre>4,94065645841247E-324 0</pre>

Das öffentliche Feld **Double.Epsilon** enthält den betragsmäßig kleinsten Wert, der in einer **double**-Variablen gespeichert werden kann (vgl. Abschnitt 3.3.4.1 zu denormalisierten Werten bei den binären Gleitkommatypen **float** und **double**).

Erfreulicherweise wird ein Unterlauf bei Zwischenergebnissen wirksam verhindert. Das folgende Programm liefert für den Ausdruck $a * 0.1 * b * 10.0 * c$ ein recht präzises Resultat, obwohl im Zwischenergebnis $a * 0.1$ ein irreversibler Unterlauf zu befürchten war:

Quellcode	Ausgabe
<pre>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); Console.WriteLine(a * 0.1 * b * 10.0 * c); } }</pre>	<pre>9.88131291682493 0 9,88131291682493</pre>

In Java 6.0 tritt der befürchtete Unterlauf beim Zwischenergebnis tatsächlich auf. Dort kann aber mit Objekten der Klasse **BigDecimal** an Stelle von **double**-Werten wirksam Abhilfe geschaffen werden.

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 (Variablen-deklarationen, 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

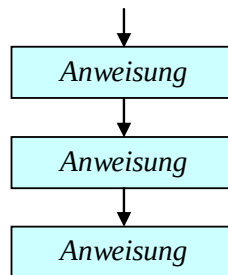
Ausführbare Programmteile, die in C# stets als Methoden von Klassen zu realisieren sind, bestehen aus Anweisungen (engl. *statements*).

Am Ende von Abschnitt 3.7 werden Sie die folgenden Sorten von Anweisungen kennen:

- **Variablendeklarationsanweisung**
Die Variablendeklarationsanweisung wurde schon in Abschnitt 3.3.5 eingeführt.
Beispiel: `int i = 1, k;`
- **Ausdrucksanweisungen**
Folgende Ausdrücke werden zu Anweisungen, sobald man ein Semikolon dahinter setzt:
 - o **Wertzuweisung** (vgl. Abschnitte 3.3.5 und 3.5.8)
Beispiel: `k = i + j;`
 - o **Prä- bzw. Postinkrement- oder -dekrementoperation**
Beispiel: `i++;`
Hier ist nur der „Nebeneffekt“ des Ausdrucks `i++` von Bedeutung. Sein Wert bleibt ungenutzt.
 - o **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 geschweifte Klammern zusammengefasst bzw. abgegrenzt werden, bildet eine **Verbund- bzw. Blockanweisung**. Wir haben uns bereits in Abschnitt 3.3.6 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.

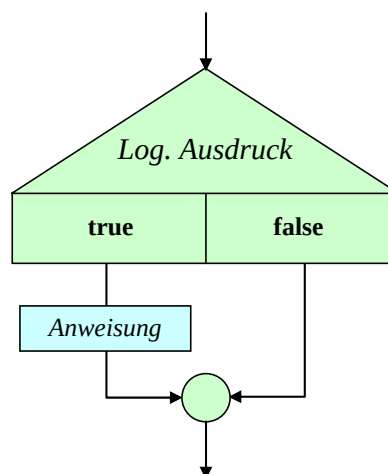
Anweisungen werden durch ein **Semikolon** abgeschlossen, sofern sie nicht mit einer schließenden Blockklammer enden.

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** bzw. **Flussdiagramm** soll eine (Block-)Anweisung nur dann ausgeführt werden, wenn ein logischer Ausdruck den Wert **true** besitzt:



Zur Realisation verwendet man die **if**-Anweisung mit der folgenden Syntax:

if-Anweisung

Als Anweisung ist allerdings keine Variablendeklaration erlaubt. Im folgenden Beispiel wird eine Meldung ausgegeben, wenn die Variable `anz` den Wert Null besitzt:

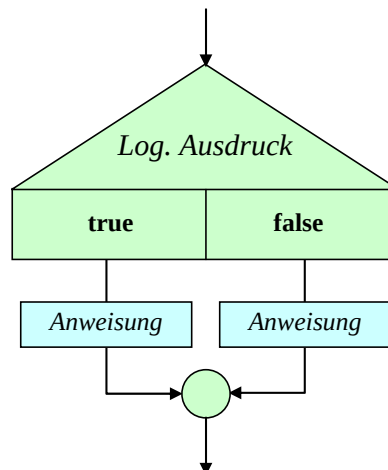
```
if (anz == 0)
    Console.WriteLine("Die Anzahl muss > 0 sein!");
```

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.

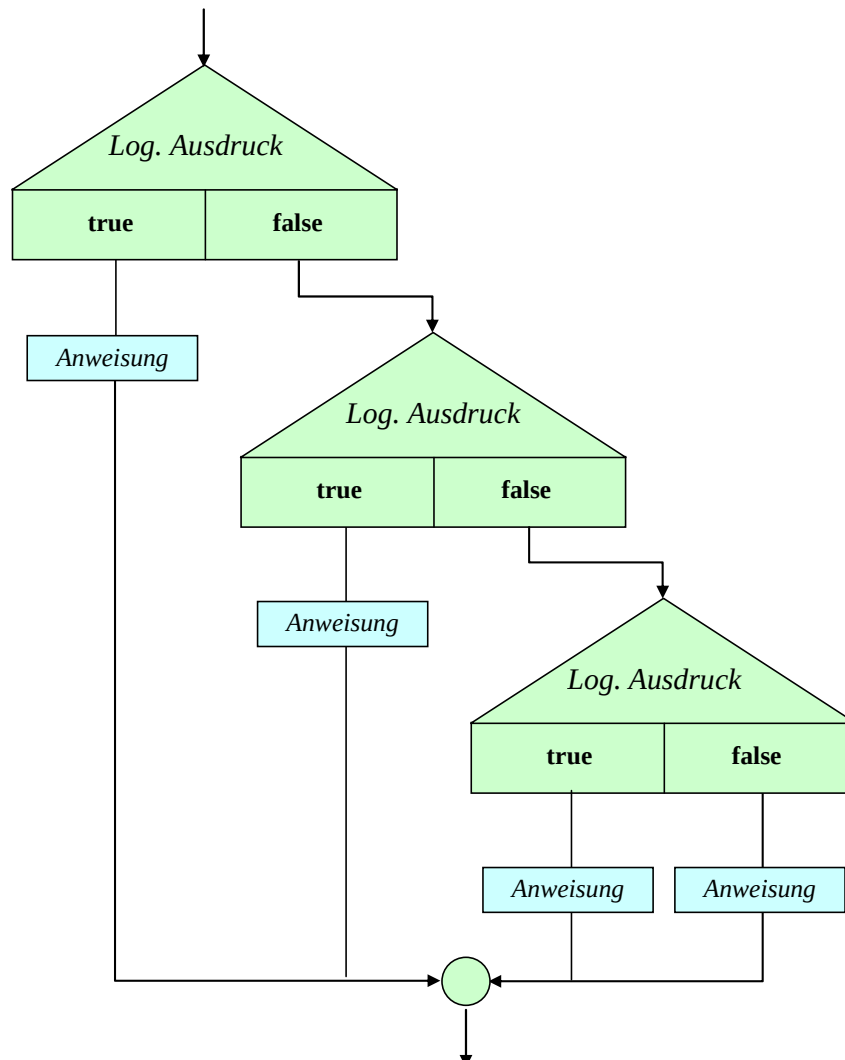
Wie schon bei der einfachen **if**-Anweisung gilt auch bei der **if-else**-Anweisung, dass Variablendeklarationen nicht als eingebettete Anweisungen erlaubt sind.

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 Ar-

gument wird vom Benutzer über eine **ToDouble()-ReadLine()** - Konstruktion erfragt (vgl. Abschnitt 3.4.1).

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { Console.WriteLine("Argument: "); double arg = Convert.ToDouble(Console.ReadLine()); if (arg > 0) Console.WriteLine("ln(" + arg + ") = " + Math.Log(arg)); else Console.WriteLine("\aArgument <= 0!"); } }</pre>	<pre>Argument: 2 ln(2) = 0,693147180559945</pre>

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.

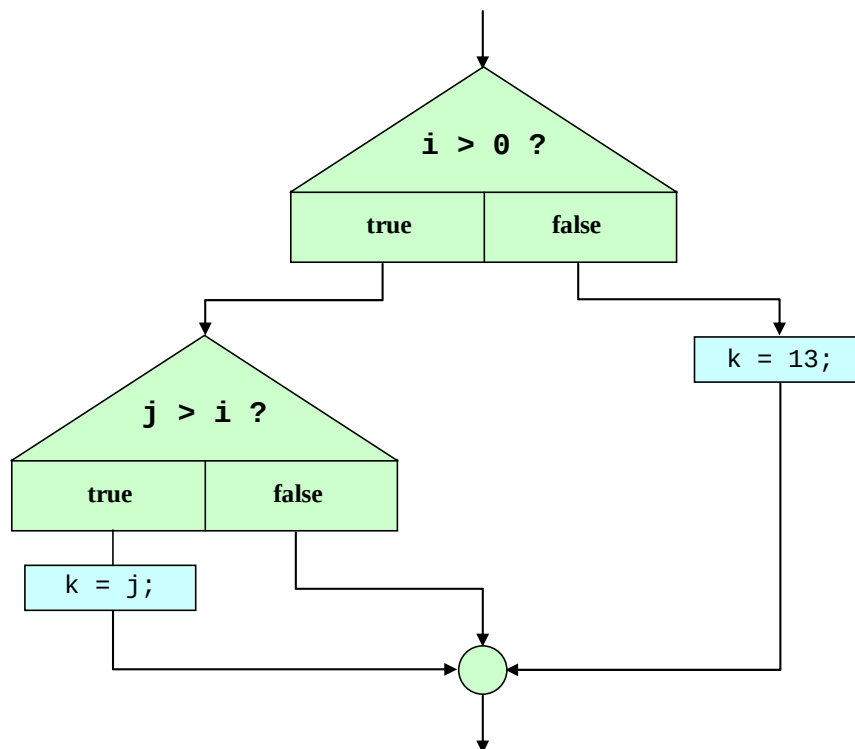
Beim Schachteln von bedingten Anweisungen kann es zum so genannten **dangling-else** - Problem¹ kommen, wobei ein Missverständnis zwischen Compiler und Programmierer hinsichtlich der Zuordnung einer **else**-Klausel besteht. Im folgenden Codefragment

```

if (i > 0)
    if (j > i)
        k = j;
else
    k = 13;

```

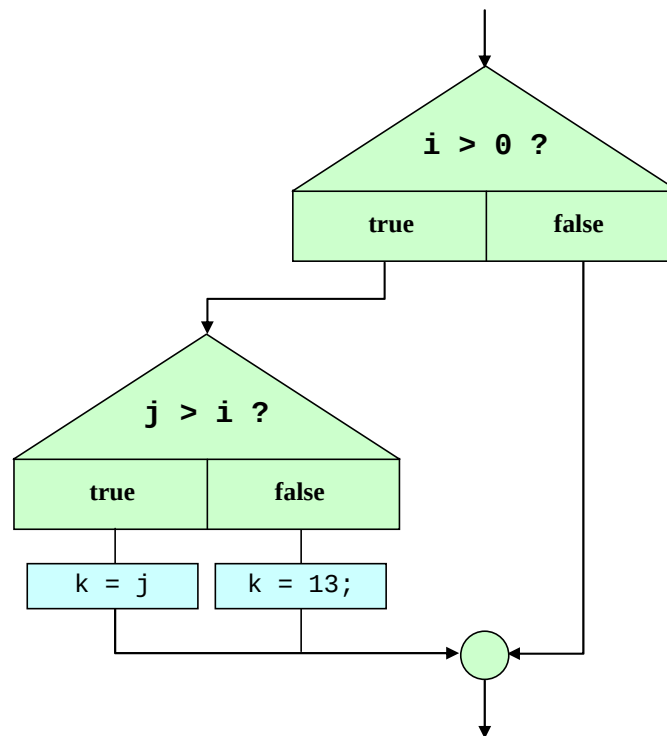
lassen die Einrücktiefen vermuten, dass der Programmierer die **else**-Klausel auf die *erste if*-Anweisung bezogen zu haben glaubt:



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;
  
```

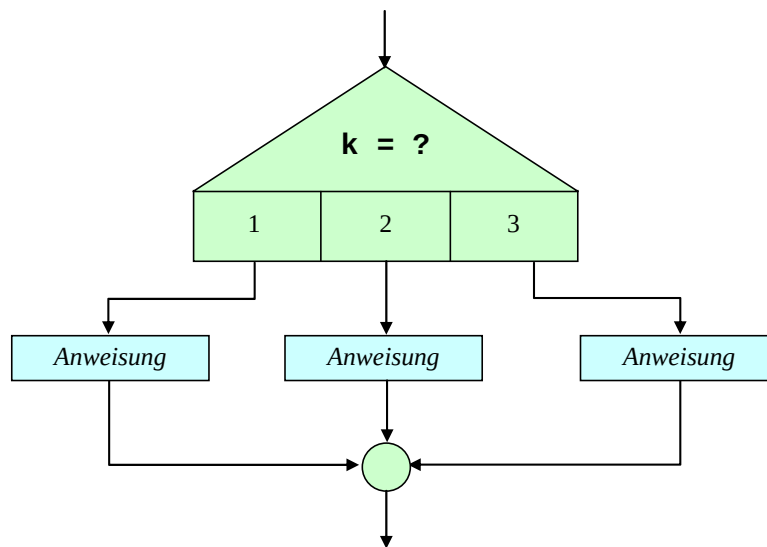
Alternativ kann man auch dem zweiten **if** eine **else**-Klausel spendieren und dabei eine leere Anweisung verwenden:

```

if (i > 0)
    if (j > i)
        k = j;
    else
        ;
else
    k = 13;
  
```

3.7.2.3 switch-Anweisung

Wenn eine Fallunterscheidung mit mehr als zwei Alternativen in Abhängigkeit vom Wert *eines* Ausdrucks vorgenommen werden soll,

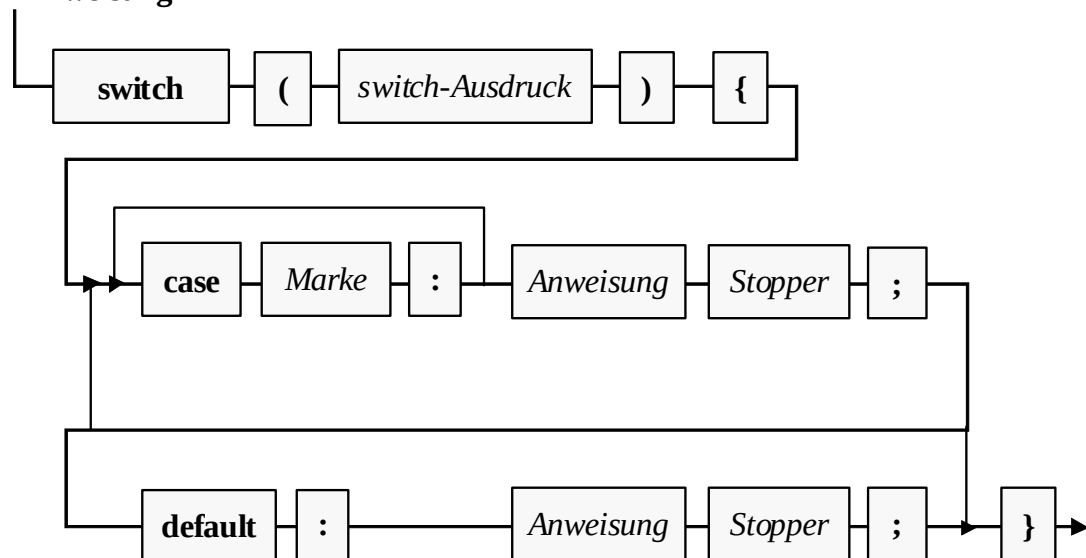


dann ist eine **switch**-Anweisung weitaus handlicher als eine verschachtelte **if-else** - Konstruktion. In Bezug auf den Datentyp des steuernden Ausdrucks ist C# sehr flexibel und erlaubt:

- numerische Datentypen
Dazu gehört auch der *integrale* Datentyp **char** (vgl. Abschnitt 3.3.3).
- 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.

switch-Anweisung



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\n (mit Durchfall per goto)"); goto case 3; case 3: case 4: Console.WriteLine("Fälle 3 und 4"); break; default: Console.WriteLine("Restkategorie"); break; } } } </pre>	Fall 2 (mit Durchfall per goto) Fälle 3 und 4

Als **case**-Marken sind *konstante* Ausdrücke erlaubt, deren Ergebnis schon der Compiler ermitteln kann (z.B. Literale, Konstanten oder mit konstanten Argumenten gebildete Ausdrücke).

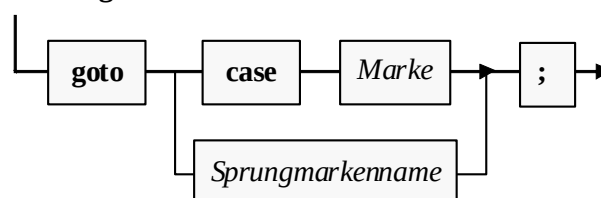
Stimmt bei der Ausführung einer Methode der Wert des **switch**-Ausdrucks mit einer **case**-Marke überein, dann wird die zugehörige Anweisung ausgeführt, ansonsten (falls vorhanden) die **default**-Anweisung.

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. 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 sind:

- **return**-Anweisung
Die Methode wird verlassen.
- **throw**-Anweisung
Es wird eine Ausnahme ausgelöst (siehe unten).
- **goto**-Anweisung
Per **goto**-Anweisung

goto-Anweisung



kann eine **case**-Marke innerhalb der **switch**-Anweisung oder eine Sprungmarke

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 = args[0][0];
        int zahl = Convert.ToInt32(args[1]);
        switch (farbe) {
            case 'r':
                Console.WriteLine("Sie sind ein emotionaler Typ.");
                break;
            case 'g':
            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':
                Console.WriteLine("Nehmen Sie nicht Alles so tragisch.");
                break;
            default:
                Console.WriteLine("Offenbar mangelt es Ihnen an Disziplin.");
                break;
        }
    }
}
```

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:

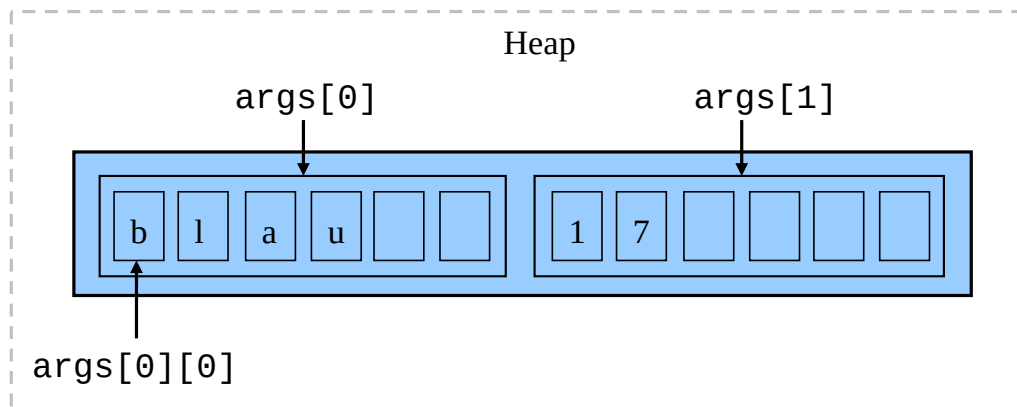
```
PerST b 17
```

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. Die zugehörigen Erklärungen werden Sie mit Leichtigkeit verstehen, sobald Methoden-Parameter sowie Arrays und Zeichenketten behandelt worden sind. An dieser Stelle greifen wir späteren Erläuterungen mal wieder etwas vor (hoffentlich mit motivierendem Effekt):

- Bei einem **Array** handelt es sich um ein Objekt, das eine Serie von Elementen desselben Typs aufnimmt, auf die man per Index, d.h. durch die mit eckigen Klammern begrenzte Elementnummer, zugreifen kann.

- In unserem Beispiel kommt ein Array mit Elementen vom Datentyp **string** zum Einsatz, wobei es sich um Zeichenketten handelt. Literale mit diesem Datentyp sind uns schon öfter begegnet (z.B. "Ha l l o").
- 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**-Elemente 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 und der **char**-Variablen **farbe** zugewiesen wird.
- 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:



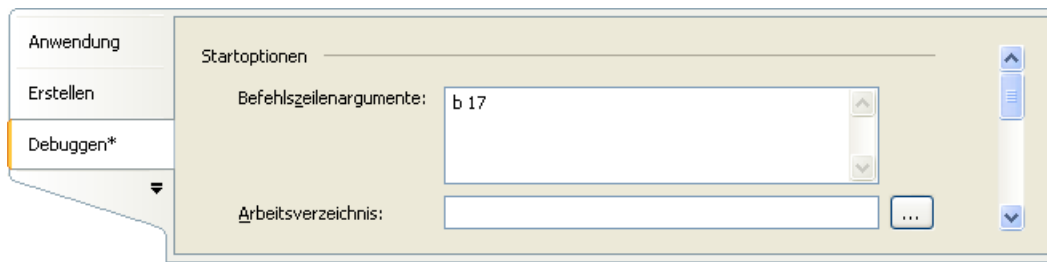
Ansonsten ist im Beispielprogramm noch die **return**-Anweisung von Interesse, welche die **Main()**-Methode und damit das Programm in Abhängigkeit von einer Bedingung sofort beendet:

```
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# 2008 - Umgebung kann man die Befehlszeilenargumente folgendermaßen vereinbaren:

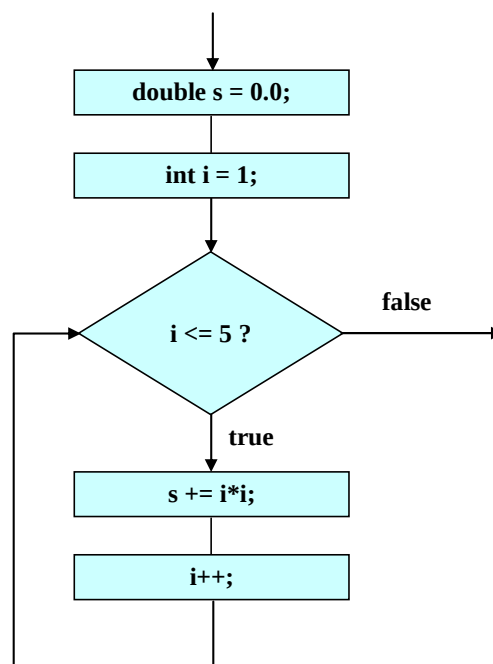
- Menübefehl **Projekt > Eigenschaften**
- Registerkarte **Debuggen**
- **Befehlszeilenargumente** eintragen, z.B.:



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:



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 *nach jedem* Durchlauf 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:

- o **true:** Die zur Schleife gehörige Anweisung wird ein weiteres mal ausgeführt.
- o **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:

- o eine Wertzuweisung, z.B.:
`i = 1`
- o 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>	Quadratsumme = 55

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.

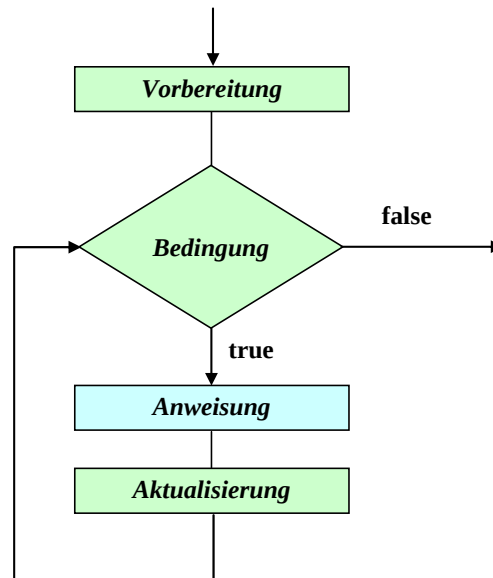
- **Bedingung**

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
<pre>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(); } } }</pre>	<pre>eins e i n s zwei z w e i</pre>

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 *lesen* 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

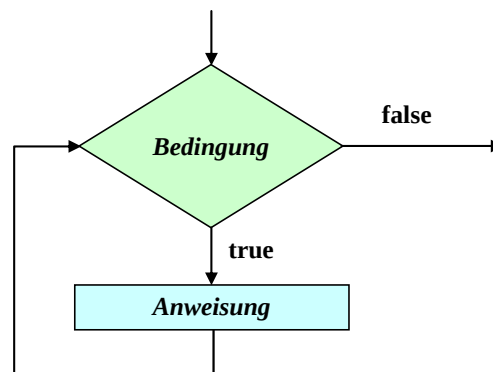
Wie die Erläuterungen zur **for**-Schleife gezeigt haben, ist die Überschrift dieses Abschnitts nicht sehr trennscharf, weil bei der **for**-Schleife ebenfalls eine beliebige Terminierungsbedingung angegeben werden darf. In vielen Fällen ist es eine Frage des persönlichen Geschmacks, welche Wiederholungsanweisung 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:

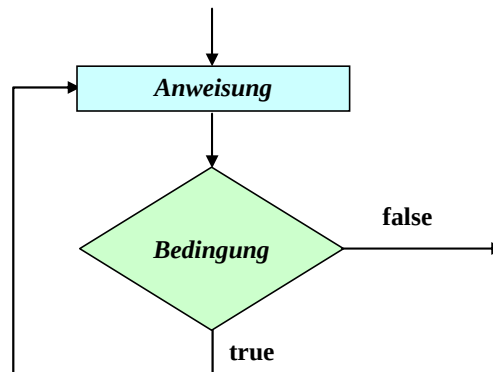


Im obigen Beispielprogramm zur Quadratsummenberechnung (vgl. Abschnitt 3.7.3.1) kann man die **for**-Schleife leicht durch eine **while**-Schleife ersetzen:

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 quitierte 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("Beenden? (j/n)? "); antwort = Console.ReadLine()[0]; } while (antwort != 'j' && antwort != 'n'); } }</pre>	<pre>Beenden? (j/n)? r Beenden? (j/n)? 4 Beenden? (j/n)? j</pre>

Bei einer **do**-Schleife mit Anweisungsblock 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 -= 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 Eingabe erscheint 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 $\leq$ Wurzel):

Sei d (≥ 1) ein echter Teiler der positiven, ganzen Zahl z , d.h. es gibt eine Zahl k (≥ 2) mit

$$z = k \cdot d$$

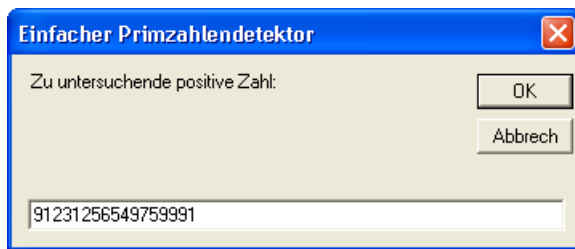
Dann ist auch k ein echter Teiler von z , und es gilt:

$$d \leq \sqrt{z} \quad \text{oder} \quad k \leq \sqrt{z}$$

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.

Mit den bereits im Kurs vorgestellten Techniken und etwas Studium der FCL - Dokumentation ist es übrigens kein großes Problem, eine GUI - Variante des Primzahlendetektors zu erstellen, z.B.:



Hier kommen die Methoden **MessageBox.Show()** (vgl. Abschnitt 2.3) und **Interaction.InputBox()** (vgl. Abschnitt 3.4.2) zum Einsatz. Für den Schönheitsfehler **Abbrechen** (offenbar entstanden durch den höheren Platzbedarf der deutschen Beschriftung im Vergleich zur englischen Variante) ist die Methode **InputBox()** verantwortlich. Das Visual Studio - Projekt mit dem GUI - Primzahlendetektor befindet sich im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\PrimitivGUI

Mit den im weiteren Verlauf des Kurses zu erwerbenden Kenntnissen über die Windows-Programmierung wird man bei der gegebenen Aufgabenstellung allerdings eher mit einer *einzelnen* Dialogbox und geeigneten Steuerelementen arbeiten.

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**?

Quellcode	Ausgabe
<pre>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); } }</pre>	b ist True Kontr.ausg. von b: False

3) Erstellen Sie eine Variante des Primzahlen-Diagnoseprogramms aus Abschnitt 3.7.3.5, die ohne **break** bzw. **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 3.3.3). 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 3.3.4.1 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) zweier natürlicher 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 folgenden Satz der mathematischen Zahlentheorie:

Für zwei natürliche Zahlen u und v (mit $u > v$) ist der ggT gleich dem ggT von u und $u \% v$ (u modulo v).

Begründung (analog zu Abschnitt 1.1.3): Für natürliche Zahlen u und v mit $u > v$ gilt:

x ist gemeinsamer Teiler von u und v

$\Leftrightarrow$

x ist gemeinsamer Teiler von u 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 Pentium 4 mit 3,0 GHz):

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:	203,125 Millisek.	Benöt. Zeit:	0 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 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

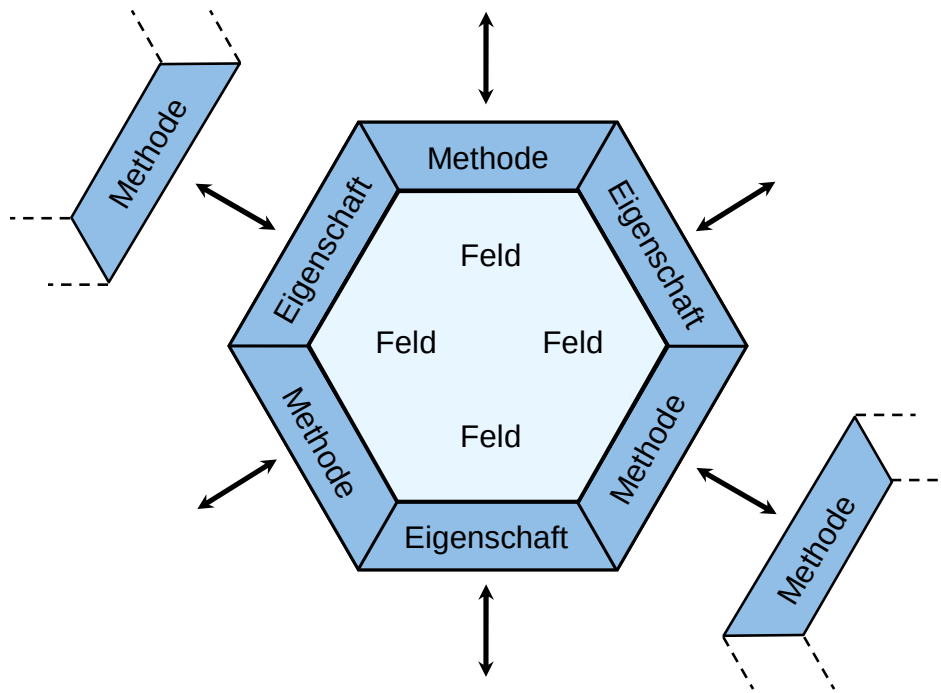
4.1.1 Einige Kernideen und Vorzüge der OOP

Lahres & Rayman (2006) 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 Methoden zum Lesen bzw. Verändern eines Feldes. Eine Klasse wird in der Regel ihre Felder gegenüber anderen Klassen verbergen (**Datenkapselung, information hiding**) und so vor ungeschickten Zugriffen 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. *Öffentliche Felder* einer Klasse gehören zu ihrer Schnittstelle und sollten konstant gesetzt (siehe Abschnitt 3.3.7), also vor Veränderungen geschützt sein (z.B. **Math.PI**).

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.

- **Produktivität**

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-

¹ Deutsche Übersetzung: Teile und herrsche!

Text-Editor über eine Klasse aus der Standardbibliothek integrieren, ohne wissen zu müssen, wie Text und Textauszeichnungen intern verwaltet werden.

- **Flexibilität, Anpassungsfähigkeit, vereinfachte Wartung**
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 Programnteile beliebig geändert werden.
- **Erfolgreiche Teamarbeit durch abgeschottete Verantwortungsbereiche**
In großen Projekten können mehrere Programmierer nach der gemeinsamen Entwicklung 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 rationaler 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, 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 statische Methoden oder erzeugte Objekte) bietet die OOP mit der Vererbungstechnik eine weitere Möglichkeit zur Wiederverwendung von Software.

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 als 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 **BruchRechnung** wird ein Bruch-Objekt (vgl. Abschnitt 1.1) nach seinem Datentyp befragt:

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { Bruch b = new Bruch(); Console.WriteLine(b.GetType().FullName); } }</pre>	Bruch

Von **GetType()** erhält man als Rückgabewert eine Referenz auf ein Objekt der Klasse **Type**. Über diese Referenz wird das **Type**-Objekt gebeten, den Wert seiner Eigenschaft **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).

Methodenaufruf, gerichtet an die Klasse **Console** Referenz auf ein **Bruch**-Objekt Eigenschaftszugriff, gerichtet an das von **GetType()** gelieferte **Type**-Objekt, liefert ein **String**-Objekt

`Console.WriteLine(b.GetType().FullName);`

Methodenaufruf, gerichtet an das **Bruch**-Objekt **b**, mit Referenz auf **Type**-Objekt als Rückgabewert

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 **ToString()**-Methode 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: Die Objekte der Klasse **Bruch** 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 **Abbildung 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, Ausnahmefehler).

4.1.2 Strukturierte Programmierung und OOP

In vielen klassischen Programmiersprachen (z.B. C, Fortran oder 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 fremder Quelle 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 definierten Typ speichern. Dies vereinfacht z.B. das Lesen, Kopieren oder Schreiben solcher Daten.

Die problemadäquate 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 der Abschnitte 2 und 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()** zusätzlich 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 Aufrufer über eine reibungslose oder verpatzte Ausführung informieren kann, wechselt der Rückgabetypp von **void** zu **bool**. Mit der **return**-Anweisung wird am Ende 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();
        Console.ReadLine();
    }
}

```

In der **Main()**-Methode dieses Programms zum Kürzen von Brüchen wird ein Objekt aus der Klasse **Bruch** erzeugt und mit Aufträgen versorgt.

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).

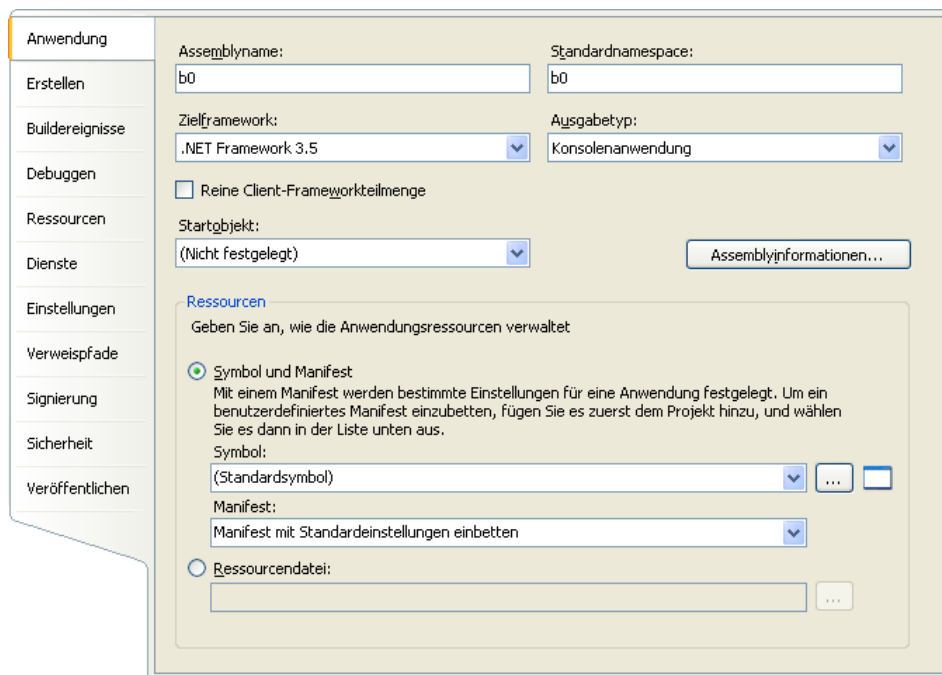
Hinsichtlich der Dateiverwaltung wird vorgeschlagen:

- Die Klassendefinitionen (z.B. bei **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# 2008 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 folgende Instanzvariablen:

- **zaehler** (Datentyp **int**)
- **nenner** (Datentyp **int**)
- **etikett** (Datentyp **string**)

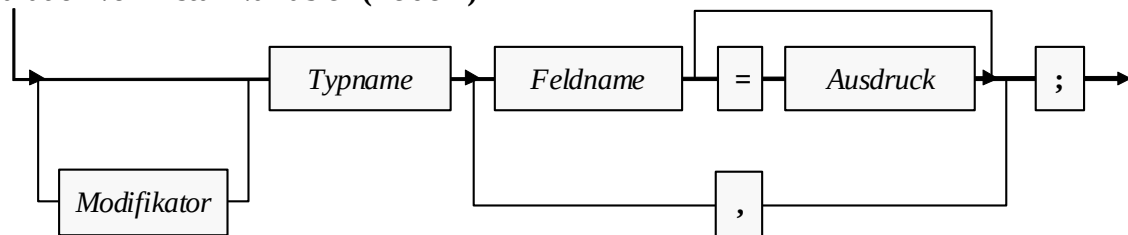
Zu den beiden Feldern **zaehler** und **nenner** vom elementaren Datentyp **int** ist das Feld **etikett** mit dem Referenzdatentyp **string** (ein Alias für den Klassennamen **String**) dazugekommen. Jedes nach dem **Bruch**-Bauplan geschaffene Objekt erhält seine *eigene, komplette* Ausstattung mit diesen Variablen.

4.2.1 Deklaration mit Wahl der Schutzstufe

Während lokale Variablen im Anweisungsteil einer Methode deklariert werden, erscheinen die Deklarationen der Instanzvariablen in der Klassendefinition *außerhalb* jeder Methodendefinition. 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) werden in die Eigenschaften `Zaehler` und `Nenner` neben der Null-Überwachung für den Nenner eventuell noch weitere Wertrestriktionen einbauen (z.B. Beschränkung auf positive Zahlen). Während solche Anpassungen von Zugriffsmethoden problemlos möglich sind, ist beim Ableiten eine Einschränkung von Zugriffsrechten verboten. Für eine als **public** deklarierte Instanzvariable `zaehler` wären also auch in abgeleiteten Klassen keine Wertrestriktionen möglich.
- Oft kann der Datentyp von gekapselten Instanzvariablen geändert werden, ohne dass die Schnittstelle inkompatibel zu vorhandenen Programmen wird, welche die Klasse benutzen. Im Fall der Bruch-Felder `zaehler` und `nenner` wird man vielleicht zur Vermeidung von Überlaufproblemen (vgl. Abschnitt 3.6.1) irgendwann den Datentyp **int** durch den größeren Typ **long** ersetzen. Davon würde z.B. die Methode `Addiere()` profitieren:

```

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

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 besser von lokalen Variablen und Formalparametern unterscheiden zu können, verwenden manche Programmierer ein Präfix, z.B.:

```
int m_nenner; //das m steht für "Member"
int _nenner;
```

4.2.2 Konstante und schreibgeschützte Instanzvariablen

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 3.3.7) 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 praktische Möglichkeit, um fremden Klassen ohne Risiko einen syntaktisch einfachen *Lesezugriff* zu gestatten. 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, und der Modifikator **static** kann *nicht* zusammen mit **const** verwendet werden. 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.:

Quellcode	Ausgabe
<pre>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); } }</pre>	7

Bei den Konstruktoren handelt es sich um spezielle Methoden, die den Namen ihrer Klasse tragen und keinen Rückgabetyt besitzen (auch nicht **void**). Im Beispiel verwendet der Konstruktor zur Initialisierung der schreibgeschützten Instanzvariablen ein Befehlszeilenargument.

4.2.3 Sichtbarkeitsbereich, Existenz und Ablage im Hauptspeicher

Von den lokalen Variablen einer Methode unterscheiden sich die Instanzvariablen (Felder) einer Klasse vor allem in Bezug auf den Sichtbarkeitsbereich, die Lebensdauer und die Ablage im Hauptspeicher:

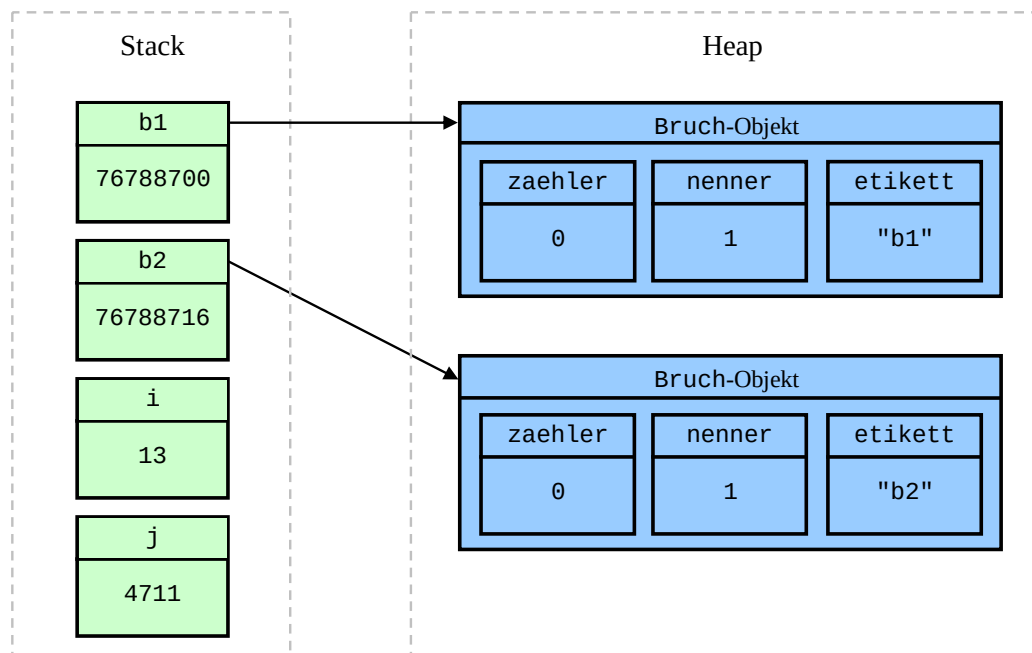
	lokale Variable	Instanzvariable
Sichtbarkeit	Eine lokale Variable ist nur in ihrer eigenen Methode sichtbar. Nach der Deklarationsanweisung kann sie in den restlichen Anweisungen des lokalsten Blocks angesprochen werden. Eingeschachtelte Blöcke gehören zum Sichtbarkeitsbereich. In aufgerufenen Methoden ist die lokale Variable nicht sichtbar.	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 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: <i>Stapel</i>) gespeichert. 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: <i>Haufen</i>) 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.



4.2.4 Initialisierung

Während bei lokalen Variablen der Programmierer für die Initialisierung verantwortlich ist, erhalten die Instanzvariablen eines neuen Objektes 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*. 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.

4.2.5 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 möglich

In diesem Abschnitt wurden eine Syntaxregel und ein Designprinzip dargestellt:

- 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.

4.3 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 Rückgabebetyp **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
```

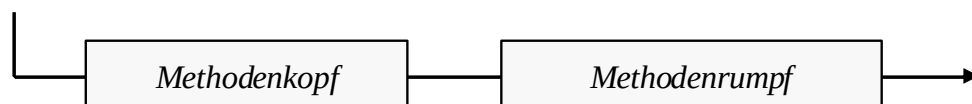
Über und unter dem Etikett steht das **string**-Objekt **luecke**, das als leere Zeichenfolge initialisiert wird. In der **for**-Schleife werden per Plusoperator Leerzeichen angehängt, bis die Länge des Etiketts erreicht ist. Diese Länge wird über die Eigenschaft **Length** des **string**-Objekts **etikett** ermittelt.

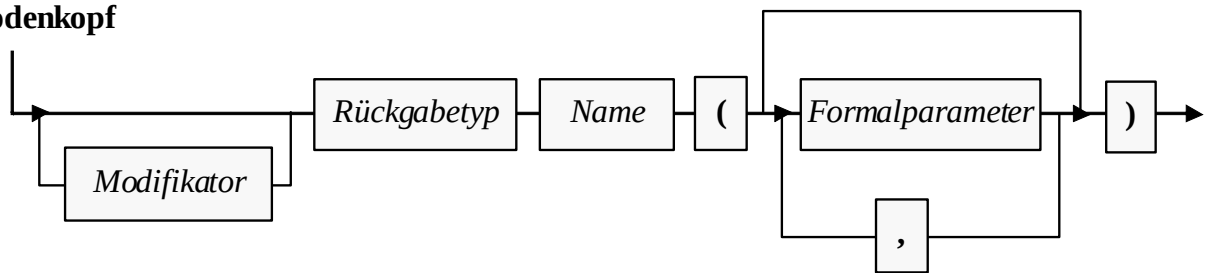
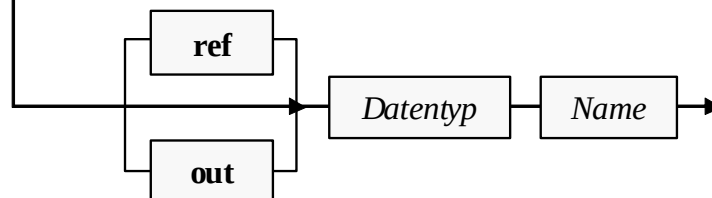
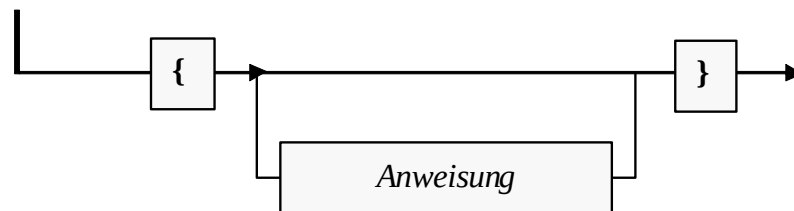
Während jedes Objekt einer Klasse seine eigenen Instanzvariablen auf dem Heap besitzt, ist der MSIL-Code der Instanzmethoden jeweils nur *einmal* im Speicher vorhanden und wird von allen Objekten verwendet.

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:

Methodendefinition



Methodenkopf**Formalparameter****Methodenrumpf**

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 4.2.1), sind die Empfehlungen für Methodennamen weniger einheitlich. Auf der MSDN-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, EventArgs 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 diese Voreinstellung 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.

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**):

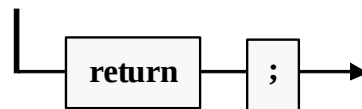
```
public bool Frage() {
    try {
        Console.WriteLine("Zaehler: ");
        int z = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Nenner : ");
        int n = Convert.ToInt32(Console.ReadLine());
        Zaehler = z;
        Nenner = n;
        return true;
    } catch {
        return false;
    }
}
```

Ist der Rückgabety 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, irreguläre Benutzereingaben 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 Methodendefinition werden über so genannte **Formalparameter** Daten von bestimmtem Typ spezifiziert, die entweder den Ablauf der Methode steuern, oder durch die Methode verändert werden. Beim späteren **Aufruf** der Methode sind korrespondierende **Aktualparameter** anzugeben (siehe Abschnitt 4.3.2), wobei je nach Parameterart Variablen oder Ausdrücke in Frage kommen.

In den Anweisungen des Methodenrumpfs sind die Formalparameter wie lokale Variablen zu verwenden, die teilweise (je nach Parameterart, siehe unten) 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 ein kleiner Anfangsbuchstabe 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 via Parameterliste (`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
<pre>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); } }</pre>	<pre>4712 4711</pre>

Die Demoklasse `Prog` ist startfähig, besitzt also eine Methode **Main()**. Dort wird ein Projekt 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 Rückgabetyt 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 Vari-

ablen, 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. Eingabeparametern 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	Ausgabe
<pre>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); } }</pre>	<pre>Vorher: x = 1, y = 2 Nachher: x = 2, y = 1</pre>

Abschließend noch ein Satz für Begriffsakrobaten: Wird eine *Referenzvariable*, die auf ein Objekt zeigt, 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 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.

4.3.1.3.3 Ausgabeparameter

Auch über Ausgabeparameter kann man einer Methode die Veränderung von Variablen der rufenden Programmeinheit ermöglichen.

Als Ausgabeparameter sind nur *Variablen* erlaubt, die vom *selben* Typ wie der Formalparameter und außerdem initialisiert 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	Ausgabe
<pre>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)); } }</pre>	<pre>x = 29 y = 5 x % y = 4</pre>

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	Ausgabe
<pre>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); } }</pre>	<pre>Die Summe ist = 2,2 Die Summe ist = 5,8</pre>

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.

4.3.2 Methodenaufruf und Aktualparameter

Beim Aufruf einer Instanzmethode, z.B.:

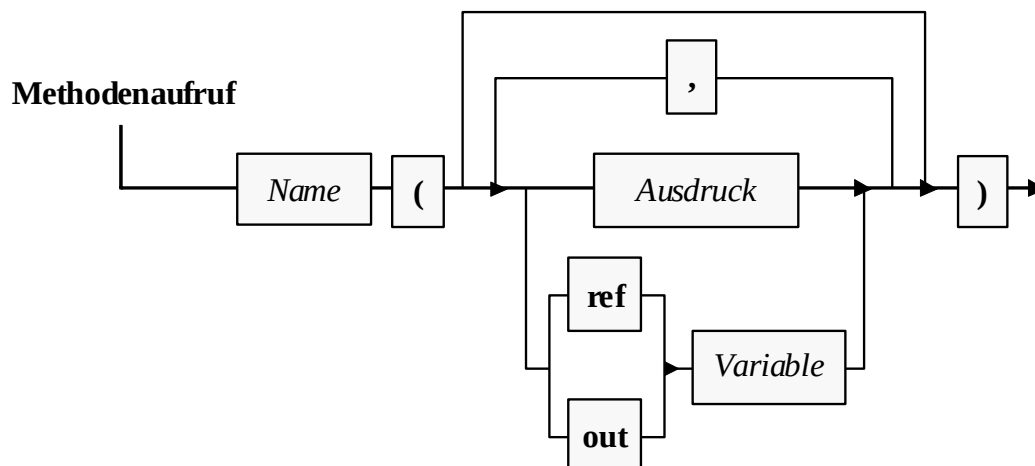
```
b1.Zeige();
```

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.

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.:

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).

```
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 aktuelle Objekt ansprechen, z.B.:

```
this.Kuerze();
```

4.3.3 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. Man spricht hier von einer *Methodenüberladung*.

Eine Überladung ist erlaubt, wenn sich die *Signaturen* der beteiligten Methoden unterscheiden. Zwei Methoden besitzen genau dann *dieselbe* Signatur, wenn die beiden folgenden Bedingungen erfüllt sind:¹

- Die Namen sind identisch.
- Die Parameterlisten sind gleich lang, und die Typen korrespondierender Parameter stimmen überein.

Für die Signatur ist der Rückgabotyp 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 wohl daraus, dass der Rückgabewert einer Methode in Anweisungen oft keine Rolle spielt (ignoriert wird).

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:

¹ Bei den später zu behandelnden *generischen* Methoden muss die Liste mit den Kriterien für die Identität von Signaturen erweitert werden.

- Für verschiedene Datentypen (z.B. **double** und **int**) werden analog arbeitende Methoden benötigt.

So besitzt z.B. die Klasse **Math** im Namensraum **System** u.a. folgende Methoden, um den Betrag einer Zahl zu berechnen:

```
public static 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, z.B.

```
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 in C# - Programmen folgende Datentypen zur Verfügung (vgl. Abschnitt 3.3.1):

- 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.

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:

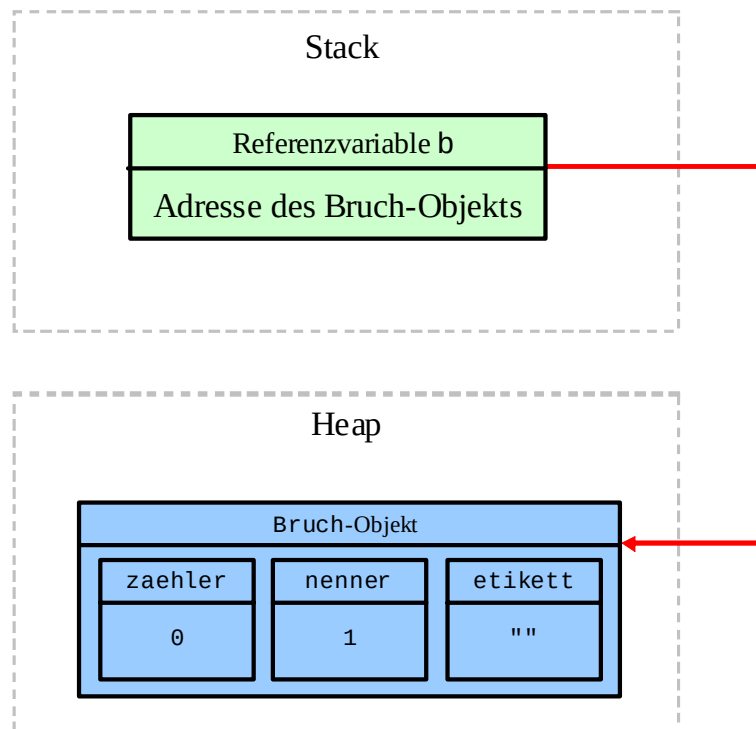
```
new Bruch()
```

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. Als Wert des Ausdrucks resultiert eine Referenz (Speicheradresse), 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.

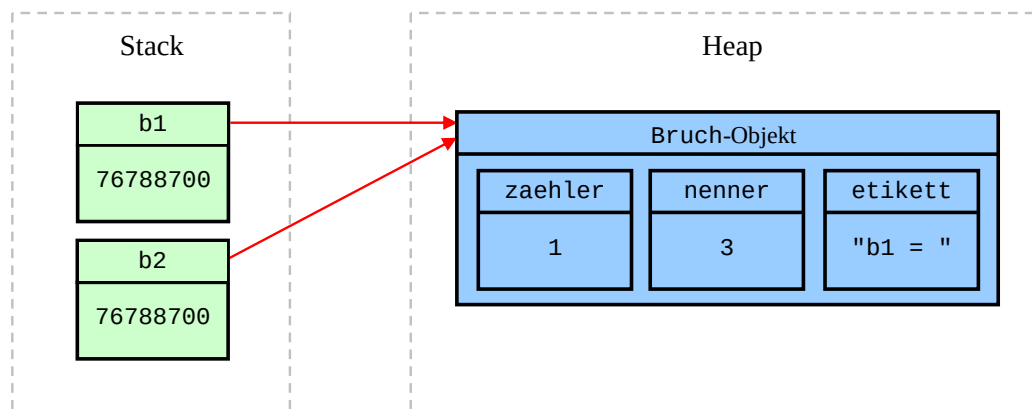
In einem Programm können *mehrere* Referenzvariablen auf *dasselbe* Objekt zeigen, z.B.:

Quellcode	Ausgabe
<pre>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(); } }</pre>	<pre> 1 b2 = ---- 3</pre>

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.

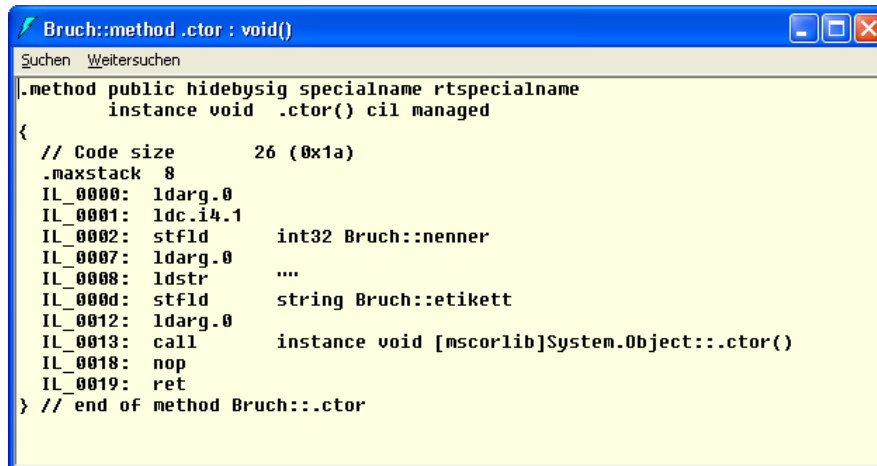
4.4.3 Objekte initialisieren über Konstruktoren

In diesem Abschnitt werden spezielle Methoden behandelt, die beim Erzeugen von neuen Objekten automatisch aufgerufen werden, um deren Instanzvariablen zu initialisieren. 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 **Standard-konstruktor** 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 enthaltenen Initialisierungen von Instanzvariablen ein:



```

Bruch::method .ctor : void()
{
    // 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 4.2.4) ist hingegen kein MSIL-Code erforderlich. Zur Anzeige des MSIL-Codes wird hier das im Windows-SDK enthaltene und mit der Visual C# 2008 Express Edition automatisch installierte Hilfsprogramm **ILDasm** verwendet¹.

Am Ende (!) des MSIL-Codes zum Standardkonstruktor wird der parameterlose Konstruktor der Basisklasse aufgerufen, wobei unsere Klasse **Bruch** direkt von der Urachnklasse **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 parameterfreier 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 Konstrukteure möglich, die alle denselben Namen und jeweils eine individuelle Parameterliste haben müssen. Das Überladen von Methoden (vgl. Abschnitt 4.3.3) ist also auch bei Konstruktoren erlaubt.

¹ Bei der Installation der Visual C# 2008 Express Edition landet das Hilfsprogramm im Ordner
%ProgramFiles%\Microsoft SDKs\Windows\v6.0A\bin

- Während der Standardkonstruktor die Schutzstufe **public** besitzt, haben explizite Konstruktor haben wie gewöhnliche Methoden die voreingestellte Schutzstufe **private**. Wenn sie für beliebige 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;
}
```

Wenn weiterhin auch ein parameterfreier Konstruktor verfügbar sein soll, muss dieser explizit definiert werden, z.B. mit leerem Anweisungsteil:

```
public Bruch() {}
```

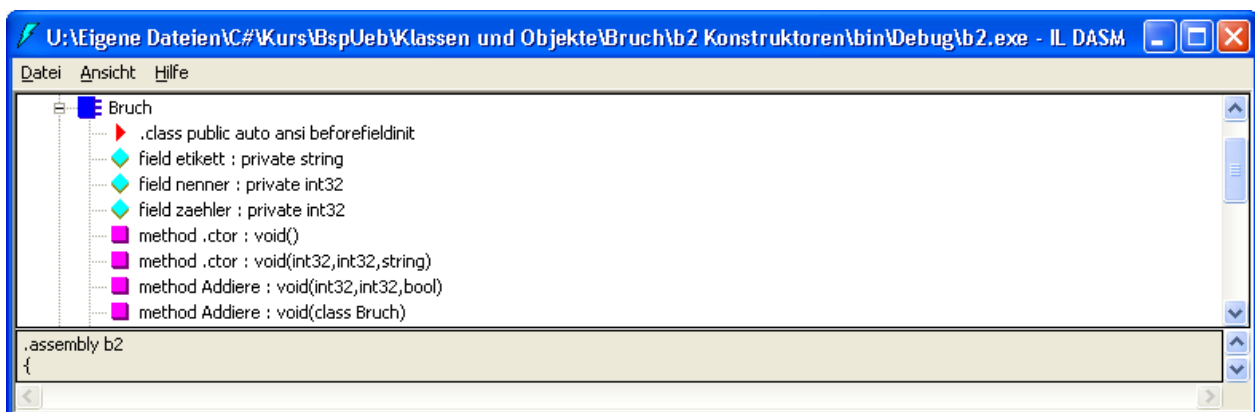
Im folgenden Programm werden beide Konstruktoren eingesetzt:

Quellcode	Ausgabe
<pre>using System; class BruchRechnung { static void Main() { Bruch b1 = new Bruch(1, 2, "b1 = "); Bruch b2 = new Bruch(); b1.Zeige(); b2.Zeige(); } }</pre>	<pre> 1 b1 = ---- 2 0 ---- 1</pre>

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 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.:

Quellcode	Ausgabe
<pre>using System; class CT { public int i; } class Prog { static void Main() { CT ct = new CT() {i = 13}; Console.WriteLine(ct.i); Console.ReadLine(); } }</pre>	13

Die neue Option wurde unter der Bezeichnung *Objektinitialisierer* zur Unterstützung der LINQ-Technik (*Language Integrated Query*) eingeführt (siehe Abschnitt 27.1.2). Bei Verwendung eines Objektinitialisierers darf eine leere Parameterliste weggelassen werden, z.B.:

```
CT ct = new CT {i = 13};
```

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 automatisch 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 Programmierneinsteiger 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, z.B. Ressourcen frei geben, die nicht von der CLR verwaltet werden, dann sind spezielle Methoden (so genannte *Destruktoren*) zu definieren (siehe unten).

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. Bei Wertparametern von *elementarem Datentyp* wirken sich Änderungen innerhalb einer Methode *nicht* auf die rufende Programmeinheit aus. Bei einem Wertparameter mit *Referenztyp* wird ebenfalls der Wert des Aktualparameters (eine Objektreferenz) beim Methodenaufruf in den Formalparameter kopiert. Es wird jedoch keinesfalls eine Kopie des referenzierten Objekts (auf dem Heap) erstellt, so dass Formal- und Aktualparameter auf *dasselbe* Objekt zeigen¹.

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`-Objektes 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:

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 werden können wie Klassentypen.

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>

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:

Quellcode	Ausgabe
<pre>using System; class BruchRechnung { static void Main() { Bruch b1 = new Bruch(1, 2, "b1 = "); b1.Zeige(); Bruch b2 = b1.Klone(); b2.Zeige(); } }</pre>	<pre> 1 b1 = ---- 2 1 b1 = ---- 2</pre>

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

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.

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.

4.5 Eigenschaften

Sollen fremde Klassen Lese- und/oder Schreibzugriff auf ein gekapseltes (also `private`s) 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, das Verwenden einer Klasse zu vereinfachen. Aus den obigen Methodendefinitionen wird die folgende Eigenschaftsdefinition:

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

Für den Klassendesigner ändert sich nicht allzu viel: Es ist ein **get**- und ein **set**-Block mit nahe liegender Syntax zu definieren. Bemerkenswert ist, dass im **set**-Block der vom Aufrufer übergebene neue Wert ohne Formalparameterdefinition über das Schlüsselwort **value** angesprochen wird.

Für den Klassenanwender sind Eigenschaften weit intuitiver zu verwenden als korrespondierende Methodenaufrufe, z.B.:

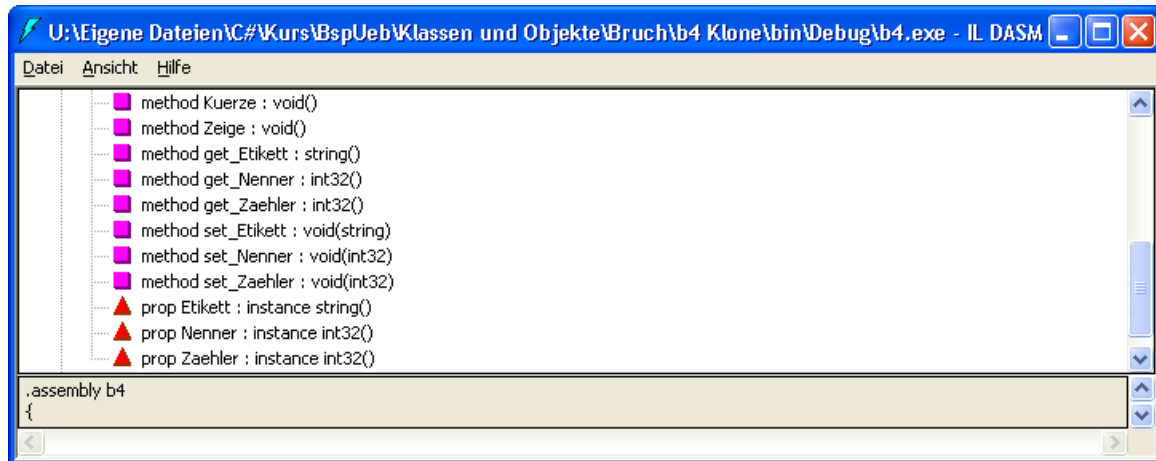
```
b1.Nenner = 2;
Console.WriteLine(b1.Nenner);
```

Sogar die Aktualisierungs-Operatoren 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:



Diese besitzen neben ihrer speziellen (und sehr begrenzten) Aufgabe eine weitere Besonderheit gegenüber den bisher besprochenen Methoden: Der JIT-Compiler der CLR fügt den (meist sehr kleinen) Maschinencode an jeder Aufrufstelle ein, 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*. Nach Richter (2006, S. 245) ist das Inlining im Debug-Modus zur Erleichterung der Fehlersuche abgeschaltet.

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);
    }
}
```

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** aufgefordert, 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 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 dazu dienen, die Anzahl der bisher erzeugten Bruch-Objekte aufzunehmen:

```
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 4.2.4). 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 <i>einmal</i> 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:
 - o Werttypen (z.B. **int**, **double**, **bool**)
 - o 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 des Programms 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. Sofern Klassenmethoden vorhanden sind, kann man auch eine Klasse als *Akteur* auf der objektorientierten Bühne betrachten.

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 Beispielpogramm 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 1.2.6).

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:

Quellcode	Eingabe (fett) und Ausgabe
<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>	<pre> Zaehler: 26 Nenner : 39 Benutzerdefiniert: 2 ---- 3 </pre>

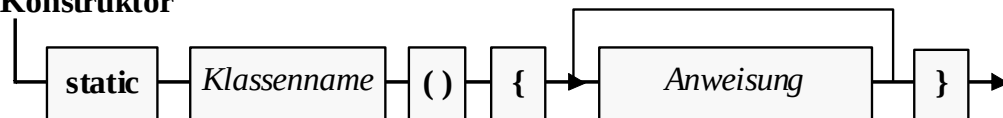
Wird eine Klassenmethode von anderen Methoden der *eigenen* Klasse (objekt- oder klassenbezogen) verwendet, muss der Klassenname *nicht* angegeben werden.

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 Konstrukturen definiert.

In einer etwas gekünstelten Erweiterung des **Bruch**-Beispiels soll der parameterfreie Instanzkonstruktor zufallsabhängige, aber pro Programmablauf 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 4.2.1). 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.:

Quellcode	Ausgabe
<pre>using System; class BruchRechnung { static void Main() { Bruch b1 = new Bruch(), b2 = new Bruch(); b1.Zeige(); b2.Zeige(); } }</pre>	<pre>Klasse Bruch geladen 1 ---- 2 1 ---- 2</pre>

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 auch ein 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:

```
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 iterative GGT-Methode `GGti()` kann durch folgende rekursive Variante `GGTr()` ersetzt werden:

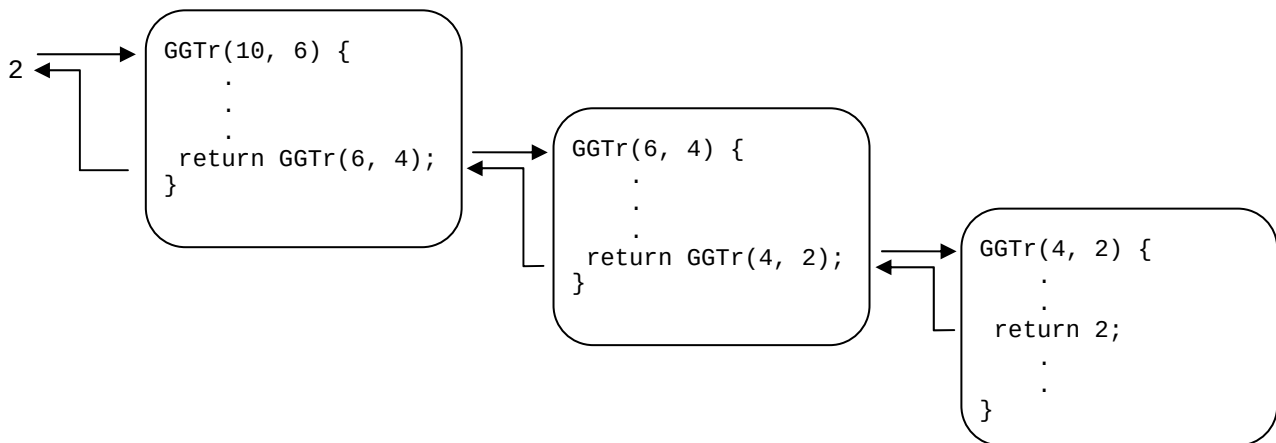
```
int GGTr(int a, int b) {
    int rest = a % b;
    if (rest == 0)
        return b;
    else
        return GGTr(b, rest);
}
```

Statt eine Schleife zu benutzen, arbeitet die rekursive Methode nach folgender Logik:

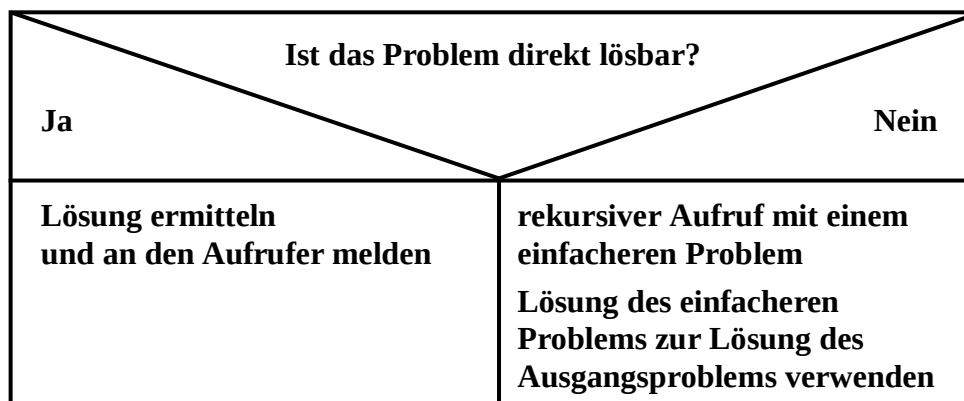
- Ist der Parameter `a` durch den Parameter `b` restfrei teilbar, dann ist `b` der ggT, und der Algorithmus ist beendet:
`return b;`
- Anderenfalls wird das Problem, den ggT von `a` und `b` zu finden, auf das einfachere Problem zurückgeführt, den ggT von `b` und `(a % b)` zu finden, und die Methode `GGTr()` 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, und 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 verketteten 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:

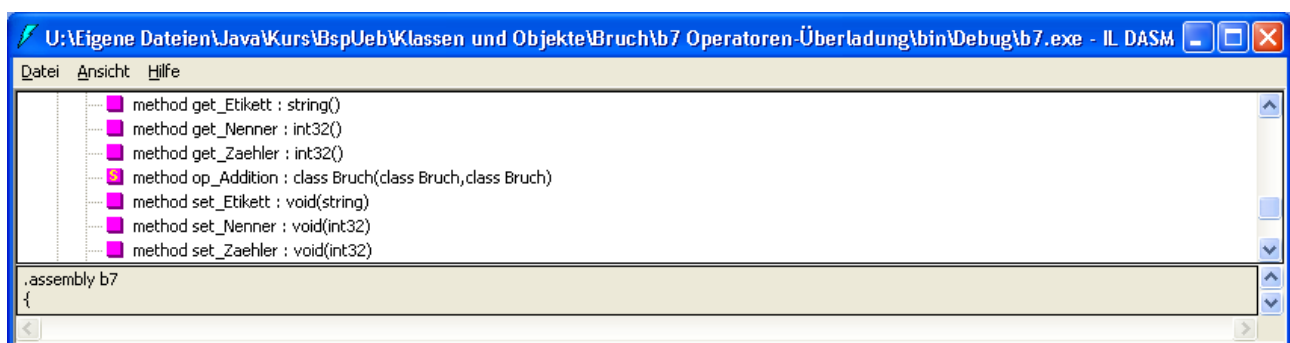
- Es ist grundsätzliche eine *statische* Definition erforderlich.
- Als Namen verwendet man das Schlüsselwort **operator** mit dem jeweiligen Operationszeichen als Suffix.

Nähere Hinweise finden sich z.B. bei Mössenböck (2003, S. 73ff).

Mit dem überladenen „+“ - Operator lassen sich Bruch-Additionen nun sehr übersichtlich formulieren:

Quellcode	Ausgabe
<pre>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(); } }</pre>	<pre>b1 = 1 --- 2 b2 = 1 --- 4 Summe = 3 --- 4</pre>

Wie das Windows-SDK - Hilfsprogramm **ILDasm** zeigt, resultiert aus unserer Operatorenüberladung im Assembly die statische Methode **op_Addition()**:



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 Objekte¹ (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. 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("\nWelchen 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('+', 1, 2, 2, 5);
        auf.Pruefe();
        auf.NeueWerte('*', 3, 4, 2, 3);
        auf.Pruefe();
        Console.ReadLine();
    }
}

```

Man kann immerhin schon ahnen, wie die praxistaugliche Endversion des Programms einmal arbeiten wird:

Berechne bitte:

```

      1          2
-----  +  -----
      2          5

```

```

Welchen Zaehler hat Dein Ergebnis:      3
                                   -----
Welchen Nenner hat Dein Ergebnis:      7

```

```

                                   9
Das korrekte Ergebnis: -----
                                   10

```

Berechne bitte:

$$\frac{3}{4} \quad * \quad \frac{2}{3}$$

Welchen Zaehler hat Dein Ergebnis: 6

Welchen Nenner hat Dein Ergebnis: 12

Gut!

4.8.2 Innere Klassen

Eine Klasse darf neben Feldern, Methoden etc. auch Klassendefinitionen enthalten, wobei *innere 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;

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();
    }

    static void Main() {
        Familie f = new Familie("Müller", "Lea", 7, "Theo", 4);
        f.Info();
        Console.ReadLine();
    }

    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);
        }
    }
}
```

```

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 umgebenden Klasse **Familie** sorgt im Beispiel für folgende Ausgabe:

Die Kinder von Familie Müller:

Ich bin die 7-jährige Tochter Lea von Familie Müller.
 Ich bin der 4-jährige Sohn Theo 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 (**f.name** ist ein privates Feld 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**.

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
<i>ohne</i> 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 etc.) unterstützt C# folgende Schutzstufen (siehe ECMA 2006, S. 31):

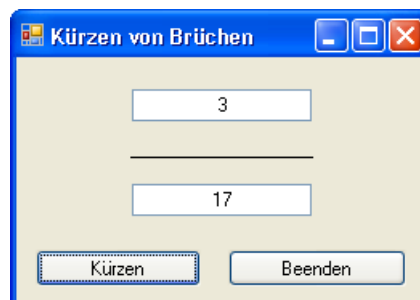
Modifikator(en)	Der Zugriff ist erlaubt für ...				
	eigene Klasse (Abk. K) u. innere Klassen	nicht von K abgel. Klassen im eigenen Assembly	von K abgeleitete Klassen		sonstige Klassen
			im eigenen Assembly	in anderen Assemblies	
<i>ohne</i> oder private	ja	nein	nein	nein	nein
internal	ja	ja	ja	nein	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 Delegates).

4.10 Bruchrechnungsprogramm mit WinForms-Benutzerschnittstelle

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 Benutzerschnittstelle:

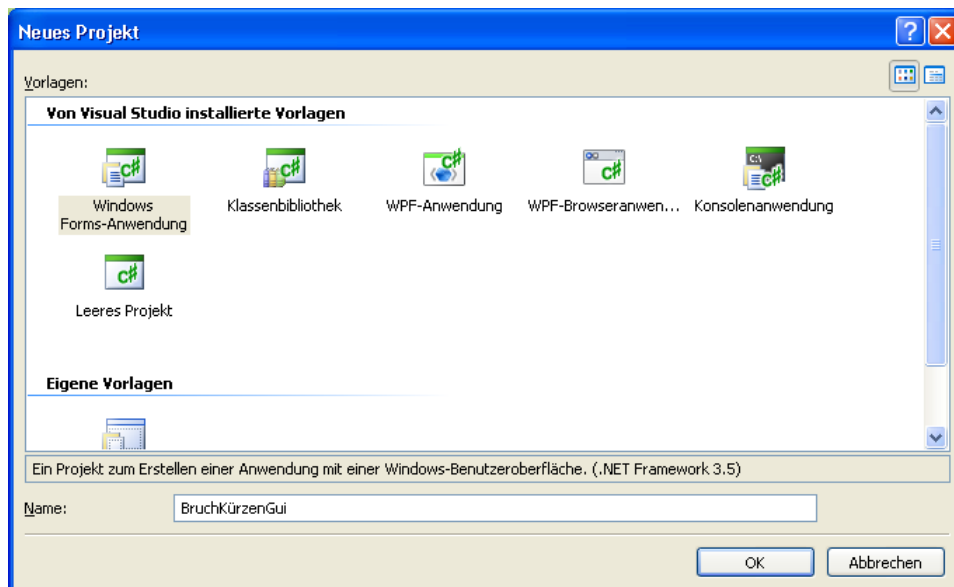


4.10.1 Projekt anlegen mit Vorlage *Windows Forms - Anwendung*

In der folgenden Beschreibung wird die Visual C# 2008 Express Edition verwendet, doch bestehen keine wesentlichen Bedienungsunterschiede zum Visual Studio 2008 Professional (vgl. Abschnitt 2.2.1.3). Verwenden Sie nach

Datei > Neu > Projekt

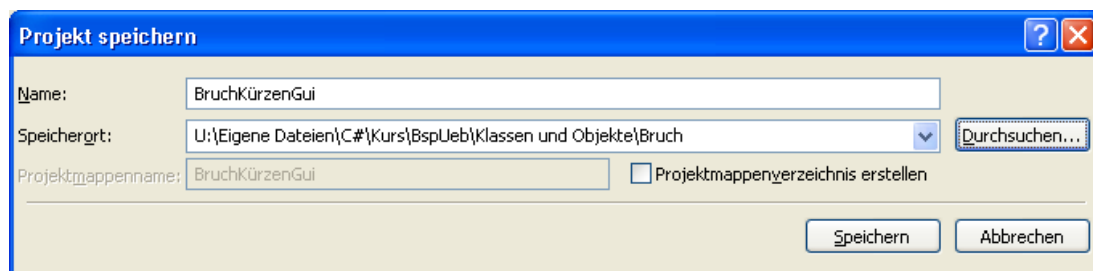
für ein neues Projekt mit dem Namen **BruchKürzenGui** die Vorlage **Windows Forms - Anwendung**:



Nach einem Mausklick auf **OK** präsentiert die Entwicklungsumgebung im Formulardesigner einen Rohling für das Hauptfenster der entstehenden Anwendung. Speichern Sie das Projekt über

Datei > Alle speichern

in einem Ordner Ihrer Wahl (festgelegt durch Namen und Basisverzeichnis), z.B.:



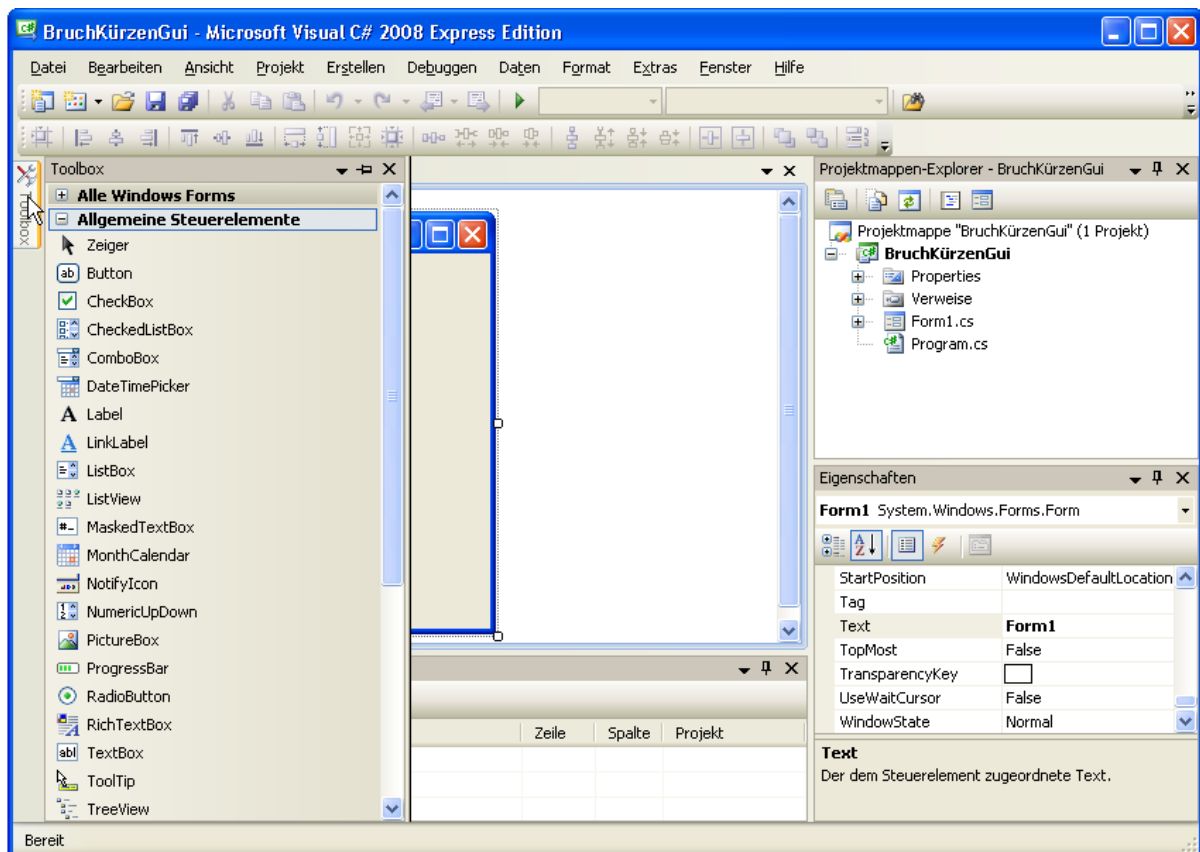
Durch den Verzicht auf ein **Projektmappenverzeichnis** wählen wir eine „flache“ Projektdatenverwaltung ohne Zusammenfassung von mehreren Projekten zu einer Mappe.

4.10.2 Steuerelemente aus der Toolbox übernehmen

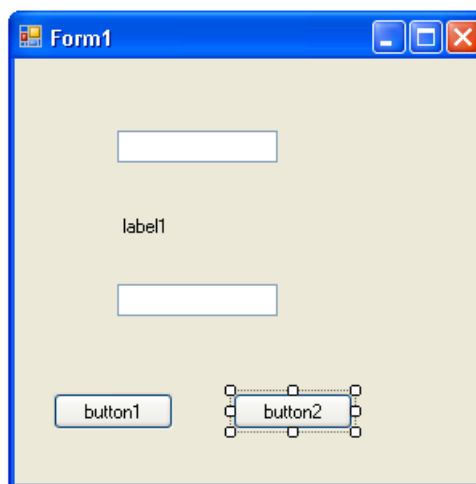
Öffnen Sie das **Toolbox**-Fenster mit dem Menübefehl

Ansicht > Toolbox

oder per Mauszeiger durch kurzes Verharren auf der **Toolbox**-Schaltfläche am linken Fensterrand:

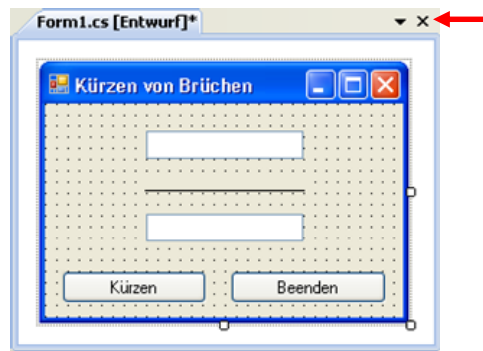


Erweitern Sie im **Toolbox**-Fenster die Liste mit den **allgemeinen Steuerelementen**, und ziehen Sie per Maus zwei **TextBox**-Objekte, ein **Label**-Objekt sowie zwei **Button**-Objekte auf das Formular, z.B.:



4.10.3 Steuerelemente gestalten

Nun können Sie wie in einem Grafikprogramm die Positionen und Größen der Fensterbestandteile verändern, um das gewünschte Layout zu erzielen, z.B.:



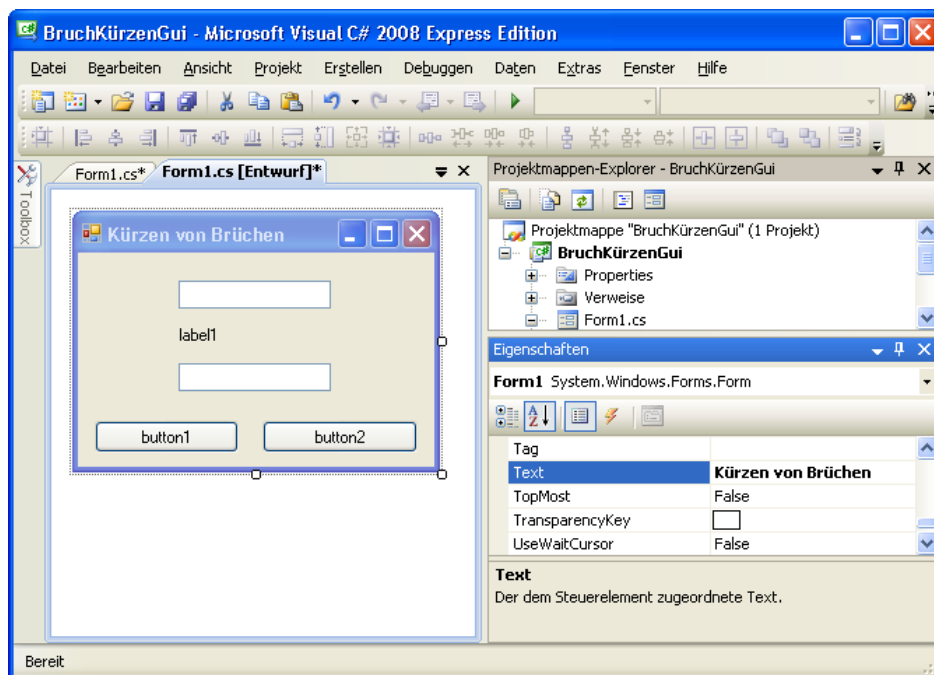
Einstellungsänderungen wirken sich auf ein in Bearbeitung befindliches Formular *nicht* aus. Ggf. müssen Sie also das Fenster des Formulardesigners schließen (siehe Pfeil in der letzten Abbildung) und dann per Doppelklick auf den Eintrag **Form1.cs** im Projektmappen-Explorer wieder öffnen.

4.10.4 Eigenschaften der Steuerelemente ändern

Im Eigenschaftsfenster der Entwicklungsumgebung, das bei Bedarf mit dem Menübefehl

Ansicht > Eigenschaftsfenster

zu öffnen ist, lassen sich diverse Eigenschaften (im Sinne von Abschnitt 4.5!) des markierten Formularobjekts durch direkte Werteingabe festlegen, z.B. der **Text** in der Titelzeile des Formulars:



Ändern Sie analog ...

- die **Text** - Eigenschaft der Schaltflächen, so dass sinnvolle Beschriftungen entstehen
- die **Text** - Eigenschaft des Labels, um einen behelfsmäßigen Bruchstrich einzuzeichnen
- für beide Textboxen die Eigenschaft **TextAlign** auf den Wert **Center**
- die **AcceptButton**-Eigenschaft des Formulars auf den Instanzvariablenamen des Befehlsschalters zum Kürzen (per Voreinstellung: `button1`), so dass dieser per **Enter**-Taste angesprochen werden kann

Man kann das aktive Objekt auch über eine Liste im Kopfbereich des Eigenschaftsfensters wählen. Außerdem lassen sich die Eigenschaften von mehreren gleichzeitig markierten Objekten in einem Arbeitsgang ändern.

4.10.5 Der Quellcode-Generator

Aufgrund Ihrer kreativen Tätigkeit erzeugt die Entwicklungsumgebung im Hintergrund Quellcode zu einer neuen Klasse namens **Form1**, die von der Klasse **Form** im Namensraum **System.Windows.Forms** abgeleitet wird (siehe unten). 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 **Form1** auf zwei Dateien verteilt:

- **Form1.cs**
Hier werden die von Ihnen erstellten Methoden landen (siehe unten).
- **Form1.Designer.cs**
Hier landet der automatisch generierte Quellcode, und Sie sollten in dieser Datei keine Änderungen vornehmen, um den Formulardesigner nicht aus dem Tritt zu bringen.

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.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace BruchKürzenGui {
    public partial class Form1 : Form {
        public Form1() {
            InitializeComponent();
        }
    }
}
```

Der **Form1**-Quellcode in der Datei **Form1.cs** enthält u.a. einen Konstruktor, welcher die private Methode **InitializeComponent()** aufruft. Diese wird in der Datei **Form1.Designer.cs** vom Formulardesigner implementiert.

Um eine ungefähre Vorstellung vom Wirken des Code-Generators zu erhalten, öffnen wir die Quelldatei **Form1.Designer.cs** (ohne Änderungsabsicht) per Doppelklick auf ihren Eintrag im Projektmappen-Explorer. Hier finden sich u.a. die Deklarationen der Instanzvariablen:

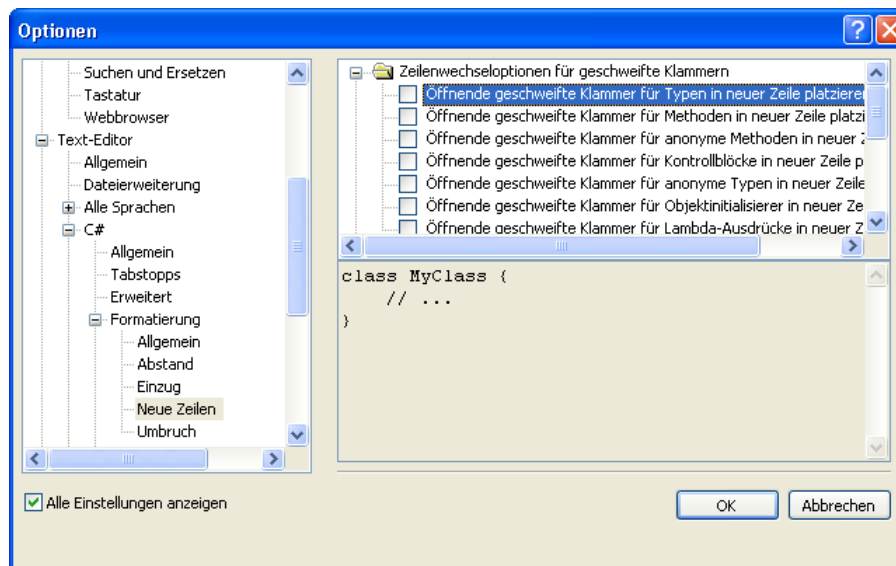
```
private System.Windows.Forms.TextBox textBox1;
private System.Windows.Forms.TextBox textBox2;
private System.Windows.Forms.Label label1;
private System.Windows.Forms.Button button1;
private System.Windows.Forms.Button button2;
```

Aufgrund unserer Tätigkeit im Formulardesigner enthält die Formularklasse **Form1** im Sinne der in Abschnitt 4.8.1 behandelten Aggregation mehrere Objekte anderer Klassen, die Steuerelemente der graphischen Benutzeroberfläche repräsentieren. Obwohl die Schutzstufe **private** für Klassen-Member voreingestellt ist, verwendet die Entwicklungsumgebung der Deutlichkeit halber den Modifikator bei jeder Deklaration. Eine weitergehende Analyse des automatisch erstellten Quellcodes sparen wir uns an dieser Stelle.

Der Quellcode-Generator setzt abweichend von der im Manuskript üblichen Praxis eröffnende geschweifte Klammern (z.B. bei Klassen- oder Methodendefinitionen) in eine eigene Zeile. Das alternative Verhalten lässt nach dem Menübefehl

Extras > Optionen

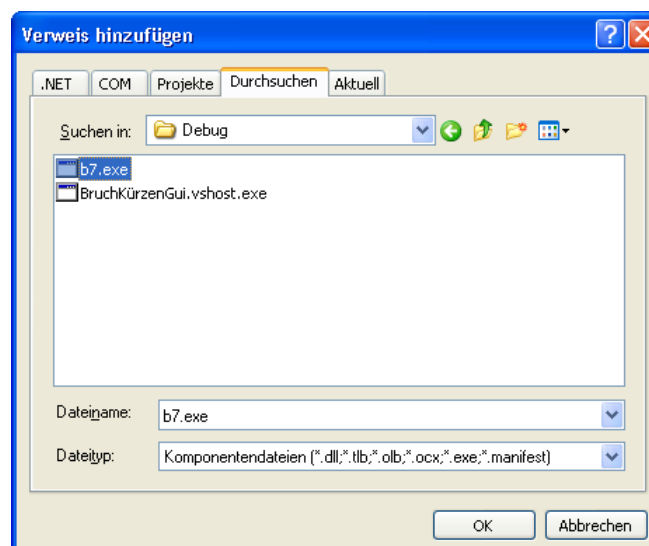
so einstellen:



4.10.6 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 Debugmodus-Ausgabeordner Ordner des neuen Projekts, also z.B. nach
 U:\\Eigene Dateien\\C#\\Kurs\\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 1.2.4.4), 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 Ent-

wicklungsumgebungen definieren grundsätzlich einen Namensraum (abgesehen von der *leeren* Projektvorlage), so auch im aktuellen Beispiel:

```
namespace BruchKürzenGui {
    . . .
}
```

4.10.7 Ereignisbehandlungsmethoden anlegen

Nun erstellen wir zu jedem Befehlsschalter 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 **Text**), so dass die Entwicklungsumgebung in **Form1.cs** die Instanzmethode `button1_Click()` mit leerem Rumpf anlegt

```
private void button1_Click(object sender, EventArgs e) {

}
```

und die Quellcodedatei in einem Editorfenster öffnet.

Mit Hilfe eines Objekts aus unserer Klasse **Bruch** ist die benötigte Funktionalität leicht zu implementieren, z.B.:

```
private void button1_Click(object sender, EventArgs 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", MessageBoxButtons.OK,
                        MessageBoxIcon.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 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) **MessageBoxButtons** bzw. **MessageBoxIcon** als Aktualparameter für die gewünschte Ausstattung der Meldungsdialogbox. 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**).

Erstellen Sie nun per Doppelklick auf den Befehlsschalter `button2` (mit dem Wert *Beenden* für die Eigenschaft **Text**) den Rohling für seine Klick-Ereignisbehandlungsmethode, und ergänzen Sie einen Aufruf der statischen Methode **Exit()** der Klasse **Environment**, die zur Beendigung des Programms führt, z.B.:

```
private void button1_Click(object sender, EventArgs e) {
    Environment.Exit(0);
}
```


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**. 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 Rückgabotyp **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 kompatibel (erweiternd konvertierbarem Typ).
6. Ändert man den Rückgabotyp 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 verfügt:

- **Länge** ermitteln

Der Betrag eines Vektors $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ ist definiert durch:

$$|x| := \sqrt{x_1^2 + x_2^2}$$

Verwenden Sie die Klassenmethode **Math.Sqrt()**, um die Quadratwurzel aus einer **double**-Zahl zu berechnen.

- Vektor auf Länge Eins **normieren**

Dazu dividiert man beide Komponenten durch die Länge des Vektors, denn mit

$\tilde{x} := (\tilde{x}_1, \tilde{x}_2)$ sowie $\tilde{x}_1 := \frac{x_1}{\sqrt{x_1^2 + x_2^2}}$ und $\tilde{x}_2 := \frac{x_2}{\sqrt{x_1^2 + x_2^2}}$ gilt:

$$|\tilde{x}| := \sqrt{\tilde{x}_1^2 + \tilde{x}_2^2} = \sqrt{\left(\frac{x_1}{\sqrt{x_1^2 + x_2^2}}\right)^2 + \left(\frac{x_2}{\sqrt{x_1^2 + x_2^2}}\right)^2} = \sqrt{\frac{x_1^2}{x_1^2 + x_2^2} + \frac{x_2^2}{x_1^2 + x_2^2}} = 1$$

- Vektoren (komponentenweise) **addieren**

Die Summe der Vektoren $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ und $y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ ist definiert durch:

$$\begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \end{pmatrix}$$

- **Skalarprodukt** zweier Vektoren ermitteln

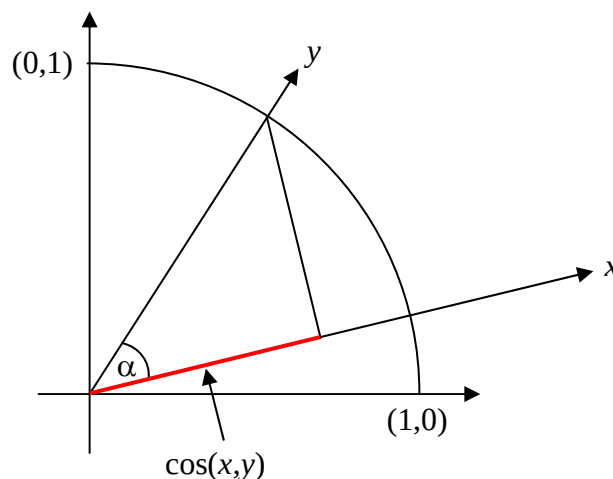
Das Skalarprodukt der Vektoren $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ und $y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ ist definiert durch:

$$x \cdot y := x_1 y_1 + x_2 y_2$$

- **Winkel** zwischen zwei Vektoren in Grad ermitteln

Für den Kosinus des Winkels, den zwei Vektoren x und y im mathematischen Sinn (links herum) einschließen, gilt:¹

$$\cos(x, y) = \frac{x \cdot y}{|x||y|}$$



Um aus $\cos(x, y)$ den Winkel α in Grad zu ermitteln, können Sie folgendermaßen vorgehen:

- mit der Klassenmethode **Math.Acos()** den Winkel im Bogenmaß ermitteln
- das Bogenmaß (*rad*) nach folgender Formel in Grad umrechnen (*deg*):

¹ Dies folgt aus dem Additionstheorem für den Kosinus.

$$deg = \frac{rad}{2\pi} \cdot 360$$

- **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 **Text**) 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	
Definition einer Instanzmethode mit Referenz-Wertparameter		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 dieselben Member eingesetzt werden wie bei einer Klassendefinition (Felder beliebigen Typs, Methoden, Eigenschaften, usw.). 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**.

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 Strukturtypen¹ eine Instanz-Maximalgröße von 16 Bytes.
- Es werden sehr viele Instanzen benötigt.
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).
- Es ist keine Vererbung erforderlich.
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 Zahlen²

¹ URL: [http://msdn.microsoft.com/de-de/library/y23b5415\(en-us\).aspx](http://msdn.microsoft.com/de-de/library/y23b5415(en-us).aspx)

² 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;

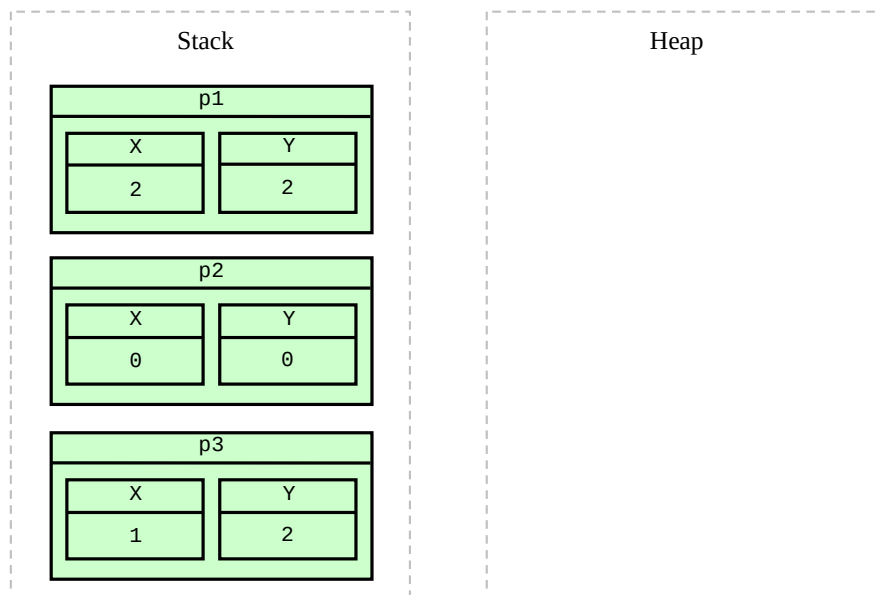
    public Punkt(double x_, double y_) {
        X = x_;
        Y = y_;
    }

    public string Inhalt {
        get {
            return "("+X+";"+Y+");";
        }
    }
}
```

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**, wobei die spätere Änderung von **p1** ohne Effekt auf **p3** bleibt:

Quellcode	Ausgabe (mit Punkt als Struktur)
<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.Inhalt+ "\np2 = "+p2.Inhalt+ "\np3 = "+p3.Inhalt); } }</pre>	<pre>p1 = (2;2) p2 = (0;0) p3 = (1;2)</pre>

Am Ende der Methode **Main()** 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() { }

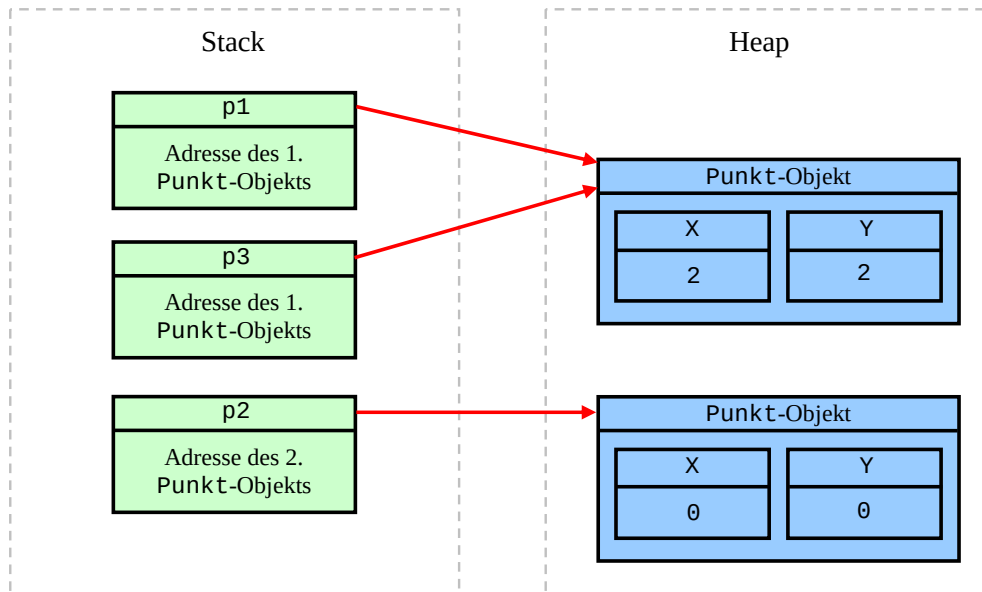
    public string Inhalt {
        get {
            return "("+X+";"+Y+")";
        }
    }
}
```

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):

Quellcode	Ausgabe (mit Punkt als Klasse)
<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.Inhalt+ "\np2 = "+p2.Inhalt+ "\np3 = "+p3.Inhalt); } }</pre>	<pre>p1 = (2;2) p2 = (0;0) p3 = (2;2)</pre>

p1, p2 und p3 sind nun lokale Referenzvariablen, die auf insgesamt zwei Objekte zeigen:



Das erste **Punkt-Objekt** kann über die Referenzvariablen **p1** und **p3** angesprochen (z.B. verändert) werden.

Während ein Objekt *eigenständig* auf dem *Heap* existiert 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 erzeugenden Methode gelöscht. Bei entsprechender Methodendefinition wird dem Aufrufer via Rückgabewert eine Kopie der Strukturinstanz übergeben.

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 und auch *nicht* ersetzt werden darf.
- Auch bei Strukturen ist eine Datenkapselung möglich. Bei der Deklaration bzw. Definition der Member kann der Zugriffsschutz über die Modifikatoren **private** (Voreinstellung), **public** und **internal** reguliert werden. Weil keine Vererbung unterstützt wird, ist der Modifikator **protected** verboten.

- 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 auch unsere Bruch-Klassendefinition in eine analoge Strukturdefinition übersetzen:

```
using System;

public struct Bruch {
    int zaehler, nenner;
    string etikett;

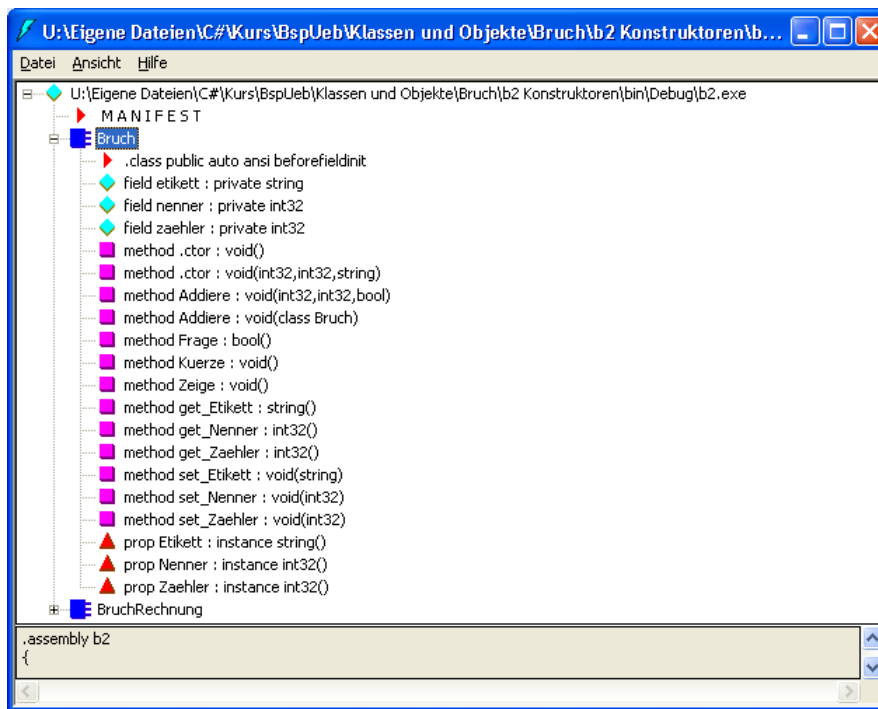
    public Bruch(int zpar, int npar, String epar) {
        zaehler = zpar;
        nenner = npar;
        etikett = epar;
    }
    . . .
}
```

Beim Einsatz der Struktur-Brüche entstehen keine Referenzvariablen mit zugehörigen Heap-Objekten, sondern lokale Variablen (auf dem Stack) mit einer kompletten Struktur als Wert. Vom eingesparten Aufwand ist im folgenden Programm, das lediglich zwei Bruch-Instanzen einsetzt, sicher nichts zu spüren:

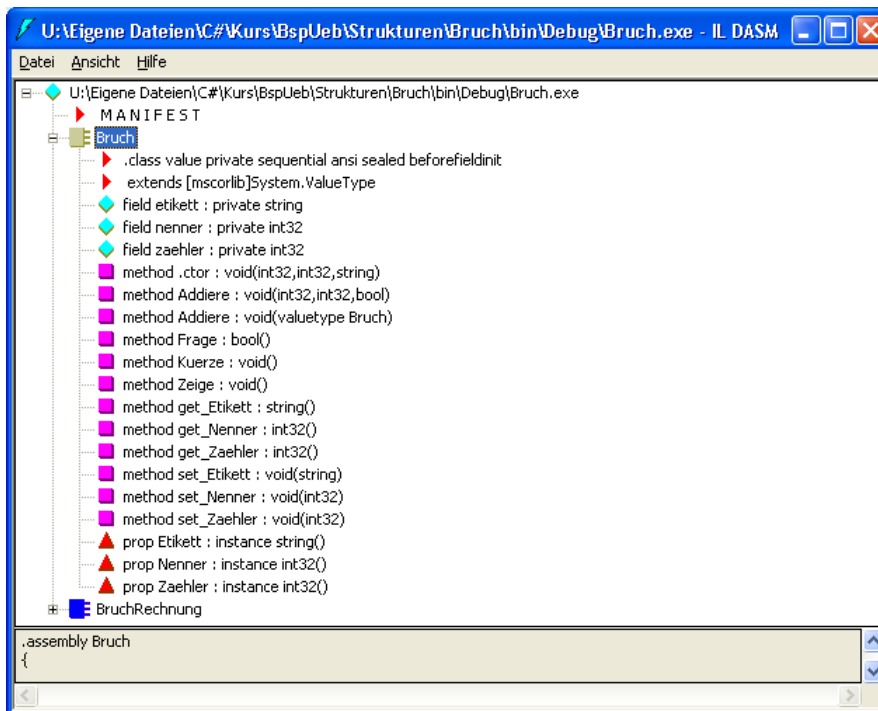
Quellcode	Ausgabe
<pre>using System; class BruchRechnung { static void Main() { Bruch b1 = new Bruch(1,2, "b1 ="); Bruch b2 = new Bruch(); b1.Zeige(); b2.Zeige(); } }</pre>	<pre> 1 b1 = ----- 2 0 ----- 0</pre>

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 identisch ausgestatteten Bruch-Klasse



bei der Bruch-Struktur nur wenige Abweichungen:



Neben der alternativen Symbolfarbe 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**:

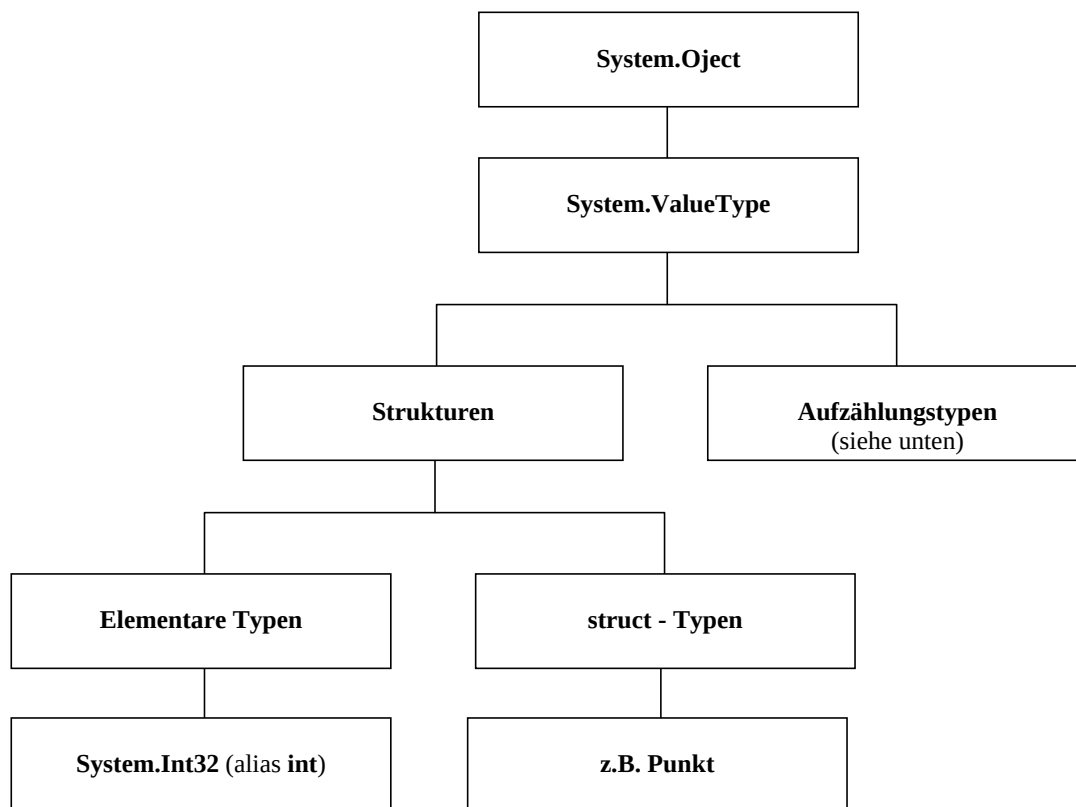
Aliasname	Struktur
sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

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 Systems** (CTS) mit der Urachtklasse **Object** einfügen:



Im folgenden Beispiel wird beim Ganzzahlliteral 13 (vom Strukturtyp **Int32**) über die von **System.Object** geerbte Methode **GetType()** erfolgreich der Datentyp erfragt:

Quellcode	Ausgabe
<pre> using System; class Prog { static void Main() { Console.WriteLine(13.GetType()); } } </pre>	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, also auch für Werttypen gelten. 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 auftritt, wo 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, verständlicherweise eine explizite Casting-Operation mit Angabe des Zieltyps:

```
int j = (int) iBox;
```

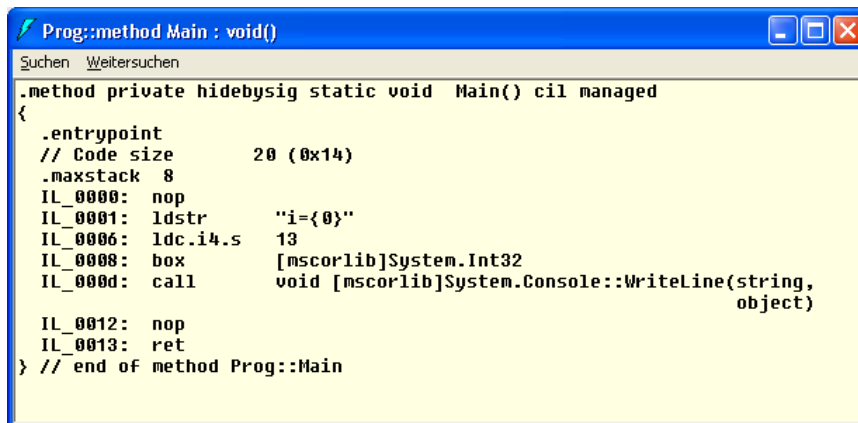
Neben dem bisher beschriebenen *impliziten* Boxing, das auch als **Autoboxing** bezeichnet wird, ist auch ein *explizites* Boxing möglich, aber nie erforderlich, z.B.

```
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)

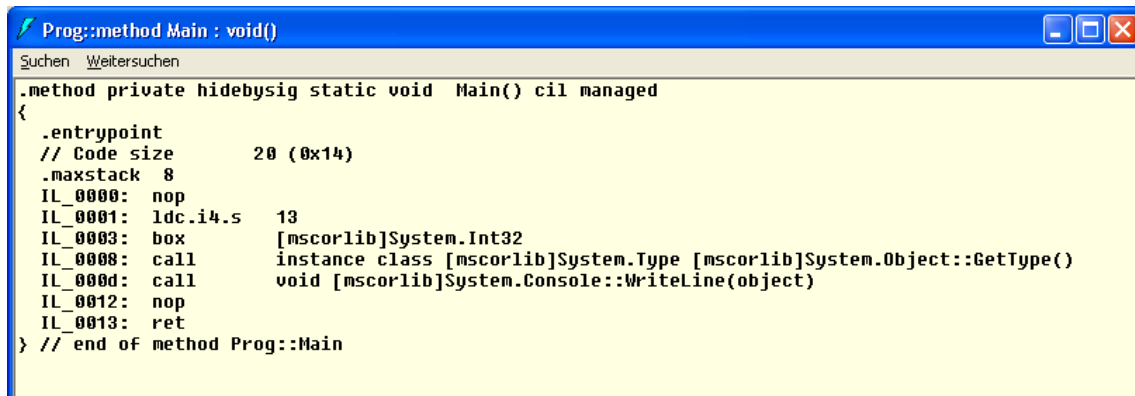
    IL_0012: nop
    IL_0013: ret
} // end of method Prog::Main

```

Anschließend wird die Methode **Console.WriteLine()** mit Aktualparametern vom Typ **String** bzw. **Object** aufgerufen. Auch zur Ausführung des in folgender Anweisung

```
Console.WriteLine(13.GetType());
```

enthaltenen **GetType()**-Aufrufs wird per **box**-OpCode ein Objekt erzeugt:



```

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: ldc.i4.s     13
    IL_0003: box          [mscorlib]System.Int32
    IL_0008: call        instance class [mscorlib]System.Type [mscorlib]System.Object::GetType()
    IL_000d: call        void [mscorlib]System.Console::WriteLine(object)

    IL_0012: nop
    IL_0013: ret
} // end of method Prog::Main

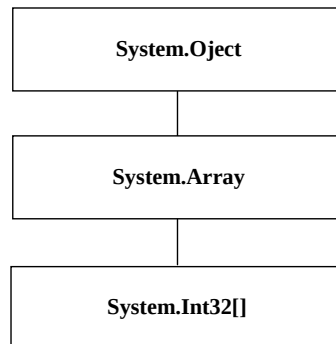
```

5.3 Arrays

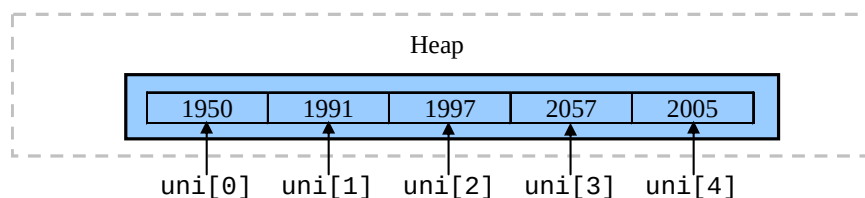
Ein Array ist ein Objekt, das als Instanzvariablen eine feste Anzahl von Elementen desselben Datentyps enthält. Man kann den kompletten Array ansprechen (z.B. als Aktualparameter an eine Methode übergeben), oder auf einzelne Elemente über einen Index zugreifen.

Arrays werden in vielen Programmiersprachen auch *Felder* genannt. In C# bezeichnet man jedoch recht einheitlich die Instanzvariablen einer Klasse oder einer Struktur als *Feld*, 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, die wir aber nicht kennen müssen.

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.

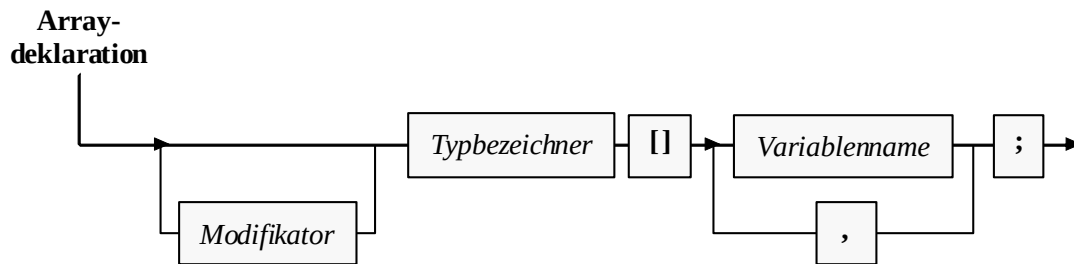
Im Vergleich zur Verwendung einer entsprechenden Anzahl von Einzelvariablen ergibt sich eine erhebliche Vereinfachung der Programmierung:

- 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`:

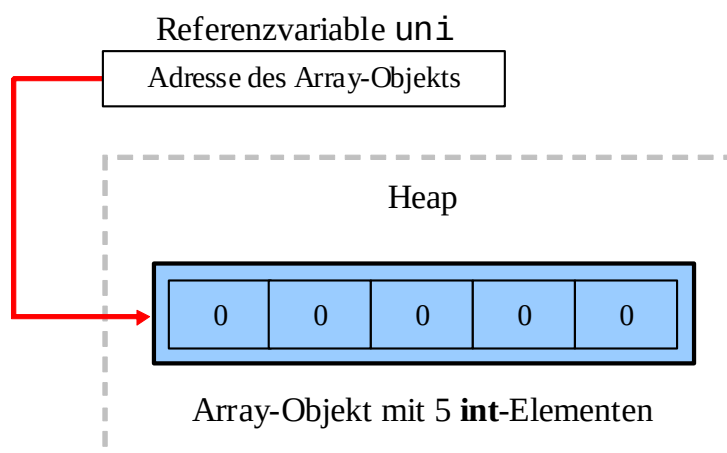
```
uni = new int[max+1];
```

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 4.2.4. 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
<pre>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]); } }</pre>	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 = 1; i <= DRL; i++)
    uni[zsg.Next(5)]++;
```

Den Indexwert liefert die **Random**-Methode **Next()** mit Rückgabotyp **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 **IndexOutOfRangeException**, z.B.:

```
Unbehandelte Ausnahme: System.IndexOutOfRangeException: Der Index war
außerhalb des Arraybereichs.
at UniRand.Main() in ...\\BspUeb\\Arrays\\UniRand\\UniRand.cs:line 7
```

Wenn das verantwortliche Programm einen solchen Ausnahmefehler nicht behandelt (siehe 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.:

Quellcode	Eingabe (fett) und Ausgabe
<pre>using System; class Prog { static void Main() { Console.WriteLine("Länge des Vektors: "); int[] wecktor = new int[Convert.ToInt32(Console.ReadLine())]; Console.WriteLine(); for (int i = 0; i < wecktor.Length; i++) { Console.WriteLine("Wert von Element " + i + ": "); wecktor[i] = Convert.ToInt32(Console.ReadLine()); } Console.WriteLine(); for(int i = 0; i < wecktor.Length; i++) Console.WriteLine(wecktor[i]); } }</pre>	Länge des Vektors: 3 Wert von Element 0: 7 Wert von Element 1: 13 Wert von Element 2: 4711 7 13 4711

Auch beim Entwurf von *Methoden* mit Array-Parametern ist es von Vorteil, dass die Länge eines übergebenen Arrays ohne entsprechenden Zusatzparameter in der Methode bekannt ist.

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 und überprüft deren Verteilung:

```

using System;
class UniRand {
    static void Main() {
        const int DRL = 10000;
        int i;
        int[] uni = new int[5];
        Random zsg = new Random();
        for (i = 1; i <= DRL; i++)
            uni[zsg.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 auftreten sollten. Im Programm dient der 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 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 Beispielpogramm verwendete Klasse **Random** und deren **Next()**-Methode liefert die als wesentlicher Bestandteil der *MSDN 2008 Express Edition* (vgl. Abschnitt 2.2.2) installierte FCL-Dokumentation ausführliche Informationen, die von Visual C# 2008 aus z.B. so zu erreichen sind:

- Starten Sie den Document-Explorer über die Option **Suchen** im **Hilfe**-Menü.
- **Suchen** Sie nach dem Klassennamen **Random**.
- Klicken Sie auf den Treffer **Random-Member**.
- Klicken Sie auf die Methode **Next** und anschließend auf die Überladung **Next(Int32)**.
- Auf der folgenden Seite werden Syntax und Semantik der Methode erklärt:



Eine weitere Anleitung zur Nutzung der vom Document-Explorer sehr gut erschlossenen 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.:

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { int[] wecktor = {1, 2, 3}; Console.WriteLine(wecktor[2]); } }</pre>	3

Die Deklarations- und Initialisierungsanweisung

```
int[] wecktor = {1, 2, 3};
```

ist äquivalent zu:

```
int[] wecktor = new int[3];
wecktor[0] = 1;
wecktor[1] = 2;
wecktor[2] = 3;
```

Weil in der Deklarations- und Initialisierungsanweisung kein **new**-Operator auftaucht, spricht man auch vom *impliziten* Erzeugen eines **Array**-Objekts.

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	Ausgabe
<pre>using System; class BruchRechnung { static void Main() { Bruch b1 = new Bruch(1, 2, "b1 = "); Bruch b2 = new Bruch(5, 6, "b2 = "); Bruch[] bruek = {b1, b2}; bruek[1].Zeige(); } }</pre>	<pre> 5 b2 = ---- 6 </pre>

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	Ausgabe
<pre>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} Dimens.,\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(); } } }</pre>	<pre> 2 Dimens., 4 Zeilen, 3 Spalten 1 2 3 2 4 6 3 6 9 4 8 12 </pre>

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.

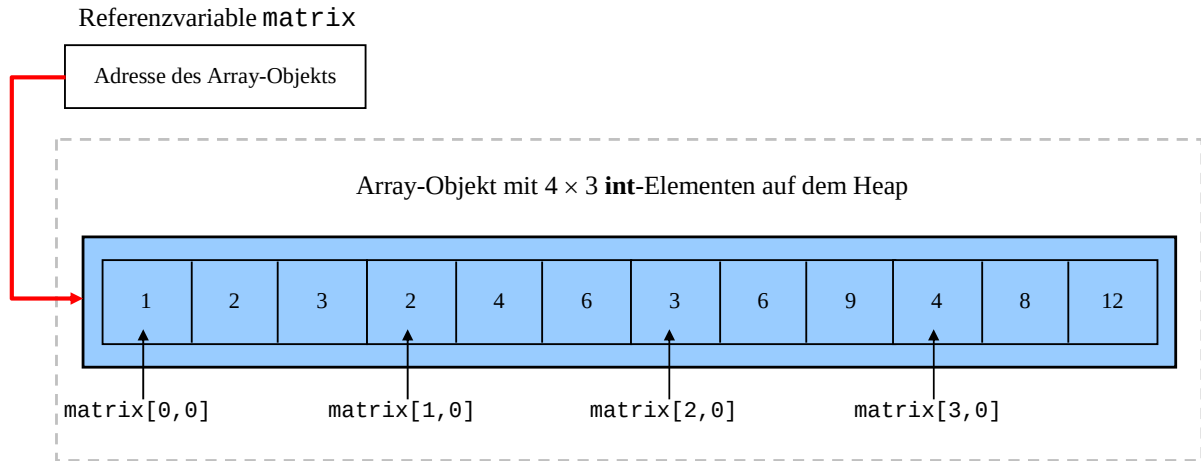
Auch im mehrdimensionalen Fall können Initialisierungslisten eingesetzt werden, z.B.:

```
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* Matrizen liegen im Speicher alle Elemente unmittelbar hintereinander, was einen schnellen Indexzugriff ermöglicht:

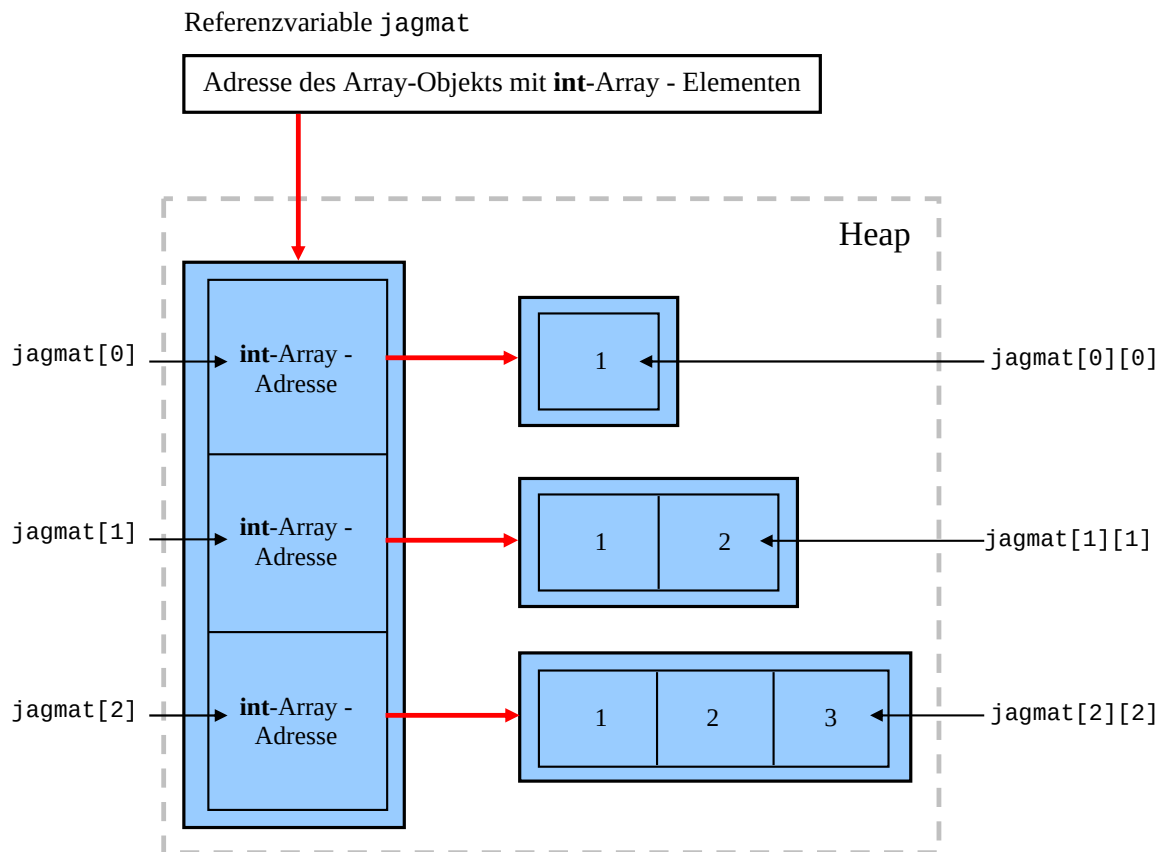


5.3.7.2 Mehrdimensionale Arrays mit unterschiedlich großen Elementen

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	Ausgabe
<pre>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(); } } }</pre>	<pre>1 1 2 1 2 3</pre>

Im Unterschied zum `int[,]` - Objekt `matrix` aus dem letzten Beispiel, das `int`-Elemente enthält, handelt es sich bei den Elementen des `int[][]` - Objekts `jagmat` um Referenzen vom Typ `int[]`, die wiederum auf entsprechende Heap-Objekte (oder `null`) zeigen können. Während `matrix` ein zweidimensionaler Array mit `int`-Elementen ist, handelt es sich bei `jagmat` um einen *eindimensionalen* 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.:

Quellcode	Ausgabe
<pre> 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]); } } </pre>	Was fällt Ihnen zu C#? : Tolle Sache : Nicht ganz trivial : Macht Spaß : Ihre Anmerkungen: Tolle Sache Nicht ganz trivial Macht Spaß

Das Fassungsvermögen des Containers wird bei Bedarf automatisch erhöht, wobei eine leistungsoptimierende Logik dafür sorgt, dass diese Anpassungsmaßnahme möglichst selten erforderlich ist. Über die Eigenschaft **Capacity** kann die momentane Kapazität festgestellt und auch eingestellt werden. Mit der Methode **TrimToSize()** reduziert man die Größe auf den momentanen Bedarf, z.B. nach der Voraussichtlich letzten Neuaufnahme.

Weil die Klasse **ArrayList** einen *Indexer* bietet (siehe Abschnitt 5.6), 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 **object** (alias **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 (Un)boxing-Technik (siehe Abschnitt 5.2) auch die Werttypen erlaubt sind, z.B.:

Quellcode	Ausgabe
<pre> 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); } } </pre>	Wort 3,14 13

Ein solcher „Gemischtwarenladen“ (allerdings mit *fester* Länge) ist durch Wahl des Element-Datentyps **object** übrigens auch mit einem einfachen Array zu realisieren.

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* Zeichenketten sollte unbedingt die Klasse **StringBuilder** verwendet werden, weil deren Objekte nach dem Erzeugen noch verändert 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:

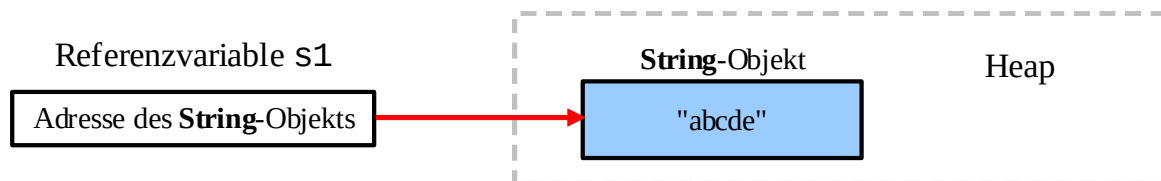
- eine **String**-Referenzvariable namens **s1** angelegt,
- ein neues **String**-Objekt mit dem Inhalt „abcd“ auf dem Heap erzeugt,
- die Adresse des neuen Heap-Objekts in der Referenzvariablen abgelegt.

Soviel objektorientierten Hintergrund sieht man der angenehm einfachen Anweisung auf den ersten Blick nicht an. In C# sind jedoch auch Zeichenketten*litterale* 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. Sie 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 Inflexibilität des **String**-Inhalts in Zweifel ziehen und ein Gegenbeispiel der folgenden Art vorbringen:

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { String testr = "abc"; Console.WriteLine("testr = " + testr); testr = testr + "def"; Console.WriteLine("testr = " + testr); } }</pre>	<pre>testr = abc testr = abcdef</pre>

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 Interner String-Pool

Wie oben erläutert, bildet ein Zeichenkettenliteral einen Ausdruck vom Typ **String** mit einer Objektadresse als Wert. Für wiederholt im Quellcode auftretende Zeichenkettenliterals lässt dieses Prinzip eine Verschwendung von Speicherplatz befürchten. Um dies 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:

Quellcode	Ausgabe
<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)); Console.ReadLine(); } }</pre>	<pre>True</pre>

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 ändern, 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 Parameterstring, wird die Adresse dieses Pool-Objekts geliefert.
- Anderenfalls wird der Parameterstring in den internen **String**-Pool aufgenommen und seine Adresse als Rückgabe geliefert.

Durch Internalisieren kann Speicherplatz gespart werden, wenn viele Strings mit identischem Inhalt zu erwarten sind. Außerdem können **String**-Vergleiche (vgl. Abschnitt 5.4.1.3.2) 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 = 20, 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;
        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;
        for (int j = 1; j < ANZ; j++)
            sar[j] = String.Intern(sar[j]);
        for (int n = 1; n <= N; n++)
            for (int i = 0; i < ANZ; i++)
                if ((sar[i] as Object) == sar[ran.Next(ANZ)])
                    hits++;
        Console.WriteLine((N * ANZ) + " Adressvergleiche (" + hits +
            " hits) benoetigen (inkl. Internieren) " +
            ((DateTime.Now.Ticks - start) / 1.0e4) + " Millisekunden");
    }
}
```

Um den Identitätsoperator zu Adressvergleichen zu zwingen, wird ein Vergleichspartner als Instanz der Klasse **Object** behandelt:

```
(sar[i] as Object) == sar[ran.Next(ANZ)]
```

Es hängt von den Aufgabenparametern ANZ, LEN und N ab, welche Vergleichsmethode überlegen ist:¹

¹ Die Ergebnisse stammen von einem PC mit Intel - CPU (Pentium 4, 3-GHz).

	Laufzeit in Millisekunden	
	Inhaltsvergleiche	Internieren u. Adressvergl.
ANZ = 50000, LEN = 20, N = 5	63	109
ANZ = 50000, LEN = 20, N = 50	547	266

Erwartungsgemäß ist das Internieren umso rentabler, je mehr Vergleiche anschließend mit den Zeichenfolgen angestellt werden. Bei **String**-Vergleichen sind sicher noch weitere Verbesserungen möglich, z.B. durch Ausnutzen der lexikographischen Ordnung.

Auch die **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.

5.4.1.3 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.

5.4.1.3.1 Verketteten von Strings

Weil die Klasse **String** den „+“-Operator geeignet überladen hat (vgl. Abschnitt 4.7.2), taugt er zum Verketteten von **String**-Objekten, wobei Operanden beliebige Datentypen bei Bedarf automatisch in **String**-Objekte konvertiert werden. Wie Sie aus Abschnitt 5.4.1.1 wissen, entsteht beim Verketteten von zwei Zeichenfolgen ein *neues* **String**-Objekt. In folgendem Beispiel wird mit Klammern dafür gesorgt, dass der Compiler die „+“-Operatoren jeweils sinnvoll interpretiert (Verketteten von Strings bzw. Addieren von Zahlen):

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { Console.WriteLine("4 + 3 = " + (4 + 3)); } }</pre>	4 + 3 = 7

5.4.1.3.2 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
<pre>using System; class Prog { static void Main() { String s1 = "abcde"; String s2 = "de"; String s3 = "abc" + s2; Console.WriteLine(s1 == s3); } }</pre>	<pre>class Prog { public static void main(String[] args) { String s1 = "abcde"; String s2 = "de"; String s3 = "abc" + s2; System.out.println(s1 == s3); } }</pre>

Mit der etwas umständlichen **s3**-Konstruktion wird verhindert, dass **s1** und **s3** auf dasselbe Objekt im internen **String**-Pool zeigen, weil dann Adressen- und Inhaltsvergleich zum selben Ergebnis kämen. Wie in Abschnitt 5.4.1.2 demonstriert wurde, ist bei einer großen Anzahl von **String**-Vergleichen eventuell durch Internalisierung und Verwendung von Adressenvergleichen eine erhebliche Leistungssteigerung zu erzielen.

Zum Testen auf **lexikographische Priorität** (z.B. beim Sortieren) kann die **String**-Methode **CompareTo()** dienen:

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { String a = "Müller, Anja", b = "Müller, Kurt", c = "Müller", d = ", Anja"; //a und c sollen nicht auf denselben Pool-String zeigen c = c + d; Console.WriteLine("< : " + a.CompareTo(b)); Console.WriteLine("= : " + a.CompareTo(c)); Console.WriteLine("> : " + b.CompareTo(a)); } }</pre>	<pre>< : -1 = : 0 > : 1</pre>

CompareTo() liefert folgende Ergebnisse zurück:

		CompareTo()-Ergebnis
Das befragte String -Objekt ist im Vergleich zum Aktualparameter lexikographisch ...	kleiner	-1
	gleich	0
	größer	1

5.4.1.3.3 Länge einer Zeichenkette

Über die Länge einer Zeichenkette informiert die **String**-Eigenschaft **Length**, z.B.:

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { Console.WriteLine("abc".Length); } }</pre>	3

5.4.1.3.4 Zeichen(folgen) extrahieren, suchen oder ersetzen

Auf einzelne Zeichen eines Strings kann man per Indexsyntax zugreifen, z.B.:

Quellcode	Ausgabe
<pre>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')); } }</pre>	<pre>a ab 2 -1 True abCd</pre>

Über die Methode

public string Substring(int start, int anzahl)

erhält man von einem **String**-Objekt die *anzahl* Zeichen ab Position *start* als Kopie.

Eben wurde eine Technik zur Beschreibung einer *vorhandenen Bibliotheksmethode* (im Unterschied zur Beschreibung einer C#-Syntaxregel) erstmals benutzt, die auch in der FCL-Dokumentation in ähnlicher Form Verwendung findet, z.B.:

```
C#
public string Substring(
    int startIndex,
    int length
)
```

Dabei wird die Benutzung einer vorhandenen Methode erläutert durch Angabe von:

- Modifikatoren (z.B. für den Zugriffsschutz)
- Rückgabetyt
- Methodenname
- Parameterliste (mit Angabe der Parametertypen)

Analog werden wir auch Eigenschaften von FCL-Klassen oder -Strukturen beschreiben.

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.

5.4.1.3.5 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.

5.4.2 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 Objektes entfällt.

Ein **StringBuilder**-Objekt kann *nicht* implizit erzeugt werden, jedoch stehen bequeme Konstrukturen 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

¹ Gemessen auf einem Rechner mit Intel-CPU (Pentium 4, Single Core, 3 GHz Taktfrequenz).

Zeit für String-Manipulation: 93,75

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.: <code>t.Append(" * ");</code> 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.: <code>sb.Insert(4, 3.14);</code>
Remove()	Ab einer Startposition wird eine Anzahl von Zeichen entfernt, z.B.: <code>sb.Remove(100, 500);</code>
Replace()	Ein Zeichen bzw. eine Zeichenfolge des StringBuilder -Objekts wird durch anderes Zeichen bzw. eine andere Zeichenfolge ersetzt, z.B.: <code>sb.Replace("alt", "neu");</code>
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: <code>Console.WriteLine(s == sb.ToString());</code>

5.5 Enumerationen

Angenommen, Sie wollen in einer Adressdatenbank auch den *Charakter* der erfassten Personen notieren und sich dabei an den vier Temperamentstypen des griechischen Philosophen Hippokrates (ca. 460 - 370 v. Chr.) orientieren: cholerisch, melancholisch, sanguin, phlegmatisch. Um dieses Merkmal mit seinen vier möglichen Ausprägungen in einer Instanzvariablen zu speichern, haben Sie 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 == 13) ...`

C# bietet mit den **Enumerationen (Aufzählungstypen)** eine Lösung, die folgende Vorteile bietet:

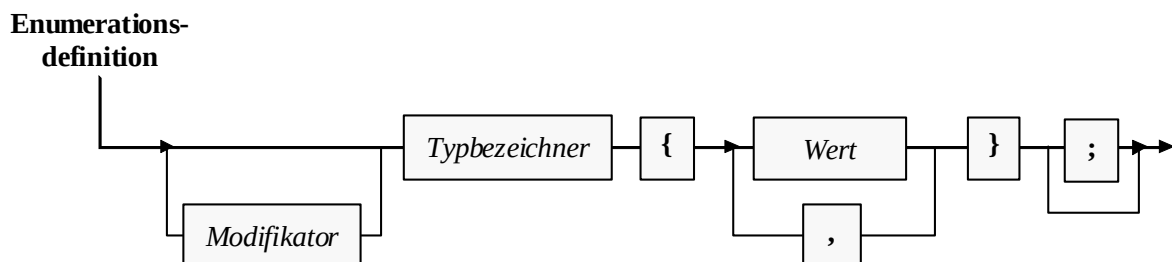
- Gut lesbarer Quellcode durch Klartextnamen für die Merkmalsausprägungen
- Falsch geschriebene Klartextnamen werden vom Compiler abgewiesen.
- Geringer Speicherbedarf

Eine Enumeration basiert auf einem zugrunde liegenden integralen Typ (meist **int**) und enthält eine (meist kleine) Menge von benannten Konstanten dieses Typs, z.B.:

Benannte Konstanten vom Typ Temperament			Werte vom Typ int	
<code>Temperament.Cholerisch</code>	$\cong$	0	-2147483648	
<code>Temperament.Melancholisch</code>	$\cong$	1	0	
<code>Temperament.Sanguin</code>	$\cong$	2	1	
<code>Temperament.Phlegmatisch</code>	$\cong$	3	2	
			3	
			.	
			.	
			.	
			-1	
			0	
			1	
			2	
			3	
			.	
			.	
			.	
			2147483647	

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 geschweifft eingeklammerte Liste mit Namen für die Konstanten:



Als Beispiel betrachten wir einen Enumerationstyp für die Erfassung des Charakters in der Adressdatenbank:

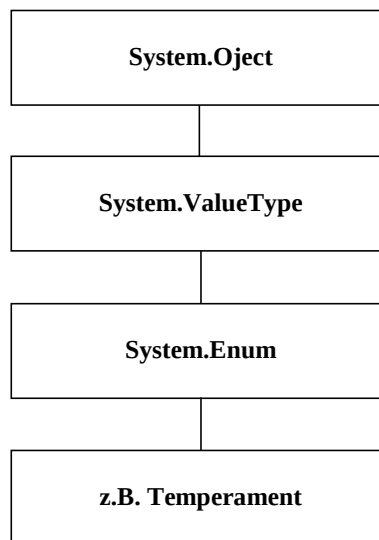
```
enum Temperament {Cholerisch, Melancholisch, Sanguin, Phlegmatisch}
```

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).

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.

Die Enumerationen sind Werttypen, die folgendermaßen in das CTS (Common Type System) der .NET – Plattform eingeordnet sind:



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.

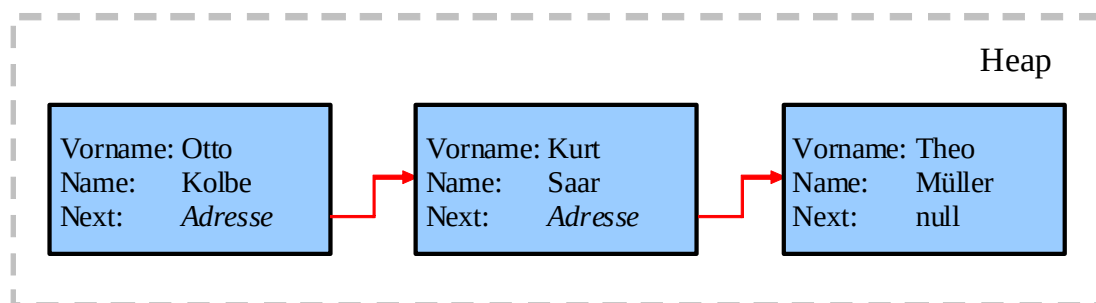
5.6 Indexer

Bei Arrays sowie bei den Klassen **String** und **ArrayList** hat sich der Indexzugriff auf die Elemente eines Objekts als sehr angenehm erwiesen. Um denselben Komfort für eine eigene Klasse oder Struktur zu realisieren, 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*.

Als Beispiel betrachten wir eine „Datenbank“, die mit einer so genannten *verketteten Liste* von Objekten der 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 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 (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) {
            Person sel = first;
            for (int j = 0; j < i - 1; j++)
                sel = sel.Next;
            value.Next = sel.Next.Next; // Nachfolger des Neulings
            sel.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**).

Einige Regeln für Indexer:

- Ihr Name lautet stets **this**.
- Den Indextyp bestimmt der abgelieferte Wert.
- 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);
        Console.ReadLine();
    }
}

```

Das Anwendungsprogramm liefert 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 eigene Klassen 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 7.5 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 nicht unerhebliche Aufwand des `PersonDB`-Beispiels wohl doch lohnt.

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?

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { object o1 = 1; object o2 = 1; Console.WriteLine(o1 == o2); } }</pre>	False

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 $\{1, 2, 3, \dots, K\}$.

- 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 $2 \cdot b, 3 \cdot b, \dots$) werden aus der Kandidatenmenge gestrichen, während die Zahl b in der Liste verbleibt.
- Das Streichverfahren kann enden, wenn für eine neue Basis b gilt:

$$b > \sqrt{K}$$

Ist eine natürliche Zahl $n \leq K$ ein Vielfaches von b , dann gibt es eine natürliche Zahl n_b mit

$$n = n_b \cdot b \text{ und } n_b < \sqrt{K}$$

Wir nehmen es ganz genau und unterscheiden zwei Fälle:

- n_b war zuvor als Basis dran:
Dann wurde n bereits als Vielfaches von n_b gestrichen.
- n_b wurde zuvor als Vielfaches einer Basis $\tilde{b}$ gestrichen ($n_b = u\tilde{b}$)
Dann wurde n bereits als Vielfaches von $\tilde{b}$ gestrichen: $n = n_b \cdot b = u\tilde{b}b = ub\tilde{b}$

Die Kandidatenmenge enthält also nur noch Primzahlen.

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 $\sqrt{18}$ ($\approx 4,24$) 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:

- Vor- und Nachnamen als Kommandozeilenargumente einlesen,
- den Anfangsbuchstaben 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 Kommandozeilenparameter übergeben wurden.

Tipps:

- Um die durch Leerzeichen getrennten Kommandozeilenargumente 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:

- Am Anfang einer neuen Ausgabezeile sollen keine Leerzeichen stehen.
- Ist ein Wort breiter als die Ausgabezeile, ist ein Umbruch innerhalb des Wortes unvermeidlich.

In folgendem 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 z.B. 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:

Quellcode	Ausgabe
<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[] { ' ', '-' }); foreach (string t in tokens) Console.WriteLine(t); } } </pre>	<pre> Dies ist der Beispiel Satz, der zerlegt werden soll. </pre>

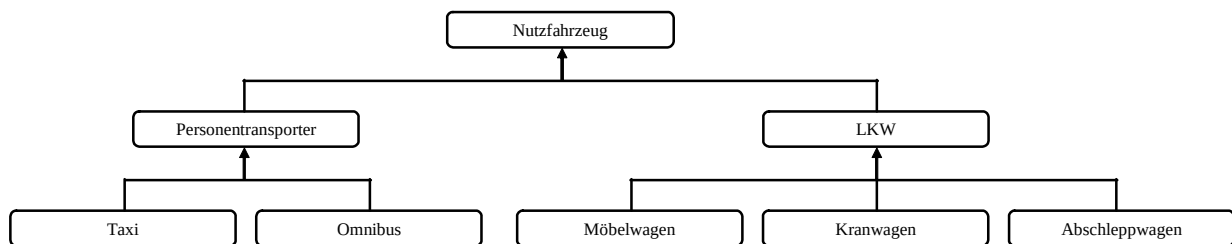
Die Trennzeichen sind *nicht* in den produzierten Teilzeichenfolgen enthalten, so dass z.B. ein als Trennzeichen definierter Bindestrich verloren geht. Mit etwas Aufwand lässt sich dieser Verlust verhindern.

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

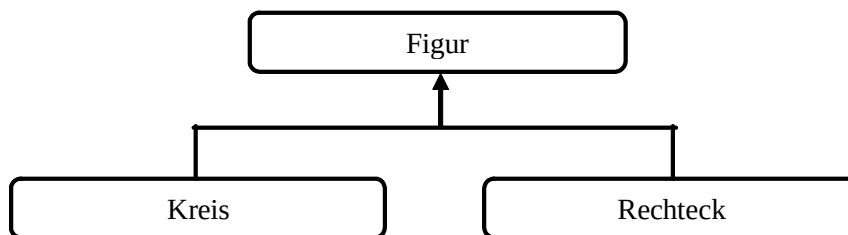
Beim Modellieren eines Gegenstandsbereiches durch Klassen, die durch Merkmale (Instanz- und Klassenvariablen) und Handlungskompetenzen (Instanz- und Klassenmethoden) gekennzeichnet sind, müssen auch die Spezialisierungs- bzw. Generalisierungsbeziehungen zwischen real existierenden Klassen abgebildet 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:

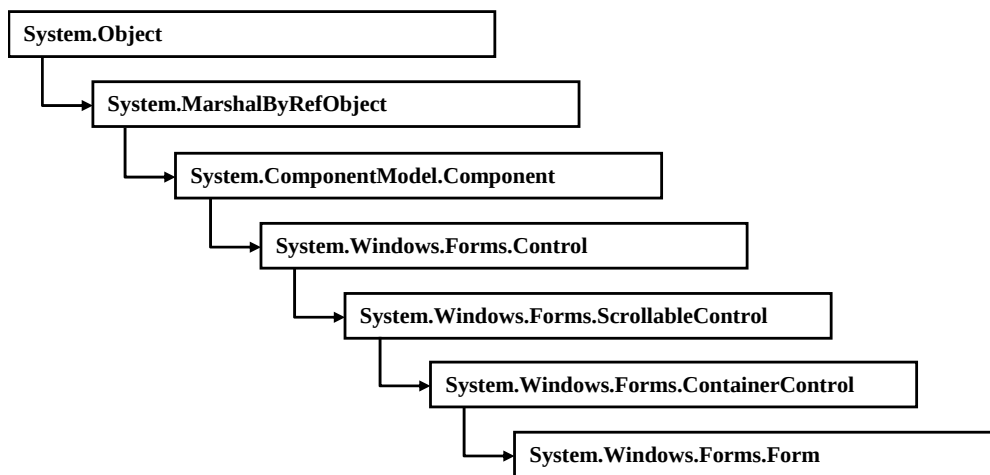


Die Vererbungstechnik der OOP

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.:

- 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 **Form** für die Hauptfenster von WinForms-Anwendungen:

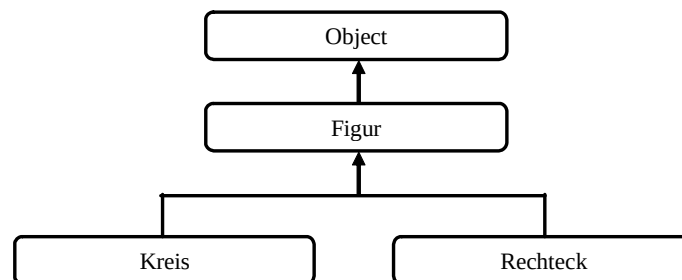


Software-Recycling

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. Spätere Verbesserungen bei einer Basisklasse kommen allen (direkt oder indirekt) abgeleiteten Klassen zu Gute. Die verbreitete Praxis, vorhanden Code per *Copy & Paste* in neuen Projekten 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, Delegaten) 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 **Object** ab. Die oben dargestellte Klassenhierarchie zum Figurenübungsbeispiel muss also folgendermaßen vervollständigt werden:

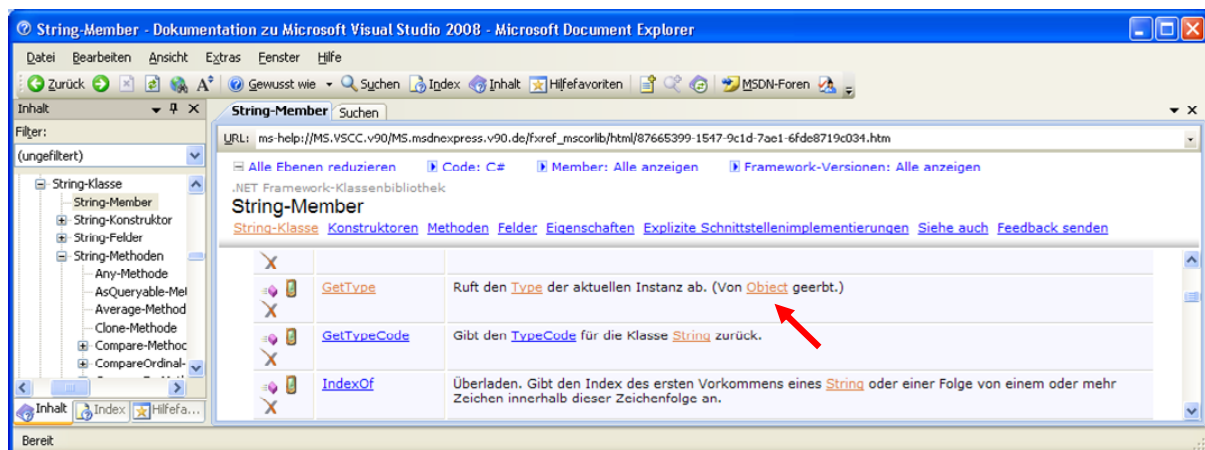


Auch die Strukturen sind in die globale Hierarchie eingeordnet: Sie stammen implizit von der Klasse **ValueType** im Namensraum **System** ab, die wiederum direkt von der Urahn-Klasse **Object** erbt (vgl. Abschnitt 5.2). Aus einer Struktur kann aber weder eine andere Struktur noch eine Klasse abgeleitet werden. Analoges gilt für die Aufzählungstypen, die von der Klasse **System.Enum** 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 **Type**-Objekt mit den verfügbaren Metadaten liefert. Im folgenden **WriteLine()**-Aufruf verraten drei **Type**-Objekte (über die implizit aufgerufene Methode **ToString()**) die Typbezeichnung samt Namensraum:

Quellcode	Ausgabe
<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>	<pre>System.Object System.String System.Int32</pre>

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) {
        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) {
        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 (und öffentlich verfügbaren) Konstruktor der Basisklasse aufruft. Weil die `Figur`-Instanzvariablen (noch) als **private** deklariert sind, wäre dem `Kreis`-Konstruktor ohnehin kein direkter Zugriff erlaubt.

Konstrukturen 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-Konstrukturen und Initialisierungs-Sequenzen

Zwar werden Konstrukturen nicht vererbt, doch ist bei der Entstehung eines Objekts einer abgeleiteten Klasse ein Konstruktor aus *jeder* Basisklasse entlang der Ahnenreihe durch impliziten oder expliziten Aufruf beteiligt. Das folgende Programm erzeugt ein Objekt aus der Basisklasse `Figur` und ein Objekt aus der abgeleiteten Klasse `Kreis`, wobei die beteiligten Konstrukturen dieser Klassen ihre Tätigkeit melden:

Quellcode	Ausgabe
<pre> using System; class Prog { static void Main() { Figur fig = new Figur(50.0, 50.0); Kreis krs = new Kreis(150.0, 200.0, 50.0); } } </pre>	<pre> Figur-Konstruktor Figur-Konstruktor Kreis-Konstruktor </pre>

Vom ebenfalls beteiligten Object-Konstruktor ist nichts zu sehen, weil die FCL-Designer natürlich keine Kontrollausgabe einbaut haben. Wir werden die Beiträge der einzelnen Konstrukturen 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) {  
    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 übrigens auch vom Standardkonstruktor der 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` aus.
- Der explizit über das Schlüsselwort **base** aufgerufene `Figur`-Konstruktor mit Positionsparametern startet und führt die Initialisierungen `xpos = 100.0` sowie `ypos = 100.0` aus.
- Der parameterlose **Object**-Konstruktor startet. Die Instanzvariablen der Klasse **Object** erhalten den Initialisierungswert laut Deklaration. 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 Konstruktoraufrufs `Figur(150.0, 200.0)` wird ausgeführt, wobei `xpos` und `ypos` die Werte 150 bzw. 200 erhalten.
- Der Rumpf des Konstruktoraufrufs `Kreis(150.0, 200.0, 50.0)` wird ausgeführt, wobei `radius` den Wert 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) {
        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 `SetzePos()` verändert werden:

```
using System;
public class Kreis : Figur {
    double radius = 75;
    public Kreis(double x, double y, double rad) : base(x, y) {
        radius = rad;
        Console.WriteLine("Kreis-Konstruktor");
    }
    public Kreis() {}
    public double Radius {
        get {return radius;}
    }
    public void SetzePos(double x, double y) {
        xpos = x;
        ypos = y;
    }
}
```

Es ist zu beachten, dass hier *geerbte Instanzvariablen* von `Kreis`-Objekten verändert werden. Bei entsprechender Methodenausstattung kann ein `Kreis`-Objekt selbstverständlich 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;           verboten
        krs.SetzePos(77.7, 99.4); // erlaubt
    }
}
```

Der Modifikator **protected** ist natürlich nicht nur bei Feldern erlaubt, sondern auch bei (instanz- oder klassenbezogenen) *Methoden*, z.B.:

```
static protected void ProSt() {
    Console.WriteLine("Protected und statisch!");
}
```

6.5 Erbstücke durch spezialisierte Varianten verdecken

6.5.1 Geerbte Methoden verdecken

Eine geerbte Basisklassenmethode kann in einer Unterklasse 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.

Bisher steht in der *Kreis*-Klasse zur Ortsangabe die geerbte Methode *Wo()*, welche die Position der linken oberen Ecke eines Objekts ausgibt. In der *Kreis*-Klasse kann aber eine bessere Ortsangabenmethode realisiert werden, weil hier auch die rechte untere Ecke definiert ist:¹

```
using System;
public class Kreis : Figur {
    double radius = 75.0;
    public Kreis(double x, double y, double rad) : base(x, y) {
        radius = rad;
    }
    public Kreis() { }
    public double Radius {
        get {return radius;}
    }
    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**² angeben, um die folgende Warnung des Compilers zu vermeiden:

```
Kreis.cs(7,17): warning CS0108: Kreis.Wo() blendet den geerbten Member
    Figur.Wo() aus. Verwenden Sie das Schlüsselwort "new", wenn das
    Ausblenden beabsichtigt ist.
```

¹ Falls Sie sich über die Berechnungsvorschrift für die Y-Koordinate der rechten unteren Kreis-Ecke wundern: Bei der Grafikausgabe von Computersystemen ist die Position (0, 0) meist in der oberen linken Ecke des Bildschirms bzw. des aktuellen Fensters angesiedelt. Die X-Koordinaten wachsen (wie aus der Mathematik gewohnt) von links nach rechts, während die Y-Koordinaten von oben nach unten wachsen. Wir wollen uns im Hinblick auf die in absehbarer Zukunft anstehende Programmierung graphischer Benutzeroberflächen schon jetzt daran gewöhnen.

² Dieser Modifikator darf nicht mit dem Operator **new** verwechselt werden.

Mit diesem Hinweis soll ein ungeplantes Verdecken durch Tippfehler oder Unachtsamkeit 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	Ausgabe
<pre>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(); } }</pre>	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 *intern* verwendet, und in der Schnittstelle der abgeleiteten Klasse bleibt das signaturgleiche Erbstück erhalten, z.B.:

Quellcode	Ausgabe
<pre>using System; class Basisklasse { public void NenneTyp() { Console.WriteLine("Basisklasse"); } } class Abgeleitet : Basisklasse { new void NenneTyp() { Console.WriteLine("Abgeleitet"); } public void DeinTyp() { NenneTyp(); } } class Prog { static void Main() { Basisklasse b = new Basisklasse(); b.NenneTyp(); Abgeleitet a = new Abgeleitet(); a.NenneTyp(); a.DeinTyp(); } }</pre>	Basisklasse Basisklasse Abgeleitet

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 anzusprechen ist, z.B.:

Quellcode	Ausgabe
<pre> using System; class Basisklasse { public static void M() { Console.WriteLine("Basisklasse"); } } class Abgeleitet : Basisklasse { public new static void M() { Console.WriteLine("Abgeleitet"); } } class Prog { static void Main() { Basisklasse.M(); Abgeleitet.M(); } } </pre>	Basisklasse Abgeleitet

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
<pre> 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(); } } </pre>	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:

```
Verdecken.cs(9,6): warning CS0108: Sohn.x blendet den geerbten Member Vater.x
aus. Verwenden Sie das Schlüsselwort "new", wenn das Ausblenden
beabsichtigt ist.
```

Mit diesem sehr aufmerksamen Hinweis soll ein ungeplantes Verdecken durch Tippfehler oder Unachtsamkeit verhindert werden.

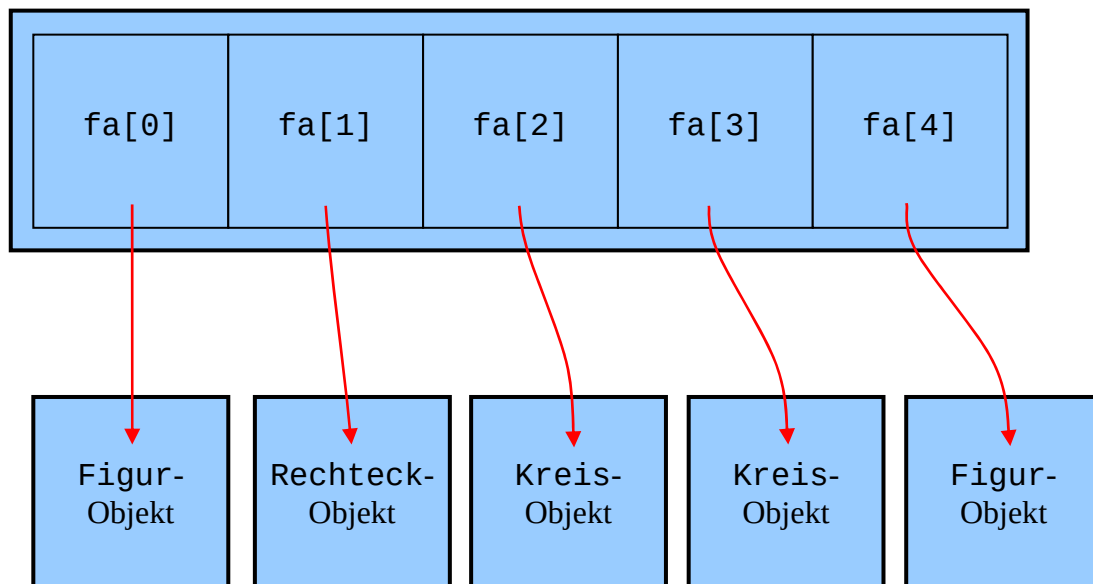
6.6 Verwaltung von Objekten über Basisklassenreferenzen

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 Grafik-Programms kommt vielleicht ein Array mit dem Elementtyp **Figur** zum Einsatz, dessen Elemente auf Objekte aus der Basisklasse oder aus einer abgeleiteten Klasse wie **Kreis** oder **Rechteck** zeigen:

Array **fa** mit Elementtyp **Figur**



Ein Array vom Typ **Figur** kann Referenzen auf Figuren und Kreise aufnehmen, z.B.:

¹ Dieser Modifikator darf nicht mit dem Operator **new** verwechselt werden.

```

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 sich über den `is`-(Typstest-)Operator vergewissern, ob das referenzierte Objekt tatsächlich zur vermuteten Klasse gehört, z.B.:

```

for (int i = 0; i < 2; i++) {
    fa[i].Wo();
    if (fa[i] is Kreis)
        Console.WriteLine("Radius: " + ((Kreis)fa[i]).Radius);
}

```

An Stelle des gewohnten Typumwandlungsoperators

```
((Kreis)fa[1]).Radius
```

kann im Beispiel auch der `as`-Operator eingesetzt werden:

```
(fa[2] as Kreis).Radius
```

Die wichtigsten Regeln für den `as`-Operator:

- Der Zieltyp im zweiten Operanden muss ein Referenztyp oder ein **null**-fähiger Typ (siehe Abschnitt 7.3) sein.
- Im ersten Operanden verlangt der Compiler ist ein Ausdruck, dessen Wert potentiell in den Zieltyp konvertiert werden kann. Dies ist z.B. bei einer Variablen vom Typ **Object** unabhängig vom konkreten Zieltyp stets der Fall. Weitere Details finden sich in der C# 3.0 - Sprachspezifikation (Microsoft 2007a, S. 198).

- Stellt sich zur Laufzeit eine Konvertierung als unmöglich heraus, liefert der **as**-Operator im Unterschied zum gewohnten Typumwandlungsoperator *keinen* Ausnahmefehler, sondern den Ergebniswert **null**.

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 unterschiedlich (jeweils unterklassentypisch) auf dieselbe Botschaft. Genau dieses Phänomen bezeichnet man als **Polymorphie**. Wer sich hier mit einem exotischen und nutzlosen Detail konfrontiert glaubt, sei an die Auffassung von Alan Kay erinnert, der wesentlich zur Entwicklung der objektorientierten Programmierung beigetragen hat. Er zählt die Polymorphie neben der Datenkapselung und der Vererbung zu den Grundelementen dieser Softwaretechnologie (Lahres & Rayman 2006).

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) {
        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) {
        radius = rad;
    }
    public Kreis() { }
    public double Radius {
        get {return radius;}
    }
    public override void Wo() {
        base.Wo();
        Console.WriteLine("Unten Rechts:\t(" + (xpos+2*radius) +
            ", " + (ypos+2*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();
        fa[0].Wo();
        fa[1].Wo();
        Console.WriteLine("Radius:          "+((Kreis)fa[1]).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();
        Console.ReadLine();
    }
}
```

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 (*späte bzw. dynamische Bindung*):

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 erfolgt, darf im Beispielprogramm die Klasse des Array-Elementes `fa[2]` vom Benutzer festgelegt werden.

Weil bei statischen Methoden keine späte Bindung stattfinden kann, sind hier die Modifikatoren **virtual** und **override** verboten. Das per **new**-Modifikator signalisierte Verdecken einer geerbten statischen Methode ist jedoch sinnvoll und erlaubt.

In der folgenden Tabelle werden die beiden Ersetzungsarten für Methoden (Überschreiben und Verdecken) in semantischer und syntaktischer Hinsicht gegenübergestellt:

Ersetzungsart	Unterstützung der Polymorphie	Syntax
Überschreiben	Ja	Im Kopf der Basisklassenmethode muss der Modifikator virtual und im Kopf der Unterklassenmethode der Modifikator override stehen. Statische Methoden können <i>nicht</i> überschrieben werden.
Verdecken	Nein	Mit dem Modifikator new im Kopf der Unterklassenmethode wird unabhängig von der Basismethodendefinition das Verdecken gewählt.

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 `Wachse()`, 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 Wachse(double faktor);
    . . .
}
```

Abstrakte Methoden sind grundsätzlich virtuell (überschreibbar, vgl. Abschnitt 6.6), 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 `Wachse()` implementieren:

```
public class Kreis : Figur {
    double radius = 75.0;
    . . .
    public override void Wachse(double faktor) {
        if (faktor > 0)
            radius *= faktor;
    }
    . . .
}
public class Rechteck : Figur {
    double breite = 50.0, hoehe = 50.0;
    . . .
    public override void Wachse(double faktor) {
        if (faktor > 0) {
            breite *= faktor;
            hoehe *= faktor;
        }
    }
    . . .
}
```

Von den beiden Ersetzungsarten kommt bei einer abstrakten (und damit grundsätzlich virtuellen) Basisklassenmethode `Wachse()` nur das Überschreiben in Frage, wobei das zugehörige Schlüsselwort **override** explizit 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:

Quellcode	Ausgabe
<pre>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].Wachse(2.0); fa[1].Wachse(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); Console.ReadLine(); } }</pre>	<pre>Fläche Figur 0: 314,16 Fläche Figur 1: 100,00 Gesamtfläche: 414,16</pre>

Neben den Methoden können auch die 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;
}
```

In den abgeleiteten Klassen `Kreis` und `Rechteck` wird die Eigenschaft 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 für Typen *ohne* gemeinsame Basisklasse realisieren.

6.9 Versiegelte Methoden und Klassen

Gelegentlich möchte man das Verdecken¹ einer Methode *verhindern*, damit sich der Nutzer einer Unterklasse darauf verlassen kann, dass dort die geerbte Methode *nicht* überschrieben worden ist. Dient etwa die Methode `Passwd()` einer Klasse `ACL` zum Abfragen eines Passwortes, will ihr Programmierer eventuell verhindern, dass `Passwd()` in einer von `ACL` abstammenden Klasse `BCl` überschrieben wird. Damit kann dem Nutzer der Klasse `BCl` die ursprüngliche Funktionalität von `Passwd()` garantiert werden.

Um das Verdecken 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 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.

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.

6.10 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();
    }
}
```

¹ Das Überschreiben einer Methode ist leicht zu verhindern, weil es eine explizite Erlaubnis durch den Modifikator **virtual** voraussetzt.

2) Im folgenden Beispiel wird die Klasse `Kreis` aus der Klasse `Figur` abgeleitet:

```
class Figur {
    double xpos, ypos;
}
class Kreis : Figur {
    double radius;
    public Kreis(double x, double y, double rad) {
        xpos = x;
        ypos = y;
        radius = rad;
    }
    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 Programmier Techniken ist bei statischen Methoden *nicht* anwendbar?

7 Generische Typen und Methoden

Nach der Vererbung lernen Sie jetzt noch eine weitere Technik zur intelligenten Mehrfachverwendung von Quellcode (ohne Copy & Paste) kennen.

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.

Die im Beispiel ausgenutzte Typflexibilität ist *nicht* wünschenswert, wenn ein Container (dynamischer Array) für Variablen bzw. Objekte eines bestimmten (*festen*) Typs gefragt ist (z.B. zur Verwaltung von **String**-Objekten):

- Wenn beliebige Objekte zugelassen sind, die intern über Referenzvariablen vom Typ **Object** verwaltet werden, kann der Compiler keine Typsicherheit garantieren. Viele Programmierfehler werden also erst zur Laufzeit (womöglich vom Benutzer) entdeckt.
- Wenn Variablen mit Werttyp verwaltet werden, resultieren leistungsschädliche (Un)Boxing-Operationen.
- Entnommene Objekte können erst nach einer expliziten Typumwandlung die Methoden ihrer Klasse ausführen.

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
```

Weil der dritte **Add()**-Aufruf einen Wert vom Typ **char** per (Autoboxing!) in den Container befördert, endet das Programm mit einem 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, einen speziellen Container zur Verwaltung von **String**-Objekten zu erstellen, um die beiden Probleme (syntaktische Umständlichkeit, mangelnde Typsicherheit) zu vermeiden. Vermutlich werden analoge funktionierende Behälter aber auch für alternative Elementklassen benötigt, und entsprechend viele strukturgleiche Klassen zu definieren, die sich nur durch den Inhalts-

typ unterscheiden, ist nicht rationell. Für eine solche Aufgabenstellung bietet C# seit der Version 2.0 die generischen Klassen. Durch Verwendung von Typparametern wird die gesamte Handlungskompetenz der Klasse typunabhängig formuliert. Bei jeder Instantiierung wird der Typ konkretisiert, so dass Typsicherheit und syntaktische Eleganz resultieren.

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 Objekts ist an Stelle des Typparameters **T** ein konkreter Datentyp anzugeben:

```
List<String> gl = new List<String>();
```

und der Compiler verhindert die Aufnahme von Elementen mit abweichendem Typ:

```
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 Elemente *unterschiedlichen* Typs 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 Delegates. Wir befassen uns in Abschnitt 7 hauptsächlich mit generischen Klassen und Strukturen, machen aber auch schon erste Erfahrungen mit generischen Schnittstellen, die wir 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) {
        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 bei `EinfachStapel`-Objekten mit numerischem Typaktualparameter.

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 (aus der Sicht des Benutzers) zu realisieren, kann man den intern zur Datenspeicherung benutzten Array in leistungsoptimierend geplanten Stufen durch ein größeres Exemplar ersetzen, z.B. mit jeweils doppelter Länge. Dabei

sind die bisherigen Elemente zu kopieren. Eventuell haben die FCL-Designer Lösungen gefunden, den Aufwand bei der „Verlängerung“ zu reduzieren.

Als Beispiel für eine generische Klasse mit zwei Typformalparametern findet sich in der FCL die Klasse **Dictionary**, die eine Tabelle mit Name-Wert - Paaren verwaltet:

```
public class Dictionary<TKey, TValue>
```

Wie bei generischen Klassen (siehe Abschnitt 7) sind auch bei generischen Schnittstellen Restriktionen für die Typformalparameter möglich.

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 5.4.1.3.2 haben Sie erfahren, dass die Klasse **String** eine solche Methode besitzt, und wie **CompareTo()** das Prüfergebnis über den Rückgabewert signalisiert. 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, nicht-generische 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 eine generische Klasse zur Verwaltung einer Liste, die eingefügte Elemente automatisch einsortiert und daher ihren Typformalparameter auf den Schnittstellentyp **IComparable<T>** einschränkt:

```
using System;
class SimpleSortedList<T> where T : IComparable<T> {
    T[] elements;
    int firstFree;

    public SimpleSortedList(int len) {
        if (len > 0)
            elements = new T[len];
    }
}
```



```

public void Add(T element) {
    if (firstFree == elements.Length)
        return;
    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++;
}

public bool Get(int index, out T value) {
    if (index >= 0 && index < firstFree) {
        value = elements[index];
        return true;
    } else {
        value = default(T);
        return false;
    }
}
}

```

Bei der Formulierung von Einschränkungen (engl.: *constraints*) 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 angeben, die durch Kommata abzugrenzen 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
<pre> 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); } } </pre>	<pre> 1 2 4 11 </pre>

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 Derived : BaseClass {
    . . .
}
```
- eine konkretisierte generische Klasse

```
class Derived : GenBase<int, double> {
    . . .
}
```
- eine generische Klasse mit kompatiblen Typformalparametern (siehe unten)

Durch das Konkretisieren von Typparametern ändert sich die Basisklasse nicht, z.B.:

Quellcode	Ausgabe
<pre>using System; class BaseClass { } class GenDerived<T> : BaseClass { } class Prog { static void Main() { Console.WriteLine((new GenDerived<int>()).GetType().BaseType); Console.Read(); } }</pre>	BaseClass

Soll eine generische Basisklasse beerbt werden, müssen ihre Typformalparameter entweder konkretisiert oder auf Typformalparameter der ebenfalls generischen abgeleiteten Klasse gesetzt werden, wobei Typrestriktionen der Basisklasse ggf. zu wiederholen sind, z.B.:

```
class GenBase<T1, T2> where T2 : IComparable<T2> {
    . . .
}
class Derived : GenBase<int, double> {
    . . .
}
class GenDerived<T> : GenBase<int, T> where T : IComparable<T> {
    . . .
}
```

Im letzten 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. Im Zusammenhang mit den generischen Schnittstellen werden wir aber doch auf ein Design-Problem im Zusammenhang mit der Vererbung stoßen (vgl. Abschnitt 8.2).

7.3 Nullable<T> als Beispiel für generische Strukturen

In diesem Abschnitt wird vor allem 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.

In Abschnitt 7.2.1 haben wir einen **EinfachStapel<T>** als Beispiel für eine generische Klasse definiert und dabei eine Methode zum Abheben entworfen, die das oberste Element per **out**-Parameter abliefert und per Rückgabewert meldet, ob überhaupt ein Element vorhanden war:

```
public class EinfachStapel<T> {
    . . .
    public bool Abheben(out T element) {
        if (aktHoehe > 0) {
            element = daten[--aktHoehe];
            return true;
        } else {
            element = default(T);
            return false;
        }
    }
    . . .
}
```

Die Methode **Abheben()** wäre etwas leichter zu verwenden, wenn sie auf einen Parameter verzichten und als Rückgabe das oberste Stapелеlement oder den Ausnahmewert **null** abliefern würde. Bei unrestringiertem Typparameter muss aber **null** durch **default(T)** ersetzt werden, so dass bei einem numerischen Werttyp die numerische Null abgeliefert wird. Dieses Verhalten ist riskant, weil man eine Ausnahmenull nicht von einer Stapelnul unterscheiden kann.

Seit .NET 2.0 lässt sich aber jede Struktur in eine Konkretisierung der generischen Struktur **Nullable<T>** verpacken, so dass z.B. einer **Nullable<double>** - Instanz der Wert **null** zugewiesen werden kann. Wie der obige Quellcode zeigt, beschränkt sich der zusätzliche Speicherplatzbedarf einer **null**-fähigen Strukturinstanz im Vergleich zum Grundtyp auf eine **bool**-Variable:

```
private bool hasValue;
internal T value;
```

Man behält den Performanzvorteil von Strukturen, wenn in einer zeitkritischen Programmphase zahlreiche Instanzen eines Typs benötigt werden.

¹ Der Quellcode stammt aus Microsofts *Shared Source Common Language Infrastructure 2.0*.

Wir können in der generischen Klasse `EinfachStapel<T>` eine zusätzliche Überladung der Methode `Abheben()` mit dem vereinfachten Verhalten anbieten:

```
public T Abheben() {
    if (aktHoehe > 0) {
        return daten[--aktHoehe];
    } else {
        return default(T);
    }
}
```

Sie sollte nur zusammen mit einem **null**-fähigen Elementtyp verwendet werden, z.B. bei der `EinfachStapel`-Konkretisierung

```
EinfachStapel<Nullable<double>> ds = new EinfachStapel<Nullable<double>>(3);
```

Die **null**-fähige Variante zu einem Strukturtyp lässt sich auch durch den Namen des Grundtyps und ein angehängtes Fragezeichen ausdrücken, z.B.:

```
EinfachStapel<double?> ds = new EinfachStapel<double?>(3);
```

Weil der Grundtyp **double** implizit (automatisch) in den Typ **double?** konvertiert wird, kann der Methode `Auflegen()` ein **double** - oder ein **double?** - Wert übergeben werden, z.B.:

```
double? d = 77.7;
ds.Auflegen(3.141); ds.Auflegen(d);
```

Beim `Abheben()` ist wegen des potentiell zu erwartenden **null**-Werts jedoch eine **Nullable<double>**-Instanz erforderlich:

```
double? d;
d = ds.Abheben();
```

Eine solche 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 (siehe Quellcode am Beginn des aktuellen Abschnitts).

Das folgende Programm zeigt einen kompletten Kurzeinsatz des Strukturtyps **Nullable<double>** als `EinfachStapel<T>` - Elementtyp:

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { EinfachStapel<double?> ds=new EinfachStapel<double?>(3); ds.Auflegen(3.141); ds.Auflegen(2.718); double? d; for (int i = 1; i <= 3; i++) { d = ds.Abheben(); if (d != null) Console.WriteLine("Oben lag: " + d); else Console.WriteLine("Stapel war leer." + "\n d.HasValue = " + d.HasValue); } } }</pre>	<pre>Oben lag: 2,718 Oben lag: 3,141 Stapel war leer. d.HasValue = False</pre>

Während der Grundtyp implizit in den zugehörigen **Nullable**-Typ konvertiert wird, ist für den umgekehrten Übergang eine *explizite* Konvertierung erforderlich, z.B.:

```
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
```

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.:

Quellcodesegment	Ausgabe
<pre>int? ni = 4; int i = ni ?? 13; Console.WriteLine(i);</pre>	4

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.:

Quellcodesegment	Ausgabe
<pre>double? d = 3.0; Console.WriteLine(d.GetType());</pre>	System.Double

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:

Quellcode	Ausgabe
<pre>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)); } }</pre>	int-max: 13 double-max: 47,11

Man benennt die **Typformalparameter** einer generischen Methode hinter dem Methodennamen zwischen spitzen Klammern und ggf. voneinander durch Kommata getrennt. Sie sind als Datenty-

pen für den Rückgabewert, Parameter und lokale Variablen erlaubt. Wie bei generischen Klassen kann man Restriktionen für Typparameter formulieren.

Bei generischen Methoden sind selbstverständlich Überladungen erlaubt, auch unter Beteiligung von gewöhnlichen Methoden, z.B.:

Quellcode	Ausgabe
<pre>using System; class Prog { static T Max<T>(T x, T y) where T : IComparable<T> { return x.CompareTo(y) > 0 ? x : y; } static int Max(int x, int y) { Console.Write("trad. "); return x > y ? x : y; } public static void Main() { Console.WriteLine("int-max:\t" + Max(12, 13)); Console.WriteLine("double-max:\t" + Max(2.16, 47.11)); Console.Read(); } }</pre>	<pre>trad. int-max: 13 double-max: 47.11</pre>

Der Compiler ermittelt zu einem konkreten Aufruf die am besten passende kompatible Methode und beschwert sich bei Zweifelsfällen.

7.5 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**:

```
public sealed class String : IComparable, ICloneable, IConvertible, IComparable<string>,
    IEnumerable<string>, 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 Mitgliedern (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. Wie bei den generischen *Klassen* gewinnt man damit Typsicherheit zur Übersetzungszeit und spart in manchen Situationen (Un)boxing-Operationen.

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
<pre>using System; class Prog { static void Main() { String[] star = {"eins", "zwei", "drei"}; Array.Sort(star); foreach (String s in star) Console.WriteLine(s); } }</pre>	<pre>drei eins zwei</pre>

- **ICloneable**

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 eine explizite Typumwandlung erforderlich, z.B.:

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { String s1 = "eins"; String s2 = (String) s1.Clone(); Console.WriteLine(s2); } }</pre>	<pre>eins</pre>

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, Indexer und Ereignisse (siehe unten) ohne Anweisungsteil. Man kann sie 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 Abstraktionsmittel sehr nützlich. Sie dürfen auf Instanzen beliebiger Typen zeigen, welche die Schnittstelle implementieren. Somit können Instanzen unabhängig von den Vererbungsbeziehungen ihrer Typen gemeinsam verwaltet werden (z.B. in einem Array), wobei Methodenaufrufe polymorph erfolgen (mit später bzw. dynamischer Bindung).

Implementiert eine Klasse oder Struktur ein Interface, dann ...

- muss sie die im Interface enthaltenen Methoden implementieren,
- werden Variablen vom Typ dieser Klasse 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 ohne überflüssige Einengung erreicht.

- 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 Softwaresystems, 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 behandeln zuerst das im Programmieralltag vergleichsweise 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:

- Interface-Modifikator **public**
Wird **public** *nicht* angegeben, ist die Schnittstelle nur innerhalb ihres Assemblies verwendbar (Schutzstufe **internal**).
- Interface-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.
- Schnittstellename
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, Konstanten 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 bei Klassen zu verwenden ist. In der FCL wird z.B. das Interface **IEnumerable** vom 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 Schnittstelle erfüllen, müssen die Interfaces im Kopf der Typdefinition aufgelistet werden. Man setzt hinter den Typbezeichner einen Doppelpunkt, gibt bei Klassen ggf. zunächst eine Basis-klassse 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 Beispiellassen besitzt. Sie implementiert das Interface **IComparable**, damit **Figur**-Kollektionen bequem sortiert werden können:

```

using System;
public class Figur : IComparable {
    string name = "unbenannt";
    double xpos, ypos;

    public Figur(string n, double x, double y) {
        name = n; xpos = x; ypos = y;
    }

    public Figur() {}

    public String Name {
        get { return name; }
    }

    public int CompareTo(object v) {
        Figur vergl = (Figur) v;
        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 Schnittstelle **IComparable** ist nur eine **public**-Methode namens **CompareTo()** mit einem Parameter vom Typ **Object** 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.

Um über den (per **IComparable**-Definition vorgeschriebenen) Parameter vom Typ **Object** die speziellen Member von **Figur**-Objekten ansprechen zu können, ist eine explizite Typanpassung erforderlich:

```
Figur vergl = (Figur) v;
```

Ist das bei einem konkreten Aufruf an die **Figur**-Implementation von **CompareTo()** übergebene Vergleichsobjekt *keine* **Figur**, dann führt die explizite Typanpassung zu einer **System.InvalidCastException** (siehe Abschnitt 8.3). Warum nicht die konkretisierte generische Schnittstelle **IComparable<Figur>** zum Einsatz kommt, wird später begründet.

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(3,14): error CS0536: Figur implementiert den Schnittstellenmember
System.IComparable.CompareTo(object) nicht. Figur.CompareTo(object) ist
statisch, nicht öffentlich oder hat den falschen Rückgabewert.
```

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 mühelos, z.B.:

Quellcode	Ausgabe
<pre>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); } }</pre>	<pre>A B C C B A</pre>

Auch Schnittstellen ändern nichts daran, dass für C# - Klassen eine Mehrfachvererbung ausgeschlossen ist. 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 ...
 - o wie ein **ITuner**
 - o 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 Unterklasse auch die Schnittstellen-Implementationen ihrer Basisklasse erbt. Wird z.B. die Klasse **Kreis** von der oben vorgestellten Klasse **Figur** abgeleitet, so übernimmt sie auch die Schnittstelle **IComparable**, und die statische **Sort()**-Methode der Klasse **Array** kann auch mit **Kreis**-Objekten arbeiten, z.B.:

Quellcode	Ausgabe
<pre>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); } }</pre>	<pre>A B C C B A</pre>

Im Zusammenhang mit diesen Erbschaftsangelegenheiten kommen wir zurück zur Frage, ob in der **Figur**-Definition nicht besser die konkretisierte generische Schnittstelle **IComparable<Figur>** implementiert werden sollte, z.B.:

```
using System;
public class Figur : IComparable<Figur> {
    . . .
    public int CompareTo(Figur vergl) {
        if (xpos < vergl.xpos)
            return -1;
        else
            if (xpos > vergl.xpos)
                return 1;
            return 0;
    }
}
```

Diese durchaus erwägenswerte Vorgehensweise hat leider auch zur Folge, dass der abgeleiteten Klasse **Kreis** die geerbte Methode **CompareTo(Figur)** vom Compiler nicht mehr als Implementierung der Schnittstelle **IComparable<Kreis>** anerkannt wird. Das folgende Programm wird kritiklos übersetzt,

```
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);
        Console.Read();
    }
}
```

läuft jedoch auf eine **InvalidOperationException**. Eine zur Verbesserung der Typüberwachung durch den Compiler gedachte Technik (Implementierung von **IComparable<Figur>** statt **IComparable**) hat also eine neue, recht subtile und vom Compiler unbemerkte Fehlerquelle geschaffen. Um die beim Ableiten verloren gegangene **IComparable<T>** - Implementierung bei **Kreis**-Objekten wieder verfügbar zu machen, kann man diese über **Figur**-Referenzen ansprechen, z.B.:

```
List<Figur> kl = new List<Figur>();
```

An dieser Stelle zeigt sich, dass die generischen Typen und Schnittstellen nachträglich (mit der Version 2.0) in das .NET - Framework aufgenommen wurden und (noch) nicht perfekt mit der Vererbungstechnik harmonieren.

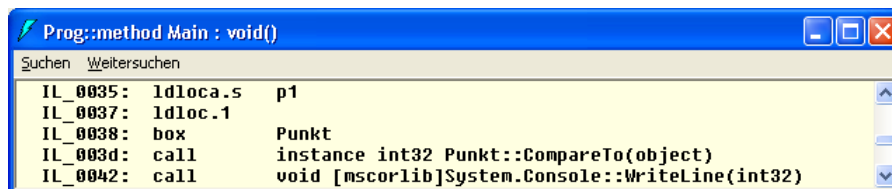
Bei einer (grundsätzlich nicht beerbbaren!) Struktur oder bei einer versiegelten Klasse entfällt das gerade diskutierte Problem der Schnittstellenimplementierungsvererbung, so dass hier nach Möglichkeit ein generisches Interface gegenüber der nicht-generischen Variante bevorzugt werden sollte, um Typsicherheit zur Übersetzungszeit zu gewinnen und (bei Strukturen) Boxing-Operationen einzusparen. 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:



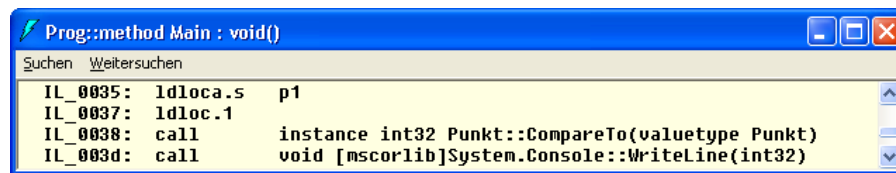
Implementiert die Struktur stattdessen das Interface **IComparable<Punkt>**,

```
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)); // Laufzeitfehler
```

und ein *gelungener* **CompareTo()** - Aufruf geht ohne Boxing über die Bühne:



8.3 Interfaces als Referenzdatentypen

Mit der Definition einer Schnittstelle wird ein neuer *Referenzdatentyp* vereinbart, der anschließend in Variablendeklarationen und Parameterlisten einsetzbar ist. Eine Referenzvariable des neuen Typs kann auf Instanzen einer implementierenden Klasse oder Struktur zeigen, z.B.:

Quellcode	Ausgabe
<pre>using System; 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>	K1 K2 S

Damit wird es möglich, Instanzen von beliebigen Klassen und Strukturen, die dasselbe Interface implementieren, in einem Array (oder einer anderen Kollektion) gemeinsam zu verwalten. Über eine Interface-Variable können die Methoden der Schnittstelle sowie die Methoden der Klasse **Object** aufgerufen werden.

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. Das gilt auch für konkretisierte generische Schnittstellen, z.B. für **IComparable<Punkt>** (vgl. Abschnitt 8.2):

```
IComparable<Punkt> ip = new Punkt(3.0, 3.0); //Boxing!
```

Weil Referenzvariablen vom Typ **IComparable<Punkt>** nur auf Objekte zeigen können, deren Typ das Interface **IComparable<Punkt>** implementiert, wird es vermutlich kaum Verwendungszwecke für diesen Datentyp geben.

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 Interface-Implementierungen

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 haben, für die aber unterschiedliche Funktionsweisen vorgesehen sind.

Für diese relativ seltene Situation bietet C# die so genannte *explizite* Schnittstellen-Implementation, 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.:

Quellcode	Ausgabe
<pre>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()); } }</pre>	<pre>13 4711</pre>

Über eine Referenzvariable von eigenem 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.

Richter (2006, S. 343) beschreibt, wie man durch explizite Implementation bei einer einzelnen nicht-generischen Schnittstelle für Compiler-garantierte Typsicherheit sorgen und Boxing-Operationen vermeiden kann. Derselbe Autor rät aber auch von übermäßigem Gebrauch der expliziten Implementation ab. Es erschwert die Verwendung einer Klasse, wenn die zu einer implementierten Schnittstelle gehörigen Methoden nur indirekt (über den Schnittstellentyp) verfügbar sind.

8.5 Übungsaufgaben zu Kapitel 8

1) Erstellen Sie eine Variante unserer **Bruch**-Klasse (vgl. z.B. Abschnitt 4.1.3), welche die Interfaces **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 abliefern 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?

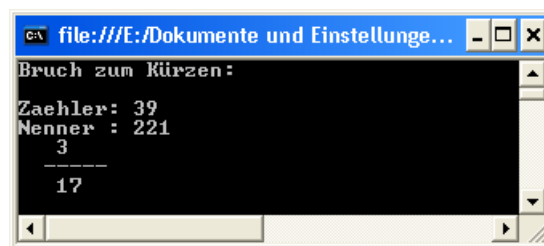
9 Einstieg in die WinForms - Programmierung

Mit den Eigenschaften und Vorteilen einer graphischen Benutzeroberfläche (engl.: *Graphical User Interface*) 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 **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.

Im Vergleich zu Konsolenprogrammen geht es nicht nur intuitiver, sondern vor allem auch ereignisreicher¹ 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, Eingabestift (bei Tablet PCs) 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 Ereignisbehandlungsmethoden 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 Ereignisbehandlungsmethoden 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“.

Betrachten wir zur Illustration eine Konsolen- und eine GUI-Anwendung zum Kürzen von Brüchen. Bei der Konsolenanwendung (vgl. Abschnitt 4.1.3)



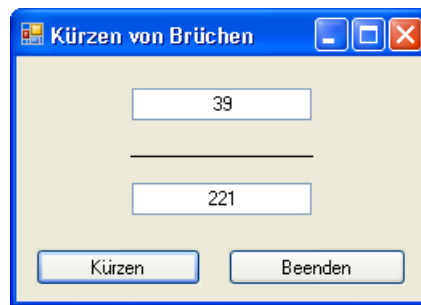
wird der gesamte Ablauf vom Programm diktiert:

- Es fragt nach dem Zähler.
- Es fragt nach dem Nenner.

¹ Momentan wird bewusst ein starker Kontrast zwischen den bisher 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 erstellen, das auf Ereignisse im Dateisystem (z.B. auf das Erstellen, Umbenennen oder Löschen von Dateien) reagiert.

- 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 Eingabefelder und zwei Schaltflächen) in beliebiger Reihenfolge verwenden kann, wobei das Programm mit seinen Ereignisbehandlungsmethoden reagiert (**benutzergesteuerter Ablauf**).

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. Formulardesigner) zusätzlich erleichtert wird. Neben dem traditionellen, sehr gut ausgestatteten und momentan noch zu bevorzugenden **Windows Forms - Framework** (kurz: *WinForms-Framework*) ist die mit .NET 3.0 vorgestellte und von Microsoft seitdem mit großem Aufwand als neue Grafikbasis entwickelte **Windows Presentation Foundation** (ehemals als *WinFX* bezeichnet) zu beachten. Als WPF - Vorteile werden u.a. attraktivere Bedienoberflächen mit 3D-Grafik und Animationen (Audio/Video) versprochen. Als Nachteile der neuen Technik sind derzeit die Beschränkung auf Betriebssysteme mit .NET 3.0 - Unterstützung (ab Windows XP mit SP2) und die im Vergleich zum WinForms - Framework fehlenden Steuerelemente (z.B. **DataGridView** zur Datenbankverwaltung) zu nennen. Microsoft hat zugesichert, die WinForms-Plattform noch viele Jahre zu unterstützen, so dass diese ausgereifte und vom Visual Studio 2008 sehr gut unterstützte Technik momentan zu bevorzugen ist, wenn 3D-Grafiken und Animationen keine Rolle spielen. Die folgende Tabelle aus einem Whitepaper von Microsoft¹ beschreibt die Leistungsumfänge verschiedener Techniken und zeigt deutlich die Zielvorgabe bei der WPF-Entwicklung:

	WinForms	PDF	WinForms + GDI	Windows Media Player	Direct3D	WPF
Forms, Controls	X		X			X
Complex text		X				X
Images			X			X
Video / Audio				X		X
2D Graphics			X			X
3D Graphics					X	X

Außerdem soll die WPF-Plattform ...

- die Kooperation zwischen Softwareentwicklern und Grafikdesignern erleichtern,
- den Unterschied zwischen Windows- und Webanwendungen (auf Silverlight-Basis) reduzieren.

Wir verwenden in diesem Manuskript das WinForms-Framework, weil es ...

¹ Im Internet zu finden über: <http://windowsclient.net/wpf/white-papers/when-to-adopt-wpf.aspx>

- eine höhere Produktivität bietet (geringerer Zeitaufwand für die Entwicklung)
- von den Entwicklungswerkzeugen besser unterstützt wird
- deutlich geringere Anforderungen an die Hard- und Software der Anwender stellt.

Diese Argumente finden sich bei Schwichtenberg (2009b) ebenso wie die Schlussfolgerung:

„Wer heute noch bewusst auf Windows Forms setzt, macht nichts falsch.“

In diesem Abschnitt soll ein grundlegendes Verständnis von Aufbau und Funktionsweise einer WinForms-Anwendung vermittelt werden. Dabei verzichten wir vorübergehend auf die Assistenten der Entwicklungsumgebungen, die einen relativ komplexen Quellcode produzieren, der den Blick auf das Wesentliche erschwert¹. Sobald wir die Assistentenprodukte verstehen können, steht ihrer Verwendung zur Vereinfachung von Routineaufgaben nichts im Wege.

9.1 Ein Projektrahmen zum Üben

Legen Sie mit der Visual C# Express Edition nach der Beschreibung in Abschnitt 2.2.2.2 ein leeres Projekt mit dem Namen **GuiEinstieg** sowie eine Quellcodedatei mit dem Namen **GuiEinstieg.cs** an. Anschließend präsentiert die Entwicklungsumgebung ein Übungsprojekt ohne eine einzige Zeile Assistenten-Quellcode.

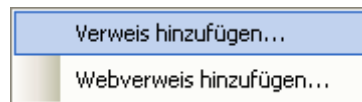
Weil wir eine WinForms-Anwendung planen (ohne eine entsprechende Projektvorlage verwendet zu haben), müssen wir folgende DLL-Assemblies in die Verweisliste des Projekts aufnehmen:

- **System.dll**
- **System.Windows.Forms.dll**

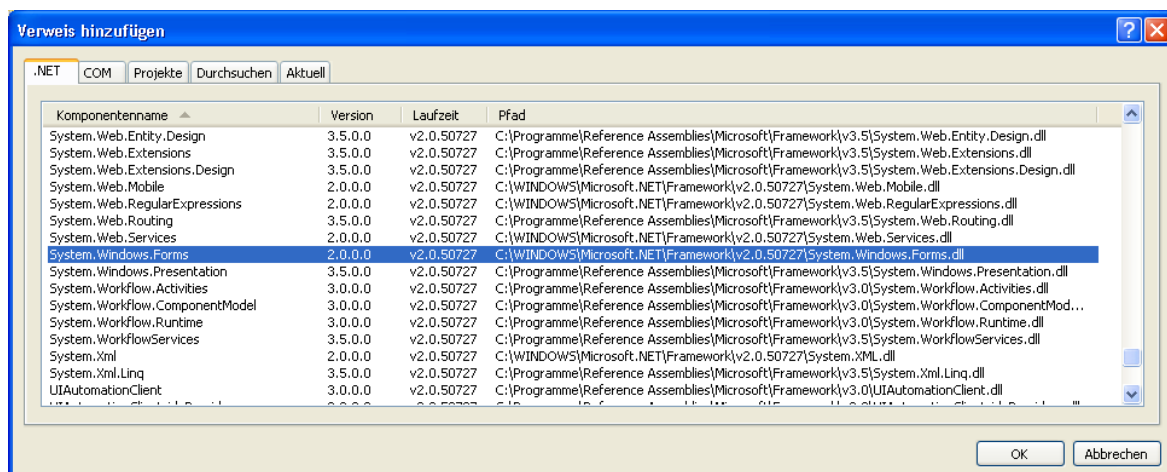
Spätere Beispielpprogramme benötigen meist auch eine Referenz auf

- **System.Drawing.dll**

Wählen Sie im **Projektmappen-Explorer** aus dem Kontextmenü zum Eintrag **Verweise** die Option **Verweise hinzufügen**:

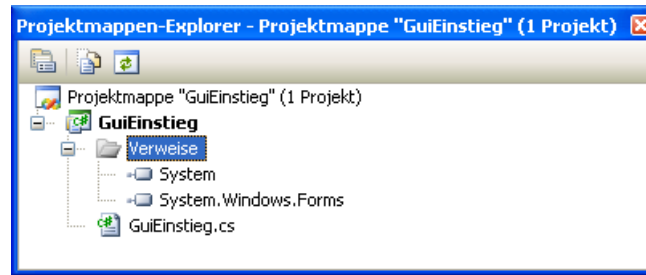


In der folgenden Dialogbox lassen sich die benötigten Verweise leicht auswählen:

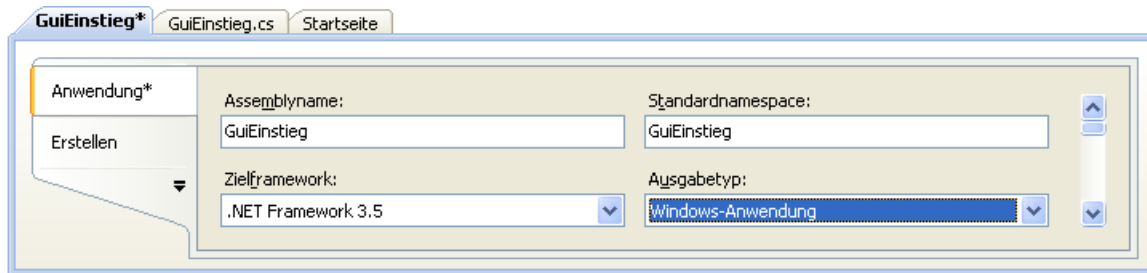


Das Ergebnis im Projektmappen-Explorer:

¹ Diese Vorgehensweise wird übrigens von Charles Petzold, einem exzellenten und sehr erfahrenen Autor von Büchern zur Windows-Programmierung, generell empfohlen. In seinem Buch zur Windows-Programmierung mit C# (Petzold 2002) schreibt er auf S. 45: „In diesem Buch werden Sie und ich den Code selbst schreiben“.



Um das Erscheinen eines Konsolenfensters beim Programmstart zu vermeiden, sollten Sie nach **Projekt > GuiEinstieg-Eigenschaften > Anwendung** den **Ausgabebetyp** auf **Windows-Anwendung** ändern:



9.2 Elementare Klassen im Windows Forms - Framework

Für ein minimalistisches WinForms-Programm in C# sind nur wenige Zeilen erforderlich:

```
using System.Windows.Forms;
class GuiEinstieg : Form {
    GuiEinstieg() {
        Text = "WinForms-Einstieg";
    }
    static void Main() {
        GuiEinstieg hf = new GuiEinstieg();
        Application.Run(hf);
    }
}
```

In diesem Beispielpogramm könnten sogar etliche Zeilen unter Verzicht auf Konsistenz mit späteren, komplexeren Programmen eingespart werden.

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 WinForms-Framework finden vermehrt Aktivitäten *hinter* der Bühne statt, die gleich erläutert werden sollen.

Allzu Funktionsumfang sollte man vom Einstiegsbeispiel 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.

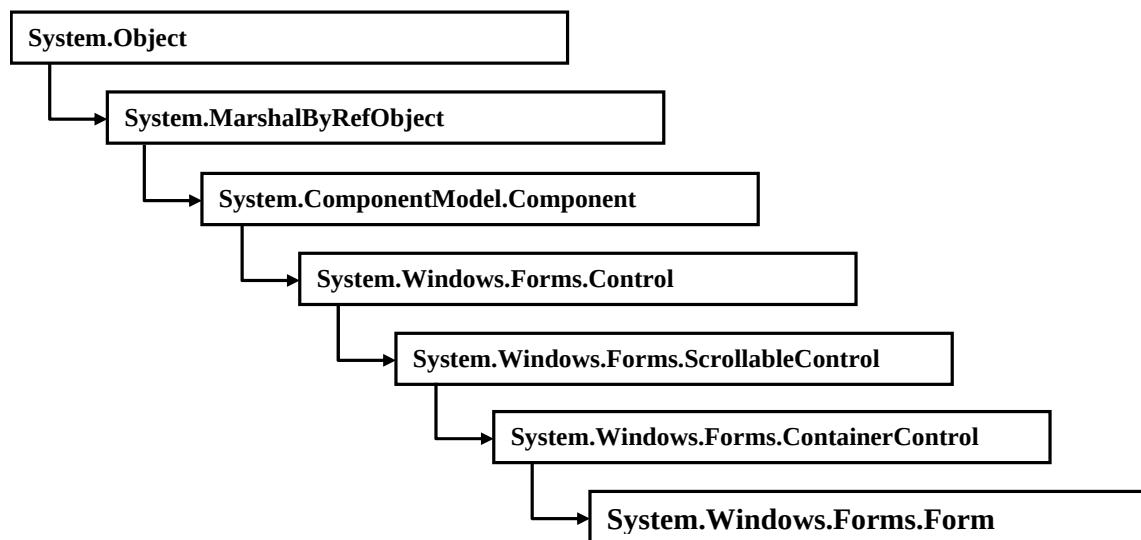
Aus dem sehr umfangreichen Namensraum **System.Windows.Forms** (mit mehreren hundert Klassen und sonstigen Typen), den man bei einer WinForms-Anwendung am besten per **using**-Direktive importiert, werden Sie im Abschnitt 9 folgende Klassen kennen lernen:

- **Form**
Aus dieser Klasse stammen alle Anwendungs- oder Dialogfenster ab, die auch als *Formulare* bezeichnet werden.
- **Application**
Diese Klasse bietet u.a. essentielle statische Methoden zum Betreiben eines WinForms-Programms (z.B. **Run()**, **Exit()**).
- Delegatenklassen (z.B. **EventHandler**)
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**)

Durch die **using**-Direktive in der ersten Zeile des Beispielprogramms wird dafür gesorgt, dass der Compiler dem Namen jeder Klasse, die nicht im projekteigenen Quellcode definiert ist, die Namensraumbezeichnung **System.Windows.Forms** voranstellt und dann die referenzierten Assemblies nach dem vervollständigten Namen durchsucht. Im Beispielprogramm geschieht dies bei den Klassen **Form** und **Application**.

9.2.1 Anwendungsfenster und die Klasse Form

Alle Formulare (Fenster) einer WinForms-Anwendung werden über Objekte einer von **Form** abstammenden Klasse verwaltet. **Form** erbt seine Funktionalität wiederum zum großen Teil von allgemeineren .NET – Klassen:



Drei Basisklassen sollen kurz skizziert werden:

- **System.Windows.Forms.Control** implementiert die für Bedienelemente der Benutzeroberfläche (z.B. Schaltflächen, Fenster) erforderlichen Methoden und Eigenschaften zur Bildschirmanzeige sowie zum Umgang mit Maus- und Tastaturereignissen, z.B.:
 - Über die Methode **Show()** veranlasst man ein Steuerelement, auf dem Bildschirm zu erscheinen.
 - Über die Eigenschaften **Height** bzw. **Width** lässt sich die Höhe bzw. Breite eines Steuerelements ermitteln und/oder setzen.
- In der Klasse **System.Windows.Forms.ScrollableControl** kommen Bildlaufleisten hinzu.
- Die Klasse **System.Windows.Forms.ContainerControl** ist leicht irreführend bezeichnet, weil *jedes Control-Objekt* untergeordnete Steuerelemente aufnehmen kann. Was die Klasse **ContainerControl** zusätzlich bietet, ist die Kompetenz zur *Fokusverwaltung* für enthaltene Steuerelemente (z.B. durch passende Reaktionen auf die Tabulatortaste).

Das obige Beispielprogramm besteht aus der von **Form** abgeleiteten Klasse **GuiEinstieg**

```
class GuiEinstieg : Form
```

und erzeugt in seiner **Main()**-Methode *ein* Objekt aus dieser Klasse (also *ein* entsprechendes Formular):

```
GuiEinstieg hf = new GuiEinstieg();
```

Als Aktualparameter im Methodenaufruf **Application.Run()**

```
Application.Run(hf);
```

wird das **GuiEinstieg**-Objekt **hf** zum Vertreter des **Anwendungs- oder Hauptfensters** im Programm. Unter den Fenstern eines Programms zeichnet sich das Hauptfenster durch folgende Besonderheiten aus:

- Der **Run()**-Aufruf sorgt u.a. für die Anzeige des Hauptfensters. Sein Auftritt muss also *nicht* über die **Control**-Methode **Show()** oder die **Control**-Eigenschaft **Visible** veranlasst werden.
- Beim Schließen des Hauptfensters kehrt der **Run()**-Aufruf zurück, so dass mit der **Main()**-Methode schließlich auch das Programm endet.
- In der Regel werden erhebliche Teile der Programm-Funktionalität im Hauptfenster angeboten.

So wie im Konstruktor des Beispielprogramms mit der Anweisung

```
Text = "WinForms-Einstieg";
```


die Titelzeilenbeschriftung des Fensters über die **Form**-Eigenschaft **Text** gewählt wird, sind zahlreiche weitere Eigenschaften dieser Klasse veränderbar, z.B.:

- Die Startgröße eines Formulars kann per **Size**-Eigenschaft (vom Typ **System.Drawing.Size**) oder über die Eigenschaften **Height** und **Width** (vom Typ **int**) geändert werden, z.B.:
`Height = 150; Width = 300;`
- Über die booleschen Eigenschaften **ControlBox**, **HelpBox**, **MaximizeBox** und **MinimizeBox** wählt man die Ausstattung der Titelzeile mit Standardschaltflächen, z.B.
`MaximizeBox = false; MinimizeBox = false;`
- Indem man der Eigenschaft **FormBorderStyle** einen Wert der Enumeration **FormBorderStyle** aus dem Namensraum **System.Windows.Forms** zuordnet, kann man das Design und/oder die Funktionalität eines Formulars beeinflussen. Z.B. führt der Wert **FixedSingle** zu einem Fenster mit fixierter Größe:
`FormBorderStyle = FormBorderStyle.FixedSingle;`
- Mit der Eigenschaft **BackColor** setzt man die Hintergrundfarbe des *Klientenbereichs* (Fensterfläche abzüglich Rahmen, Titelzeile, Menüzeile, Rollbalken und ggf. Symbolleisten), z.B.:
`BackColor = System.Drawing.Color.Beige;`

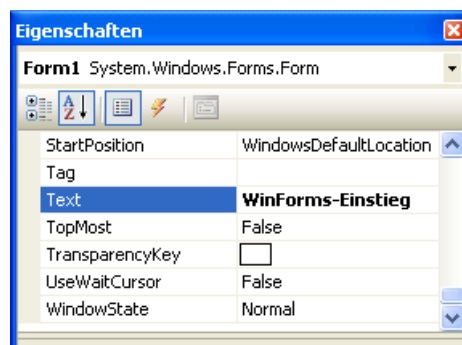
Man kann z.B. über statische Eigenschaften der Struktur **System.Drawing.Color** zwischen vordefinierten Standardfarben wählen oder mit der statischen **Color**-Methode **FromArgb()**, eine eigene Farbe kreieren. Wir werden uns in Abschnitt 15.5.1 ausführlich mit der .NET - Farbtechnik beschäftigen.

Beim äußerst simplen Einstiegsbeispiel hätten wir auf eine eigene, von **Form** abstammende GUI-Klasse verzichten und ein **Form**-Objekt als Hauptfenster verwenden können:

```
using System.Windows.Forms;
class GuiEinstieg {
    static void Main() {
        Form hf = new Form();
        hf.Text = "WinForms-Einstieg";
        Application.Run(hf);
    }
}
```

Die oben und in allen weiteren GUI-Programmen verwendete Architektur hat aber z.B. den Vorteil, dass wir das Hauptfenster im Konstruktor seiner Klasse initialisieren können, anstatt z.B. umständlich und mit kaum objektorientiertem Programmstil in der **Main()**-Methode. Außerdem werden wir später als **protected** deklarierte virtuelle Methoden der Klasse **Form** überschreiben, was nur in abgeleiteten Klassen möglich ist.

Beim Einsatz eines Formulardesigners erleichtert ein spezielles Fenster der Entwicklungsumgebung die Eigenschaftsmodifikation zur Entwurfszeit, und der zugehörige Quellcode wird automatisch geschrieben.



Wir verzichten momentan auf diesen Komfort, um die Grundstruktur einer WinForms-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.:



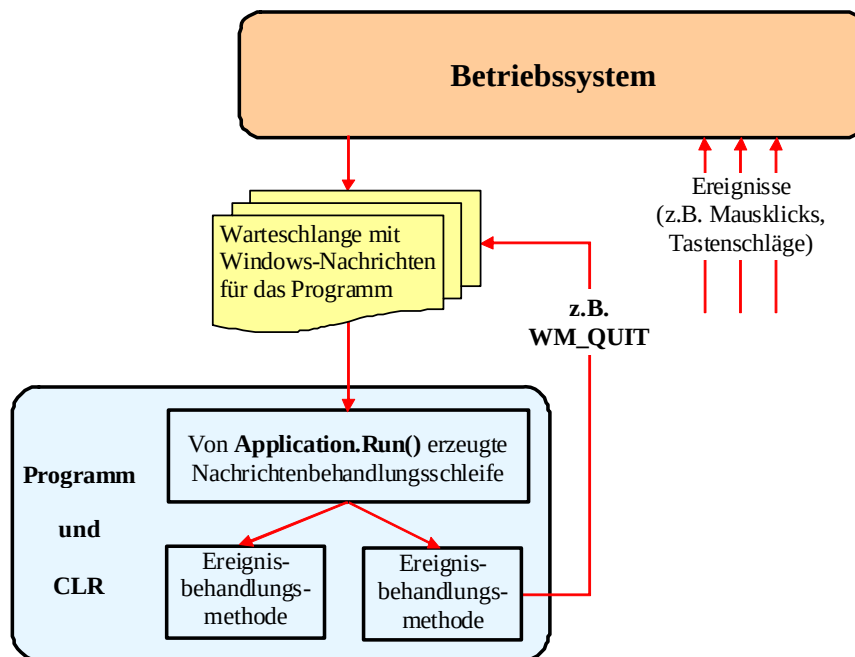
9.2.2 Windows-Nachrichten und die Klasse Application

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 WinForms-Anwendung eine per **Application.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 geschrieben haben.

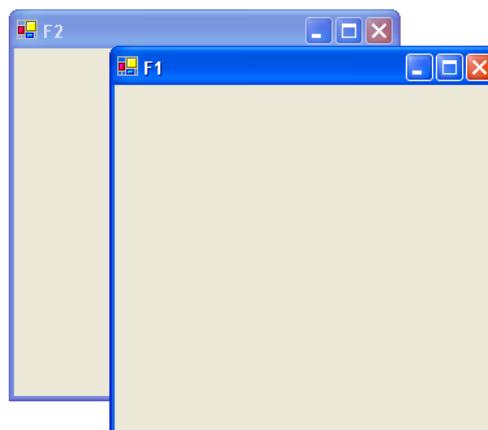
Die folgende Abbildung zeigt einige Details zum Nachrichtenverkehr zwischen dem Betriebssystem, der von **Application.Run()** initiierten Nachrichtenbehandlungsschleife und den Ereignisbehandlungsmethoden des Programms:



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 Methode **Application.Exit()** generieren. Sind beim Eintreffen der **WM_QUIT** – Nachricht noch Fenster einer Anwendung offen, so werden diese allesamt geschlossen. Beim Einsatz des folgenden Beispielsprogramms

```
using System;
using System.Windows.Forms;
class F1F2 : Form {
    F1F2(String titel) {
        Text = titel;
    }
    static void Main() {
        F1F2 p1 = new F1F2("F1");
        F1F2 p2 = new F1F2("F2");
        p2.Show();
        Application.Run(p1);
    }
}
```

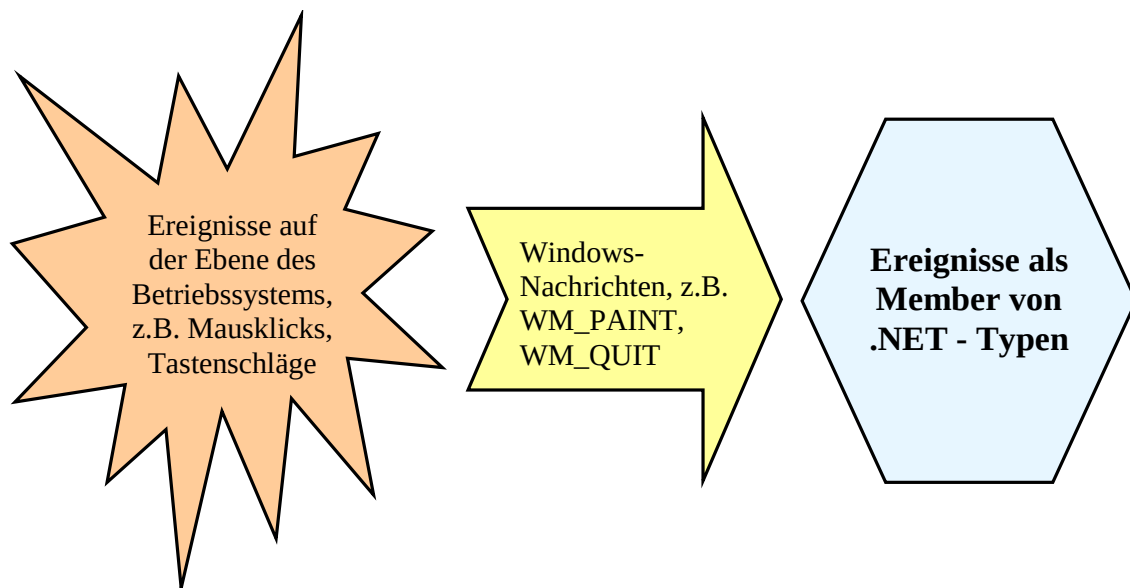
lässt sich das mit *F2* beschriftete Fenster *separat* beenden, während beim Schließen des mit *F1* betitelten Hauptfensters das Programm endet und damit auch das zweite Fenster verschwindet.



Fassen wir leicht vereinfachend zusammen, welche Bedeutung der Methodenaufruf **Application.Run()** in einer WinForms-Anwendung hat:

- Das **Hauptfenster** wird geöffnet.
- Das Programm¹ 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.

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 **ApplicationExit** 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) reagieren 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 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 einer bestimmten Signatur (definiert durch Rückgabotyp und Formalparameterliste) zeigen. Dazu enthalten Delegatenobjekte eine (ein- oder mehrelementige) Aufrufliste mit kompatiblen Methoden. Diese Liste kann Instanz- und Klassenmethoden enthalten.

¹ Genau genommen ist für jeden Thread (vgl. Abschnitt 13), der ein Fenster auf dem Bildschirm präsentiert, eine Nachrichtenwarteschlange und dementsprechend eine Nachrichtenschleife erforderlich.

Über ein Delegatenobjekt lassen sich mit *einem* Aufruf alle Methoden seiner Aufrufliste nacheinander starten, wobei die zuletzt ausgeführte Methode ggf. den Rückgabewert des Aufrufs liefert¹.

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);`

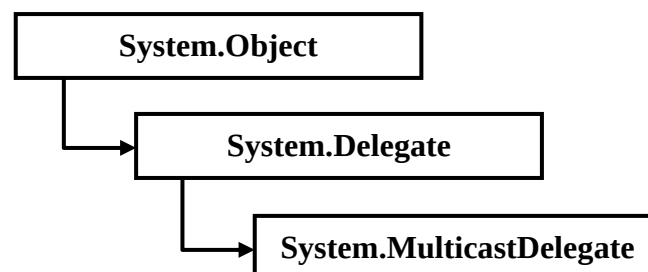
Ein **Delegatenobjekt** besitzt:

- einen Typ, z.B.
`DelType`
- eine veränderbare Aufrufliste mit kompatiblen 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 Konsolenanwendung.

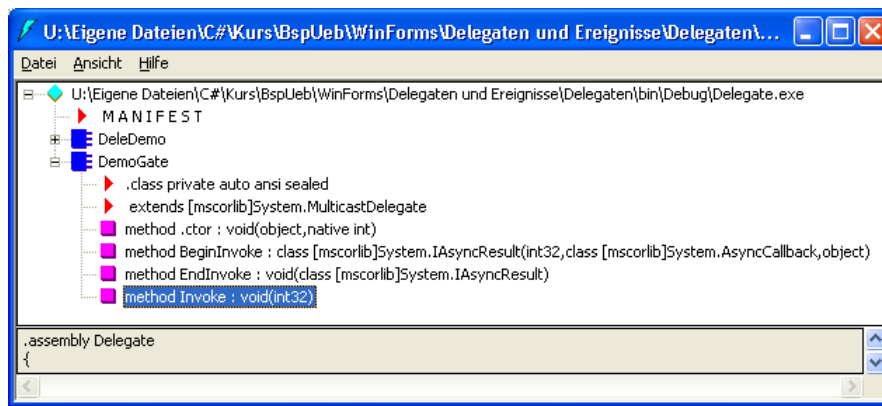
9.3.1.1 Delegatentypen definieren

Mit der folgenden Anweisung wird nach dem einleitenden Schlüsselwort **delegate** der Delegatentyp `DemoGate` definiert:

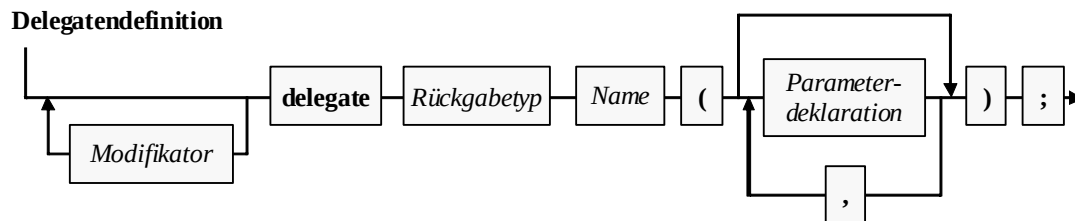
```
delegate void DemoGate(int w);
```

Über Objekte dieses Typs können Instanz- oder Klassenmethoden mit dem Rückgabotyp **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:

¹ Somit bietet C# mit den Delegaten eine objektorientierte Variante der Funktions-Pointer aus anderen Programmiersprachen (z.B. C++, Pascal, Modula).



Das leicht vereinfachte Syntaxdiagramm zur Definition eines Delegetentyps:



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 Delegetentyps besitzt,
- über ein Objekt dieses Delegetentyps erfolgen kann.

Die folgende Klasse

```
class DeleDemo {
    static void SagA(int w){
        for (int i = 1; i <= w; i++)
            Console.WriteLine('A');
    }
    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 Delegetentyp **DemoGate** (definiert in Abschnitt 9.3.1.1).
- In der **Main()**-Methode wird die lokale Referenzvariable **DemoVar** vom Delegetentyp **DemoGate** deklariert und initialisiert. Über den implizit definierten **DemoGate**-Konstruktor wird ein Objekt des Delegetentyps erzeugt, das auf die Methode **SagA()** zeigt. Die Adresse dieses Objekts landet in der Referenzvariablen:

```
DemoGate DemoVar = new DemoGate(SagA)
```

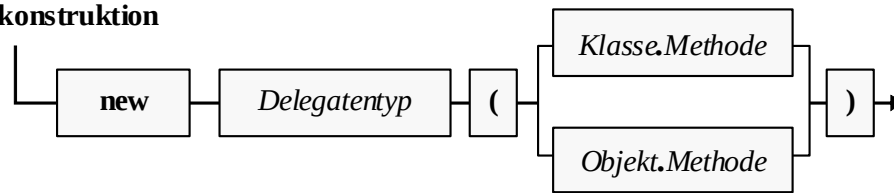
Einen parameterfreien **DemoGate**-Konstruktor gibt es übrigens *nicht*.

- In der Aufrufliste des über die Referenzvariable **DemoVar** ansprechbaren **DemoGate**-Objekts befindet sich ausschließlich die statische Methode **SagA()**. Ein Aufruf der Delegetenobjekts 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 (notigenfalls mit vorangestelltem Klassennamen) *oder* den Namen einer Instanzmethode (notigenfalls mit vorangestellter Objektreferenz):

Delegatenkonstruktion

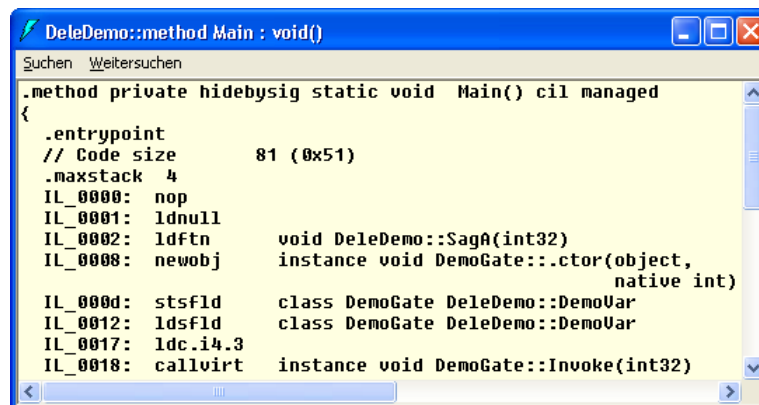


Dabei wird an den Methodennamen *keine* Parameterliste angehängt.

Eine Assembly-Inspektion mit dem Windows-SDK - Hilfsprogramm **ILDasm** zeigt, dass die Anweisungen

```
DemoVar = new DemoGate(SagA);
DemoVar(3);
```

in der DeLeDemo-Methode **Main()**



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 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 Delegatenobjektes 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>	<pre>AAA</pre>

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
<pre> 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); } } </pre>	AAA

Ein Nachteil anonymer Methoden besteht darin, dass sie im Unterschied zu benannten nicht an anderen Stellen benutzt werden können.

In C# 3.0 wurde zur Unterstützung der LINQ-Technik (siehe Kapitel 27) mit den so genannten *Lambda-Ausdrücken* eine zu den anonymen Methoden funktional äquivalente und dabei deutlich flexiblere Syntax eingeführt, die von Microsoft (2007, Abschnitt 7.14) 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>** 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); Console.Read(); } } </pre>	Anita Fritz Theo Anton

An Stelle der expliziten Delegatenobjektcreation

```
li.Sort(new Comparison<String>(Compar));
```

erlaubt der Compiler die Abkürzung:

```
li.Sort(Compar);
```

9.3.2 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.

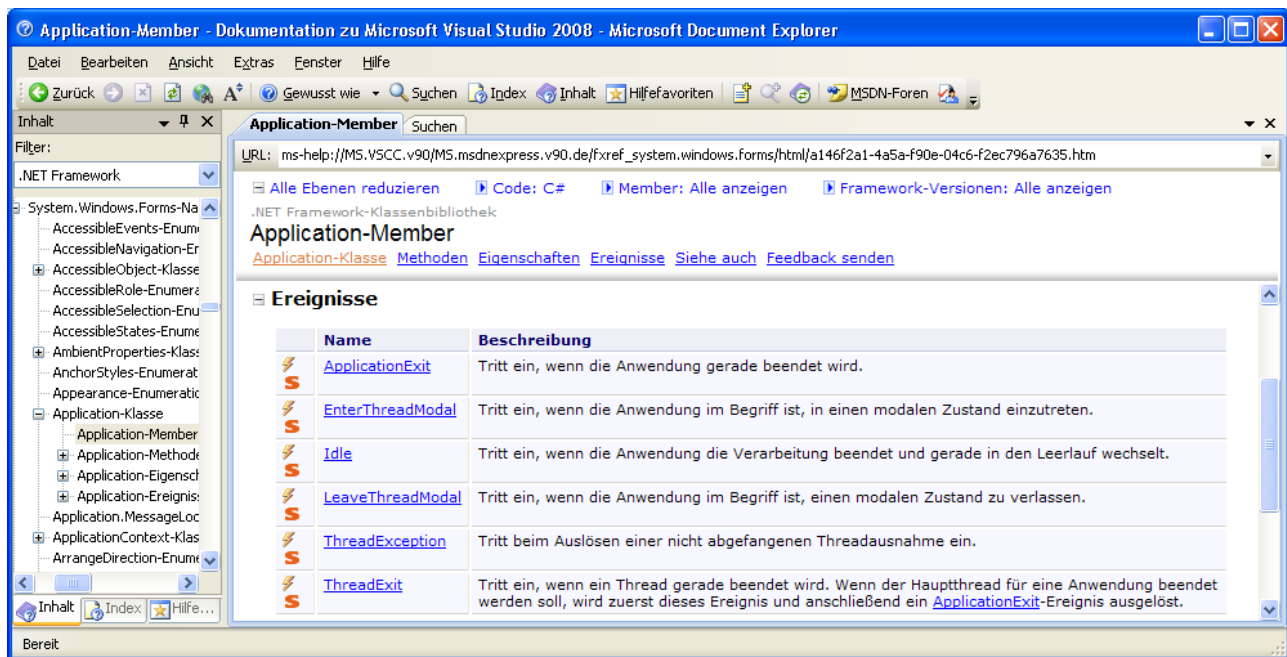
Ereignisse stellen im .NET – Framework ein wichtiges Kommunikationsmittel dar. Sie ermöglichen es einer Klasse oder einem Objekt, andere Akteure darüber zu informieren, dass etwas Bestimmtes passiert ist. So kann z.B. das Objekt zu einem Befehlsschalter registrierte Interessenten darüber informieren, dass der Benutzer den Schalter ausgelöst (angeklickt) hat. Das zugehörige Instanzergebnis der Klasse **System.Windows.Forms.Button** (siehe unten) heißt **Click**.

Obwohl Ereignisse in der Regel als **public** deklariert werden (siehe Abschnitt 9.3.2.2), resultiert in der veröffentlichenden Klasse stets eine *private* Delegatenvariable, der also fremde Methoden z.B. nicht den Wert **null** zuweisen können. Der Compiler ergänzt jedoch *öffentliche* Zugriffsmethoden zum Erweitern und Verkürzen der Aufrufliste durch fremde Klassen (siehe Beispiel in Abschnitt 9.3.2.2). 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

Bei sinnvollen Zugriffen kann ein Ereignis also von fremden Klassen wie eine öffentliche Delegatenvariable behandelt werden. Im Unterschied zu Delegatenvariablen dürfen Ereignisse (auch bei **public**-Deklaration) in klassenfremden Methoden aber ausschließlich als linkes Argument des „-=“ oder „+=“ – Operators auftreten. Dies ist keine Einschränkung, sondern eine sinnvolle Sicherheitsmaßnahme.

Die in Abschnitt 9.2.2 vorgestellte Klasse **Application** bietet z.B. mit dem Ereignis **Application-Exit** anderen Klassen die Möglichkeit, sich über das bevorstehende Programmende informieren zu lassen. In der FCL-Dokumentation sind die (allesamt statischen) Ereignisse der Klasse **Application** beschrieben:



9.3.2.1 Behandlungsmethoden registrieren

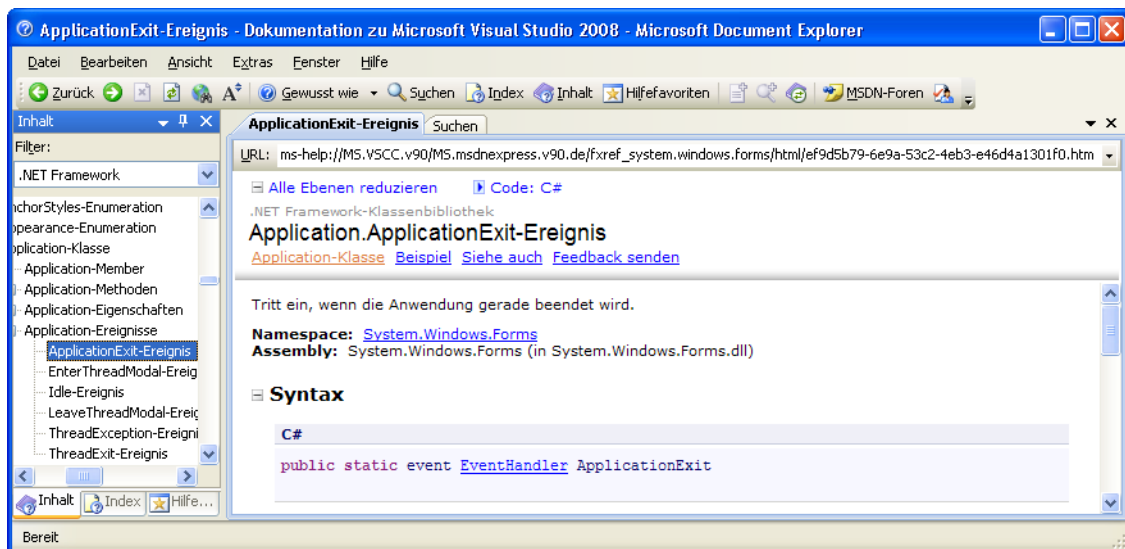
Um auf ein Ereignis einer .NET - Klasse reagieren zu können, ...

- 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.

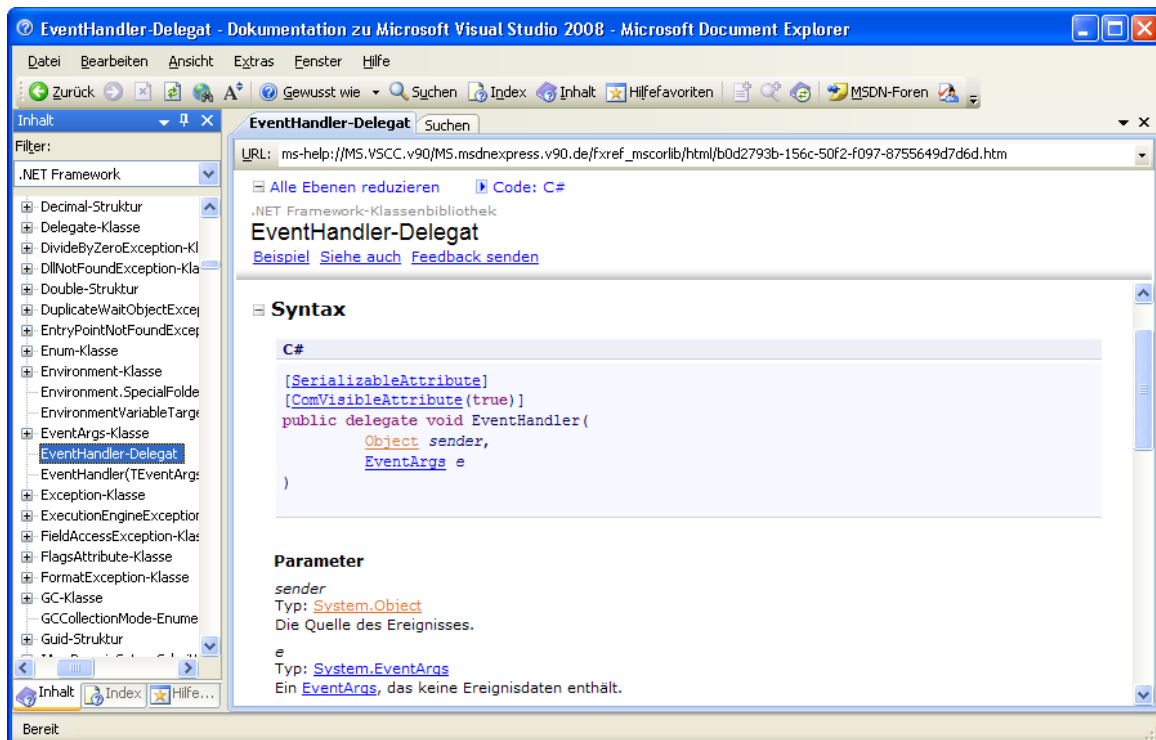
In der folgenden Variante des GUI-Einstiegsbeispiels aus Abschnitt 9.2 wird die Methode `ApplicationOnExit()` implementiert und beim Ereignis **ApplicationExit** der Klasse **Application** registriert, um auf das bevorstehende Programmende reagieren zu können:

```
using System;
using System.Windows.Forms;
class ApplicationExitDemo : Form {
    ApplicationExitDemo() {
        Text = "ApplicationExit-Ereignisbehandlung";
        Width = 340;
        Application.ApplicationExit += new EventHandler(ApplicationOnExit);
    }
    void ApplicationOnExit(object sender, EventArgs e) {
        MessageBox.Show("Vielen Dank für den Einsatz dieser Software!",
            "WinForms-Einstieg");
    }
    static void Main() {
        Application.Run(new ApplicationExitDemo());
    }
}
```

Zu welchem Delegatentyp eine Ereignisbehandlungsmethode kompatibel sein muss, erfährt man in der FCL-Dokumentation. Beim Ereignis **ApplicationExit** handelt es sich um den Typ **EventHandler** aus dem Namensraum **System**:



Nach einem Mausklick auf den Delegatentyp **EventHandler** ist bekannt, welchen Rückgabetyt und welche Parameter eine kompatible Methode benötigt, z.B.:



Die **EventHandler**-Signatur verlangt von kompatiblen Behandlungsmethoden zwei Parameter:

- Mit dem **Objekt**-Parameter **sender** wird die Ereignisquelle identifiziert. Es ist oft sinnvoll, eine Behandlungsmethode bei *mehreren* Ereignisquellen anzumelden, z.B. bei mehreren Befehlsschaltern. Weil der Methode beim Aufruf die Ereignisquelle im Parameter **sender** mitgeteilt wird, kann sie situationsgerecht reagieren. Den beim Ereignis **Application.ApplicationExit** registrierten Methoden wird allerdings der irrelevante **sender**-Wert **null** übergeben. Trotzdem muss natürlich in der Definition einer **ApplicationExit**-Behandlungsmethode der erste Parameter vom Typ **Object** sein.
- Behandlungsmethoden erhalten im Allgemeinen über den zweiten Parameter nähere Informationen zum Ereignis. Dabei wird meist ein Objekt aus der Klasse **System.EventArgs** oder aus einer abgeleiteten Klasse verwendet. Weil die Ereignisbeschreibungs-Basisklasse **EventArgs** keinerlei Instanz-Member enthält, sind ihre Objekte ziemlich informationsarm. Genau ein solches Dummy-Informationsobjekt aus der Klasse **EventArgs** wird den beim

Ereignis **Application.ApplicationExit** registrierten Methoden beim Aufruf übergeben. Trotzdem muss in der Definition einer **ApplicationExit**-Behandlungsmethode der zweite Parameter vom Typ **EventArgs** sein.

Die FCL-Designer haben auch beim Ereignis **ApplicationExit**, wo weder eine Absenderangabe noch eine Ereignisbeschreibung erforderlich ist, den Delegatentyp **EventHandler** gewählt. Sie halten sich damit konsequent an das von Microsoft vorgeschlagene *Entwurfsmuster zur Ereignisbehandlung*.

Im frei wählbaren Namen einer Behandlungsmethode nennt man in der Regel die Quelle (im Beispiel: Klasse **Application**) und das Ereignis (im Beispiel abgekürzt: **Exit**), häufig mit dem Verbindungswort **On**.

In der folgenden Anweisung wird ein **EventHandler**-Delegatenobjekt erzeugt, das auf die Methode `ApplicationOnExit()` zeigt:

```
Application.ApplicationExit += new EventHandler(this.ApplicationOnExit);
```

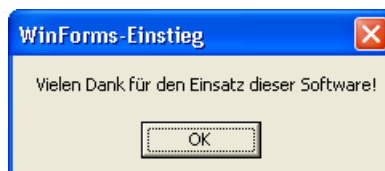
Weil es sich um eine Instanzmethode handelt, ist prinzipiell anzugeben, welches Objekt die Methode ausführen soll. Im Beispiel ist das gerade im Entstehen begriffene **ApplicationExitDemo**-Objekt gemeint, so dass die Objektreferenz entfallen oder mit dem Schlüsselwort **this** ausgedrückt werden kann.

Statt den beteiligten Delegatenkonstruktor explizit zu notieren, kann man die folgende Abkürzung verwenden:

```
Application.ApplicationExit += ApplicationOnExit;
```

Das neue **EventHandler**-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 `ApplicationOnExit()` z.B. zum Einsatz,



wenn der Benutzer auf das Schließkreuz in der Titelzeile des Anwendungsfensters klickt (vgl. Abschnitt 9.2.2). Dann wird nach einigen Zwischenschritten in einer Methode der Klasse **Application** das Ereignis **ApplicationExit** aufgerufen:¹

```
if (Application.ApplicationExit != null)
    Application.ApplicationExit(null, EventArgs.Empty);
```

Es ist übrigens bemerkenswert, dass die *private* Methode `ApplicationOnExit()` der Klasse **ApplicationExitDemo** durch eine Methode der fremden Klasse **Application** aufgerufen werden kann. Dies ist aber *keine* Aufweichung der strengen .NET- Zugriffskontrolle, weil die Registrierung

```
Application.ApplicationExit += ApplicationOnExit;
```

durch die berechtigte (klasseneigene) **Main()**-Methode vorgenommen wurde.

¹ Der Quellcode stammt aus dem Mono-Projekt.

9.3.2.2 Ereignisse anbieten

Das Registrieren eigener Behandlungsmethoden bei Ereignissen von FCL-Klassen ist für uns erheblich relevanter als das Anbieten von eigenen Ereignissen. 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 Sieben wird das Ereignis **Seven** ausgelöst. Weil zur Demonstration von ereignisorientierter Kommunikation ein GUI-Programm eindeutig am besten geeignet ist, wird ein Vorgriff auf den Abschnitt 9.4 über Befehlsschalter und sonstige Steuerelemente in Kauf genommen:

```
class EventProd : Button {
    public event MyDelegate Seven;
    Random zzg = new Random();
    protected override void OnClick(EventArgs e) {
        base.OnClick(e);
        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.

Im Unterschied zu einer Variablendeklaration ist bei Ereignissen das Schlüsselwort **event** anzugeben:

```
public 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.

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;
```

Die folgende WinForms-Anwendung enthält einen Schalter aus der Klasse EventProd auf Ihrem Formular:

```
class EventConsumer : Form {
    EventConsumer() {
        Height = 75; Width = 266;
        Text = "Schalter sendet Ereignisse";
        EventProd ep = new EventProd();
        ep.Text = "Teste Dein Glück!";
        ep.Top = 10;
        ep.Left = 30;
        ep.Width = 200;
        Controls.Add(ep);
        ep.Seven += new MyDelegate(this.EventProdOnSeven);
    }
    void EventProdOnSeven(object sender, MyEventArgs e) {
        MessageBox.Show("7 an Position " + e.Pos + " gezogen",
            "Ereignis-Verarbeitung");
    }
    static void Main() {
        Application.Run(new EventConsumer());
    }
}
```

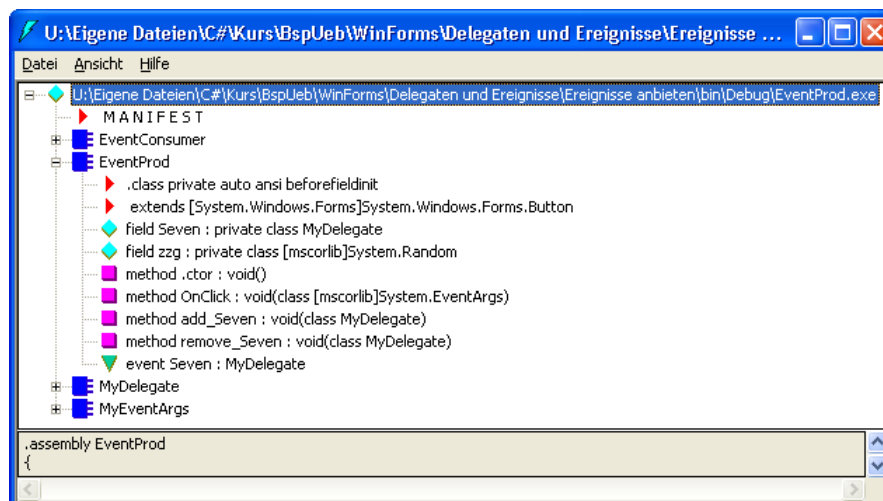
Um auf das Ereignis Seven reagieren zu können, implementiert die Klasse EventConsumer eine Methode mit kompatibler Signatur und registriert diese beim Ereignis:

```
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.:



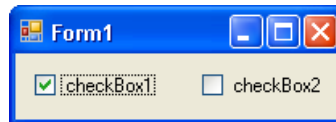
Eine Assembly-Inspektion mit dem Windows-SDK - Hilfsprogramm **ILDasm** bestätigt die zu Beginn von Abschnitt 9.3.2 formulierte Bemerkung über die Ereignis-Innenarchitektur. In der Klasse EventProd befinden sich aufgrund der Seven-Ereignisdefinition neben einem privaten Feld vom Typ MyDelegate zwei öffentliche Methoden zur Veränderung der Ereignisaufrufliste (add_Seven() und remove_Seven()):



9.4 Steuerelemente

Das WinForms-Framework 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 Formular einbauen, erscheint bzw. verschwindet bei einem Mausklick des Benutzers die Markierung,

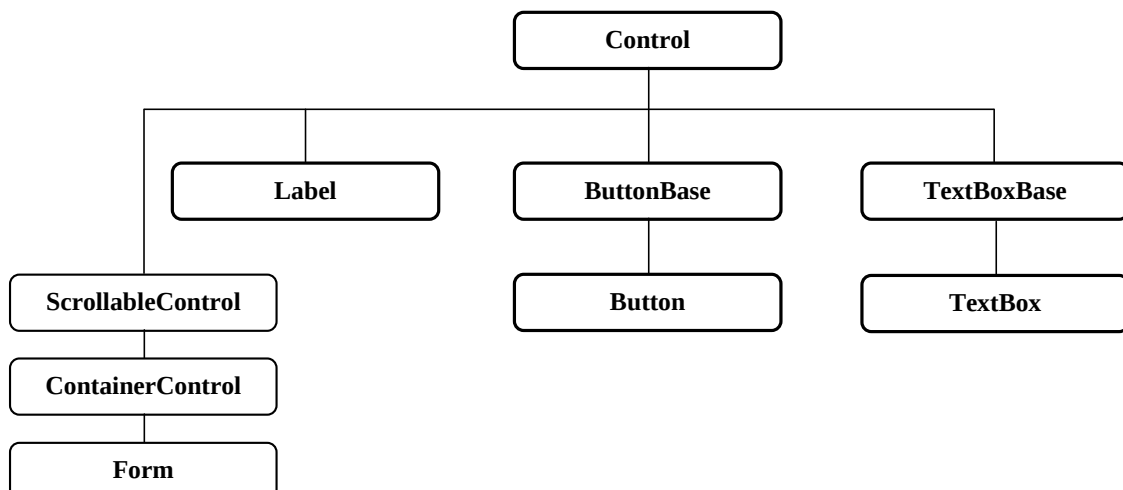


ohne dass wir uns um diese Anpassung der Optik kümmern müssten.

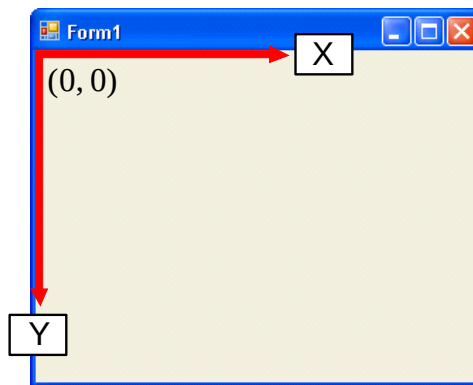
- Die Steuerelemente kommunizieren über **Ereignisse** (im Sinn von Abschnitt 9.3.2) mit anderen Klassen. Will man z.B. über den Markierungswechsel bei einem Kontrollkästchen informiert werden, registriert man eine Behandlungsmethode bei seinem **CheckedChanged**-Ereignis.
- Ihre Eigenschaften (z.B. **Text**, **Height**, **Left**, **Font**, **TextAlign**) können zur Entwurfszeit über Werkzeuge der Entwicklungsumgebungen konfiguriert werden (siehe z.B. Abschnitt 4.10.4). Wir machen aus didaktischen Gründen von dieser Option zunächst nur wenig Gebrauch. In der Alltagsroutine trägt sie aber zur Produktivität der Software-Entwicklung bei.

In diesem Abschnitt werden einfache Steuerelemente vorgestellt und wichtige Fragen zur Kooperation zwischen einem Formular und den dort als *Kindfenster* enthaltenen Steuerelementen behandelt. In späteren Abschnitten werden zahlreiche weitere elementare oder komplexe Steuerelemente vorgestellt.

Neben der Hauptfenster-Basisklasse **Form** stammen auch zahlreiche Klassen für Steuerelemente von der Klasse **Control** im Namensraum **System.Windows.Forms** ab, z.B.:



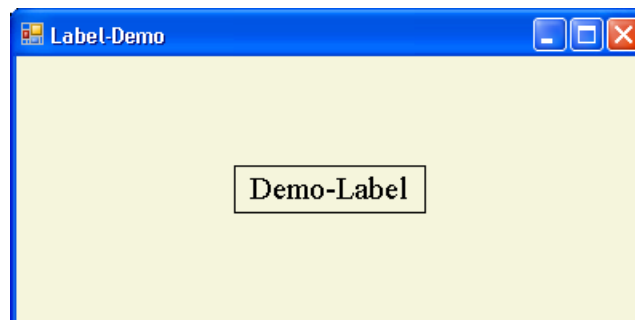
Bei den für Steuerelemente häufig erforderlichen Positionsangaben in Bildschirmpunkten (Pixeln) liegt das folgende Koordinatensystem mit dem Ursprung (Nullpunkt) in der linken oberen Ecke des Formular-Klientenbereichs zugrunde:



Wird ein Steuerelement allerdings nicht direkt in ein Formular, sondern in einen untergeordneten Container eingefügt, dann beziehen sich die Positionskoordinaten auf den Klientenbereich des Containers.

9.4.1 Einsatz von Steuerelementen am Beispiel des Labels

Wir betrachten zunächst das **Label**-Steuerelement zur Anzeige von Texten und besprechen anhand eines einfachen Beispielsprogramms,



wie man ...

- ein Steuerelement erzeugt und konfiguriert,
- in ein Formular (allgemein: in den übergeordneten Container) aufnimmt
- und mit einer Behandlungsmethode auf Ereignisse eines Steuerelements reagiert.

Dazu wird die aus **Form** abgeleitete Klasse `LabelDemo`

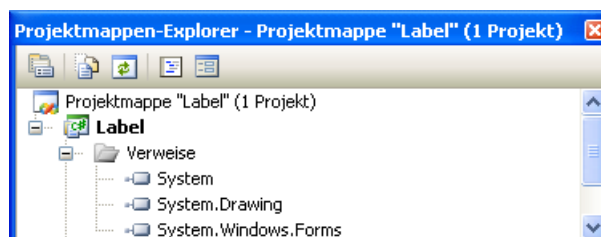
```
class LabelDemo : Form {
    . . .
}
```

sukzessiv vorgestellt.

Wie bei den meisten WinForms-Anwendungen empfiehlt sich der Import der folgenden Namensräume:

```
using System;
using System.Windows.Forms;
using System.Drawing;
```

Dementsprechend benötigen wir Verweise auf die folgenden Assemblies:



9.4.1.1 Steuerelemente erzeugen und konfigurieren

Im Konstruktor der Formalklasse

```
LabelDemo() {
    Size = new Size(400, 200);
    BackColor = Color.LightGray;
    Text = "Label-Demo";

    Label info = new Label();
    info.Text = "Demo-Label";
    info.Size = new Size(120, 30);
    info.Left = (ClientSize.Width-info.Width)/2;
    info.Top = (ClientSize.Height-info.Height)/2;
    info.Font = new Font("Times New Roman", 15);
    info.TextAlign = ContentAlignment.MiddleCenter;
    info.BorderStyle = BorderStyle.FixedSingle;
    . . .
}
```

wird nach Anpassung der **Form**-Eigenschaften **Height**, **Width**, **BackColor** und **Text** eine **Label**-Instanzvariable deklariert und ein **Label**-Objekt erzeugt

```
Label info = new Label();
```

Anschließend werden Eigenschaften des **Label**-Objekts modifiziert:

- Mit der Eigenschaft **Text** legt man die Beschriftung des Labels fest. Während Benutzer diese Beschriftung nicht direkt beeinflussen können, sind einer Veränderung durch das Programm wenig Grenzen gesetzt.
- Eine der wenigen Grenzen für die Wahl einer (sinnvollen) **Label**-Beschriftung resultiert aus der aktuellen Größe des Steuerelements. Man kann sie per **Size**-Eigenschaft (vom Typ **System.Drawing.Size**, vgl. Abschnitt 15.4.2)

```
info.Size = new Size(120, 30);
```

oder (bei minimal höherem Schreibaufwand) über die Eigenschaften **Width** und **Height** (vom Typ **int**)

```
info.Width = 120;
info.Height = 30;
```

verändern. Alle genannten Eigenschaften sind schon in der Basisklasse **Control** definiert. Sind Breite und Höhe eines Steuerelements zu setzen, werden wir meist eine Instanz der **Size**-Struktur verwenden und so die zweifache Nennung des Referenzvariablennamens vermeiden.

- Mit den **Control**-Eigenschaften **Left** bzw. **Top** spricht man die Position der linken bzw. oberen Grenze eines Steuerelements bezogen auf den Klientenbereich des umgebenden Containers an. Im Beispiel werden diese Eigenschaften so aus der Größe des Formular-Klientenbereichs abgeleitet, dass ein horizontal und vertikal zentriertes Label resultiert.
- Steuerelemente übernehmen die Vorder- und Hintergrundfarbe sowie die Schriftart des umgebenden Containers.
- Um für ein Steuerelement eine alternative Schrift zu wählen, übergibt man seiner **Font**-Eigenschaft eine Referenz auf ein (notigenfalls neu zu erzeugendes) **Font**-Objekt, z.B.:

```
info.Font = new Font("Times New Roman", 15);
```

Wir werden uns später noch ausführlich mit Schriftarten beschäftigen.

- Über die Eigenschaft **TextAlign** bestimmt man die Textposition innerhalb der rechteckigen **Label**-Fläche, wobei die Werte des Aufzählungstyps **System.Drawing.ContentAlignment** zu verwenden sind.
- Ist ein Rahmen gewünscht, steht die **Label**-Eigenschaft **BorderStyle** bereit, wobei die Werte des Aufzählungstyps **System.Windows.Forms.BorderStyle** zu verwenden sind. Weil Labels üblicherweise zur Beschriftung anderer Steuerelemente dienen, haben sie per Voreinstellung *keinen* Rahmen.

Eine Formalklasse benötigt zu einem enthaltenen Steuerelement nur dann eine Instanzreferenzvariable, wenn das Steuerelement auch außerhalb des Konstruktors (also in anderen Methoden) angesprochen werden soll. Im gerade entstehenden Demonstrationsprogramm mit **Label**-Steuerelement ist dies nicht der Fall, so dass im Fensterkonstruktor eine *lokale* Referenzvariable erzeugt wird:

```
class LabelDemo : Form {
    LabelDemo() {
        Label info = new Label();
        . . .
    }
    . . .
}
```

In der alternativen Situation wäre z.B. die folgende Konstruktion angemessen:

```
class LabelDemo : Form {
    Label info;
    LabelDemo() {
        info = new Label();
        . . .
    }
    . . .
}
```

Eine überflüssige Instanzvariable erhöht das Risiko eines Programmierfehlers (z.B. durch eine ungeplante Steuerelement-Modifikation per Tippfehler).

9.4.1.2 Ab in den Container

Die Objekte der von **Control** abstammenden Klassen können Steuerelemente als Kindfenster aufnehmen, wobei jedoch erst in der Klasse **ContainerControl** die Fähigkeit zur Fokusverwaltung für untergeordnete Steuerelemente vorhanden ist. Ein Anwendungsfenster taugt als **Form**-Objekt in jedem Fall als funktionstüchtiger Container.

Die in einem Container enthaltenen Steuerelemente werden über eine Liste verwaltet, welche über die **Controls**-Eigenschaft des Containers anzusprechen ist. Es handelt sich um ein Objekt der Klasse **ControlCollection**, die innerhalb der Klasse **Control** definiert ist. Dieses Objekt beherrscht diverse Methoden zur Aufnahme oder zum Entfernen von Steuerelementen und erlaubt auch einen Indexzugriff auf die Elemente.

Mit der Methode **Add()** kann ein Steuerelement in die Liste aufgenommen werden, z.B.:

```
LabelDemo() {
    . . .
    Label info = new Label();
    . . .
    Controls.Add(info);
    . . .
}
```

Ein äquivalentes Aufnahmeverfahren besteht darin, eine Referenz auf den umgebenden Container in die **Parent**-Eigenschaft des Steuerelements zu schreiben, z.B.:

```
info.Parent = this;
```

Alle aufgenommenen Steuerelemente erscheinen im Klientenbereich des elterlichen Containers (Gesamtfläche ohne Rahmen, Titelzeile etc.), ihre Positionseigenschaften beziehen sich auf diese Fläche, und darüber hinaus ragende Steuerelementeile werden abgeschnitten.

Ein Steuerelement verwendet nicht nur die Klientenfläche des übergeordneten Containers, sondern übernimmt auch einige Eigenschaften (z.B. **Font**, **BackColor** und **ForeColor**), wobei diese Voreinstellungen natürlich durch individuelle Werte ersetzt werden können.

Ist die Neuanzeige eines Containers erforderlich, werden entsprechende Nachrichten automatisch an die Kindfenster propagiert. Weil die Steuerelemente aus der .NET – Standardbibliothek mit passenden Ereignisbehandlungsmethoden ausgestattet sind, müssen wir uns bei den im Abschnitt 9 vorgestellten GUI-Anwendungen um das außerordentlich wichtige und häufige **Paint**-Ereignis (definiert in der Klasse **Control**) nicht kümmern (vgl. Abschnitt 15.3).

Statt Bedienelemente direkt auf ein Formular zu setzen, werden bei komplexen Formularen auf dieser Ebene oft Container-Steuerelemente (z.B. aus den Klassen **Panel** oder **TableLayoutPanel**, siehe unten) eingesetzt, die dann wiederum Steuerelemente aufnehmen und räumlich zusammenfassen.

9.4.1.3 Ereignisbehandlungsmethoden

Nach Bedarf werden in der Formalklasse Behandlungsmethoden definiert und bei Ereignissen des Formulars oder der Steuerelemente registriert. Im Beispiel wird die `LabelDemo`-Instanzmethode `InfoOnClick()` definiert und beim **Click**-Ereignis des **Label**-Objekts `info` registriert:

```
LabelDemo() {
    info.Click += new EventHandler(InfoOnClick);
}

void InfoOnClick(object sender, EventArgs e) {
    if (BackColor == Color.Beige)
        BackColor = Color.LightGray;
    else
        BackColor = Color.Beige;
}
```

`InfoOnClick()` schaltet die Hintergrundfarbe des Formulars zwischen **Color.Beige** und **Color.LightGray** um, wobei die Farben über statische Eigenschaften der Struktur **Color** aus dem Namensraum **System.Drawing** angesprochen werden.

Beim Registrieren einer Ereignisbehandlungsmethode wird ...

- ein Objekt des geforderten Delegetentyps (im Beispiel: **EventHandler**) erzeugt,
- dem Ereignis (also letztlich einer Delegetenvariablen) die Adresse des neuen Delegetenobjekts zugewiesen.

Ist bereits ein Delegetenobjekt für das Ereignis vorhanden, entsteht ein neues Objekt mit einer entsprechend verlängerten Aufrufliste (vgl. Abschnitt 9.3).

Zur Benennung von Ereignisbehandlungsmethoden wird folgendes Schema vorgeschlagen:

- Am Anfang steht eine Bezeichnung für die Ereignisquelle. Ist eine Methode für alle Steuerelemente aus einer Klasse zuständig, bietet sich der Klassenname an. Ansonsten kommt z.B. der Instanzvariablenname eines einzelnen Steuerelements in Frage, z.B.:
 - o `LabelOnClick()`
 - o `InfoOnClick()`
- Es folgt das Wort **On** und schließlich der Ereignisname.

Der Quellcode zum Programm `LabelDemo` im Überblick:

```
using System;
using System.Windows.Forms;
using System.Drawing;

class LabelDemo : Form {
    LabelDemo() {
        Size = new Size(400, 200);
        BackColor = Color.LightGray;
        Text = "Label-Demo";
        Label info = new Label();
        info.Text = "Demo-Label";
        info.Size = new Size(120, 30);
        info.Left = (ClientSize.Width-info.Width)/2;
        info.Top = (ClientSize.Height-info.Height)/2;
        info.TextAlign = ContentAlignment.MiddleCenter;
        info.BorderStyle = BorderStyle.FixedSingle;
        Controls.Add(info);
        info.Click += new EventHandler(InfoOnClick);
    }

    void InfoOnClick(object sender, EventArgs e) {
        if (BackColor == Color.Beige)
            BackColor = Color.LightGray;
        else
            BackColor = Color.Beige;
    }

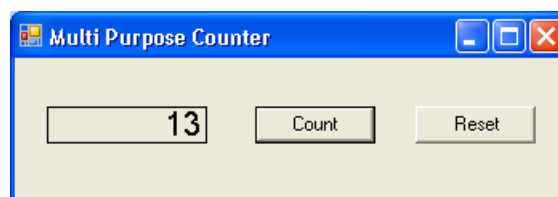
    static void Main() {
        Application.Run(new LabelDemo());
    }
}
```

9.4.2 Befehlsschalter

Befehlsschalter werden im WinForms-Framework durch die Klasse **Button** realisiert.

9.4.2.1 Beispiel

Im folgenden *Multi Purpose Counter*, der z.B. zur Verkehrszählung taugt, kommen zwei **Button**-Objekte zum Einsatz:



Das Erzeugen und Konfigurieren der **Button**-Steuerelemente erfolgt im Fensterkonstruktor. Weil die beiden Befehlsschalter und das **Label**-Objekt für den Zählerstand auch in Ereignisbehandlungsroutinen angesprochen werden, sind Instanzvariablen erforderlich:

```

using System;
using System.Windows.Forms;
using System.Drawing;

class ButtonDemo : Form {
    Label stand;
    long anzahl;
    Button count, reset;

    ButtonDemo() {
        Width = 350; Height = 120;
        Text = "Multi Purpose Counter";
        FormBorderStyle = FormBorderStyle.FixedSingle;

        stand = new Label();
        stand.Text = anzahl.ToString();
        stand.Font = new Font("Arial", 15);
        stand.Left = 20; stand.Top = 30;
        stand.Width = 100;
        stand.TextAlign = ContentAlignment.MiddleRight;
        stand.BorderStyle = BorderStyle.FixedSingle;
        Controls.Add(stand);

        count = new Button();
        count.Text = "Count";
        count.Left = 150; count.Top = 30;
        Controls.Add(count);

        reset = new Button();
        reset.Text = "Reset";
        reset.Left = 250; reset.Top = 30;
        Controls.Add(reset);

        EventHandler eh = new EventHandler(ButtonOnClick);
        count.Click += eh;
        reset.Click += eh;
    }

    void ButtonOnClick(object sender, EventArgs e) {
        if (sender == count)
            anzahl++;
        else
            anzahl = 0;
        stand.Text = anzahl.ToString();
    }

    static void Main() {
        Application.Run(new ButtonDemo());
    }
}

```

Weil eine Änderung der Formulargröße nicht erwünscht ist, erhält die **Form**-Eigenschaft **FormBorderStyle** den Wert **FixedSingle** aus der Enumeration **FormBorderStyle** im Namensraum **System.Windows.Forms**:

```
FormBorderStyle = FormBorderStyle.FixedSingle;
```

Die **Height**-Eigenschaften der drei Steuerelemente werden nicht spezifiziert, und C# verwendet infolgedessen den voreingestellten Wert 23, der vermutlich folgendermaßen zu Stande kommt:

Höhe der Standardschrift (= 13 Pixel) plus jeweils 5 Pixel für den oberen und den unteren Rand

Um für das **Label**-Steuerelement eine alternative Schrift zu wählen, erhält seine **Font**-Eigenschaft eine Referenz auf ein neues **Font**-Objekt mit den gewünschten Eigenschaften:

```
stand.Font = new Font("Arial", 15);
```

Außerdem erhält das Label einen Rahmen und die (bei einer Zahlenanzeige übliche) rechtsbündige Textausrichtung:

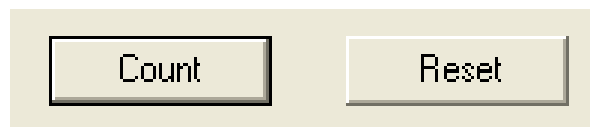
```
stand.TextAlign = ContentAlignment.MiddleRight;  
stand.BorderStyle = BorderStyle.FixedSingle;
```

Die für das **Click**-Ereignis *beider* **Button**-Steuerelemente registrierte Ereignisbehandlungsmethode `ButtonOnClick()` wertet den Parameter **sender** aus und orientiert ihr Verhalten an der Ereignisquelle. Nachdem das Feld `anzahl` seinen neuen Wert erhalten hat, wird das **Label**-Steuerelement auf den neuen Stand gebracht:

```
stand.Text = anzahl.ToString();
```

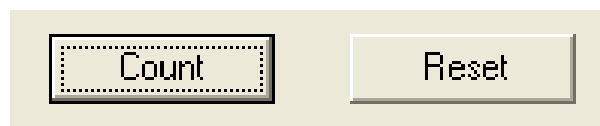
9.4.2.2 Eingabefokus und Standardschaltfläche

Obwohl beide Schalter des Multi-Counters im Fensterkonstruktor analog konfiguriert werden, sehen sie nach dem Programmstart etwas unterschiedlich aus:¹



Die zuerst auf das Formular gesetzte Schaltfläche (ansprechbar über die Instanzvariable `count`) besitzt den **Eingabefokus** und ist mit einer durchgezogenen schwarzen Umrisslinie dekoriert. Folglich werden alle Tastaturereignisse, die nicht zur Fokussteuerung durch den Container dienen, an den `count`-Schalter geleitet. Hier wird durch die **Enter**- und die Leertaste² wie durch einen linken Mausklick das **Click**-Ereignis ausgelöst, was die Bedienung des Programms erleichtert.

Zum kompletten Dekor einer Schaltfläche mit Eingabefokus gehört auch eine innen liegende gepunktete Linie:



¹ In Abschnitt 9.6.4 werden wir vom Visual Studio 2008 - Quellcode-Assistenten lernen, wie man die in Windows-Versionen ab XP enthaltenen visuellen Stile unterstützt (mit einer zusätzlichen Anweisung in der **Main()**-Methode). Dann werden die Steuerelemente per Voreinstellung (abhängig von den Einstellungen des Benutzers) attraktiver aussehen, wobei aber die Ausführungen des aktuellen Abschnitts gültig bleiben.

² Die Leertaste löst das **Click**-Ereignis eines fokussierten Schalters *nicht* aus, wenn eine Behandlungsmethode beim Ereignis **KeyDown** registriert ist.

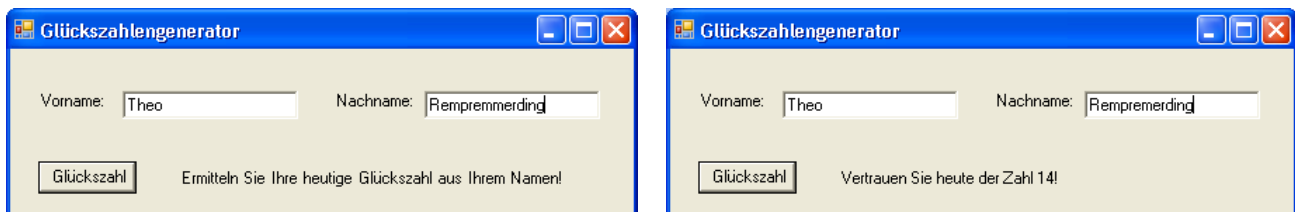
Bei älteren Windows-Versionen (vor XP) erhält der **count**-Button schon beim Programmstart das komplette Fokusdekor, bei Windows XP und Vista erscheint die gepunktete Linie erst nach einem benutzerinitiierten Fokuswechsel.¹

Dank der Fokusverwaltung durch das Formular können Benutzer per Tabulator- oder Pfeiltaste den Eingabefokus zwischen eingabetauglichen Steuerelementen wandern lassen, wobei die Aktivierungsreihenfolge von den **TabIndex**-Werten der Steuerelemente abhängt. Neben Schaltflächen können z.B. auch Kontrollkästchen und Listenfelder den Eingabefokus erhalten.

Für jedes Formular kann über die **Form**-Eigenschaft **AcceptButton** eine **Standardschaltfläche** festgelegt werden. Ihr **Click**-Ereignis wird per **Enter**-Taste ausgelöst, sofern keine andere Schaltfläche des Formulars den Eingabefokus besitzt. Im Beispielprogramm zum **TextBox**-Steuerelement (siehe Abschnitt 9.4.3) erhält der Schalter zur Anforderung einer persönlichen Glückszahl die Rolle der Standardschaltfläche:

```
AcceptButton = cbGluecksZahl;
```

Folglich kann der Schalter auch dann per **Enter**-Taste ausgelöst werden, wenn eines der Textfelder den Eingabefokus besitzt (erkennbar an der Texteingabemarke):



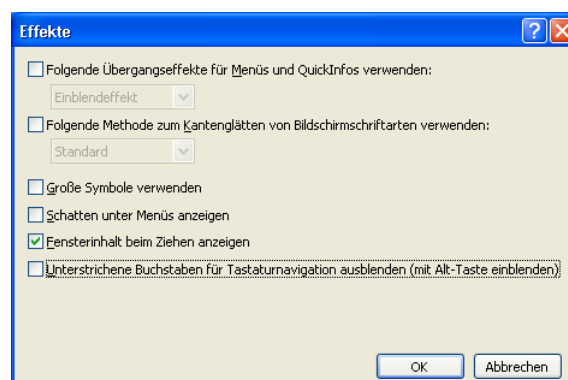
Im Glückszahlenbeispiel bietet es sich sogar an, den Schalter über den Wert **false** seiner **TabStop**-Eigenschaft aus der Liste der per Tastatur fokussierbaren Steuerelemente zu entfernen, damit die Tabulatortaste nur noch zwischen den beiden Textfeldern wechselt:

```
cbGluecksZahl.TabStop = false;
```

Analog zum **AcceptButton** kann für jedes Formular über die **Form**-Eigenschaft **CancelButton** eine **Escape-Schaltfläche** festgelegt werden. Ihr **Click**-Ereignis wird per **Esc**-Taste ausgelöst, sofern keine andere Schaltfläche des Formulars den Eingabefokus besitzt.

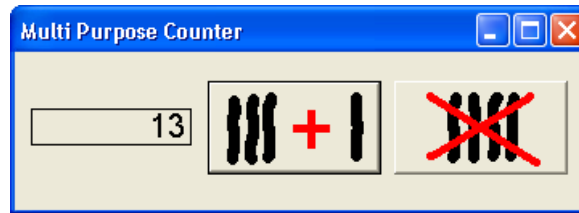
¹ In Windows XP lässt sich das Fokusdekors-Verhalten der älteren Windows-Versionen folgendermaßen reaktivieren:

- Im Systemsteuerungsapplet **Anzeige** auf der Registerkarte **Darstellung** mit einem Klick auf den Schalter **Effekte** den gleichnamigen Dialog öffnen
- Beim Kontrollkästchen **Unterstrichene Buchstaben für Tastaturnavigation ausblenden** die Markierung entfernen



9.4.2.3 Bitmaps auf Schaltflächen

Um dem oben vorgestellten Mehrzweck-Zählprogramm ein individuelles Design zu geben, kann man die Beschriftungen der Schaltflächen durch Bitmaps ersetzen, z.B.:



Zunächst sind 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. Im Formularkonstruktor sind im Vergleich zu *beschrifteten* Schaltflächen (siehe Abschnitt 9.4.2.1) nur wenige Änderungen erforderlich:

```
count = new Button();
count.Image = new Bitmap("count.bmp");
count.Left = 120; count.Top = 18;
count.Width = count.Image.Width + 8;
count.Height = count.Image.Height + 8;
Controls.Add(count);

reset = new Button();
reset.Image = new Bitmap("reset.bmp");
reset.Left = 230; reset.Top = 18;
reset.Width = reset.Image.Width + 8;
reset.Height = reset.Image.Height + 8;
Controls.Add(reset);
```

Den **Image**-Eigenschaften der Schaltflächen werden **Bitmap**-Objekte zugewiesen, die aus **bmp**-Dateien entstehen. Beim empfehlenswerten Verzicht auf eine Pfadangabe werden die Dateien im Verzeichnis mit dem Assembly erwartet (z.B. in **...\Button\Bitmap\bin\Debug**). Später werden Sie eine Möglichkeit kennen lernen, Bitmap-Dateien als *Ressourcen* in ein Assembly aufzunehmen.

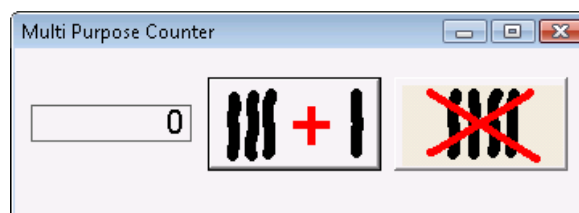
Wählen Sie die Größe eines zu bemalenden Schalters so, dass um das Bild herum mindestens vier Pixel Rand verbleiben, damit dort ggf. der gepunktete Fokusrahmen (siehe Abschnitt 9.4.2.2) angezeigt werden kann, z.B.:

```
count.Width = count.Image.Width + 8;
count.Height = count.Image.Height + 8;
```

Mit etwas Zusatzaufwand macht man die Hintergrundfarbe der verwendeten Bitmaps transparent, so dass die bemalten Schalter unabhängig vom eingestellten Windows-Design optisch mit den restlichen Fensterbestandteilen harmonieren, z.B.:

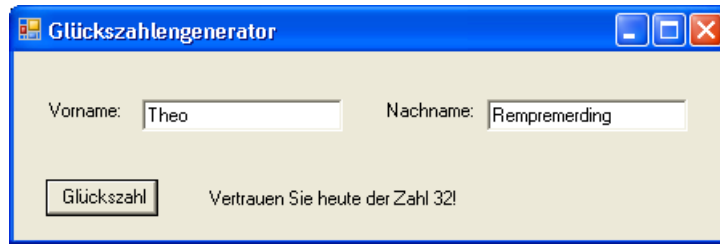
```
Bitmap bmp = new Bitmap("count.bmp");
bmp.MakeTransparent(bmp.GetPixel(1, 1));
count.Image = bmp;
```

Wenn im Beispielprogramm ausschließlich der *linke* Schalter eine transparente Hintergrundfarbe erhält, resultiert unter Windows Vista folgendes Erscheinungsbild:



9.4.3 Textfelder

Kurze Texteingaben der Benutzer erfasst man im WinForms-Framework mit Steuerelementen der Klasse **TextBox**. Auf dem bereits in Abschnitt 9.4.2.2 präsentierten Formular eines Programms zur Berechnung der persönlichen Glückszahl in Abhängigkeit vom Vor- und Nachnamen werden zwei **TextBox**-Objekte verwendet:



Für Steuerelemente eines Programms, die nicht nur im Formulkonstruktor, sondern auch in anderen Methoden (z.B. Ereignisbehandlungsroutinen) angesprochen werden sollen, sind Instanzvariablen erforderlich:

```
class TextBoxDemo : Form {
    Label lbInfo;
    TextBox tbVorname, tbNachname;
    . . .
}
```

Das Erzeugen und Konfigurieren der **TextBox**-Steuerelemente erfolgt im Konstruktor der Formulklass:

```
TextBoxDemo() {
    Width = 450; Height = 150;
    Text = "Glückszahlengenerator";
    FormBorderStyle = FormBorderStyle.FixedSingle;
    Label lbVorname = new Label(); Label lbNachname = new Label();
    lbVorname.Text = "Vorname:"; lbNachname.Text = "Nachname:";
    lbVorname.Width = 60; lbNachname.Width = 65;
    lbVorname.Left = 20; lbVorname.Top = 30;
    lbNachname.Left = 230; lbNachname.Top = 30;
    Controls.Add(lbVorname); Controls.Add(lbNachname);

    tbVorname = new TextBox(); tbNachname = new TextBox();
    tbVorname.Left = 80; tbVorname.Top = 30; tbVorname.Width = 125;
    tbNachname.Left = 295; tbNachname.Top = 30; tbNachname.Width = 125;
    EventHandler eh = new EventHandler(TextBoxOnTextChanged);
    tbVorname.TextChanged += eh;
    tbNachname.TextChanged += eh;
    Controls.Add(tbVorname); Controls.Add(tbNachname);

    Button cbGluecksZahl = new Button();
    cbGluecksZahl.Text = "Glückszahl";
    cbGluecksZahl.Top = 80; cbGluecksZahl.Left = 20; cbGluecksZahl.Width = 70;
    Controls.Add(cbGluecksZahl);
    cbGluecksZahl.TabStop = false;
    cbGluecksZahl.Click += new EventHandler(ButtonOnClick);
    AcceptButton = cbGluecksZahl;

    lbInfo = new Label();
    lbInfo.Text = strInst;
    lbInfo.Left = 120; lbInfo.Top = 85;
    lbInfo.AutoSize = true;
    Controls.Add(lbInfo);
}
```

Über die **TextBox**-Eigenschaft **Text** kann man auf den Inhalt eines Texteingabefeldes zugreifen, z.B. in der folgenden Behandlungsmethode zum **Click**-Ereignis des Befehlsschalters:

```

void ButtonOnClick(object sender, EventArgs e) {
    String vn = tbVorname.Text.ToUpper();
    String nn = tbNachname.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 zsg = new Random(DateTime.Today.Day + seed);
        lbInfo.Text = "Vertrauen Sie heute der Zahl " +
            (zsg.Next(100)+1).ToString() + "!";
        bValToRemove = 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 die zugehörigen Texte geändert haben:

```

void TextBoxOnTextChanged(object sender, EventArgs e) {
    if (bValToRemove) {
        lbInfo.Text = strInst;
        bValToRemove = false;
    }
}

```

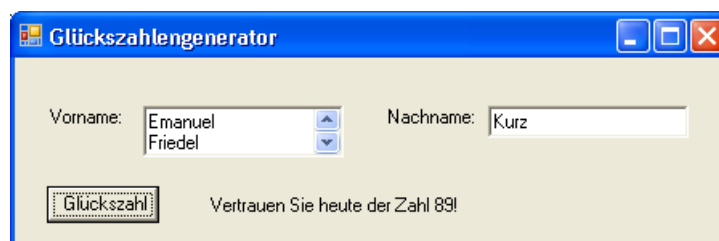
TextBox-Steuerelemente besitzen etliche Kompetenzen, die den Benutzer erfreuen und dabei den Programmierer wenig belasten, z.B.:

- Textmarkierung per Maus oder Tastatur
- Kommunikation mit der Zwischenablage über die Tastenkombinationen **Strg+C**, **Strg+X** und **Strg+V**
- Rücknahme der letzten Änderung über **Strg+Z**
- Kontextmenü mit **Bearbeiten**-Funktionen
- Automatische Textvervollständigung, wobei allerdings der Programmierer eine **Auto-CompleteSource** beisteuern muss

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

...\\BspUeb\\WinForms\\Steuerelemente\\Texterfassung\\TextBox

Um ein **mehrzeiliges Textfeld** zu erhalten, setzt man die **TextBox**-Eigenschaft **Multiline** auf den Wert **true** und sorgt für genügend Platz. Sind vertikale Bildlaufleisten erwünscht, setzt man auch die **TextBox**-Eigenschaft **ScrollBars** auf den Wert **true**. Für die folgende Variante des Glückszahlenprogramms



sind im Vergleich zum bisherigen Stand nur drei zusätzliche Zeilen erforderlich (im Fensterkonstruktor):

```

tbVorname.Multiline = true;
tbVorname.Height = 32;
tbVorname.ScrollBars = ScrollBars.Vertical;

```

Außerdem sollte man auf eine Standardschaltfläche verzichten, also die Anweisung

```
AcceptButton = cbGluecksZahl;
```

entfernen, weil anderenfalls im mehrzeiligen Textfeld keine Zeilenwechsel per **Enter**-Taste möglich sind.

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.

9.4.4 Spezielle Layout-Techniken

In diesem Abschnitt werden spezielle Aspekte der WinForms - Layoutlogik behandelt. Zunächst geht es um die Effekte einer (partiellen) Überlagerung von Steuerelementen am selben Ort. Dann werden Techniken behandelt, die für eine ergonomische Steuerelementanordnung bei *veränderlicher* Formulargröße sorgen können.

9.4.4.1 Z-Anordnung

Wird ein Steuerelement *A* vor einem Steuerelement *B* in die (per **Controls**-Eigenschaft ansprechbare) Steuerelementliste des gemeinsamen Containers eingefügt, dann hat *A* den kleineren **Controls**-Index, und *A* liegt in der Z-Anordnung *über B*, deckt also *B* eventuell (teilweise) ab.

Im folgenden Programm zur Demonstration der Stapelung werden die drei **Label**-Steuerelemente **lbRed**, **lbGreen** und **lbBlue** mit den Hintergrundfarben Rot, Grün und Blau nacheinander in ein Formular aufgenommen:

```
using System;
using System.Drawing;
using System.Windows.Forms;

class Stapelung : Form {
    Stapelung() {
        Text = "Stapelung";
        ClientSize = new Size(300,150);

        Label lbRed = new Label();
        lbRed.Size = new Size(250,75);
        lbRed.Left = 0;
        lbRed.BackColor = Color.Red;
        lbRed.Parent = this;
        lbRed.Click += new EventHandler(LabelOnClick);

        Label lbGreen = new Label();
        lbGreen.Size = new Size(200,100);
        lbGreen.Left = 100;
        lbGreen.BackColor = Color.Green;
        lbGreen.Parent = this;
        lbGreen.Click += new EventHandler(LabelOnClick);

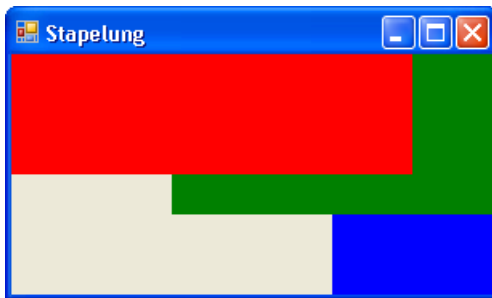
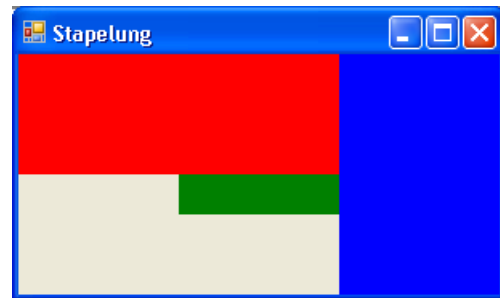
        Label lbBlue = new Label();
        lbBlue.Size = new Size(100,150);
        lbBlue.Left = 200;
        lbBlue.BackColor = Color.Blue;
        lbBlue.Parent = this;
        lbBlue.Click += new EventHandler(LabelOnClick);
    }

    void LabelOnClick(object sender, EventArgs e) {
        (sender as Control).BringToFront();
    }

    static void Main() {
        Application.Run(new Stapelung());
    }
}
```

Im gemeinsamen **Click**-Ereignishandler der **Label**-Steuerelemente gelangt das getroffene Objekt durch Aufruf der Methode **BringToFront()** in den Vordergrund und erhält den neuen **Controls**-Index Null, z.B.:

Nach dem Start

Nach einem Mausklick auf das blaue **Label**

9.4.4.2 Verankern

Ist ein Steuerelement an einer Formularseite *verankert*, behält es bei einer Veränderung der Fenstergröße nach Möglichkeit seinen Abstand zu dieser Seite konstant. Gesteuert wird dieses Verhalten durch die **Anchor**-Eigenschaft, der eine beliebige bitorientierte ODER-Verknüpfung von Werten der **AnchorStyles**-Enumeration aus dem Namensraum **System.Windows.Forms** zugewiesen werden kann:

Name	Wert
None	0
Top	1
Bottom	2
Left	4
Right	8

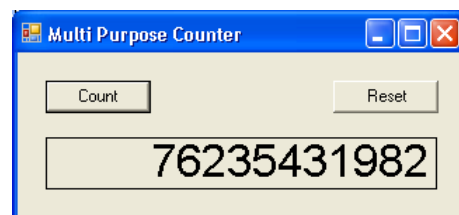
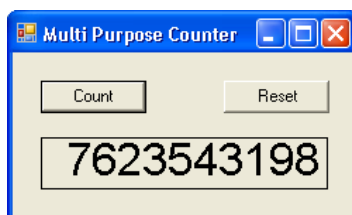
Bei der bitorientierten ODER-Verknüpfung (vgl. Abschnitt 3.5.6) von zwei Argumenten mit integralem Typ (z.B. **int**) entsteht ein neuer Wert desselben Typs. Im Bitmuster des Funktionswerts sind alle Bits eingeschaltet, die in mindestens einem Argument eingeschaltet waren.

In den **AnchorStyles**-Werten ist entweder *kein* oder *ein* Bit eingeschaltet, und per ODER-Verknüpfung lässt sich eine beliebige Teilmenge der vier Seitenbits einschalten. Per Voreinstellung behalten Steuerelemente bei Veränderungen der Fenstergröße ihren Abstand zum linken und zum oberen Rand bei, weil ihre **Anchor** – Eigenschaft den Wert

`AnchorStyles.Left | AnchorStyles.Top`

besitzt.

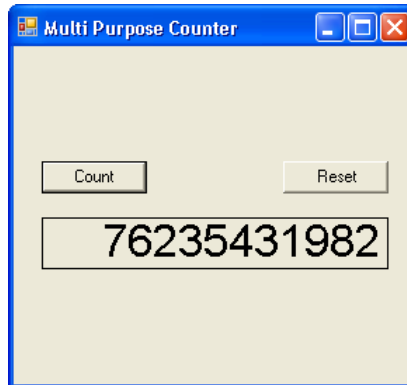
Bei der folgenden Variante des Mehrzweck-Zählprogramms aus Abschnitt 9.4.2 mit neu angeordneten Steuerelementen ist das Label mit dem Zählerstand links und rechts verankert, so dass es mit zunehmender Formularbreite mehr Platz bietet:



Mit den folgenden **AnchorStyles**-Zuweisungen wird außerdem dafür gesorgt, dass der **count**-Schalter am linken und der **reset**-Schalter am rechten Fensterrand verbleibt:

```
stand.Anchor = AnchorStyles.Left | AnchorStyles.Right;
count.Anchor = AnchorStyles.Left;
reset.Anchor = AnchorStyles.Right;
```

Weil die drei Steuerelemente weder oben noch unten verankert sind, bleiben sie bei einer Änderung der Fensterhöhe als Gruppe vertikal zentriert:



Über die **Form**-Eigenschaft **MinimumSize** kann verhindert werden, dass ein Benutzer durch extremes Verkleinern des Formulars in Schwierigkeiten gerät und die Hotline des Software-Herstellers mit der Klage belästigt, die Bedienelemente seien plötzlich verschwunden:

```
MinimumSize = new Size(Width, Height);
```

Nach den beschriebenen Modifikationen sieht der Fensterkonstruktor so aus:

```
Veranken() {
    Width = 248; Height = 150;
    Text = "Multi Purpose Counter";
    MinimumSize = new Size(Width, Height);

    stand = new Label();
    stand.Text = anzahl.ToString();
    stand.Font = new Font("Arial", 24);
    stand.Left = 20; stand.Top = 60;
    stand.Width = 200;
    stand.Height = stand.Font.Height;
    stand.BorderStyle = BorderStyle.FixedSingle;
    stand.TextAlign = ContentAlignment.MiddleRight;
    stand.Anchor = AnchorStyles.Left | AnchorStyles.Right;
    Controls.Add(stand);

    count = new Button();
    count.Text = "Count";
    count.Left = 20; count.Top = 20;
    count.Width = 70;
    count.Anchor = AnchorStyles.Left;
    Controls.Add(count);

    reset = new Button();
    reset.Text = "Reset";
    reset.Left = 150; reset.Top = 20;
    reset.Width = 70;
    reset.Anchor = AnchorStyles.Right;
    Controls.Add(reset);

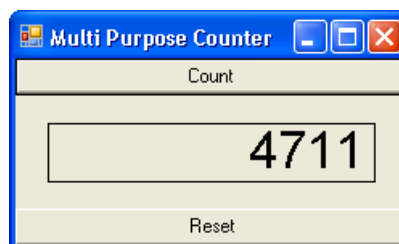
    EventHandler eh = new EventHandler(ButtonOnClick);
    count.Click += eh;
    reset.Click += eh;
}
```

9.4.4.3 Andocken

Oft ist es erwünscht, dass ein Steuerelement nahtlos an einer Formularseite andockt und sich bei jeder Fenstergröße über die komplette Seitenlänge erstreckt. Dieses Verhalten wäre durch Verankern an drei Seiten zu erzwingen (siehe Abschnitt 9.4.4.2), ist aber bequemer über die **Dock**-Eigenschaft zu realisieren, der ein einzelner Wert (*keine ODER-Kombination!*) der **DockStyle**-Enumeration zugewiesen werden kann:

DockStyle-Wert	Beschreibung
None	Nicht andocken
Top	Oben andocken
Bottom	Unten andocken
Left	Links andocken
Right	Rechts andocken
Fill	Allseitig andocken, also die gesamte Fläche des Containers belegen

Bei folgender Variante des Zählprogramms sind die beiden Schalter oben bzw. unten andockt:



Die erforderlichen Anweisungen im Formularkonstruktor:

```
count.Dock = DockStyle.Top;
reset.Dock = DockStyle.Bottom;
```

Das **Label**-Steuerelement ist wie bei der Programmvariante in Abschnitt 9.4.4.2 links und rechts verankert:

```
stand.Anchor = AnchorStyles.Left | AnchorStyles.Right;
```

Wollen *mehrere* Steuerelemente an derselben Seite ihres gemeinsamen Containers andocken, hängt das Ergebnis von der Z-Anordnung, d.h. von der Aufnahmereihenfolge ab. Wird das Steuerelement *A* vor dem Steuerelement *B* aufgenommen, dann rückt *A* nach Möglichkeit vom Rand ab, um Platz für *B* zu schaffen. Dies ist jedoch *nicht* möglich, wenn *B* auf **DockStyle.Fill** eingestellt ist. Daher sollte nur das zuerst andockte Steuerelement diesen **DockStyle** erhalten.

Im folgenden Fensterkonstruktor werden die drei **Label**-Steuerelemente **lbRed**, **lbGreen** und **lbBlue** nacheinander links andockt, wobei **lbRed** über die **Dock**-Eigenschaftsausprägung **DockStyle.Fill** auch an den drei restlichen Seiten andockt:

```
GemeinsameDockseite() {
    Text = "Gemeinsame Dockseiten: Links";
    ClientSize = new Size(300, 100);
    ForeColor = Color.White;

    Label lbRed = new Label();
    lbRed.BackColor = Color.Red;
    lbRed.Text = "0";
    lbRed.Parent = this;
    lbRed.Dock = DockStyle.Fill;

    Label lbGreen = new Label();
    lbGreen.Width = 100;
    lbGreen.BackColor = Color.Green;
    lbGreen.Text = "1";
    lbGreen.Parent = this;
```

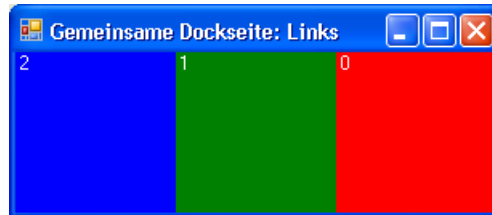
```

lbGreen.Dock = DockStyle.Left;

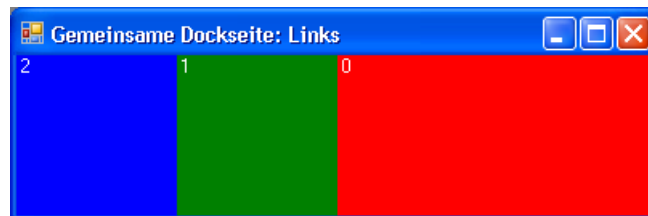
Label lbBlue = new Label();
lbBlue.Width = 100;
lbBlue.BackColor = Color.Blue;
lbBlue.Parent = this;
lbBlue.Text = "2";
lbBlue.Dock = DockStyle.Left;
}

```

Das zuerst aufgenommene Steuerelement **lbRed** (siehe **Controls**-Indexnummern links oben, vergeben in der Aufnahmereihenfolge) macht für Neulinge Platz und ist schließlich am weitesten vom linken Fensterrand entfernt:



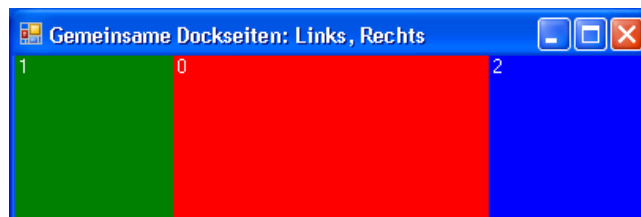
Bei einer Verbreiterung des Fensters passt sich **lbRed** wegen seiner **Dock**-Eigenschaftsausprägung **DockStyle.Fill** an, während die anderen **Label**-Objekte nicht reagieren:



Nach der folgenden Änderung im Fensterkonstruktor

```
lbBlue.Dock = DockStyle.Right;
```

überlässt das dienstälteste und auf allseitiges Andocken eingestellte **Label**-Steuerelement (Indexnummer Null, **DockStyle.Fill**) gleich an zwei Seiten jüngeren **Dock**-Konkurrenten den Fensterrand:



Der **DockStyle.Fill** eignet sich z.B. für ein **RichTextBox**-Steuerelement (also für einen kompletten Texteditor). Hier soll die gesamte Formularfläche genutzt werden, wobei aber an einem Rand klebende Bedienelemente Vorrang haben müssen.

9.4.4.4 TableLayoutPanel

Seit der Version 2.0 bietet das .NET - Framework mit dem **TableLayoutPanel** einen Container mit Tabellenstruktur, der in jeder Zelle ein Steuerelement aufnehmen kann. An Stelle gewöhnlicher Steuerelemente können prinzipiell auch weitere **TableLayoutPanel**-Objekte in die Zellen gesteckt werden.

Hauptzweck eines **TableLayoutPanel**-Objekts ist *nicht* die Unterstützung bei der *statischen* Anordnung von Steuerelementen, sondern der Entwurf von Formularen, die bei einer Größenänderung den verfügbaren Platz sinnvoll neu verteilen. Daher sind für die Zeilen und Spalten neben absoluten auch *prozentuale* Größenangaben erlaubt.

Der Bequemlichkeit halber verwenden wir zur Demonstration erneut das in Abschnitt 9.4.2 vorgestellten Mehrzweck-Zählprogramm, obwohl für sein einfaches Formular eigentlich kein **TableLayoutPanel** erforderlich ist. In den folgenden Anweisungen des Formularkonstruktors

```
TableLayoutPanel table = new TableLayoutPanel();
table.ColumnStyles.Add(new ColumnStyle(SizeType.Percent, 50F));
table.ColumnStyles.Add(new ColumnStyle(SizeType.Percent, 25F));
table.ColumnStyles.Add(new ColumnStyle(SizeType.Percent, 25F));
Controls.Add(table);
table.Dock = DockStyle.Fill;
```

wird eine Tabelle erstellt und gestaltet:

- Die Anzahl der Zeilen und Spalten ermittelt der Compiler automatisch.
- Die angegebenen prozentualen Spaltenbreiten werden den ersten drei Spalten nacheinander zugeordnet. Dazu sind **ColumnStyle**-Objekte mit einem Wert der **SizeType**-Enumeration und einer Größenangabe (mit Datentyp **float**) als Konstruktor-Parameter zu erzeugen und per **Add()**-Methode in die Kollektion der Spaltenstile einzufügen. Wir reservieren 50 % der verfügbaren Breite für den Zählerstand und jeweils 25 % für die beiden Schalter.
- Das **TableLayoutPanel**-Steuerelement wird auf das Formular gesetzt und darf den gesamten dort verfügbaren Platz belegen (**DockStyle.Fill**).

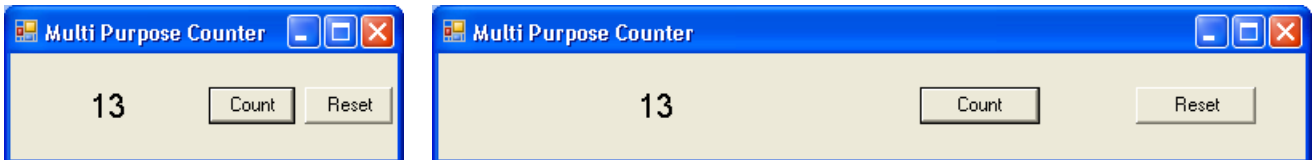
Das **Label** und die beiden **Button**-Objekte bleiben in ihrer Zelle bei jeder Änderung der Fenstergröße horizontal und vertikal zentriert (**AnchorStyles.None**):

```
table.Controls.Add(stand, 0, 0);
stand.Anchor = AnchorStyles.None;

table.Controls.Add(count, 1, 0);
count.Anchor = AnchorStyles.None;

table.Controls.Add(reset, 2, 0);
reset.Anchor = AnchorStyles.None;
```

Die verfügbare Breite wird stets in der festgelegten Weise verteilt:

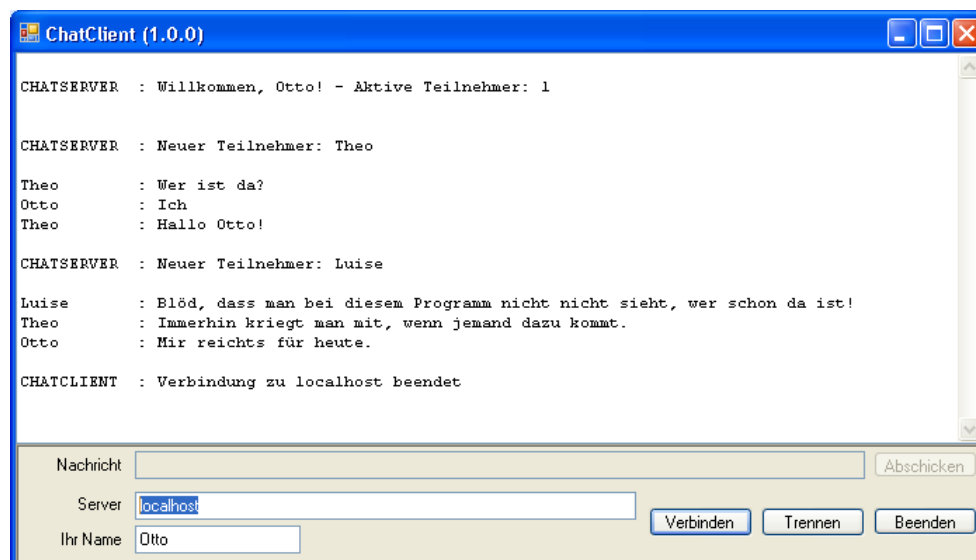


Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

...\\BspUeb\\WinForms\\Spezielle Layout-Techniken\\TableLayoutPanel

Microsoft warnt wegen potentieller Schwierigkeiten bei der Fehlersuche davor, **TableLayoutPanel**-Steuerelemente zu schachteln¹. Im folgenden Beispielprogramm wird allerdings genau diese Technik problemlos für ein flexibles Formulardesign verwendet:

¹ Siehe: <http://msdn.microsoft.com/en-us/library/ms171689.aspx>



Es handelt sich um den Lösungsvorschlag zu einer Übungsaufgabe in Abschnitt 14.5 (Chat-Server und -Client).

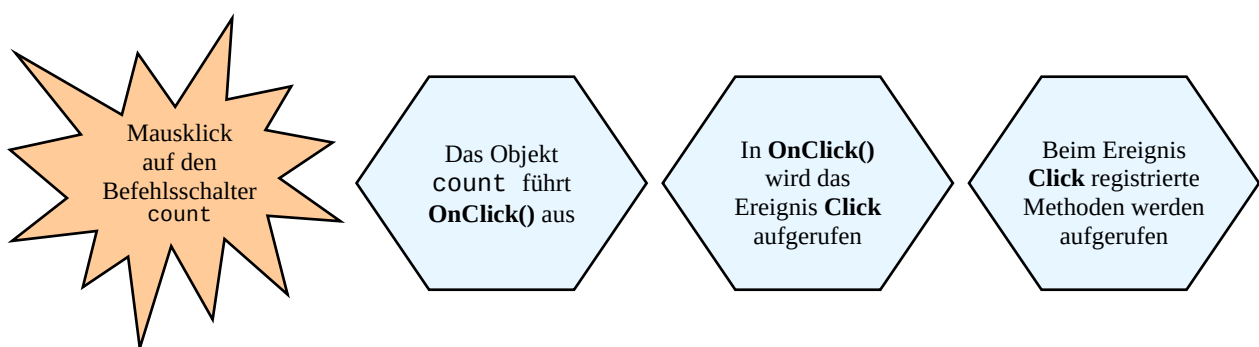
Eine weitere Option zur Gestaltung größenflexibler Formulare ist das (ebenfalls seit .NET 2.0 in der FCL vorhandene) Steuerelement **FlowLayoutPanel**.

9.5 Ereignisse und On-Methoden

Zur bisher vorgestellten Technik der Ereignisbehandlung gibt es eine oft benutzte Alternative, deren Behandlung relevantes Hintergrundwissen vermittelt. Jedes im Namensraum **System.Windows.Forms** definierte Ereignis wird in einer (meist von **System.Windows.Forms.Control** geerbten) Methode ausgelöst, deren Name aus dem Präfix **On** und dem Ereignisnamen besteht, z.B.

- **OnClick()**
- **OnMouseEnter()**

Andererseits können wir uns darauf verlassen, dass dank Windows und .NET - Plattform die **On**-Methode zu einem .NET - Ereignis im passenden Moment aufgerufen werden, z.B. nach einem Mausklick auf einen Befehlsschalter. Man kann das Geschehen so skizzieren:



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. Diese Vorgehensweise ist speziell beim **Paint**-Ereignis sehr verbreitet, mit dem wir uns in Kapitel 15 intensiv beschäftigen werden (siehe speziell Abschnitt 15.3.2).

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 Formularereignissen ist die Technik hingegen leicht nutzbar, weil wir die (von **Control** abstammende) Klasse **Form** regelmäßig beerben. Im folgendem Programm ist die **OnMouseDown**-

Methode des Formulars überschrieben und außerdem die Methode `FormOnMouseDown()` beim **MouseDown**-Ereignis des Formulars registriert:

```
using System;
using System.Windows.Forms;

class OnMethodenDemo : Form {
    Label lab;
    OnMethodenDemo() {
        Text = "OnMethoden-Demo";
        lab = new Label();
        lab.Parent = this;
        MouseDown += FormOnMouseDown;
    }

    protected override void OnMouseDown(MouseEventArgs e) {
        base.OnMouseDown(e);
        lab.Text = "(" + e.X.ToString() + "; " + e.Y.ToString() + ")";
    }

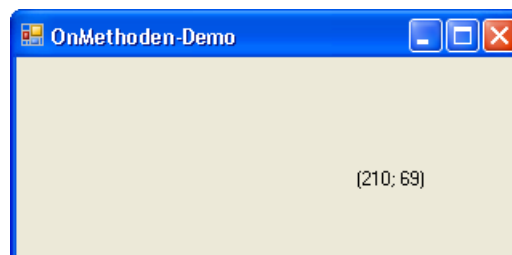
    void FormOnMouseDown(object sender, MouseEventArgs e) {
        lab.Left = e.X;
        lab.Top = e.Y;
    }

    static void Main() {
        Application.Run(new OnMethodenDemo());
    }
}
```

Beide Methoden teilen sich bei einem Mausklick folgende Arbeiten zur Aktualisierung der Eigenschaften eines **Label**-Objekts:

- **OnMouseDown()** legt die linke obere Ecke des Labels neu auf die Klickstelle fest, die der Methode über Eigenschaften eines **MouseEventArgs**-Objekts bekannt wird.
- `FormOnMouseDown()` verarbeitet die Koordinaten der Klickstelle zum neuen Wert der **Text**-Eigenschaft des Labels.

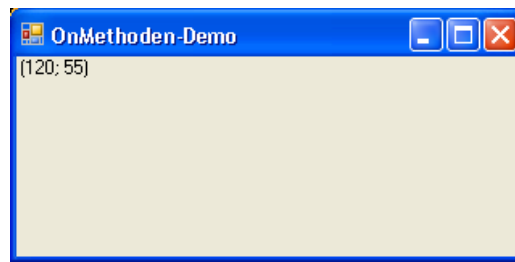
Damit die Kooperation gelingt, muss **OnMouseDown()** die überschriebene Basisklassenmethode aufrufen. Aufgrund der umständlichen (aber lehrreichen) Ereignisbehandlung kann das Programm die Koordinaten einer Klickstelle am Ort des Geschehens anzeigen, z.B.:



Unterlässt man den Aufruf der Basisklassenmethode,

```
protected override void OnMouseDown(MouseEventArgs e) {
    lab.Text = "(" + e.X.ToString() + "; " + e.Y.ToString() + ")";
}
```

wird die beim **MouseDown**-Ereignis registrierte Methoden still gelegt, so dass zwar die **Label**-Beschriftung weiterhin (durch **OnMouseDown()**) aktualisiert wird, die Koordinaten aber unverändert bei den Initialisierungswerten (0, 0) bleiben:



Man darf also in einer überschreibenden **On**-Methode nur dann auf den Aufruf der Basisklassenvariante verzichten, wenn ganz sicher kein Delegat beim betroffenen Ereignis registriert ist, weder aktuell noch bei einer möglichen späteren Verwendung der eigenen Klasse. Bei einem komplexen Softwaresystem, an dem eventuell mehrere Programmierer beteiligt sind, sollte man also in einer **On**-Überschreibung auf jeden Fall die Basisklassenmethode aufrufen.

In der Regel wird man den Basismethodenaufruf gleich zu Beginn der Überschreibung vornehmen.

9.6 WinForms - RAD

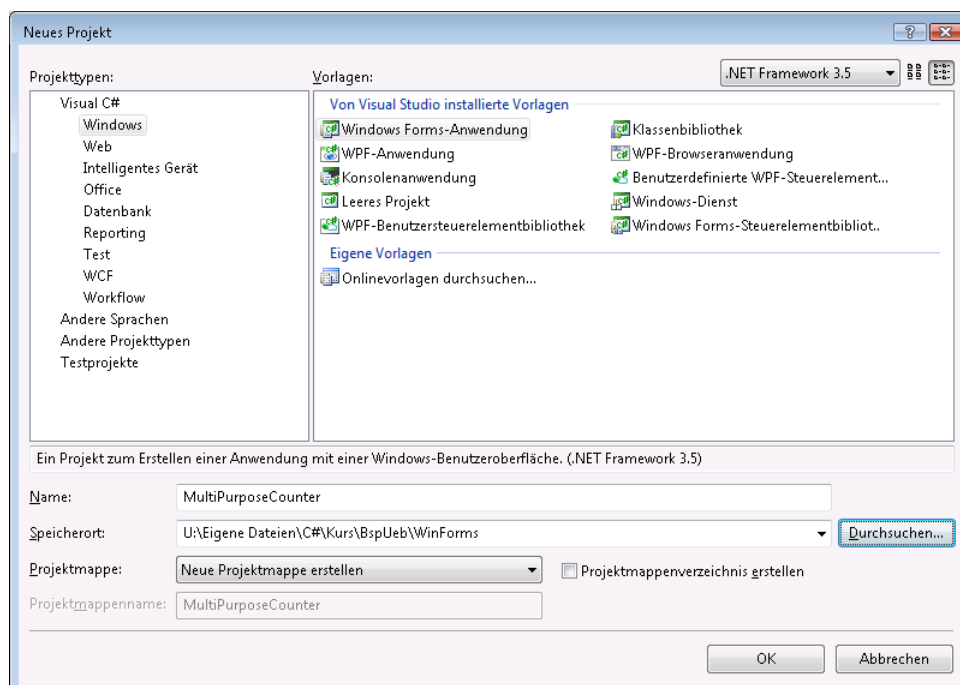
Mit Hilfe moderner Entwicklungsumgebungen lässt sich das Programmieren erheblich vereinfachen und beschleunigen (*Rapid Application Development*, RAD). Nachdem wir in Abschnitt 4.10 das Vorgehen mit der Visual C# 2008 Express Edition geübt haben, soll nun die Visual Studio 2008 Professional Edition zum Einsatz kommen.

9.6.1 Projekt anlegen mit Vorlage Windows Forms - Anwendung

Verwenden Sie nach

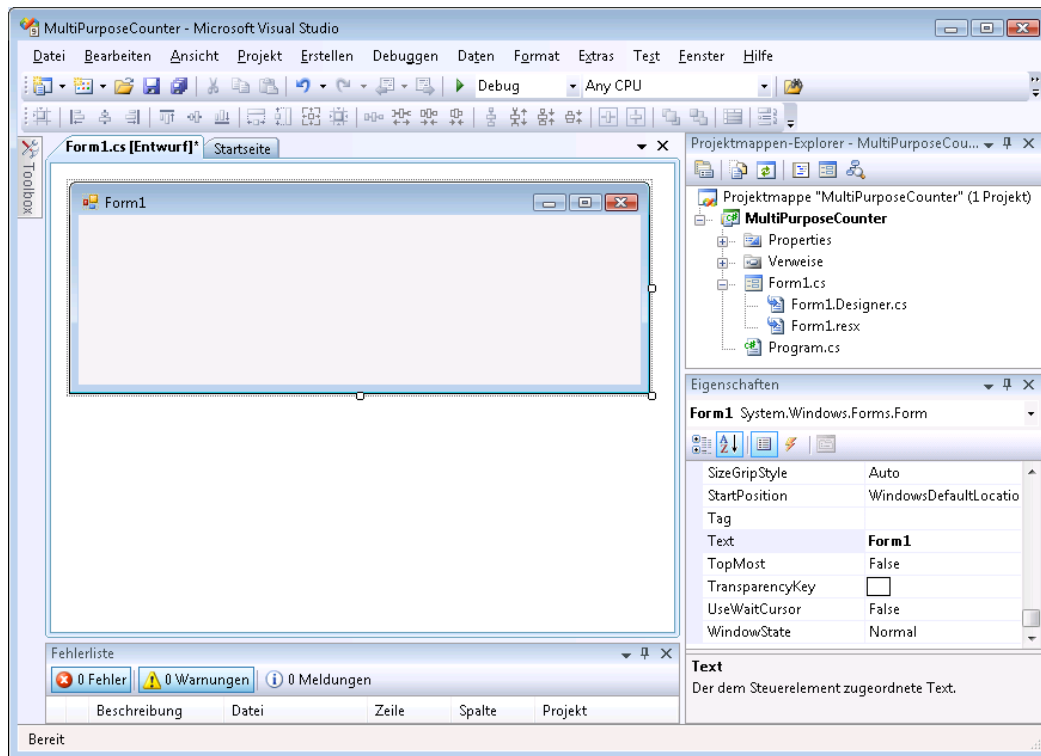
Datei > Neu > Projekt

für ein neues Projekt mit dem Namen **MultiPurposeCounter** die Vorlage **Windows Forms - Anwendung** - Anwendung:



Durch den Verzicht auf ein **Projektmappenverzeichnis** wählen wir eine „flache“ Projektdatenverwaltung ohne Zusammenfassung von mehreren Projekten zu einer Mappe.

Nach einem Mausklick auf **OK** präsentiert die Entwicklungsumgebung im Formulardesigner einen Rohling für das Hauptfenster der entstehenden Anwendung. Wir bringen das Formular wie in einem Grafikprogramm mit Hilfe der Anfasser in die gewünschte rechteckige Form:

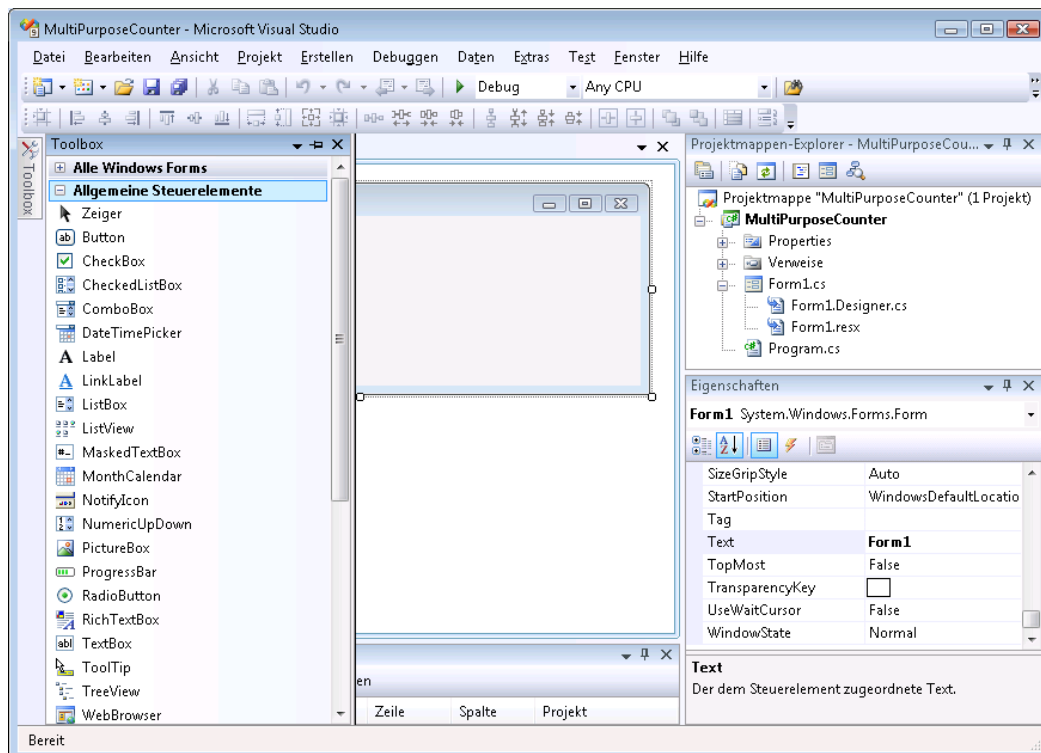


9.6.2 Steuerelemente aus der Toolbox übernehmen

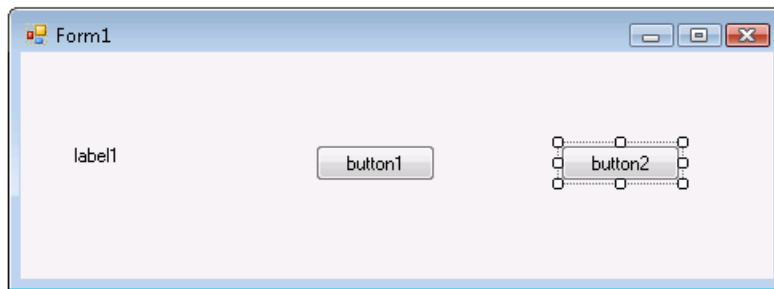
Öffnen Sie das **Toolbox**-Fenster mit dem Menübefehl

Ansicht > Toolbox

oder per Mauszeiger durch kurzes Verharren auf der **Toolbox**-Schaltfläche am linken Fensterrand:



Erweitern Sie im **Toolbox**-Fenster die Liste mit den **allgemeinen Steuerelementen**, und ziehen Sie per Maus ein **Label**-Objekt sowie zwei **Button**-Objekte auf das Formular, z.B.:



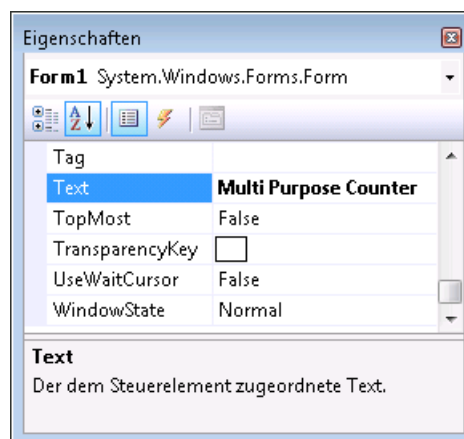
Über die Optionen zur Gestaltung der Größen und Positionen von Steuerelementen wurde schon in Abschnitt 4.10.3 berichtet.

9.6.3 Eigenschaften der Steuerelemente ändern

Im Eigenschaftsfenster der Entwicklungsumgebung, das bei Bedarf mit dem Menübefehl

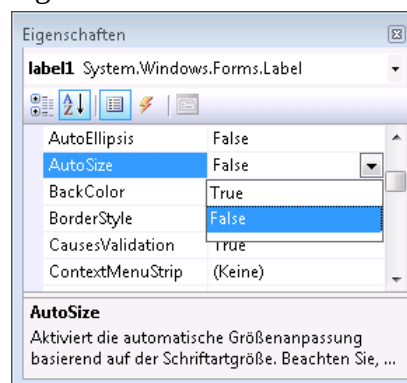
Ansicht > Eigenschaftsfenster

zu öffnen ist, lassen sich die Eigenschaften des markierten Steuerelements durch direkte Werteingabe festlegen, z.B. der **Text** in der Titelzeile des Formulars:



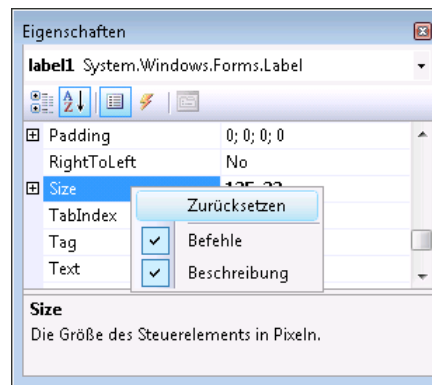
Ändern Sie außerdem ...

- die **Text** - Eigenschaften der Steuerelemente
- via Klappenmenü die **AutoSize**-Eigenschaft des Labels auf den Wert **false**:



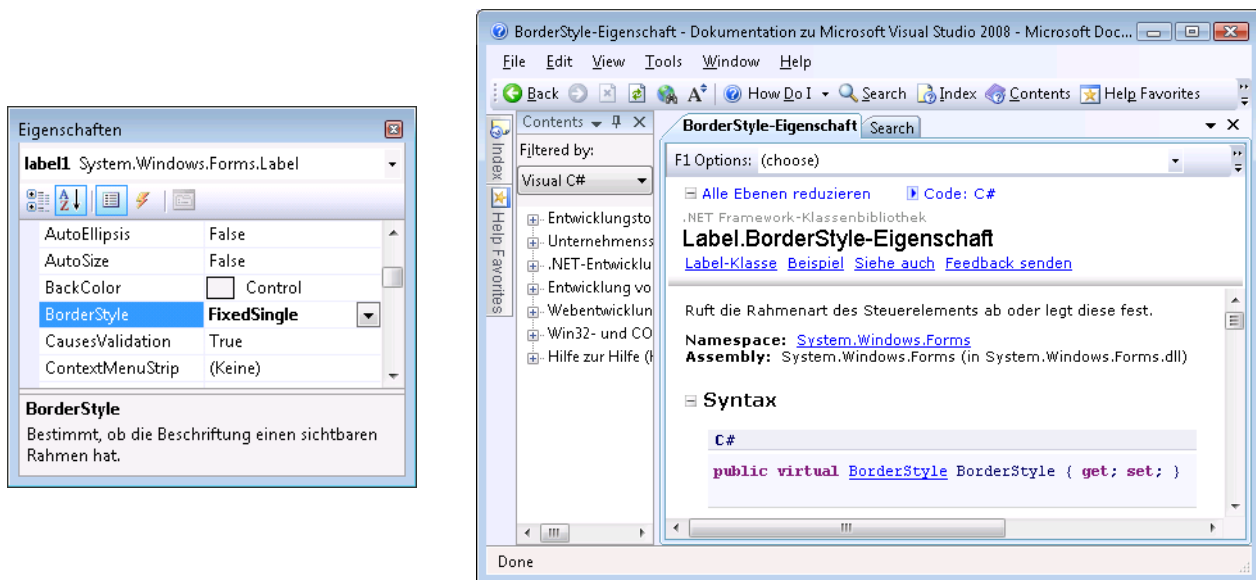
- die **BorderStyle**-Eigenschaft des Labels auf den Wert **FixedSingle**
- die **TextAlign**-Eigenschaft des Labels auf den Wert **MiddleRight**

Über das Kontextmenü zu einer Eigenschaft kann man zum voreingestellten Wert zurückkehren oder die Eigenschaftsbeschreibung am unteren Fensterrand (de)aktivieren:



Das aktuell zu gestaltende Steuerelement lässt sich auch über eine Liste im Kopfbereich des Eigenschaftsfensters wählen. Außerdem kann man die Eigenschaften von mehreren gleichzeitig markierten Objekten in einem Arbeitsgang ändern.

Sehr nützlich ist die Möglichkeit, die FCL-Dokumentation zur momentan markierten Eigenschaft über die Taste **F1** aufzurufen, z.B.:



9.6.4 Der Quellcode-Generator

Aufgrund Ihrer kreativen Tätigkeit erzeugt die Entwicklungsumgebung im Hintergrund Quellcode zu einer neuen Klasse namens **Form1**, die erwartungsgemäß von der Klasse **Form** im Namensraum **System.Windows.Forms** 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 **Form1** auf zwei Dateien verteilt:

- **Form1.cs**
Hier werden die von Ihnen erstellten Methoden landen (siehe unten).
- **Form1.Designer.cs**
Hier landet der automatisch generierte Quellcode, und Sie sollten in dieser Datei keine Änderungen vornehmen, um den Formulardesigner nicht aus dem Tritt zu bringen.

Dem C# - Compiler wird durch das Schlüsselwort **partial** in der Klassendefinition signalisiert, dass der Quellcode auf mehrere Dateien verteilt ist. Vor der Erstellung von Ereignisbehandlungsmethoden (siehe unten) enthält die Datei **Form1.cs** vor allem einen parameterfreien **Form1**-Konstruktor:

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace MultiPurposeCounter
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}

```

Hier wird die private Methode **InitializeComponent()** aufgerufen, die der Formulardesigner in der Datei **Form1.Designer.cs** implementiert.

Um eine ungefähre Vorstellung vom Wirken des Code-Generators zu erhalten, öffnen wir die Quelldatei **Form1.Designer.cs** (ohne Änderungsabsicht) per Doppelklick auf ihren Eintrag im Projektmappen-Explorer. Hier finden sich u.a. die Deklarationen der Instanzvariablen zu den Steuerelementen:

```

private System.Windows.Forms.Label label1;
private System.Windows.Forms.Button button1;
private System.Windows.Forms.Button button2;

```

Obwohl die Schutzstufe **private** für Klassen-Member voreingestellt ist, verwendet die Entwicklungsumgebung der Deutlichkeit halber den Modifikator bei jeder Deklaration.

Außerdem erstellt der Quellcode-Generator eine statische Startklasse namens **Program**

```

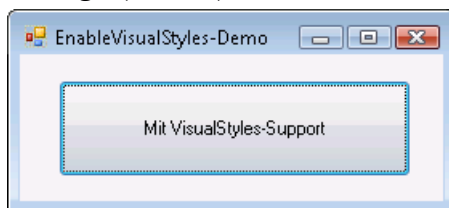
static class Program {
    [STAThread]
    static void Main() {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new Form1());
    }
}

```

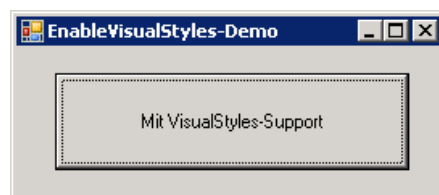
mit einigen erläuterungsbedürftigen Bestandteilen:

- Das Attribut **STAThread** wird in Abschnitt 11 behandelt.
- Die statische Methode **Application.EnableVisualStyles()** sorgt für die Unterstützung der mit Windows XP eingeführten visuellen Stile (Themen). Bei aktivierter Unterstützung (also vorhandenem **EnableVisualStyles()** - Aufruf) sieht z.B. ein Befehlsschalter unter Windows Vista beim so aus:

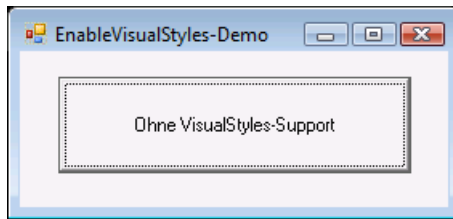
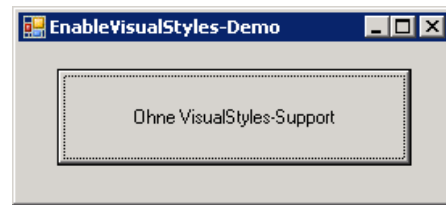
Design (Thema) **Windows Vista**



Design (Thema) **Windows - klassisch**



Ohne Unterstützung für visuellen Stile (Themen) erhält man unter Windows Vista folgende Ergebnisse:

Design (Thema) **Windows Vista**Design (Thema) **Windows - klassisch**

Bei aktiviertem Vista-Stil erhält man einen dementsprechenden Rahmen, doch im Klientenbereich der .NET - Anwendung haben die Steuerelemente das althergebrachte Erscheinungsbild. Unter Windows XP hat die Methode **Application.EnableVisualStyles()** einen analogen Effekt, und wir werden ab jetzt auch bei komplett selbst geschriebenen GUI-Anwendungen mit einer zusätzlichen Anweisung in der **Main()**-Methode eine attraktivere Optik ermöglichen.

- Der Aufruf **Application.SetCompatibleTextRenderingDefault(false)** sorgt dafür, dass Steuerelemente beim Zeichnen von Text die mit .NET 2.0 eingeführte Technik verwenden, die Verbesserungen bei der Performanz und Präzision bringt. Der voreingestellte Kompatibilitätsmodus ist nur empfehlenswert bei der Aktualisierung von Anwendungen, die ursprünglich für .NET 1.x entwickelt worden sind¹.

Der Quellcode-Generator setzt abweichend von der im Manuskript üblichen Praxis eröffnende geschweifte Klammern (z.B. bei Klassen- oder Methodendefinitionen) in eine eigene Zeile. Das alternative Verhalten ist über den Menübefehl

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

erreichbar (vgl. Abschnitt 4.10.5).

9.6.5 Ereignisbehandlungsmethoden anlegen

Nun erstellen wir zu jedem Befehlsschalter 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 *Count* für die Eigenschaft **Text**), so dass die Entwicklungsumgebung in **Form1.cs** die Instanzmethode **button1_Click()** mit leerem Rumpf anlegt

```
private void button1_Click(object sender, EventArgs e) {  
  
}
```

und die Quellcodedatei in einem Editorfenster öffnet. Außerdem wird die neue Methode beim zugehörigen Ereignis registriert (siehe Methode **InitializeComponent()** in der Datei **Form1.Designer.cs**):

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

Wir ergänzen in der **Form1**-Klassendefinition die **int**-Instanzvariable **anzahl** für den Zählerstand. In der Ereignisbehandlungsmethode **button1_Click()** wird diese Variable inkrementiert. Abschließend ist nur noch die Text-Eigenschaft des Labels zu aktualisieren:

¹ Bei der in .NET 2.0 eingeführten Neuerung geht es offenbar um die Ersetzung der in .NET 1.x enthaltenen, auf dem GDI+ basierenden Textdarstellung durch eine alternative Technik, die auf dem älteren GDI (ohne Plus) basiert (vgl. Abschnitt 15.7).

```

public partial class Form1 : Form
{
    int anzahl;
    public Form1()
    {
        InitializeComponent();

        private void button1_Click(object sender, EventArgs e) {
            anzahl++;
            label1.Text = anzahl.ToString();
        }
    }
}

```

Erstellen Sie analog die **Click**-Ereignisbehandlungsmethode zum Befehlsschalter `button2` (mit dem Wert `Reset` für die Eigenschaft **Text**):

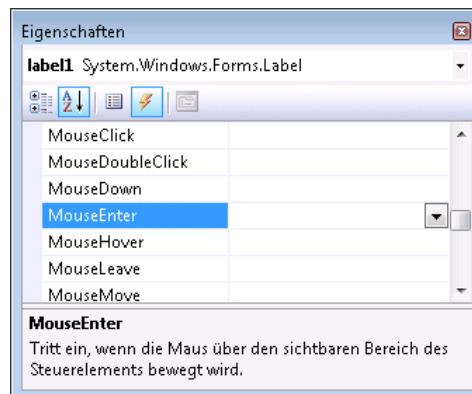
```

private void button2_Click(object sender, EventArgs e) {
    anzahl = 0;
    label1.Text = anzahl.ToString();
}

```

Das wars! Mit der Funktionstaste **F5** können Sie Ihr fertiges Programm übersetzen und ausführen lassen. Die Assembly-Datei **MultiPurposeCounter.exe** findet sich im Projektunterordner **...\bin\Debug**.

Ein Steuerelement bietet in der Regel *mehrere* Ereignisse an, wobei ein *Standardereignis* festgelegt ist (z.B. **Click** bei einem **Button**-Objekt). Wie eben zu sehen war, legt man im Formulardesigner die Behandlungsroutine zum Standardereignis per Doppelklick auf das Steuerelement an. Um die Behandlungsroutine zu einem anderen Ereignis per Formulardesigner anzulegen, wechselt man im Eigenschaftsfenster über den Schalter  zur Liste der unterstützten Ereignisse und setzt einen Doppelklick auf das zu behandelnde Ereignis, z.B.:



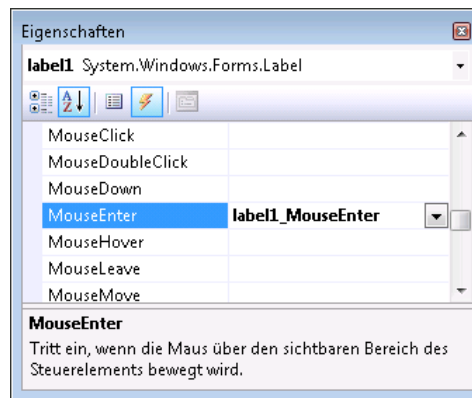
Das Visual Studio legt daraufhin in der Quellcodedatei **Form1.cs** eine neue Methode an

```

private void label1_MouseEnter(object sender, EventArgs e) {
}

```

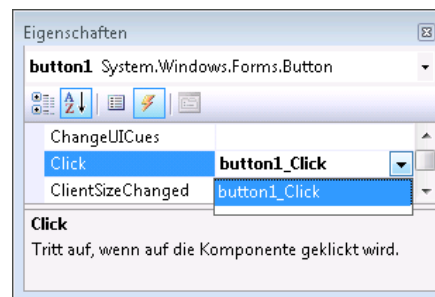
und registriert sie beim Ereignis. Im (mäßig sinnvollen) Beispiel könnten wir die Hintergrundfarbe des Labels ändern, sobald sich die Maus darüber befindet (Ereignis **MouseEnter**), wobei dann auch noch das Ereignis **MouseLeave** behandelt werden sollte. Das Quellcode-Äquivalent zu dieser Eigenschaftsfenster-Repräsentation der Ereignisregistrierung:



findet sich in der Methode **InitializeComponent()** (siehe Datei **Form1.Designer.cs**):

```
this.label1.MouseEnter += new System.EventHandler(this.label1_MouseEnter);
```

Soll eine Ereignismethode per Formulardesigner bei *mehreren* Steuerelementen registriert werden, legt man sie zum ersten Ereignisemittenden neu an und ordnet sie später den übrigen Emittenten im Eigenschaftsdialog per Klappmenü zu:

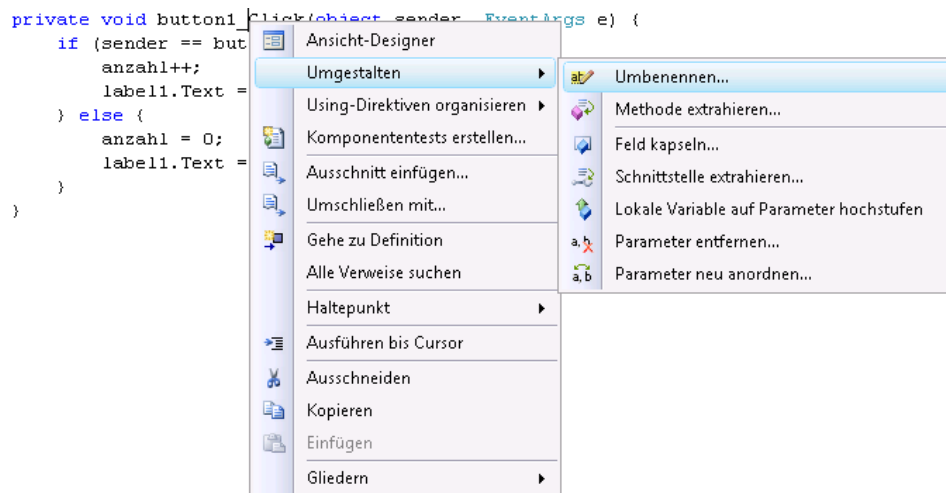


Damit im Beispiel die Methode **button1_Click()** für *beide* Schalter taugt, muss sie natürlich modifiziert werden (vgl. Abschnitt 9.4.2.1):

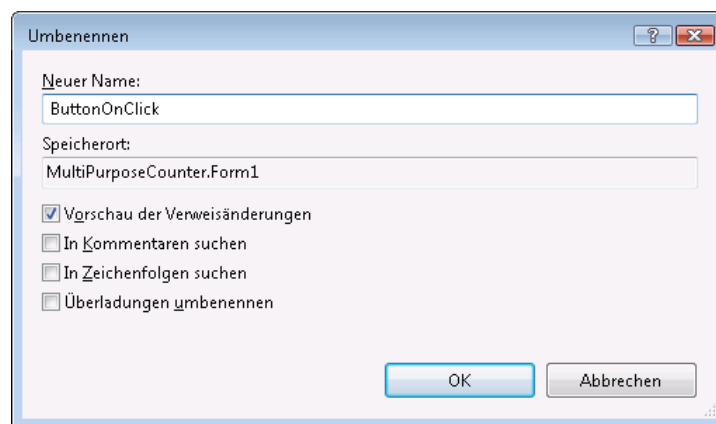
```
private void button1_Click(object sender, EventArgs e) {
    if (sender == button1) {
        anzahl++;
        label1.Text = anzahl.ToString();
    } else {
        anzahl = 0;
        label1.Text = anzahl.ToString();
    }
}
```

9.6.6 Quellcode-Umgestaltung (Refactoring)

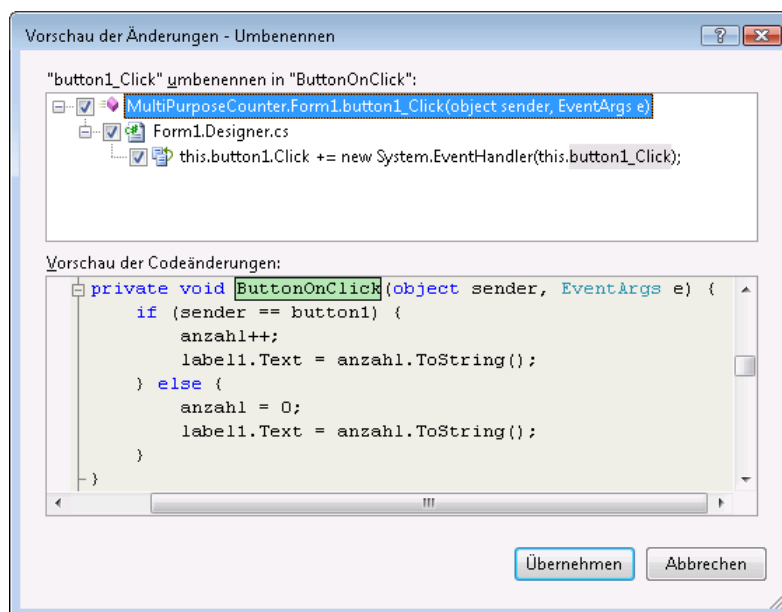
Nachdem im Beispiel die Ereignisbehandlungsmethode **button1_Click()** für *beide* **Button**-Objekte zuständig geworden ist, passt ihr Name nicht mehr. Wir ändern ihn ab und lernen dabei eine einfache Anwendung der außerordentlich nützlichen Quellcode-Umgestaltung kennen (engl. *Refactoring*). Nach einem Rechtsklick auf die Methode und Auswahl der Kontextmenü-Option **Umgestaltung > Umbenennen**



wählen wir den Namen `ButtonOnClick()` (vgl. Benennungsvorschlag in Abschnitt 9.4.1.3):



Die Entwicklungsumgebung gibt eine Vorschau auf den Effekt der Umbenennung:



Nach dem **Übernehmen** wird das Umbenennen in allen Quellcode-Dateien des Projekts ausgeführt.

9.7 Zusammenfassung zu Abschnitt 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.

Das WinForms-Framework bietet im Namensraum **System.Windows.Forms** leistungsfähige Klassen zur GUI-Programmierung. Als elementare WinForms-Klassen haben Sie kennen gelernt:

- **Form**
Alle Formulare einer WinForms-Anwendung werden über Objekte einer von **Form** abstammenden Klasse verwaltet.
Das **Hauptfenster** einer WinForms-Anwendung spielt eine besondere Rolle:
 - Es wird dem Methodenaufruf **Application.Run()** als Parameter übergeben und erscheint daraufhin auf dem Bildschirm.
 - Ein sinnvoll entworfenes WinForms-Programm endet beim Schließen des Hauptfensters.
- **Application**
Diese Klasse bietet u.a. essentielle statische Methoden zum Betreiben einer WinForms-Anwendung. Durch den Methodenaufruf **Application.Run()** wird ...
 - das Programm um eine **Nachrichtenschleife** erweitert,
 - das **Hauptfenster** der Anwendung geöffnet.

Delegaten

Ein Objekt eines Delegates zeigt auf eine Methode (oder auf eine Liste von Methoden) mit einer bestimmten Signatur. Beim Aufruf eines Delegateobjekts werden alle Methoden seiner Aufruf-liste nacheinander ausgeführt.

Ereignisse

Ein .NET - Ereignis ist im Wesentlichen eine Delegatevariable, die vom Ereignisanbieter bei bestimmten Gelegenheiten aufgerufen werden (z.B. bei einem Klick auf einen Schalter). 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 Aufruf-liste zum Ereignisobjekt.

Steuerelemente

Das WinForms-Framework 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**, **Left**, **Font**, **TextAlign**) können zur Entwurfszeit über Werkzeuge der Entwicklungsumgebungen konfiguriert werden.

Einfache Beispiele:

- **Label**
- **Button**
- **TextBox**

Die folgenden Layout-Techniken sind u.a. bei der dynamischen Positions- und Größenpassung der Steuerelemente an eine vom Benutzer geänderte Formulargröße relevant:

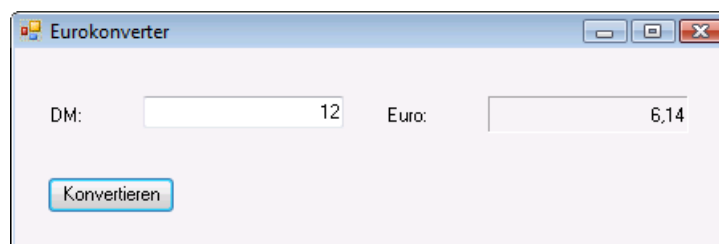
- Verankern
- Andocken
- **TableLayoutPanel**

9.8 Übungsaufgaben zu Kapitel 9

1) Welche von den folgenden Aussagen sind richtig?

1. In der Klassendefinition zu einem Formular wird für jedes aufzunehmende Steuerelement eine Instanzvariable benötigt.
2. Steuerelemente können über Ereignisse mit dem elterlichen Formular (**Control**-Eigenschaft **Parent**) kommunizieren.
3. Bei den Ereignissen eines Steuerelements kann jeweils nur *eine* Behandlungsmethode registriert werden.
4. Steuerelemente übernehmen die Schriftart vom elterlichen Container (z.B. vom Formular).

2) Erstellen Sie (ohne Formular-Designer) ein WinForms-Programm, das DM-Beträge in Euro-Beträge konvertiert (per Division durch 1,95583), z.B. mit folgender Benutzeroberfläche:



Ein angezeigter Euro-Betrag sollte verschwinden, sobald sich der korrespondierende Inhalt des DM-Textfelds ändert.

Um die attraktive Optik der Steuerelemente zu ermöglichen, müssen Sie in der Main()-Methode die Unterstützung der seit Windows XP vorhandenen visuellen Stile aktivieren (vgl. Abschnitt 9.6.4), z.B.:

```
static void Main() {
    Application.EnableVisualStyles();
    Application.Run(new DM2Euro());
}
```

3) Welche Programmierhilfen bietet der Quellcode-Editor unserer Entwicklungsumgebung (Visual C# 2008 Express Edition oder Visual Studio 2008)?

10 Ausnahmebehandlung

Durch Programmierfehler (z.B. versuchter Feldzugriff mit ungültigem Indexwert) oder durch besondere Umstände (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** aus dem Paket **java.lang** 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 **SystemException** 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 Kommandozeilenargument 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");
            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 **IndexOutOfRangeException** kommt, wenn der Benutzer das Programm *ohne* Kommandozeilenargument 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 Betriebssystem 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.

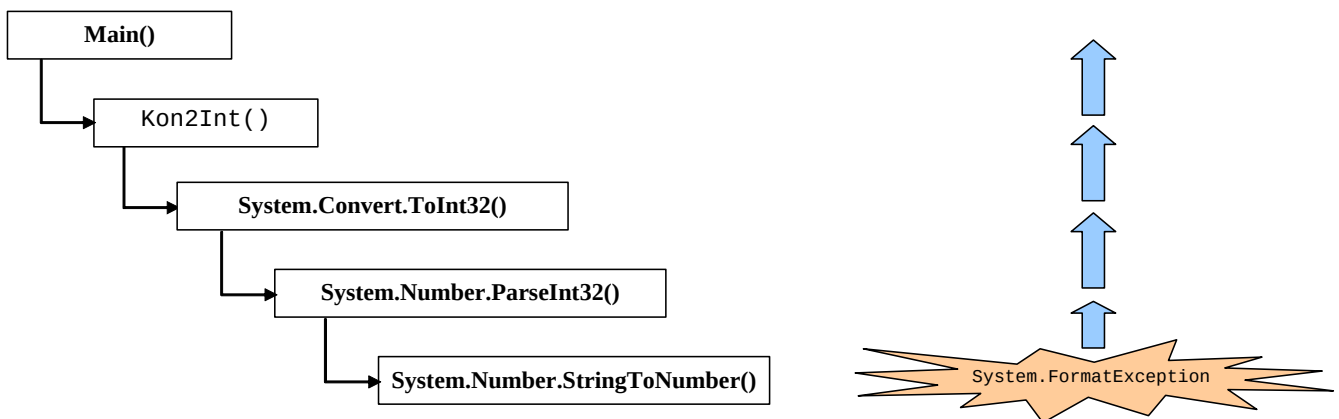
Dies kann als akzeptable Fehlerbehandlung gelten:

- An Stelle der 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 Ar-

gumente vermieden werden. Diese traditionelle Fehlerbehandlung per Rückgabewert ist nicht grundsätzlich als überholt und ineffizient zu bezeichnen, aber in vielen Situationen doch der gleich vorzustellenden Kommunikation über Ausnahmeobjekte unterlegen (siehe Abschnitt 10.3 zum Vergleich von Fehlerrückmeldung und Ausnahmebehandlung).

Trotz seiner präventiven Bemühungen ist das Programm leicht aus dem Tritt zu bringen, indem man es mit einer nicht konvertierbaren Zeichenfolge füttert (z.B. „Vier“). Die zunächst betroffene, 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 instr) in U:\Eigene Dateien\ ... \Fakul.cs:Zeile 6.
    bei Fakul.Main(String[] args) in U:\Eigene Dateien\ ... \Fakul.cs:Zeile 19.
  
```

10.2 Ausnahmen abfangen

Die Startversion des obigen 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 {
    Überwacher 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 elementige 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 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 Ausnahmebasisklassen 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 Formalparameternamens verzichten, hat dann aber im **catch**-Block kein Ausnahmeobjekt (mit Unfallbericht, siehe unten) zur Verfügung:

```

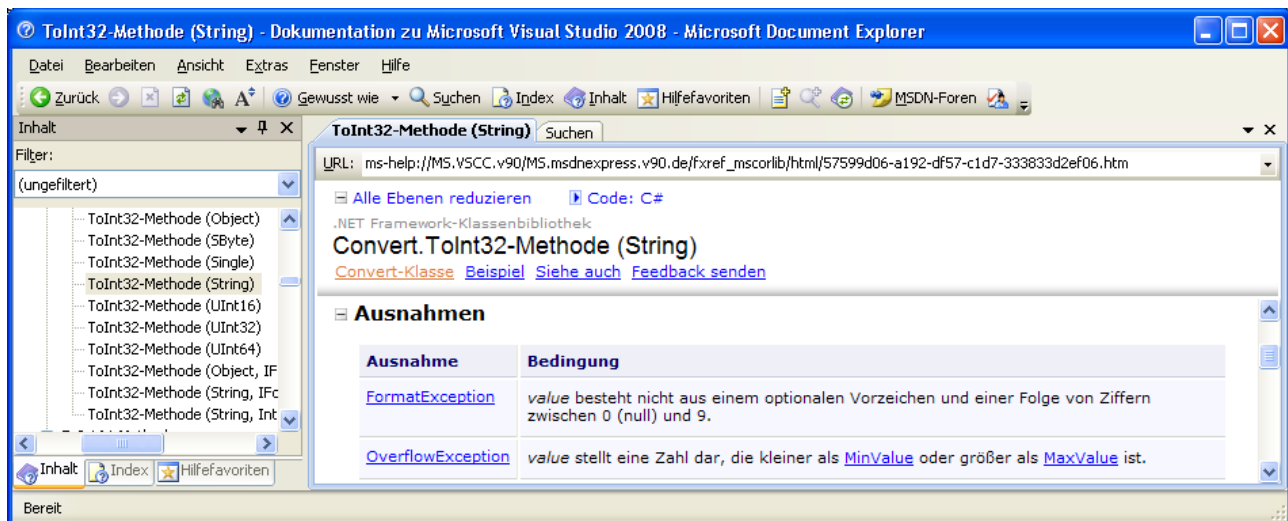
catch (Ausnahmeklasse) {
    Anweisungen für die Behandlung der Ausnahme
}

```

- Fehlt bei einem **catch**-Block auch die Typangabe, ist die (maximal breite) Ausnahme-Basisklasse **Exception** eingestellt, was offensichtlich nur beim *letzten* **catch**-Block sinnvoll ist:

```
catch {
    Anweisungen für die Behandlung der Ausnahme
}
```

Welche Ausnahmen von den Methoden eines FCL-Typs zu befürchten sind, erfährt man in der Dokumentation, z.B. bei der Methode **Convert.ToInt32()**:



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. Im konkreten Beispiel wäre gegen diese Vorgehensweisen nicht viel einzuwenden (siehe Performanzüberlegungen in Abschnitt 10.3). Generell ist die **try-catch** - Ausnahmebehandlung jedoch eine außerordentlich wichtige Option, und hier wurde ein möglichst kurzes Demonstrationsbeispiel benötigt. Dabei sollte sich der **catch**-Block trotz der Kürze nicht auf eine pure Ausgabe zum Existenznachweis beschränken, sondern einen ernsthaften Reparaturversuch unternehmen. Oft wird ein **catch**-Block versuchen, bereits realisierte und aufgrund der Ausnahme nunmehr unerwünschte Effekte des unterbrochenen **try**-Blocks wieder rückgängig zu machen. Viele **catch**-Blöcke betätigen sich als Informationsvermittler und werfen selbst eine Ausnahme, um ihrem 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 Kommandozeilenarguments „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.

Man kann das Abfangen von Ausnahmen auch übertreiben. Richter (2006, S. 455f) warnt davor, in Methoden von Bibliotheksklassen zu viele Ausnahmetypen abzufangen und damit „aus der Welt zu schaffen“, z.B. nach dem folgenden Muster:

```
try {
    . . .
} catch (Exception e) {
    . . .
}
```

Eventuell möchten einige Anwendungen, welche die Bibliotheksmethode verwenden, auf manche Ausnahmetypen reagieren. In diesem Fall ist es besser, einen Ausnahmetyp in der Bibliotheksmethode unbehandelt zu lassen oder nach einer Behandlung erneut zu werfen (siehe Abschnitt 10.6).

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 Abschnitt 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 eigentlich interessanten Fall einer Störung *nach* dem erfolgreichen Öffnen der Datei geht es erst in der zweiten **try**-Anweisung. Eine geöffnete Datei muss 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

Tritt bei der Ausführung einer Anweisung in der Methode `M()` eine Ausnahme auf (von der CLR oder von einer aufgerufenen Methode geworfen), dann sucht das Laufzeitsystem nach einem zuständigen Exception-Handler. Befindet sich die betroffene Anweisung in einem **try**-Block, beginnt die Suche bei den zugehörigen **catch**-Blöcken. Wird das Laufzeitsystem *nicht* fündig, durchsucht es den Aufrufer der Methode `M()` und macht ggf. weiter entlang der Aufrufsequenz. Man kann also ganz einfach (ohne Weiterleitungsaufwand) die Behandlung einer Ausnahme der bestgerüsteten Methode 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 Nützlichkeit und 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:

- Normaler Ablauf
- Exception in `Calc()`, die dort auch behandelt wird
- Exception in `Calc()`, die in **Main()** behandelt wird
- Exception in **Main()**, die nirgends behandelt wird

a) Normaler Ablauf

Beim Programmablauf *ohne* Ausnahmen (hier mit Kommandozeilen-Argument „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 Kommandozeilenargument „Vier“ 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 % Vier = 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 Kommandozeilenargument „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.

```
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 Kommandozeilen-Argument, tritt in **Main()** bei 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

In einer komplexen Methode ist es oft sinnvoll, **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 mit der 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 Wert 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 wird (nach Mössenböck 2005, S. 254):


```

returncode = m1();
if (returncode == 0) {
    returncode = m2();
    if (returncode == 0) {
        returncode = m3();
        if (returncode == 0) {
            . . .
        }
        else {
            // Behandlung für diverse m3()-Fehler}
        }
    }
    else {
        // Behandlung für diverse m2()-Fehler}
    }
}
else {
    // Behandlung für diverse m1()-Fehler
}

```

Mit Hilfe der Ausnahmetechnik bleibt beim Kernalgorithmus die Übersichtlichkeit erhalten:

```

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 Rückgabewertes 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.

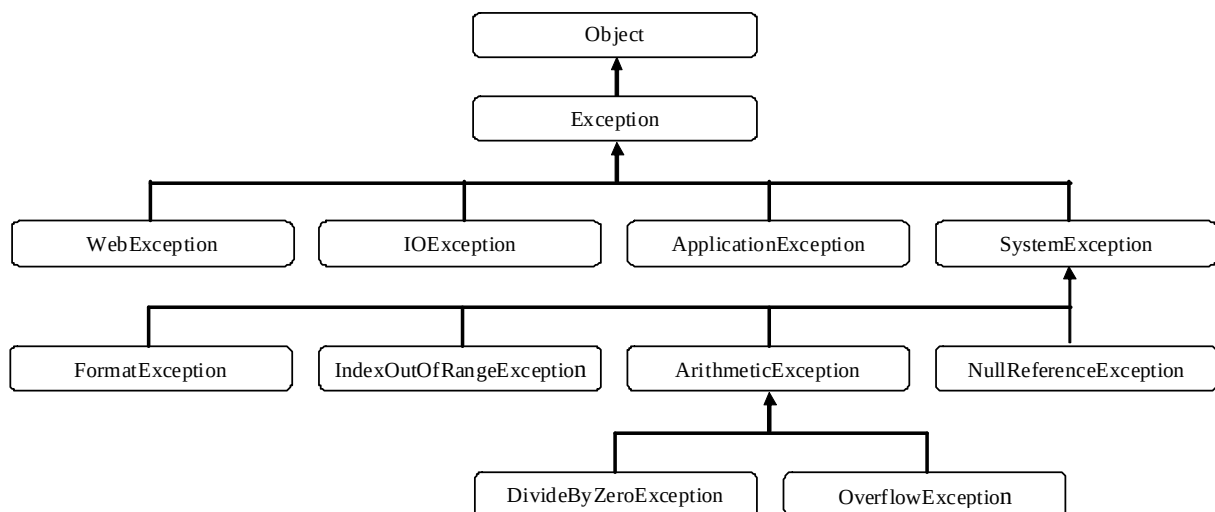
Wie die Realisation des **catch**-Blocks zur **FormatException** in Abschnitt 10.2.1 gezeigt hat, ist die Fehlermeldung per Ausnahmeobjekt dem klassischen Rückgabewert nicht grundsätzlich überlegen. Wenn ein Problem mit erheblicher Wahrscheinlichkeit auftritt, also keinesfalls ungewöhnlich ist, ...

- 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 Performanceinbußen und einen unübersichtlichen Quellcode zur Folge. Hier sollte man es besser auf eine Ausnahme ankommen lassen. Eine Überwachung über 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 **Convert.ToInt32()** 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 6.
bei Fakul.Main(String[] args) in U:\Eigene Dateien\ ... \Fakul.cs:Zeile 19.
```

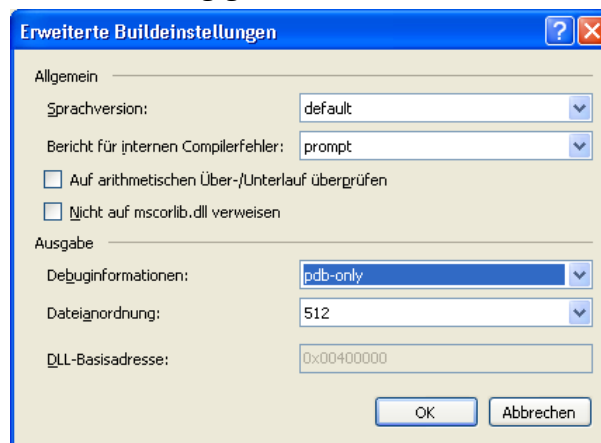
Dateinamen und Zeilennummern enthält die Aufrufreihenfolge nur dann, wenn das betroffene Assembly mit aktiver Debug-Option übersetzt worden ist. Bei direkten Aufruf des Compilers in einem Konsolenfenster ist die Option **debug** anzugeben, z.B.:

```
csc /debug Sequenzen.cs
```

Im Visual Studio 2008 (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.
```

Bis zur .NET - Version 2.0 galt die Empfehlung an Anwendungsprogrammierer, eigene Ausnahmeklassen (siehe Abschnitt 10.6) aus **ApplicationException** abzuleiten. Seit der .NET - Version 3.0 rät die FCL-Dokumentation hingegen:

Bei der Entwicklung von Anwendungen, in denen eigene Ausnahmen erstellt werden müssen, wird empfohlen, benutzerdefinierte Ausnahmen von der **Exception**-Klasse abzuleiten. Ursprünglich sollten benutzerdefinierte Ausnahmen von der **ApplicationException**-Klasse abgeleitet werden. Die Praxis hat jedoch gezeigt, dass sich hierdurch keine wesentlichen Vorteile ergeben.

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 Standardbeispiel von Abschnitt 10 wird ein Ausnahmeobjekt aus der Klasse **ArgumentOutOfRangeException** erzeugt und geworfen, wenn die erfolgreiche Interpretation des Parameters `instr` ein unzulässiges Fakultätsargument ergibt:

```
static int Kon2Int(string instr) {
    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 verwendet, wobei Name und Wert des irregulären Arguments und eine Fehlermeldung anzugeben sind. Aus dem Aufruf

```
try {
    argument = Kon2Int(args[0]);
} catch (Exception e) {
    Console.WriteLine(e.Message);
}
```

resultiert ggf. 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 (siehe Abschnitt 10.6). Damit die ursprüngliche Ausnahme als Anlage beigefügt werden kann, bieten die FCL-Ausnahmeklassen einen Konstruktor mit Parameter vom Typ **Exception** (siehe oben).

In der aktuellen `Kon2Int()`-Variante wird auf die (in Abschnitt 10.2.1 sehr provisorisch ausgeführte) Behandlung der von **Convert.ToInt32** potentiell zu erwartenden Ausnahmen verzichtet. Folglich wirft die aktuelle `Kon2Int()`-Version (als Vermittler oder Initiator) folgende Ausnahmen:

- **FormatException**
- **OverflowException**
- **ArgumentOutOfRangeException**

In einer sorgfältigen Dokumentation müssen Anwender der Methode 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, wobei der „Sender“ per Ausnahmeobjekt übermittelt werden kann. Diese Technik wird im folgenden Beispiel ohne störenden Inhalt demonstriert:

Quellcode	Ausgabe
<pre>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); } }</pre>	A

10.6 Ausnahmen definieren

Mit Hilfe von Ausnahmeobjekten kann eine Methode beim Auftreten von Fehlern den Aufrufer ausführlich und präzise 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.:

```

public sealed class BadFakulArgException : Exception {
    int type, value;
    string input;
    public BadFakulArgException() : base() {
    }
    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_;
        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 Microsoft Empfehlungen für selbst definierte Ausnahmeklasse (siehe z.B. <http://msdn.microsoft.com/de-de/library/87cdya3t.aspx>):

- Als Basisklasse sollte **System.Exception** verwendet werden (vgl. Abschnitt 10.4).
- 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
Bei der Klasse **ArgumentOutOfRangeException** (siehe Abschnitt 10.5) erwartet diese Konstruktor-Überladung allerdings den Name des betroffenen Parameters.
 - Einen Konstruktor mit einem **String**- und einem **Exception**-Parameter für eine Fehlermeldung und den Verweis auf inneres Ausnahmeobjekt, das zuvor aufgefangen wurde und nun in ein informativeres Ausnahmeobjekt als Anlage aufgenommen wird (siehe unten).

Bei serialisierbaren Ausnahmeklassen (siehe Abschnitt 12.2.3) sollte außerdem ein Konstruktor mit Parametern vom Typ **SerializationInfo** und **StreamingContext** implementiert werden, der zur Wiederherstellung eines zuvor per Datenstrom geschriebenen Objekts dient.

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:
 - o 1: Zeichenfolge kann nicht konvertiert werden
 - o 2: **int**-Überlauf
 - o 3: **int**-Wert außerhalb [0, 170]
- in der Eigenschaft **Value** steht das Konvertierungsergebnis (falls vorhanden, sonst -1)

Die endgültige `Kon2Int()`-Version kümmert sich um die **Convert.ToInt32()** zu befürchtenden Ausnahmen und wirft bei allen Fehlerursachen eine spezielle `BadFakulArgException`:

```
static int Kon2Int(string instr) {
    int arg;
    try {
        arg = Convert.ToInt32(instr);
        if (arg < 0 || arg > 170)
            throw new BadFakulArgException("Wert ausserhalb [0, 170]",instr,3,arg);
        else
            return arg;
    }
    catch (OverflowException e) {
        throw new BadFakulArgException("Integer-Überlauf",instr,2,-1,e);
    }
    catch (FormatException e) {
        throw new BadFakulArgException("Fehler beim Konvertieren",instr,1,-1,e);
    }
}
```

Den in **catch**-Blöcken geworfenen `BadFakulArgException`-Objekten wird das aufgefangene Ausnahmeobjekt beigelegt, 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);
    }
    try {
        argument = Kon2Int(args[0]);
    }
    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}", args[0], 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.
```

Hier wird auch das beigelegte *innere* Ausnahmeobjekt angesprochen.

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 **DictionaryEntry**) 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 Ergebnis **NaN** auftaucht (in der Ausgabe: **n. def.**). In folgendem Beispiel wird eine Methode namens **Log2()** zur Berechnung des dualen Logarithmus¹ verwendet, welche auf die FCL-Methode **Math.Log()** zurückgreift und daher bei ungeeigneten Argumenten (≤ 0) als Rückgabewert **Double.NaN** liefert.

¹ Für eine positive Zahl a ist ihr Logarithmus zur Basis b (> 0) definiert durch:

$$\log_b(a) := \frac{\ln(a)}{\ln(b)}$$

Dabei steht $\ln()$ für den natürlichen Logarithmus zur Basis e (Eulersche Zahl).

Quellcode	Ausgabe
<pre>using System; class DualLog { 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); } }</pre>	n. def.

Erstellen Sie eine Version, die bei ungeeigneten Argumenten eine **ArgumentOutOfRangeException** wirft.

4) Erstellen Sie eine Variante der in Abschnitt 7.2.1 vorgestellten generischen Stapelverwaltungs-klasse mit Methoden **Auflegen()** und **Abheben()**, die bei besetztem bzw. leerem Stapel eine **InvalidOperationException**-Ausnahme werfen.

11 Attribute

An Typen (Klassen, Strukturen, Schnittstellen, usw.), Member (Methoden, Eigenschaften, usw.), Parameter und Rückgabewerte von Methoden sowie 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 (abstammend von der abstrakten Basisklasse **System.Attribute**). Der Compiler legt die Attribut-Objekte 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 moderne Option zur *deklarativen Programmierung* auffassen. Sie ergänzen die im C# - Sprachumfang verankerten *Modifikatoren* für Typen, Methoden etc. und bieten dabei eine enorme Flexibilität.

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<string>, 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.

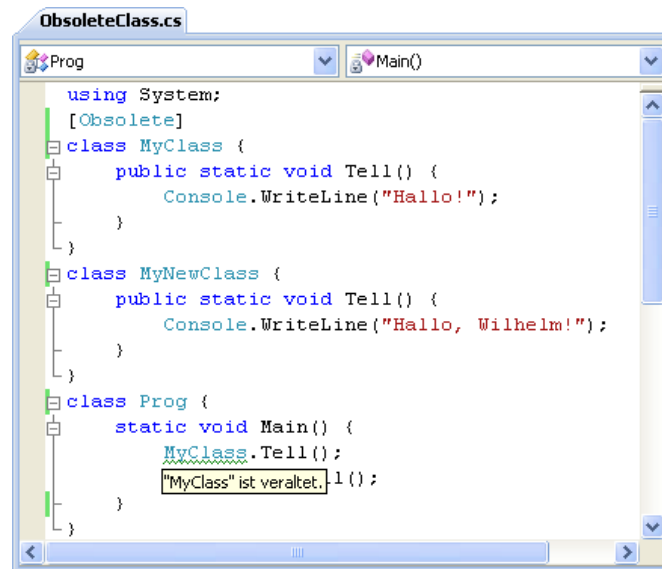
Wir müssen uns nicht auf die Vergabe von FCL-Attributen beschränken, sondern können auch eigene Attribute definieren und verwenden.

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# 3.0 - Sprachspezifikation (Microsoft 2007a, 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:



Im Editor der Visual C# 2008 Express Edition wird der unerwünschte Zugriff auf die obsolete Klasse unterschlängelt, und der Compiler moniert:

ObsoleteClass.cs(19,9): warning CS0612: "MyClass" ist veraltet.

Wer als Klassenbibliotheksdesigner und -renovierer diese Compiler-Meldung zu dürftig findet, kann zum Erstellen des **Obsolete**-Attributs eine alternative Konstruktor-Überladung verwenden und die **Message**-Eigenschaft des Objekts mit einer Zeichenfolge versorgen, z.B.:

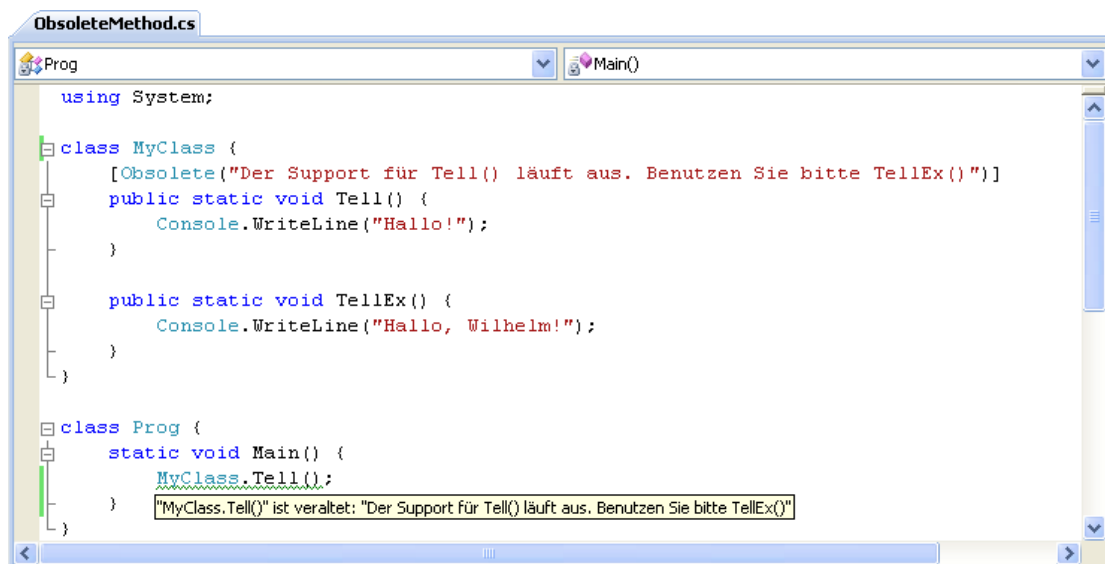
```
[Obsolete("Benutzen Sie bitte MyNewClass")]
```

Dann meldet der Compiler beim unerwünschten Zugriff:

ObsoleteClass.cs(19,9): warning CS0618: "MyClass" ist veraltet:
"Benutzen Sie bitte MyNewClass"

Bei der Vergabe eines Attributes entsteht ein Objekt der entsprechenden Klasse und ergänzt die Metadaten der Trägers. Als syntaktische Besonderheit darf bei der Vergabe eines Attributes (also beim Erstellen eines Attributobjekts) der Namensteil *Attribute* 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:



Beim Übersetzen meldet der Compiler:

```
ObsoleteMethod.cs(16,3): warning CS0618: "MyClass.Tell()" ist  
veraltet: "Der Support für Tell() läuft aus. Benutzen Sie bitte  
TellEx()"
```

In den vom Visual Studio unter Verwendung der Vorlage **Windows Forms-Anwendung** erstellten Projekten ist die Startmethode **Main()** (zu finden in der Quellcodedatei **Program.cs**) mit dem Attribut **STAThread** dekoriert, z.B.:

```
[STAThread]  
static void Main() {  
    Application.EnableVisualStyles();  
    Application.SetCompatibleTextRenderingDefault(false);  
    Application.Run(new Form1());  
}
```

Damit wird festgelegt, dass bei erforderlicher Interoperabilität mit dem Component Object Model (COM), der noch sehr weit verbreiteten Windows-Komponententechnologie, das Threading-Modell STA (*Singlethread-Apartment*) zum Einsatz kommen soll, um Synchronisationsprobleme bei Steuerelementen zu verhindern. Eine WinForms-Anwendung kann z.B. im Zusammenhang mit der Windows-Zwischenablage, bei Standarddialogen (z.B. zur Dateiauswahl) oder beim Drag & Drop (Ziehen & Ablegen) mit der COM-Technologie in Kontakt kommen, und Microsoft empfiehlt die Verwendung des **STAThread**-Attributs nachdrücklich:¹

STAThreadAttribute gibt an, dass das COM-Threadingmodell für die Anwendung Singlethread-Apartment ist. Dieses Attribut muss am Einstiegspunkt jeder Anwendung vorhanden sein, die Windows Forms verwendet. Wird es weggelassen, funktionieren die Windows-Komponenten eventuell nicht richtig. Wenn das Attribut nicht vorhanden ist, verwendet die Anwendung das Multithreaded-Apartmentmodell, das von Windows Forms nicht unterstützt wird.

Bei unseren bisherigen WinForms-Programmen haben wir das Attribut **STAThread** aus didaktischen Gründen weggelassen und dabei keinerlei Probleme festgestellt. Ab jetzt werden wir das Attribut aber richtlinienkonform setzen.

Im weiteren Verlauf von Abschnitt 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 Typen 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.:

¹ Quelle: <http://msdn.microsoft.com/de-de/library/ms182351.aspx>

Quellcode	Ausgabe
<pre>using System; [Obsolete("Benutzen Sie bitte MyNewClass")] class MyClass { public static void Tell() { Console.WriteLine("Hallo!"); } } class Prog { static void Main() { if (Attribute.IsDefined(typeof(MyClass), typeof(ObsoleteAttribute))) Console.WriteLine("MyClass ist obsolet"); } }</pre>	<pre>MyClass ist obsolet</pre>

Die verwendete **IsDefined()**-Überladung erwartet als ersten Parameter ein Objekt der Klasse **MemberInfo**, von der auch die Klasse **Type** abstammt, die einen Datentyp (Klasse, Struktur, Schnittstelle etc.) repräsentiert. Als zweiter Parameter ist das **Type**-Objekt zur fraglichen Attributklasse anzugeben. Im Beispiel werden die beiden **Type**-Objekte per **typeof**-Operator erzeugt, wobei (anders als bei der Attributvergabe, siehe Abschnitt 11.1) auch der Name der Attributklasse vollständig (inkl. Namenbestandteil *Attribute*) zu schreiben ist.

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()**. Der folgende **ObsoleteMethodCheck()** prüft für alle öffentlichen Methoden eines Typs, ob sie als obsolet markiert sind:

```
static void ObsoleteMethodCheck(Type tt) {
    Attribute attrib;
    MemberInfo[] member = tt.FindMembers(MemberTypes.Method,
        BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public,
        Type.FilterName, "*");
    Console.WriteLine("Obsolete-Prüfung für die Methoden der Klasse {0}:",
        tt.FullName);
    foreach (MemberInfo mi in member) {
        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);
    }
}
```

Das per Parameter übergebene **Type**-Objekt wird per **FindMembers()** aufgefordert, Information über seine Member in einem Array vom Typ **MemberInfo** (aus dem Namensraum **System.Reflection**) zu liefern:

- Im ersten **FindMembers()**-Parameter wählt man die Member-Kategorie.
- Im zweiten Parameter legt man per ODER-Verknüpfung von Werten der Enumeration **BindingFlags** fest, dass öffentliche Instanz- und Klassen-Member gefragt sind.
- Der dritte Parameter sorgt für eine namensorientierte Filterung.
- Im letzten Parameter werden per Jokerzeichen beliebige Namen einbezogen.

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>

Auf einige *Assembly*-Attribute (z.B. mit Versionsangaben) kann man sehr bequem über statische Eigenschaften der Klasse **Application** zugreifen (siehe Abschnitt 11.4).

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 Konstruktor-Parameter **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 27.1.2).

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: AssemblyVersion("1.4.2.3")]
[assembly: AssemblyDescription("Ideale Tools für Ihr Projekt")]
[assembly: AssemblyCompany("Marco Soft")]
[assembly: AssemblyProduct("YourTools")]
```

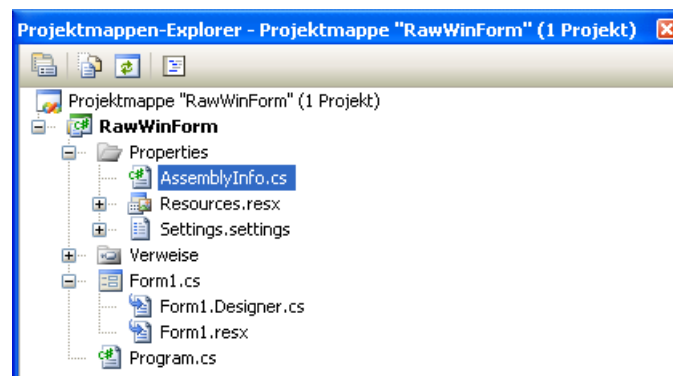
Hier wird jeweils das vom Compiler zu erzeugende Assembly als Träger des nachfolgenden Attributs festgelegt.

Derartige Assembly-Attribut - Deklarationen erzeugt das Visual Studio beim Erstellen eines neuen Projekts unter Verwendung der Vorlage **Windows Forms-Anwendung** automatisch, wobei die Entwicklungsumgebung eine Datei namens **AssemblyInfo.cs** anlegt, z.B. (Kommentare aus Platzgründen weggelassen):

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

[assembly: AssemblyTitle("RawWinForm")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("Universität Trier")]
[assembly: AssemblyProduct("RawWinForm")]
[assembly: AssemblyCopyright("Copyright © Universität Trier 2009")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
[assembly: ComVisible(false)]
[assembly: Guid("65c809f4-3766-4cf8-8b1d-0de532780974")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

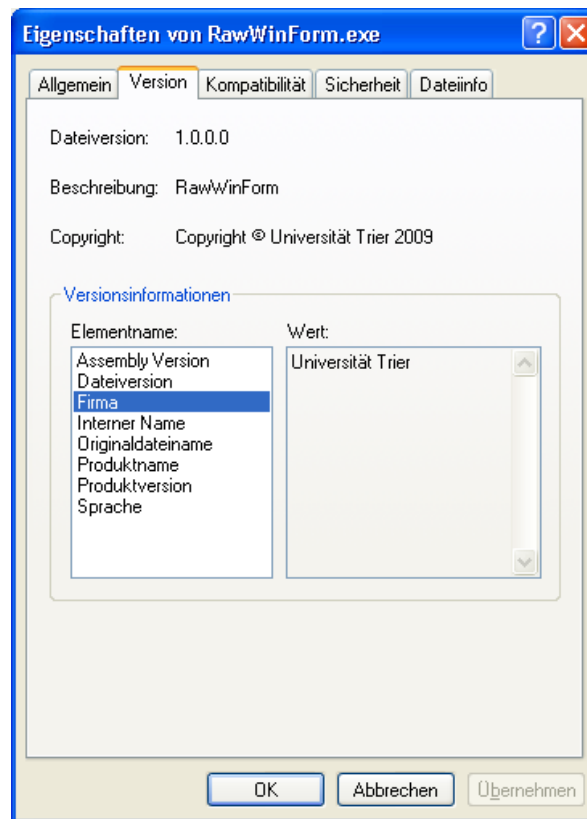
Zur Modifikation dieser Attribute öffnen man die Datei **AssemblyInfo.cs** über den Projektmappe-nexplorer:



Auf einige Assembly-Attribute kann man im Programm über statische Eigenschaften der Klasse **Application** (im Namensraum **System.Windows.Forms**) zugreifen, z.B.:

```
private void button1_Click(object sender, EventArgs e) {
    MessageBox.Show("Version "+Application.ProductVersion,
        Application.ProductName, MessageBoxButtons.OK,
        MessageBoxIcon.Information);
}
```

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 in Abschnitt 9.4.4.2 vorgestellten Enumeration **AnchorStyles** stehen die ersten vier Bits (mit den dezimalen Wertigkeiten 2^0 , 2^1 , 2^2 und 2^3) für die Verankerung eines Steuerelements an den vier Seiten des elterlichen Containers¹:

```
[Flags]
public enum AnchorStyles {
    None = 0,
    Top = 1,
    Bottom = 2,
    Left = 4,
    Right = 8
}
```

- Jede bitweise ODER-Kombination von zwei benannten Werten der Enumeration ergibt ein sinnvoll interpretierbares Bitfeld, also eine zulässige Kombination der dichotomen Einzelmerkmale. WinForms-Steuerelemente haben z.B. die folgende **AnchorStyles**-Voreinstellung (Verankerung links und oben):

```
AnchorStyles.Left | AnchorStyles.Top
```

¹ Der Quellcode wurde leicht abgewandelt aus dem Mono-Projekt (http://www.mono-project.com/Main_Page) übernommen.

Bei einem gewöhnlichen Enumerationstyp (ohne **FlagsAttribute**) ...

- stehen die Werte für die sich *gegenseitig ausschließenden* Ausprägungen eines Merkmals. Bei der in Abschnitt 9.4.4.2 vorgestellten Enumeration **DockStyle** kodieren die ersten sechs nicht-negativen **int**-Werte jeweils ein Andockverhalten gegenüber dem elterlichen Container:

```
public enum DockStyle {
    None = 0,
    Top = 1,
    Bottom = 2,
    Left = 3,
    Right = 4,
    Fill = 5
}
```

- Die (syntaktisch erlaubte und vom Compiler nicht bemängelte) ODER-Verknüpfung von zwei benannten Werten führt in der Regel *nicht* zum erwarteten Ergebnis. Z.B. liefert `DockStyle.Top | DockStyle.Left` denselben Wert Drei wie `DockStyle.Left`

Das **FlagsAttribute** dient nicht nur zur Kommunikation zwischen dem Designer und dem Benutzer eines Enumerationstyps. Auch der Designer der FCL-Enumerationsbasisklasse **Enum** überprüft die Existenz des Attribut per **IsDefined()**-Aufruf (vgl. Abschnitt 11.2)¹

```
eT.IsDefined(typeof(System.FlagsAttribute), false)
```

und liefert in seiner **ToString()** - Überschreibung ggf. eine kommaseparierte Liste der Merkmale mit einem angeschalteten Bit, z.B.:

Quellcode	Ausgabe
<pre>using System; using System.Windows.Forms; class Prog { static void Main() { Console.WriteLine("AnchorStyles-Werte:"); for (AnchorStyles i = 0; (int) i < 8; i++) Console.WriteLine(" {0}: {1}", (int) i, i); Console.WriteLine("\nDockStyle-Werte:"); for (DockStyle i = 0; (int)i < 8; i++) Console.WriteLine(" {0}: {1}", (int)i, i); } }</pre>	<pre>AnchorStyles-Werte: 0: None 1: Top 2: Bottom 3: Top, Bottom 4: Left 5: Top, Left 6: Bottom, Left 7: Top, Bottom, Left DockStyle-Werte: 0: None 1: Top 2: Bottom 3: Left 4: Right 5: Fill 6: 6 7: 7</pre>

11.5.2 Unions per StructLayoutAttribute und FieldOffsetAttribute

Bei der in Abschnitt 3.3.4.1 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

¹ Der Quellcode stammt aus Microsofts *Shared Source Common Language Infrastructure 2.0*.

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 **StructLayoutAttribut** mit Konstruktor-Parameter **LayoutKind.Explicit** wird dem Compiler mitgeteilt, dass die Speicheradressen der beiden Felder explizit durch **FieldOffsetAttribut** festgelegt werden sollen. So wird es möglich, die beiden (gleich langen) Felder an derselben Anfangsadresse Null beginnen zu lassen.

Das Programm schreibt den vom Benutzer gewünschten `float`-Wert in die vier `Union`-Bytes und liest anschließend aus demselben Speicherbereich einen `int`-Wert, der bequem 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.WriteLine("float: ");
        f = Convert.ToSingle(Console.ReadLine());
        Console.WriteLine("\nBits: ");
        uni.f = f;
        int bits = uni.i;
        Console.WriteLine("1 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 `NonsenseAttribute`-Objekts ausgibt.

12 Ein-/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 Abschnitt 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 Abschnitt über Netzwerkprogrammierung verwendbar sein.

Die .NET - Klassen zur Datenein- und -ausgabe befinden sich im Namensraum **System.IO**, der folglich in der Regel importiert werden sollte:

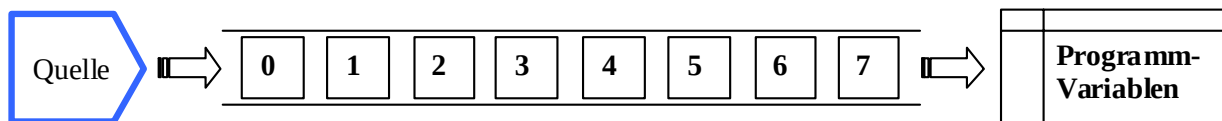
```
using System.IO;
```

12.1 Datenströme aus Bytes

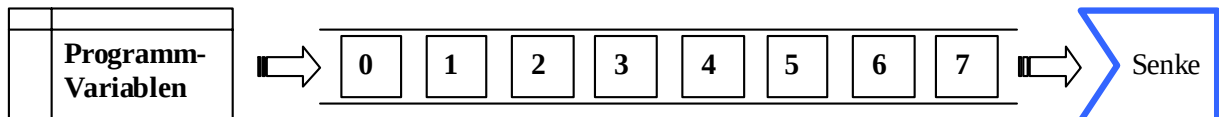
12.1.1 Das Grundprinzip

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):



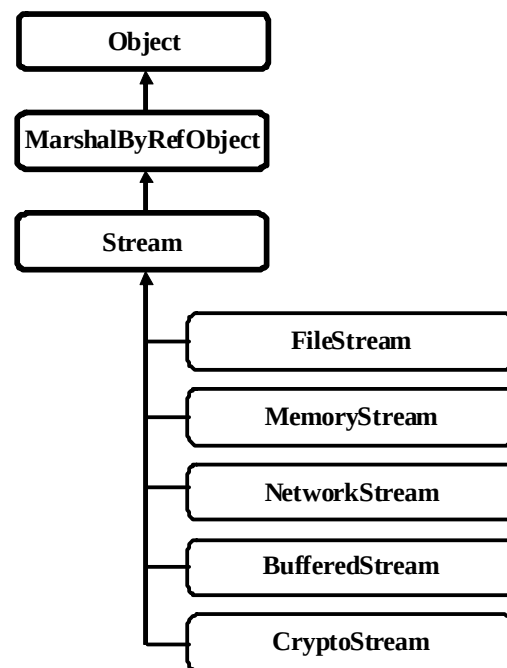
Ein- bzw. Ausgabeströme werden in .NET - Programmen durch Objekte aus geeigneten Klassen des Namensraums **System.IO** repräsentiert, wobei die Auswahl u.a. von der angeschlossenen Datenquelle bzw. -senke (z.B. Datei versus Netzwerkverbindung) sowie vom Typ der zu transportierenden Daten abhängt.

Ziel des Datenstromkonzeptes ist es, Ein- und Ausgaben möglichst unabhängig von den Besonderheiten konkreter Datenquellen und -senken formulieren zu können.

Nach so vielen allgemeinen bzw. abstrakten Bemerkungen wird es Zeit für ein konkretes Beispiel, wobei der Einfachheit halber auf Sinn und Sicherheit 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:



Zu den noch unklaren Details im Beispielprogramm folgen Erläuterungen in den nächsten Abschnitten.

12.1.2 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 12.1.1 wird die Lese-/Schreibposition 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 aus Performanzgründen einen Puffer, z.B. um die Anzahl der zeitaufwendigen Dateizugriffe 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** (siehe unten) 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 12.1.3.
- **public long Length {get;}**
Mit dieser Eigenschaft wird die Länge des Stroms (z.B. die Dateigröße) in Bytes angesprochen.
- **public long Seek(long offset, SeekOrigin 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 und Positionieren erlaubt, z.B.:

Quellcode	Ausgabe
<pre>Console.WriteLine("CanRead: " + fs.CanRead); Console.WriteLine("CanSeek: " + fs.CanSeek); Console.WriteLine("CanWrite: " + fs.CanWrite);</pre>	CanRead: True CanSeek: True CanWrite: True

12.1.3 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 Lesen- 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 (siehe Richter 2006, S. 501).

Derartige Pannen sind ausgeschlossen, wenn Sie ein überflüssig gewordenes Datenstromobjekt dem Garbage Collector überlassen. Über die Finalisierungsmethode 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 Aufräumarbeiten (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 Aufräumarbeiten des Garbage Collectors sind allerdings Zeitpunkt und Reihenfolge unbestimmt, so dass bei schreibenden Datenströmen das explizite Schließen oft erforderlich ist. 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.

Außerdem ist es oft wichtig, Ströme durch ein explizites **Close()** so früh wie möglich zu schließen, um (exklusive) Zugriffe durch andere Programme zu ermöglichen.

Im Beispiel FSDemo von Abschnitt 12.1.1 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);
    }
}
```

Per **using**-Anweisung wird das Schließen eines Stroms elegant gelöst, wobei der Compiler im Hintergrund eine **try-finally** - Anweisung erstellt. In vielen Fällen ist es aber auch sinnvoll bzw. erforderlich, die von Ein-/Ausgabe – Methoden zu erwartenden Ausnahmen zu behandeln.

12.1.4 Ausnahmen abfangen

Weil Methodenaufrufe bei Datenstromobjekten diverse Ausnahmen produzieren können (z.B. **FileNotFoundException**, **UnauthorizedAccessException**, **IOException**), sind sie am besten in einem **try** – Block untergebracht. Wenn ein **Close()**-Aufruf angemessen ist (vgl. die Warnungen und Empfehlungen in Abschnitt 12.1.3), 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 12.1.3). In der folgenden Variante des Einstiegsbeispiels ist das Abfangen der Ausnahmen nur angedeutet:

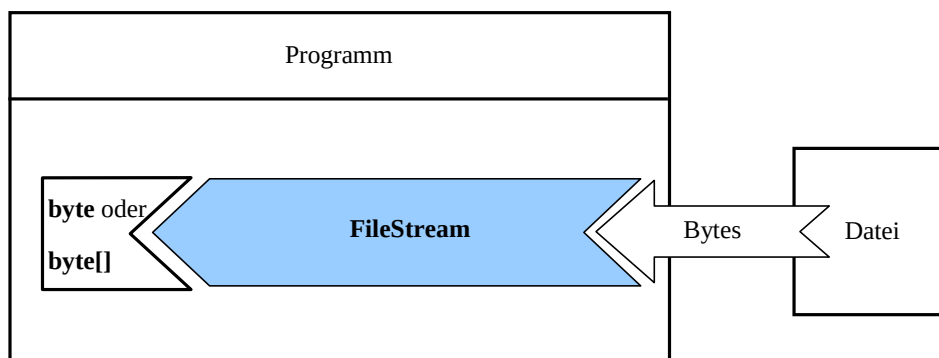
```
using System;
using System.IO;

class FSDemo {
    static void Main() {
        String name = "demo.bin";
        byte[] arr = {0,1,2,3,4,5,6,7};
        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);
        }
    }
}
```

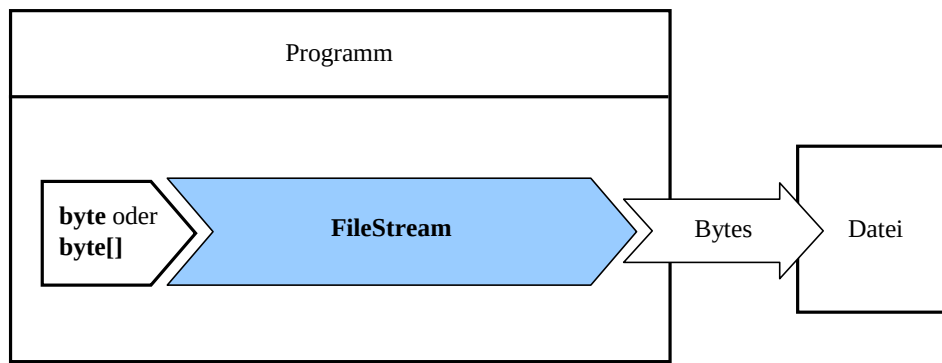
Bei den Beispielen im weiteren Verlauf von Abschnitt 12 werden wir der Einfachheit halber meist auf eine Ausnahmebehandlung verzichten.

12.1.5 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.

12.1.5.1 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	Es wird eine neue Datei angelegt oder eine vorhandene überschrieben.
CreateNew	Es wird eine neue Datei angelegt, oder eine IOException geworfen, falls eine Datei mit dem gewünschten Namen bereits existiert.
Open	Es wird eine vorhandene Datei geöffnet, oder eine IOException geworfen, falls keine Datei mit dem gewünschten Namen existiert.
OpenOrCreate	Es wird eine neue Datei erzeugt oder eine vorhandene geöffnet, jedoch im Unterschied zu Create <i>nicht</i> automatisch überschrieben.
Truncate	Es wird eine vorhandene Datei geöffnet und entleert, oder eine IOException geworfen, falls keine Datei mit dem gewünschten Namen existiert.

Beim Öffnungsmodus **Append** befindet sich der Schreibposition am Ende der Datei, und es ist kein lesender Zugriff möglich. Mit den anderen Öffnungsmodi sind keine Zugriffsbeschränkungen verbunden, und der Dateizeiger steht am Anfang.

12.1.5.2 Zugriffsmöglichkeiten für den eigenen Prozess

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);
```

12.1.5.3 Optionen für gemeinsamen Zugriff

Der Eröffnungsmodus ist auch relevant für die Rechte anderer Prozesse zum simultanen Dateizugriff. So erlauben z.B. die Eröffnungsmodi **FileMode.Append** und **FileMode.Create** ein zusätzliches Öffnen zum Lesen. 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);
```

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.

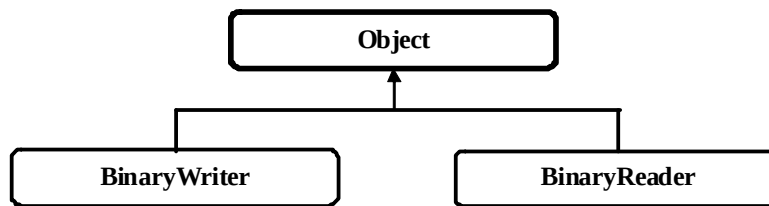
Bei den Dateien auf einem Rechner kann man unterscheiden:

- **Binärdateien**
Ein Programm kann Daten beliebigen Typs problemlos in eine Binärdatei schreiben oder aus einer Binärdatei mit bekanntem Aufbau lesen. Ein simpler Texteditor zeigt hingegen nach dem Öffnen einer Binärdatei nur eine wirre Folge von (Sonder-)zeichen an. 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 einem Texteditor gelesen und/oder bearbeitet werden, sofern beim Erstellen der Datei eine passende Kodierung gewählt wurde. 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

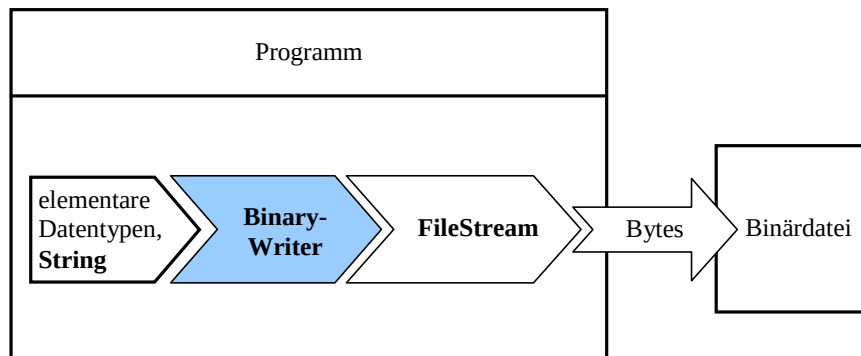
Um Werte von einem beliebigen elementaren Datentyp (z.B. **int**, **double**) sowie Zeichenfolgen in einen binär organisierten Strom (z.B. in eine Binärdatei) zu schreiben bzw. aus einem Binärstrom mit bekanntem Aufbau zu lesen, verwendet man ein Objekt der Klasse **BinaryWriter** bzw. **BinaryReader**.



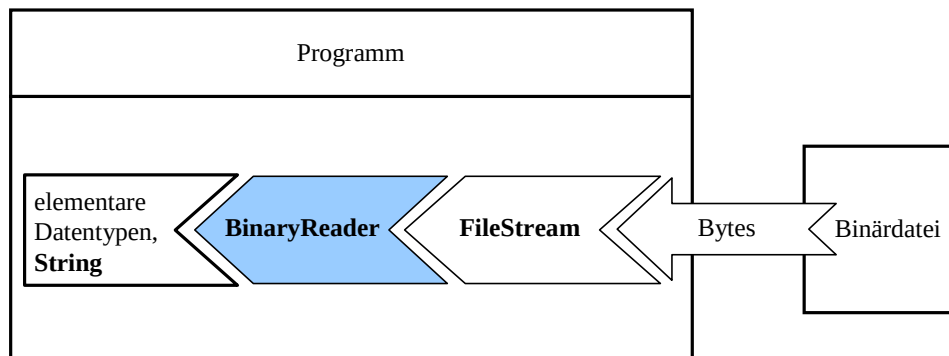
Die beiden Klassen stammen *nicht* von **Stream** ab, verwenden aber für die Verbindung mit einer Datenquelle oder –senke ein **Stream**-Objekt, das im Konstruktor anzugeben ist, z.B.:

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());
    }
}

```

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. Auch ein **Flush()**-Aufruf an ein **BinaryWriter**-Objekt zum Entleeren des Puffers wird durchgereicht.

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 weiter¹.

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

```

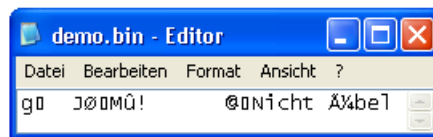
Es macht wenig Sinn, die vom Beispielprogramm erzeugte Binärdatei mit einem Texteditor zu öffnen:

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);
    OutputStream.Write(_buffer, 0, 4);
}

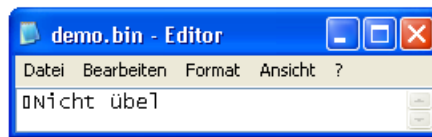
```



Im Beispiel wird die vom **BinaryWriter** geschriebene Zeichenfolge allerdings vom Texteditor (fast) korrekt dargestellt. Bei der Ausgabe von **String**-Variablen ist ausschließlich die verwendete *Kodierung* relevant, und hier verwendet die Klasse **BinaryWriter** per Voreinstellung dasselbe **UTF8Encoding** wie die später vorzustellenden **TextWriter**-Klassen. 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 Ausgabe ausschließlich Text, kann der Windows-Editor beim Öffnen der Ergebnisdatei die zugrunde liegende Kodierung 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 **BinaryReaders** vor jede Zeichenfolge ihre Länge (Anzahl der Bytes) und verwendet dabei den Datentyp **uint** mit einer speziellen 7-Bit - Kodierung: Jeweils 7 Bits werden als ein Byte ausgegeben, wobei im führenden Bit eine Eins signalisiert, dass noch ein weiteres 7-Bit-Paket 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

Länge der Zeichenfolge in Bytes	uint-Bits	Längenpräfix
11	0...0 00000000 00001011	00001011
258	0...0..00000001 00000010	10000010 00000010

Bei einer Zeichenfolge mit 258 Bytes Länge resultiert ein Längenpräfix mit zwei Bytes (siehe Tabelle).

Es besteht übrigens kein Risiko, wenn ein Gespann aus einem **BinaryWriter**-Objekt und einem **FileStream**-Objekt dem Garbage Collector anheim fallen, obwohl beim automatischen Finalisieren (ohne garantierte Reihenfolge!) zuerst das **FileStream**-Objekt beseitigt werden könnte:

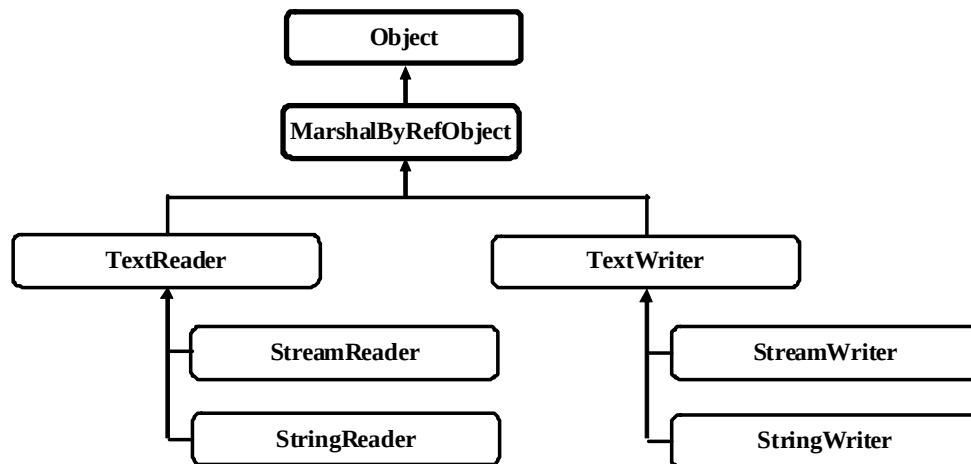
- In der Klasse **BinaryWriter** ist keine Finalisierungsmethode vorhanden, so dass beim Abräumen kein Zugriff auf das zugrunde liegende (und eventuell nicht mehr existente) **Stream**-Objekt stattfinden kann.
- **BinaryWriter**-Objekte besitzen keinen lokalen Puffer, der beim Abräumen geleert werden müsste.

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) sicher gestellt ist (eventuell per **using**-Block automatisiert).

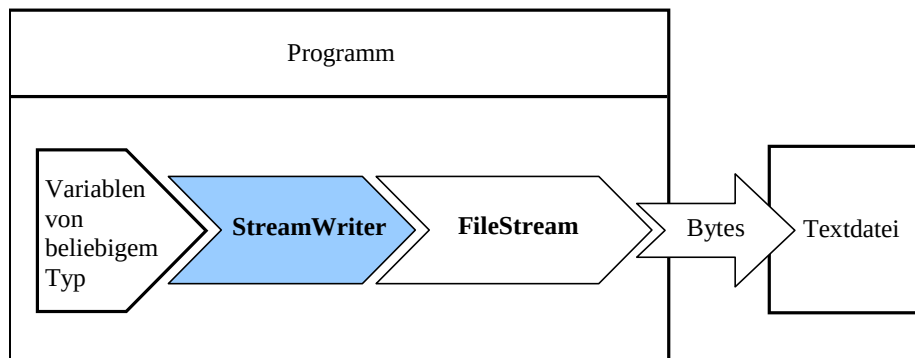
12.2.2 Schreiben und Lesen im Textformat

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

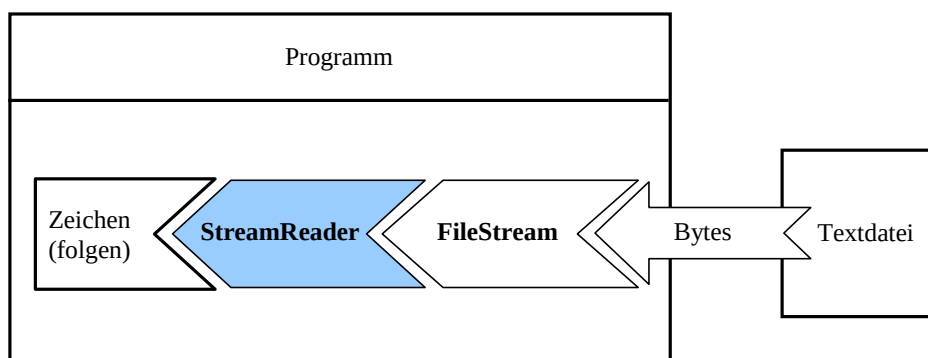
Übungsaufgabe in Abschnitt 12.4). Beide Klassen sind abstrakt, doch bietet die FCL auch konkrete Ableitungen für **Stream**- bzw. **String**-Objekte als Senken bzw. Quellen:



Um in eine *Textdatei* zu schreiben bzw. von dort zu lesen, verwendet man Objekte der Klassen **StreamWriter** bzw. **StreamReader**, die jeweils über eine Instanzvariable auf ein **FileStream**-Objekt aufsetzen (wie bei den Klassen **BinaryWriter** und **BinaryReader**)¹. Beim Schreiben haben wir also folgende Situation:



Beim Lesen aus einer Textdatei reisen die Daten in umgekehrter Richtung:



Man kann den Basisstrom für einen **StreamWriter** oder **-Reader** auch *implizit* erzeugen lassen, wenn die voreingestellten Kurationsparameter 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.

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.

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();

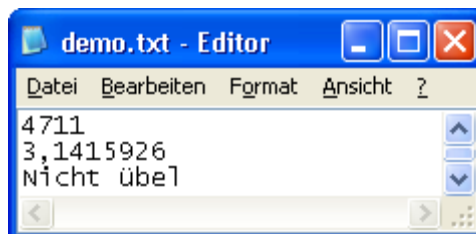
        StreamReader sr = new StreamReader(
            new FileStream(name, FileMode.Open, FileAccess.Read));
        Console.WriteLine("Inhalt der Datei {0}\n",
            ((FileStream)sr.BaseStream).Name);
        for (int i = 0; sr.Peek() >= 0; i++) {
            Console.WriteLine("{0}:\t{1}", i, sr.ReadLine());
        }
    }
}
```

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 Geräten sinnvoll) oder zwischengepuffert werden (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 ausgegeben
- 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.
- **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.
- **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 ausgegeben werden, ohne die Speicherplatzverschwendung durch führende Null-Bytes bei den sehr oft auftretenden ASCII-Zeichen (mit Unicode-Nummern ≤ 127) in Kauf nehmen zu müssen:

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);
```

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);
}
```

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 **ANSI-Textdateien** 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());
        Console.ReadLine();
    }
}
```

wird eine ANSI-Textdatei mit folgendem Inhalt

ANSI-kodierte Umlaute: üöä

zuerst mit der ungeeigneten UTF8- und dann mit dem korrekten ANSI-Kodierung eingelesen, was zu folgender Ausgabe führt:

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 Systems passende Kodierung.

12.2.3 Serialisieren von Objekten

Wer objektorientiert programmiert, möchte natürlich auch objektorientiert speichern und laden. Erfreulicherweise können in C# Objekte von Klassen tatsächlich genau so einfach wie elementare Datentypen in einen Datenstrom geschrieben bzw. von dort gelesen werden. Die Übersetzung eines Objekts mit all seinen Instanzvariablen und den enthaltenen (d.h. von Feldern 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 spricht von (De)serialisieren eines *Objektdiagramms*. Dank Autoboxing lassen sich auch Strukturinstanzen (de)serialisieren.

Das Serialisieren der Instanzen eines Typs muss explizit über das Attribut **Serializable** oder das Implementieren der Schnittstelle **ISerializable** erlaubt werden. Bei der letztgenannten Variante gewinnt man Kontrolle über die (De)serialisierung, muss aber auch einen hohen Eigenbeitrag leisten. Wir beschränken uns auf bequeme Attribut-Vergabe, die aber nicht unbedacht erfolgen darf, 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**.

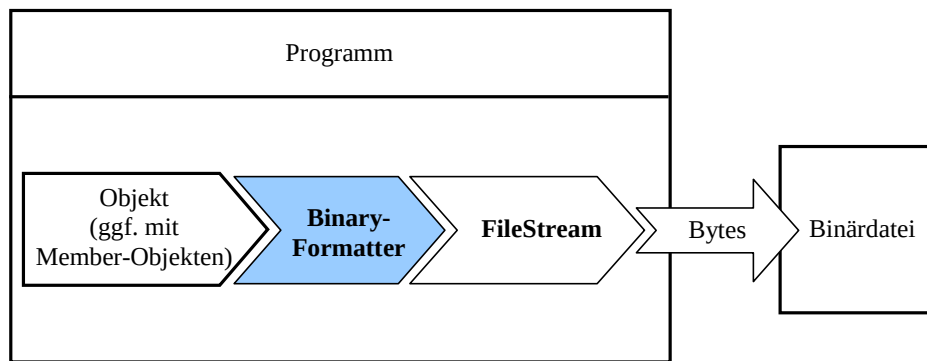
Die in folgender Quelle definierte Klasse **Kunde** ist als serialisierbar definiert, wobei jedoch für das Feld **stimmung** eine Ausnahme gemacht wird:

```
using System;
using System.Runtime.Serialization;

[Serializable]
public class Kunde {
    int nr;
    string vorname;
    string name;
    [NonSerialized]
    int stimmung;
    int nkaeufe;
    double aussen;
    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 nicht gerade trivialen 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 werden zwei **Kunde**-Objekte mitsamt den 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();
    }
}

```

Pro **Serialize()**-Aufruf wird *ein* Wurzelobjekt mitsamt den Werttyp-Feldern sowie den direkt oder indirekt referenzierten Objekten geschrieben. Um weitere Wurzelobjekte 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
```

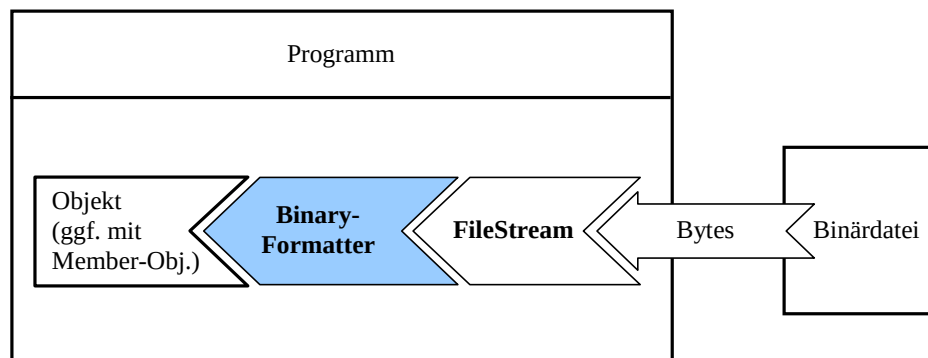
Rekonstruiert:

```
Kundennummer: 1
Name:         Fritz Orth
Stimmung:     0
Anz.Einkäufe: 13
Aussenstände: 426,89

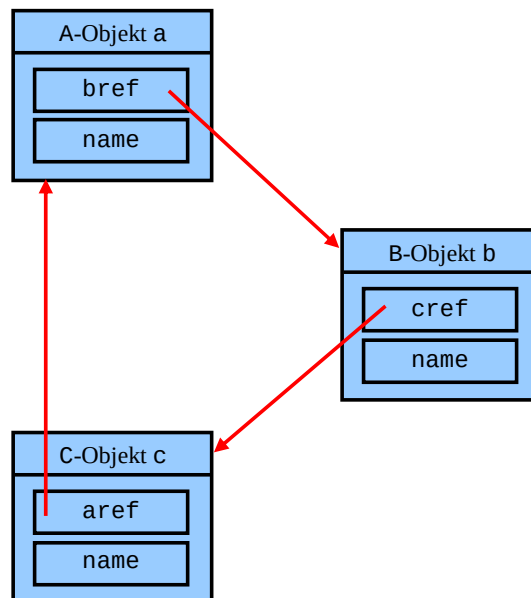
Kundennummer: 2
Name:         Ludwig Knüller
Stimmung:     0
Anz.Einkäufe: 17
Aussenstände: 89,1
```

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 **a**, **b** und **c** 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 Objekt **a** 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 von **a**

```

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
```

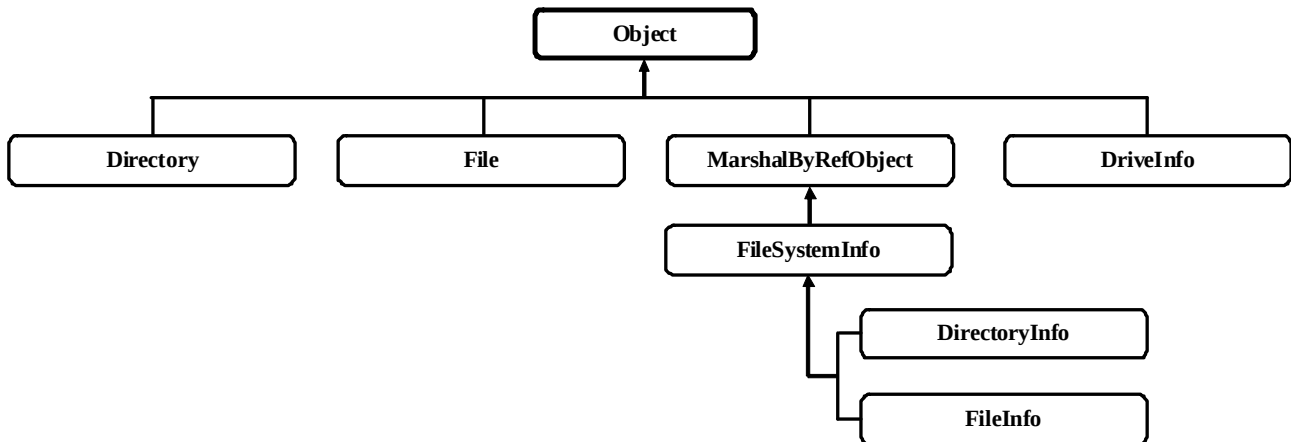
Unter ihren früheren Namen (**a**, **b** und **c**) sind die rekonstruierten Objekte *nicht* mehr ansprechbar, weil es sich hier um die Namen von lokalen Variablen (der Methode **Main()**) handelt, die beim Serialisieren nicht berücksichtigt werden.

Ist eine XML-basierte Ausgabe gefragt, kommen an Stelle des **BinaryFormatters** die folgenden FCL-Klassen in Frage:

- **SoapFormatter**
Diese Klasse produziert das XML-Format des SOAP-Internetprotokolls (*Simple Object Access Protocol*) benutzt. Allerdings wird die Klasse nicht mehr weiterentwickelt und unterstützt z.B. keine generischen Typen.
- **XmlSerializer**
Diese Klasse erlaubt über Attribute eine Gestaltung der XML-Ausgabe. Im Unterschied zu den Klassen **BinaryFormatter** und **SoapFormatter** werden 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 löschen, kopieren, umbenennen und verschieben sowie diverse Datei- bzw. Verzeichnisattribute einsehen und verändern. Im Zusammenhang mit dem **Tree-View**-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 3.3.8.5 über die Syntax von Zeichenkettenliteralen besprochen, wird mit dem Präfix „@“ vor einer Zeichenkette die Auswertung von Escape-Sequenzen abgeschaltet, so dass die in Windows-Pfadnamen üblichen Rückwärtsschrägstrich nicht mehr durch Verdoppeln von ihrer Sonderfunktion befreit beraubt 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)**
Von den zahlreichen Methoden zum Modifizieren von Dateiattributen wird im obigen Beispiel **SetLastWriteTime()** verwendet. Damit lässt sich 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("Textdateien 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()**
 Bei Aufruf seiner Instanzmethode **GetFiles()** liefert ein **DirectoryInfo**-Objekt einen Array mit **FileInfo**-Objekten zu allen Dateien im Ordner, wobei auch Dateiauswahlfilter mit Jokerzeichen möglich sind.
- **public DirectoryInfo[] GetDirectories()**
 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 die Veränderungen bei den Textdateien (Extension **.txt**) in dem per Kommandozeile angegebenen Ordner:

```
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("Überwacher 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

Überwacher 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 ab .NET 2.0 verfügbaren **Console**-Methode **ReadKey()** kann man sofort auf Tastendrucke 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. 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 Threads

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 Programmierer 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. nicht untätig den quälend langsamen Aufbau einer Seite abwarten, sondern kann in einem anderen Browser-Fenster Suchbegriffe eingeben etc.

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.

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).

Die zu einem Prozess gehörenden Threads laufen im selben Adressraum ab und verwenden einen gemeinsamen Heap. Allerdings benötigt jeder Thread als selbständiger Kontrollfluss bzw. Ausführungsfaden einen eigenen Stack.

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 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 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.

13.1 Threads erzeugen

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 Deleгатentyp

public delegate void ThreadStart()

oder den Deleгатentyp

public delegate void ParameterizedThreadStart(Object obj)

erfüllt, kann in einem eigenen (dedizierten) Thread gestartet werden, indem ein zugehöriges Deleгатenobjekt erzeugt und dem **Thread**-Konstruktor übergeben wird, z.B.:

```
Thread pt = new Thread(new ThreadStart(pro.Run));
```

Man kann sich das explizite Notieren des Deleгатen-Konstruktors sparen:

```
Thread pt = new Thread(pro.Run);
```

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.

Das **Lager**-Objekt führt die Aufträge **Ergaenze()** und **Liefere()** aus, solange nicht eine Maximalzahl von Lagerzugriffen überschritten ist:

```
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("Der Laden 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();
}
}

```

Die **Main()**-Methode der Klasse **Lager** erzeugt als Startmethode des Programms die beteiligten Objekte:¹

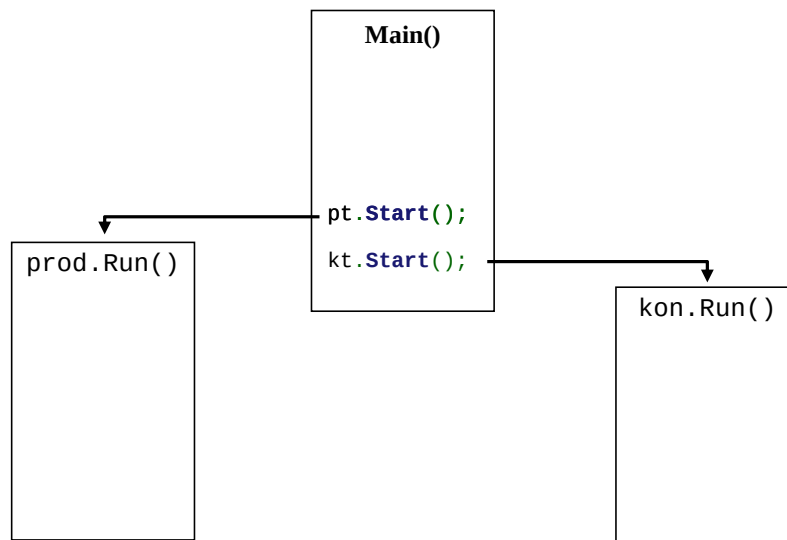
- einen Lageristen (Objekt **lager** aus der Klasse **Lager**)
- einen Produzenten (Objekt **pro** aus der Klasse **Produzent**)
- einen Konsumenten (Objekt **kon** aus der Klasse **Konsument**)
- einen Thread, dessen Ausführung mit der **Run()**-Methode des Produzenten startet (Objekt **pt** aus der Klasse **Thread**)
- einen Thread, dessen Ausführung mit der **Run()**-Methode des Konsumenten startet (Objekt **kt** aus der Klasse **Thread**)

Schließlich erhalten die Threads einen Namen und werden gestartet.

Unmittelbar vor dem Ende der **Main()**-Methode sind **drei** Threads aktiv:

- 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).

¹ Das Beispiel mit seinem recht reichhaltigen Objekt-Ensemble demonstriert übrigens, dass einige Objekte direkt aus der Abbildung des Anwendungsbereichs stammen (Lagerist, Produzent, Konsument), während andere Objekte einen informationstechnologischen Ursprung haben (die beiden Threads).



Die 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-Thread¹.

Dass beim Ende der **Main()**-Methode die einzigen Referenzen auf die **Thread**-Objekte (**pt** und **kt**) verschwinden, spielt keine Rolle.

Weil Produzent und Konsument mit dem **Lager**-Objekt kooperieren sollen, erhalten sie als Konstruktor-Parameter eine entsprechende Referenz.

Der Lagerist beherrscht die folgenden Methoden, um den Produzenten oder den Konsumenten zu bedienen:

- `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.

Produzent und Konsument kommen mit einer recht simplen Klassendefinition aus:

¹ Ist ein Thread über seine Eigenschaft **IsBackground** als **Hintergrund-Thread** markiert, dann kann er einen Prozess nicht aufrecht erhalten, sondern wird ggf. automatisch mit dem letzten Vordergrund-Thread beendet.


```

using System;
using System.Threading;
public class Produzent {
    Lager pl;
    bool offen;

    public Produzent(Lager ptr) {
        pl = ptr;
    }
    public void Run() {
        Random rand = new Random(1);
        do {
            offen = pl.Ergaenze(5 + rand.Next(100));
            Thread.Sleep(1000 + rand.Next(3000));
        } while (offen);
    }
}

public class Konsument {
    Lager pl;
    bool offen;

    public Konsument(Lager ptr) {
        pl = ptr;
    }

    public void Run() {
        Random rand = new Random(2);
        do {
            offen = pl.Liefere((5 + rand.Next(100)));
            Thread.Sleep(1000 + rand.Next(3000));
        } while (offen);
    }
}

```

Neben dem Konstruktor ist jeweils nur eine Methode namens `Run()` vorhanden, die sich auf eine **do-while**-Schleife beschränkt. In jedem Durchgang wird ein Auftrag zum Ein- bzw. Auslagern einer zufallsbestimmten Menge an das `Lager`-Objekt geschickt. 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). Solange die festen Startwerte für die Pseudozufallszahlengeneratoren beibehalten werden, resultiert stets derselbe Lagerverlauf.

Wie bereits bekannt, entsteht aus den `Run()`-Methoden jeweils ein Delegatenobjekt vom Typ **ThreadStart** und daraus wiederum ein Objekt der Klasse **Thread**:

```

Thread pt = new Thread(new ThreadStart(pro.Run));
Thread kt = new Thread(new ThreadStart(kon.Run));

```

Der über das **Thread**-Objekt `pt` ansprechbare Thread startet also mit der Ausführung der `Run()`-Methode durch das Objekt `pro` aus der Klasse **Produzent**.

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 am Ende eines **do-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.

Im Ausführungsfaden `pt` 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 ab, so dass in einem Ausführungsfaden beliebig viele Akteure tätig werden können.

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 tätig ist, dann stelle man sich ein *Team* von Lagerarbeitern vor, was der Realität vieler Betriebe recht gut entspricht.

In einem typischen Ablaufprotokoll des Programms zeigen sich einige Ungereimtheiten, verursacht durch das unkoordinierte Agieren der beiden Threads:

Der Laden 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.

13.2 Threads synchronisieren

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.:

```
public bool Ergaenze(int add) {
    lock(this) {
        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;
        }
    }
}
```

Hier wird als Sperre das handelnde Lager-Objekt verwendet, jedoch kommt auch jedes andere Objekt in Frage, das beiden Threads bekannt ist, z.B.:

```
object sperre = new object();
...
lock(sperre) {
    ...
}
```

Richter (2006, S. 642ff) rät zur Verwendung eines *privaten* Member-Objekts, weil ein öffentlich bekanntes Sperrobjekt zum Blockieren der Anwendung missbraucht werden könnte. Ein Schädling muss lediglich den geschützten Bereich betreten und sich dort auf ewig niederlassen. In unserem

Beispiel sind die Prozessbeteiligten aber so leicht zu überschauen, dass mit dem Einschleusen von böswilliger Software nicht zu rechnen ist.

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.

Weil ein Sperrobjekt ähnlich wirkt wie ein Eisenbahnsignal, das an einem Mast befestigt die Durchfahrt erlaubt oder verbietet, wird es oft als *Semaphor* bezeichnet (griech.: *Signalträger*). Meist wird auch der Ausdruck *Mutex* (engl. *mutual exclusion*) in derselben Bedeutung verwendet.

Um andere Threads möglichst wenig zu behindern, muss ein Sperrobjekt so schnell wie möglich wieder frei gegeben werden.

In unserem Beispielprogramm unterbleiben nun die wirren Protokolleinträge, doch es kommt nach wie vor zu einem negativen Lagerzustand:

Der Laden ist offen (Bestand: 100)

```
Nr. 1: Konsument entnimmt 82 um 02:02:25 Uhr. Stand: 18
Nr. 2: Produzent ergänzt 29 um 02:02:25 Uhr. Stand: 47
Nr. 3: Produzent ergänzt 51 um 02:02:27 Uhr. Stand: 98
Nr. 4: Konsument entnimmt 21 um 02:02:27 Uhr. Stand: 77
Nr. 5: Produzent ergänzt 70 um 02:02:30 Uhr. Stand: 147
Nr. 6: Konsument entnimmt 15 um 02:02:31 Uhr. Stand: 132
Nr. 7: Produzent ergänzt 40 um 02:02:32 Uhr. Stand: 172
Nr. 8: Konsument entnimmt 85 um 02:02:33 Uhr. Stand: 87
Nr. 9: Konsument entnimmt 27 um 02:02:36 Uhr. Stand: 60
Nr. 10: Produzent ergänzt 15 um 02:02:36 Uhr. Stand: 75
Nr. 11: Konsument entnimmt 81 um 02:02:37 Uhr. Stand: -6
Nr. 12: Konsument entnimmt 5 um 02:02:38 Uhr. Stand: -11
Nr. 13: Produzent ergänzt 7 um 02:02:39 Uhr. Stand: -4
Nr. 14: Konsument entnimmt 43 um 02:02:40 Uhr. Stand: -47
Nr. 15: Produzent ergänzt 37 um 02:02:41 Uhr. Stand: -10
Nr. 16: Konsument entnimmt 75 um 02:02:42 Uhr. Stand: -85
Nr. 17: Konsument entnimmt 78 um 02:02:44 Uhr. Stand: -163
Nr. 18: Produzent ergänzt 73 um 02:02:45 Uhr. Stand: -90
Nr. 19: Konsument entnimmt 13 um 02:02:45 Uhr. Stand: -103
Nr. 20: Konsument entnimmt 37 um 02:02:47 Uhr. Stand: -140
```

Lieber Produzent, es ist Feierabend!

Lieber Konsument, es ist Feierabend!

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 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** über den Erfolg der Bewerbung.

13.2.3 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 (this) {
        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(this);
            }
            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 **Started** zu versetzen:

```

public bool Ergaenze(int add) {
    lock (this) {
        if (anz < MANZ) {
            bilanz += add;
            anz++;
            Rumoren();
            Console.WriteLine("Nr. {0,2}: {1,10} ergnzt {2,3} um {3} Uhr. Stand:
{4}",
anz, Thread.CurrentThread.Name, add, DateTime.Now.ToString("T", ci),
bilanz);
            Monitor.Pulse(this);
            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 der **Wait()** - Aufruf in einer **while**-Schleife mit einleitender Bedingungsprfung statt.

Nun produziert das Beispielprogramm nur noch realistische Lagerprotokolle, z.B.:

Der Laden ist offen (Bestand: 100)

```

Nr. 1: Konsument entnimmt 82 um 03:08:29 Uhr. Stand: 18
Nr. 2: Produzent ergnzt 29 um 03:08:29 Uhr. Stand: 47
Nr. 3: Produzent ergnzt 51 um 03:08:30 Uhr. Stand: 98
Nr. 4: Konsument entnimmt 21 um 03:08:31 Uhr. Stand: 77
Nr. 5: Produzent ergnzt 70 um 03:08:33 Uhr. Stand: 147
Nr. 6: Konsument entnimmt 15 um 03:08:35 Uhr. Stand: 132
Nr. 7: Produzent ergnzt 40 um 03:08:36 Uhr. Stand: 172
Nr. 8: Konsument entnimmt 85 um 03:08:37 Uhr. Stand: 87
Nr. 9: Konsument entnimmt 27 um 03:08:39 Uhr. Stand: 60
Nr. 10: Produzent ergnzt 15 um 03:08:40 Uhr. Stand: 75
!!!!!!! Konsument muss warten: Keine 81 Einheiten vorhanden um 03:08:40 Uhr.
Nr. 11: Produzent ergnzt 7 um 03:08:43 Uhr. Stand: 82
Nr. 12: Konsument entnimmt 81 um 03:08:43 Uhr. Stand: 1
!!!!!!! Konsument muss warten: Keine 5 Einheiten vorhanden um 03:08:44 Uhr.
Nr. 13: Produzent ergnzt 37 um 03:08:44 Uhr. Stand: 38
Nr. 14: Konsument entnimmt 5 um 03:08:44 Uhr. Stand: 33
!!!!!!! Konsument muss warten: Keine 43 Einheiten vorhanden um 03:08:47 Uhr.
Nr. 15: Produzent ergnzt 73 um 03:08:48 Uhr. Stand: 106
Nr. 16: Konsument entnimmt 43 um 03:08:48 Uhr. Stand: 63
!!!!!!! Konsument muss warten: Keine 75 Einheiten vorhanden um 03:08:50 Uhr.
Nr. 17: Produzent ergnzt 33 um 03:08:51 Uhr. Stand: 96
Nr. 18: Konsument entnimmt 75 um 03:08:51 Uhr. Stand: 21
!!!!!!! Konsument muss warten: Keine 78 Einheiten vorhanden um 03:08:53 Uhr.
Nr. 19: Produzent ergnzt 75 um 03:08:54 Uhr. Stand: 96
Nr. 20: Konsument entnimmt 78 um 03:08:54 Uhr. Stand: 18

```

Lieber Konsument, es ist Feierabend!

Lieber Produzent, es ist Feierabend!

An Stelle der Methode **Pulse()**, die den Thread mit dem ltesten **Wait()**-Aufruf anspricht, ist oft die Methode **PulseAll()** sinnvoller, die alle wartenden Threads weckt.

Die **Monitor**-Methoden **Wait()**, **Pulse()** und **PulseAll()** drfen nur in einem synchronisierten Block aufgerufen werden.

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 kundenfeindliche Variante des Produzenten-Lager-Konsumenten-Beispiels dienen: der Lagerverwalter terminiert den Konsumenten-Thread, sobald dieser mit einem Wunsch über den Lagerbestand hinausgeht:

```
public bool Liefere(int sub) {
    lock (this) {
        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 Konsumententente-Thread in der Zustand **Stopped**, und im Lager taucht nur noch der Produzent auf:

Der Laden 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!

Reagiert ein Thread *nicht* auf die **ThreadAbortException**, wird er ebenfalls gestoppt.

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 lässt. Erlaubt sind die folgenden Werte des Enumerationstyps **ThreadPriority**:

- **Highest**
- **AboveNormal**
- **Normal**
- **BelowNormal**
- **Lowest**

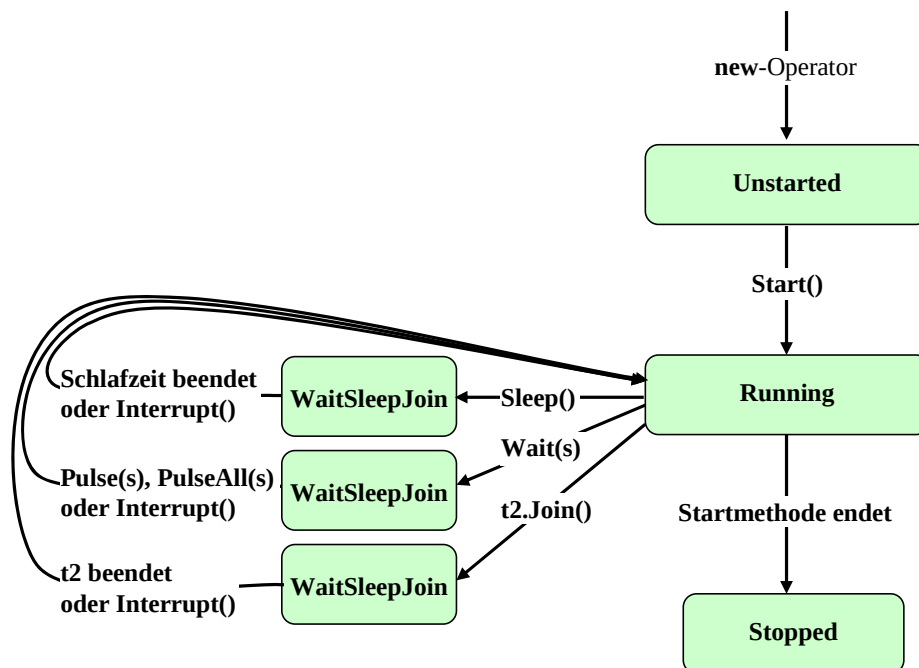
Per Voreinstellung haben Threads die Priorität **Normal**.

Der Scheduler bevorzugt Threads mit höherer Priorität in einem *strengen* Sinn und verwendet bei gleicher Priorität ein **preemptives Zeitscheibenverfahren**. Auf einem *Einprozessor*-System resultiert folgendes Verhalten:

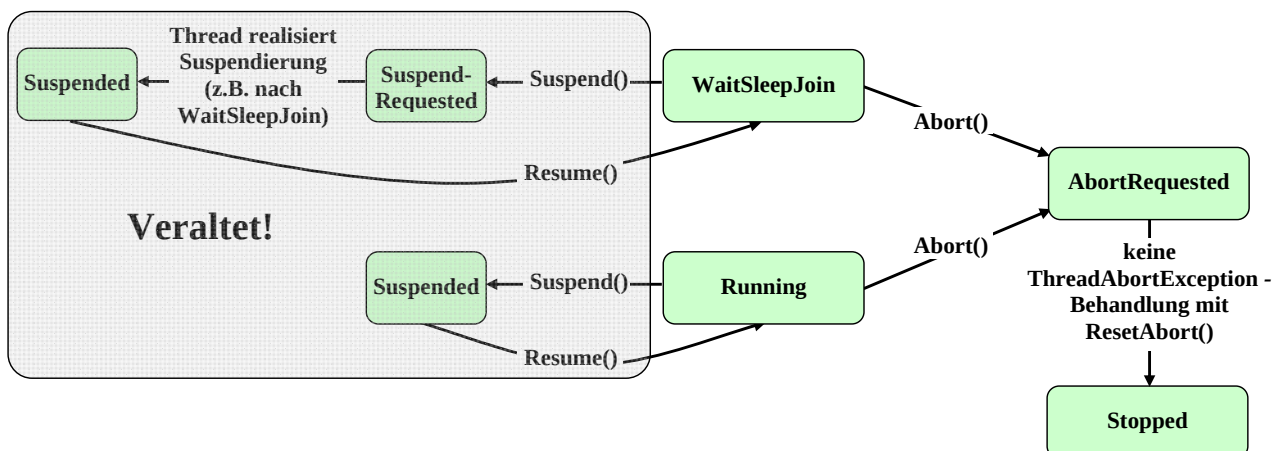
- Ein Thread kann nur dann Zugang zum Prozessor erhalten, wenn *kein* Thread mit höherer Priorität arbeitswillig ist. Ein *Verhungern* (engl. *starvation*) von Threads mit niedriger Priorität, die permanent den Kürzeren ziehen, wird also *nicht* verhindert.
- Die Threads gleicher Priorität werden reihum (*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:

- **Join()**
Der aufrufende Thread wartet bis der angesprochene Thread beendet ist. Führt z.B. der Thread `t1` die folgende Anweisung
`t2.Join();`
aus, dann wartet `t1` anschließend auf das Ende von `t2`.
- **Interrupt()**
Diese Instanzmethode dient dazu, einen Thread vom Zustand **WaitSleepJoin** in den Zustand **Running** zu versetzen. Dazu wird dem angesprochenen Thread eine **ThreadInterruptedException** 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.
- **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.

13.5 Deadlock

Wer sich beim Einsatz von exklusiven Blöcken 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;

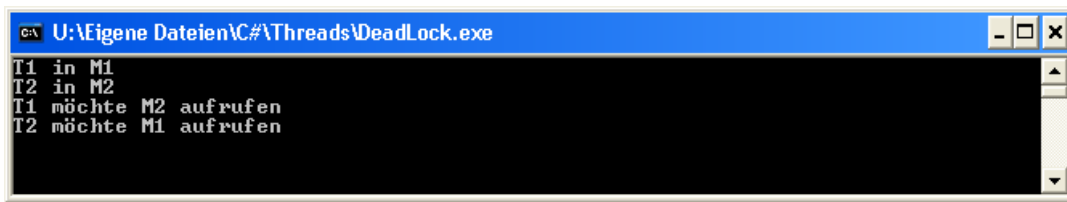
class DeadLock {
    static object lock1 = new object();
    static object lock2 = new object();

    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“ Block ungestört betreten können. Anschließend versuchen beide, in den jeweiligen fremden Block zu gelangen, und das Programm hängt fest:



```
U:\Eigene Dateien\C#\Threads\DeadLock.exe
T1 in M1
T2 in M2
T1 möchte M2 aufrufen
T2 möchte M1 aufrufen
```

Das vorgestellte Problem hat folgende Struktur:

- Thread 1 besetzt den exklusiven Block A.
- Thread 2 besetzt den exklusiven Block B.
- Thread 1 möchte Block B betreten, wartet also darauf, dass Thread 2 diesen Block verlässt.
- Dies wird Thread 2 aber nicht tun, bevor er in Block A gewesen ist. Thread 2 wartet also darauf, dass Thread 1 diesen Block verlässt.

13.6 Treadpool und APM

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 erforderlich, ...

- 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).

13.6.1 Die ThreadPool-Methode QueueUserWorkItem()

Soll eine Methode im Hintergrund durch einen Arbeits-Thread ausgeführt werden, muss sie den Delegationstyp **WaitCallback**:

public delegate void WaitCallback(Object state)

erfüllen, z.B.

```
static void HintergrundAktion(object anz) {
    long summe;
    for (int i = 0; i < (int)anz; i++) {
        summe = 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.

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: 495135074
Aktuelle Zufallssumme: 990079908
Aktuelle Zufallssumme: 1485023745
Aktuelle Zufallssumme: 1979998759
Aktuelle Zufallssumme: 2474842407
```

```
Der Arbeits-Thread ist fertig
```

```
Der primäre Thread ist aufgewacht.
```

13.6.2 Die Delegatenmethoden **BeginInvoke()** und **EndInvoke()**

In diesem Abschnitt werden zentrale Typen und Methoden im APM (*Asynchronous Programming Model*) vorgestellt, das den effizienten Einsatz von Hintergrund-Threads erlaubt und dabei ohne explizite Thread-Kreation durch Anwendungsprogrammierer auskommt.

Zu jeder Delegatendefinition, z.B.:

```
public delegate long[] 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 long[] RandomSums(int anz) {
    long[] erg = new long[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 **long**-Elementen, die jeweils eine Summe von 10000000 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(7, 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()** als Parameter übergeben werden, um das Ergebnis der abgeschlossenen Operation (im Beispiel: die **long[]**-Rückgabe der Arbeitsmethode) zu erhalten,
- kann mit seiner booleschen Eigenschaft **IsCompleted** über den Abschluss der Arbeiten informieren.

BeginInvoke() besitzt alle Parameter der Delegatendefinition 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 des Hintergrund-Threads aufgerufen wird und das Operationsobjekt als Parameter erhält. In der ersten Variante des Beispielsprogramms 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 (noch im Pool-Thread) aufgerufenen Rückrufmethode den Arbeitsdelegaten bekannt zu machen. In der ersten Variante des Beispielprogramms wird auf diese Option verzichtet und dem **BeginInvoke()**-Aufruf **null** als vierter 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 noch unbeholfen, ohne jeden Nutzen aus der APM-Technik, über 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);
    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:

```
foreach (long zs in rs.EndInvoke(ar))
    Console.WriteLine(" " + zs);
```

Die **EndInvoke()**-Methode hat denselben Rückgabewert wie die Arbeitsmethode (im Beispiel: die **long[]**).

Statt in der obigen **while**-Schleife zwischen zwei **IsCompleted**-Abfragen den primären Thread schlafen zu legen, 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 lahm legt.

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:");
    foreach (long zs in z.EndInvoke(ar))
        Console.WriteLine(" "+zs);
}
```

Das zugehörige Delegatenobjekt wird im **BeginInvoke()**-Aufruf als dritter Parameter übergeben, z.B.:

```
rs.BeginInvoke(3, new AsyncCallback(Report), rs);
```

Sobald der Hintergrund-Thread die Arbeitsmethode ausgeführt hat, ruft er die Rückrufmethode auf und übergibt das Operationsobjekt. Im Beispiel soll die Rückrufmethode **Report()** das Operationsobjekt in einem **EndInvoke()**-Aufruf an den Arbeitsdelegaten verwenden, um die Ergebnisse anzufordern. Die nötige Referenz wird folgendermaßen in die Rückrufmethode transportiert:

- Im **BeginInvoke()**-Aufruf an den Arbeitsdelegaten wird seine eigene Adresse als vierter 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 long[] Zumme(int anz);

class Prog {
    static Random zzg = new Random();

    static long[] 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 Poolthread 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");
        Console.ReadLine();
    }
}
```

demonstriert die APM-Rückruftechnik, verzichtet aber der Übersichtlichkeit halber auf eine sinnvolle Aktivität im Vordergrund-Thread.

Der Arbeitsdelegat soll per Poolthread 3 Summen von Zufallszahlen ermitteln.

Der primäre Thread schläft 5 Sekunden.

Report der ermittelten Summen:

494898976

494910016

494969138

Der primäre Thread ist aufgewacht.

Auf einen **BeginInvoke()**-Aufruf sollte unbedingt ein **EndInvoke()**-Aufruf an denselben Arbeitsdelegaten folgen, weil ansonsten von der asynchronen Operation belegte CLR-Ressourcen nicht mehr frei gegeben werden (Richter 2006, S. 621).

13.7 Timer

Mit Hilfe der Klasse **Timer** im Namensraum **System.Threading** kann man die CLR beauftragen, eine Methode regelmäßig in einem Pool-Thread auszuführen. Im diesem Beispiel

```
tim = new System.Threading.Timer(this.HintergrundAktion, null, 0, 1000);
```

kommt eine Konstruktor-Überladung mit folgenden Parameter-Datentypen zum Einsatz:

- **TimerCallback** *callback*

Dieser Delegationstyp verlangt für die regelmäßig auszuführende Methode folgende Signatur:

void TimerCallback (Object state)

Im obigen Beispiel kommt die vereinfachte Syntax mit impliziter Konstruktion des Delegationobjekts 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 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** einmal pro Sekunde die Summe aus 10 Millionen Zufallszahlen:

```
using System;
using System.Windows.Forms;
using System.Threading;

delegate void LabelUpdateDelegate(string s);

class ThreadingTimer : Form {
    Label anzeige;
    System.Threading.Timer tim;
    Random zsg;
    LabelUpdateDelegate laup;

    public ThreadingTimer() {
        Height = 130; Width = 350;
        Text = "System.Threading.Timer";
        anzeige = new Label();
        anzeige.Width = 300;
        anzeige.Left = 20; anzeige.Top = 20;
        Controls.Add(anzeige);
        zsg = new Random();
        laup = new LabelUpdateDelegate(this.UpdateLabel);
        tim = new System.Threading.Timer(this.HintergrundAktion, null, 0, 1000);

        TextBox eingabe = new TextBox();
        eingabe.Left = 20; eingabe.Top = 50; eingabe.Width = 125;
        Controls.Add(eingabe);
    }
}
```



```

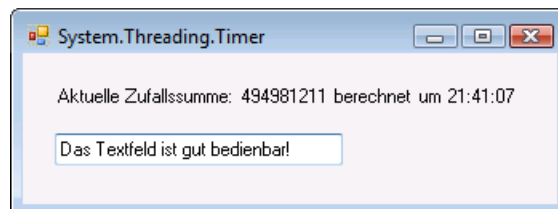
void HintergrundAktion(object info) {
    long l = 0;
    for (int i = 0; i < 100000000; i++)
        l += zsg.Next(100);
    BeginInvoke(laup, l.ToString());
}

void UpdateLabel(string s) {
    anzeige.Text = "Aktuelle Zufallssumme: "+s+" berechnet um "+
        DateTime.Now.ToLongTimeString();
}

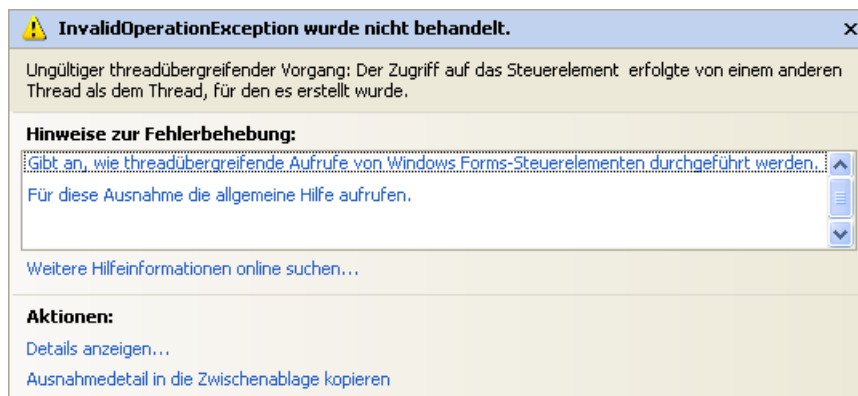
[STAThread]
static void Main() {
    Application.EnableVisualStyles();
    Application.Run(new ThreadingTimer());
}
}

```

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: `LabelUpdater()`),
- definiert einen passenden Delegates (im Beispiel: `LabelUpdaterDelegate`),
- erzeugt ein Delegatesobjekt (`new LabelUpdateDelegate(LabelUpdater)`)
- und veranlasst den GUI-Thread über die **Invoke()**-Methode oder die **BeginInvoke()**-Methode des *Formulars*, das Delegatesobjekt 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 13.6.2) der zugehörig **EndInvoke()** - Aufruf durchgeführt werden sollte, ist im vorliegenden Fall ausnahmsweise *kein EndInvoke()*-Aufruf erforderlich.

Im Namensraum **System.Windows.Forms** ist ebenfalls eine Klasse namens **Timer** vorhanden. Diese arbeitet ereignisorientiert und ist sehr bequem in eine GUI-Anwendung einzubinden, z.B.:

```
using System;
using System.Windows.Forms;

class WinFormsTimer : Form {
    Label anzeige;
    Random zzg;

    public WinFormsTimer() {
        Height = 130; Width = 350;
        Text = "System.Windows.Forms.Timer";
        anzeige = new Label();
        anzeige.Width = 300;
        anzeige.Left = 20; anzeige.Top = 20;
        Controls.Add(anzeige);
        zzg = new Random();

        Timer tim = new Timer();
        tim.Tick += new EventHandler(TimerAktion);
        tim.Interval = 1000;
        tim.Start();

        TextBox eingabe = new TextBox();
        eingabe.Left = 20; eingabe.Top = 50; eingabe.Width = 125;
        Controls.Add(eingabe);
    }

    void TimerAktion(Object myObject, EventArgs myEventArgs) {
        long l = 0;
        for (int i = 0; i < 100000000; i++)
            l += zzg.Next(100);
        anzeige.Text = "Aktuelle Zufallssumme: " + l.ToString() +
            " berechnet um " + DateTime.Now.ToLongTimeString();
    }

    [STAThread]
    static void Main() {
        Application.EnableVisualStyles();
        Application.Run(new WinFormsTimer());
    }
}
```

Diesmal laufen die vom Timer angestoßenen Methodenaufrufe jedoch im GUI-Thread ab, und bei gleicher Rechenlast resultiert eine unergonomisch zähe Benutzeroberfläche.

Derartige Probleme lassen sich gelegentlich beherrschen, indem eine zeitaufwändige Ereignisbehandlungsmethode ihre Tätigkeit unterbricht, um die Bearbeitung anderer Ereignisse zu ermöglichen. Diese Form von Kooperation innerhalb des GUI-Threads wird durch Aufrufe der statischen Methode **DoEvents()** der Klasse **Application** realisiert, z.B.:

```

void TimerAktion(Object myObject, EventArgs myEventArgs) {
    long l = 0;
    for (int j = 0; j < 100; j++) {
        Application.DoEvents();
        for (int i = 0; i < 100000; i++)
            l += zzg.Next(100);
    }
    anzeige.Text = "Aktuelle Zufallssumme: " + l.ToString() +
        " berechnet um " + DateTime.Now.ToLongTimeString();
}

```

Im Beispiel wird eine Schleife mit sehr hoher Umdrehungszahl durch eine Doppelschleife ersetzt und bei jedem Durchlauf der äußeren Schleife die **Application**-Methode **DoEvents()** aufgerufen. Durch derartige Maßnahmen wird ein Algorithmus nicht übersichtlicher, und die eventuell vorhandene Rechenleistung weiterer CPU-Kerne bleibt ungenutzt.

Für das regelmäßige Starten von Methoden mit geringem Rechenzeitbedarf ist die WinForms - Timer-Komponente aber durchaus geeignet und wegen der einfachen Verwendbarkeit attraktiv (siehe Abschnitt 16.6).

13.8 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. Die Methoden der WinForms – Steuerelemente sind in der Regel Thread – sicher.
3. Bei der **lock**-Anweisung

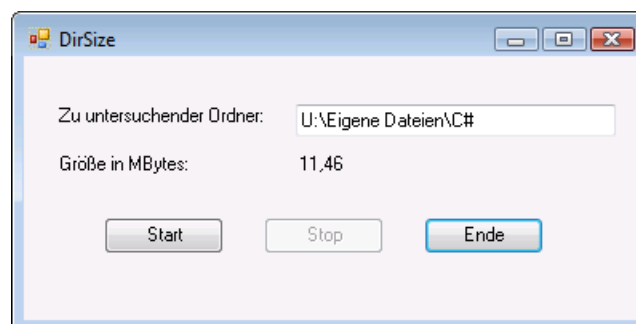
```

lock(sperre) {
    ...
}

```

ist sichergestellt, dass der kritische Block auch beim Auftreten einer Ausnahme verlassen wird.

2) 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 (z.B. HTTP) und einer Programmierung auf elementaren Protokollebenen mit einer entsprechend weiter reichenden Kontrolle wählen kann.

Die serverseitige Webprogrammierung wird in diesem Manuskript *nicht* behandelt. Dabei geht es vor allem um folgende Projekttypen:

- **Dynamische Webseiten**

Auf der Serverseite werden (oft in Kooperation mit ADO.NET) HTML-Seiten aufgrund einer speziellen Anforderung individuell erstellt und dann zum klientenseitigen Browser gesandt. Benutzer produzieren über die GUI-Elemente der HTML-Seite Ereignisse für das Serverprogramm.

- **Webdienste**

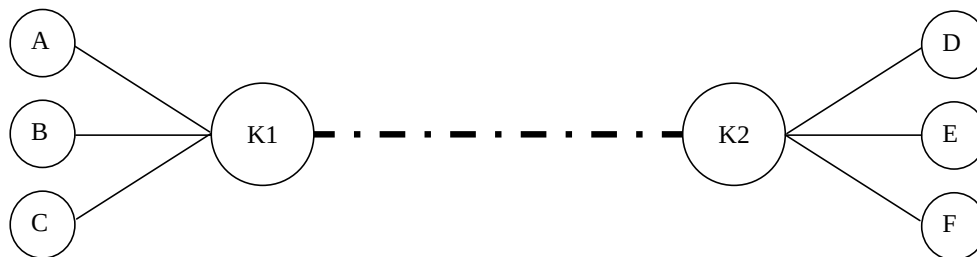
Mit dieser Technik wird der Aufruf von Prozeduren auf entfernten Rechnern (*Remote Procedure Call*) via Internet unterstützt¹. Dabei kommt das XML-basierte **SOAP**-Protokoll (*Simple Object Access Protocol*) zum Einsatz. So kann z.B. eine lokale Anwendung vom Server einer Fluglinie Daten über Verbindungen, 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.

Bei den meisten aktuellen Netzwerkprotokollen werden Daten **paketweise** übertragen. Zwischen zwei Kommunikationspartnern jeweils eine feste Leitung zu schalten und auch in „Funkpausen“ aufrecht zu erhalten, wäre unökonomisch. Wenn über dieselbe Leitung, z.B. zwischen den Verbindungsknoten K1 und K2, Pakete zwischen verschiedenen Kommunikationspartnern, z.B. (A ↔ D), (B ↔ E), ausgetauscht werden, ist eine **Adressierung** der Pakete unabdingbar.



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.

¹ Server und Klienten für Webdienste müssen allerdings nicht als .NET – Anwendungen realisiert werden.

14.1.1 Das OSI-Modell

Nach dem **OSI – Modell** (*Open System Interconnection*) der ISO (*International Standards Organization*) werden 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.

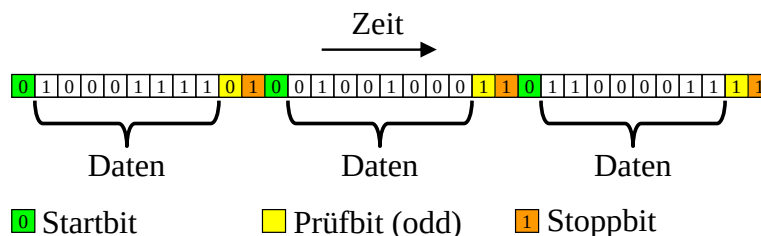
2. Link-Ebene (gesicherte Frame-Übertragung, z.B. per Ethernet)

Hier wird vereinbart, wie 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 z.B. dieser 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. auf der Telefonstrecke zum Provider eine andere als auf dem weiteren Weg über Ethernet- oder ATM-Verbindungen). 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.

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 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:0000:0000:88c7:c79c

Innerhalb eines Blocks dürfen führende Nullen weggelassen werden, z.B.:

2001:88c7:c79c: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. Eine weitere Aufgabe der dritten Protokollebene besteht in der **Datenflusskontrolle** zur Vermeidung von Überlastungen.

5. Sitzungsebene (Übertragung von Byte-Strömen zwischen Endpunkten, 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 zwischen 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 Paketversand ohne Garantie für eine vollständige Auslieferung in korrekter Reihenfolge. Aufgrund der somit eingesparten Verwaltungskosten eignet sich das UDP zur Übertragung größerer Datenmengen, wenn dabei der Verlust einzelner Pakete zu verschmerzen ist (z.B. beim Multimedia - Streaming).

6. Präsentation

Hier geht es z.B. um die Verschlüsselung oder Komprimierung von Daten. Die TCP/IP - Protokollfamilie kümmert sich nicht darum, sondern überlässt derlei Arbeiten den Anwendungen.

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-Server - Programm (*Simple Mail Transfer Protocol*), der an Port 25 lauert, kann ein Klientenprogramm z.B. folgendermaßen eine Mail übergeben:

Klient	Serverantwort
telnet srv.srv-dom.de 25 HELO mainpc.client-dom.de MAIL FROM: otto@client-dom.de RCPT TO: empf@srv-dom.de DATA From: egal@weg.te To: ziel@soso.te Subject: Thema Dies ist der Inhalt! . QUIT	220 srv.srv-dom.de ESMTP Postfix 250 srv.srv-dom.de 250 Ok 250 Ok 354 End data with <CR><LF>.<CR><LF> 250 Ok: queued as 43A7D6D91AC 221 Bye

Der Mailempfänger glaubt hoffentlich nicht an die angezeigten Adressen:

Datum: Thu, 8 Feb 2007 14:41:54 +0100 (CET)

Von: egal@weg.te

An: ziel@soso.te

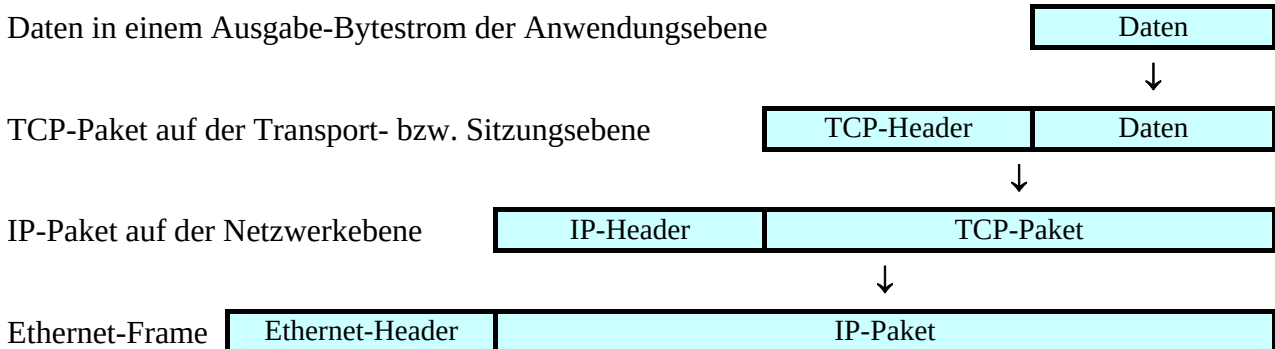
Betreff: Thema

Dies ist der Inhalt!

Noch häufiger als die Mail-Protokolle 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 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 ein-
treffendes 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 zusätzliche Header z.B. die MAC-Adresse des Routers aufnimmt.

Auf dem Zielrechner 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 Win32-API (z.B. für die Inter-Prozess-Kommunikation über *named pipes*) 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:	Proto-	://	User:Pass@	Rechner	:Port	Pfad	? URL-Parameter
	koll		(optional)		(optional)		(optional)
Beispiel:	http	://		www.egal.de	:81	/cgi-bin/beispiel/cgi.pl	?vorname=Kurt

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>

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

14.2.1 Statische Webinhalte anfordern

Zum Abrufen einer per URI beschriebenen 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** - Unter-

klasse ansprechen. Aufgrund der folgenden **UserAgent** - Manipulation stellt sich ein C# - Programm beim Webserver als Firefox - Browser vor:

```
((HttpRequest)request).UserAgent =
    "Mozilla/5.0 (Windows; U; Windows NT 5.1; de; rv:1.9.0.5) "+
    "Gecko/2008120122 Firefox/3.0.5 (.NET CLR 3.5.30729)";
```

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 content = 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(content, Encoding.GetEncoding(cs));
                    break;
    default:      reader = new StreamReader(content, 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 RequestResponse {
    static void Main() {
        // Request-Objekt zu URI erzeugen
        WebRequest request = WebRequest.Create("http://www.uni-trier.de/");
        // Protokollspezifische Request-Konfiguration
        ((HttpRequest)request).UserAgent =
            "Mozilla/5.0 (Windows; U; Windows NT 5.1; de; rv:1.9.0.5) " +
            "Gecko/2008120122 Firefox/3.0.5 (.NET CLR 3.5.30729)";
        // 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.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);
    Console.ReadLine();
}
// Datenstrom schließen
response.Close();
}
}

```

Ein Programmlauf (am 6.2.2009) bringt folgendes Ergebnis:

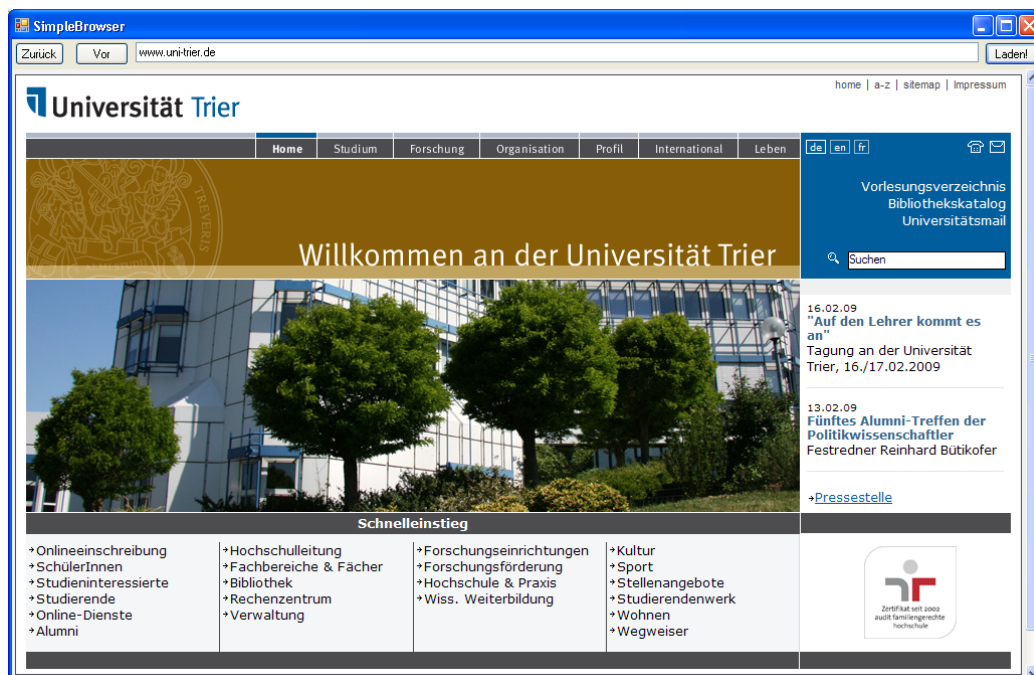
Letzte Änderung: 06.02.2009 06:04:34
 Zeichensatz: iso-8859-1

```

<?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"/>
    . . .
    . . .

```

Seit der .NET - Version 2.0 findet sich im Namensraum **System.Windows.Forms** die Steuerelementklasse **WebBrowser** mit der Fähigkeit, HTML zu rendern. Sie sollen als Übungsaufgabe einen kleinen Webbrowser erstellen, der HTML-Seiten attraktiver anzeigen kann, 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 herunterladen. Wir nutzen zu diesem Zweck die Klasse **WebClient**, welche durch Verpacken der ohnehin schon recht bequemen Klassen

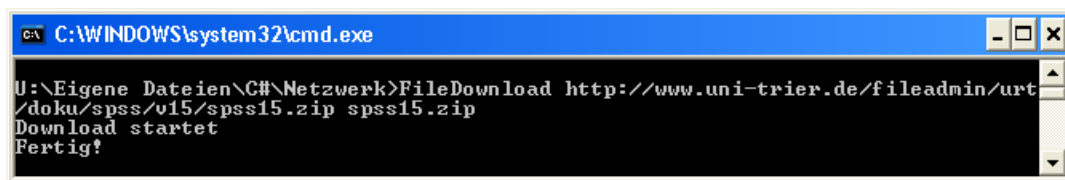
WebRequest und **WebResponse** den Komfort auf die Spitze treibt:

```
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);
        }
    }
}
```

Man verwendet die Methode **DownloadFile()** und überlässt Routinen 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 ermöglicht den Datei-Download per Kommandozeile, was auch im GUI - Zeitalter noch gefragt ist, z.B.:



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 Kommandozeilenargumente 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**:¹

¹ Weil Fehlermeldungen des Webservers relativ wahrscheinlich sind (z.B. durch Tippfehler im URI), ist nach den Überlegungen aus Abschnitt 10.3 die Fehlerkommunikation per Ausnahmeobjekt nicht unbedingt ideal.

```

using System;
using System.IO;
using System.Net;

class FileDownload {
    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> <Datum>");
            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();
        }
    }
}

```

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 bieten 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, 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 geschieht dies über das so genannte **Common Gateway Interface (CGI)**, wobei das externe 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. ASP.NET als IIS - Erweiterung, Java-Container wie Tomcat, 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 Informationsquelle 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 (im Rahmen des ASP.NET - Frameworks) *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.

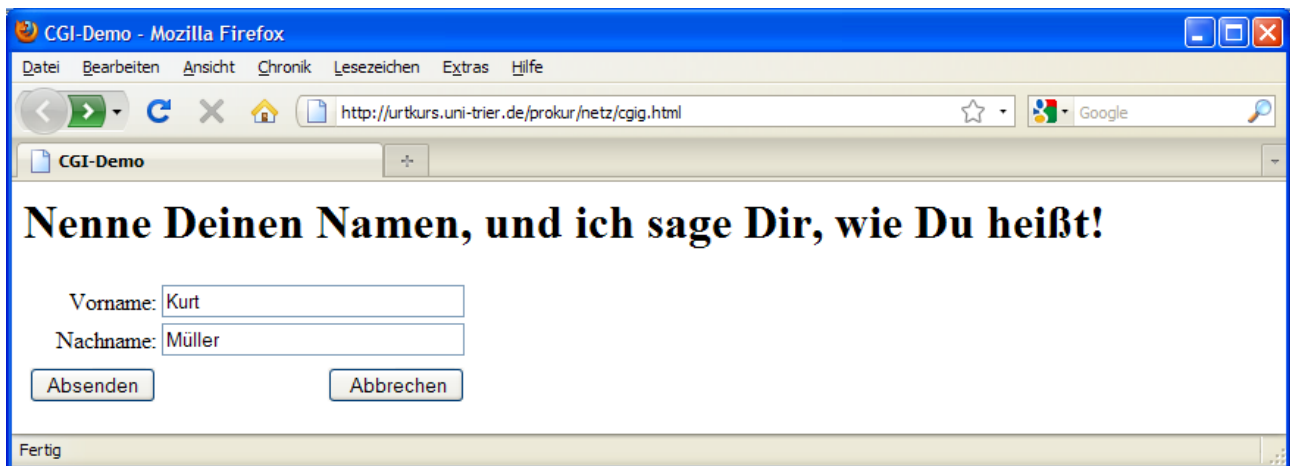
14.2.3.2 Arbeitsablauf

Beim CGI - Einsatz sind üblicherweise folgende Programme beteiligt:

- WWW-Browser
- WWW-Server
- CGI-Lösung

Der Browser zeigt eine vom Server erhaltene HTML-Seite mit Formular an, über die Benutzer eine CGI-Anfrage konfigurieren und abschicken können. Wir betrachten ein einfaches Formular mit zugehörigem CGI-Skript, um einige technische Details zu erläutern.

In diesem Browser-Fenster



wird eine HTML-Datei angezeigt, die über den URL

<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 heißt!</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%FC11ler`

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, verwendet der Browser eine Voreinstellung (im Beispiel: 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.
- Die mit einer speziellen Bedeutung belasteten Zeichen (also &, +, = und %) werden durch ihren Hexadezimalwert im Zeichensatz dargestellt.

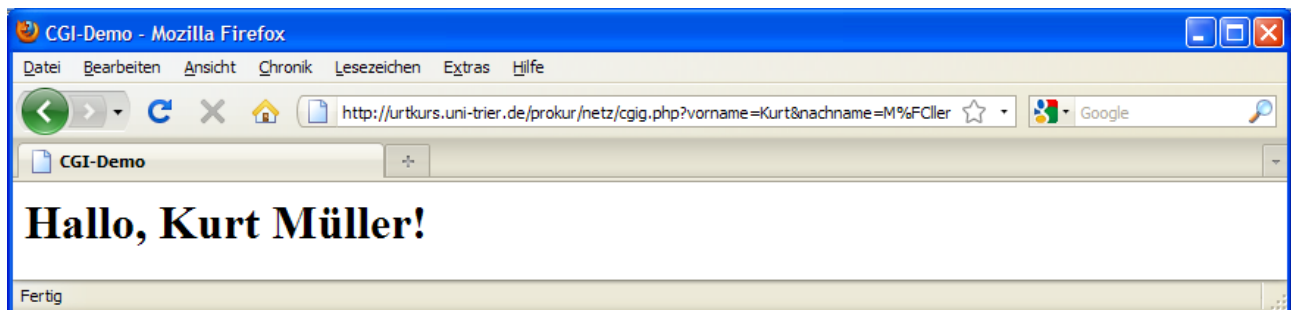
Auf gleich noch näher zu erläuternde Weise übergibt der WWW-Server die Formulardaten an das im **action**-Attribut der Formulardefinition angegebene externe Programm oder Skript. Im Beispiel handelt es sich um folgendes **PHP**-Skript, das wenig kreativ aus den übergebenen Namen einen Gruß formuliert:

```
<?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-Quelltext



anzeigt:



14.2.3.3 GET

Zum Versandt von CGI-Parametern an einen WWW-Server kennt das HTTP-Protokoll zwei Methoden (**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. die POST-Technik verwenden, die 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 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("\nAnforderung wird übertragen ...");

        WebClient client = new WebClient();
        byte[] antwort = null;
        try {
            antwort = client.DownloadData(cgiuri + "?vorname=" + vorName +
                "&nachname=" + nachName);
            String s = HttpUtility.UrlDecode(antwort, Encoding.Default);
            Console.WriteLine("\nAntwort :\n" + s);
        } catch (Exception e) {
            Console.WriteLine(e.Message);
        }
        Console.ReadLine();
    }
}
```

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 ist auch beteiligt, wenn der vom Webserver gelieferte Byte-Array in der statischen **HttpUtility**-Methode **UrlDecode()** in einen **String** gewandelt wird, wobei für die Lieferung des Webserver eine zum lokalen System kompatible Kodierung angenommen wird. Bei einer unbekannten CGI-Lösung sollte man die Zeichensatz - Metainformationen der Serverantwort auswerten (z.B. über die **HttpWebResponse**-Eigenschaft **CharacterSet**, siehe Abschnitt 14.2.1).

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

Anforderung wird übertragen ...

Antwort:
<html>
<head><title>CGI-Demo</title></head>
<h1>Hallo, Kurt Müller!</h1>
</html>
```

14.2.3.4 POST

Bei der **POST**-Technik, die man im **<form>** - Tag einer HTML-Seite durch die Angabe `method="post"`

wählt, werden die Parameter (im selben Format wie bei der GET-Methode) 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);
```

Die von **UploadValues()** als Rückgabe gelieferte Server-Antwort kann mit der **HttpUtility**-Methode **UrlDecode()** (siehe Abschnitt 14.2.3.3) unter Verwendung der lokal gültigen Kodierung in einen **String** verwandelt werden:

```
String s = HttpUtility.UrlDecode(antwort, Encoding.Default);
```

Insgesamt unterscheidet sich das GET-Beispielprogramm nur wenig von der POST-Variante:

```
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=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);

        NameValueCollection formData = new NameValueCollection();
```

```

formData.Add("vorname", vorName);
formData.Add("nachname", nachName);

Console.WriteLine("\nAnforderung wird übertragen ...");
WebClient client = new WebClient();
byte[] antwort = null;
try {
    antwort = client.UploadValues(cgiuri, formData);
    String s = HttpUtility.UrlDecode(antwort, Encoding.Default);
    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 angeschlossenen 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

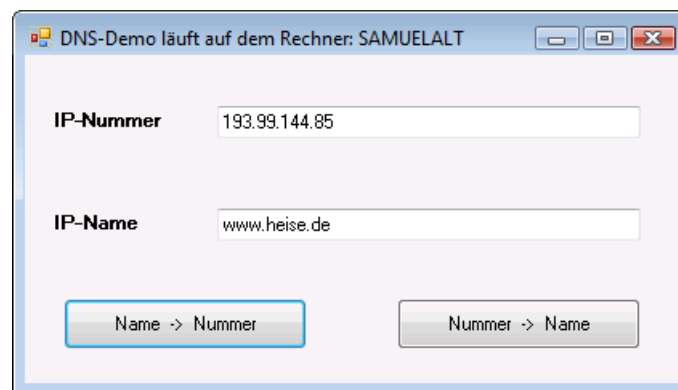
```

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 Beispielsprogrammen in Abschnitt 14. Daher werden nur die Ereignisbehandlungsmethoden zu den beiden Schaltflächen wiedergegeben:

```

void CbToNumberClick(object sender, System.EventArgs e) {
    try {
        IPEndPoint host = Dns.GetHostEntry(tbName.Text);
        tbNumber.Text = host.AddressList[0].ToString();
    } catch (Exception ex) {
        MessageBox.Show(ex.ToString(), "Fehler");
    }
}

void CbToNameClick(object sender, System.EventArgs e) {
    try {
        IPEndPoint 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 (siehe 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 fest an einen Port gebunden ist und auf dort 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[0];

server = new TcpListener(ip, svrPort);
```

Vom veralteten Konstruktoraufruf *ohne* IP-Adresse und der damit erforderlichen automatischen Wahl einer IP-Adresse hält der Compiler nichts mehr:

```
TcpListener.cs(13,13): 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.
```

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. Das Senden kann durch einen **BinaryWriter** (für elementare Datentypen) oder einen **StreamWriter** (für Textdaten) vereinfacht werden, wobei ein Protokoll auf der Anwendungsebene zu verabreden oder zu beachten ist.

Am Ende der Klientenbedienung wird per **Shutdown()**-Befehl an das angeschlossene **Socket** - Objekt die sofortige Ausführung von eventuell noch anstehenden Übertragungen angefordert:

```
tcpClient.Client.Shutdown(SocketShutdown.Send);
```

Auf den **Close()** - Aufruf

```
tcpClient.Close();
```

reagiert das **TcpClient** – Objekt so:

- Das interne **NetworkStream** - Objekt erhält einen **Close()** – Aufruf, und die Referenz auf dieses Objekt wird auf **null** gesetzt.
- Das interne **Socket** - Objekt erhält einen **Close()** – Aufruf, und die Referenz auf dieses Objekt wird auf **null** gesetzt.

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[0];
        TcpClient tcpClient = null;
        NetworkStream stream = null;

        try {
            server = new TcpListener(ip, svrPort);
            server.Start();
            Console.WriteLine("Zeitserver lauscht seit " + DateTime.Now + " (IP: " +
                ip + ", Port: " + svrPort + ")");
            while (true) {
                // Achtung: Aufruf blockiert!
                tcpClient = server.AcceptTcpClient();
                Console.WriteLine("\n" + DateTime.Now + " Anfrage von\n IP-Nummer:\t"
                    + (tcpClient.Client.RemoteEndPoint as IPEndPoint).Address + "\n Port: \t"
                    + (tcpClient.Client.RemoteEndPoint as IPEndPoint).Port);
                stream = tcpClient.GetStream();
                byte[] msg = Encoding.ASCII.GetBytes(DateTime.Now.ToString());
                stream.Write(msg, 0, msg.Length);
                tcpClient.Client.Shutdown(SocketShutdown.Send);
                tcpClient.Close();
            }
        } catch (Exception e) {
            Console.WriteLine(e);
            Console.ReadLine();
        }
    }
}

```

Der Kürze halber verzichten wir auf ein Verfahren zum regulären Beenden des Programms, so dass es rabiab abgebrochen werden muss.

Sobald *nicht* die interne IP-Adresse 127.0.0.1 (verbunden mit dem Host-Namen *localhost*) verwendet wird, ist unter Windows XP ab SP 2 für den Betrieb des Zeitservers eine Firewall-Ausnahme erforderlich:



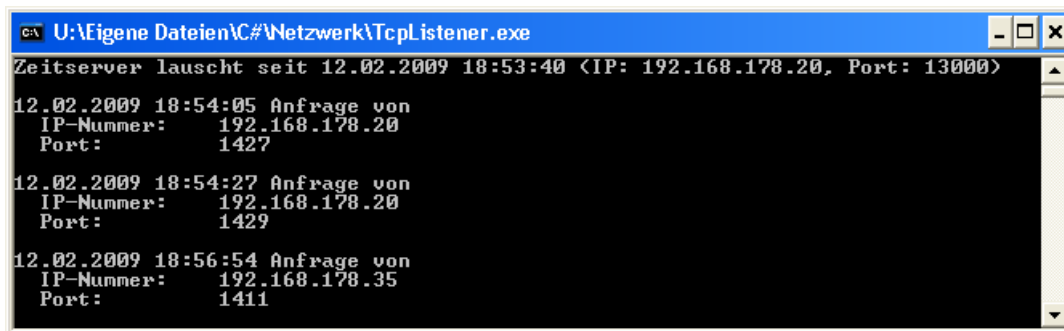
Der Zeitserver kann z.B. per Telnet-Klient angesprochen werden, z.B. mit dem folgenden Aufruf:

```
telnet localhost 13000
```

Der Server antwortet:



und protokolliert seine Dienstleistungen:



So kommt das Programm an die IP-Adresse und den Port des verbundenen Klienten heran:

```
(tcpClient.Client.RemoteEndPoint as IPEndPoint).Address
(tcpClient.Client.RemoteEndPoint as IPEndPoint).Port
```

Die **RemoteEndPoint**-Eigenschaft des **Socket**-Objekts, das über die **TcpClient**-Eigenschaft **Client** angesprochen wird, zeigt auf ein Objekt vom deklarierten Datentyp **EndPoint**, das tatsächlich zur Klasse **IPEndPoint** gehört und daher die Eigenschaften **Address** und **Port** mit den gewünschten Informationen besitzt.

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 Konstruktion - 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 **IOException** 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);
```

Auf den **Close()** - Aufruf

```
tcpClient.Close();
```


reagiert das **TcpClient** – Objekt so:

- Das interne **NetworkStream** - Objekt erhält einen **Close()** – Aufruf, und die Referenz auf dieses Objekt wird auf **null** gesetzt.
- Das interne **Socket** - Objekt erhält einen **Close()** – Aufruf, und die Referenz auf dieses Objekt wird auf **null** gesetzt. Dies beendet die Verbindung zum Server und gibt die Netzwerk-Ressourcen (z.B. den belegten Port) an das Betriebssystem zurück.

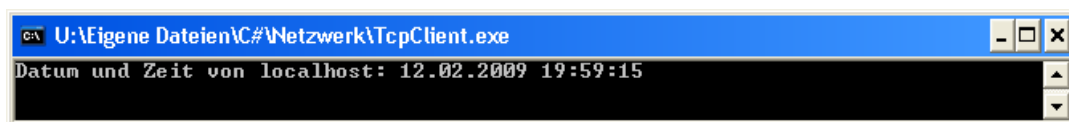
Die explizite Rückgabe von Ressourcen ist vor allem dann relevant, wenn ein Programm anschließend weiter aktiv bleiben soll. Bei Beendigung eines Programms werden die belegten Ressourcen automatisch frei gegeben.

Der Quellcode im Überblick:

```
using System;
using System.Net.Sockets;
using System.Text;

class TcpClientDemo {
    static void Main() {
        String svr = "localhost";
        int port = 13000;
        try {
            TcpClient 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);
            tcpClient.Close();
        } catch (Exception e) {
            Console.WriteLine(e);
        }
    }
}
```

Sofern Netz und Server mitspielen, liefert das Programm Datum und Uhrzeit, z.B.:



14.4.3 Simultane Bedienung mehrerer Klienten

Bei einer ernsthaften Serverprogrammierung 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.Client.Shutdown(SocketShutdown.Both);
    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 Aufräumarbeiten 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) {
        Console.WriteLine(e);
        Console.ReadLine();
    }
}

```

Sobald der Lauscher vom Typ **TcpListener** eine Klientenanfrage feststellt, wird ein **Echo-Handler** - Objekt erzeugt, das im Konstruktoraufbau 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 die `TcpListener` - Methode `AcceptTcpClient()` erzeugt hat
- eine Klientennummer 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.

```

U:\Eigene Dateien\C#\Netzwerk\MultiClientEchoServer.exe
12.02.2009 21:03:26 MultiClientEchoServer gestartet

12.02.2009 21:03:34 Klient 1 akzeptiert. Aktiv: 1
IP-Nummer: 127.0.0.1
Port: 1484

12.02.2009 21:03:35 Klient 2 akzeptiert. Aktiv: 2
IP-Nummer: 127.0.0.1
Port: 1485

12.02.2009 21:03:35 Klient 3 akzeptiert. Aktiv: 3
IP-Nummer: 127.0.0.1
Port: 1486

U:\Eigene Dateien\C#\Netzwerk\DauerSender.exe
Sendung 33: 4436215458 Echo: 4436215458
Sendung 34: 4429803588 Echo: 4429803588
Sendung 35:
Sendung 36:

U:\Eigene Dateien\C#\Netzwerk\DauerSender.exe
Sendung 37: Sendung 32: 8939302557 Echo: 8939302557
Sendung 38: Sendung 33: 2710539856 Echo: 2710539856
Sendung 39: Sendung
Sendung 40: Sendung
Sendung 41: Sendung
Sendung 42: Sendung
Sendung 43: Sendung
Sendung 44: Sendung
Sendung 45: Sendung
Sendung 46: Sendung
Sendung 47: Sendung
Sendung 48: Sendung
Sendung 49: Sendung
Sendung 50: Sendung
Sendung 51: Sendung
Sendung 52: Sendung
Sendung 53: Sendung
Sendung 54: Sendung
Sendung 55: Sendung
Sendung 56: Sendung
Sendung 57: Sendung

U:\Eigene Dateien\C#\Netzwerk\DauerSender.exe
Sendung 31: 2357272836 Echo: 2357272836
Sendung 32: 4020908284 Echo: 4020908284
Sendung 33: 1728329273 Echo: 1728329273
Sendung 34: 3055570290 Echo: 3055570290
Sendung 35: 8490958268 Echo: 8490958268
Sendung 36: 7859151640 Echo: 7859151640
Sendung 37: 0010245752 Echo: 0010245752
Sendung 38: 0627577197 Echo: 0627577197
Sendung 39: 4759554539 Echo: 4759554539
Sendung 40: 6379283984 Echo: 6379283984
Sendung 41: 0064806843 Echo: 0064806843
Sendung 42: 8977615247 Echo: 8977615247
Sendung 43: 9935668357 Echo: 9935668357
Sendung 44: 4824730005 Echo: 4824730005
Sendung 45: 9619066021 Echo: 9619066021
Sendung 46: 2040507081 Echo: 2040507081
Sendung 47: 1226906203 Echo: 1226906203
Sendung 48: 0419827349 Echo: 0419827349
Sendung 49: 0804433286 Echo: 0804433286
Sendung 50: 5928364400 Echo: 5928364400
Sendung 51: 9309089522 Echo: 9309089522
Sendung 52: 3358132826 Echo: 3358132826
Sendung 53: 7433702352 Echo: 7433702352
Sendung 54: 8234068036 Echo: 8234068036
Sendung 55: 4795313360

```

Den vollständigen Quellcode finden Sie im Ordner

... \BspUeb\Netzwerk\MultiClientEchoServer\MultiThreadEchoServer

14.5 Übungsaufgaben zu Kapitel 14

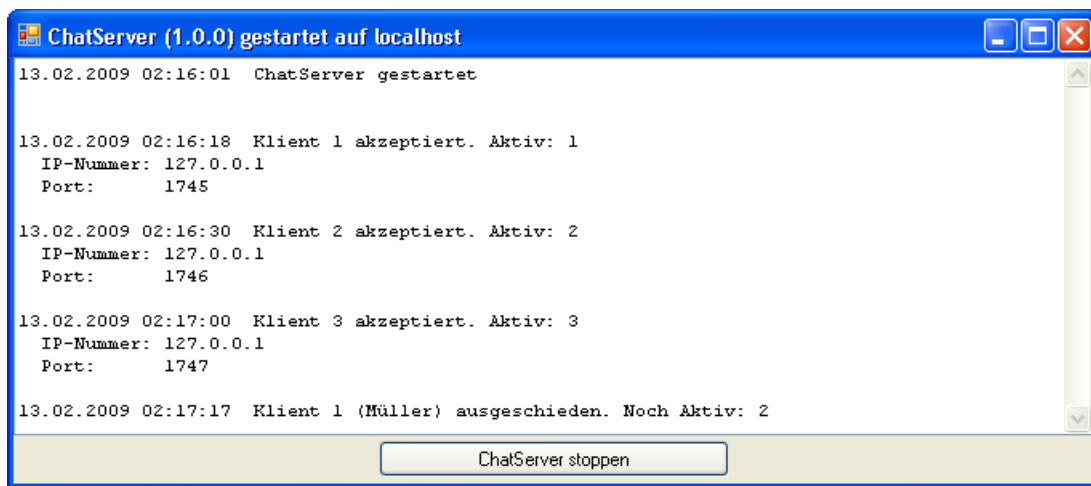
1) Erstellen Sie mit Hilfe der Steuerelementklasse **WebBrowser** eine WinForms-Anwendung, die eine **TextBox** zur Eingabe einer Webadresse bietet, HTML anzeigt, sinnvoll auf Änderungen der Formulargröße reagiert und elementare Navigationsmöglichkeiten bietet (siehe Bildschirmphoto in Abschnitt 14.2.1).

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 (Wert **true** für die Eigenschaft **Multiline**) mit vertikalem Rollbalken (Wert **Vertical** für die Eigenschaft **ScrollBars**)
- Ein **Button**-Steuerelement anbieten zum Beenden des Programms, wobei die aktiven Klienten über das Dienstende zu benachrichtigen sind

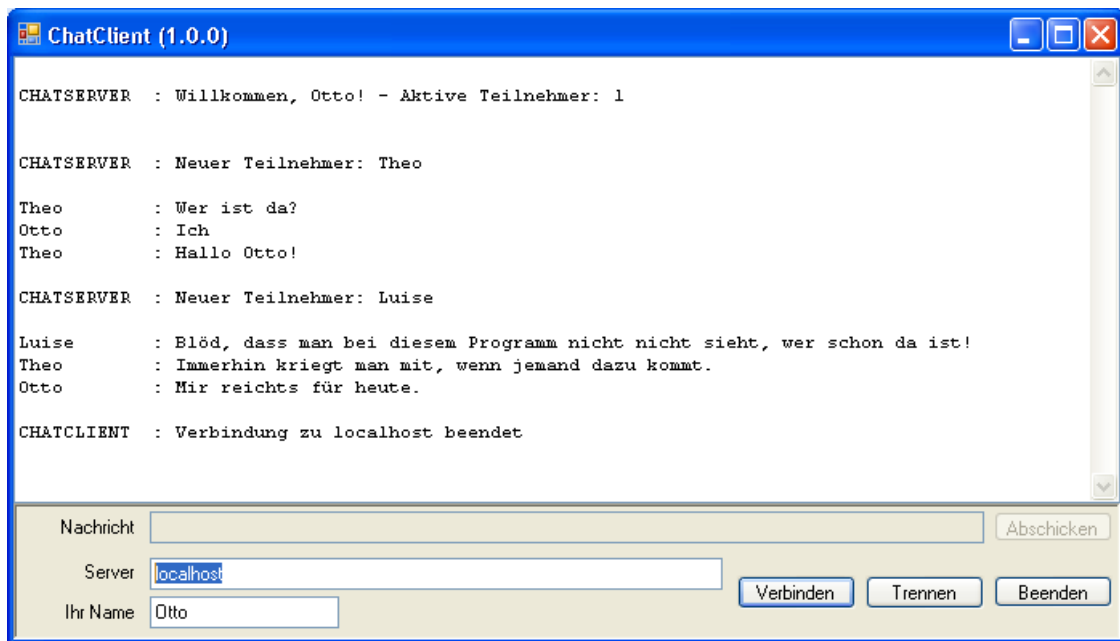
Ein typisches Server-Verlaufsprotokoll könnte so aussehen:



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 (Wert **true** für die Eigenschaft **Multiline**) mit vertikalem Rollbalken (Wert **Vertical** für die Eigenschaft **ScrollBars**)

Die Benutzeroberfläche der Klientenanwendung sollte ungefähr so aussehen:



Im ChatRoom - Anwendungsprotokoll sollen folgende Nachrichten eine spezielle Bedeutung haben:

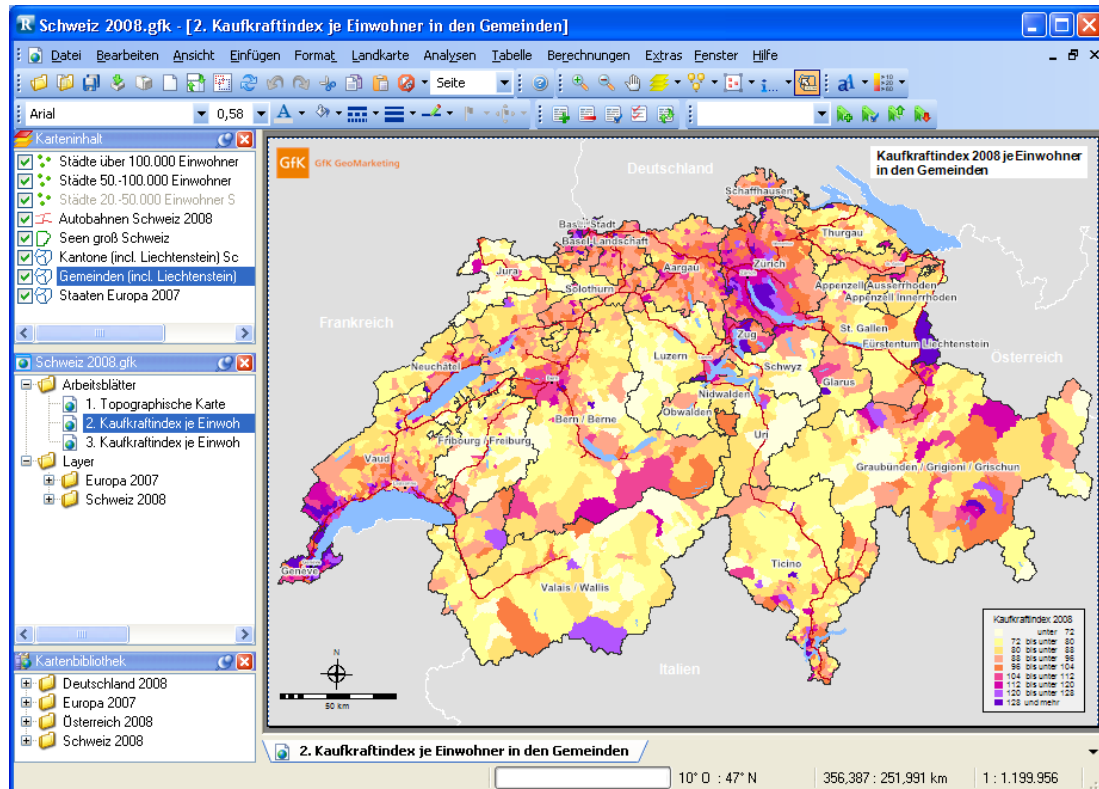
#Quit!	Ein Klient meldet sich ab.
#Bye!	Serverantwort auf die Abmeldung eines Klienten
#UserName='name'!	Ein Klient nennt seinen Wunschnamen.
#EndOfService!	Der Server informiert die Klienten über den bevorstehenden Dienstschluss.

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 **Control** - Methode **Select()** setzt man den Eingabefokus auf die gerade benötigte **TextBox**, z.B. nach dem Verbindungsaufbau auf die Nachrichten - **TextBox**.
- Über die **Form**-Eigenschaft **AcceptButton** verbindet man die **Enter**-Taste mit der gerade am ehesten benötigten Schaltfläche.

15 Grafikausgabe mit dem GDI+

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 neu entwickelte Geografische Informationssystem *RegioGraph* der Firma *GfK Geomarketing* (Version 10) 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).

Windows stellt den Anwendungsprogrammen seit jeher mit dem **Graphics Device Interface (GDI)** eine umfangreiche Sammlung von graphischen Ausgaberroutinen zur Verfügung. Mit dieser Software-schicht wird die 2D-Grafik - Programmierung von den spezifischen Eigenschaften bestimmter Ausgabegeräte (z.B. Grafikkarte, Drucker) unabhängig. Das.NET – Framework bietet für die erweiterte Fassung **GDI+** mit zahlreichen Lösungen für anspruchsvolle Vektor- und Pixelgrafiken eine objektorientierte Schnittstelle, wobei die zuständigen Klassen im Namensraum **System.Drawing** und diversen Unterräumen (z.B. **System.Drawing.Text**) zu finden sind.

Als Zeichengrundlage werden Objekte der Klassen **Form** und **Panel** bevorzugt, doch kommen grundsätzlich auch andere Steuerelemente in Frage.

15.1 Die Klasse Graphics

Bei der Grafikausgabe im .NET – Framework spielen Objekte der Klasse **Graphics** aus dem Namensraum **System.Drawing** eine zentrale Rolle:

- Ein solches Objekt ist mit einem **Windows-Gerätekontext** verbunden und eröffnet damit den Zugang zu einer Zeichenoberfläche.

- Die Eigenschaften eines **Graphics**-Objekts enthalten relevante Informationen für die Grafikausgabe, z.B. die horizontale und vertikale Auflösung des Ausgabegeräts.
- Ein **Graphics**-Objekt beherrscht diverse Methoden zur Ausgabe von elementaren Grafikelementen (z.B. **DrawLine()**, **FillRectangle()**), Texten (z.B. **DrawString()**) und Bildern (z.B. **DrawImage()**).

In der **FormOnMouseDown()** - Methode des folgenden Beispielprogramms, die beim Ereignis **MouseDown** des Anwendungsfensters registriert ist, wird mit der **Control**-Methode **CreateGraphics()** ein **Graphics**-Objekt erzeugt, das den Klientenbereich des Anwendungsfensters als Zeichenoberfläche erschließt:

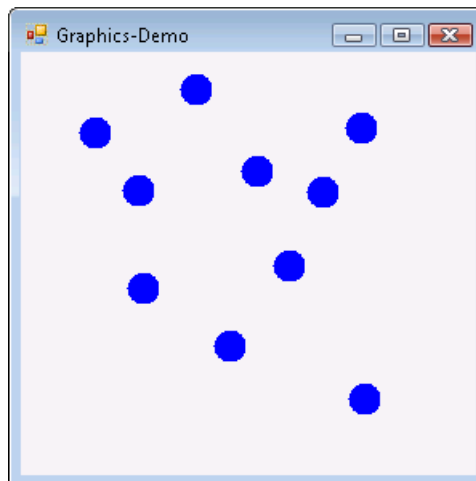
```
using System;
using System.Windows.Forms;
using System.Drawing;

class GraphicsDemo : Form {
    GraphicsDemo() {
        Text = "Graphics-Demo";
        MouseDown += new MouseEventHandler(FormOnMouseDown);
    }

    void FormOnMouseDown(object sender, MouseEventArgs e) {
        Graphics g = this.CreateGraphics();
        g.FillEllipse(Brushes.Blue, e.X - 10, e.Y - 10, 20, 20);
        g.Dispose();
    }

    [STAThread]
    static void Main() {
        Application.Run(new GraphicsDemo());
    }
}
```

Das **Graphics**-Objekt wird per **FillEllipse()**-Aufruf gebeten, mit Hilfe eines blauen Pinsels, ansprechbar über die statische Eigenschaft **Blue** der Klasse **Brushes**, um die Klickstelle (**e.X**, **e.Y**) einen Kreis mit einem Durchmesser von 20 Pixeln zu malen. Dies ermöglicht es dem Benutzer, mit mehreren Mausklicks ein Pünktchenmuster nach eigenem Geschmack zu entwerfen, z.B.



Neben seinen Zeichenkünsten bietet das **Graphics**-Objekt in etlichen Eigenschaften auch Informationen und Einstellmöglichkeiten. So lässt sich z.B. mit den Eigenschaften **DpiX** und **DpiY** die Auflösung des angeschlossenen Gerätes in horizontaler und vertikaler Richtung ermitteln, und per **TextRenderingHint** kann man die Schriftglättung beeinflussen.

Den Ausgabemethoden (wie z.B. **FillEllipse()**) müssen als Werkzeuge zu verwendende Objekte übergeben werden (z.B. Pinsel und Stifte, siehe unten).

15.2 GDI-Ressourcen schonen

Ein **Graphics**-Objekt verbraucht in erheblichem Umfang GDI-Ressourcen des Betriebssystems, die zu den *unverwalteten*, nicht unter der Kontrolle der CLR stehenden, Ressourcen gehören (wie z.B. auch Dateien oder Netzwerkverbindungen, siehe oben). Engpässe bei GDI-Ressourcen können (z.B. in Anwendungen mit vielen Formularen) durchaus für erhebliche Probleme sorgen, z.B. für Darstellungsfehler beim Zeichnen der Steuerelemente. Daher sollte man für ein per **CreateGraphics()** erstelltes Objekt nach getaner Arbeit die **Dispose()**-Methode aufrufen, um die GDI-Ressourcen sofort frei zu geben, z.B.:

```
g.Dispose();
```

Zwar ruft auch der .NET - Garbage Collector die **Dispose()**-Methode auf, doch orientiert er seine Arbeitsplanung am verfügbaren *verwalteten* Speicher und an der verfügbaren Prozessorkapazität. Somit ist *nicht* garantiert, dass die GDI-Ressourcen der obsolet gewordenen (nicht mehr referenzierten) **Graphics**-Objekte rechtzeitig frei gegeben werden.

Ein voreiliges **Dispose()** ist natürlich strikt zu vermeiden, weil es zu einem Laufzeitfehler führt. Vielleicht findet Microsoft eine Möglichkeit, das Verhalten des Garbage Collectors so zu optimieren, dass Anwendungsprogrammierer nicht mehr zwischen verwalteten und unverwalteten Ressourcen unterscheiden müssen.

Zu den GDI-Ressourcen, die möglichst früh zurückgegeben werden sollten, gehören neben den Gerätekontexten auch (zugehörige FCL-Klasse in Klammern):

- Pinsel (**Brush**)
- Stifte (**Pen**)
- Pixel-Grafiken (**Bitmap**)
- Schriftarten (**Font**)

Bei den vordefinierten, über statische Eigenschaften der Klassen **Pens**, **SystemPens**, **Brushes**, **SystemBrushes** und **SystemFonts** (vgl. Abschnitte 15.5 und 15.7) verfügbaren Objekten (z.B. **Brushes.Blue**) ist ein **Dispose()**-Aufruf allerdings überflüssig und schädlich (vgl. Bayer 2006, S. 874).

Für die Grafikausgaben in **Paint**-Ereignisbehandlungsmethoden (siehe Abschnitt 15.3) wird uns (über eine Eigenschaft des Ereignisbeschreibungsobjekts) ein **Graphics**-Objekt frei Haus geliefert, das wir nach Gebrauch einfach vergessen dürfen.

Ohne **Dispose()** kommt man auch bei GDI-lastigen Objekten aus, die im Konstruktor des Hauptformulars zur Verwendung im gesamten Programm erstellt werden.

Im weiteren Verlauf von Abschnitt 15 wird bei trivialen Beispielprogrammen der Einfachheit halber auf die konsequente Schonung von GDI-Ressourcen verzichtet.

15.3 Fensterrenovierungsmethoden

Wenn ein Fenster (neu) gezeichnet werden muss (z.B. beim ersten Auftritt oder nach der Rückkehr aus der Taskleiste), dann erhält die zugehörige Anwendung von Windows eine **WM_PAINT**-Nachricht. In den von **System.Windows.Forms.Control** abgeleiteten Klassen wird daraufhin die Methode **OnPaint()** aufgerufen, welche ihrerseits das **Control**-Ereignis **Paint** auslöst, d.h. die registrierten Delegaten benachrichtigt. Wir haben uns in Abschnitt 9.5 ausführlich mit der Rolle der **On**-Methoden bei der Ereignisbehandlung beschäftigt.

Ein Fensterobjekt ist selbst dafür verantwortlich, seinen Klientenbereich (= Gesamtfläche abzüglich Rahmen, Titelzeile, Menüzeile, Rollbalken und ggf. Symbolleisten) in einer geeigneten Ereignisbehandlungsmethode neu zu zeichnen. Weil dem Pünktchen-Programm in Abschnitt 15.1 derartige

Vorkehrungen fehlen, verliert das Fenster z.B. bei Besuchen in der Taskleiste sein Pünktchenmuster, was betroffene Künstler sicher frustrieren wird.

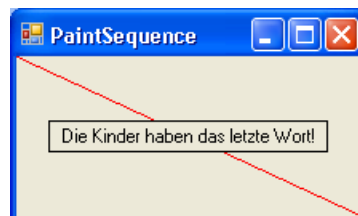
Im .NET - Framework stehen zwei Möglichkeiten zur Verfügung, auf die **WM_PAINT**-Nachricht zu reagieren:

- **PaintEventHandler**
Es wird eine Methode erstellt und über einen Delegaten vom Typ **PaintEventHandler** beim **Paint**-Ereignis des betroffenen Objekts registriert. Wir haben uns in Abschnitt 9.3 ausführlich mit dem Erstellen und Registrieren von Ereignisbehandlungsmethoden beschäftigt.
- **OnPaint()-Überschreibung**
Die von **System.Windows.Forms.Control** geerbte Methode **OnPaint()** wird überschrieben, was natürlich nur in einer eigenen Klassendefinition möglich ist. Da in unseren GUI-Programmen das Formular (Hauptfenster) stets Objekt einer aus **System.Windows.Forms.Form** abgeleiteten Klasse ist, steht die **OnPaint()**-Technik dort zur Verfügung.

Unabhängig von der gewählten Technik muss eine Methode zum Zeichnen des Fensterinhalts ihre Arbeit sehr flott verrichten, weil sie oft und zu beliebigen Zeitpunkten aufgerufen wird. Es wäre z.B. keine gute Idee, Datei- oder gar Netzwerkzugriffe in einer solchen Methode vorzunehmen.

Beim *Verschieben* eines Fensters wird übrigens *keine* **WM_PAINT** - Ereignisbehandlungsmethode beauftragt, um den Klientenbereich am neuen Ort zu zeichnen. Diese Routinearbeit erledigt das Betriebssystem selbständig. Wird ein Fenster allerdings über den Bildschirmrand hinaus geschoben, geht sein Inhalt teilweise verloren und muss ggf. per **WM_PAINT** - Ereignisbehandlungsmethode wiederhergestellt werden.

Sind Steuerelemente in einem Container enthalten, werden eintreffende **WM_PAINT**-Nachrichten an diese Kindfenster propagiert. Weil die **Paint**-Methoden der Kindfenster nach der **Paint**-Methode des Containers ausgeführt werden, haben (wie in einer guten Familie) die Kinder das letzte Wort, z.B. (Label auf einem Formular mit einer roten Diagonale):



15.3.1 PaintEventHandler

Im folgenden Programm kommt die **PaintEventHandler**-Technik zum Einsatz, um auf **Paint**-Ereignisse mit dem Renovieren des Fensterinhalts zu reagieren:

```
using System;
using System.Windows.Forms;
using System.Drawing;
using System.Collections.Generic;

class PEHDemo : Form {
    List<Point> punkte = new List<Point>();

    PEHDemo() {
        Text = "PaintEventHandler-Demo";
        MouseDown += new MouseEventHandler(FormOnMouseDown);
        Paint += new PaintEventHandler(FormOnPaint);
    }
}
```

```

void FormOnPaint(object sender, PaintEventArgs e) {
    foreach (Point p in punkte)
        e.Graphics.FillEllipse(Brushes.Blue, p.X, p.Y, 20, 20);
}

void FormOnMouseDown(object sender, MouseEventArgs e) {
    Graphics g = this.CreateGraphics();
    g.FillEllipse(Brushes.Blue, e.X - 10, e.Y - 10, 20, 20);
    g.Dispose();
    punkte.Add(new Point(e.X - 10, e.Y - 10));
}

[STAThread]
static void Main() {
    Application.Run(new PEHDemo());
}
}

```

Im Vergleich zu der Punktverlustversion in Abschnitt 15.1 besitzt das neue Programm ein Objekt der generischen Kollektionsklasse **List<Point>** (vgl. Abschnitt 7.2), um die vom Benutzer angeklickten Positionen als Instanzen der Struktur **Point** (siehe unten) zu speichern, was in der Ereignismethode **FormOnMouseDown()** geschieht. Dort wird außerdem weiterhin für die sofortige Anzeige neuer Punkte per **FillEllipse()**-Aufruf gesorgt.

Dem registrierten **PaintEventHandler** **FormOnPaint()** wird beim Aufruf ein **Graphics**-Objekt zum Bemalen des Fensters frei Haus geliefert: Es ist über die **Graphics**-Eigenschaft des übergebenen **PaintEventArgs**-Objekts ansprechbar. Zudem müssen wir uns *nicht* um die Freigabe der mit diesem **Graphics**-Objekt verbundenen GDI-Ressourcen kümmern und insbesondere seine **Dispose()**-Methode *nicht* aufrufen (vgl. Abschnitt 15.2).

15.3.2 OnPaint() überschreiben

Im folgenden Programm wird zur Behandlung die geerbte **OnPaint()**-Methode überschrieben, um auf die **WM_PAINT**-Nachricht zur Renovierung des Fensterinhalts zu reagieren:

```

using System;
using System.Windows.Forms;
using System.Drawing;
using System.Collections.Generic;

class OnPaintDemo : Form {
    List<Point> punkte = new List<Point>();

    OnPaintDemo() {
        Text = "OnPaint-Demo";
        MouseDown += new EventHandler(FormOnMouseDown);
    }

    protected override void OnPaint(PaintEventArgs e) {
        foreach (Point p in punkte)
            e.Graphics.FillEllipse(Brushes.Blue, p.X, p.Y, 20, 20);
    }

    void FormOnMouseDown(object sender, MouseEventArgs e) {
        Graphics g = this.CreateGraphics();
        g.FillEllipse(Brushes.Blue, e.X - 10, e.Y - 10, 20, 20);
        g.Dispose();
        punkte.Add(new Point(e.X - 10, e.Y - 10));
    }
}

```

```
[STAThread]
static void Main() {
    Application.Run(new OnPaintDemo());
}
```

Wie eine **Paint**-Behandlungsmethode erhält auch **OnPaint()** beim Aufruf ein **Graphics**-Objekt per **PaintEventArgs**-Parameter, ohne sich um die Freigabe von GDI-Ressourcen kümmern zu müssen (vgl. Abschnitt 15.2).

Kommen **PaintEventHandler** zusammen mit einer **OnPaint()**-Überschreibung zum Einsatz, muss in der Überschreibung die **OnPaint()**-Methode der Basisklasse aufgerufen werden, damit die **PaintEventHandler** nicht abgekoppelt werden (vgl. Abschnitt 9.5).

15.3.3 Fensteraktualisierung per Programm anfordern

Bei allen bisherigen Varianten des Pünktchen-Programms wurde (auch) in einem **MouseEventHandler**, also außerhalb jeder **WM_PAINT** - Behandlungsmethode, auf die Fensteroberfläche gezeichnet. Dies geschah zunächst naiv, später in der Absicht, neben der weit blickenden Vorbereitung auf eine vom Betriebssystem angeforderte Fensterrenovierung auch für ein sofort sichtbares Ergebnis zu sorgen.

Eine Alternative zu der (ergänzenden) Direktausgabe besteht in manchen Fällen darin, für ein Fenster mit der **Control**-Methode **Invalidate()**-Aufruf die Ausführung der **WM_PAINT** - Behandlungsmethode anzufordern, z.B.:

```
void FormOnMouseDown(object sender, MouseEventArgs e) {
    punkte.Add(new Point(e.X - 10, e.Y - 10));
    Invalidate();
}
```

Im Beispiel ist allerdings bei zunehmender Anzahl von Klecksen an einem störenden Flackern zu bemerken, dass nun bei jedem neuen Mausklick der gesamte Klientenbereich neu gezeichnet wird. Bei jeder Fensterrenovierung wird zunächst die gesamte Fläche grundiert, d.h. in der Hintergrundfarbe neu gestrichen. Danach werden die eigentlichen Zeichenmethoden ausgeführt, so dass an vielen Stellen in kurzer Folge unterschiedliche Farben auftauchen, was zum Eindruck des Flackerns führt (Luis & Strasser 2008, S. 871). Alternative **Invalidate()**-Überladungen erlauben es, über eine **Rectangle**-Instanz (siehe Abschnitt 15.4.3) bzw. ein **Region**-Objekt (siehe Abschnitt 15.10.3) einen rechteckigen oder anders geformten *Teil* eines Fensters zu verwerfen. Weil dann (wie bei einem Zeichenbereich, vgl. Abschnitt 15.10.2) die Ausgaben der **Paint**-Ereignismethode nur noch auf den defekten Bereich des Fensters auswirken, lässt sich oft ein Flackern vermeiden. Im Beispiel wird dieser Effekt durch eine einfache Erweiterung der Methode **FormOnMouseDown()** zuverlässig erreicht:

```
void FormOnMouseDown(object sender, MouseEventArgs e) {
    Point punkt = new Point(e.X - 10, e.Y - 10);
    punkte.Add(punkt);
    Invalidate(new Rectangle(punkt, new Size(20, 20)));
}
```

Eine andere Möglichkeit zur Vermeidung des Flackerns besteht in der **Doppelpufferungstechnik** (engl.: *double buffering*), die im Formalkonstruktor eingeschaltet werden kann:

```
DoubleBuffered = true;
```

Nach Aktivierung der Doppelpufferung zeichnen **Paint**-Ereignismethoden zunächst auf eine unsichtbare Kopie der betroffenen Fläche und ersetzen erst nach Abschluss aller Arbeiten das Original durch die neue Version. In der Regel unterscheiden sich die beiden nacheinander sichtbaren Bilder nicht stark, und insbesondere unterbleibt das zwischenzeitliche Grundieren der Zeichenfläche in der

Hintergrundfarbe. Weil kaum schnelle Farbwechsel auftreten, unterbleibt das Flackern. Die per **Form**-Eigenschaft **DoubleBuffered** aktivierte automatische Doppelpufferung wirkt sich nur auf **Paint**-Ereignismethoden aus. Luis & Strasser (2008, S. 874) beschreiben, wie man bei beliebigen Grafikausgaben eine eigene Doppelpufferung implementieren kann.

Vor der Reaktion auf eine per **Invalidate()**-Aufruf ausgelöste **WM_PAINT**-Nachricht, kann einige Zeit vergehen:

- Zunächst muss die aktuell ausgeführte Ereignisbehandlungsmethode (z.B. der **MouseEvent-Handler**) enden.
- Befinden sich andere Nachrichten in der Warteschlange, werden diese zuerst abgearbeitet.

Man kann allerdings mit der **Control**-Methode **Update()** eine sofortige Aktualisierung der für ungültig erklärten Klientenbereiche erzwingen, z.B.:

```
void FormOnMouseDown(object sender, MouseEventArgs e) {
    punkte.Add(new Point(e.X - 10, e.Y - 10));
    Invalidate();
    Update();
}
```

Beim gemeinsamen Einsatz der Methoden **Invalidate()** und **Update()** bestimmt erstere den Umfang der Renovierung und letztere den Zeitpunkt. Einen **Invalidate()**-Aufruf ohne Angabe der Schadstelle mit anschließendem **Update()**-Aufruf kann man äquivalent durch einen Aufruf der **Control**-Methode **Refresh()** ersetzen, z.B.:

```
void FormOnMouseDown(object sender, MouseEventArgs e) {
    punkte.Add(new Point(e.X - 10, e.Y - 10));
    Refresh();
}
```

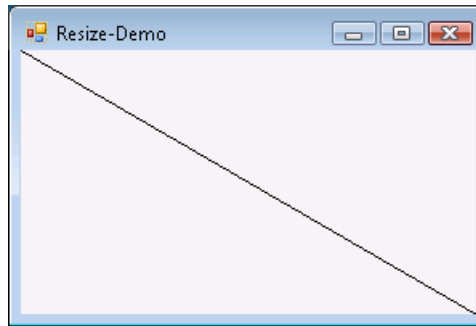
15.3.4 ResizeRedraw

In Abschnitt 15.3.3 wurde im Zusammenhang mit der **Control**-Methode **Invalidate()** von der Möglichkeit berichtet, eine Fensterrenovierung aus Kostengründen auf den beschädigten Bereich zu beschränken. Manchmal führt jedoch übertriebene Sparsamkeit zu einer unglücklichen Kombination von alten und neuen Fensteranteilen. Das folgende Programm

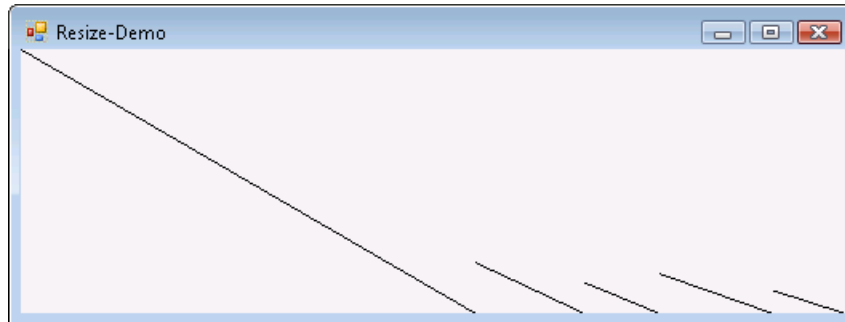
```
using System;
using System.Windows.Forms;
using System.Drawing;

class ResizeDemo : Form {
    ResizeDemo() {
        Text = "Resize-Demo";
        Height = 200; Width = 300;
    }
    protected override void OnPaint(PaintEventArgs e) {
        e.Graphics.DrawLine(Pens.Black, 0, 0, ClientSize.Width, ClientSize.Height);
    }
    [STAThread]
    static void Main() {
        Application.Run(new ResizeDemo());
    }
}
```

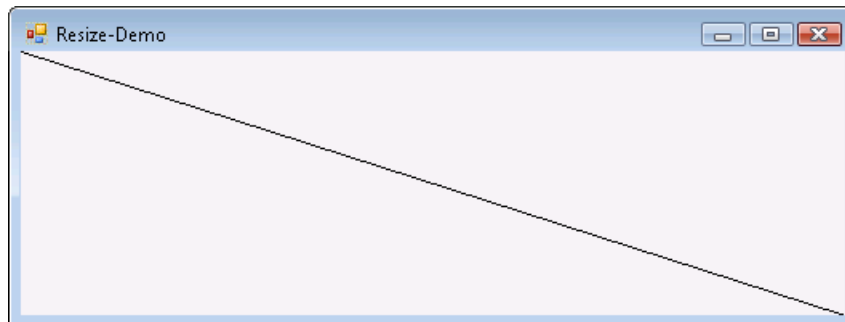
zeichnet eine Diagonale auf sein Fenster:



Bei einer Verbreiterung des Fensters erscheinen „wirre“ Muster, die eventuell (bei Benutzereinstellung **Fensterinhalt beim Ziehen anzeigen**) vom Tempo der Mausbewegung abhängen, z.B.:



Nach der Rückkehr aus der Taskleiste zeigt sich wieder das erwartete Bild:



Per Voreinstellung beschränkt das der **OnPaint()**-Methode übergebene **Graphics**-Objekt alle Ausgaben auf das renovierungsbedürftige Rechteck, im Beispiel also auf den hinzugekommenen Bereich. Alle Ausgaben außerhalb dieses Rechtecks werden abgeschnitten, und der alte Fensterbereich bleibt unverändert. Je nach Mausgeschwindigkeit beim Verbreitern des Fensters wird die **OnPaint()**-Methode unterschiedlich oft aufgerufen, so dass ein „Zufallsmuster“ entsteht. Nach der Rückkehr aus der Taskleiste umfasst der **Graphics**-Arbeitsbereich das gesamte Fenster.

Hier wird offenbar an der falschen Stelle gespart, und wir setzen im Fensterkonstruktor die **Control**-Eigenschaft **ResizeRedraw** auf den Wert **true**, damit bei jeder Größenänderung der gesamte Klientenbereich des Fensters neu gezeichnet wird:

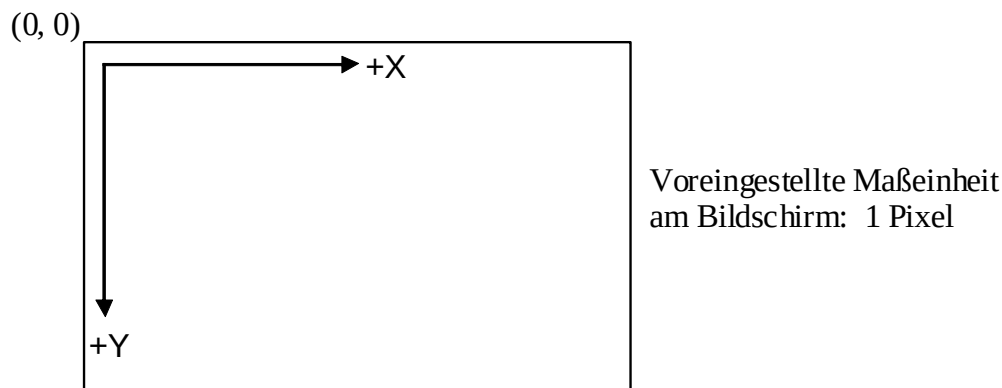
```
ResizeRedraw = true;
```

15.4 Positionen und Größen

In diesem Abschnitt werden wichtige Typen zur einfachen Verwaltung von Positionen und Größen vorgestellt.

15.4.1 Standardkoordinatensystem

Wie Sie längst wissen, befindet sich der Ursprung (0, 0) des GDI+ - Standardkoordinatensystems per Voreinstellung in der linken oberen Ecke des Klientenbereichs. Die **X**-Werte wachsen nach rechts und die **Y**-Werte nach unten:



Wer Koordinaten in Bezug auf den gesamten Bildschirm ermitteln oder zwischen Bildschirm- und Klientenbereichskordinaten konvertieren möchte, kann die folgenden **Control**-Methoden verwenden:

- **Point PointToScreen(Point ptClient)**
- **Point PointToClient(Point ptScreen)**

Im Abschnitt 15.8 werden transformierte Koordinatensysteme behandelt, wobei man ...

- den Ursprung verschieben
- und/oder die Maßeinheiten für die Achsen verändern kann.

Wir verwenden zunächst der Einfachheit halber die Maßeinheit **Pixel**. Um unterschiedliche Bildschirmauflösungen zu unterstützen oder eine Druckausgabe zu realisieren, benötigen wir jedoch metrische Maßeinheiten. Über von einem **Graphics**-Objekt verwendete Maßeinheit lässt sich über seine Eigenschaft **PageUnit** (vom Enumerationstyp **GraphicsUnit**) ermitteln oder verändern (siehe Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**).

15.4.2 Point und Size

Eine Instanz der **Point**-Struktur repräsentiert einen Punkt der Zeichenfläche mit den Koordinaten **X** und **Y**, die als Eigenschaften vom Typ **int** (mit Lese- und Schreibzugriff) implementiert sind, z.B.:

```
Point pkt1 = new Point();
```

Viele **Graphics**-Methoden fordern bzw. akzeptieren Ortsangaben über **Point**-Instanzen, z.B.:

```
g.DrawLine(pen, pkt1, pkt2);
```

Mit der **Point**-Methode **Offset()** gelingt das Verschieben eines Punktes in **X**- und **Y**-Richtung besonders bequem, z.B.:

```
pkt1.Offset(distX, distY);
```

Dieser Aufruf wirkt wie:

```
pkt1.X += distX;
pkt1.Y += distY;
```

Was die **Point**-Struktur für Positionsangaben leistet, bewerkstelligt die analog aufgebaute Struktur **Size** für Größenangaben, die wir seit Abschnitt 9 schon oft bei der Konfiguration von Steuerelementen verwendet haben. Ihre Instanzen verfügen über die Eigenschaften **Height** und **Width** vom Typ **Int32** (mit Lese- und Schreibzugriff). Viele FCL-Klassen besitzen Eigenschaften vom Typ **Size**, so dass für eine Wertzuweisung **Size**-Instanzen benötigt werden, z.B. bei der **ClientSize**-Eigenschaft eines Formulars:

```
ClientSize = new Size(200, 200);
```

Die Größe des Klientenbereichs festzustellen oder zu verändern ist für ein Programm oft interessanter als analoge Operationen für das gesamte Fenster (inkl. Rahmen, Titelzeile, Bildlaufleisten etc.) über die **Form**-Eigenschaft **Size** auszuführen.

Die Verwandtschaft der Strukturen **Point** und **Size** geht so weit, dass mit Hilfe des überladenen Additions- bzw. Subtraktions-Operators eine **Size**-Instanz zu einer **Point**-Instanz addiert bzw. von ihr subtrahiert werden kann, z.B.:

```
Point pkt2 = pkt1 + ClientSize;
```

Alle in diesem Abschnitt vorgestellten Anweisungen entstammen dem folgenden Programm:

```
using System;
using System.Windows.Forms;
using System.Drawing;

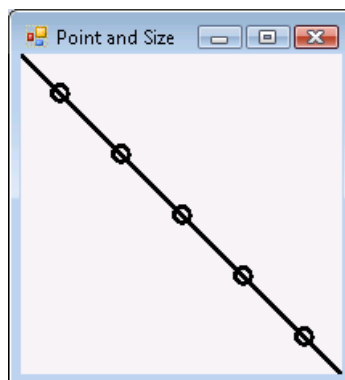
class PointAndSize : Form {
    Pen pen = new Pen(Color.Black, 3);

    PointAndSize() {
        Text = "Point and Size";
        ResizeRedraw = true;
        ClientSize = new Size(200, 200);
    }

    protected override void OnPaint(PaintEventArgs e) {
        Graphics g = e.Graphics;
        Point pkt1 = new Point(); // Initialisierung mit (0, 0)!
        Point pkt2 = pkt1 + ClientSize;
        g.DrawLine(pen, pkt1, pkt2);
        int distX = (ClientSize.Width - 10) / 5;
        int distY = (ClientSize.Height - 10) / 5;
        pkt1.Offset(distX/2, distY/2);
        for (int i = 0; i < 5; i++) {
            g.DrawEllipse(pen, pkt1.X, pkt1.Y, 10, 10);
            pkt1.Offset(distX, distY);
        }
    }

    [STAThread]
    static void Main() {
        Application.Run(new PointAndSize());
    }
}
```

Es zeichnet eine Diagonale vom Punkt `pkt1` bis zum `pkt1 + ClientSize` sowie fünf Kreise um einen per **Offset()** verschobenen Mittelpunkt:



Später werden wir auch mit *metrischen* Koordinatensystemen arbeiten, die nicht auf Pixeln basieren. Dort werden die Strukturen **PointF** bzw. **SizeF** benötigt, die sich von **Point** bzw. **Size** im Wesentlichen durch Positions- bzw. Größenangaben vom Typ **float** unterscheiden.

15.4.3 Rectangle

Viele **Graphics**-Methoden akzeptieren als Orts- und Erstreckungsangabe ein Rechteck, das in der FCL durch eine Instanz der **Rectangle**-Struktur repräsentiert wird, z.B.:

```
Rectangle ra = new Rectangle(0, 0, 100, 100);

e.Graphics.DrawEllipse(rotstift, ra);
```

Der im Beispiel verwendete **Rectangle**-Konstruktor erwartet die Koordinaten der linken oberen Ecke sowie die Breite und Höhe des Rechtecks. Ein alternativer Konstruktor akzeptiert ein Paar aus einer **Point**- und einer **Size**-Instanz.

Von den diversen **Rectangle**-Methoden kommen im folgenden Programm **Offset()** und **Inflate()** zum Einsatz:

```
using System;
using System.Windows.Forms;
using System.Drawing;

class RectangleDemo : Form {
    Rectangle ra = new Rectangle(0, 0, 100, 100);
    Pen rotstift = new Pen(Color.Red, 5);

    RectangleDemo() {
        Text = "Rectangle - Demo";
        ClientSize = new Size(400, 400);
        FormBorderStyle = FormBorderStyle.FixedSingle;
    }

    protected override void OnPaint(PaintEventArgs e) {
        e.Graphics.DrawEllipse(rotstift, ra);
    }

    protected override void OnMouseDown(MouseEventArgs e) {
        if (e.Button == MouseButtons.Left) {
            if (e.X > Width / 2) {
                if (ra.Right < Width - 20) ra.Offset(20, 20);
            } else {
                if (ra.X >= 20) ra.Offset(-20, -20);
            }
        } else {
            if (e.X > Width / 2) {
                if (ra.X >= 20 && ra.Right < Width - 20) ra.Inflate(20, 20);
            } else {
                if (ra.Width > 20) ra.Inflate(-20, -20);
            }
        }
        Refresh();
    }

    [STAThread]
    static void Main() {
        Application.Run(new RectangleDemo());
    }
}
```

Nach einem *linken* Mausklick in die linke bzw. rechte Formularhälfte wird das begrenzende Rechteck des Kreises per **Offset()**-Aufruf nach links oben bzw. rechts unten verschoben. Der Aufruf

```
ra.Offset(20, 20)
```

wirkt wie:

```
ra.X += 20;
ra.Y += 20;
```

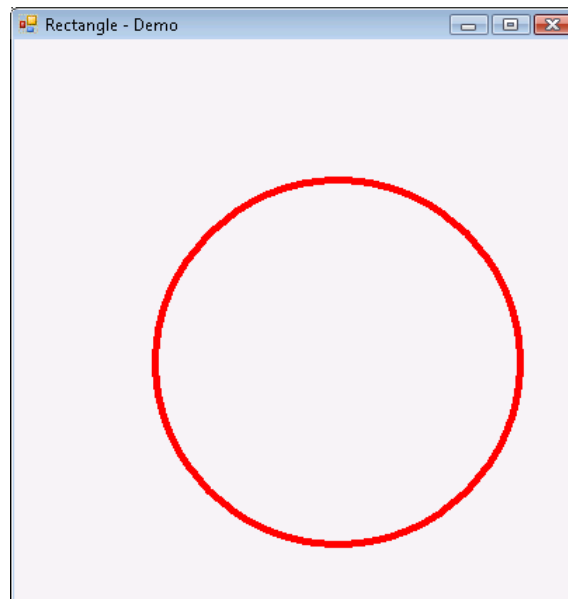
Mit der *rechten* Maustaste können Benutzer den Kreis durch Klick in die linke Fensterhälfte verkleinern bzw. durch Klick in die rechte Fensterhälfte vergrößern. Dazu wird das Begrenzungsrechteck per **Inflate()**-Methode aufgefordert, sich bei konstantem Mittelpunkt zu vergrößern bzw. zu verkleinern. Der Aufruf

```
ra.Inflate(20, 20)
```

wirkt wie:

```
ra.X -= 20;
ra.Y -= 20;
ra.Width += 40;
ra.Height += 40;
```

Es kommt also auf der X- und der Y-Achse bei fixiertem Mittelpunkt in positiver *und* in negativer Richtung zu einer Größenzunahme gemäß Parameterwert. Insgesamt findet der Benutzer mit wenigen Mausklicks seine bevorzugte Position und Größe für den vom Rechteck begrenzten Kreis, z.B.:



Mit Hilfe der folgenden **Rectangle**-Eigenschaften verhindert das Programm unerwünschte Positionen und Größen:

- **Width**
Breite des Rechtecks
- **Right**
Container-relative X-Koordinate des rechten Randes (= **Left** + **Width**)

Weitere Eigenschaften und Methoden der **Rectangle**-Struktur sind in der FCL-Dokumentation zu finden.

Die später benötigte Struktur **RectangleF** steht zu **Rectangle** in derselben Beziehung wie **Point** bzw. **Size** zu **PointF** bzw. **SizeF**.

15.5 Farben und Zeichenwerkzeuge

Um per **Graphics**-Objekt auf eine Fensterfläche zu zeichnen, benötigt man Farben und Werkzeuge (Stifte, Pinsel).

15.5.1 Farben

Im .NET – Framework wird ein **ARGB-Farbmodell** mit einem Alpha- (Transparenz-) Kanal sowie den Farbkanälen Rot, Grün und Blau verwendet. Für alle 4 Kanäle stehen Ausprägungen von 0 bis

255 zur Verfügung, wobei der Alpha-Wert 0 für Transparenz und der Alpha-Wert 255 für komplette Deckung steht.

Zum Verwalten von Farben dient die Struktur **Color** aus dem Namensraum **System.Drawing**. Sie enthält 140 statische Eigenschaften zur Spezifikation von Standardfarben, z.B.

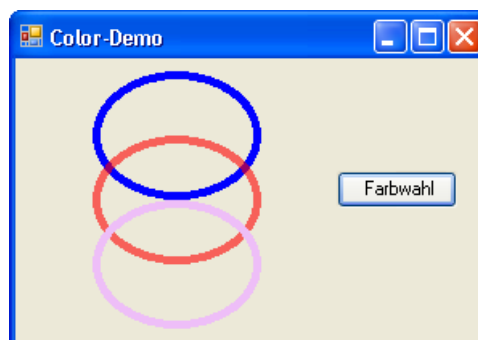
`Color.Red`

Die Liste der Farbnamen ist übrigens relativ kompatibel mit den von Webbrowsern beim Rendern von HTML-Seiten unterstützten Bezeichnungen.

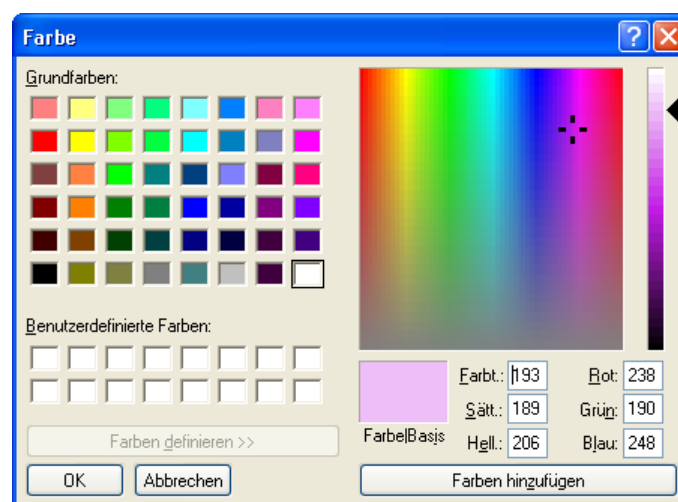
Eigene Farben mixt man mit der statischen **Color**-Methode **FromArgb()** zusammen. In der folgenden **OnPaint()**-Überschreibung wird über eine Ellipse in sattem Blau eine zweite in transparentem Rot (Deckungsgrad 150) gemalt:

```
protected override void OnPaint(PaintEventArgs e) {
    e.Graphics.DrawEllipse(penBlue, 50, 10, 100, 75);
    e.Graphics.DrawEllipse(penRed, 50, 50, 100, 75);
    if (puc) {
        Pen penUs = new Pen(uc, 5);
        e.Graphics.DrawEllipse(penUs, 50, 90, 100, 75);
        penUs.Dispose();
    }
}
```

Das Ergebnis:



Die Farbe einer dritten Ellipse kann der Benutzer nach einem Klick auf den Schalter **Farbwahl** festlegen:



Es bedarf keiner großen Anstrengung, diesen Standarddialog zur Farbauswahl zu präsentieren und sein Ergebnis auszuwerten, wie die Behandlungsmethode zum **Click**-Ereignis des Befehlsschalters zeigt:

```
protected void FarbwahlOnClick(object sender, EventArgs e) {
    if (cd.ShowDialog() == DialogResult.OK) {
        uc = cd.Color;
        puc = true;
        Refresh();
    }
}
```

Es wird ein Objekt der Klasse **ColorDialog** aus dem Namensraum **System.Windows.Forms** benutzt, das nach dem Einsatz seiner **ShowDialog()**-Methode per Rückgabewert über die vom Benutzer verwendete Terminierungsmethode (**OK** oder **Cancel**) sowie ggf. per **Color**-Eigenschaft über die gewählte Farbe zu berichten weiß. Weitere Informationen zur Klasse **ColorDialog** folgen später im Kontext mit den übrigen Windows-Standarddialogen (zur Datei- und Schriftauswahl).

Die bisher im aktuellen Abschnitt vorgestellten Programmsegmente stammen aus dem Projekt:

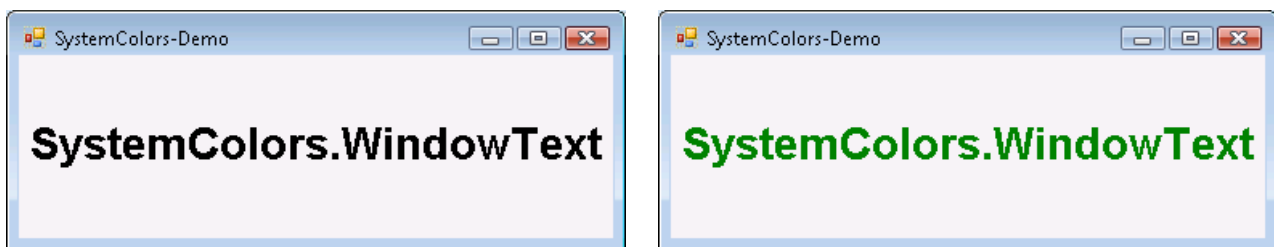
...\BspUeb\GDI+\Color

Durch Verwendung der **Windows-Systemfarben**, die über statische Eigenschaften der Klasse **System.Drawing.SystemColors** ansprechbar sind, kann man das vom Benutzer bevorzugte Windows-Design berücksichtigen. In der folgenden **OnPaint()**-Methode

```
Font ab20 = new Font("Arial", 20, FontStyle.Bold);
SolidBrush sbText = new SolidBrush(SystemColors.WindowText);

protected override void OnPaint(PaintEventArgs e) {
    e.Graphics.DrawString("SystemColors.WindowText", ab20, sbText, 3, 40);
    ab20.Dispose(); sbText.Dispose();
}
```

wird zum Schreiben die vom Benutzer gewählte Systemfarbe für Fenstertext verwendet:



Wie gleich im Abschnitt 15.5.3 zu erfahren ist, muss man ein **SolidBrush**-Objekt zum Malen (von Text etc.) in der Systemfarbe **WindowText** nicht per **new** erzeugen, sondern kann auf ein vordefiniertes Exemplar zugreifen:

```
e.Graphics.DrawString("SystemColors.WindowText", ab20,
    SystemBrushes.WindowText, 3, 40);
```

15.5.2 Pen

Beim Zeichnen von Linien (z.B. mit der **Graphics**-Methode **DrawLine()**) ist ein Objekt der Klasse **Pen** (dt.: *Stift*) zu verwenden. Beim Herstellen dieses Werkzeugs kann man über diverse Linienattribute entscheiden. Im folgenden **Pen**-Konstruktor werden Farbe und Linienstärke festgelegt:

```
Pen stift = new Pen(Color.Firebrick, 10);
```

Die zahlreichen Eigenschaften der **Pen**-Klasse bieten weitere Gestaltungsmöglichkeiten, z.B.:

- **DashStyle** (Linienstil), **StartCap** (Stil für den Linienanfang), **EndCap** (Stil für das Linienende) und **DashCap** (Stil für die Segmentgrenzen in einer gestichelten Linie)

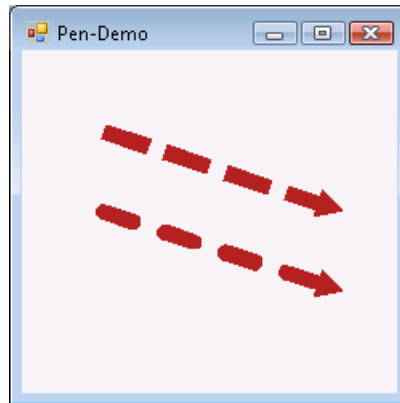
Mit diesem Stift

```
Pen stift = new Pen(Color.Firebrick, 10);
```

zeichnet die folgende **OnPaint()**-Überschreibung

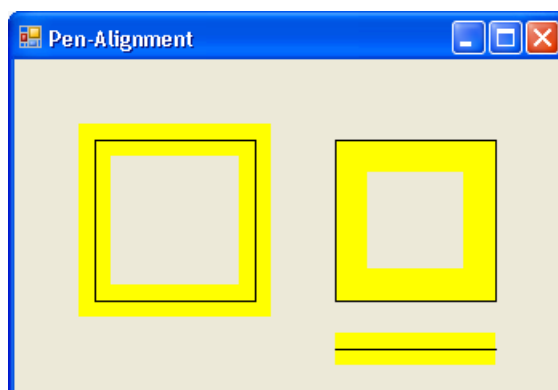
```
protected override void OnPaint(PaintEventArgs e) {
    stift.DashStyle = DashStyle.Dash;
    stift.EndCap = LineCap.ArrowAnchor;
    e.Graphics.DrawLine(stift, 50, 50, 200, 100);
    stift.DashCap = DashCap.Round;
    stift.StartCap = LineCap.Round;
    e.Graphics.DrawLine(stift, 50, 100, 200, 150);
}
```

einen gestrichelten Pfeil zunächst mit flachen und dann mit runden Segmentbegrenzungen:



Die zur Wahl der Stile benutzten Enumerationen **LineCap**, **DashStyle** und **DashCap** befinden sich im Namensraum **System.Drawing.Drawing2D**. Wenn keinen passenden Linienstil findet, kann per **DashPattern**-Eigenschaft einen eigenen definieren und über den Wert **DashStyle.Custom** der Eigenschaft **DashStyle** aktivieren.

- **Brush**
Speziell bei breiteren Linien kann man zum Füllen der Fläch(en) statt einer Farbe auch ein **Brush**-Objekt angeben (siehe Abschnitt 15.5.3) und so z.B. einen Farbverlauf erreichen.
- **Alignment**
Bei geschlossenen Kurven oder Vielecken legt diese Eigenschaft das Verhalten eines **Pen**-Objekts mit Breite größer Eins an der Umrisslinie fest. Per Voreinstellung (Wert **PenAlignment.Center**) läuft die Mitte des Stifts *auf* der Umrisslinie entlang, so dass auf beiden Seiten ein gleich breiter Streifen eingefärbt wird. So entstand das linke Quadrat im folgenden Fenster:



Es wurde unter Verwendung des Stifts

```
Pen dick = new Pen(Color.Yellow, 20);
```

von dieser **OnPaint()**-Überschreibung gezeichnet:

```
protected override void OnPaint(PaintEventArgs e) {
    Graphics g = e.Graphics;
    g.DrawRectangle(dick, 50, 50, 100, 100);
    g.DrawRectangle(Pens.Black, 50, 50, 100, 100);
    dick.Alignment = PenAlignment.Inset;
    g.DrawRectangle(dick, 200, 50, 100, 100);
    g.DrawRectangle(Pens.Black, 200, 50, 100, 100);
    g.DrawLine(dick, 200, 180, 300, 180);
    g.DrawLine(Pens.Black, 200, 180, 300, 180);
    dick.Alignment = PenAlignment.Center;
}
```

Erhält der Stift die alternative **Alignment**-Einstellung **PenAlignment.Inset**, dann landet die gesamte Linienbreite im Innenraum (siehe rechtes Quadrat). In der Ausgabe des Beispielprogramms sind außerdem die mit einem schwarzen Stift der Stärke Eins gezeichneten Umrisslinien der Quadrate zu sehen. Unter dem rechten Quadrat demonstriert eine breite *Linie*, dass sich der **Inset**-Modus nur auf *geschlossene* Figuren auswirkt.

Eine vollständige Liste mit den Eigenschaften und sonstigen Mitgliedern der **Pen**-Klasse findet sich in der FCL – Dokumentation.

Die obigen **OnPaint()**-Überschreibungen haben der Deutlichkeit halber die benötigten **Pen**-Objekte vor Ort erzeugt und die GDI-Ressourcen vor dem Verlassen der Methode per **Dispose()**-Aufruf wieder frei gegeben. In einem realen Programm sollte man es aber unbedingt vermeiden, den Erzeugungs- und Entsorgungsaufwand bei jedem **Paint**-Ereignis zu betreiben. Stattdessen sollte man die wiederholt benötigten Werkzeuge im Fensterkonstruktor erstellen und dann bis zum Programmende verwenden. Dieser Hinweis zur Effizienz stammt wie der folgende aus Louis & Strasser (2008, S. 830).

Weil die meisten Eigenschaften der Klasse **Pen** verändert werden können, ist es z.B. bei einem Wechsel der Farbe (Eigenschaft **Color**) oder Linienstärke (Eigenschaft **Width**) *nicht* erforderlich, einen neuen Stift zu erzeugen.

Die Klasse **Pens** bietet über statische Eigenschaften für jede *Standardfarbe* einen zugehörigen Stift mit der Breite Eins, z.B.:

```
e.Graphics.DrawRectangle(Pens.Black, 200, 50, 100, 100);
```

Die **SystemPens** bietet über statische Eigenschaften für jede *Windows-Systemfarbe* einen zugehörigen Stift mit der Breite Eins, z.B.:

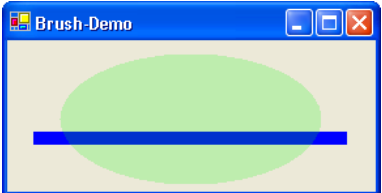
```
e.Graphics.DrawLine(SystemPens.ControlDark, 50, 50, 200, 100);
```

15.5.3 Brush

Um das *Innere* von Rechtecken, Ellipsen, Buchstaben etc. zu füllen, verwendet man ein **Brush**-Objekt (dt.: einen *Pinself*). Zur abstrakten Basisklasse **Brush** existieren diverse Konkretisierungen für spezielle optische Effekte, z.B.:

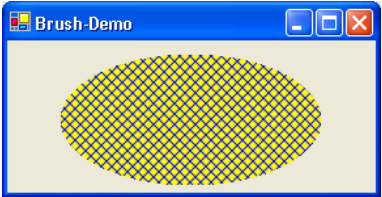
- **SolidBrush** (Namensraum **System.Drawing**)

Diese **Brush**-Ableitung ist für einfarbige Füllungen zuständig, z.B.:

Quellcodefragment	Ausgabe
<pre>g.DrawLine(new Pen(Color.Blue, 10), 20, 75, 260, 75); g.FillEllipse(new SolidBrush(Color.FromArgb(50, 0, 255, 0)), ra);</pre>	

- **HatchBrush** (Namensraum **System.Drawing.Drawing2D**)

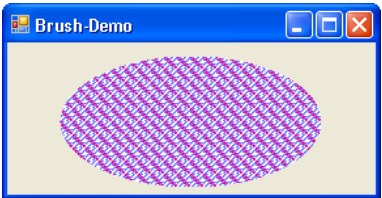
Diese **Brush**-Ableitung ist zuständig für schraffierte Füllungen, die durch ein vektoriell beschriebenes Muster, eine Vordergrundfarbe und eine Hintergrundfarbe definiert sind, z.B.:

Quellcodefragment	Ausgabe
<pre>g.FillEllipse(new HatchBrush(HatchStyle.DiagonalCross, Color.Blue, Color.Yellow), ra);</pre>	

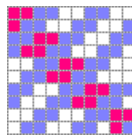
Über die **HatchStyle**-Eigenschaft lassen sich diverse Werte aus der gleichnamigen Enumeration als Muster einstellen (siehe Übungsaufgabe in Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**).

- **TextureBrush** (Namensraum **System.Drawing**)

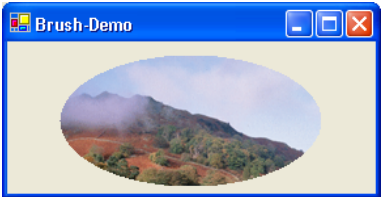
Diese **Brush**-Ableitung füllt das Innere einer Figur mit einer Bitmap-Grafik, wobei das Bild nötigenfalls bis zur Füllung der Fläche wiederholt ausgegeben wird (auf mehreren Kacheln). Für das erste Beispiel:

Quellcodefragment	Ausgabe
<pre>g.FillEllipse(new TextureBrush(new Bitmap("textur.bmp")), ra);</pre>	

wurde mit dem Windows-Zubehör *Paint* eine Bitmap-Grafik mit 10×10 Punkten erstellt:

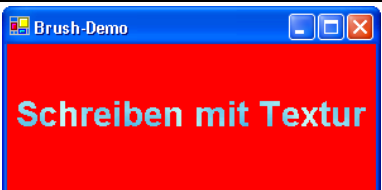


Im folgenden Beispiel ist die Bitmap-Grafik so groß, dass es *nicht* zu einer Mehrfachdarstellung kommt:

Quellcodefragment	Ausgabe
<pre>g.FillEllipse(new TextureBrush(new Bitmap("land.bmp")), ra);</pre>	

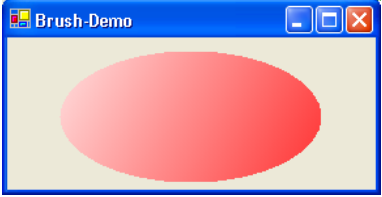
Eine verwendete Bitmap-Datei wird beim empfehlenswerten Verzicht auf eine Pfadangabe im Verzeichnis mit dem Assembly erwartet (z.B. in **...\Brush\bin\Debug**). Später werden Sie eine Möglichkeit kennen lernen, Bitmap-Dateien als so genannte *Ressourcen* in das Assembly aufzunehmen.

Mit Texturen kann man nicht nur Figuren füllen, sondern auch Schriftzüge, z.B.:

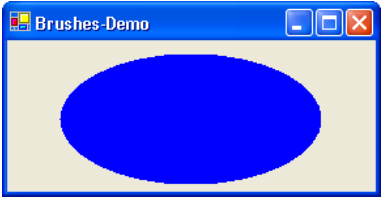
Quellcodefragment	Ausgabe
<pre>BackColor = Color.Red; g.DrawString("Schreiben mit Textur", new Font("Arial", 20, FontStyle.Bold), new TextureBrush(new Bitmap("water.bmp")), 3, 40);</pre>	

- **LinearGradientBrush** (Namensraum **System.Drawing.Drawing2D**)

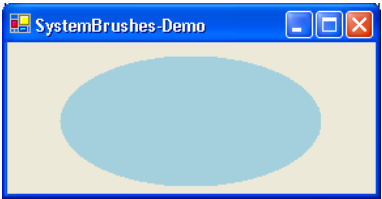
Diese **Brush**-Ableitung erzeugt Füllungen mit Farbverlauf, z.B.:

Quellcodefragment	Ausgabe
<pre>g.FillEllipse(new LinearGradientBrush(new Point(0, 0), new Point(width, height), Color.White, Color.Red), ra);</pre>	

Die Klasse **Brushes** (Namensraum **System.Drawing**) bietet über statische Eigenschaften für jede *Standardfarbe* einen zugehörigen Pinsel, z.B.:

Quellcodefragment	Ausgabe
<pre>g.FillEllipse(Brushes.Blue, ra);</pre>	

Die Klasse **SystemBrushes** (Namensraum **System.Drawing**) bietet über statische Eigenschaften für jede *Windows-Systemfarbe* einen zugehörigen Pinsel, z.B. den Pinsel in der Desktop-Farbe:

Quellcodefragment	Ausgabe
<pre>g.FillEllipse(SystemBrushes.Desktop, ra);</pre>	

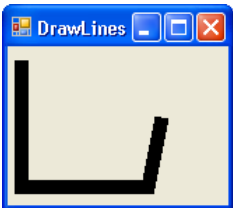
15.6 Linien, Kurven und Flächen

15.6.1 Einfache Linien und Figuren

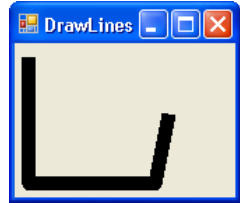
Graphics-Objekte beherrschen zahlreiche Methoden zum Zeichnen von einfachen Linien und Figuren:

- **DrawLine()** und **DrawLines()**

Zum Zeichnen einer Linie zwischen Start- und Endpunkt mit einer der **DrawLine()**-Überladungen, die allesamt ein **Pen**-Objekt als ersten Parameter erwarten, ist wenig zu berichten. Interessanter ist schon die Methode **DrawLines()** zum Zeichnen einer *Serie* von verbundenen Liniensegmenten. Im Vergleich zu mehreren **DrawLine()**-Aufrufen spart man Aufwand, weil die Verbindungspunkte nur einfach angegeben werden müssen. Außerdem sorgt das Grafiksystem für „ansatzfreie“ Übergänge, z.B.:

Quellcodefragment	Ausgabe
<pre>Pen stift = new Pen(Color.Black, 10); Point[] pts = {new Point(10, 10), new Point(10, 100), new Point(100, 100), new Point(110, 50)}; g.DrawLines(stift, pts);</pre>	

Über die **Pen**-Eigenschaft **LineJoin** lassen sich u.a. abgerundete Linienübergänge erzielen:

Quellcodefragment	Ausgabe
<pre>Pen stift = new Pen(Color.Black, 10); Point[] pts = {new Point(10, 10), new Point(10, 100), new Point(100, 100), new Point(110, 50)}; stift.LineJoin = LineJoin.Round; g.DrawLines(stift, pts);</pre>	

- **DrawRectangle()**

Weil die Ausgabe eines Rechtecks grundsätzlich unspektakulär ist, konzentrieren wir uns in diesem Zusammenhang auf winzige Details, die aber doch stören können. Das folgende Codefragment zeichnet zwei Quadrate an der Position (5, 5) mit der angeforderten Kantenlänge 5, wobei verschiedene Linienstärken verwendet werden:

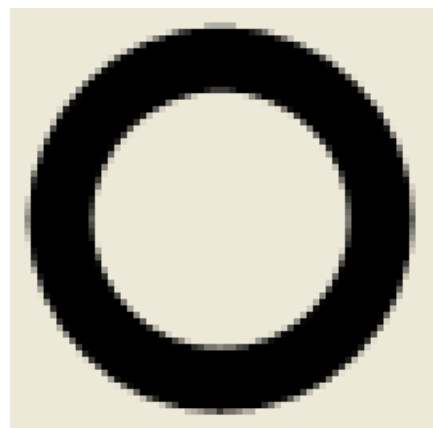
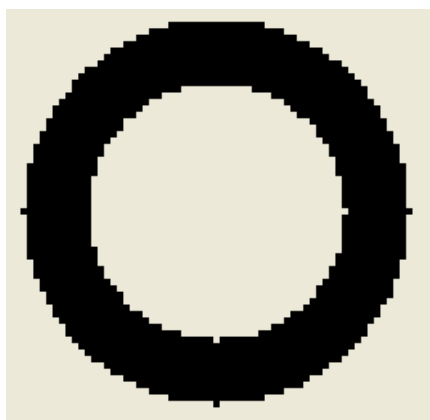
Quellcodefragment	Ausgabe
<pre>Pen stift = new Pen(Color.Black, 3); g.DrawRectangle(stift, 5, 5, 5, 5); g.DrawRectangle(Pens.Red, 5, 5, 5, 5);</pre>	

Der stark vergrößerte Ausschnitt (mit nachträglich in *Paint* eingezeichneten rot-weißen Linien) zeigt:

- o Bei einer Linienstärke von einem Pixel bildet wie erwartet der Pixel (5,5) die linke obere Ecke (siehe rotes Quadrat).
- o Die Breite des roten Quadrates beträgt aber **6** Pixel! Petzold (2002, S. 176) spricht hier vom *1-Pixel-Fehler*.
- o Die 3 Pixel breite schwarze Umrisslinie wird mittig gezeichnet (vgl. Abschnitt 15.5.2), so dass beim schwarzen Quadrat eine Gesamtbreite von 8 Pixeln resultiert.

- **DrawEllipse()**

Im Zusammenhang mit dem Zeichnen einer Ellipse per **DrawEllipse()** soll die Kantenglättung (das so genannte Antialiasing¹) demonstriert werden. Wie ein Vergleich der folgenden Kreise zeigt, kann sich das Einschalten der Kantenglättung über die **Graphics**-Eigenschaft **SmoothingMode** durchaus lohnen:



Der rechte Kreis entstand über folgende Syntax:

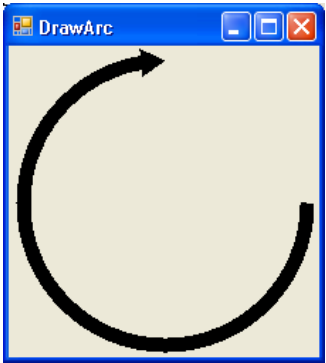
¹ Vermutlich stammt der Begriff *Antialiasing* aus der Signalverarbeitung, wo eine Signalverfälschung durch eine zu geringe Abtastfrequenz als *Alias-Effekt* bezeichnet wird. Analog kann z.B. eine Treppe nur als Ersatz (Alias) für eine schräge Gerade gelten. Versuche, die Abweichung vom Original zu mildern, bezeichnet man als *Antialiasing*.

```
g.SmoothingMode = SmoothingMode.HighQuality;
Pen stift = new Pen(Color.Black, 10);
g.DrawEllipse(stift, 15, 15, 50, 50);
```

Gehört ein Pixel nur teilweise zum Kreis, erhält er einen passend abgeschwächten Farbton. Ganz ohne Geschwindigkeitseinbuße sind derartige Verschönerungen natürlich nicht zu haben.

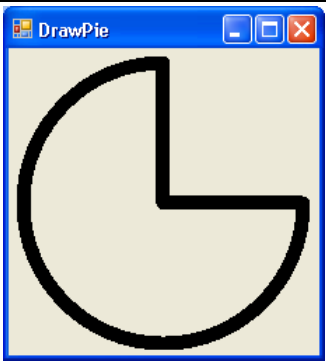
- **DrawArc()**

Im folgenden Beispiel wird einem Quadrat mit 200 Pixeln Kantenlänge ein Bogen von 0° bis 270° (jeweils ab X-Achse im Uhrzeigersinn gemessen) eingezeichnet:

Quellcodefragment	Ausgabe
<pre>Pen stift = new Pen(Color.Black, 5); stift.EndCap = LineCap.ArrowAnchor; g.DrawArc(stift, 10, 10, 200, 200, 0, 270);</pre>	

- **DrawPie()**

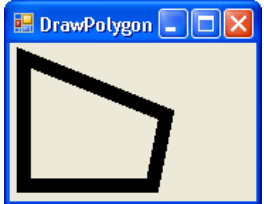
Im folgenden Beispiel wird einem Quadrat mit 200 Pixeln Kantenlänge ein Kreissegment von 0° bis 270° (jeweils ab X-Achse im Uhrzeigersinn gemessen) eingezeichnet:

Quellcodefragment	Ausgabe
<pre>Pen stift = new Pen(Color.Black, 10); stift.LineJoin = LineJoin.Round; g.DrawPie(stift, 10, 10, 200, 200, 0, 270);</pre>	

Mit der **Pen**-Eigenschaft **LineJoin** wird für glatte Übergänge zwischen dem Kreisbogen und den beiden Radien gesorgt.

- **DrawPolygon()**

Im Unterschied zur verwandten Methode **DrawLines()** liefert **DrawPolygon()** eine geschlossene Figur, z.B.:

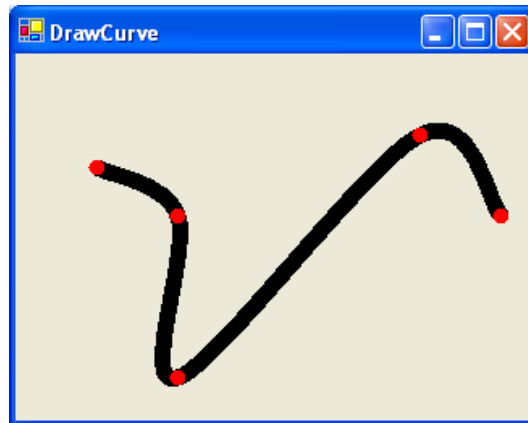
Quellcodefragment	Ausgabe
<pre>Pen stift = new Pen(Color.Black, 10); Point[] pts = {new Point(10, 10), new Point(10, 100), new Point(100, 100), new Point(110, 50)}; g.DrawPolygon(stift, pts);</pre>	

15.6.2 Splines

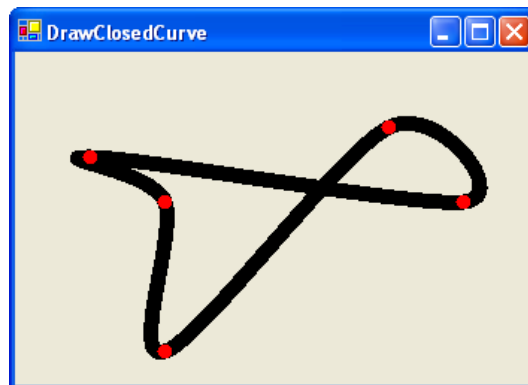
Bei den mit **DrawCurve()** erzeugten *kardinalen* (kanonischen, traditionellen) Splines wird durch vorgegebene Punkte eine geglättete Linie gelegt, z.B.:

```
Pen stift = new Pen(Color.Black, 10);
Point[] pts = {new Point(50, 70),
               new Point(100, 100),
               new Point(100, 200),
               new Point(250, 50),
               new Point(300, 100)};
g.DrawCurve(stift, pts);
```

Im Quellcodesegment fehlen die Anweisungen zum Markieren der Stützstellen:



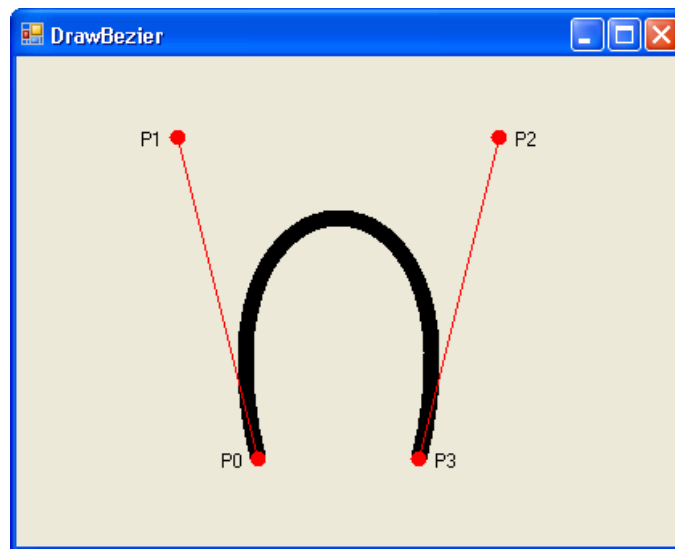
Mit **DrawClosedCurve()** erhält man einen *geschlossenen* kardinalen Spline, z.B.:



Bei den mit **DrawBezier()** zu erzeugenden *Bézier-Splines* sind genau 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.:

```
Pen stift = new Pen(Color.Black, 10);
Point[] pts = {new Point(150, 250),
               new Point(100, 50),
               new Point(300, 50),
               new Point(250, 250)};
g.DrawBezier(stift, pts[0], pts[1], pts[2], pts[3]);
```

Im Quellcodesegment fehlen die Anweisungen zum Zeichnen der Punkte und Hilfslinien:



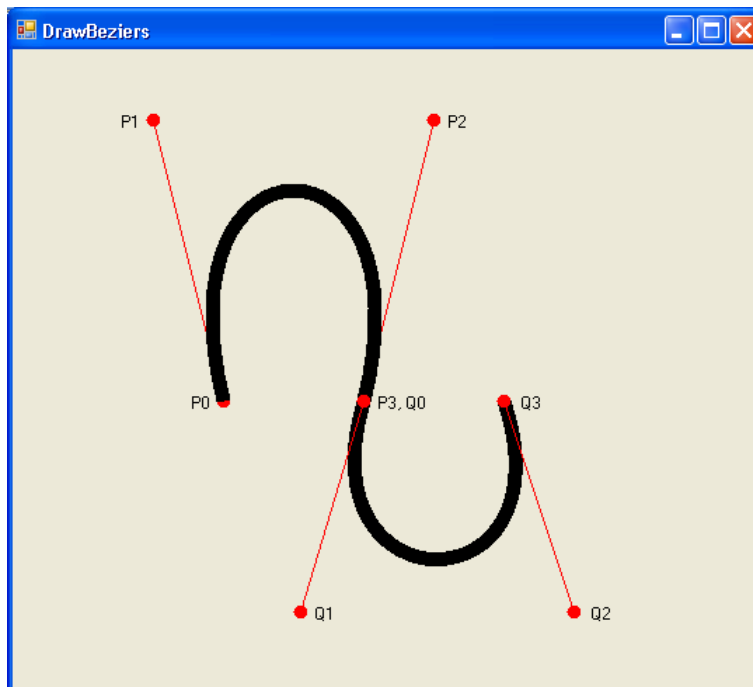
Wenn bei einer mit **DrawBeziers()** gezeichneten Sequenz von mehreren Bézier-Splines die Übergänge glatt verlaufen sollen, dann muss gelten:

- Jeder Spline endet dort, wo sein Nachfolger beginnt.
- Folgende Punkte liegen jeweils auf einer Geraden:
 - zweiter Kontrollpunkt der ersten Kurve
 - Grenzpunkt
 - erster Kontrollpunkt der zweiten Kurve

Im Quellcodesegment

```
Pen stift = new Pen(Color.Black, 10);
Point p0 = new Point(150, 250); Point p1 = new Point(100, 50);
Point p2 = new Point(300, 50); Point p3 = new Point(250, 250);
Point q1 = new Point(205, 400);
Point q2 = new Point(400, 400); Point q3 = new Point(350, 250);
Point[] pts = {p0, p1, p2, p3, q1, q2, q3};
g.DrawBeziers(stift, pts);
```

zum folgenden **DrawBeziers()**-Ergebnis:

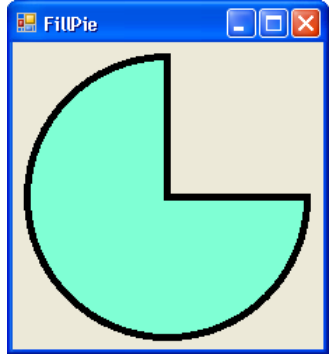


fehlen die Anweisungen zum Zeichnen der Punkte und Hilfslinien.

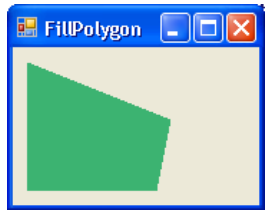
15.6.3 Flächen

Zum Zeichnen von Flächen (gefüllten Formen) bietet das Grafiksystem folgende Methoden, wobei ein Pinsel (**Brush**-Objekt, siehe Abschnitt 15.5.3) für Gestaltungsmöglichkeiten sorgt:

- **FillRectangle()**
Mit dieser Methode erstellt man gefüllte Rechtecke, wobei es auch möglich ist, exakt *ein* Pixel einzufärben (vgl. Abschnitt 15.6.1 zum 1-Pixel-Fehler bei **DrawRectangle()**), z.B.
`g.FillRectangle(Brushes.Red, 5, 5, 1, 1);`
- **FillEllipse()**
Mit dieser Methode erstellt man gefüllte Ellipsen, z.B.:
`g.FillEllipse(Brushes.Blue, 10, 10, 30, 25);`
Wir haben sie schon im ersten Beispiel von Abschnitt 15 dazu verwendet, um gefüllte Kreise zu malen.
- **FillPie()**
Im folgenden Beispiel wird einem Quadrat mit 200 Pixeln Kantenlänge ein gefülltes Kreis-segment von 0° bis 270° (jeweils ab X-Achse im Uhrzeigersinn gemessen) eingezeichnet:

Quellcodefragment	Ausgabe
<pre>Pen pen = new Pen(Color.Black, 5); g.FillPie(Brushes.Aquamarine, 10, 10, 200, 200, 0, 270); g.DrawPie(pen, 10, 10, 200, 200, 0, 270);</pre>	

- **FillPolygon()**
Mit dieser Methode erstellt man gefüllte Polygone, z.B.:

Quellcodefragment	Ausgabe
<pre>Point[] pts = {new Point(10, 10), new Point(10, 100), new Point(100, 100), new Point(110, 50)}; g.FillPolygon(Brushes.MediumSeaGreen, pts);</pre>	

Beim Füllen von Polygonen und geschlossenen Standard - Splines (mit **FillClosedCurve()**) wird es interessant bei sich kreuzenden Linien (bzw. Kurven), weil man dabei unterschiedliche Begriffe vom *Inneren* der Figur haben kann. Im folgenden Beispiel (gefunden bei Petzold 2002, S. 178) wird ein fünfzackiger Stern (mit insgesamt zehn Ecken) geschickt über ein Polygon mit lediglich fünf Punkten gezeichnet:

```
using System;
using System.Windows.Forms;
using System.Drawing;
using System.Drawing.Drawing2D;

class Stern : Form {
    Point[] pts = new Point[5];
```

```

Stern() {
    Text = "Stern";
    ClientSize = new Size(350, 350);

    // Stern-Eckpunkte berechnen
    int shift = 180, len = 150;
    double winkel = 3 * Math.PI / 2;
    for (int i = 0; i < pts.Length; i++) {
        double kosinus = Math.Cos(winkel);
        double sinus = Math.Sin(winkel);
        int x = (int) (shift + kosinus * len);
        int y = (int) (shift + sinus * len);
        pts[i] = new Point(x, y);
        winkel += 4 * Math.PI / 5;
    }
}

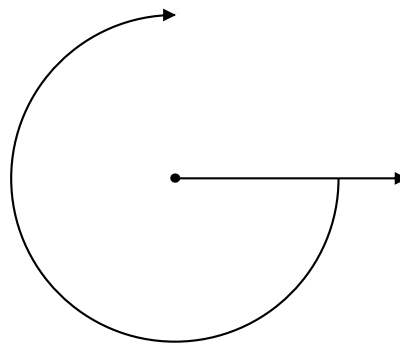
protected override void OnPaint(PaintEventArgs e) {
    for (int i = 0; i < pts.Length; i++) {
        e.Graphics.DrawString(i.ToString(), Font, SystemBrushes.WindowText,
                               pts[i].X - 5, pts[i].Y - 5);
    }
    e.Graphics.FillPolygon(Brushes.MediumSeaGreen, pts, FillMode.Alternate);
}

[STAThread]
static void Main() {
    Application.Run(new Stern());
}
}

```

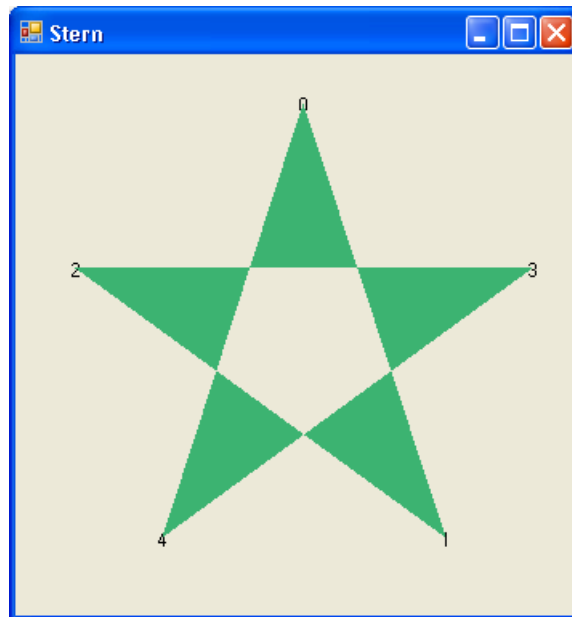
Zur Berechnung der Koordinaten ist ein wenig Trigonometrie erforderlich:

- Die Koordinaten Eckpunkte des Sterns werden über die Kosinus- bzw. Sinus-Funktion des aktuellen Winkels ermittelt, wobei zusätzlich eine Streckung und schließlich noch eine Verschiebung erfolgt.
- Weil das GDI+ Winkel im Uhrzeigersinn ab der positiven X-Achse interpretiert, wird der Winkel initial auf $\frac{3}{2}\pi$ ($= 270^\circ$) gesetzt, damit der Stern „aufrecht“ steht.



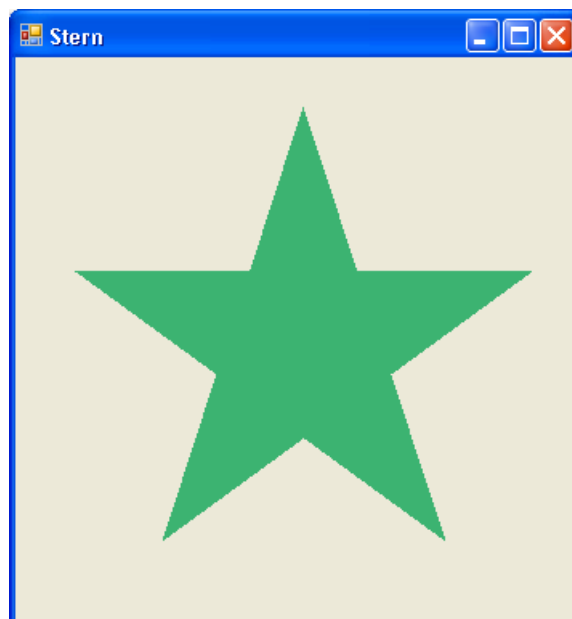
- Beim fünfmaligen Durchlaufen der **for**-Schleife mit einer Winkelaktualisierung um $\frac{4}{5}\pi$ werden die Eckpunkte auf den Kreisbogen gesetzt.

Beim voreingestellten Füllmodus **Alternate** (*abwechselnd*) bleibt ein fünfeckiger Bereich ungefüllt:



Zur Klärung der Konstruktion wurden hier zusätzlich die Nummern der Eckpunkte per **DrawString()** (siehe unten) ausgegeben.

Der Füllmodus **Winding** färbt in der Regel alle umgrenzten Flächen:



15.7 Text malen

Zur Ausgabe einer individuell gestalteten Beschriftung erzeugt man zunächst ein Objekt der Klasse **System.Drawing.Font**, z.B.:

```
Font font = new Font("Arial", 12, FontStyle.Bold);
```

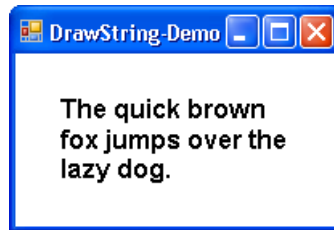
Position und Ausdehnung einer Beschriftung lassen sich z.B. über eine Instanz der Struktur **System.Drawing.Rectangle**-Instanz festlegen:

```
Rectangle rect = new Rectangle(50, 50, 150, 70);
```

Der folgende **DrawString()**-Aufruf an ein **Graphics**-Objekt

```
g.DrawString(text, font, SystemBrushes.WindowText, rect);
```

liefert die Ausgabe:



Als Werkzeug verwendet **DrawString()** keinen Stift, sondern einen *Pinself* (ein **Brush**-Objekt). Text wird im GDI+ also nicht per Stift geschrieben, sondern per Pinsel *gemalt*.

In den nächsten Abschnitten werden wir uns mit den beteiligten Klassen, Eigenschaften und Methoden näher beschäftigen. Der tiefere Sinn des Beispielsatzes liegt übrigens darin, dass er alle Buchstaben des englischen Alphabets enthält und daher im englischen Sprachraum zum Testen von Textein- und Ausgabeverfahren geeignet ist.

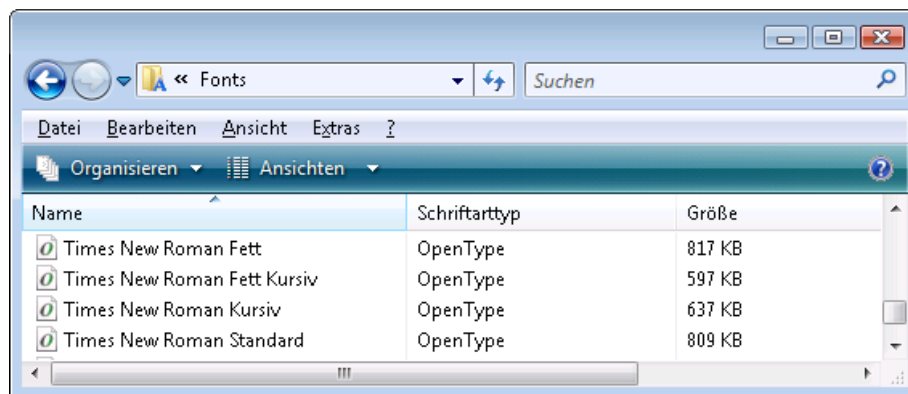
15.7.1 Schriftarten und -familien

Von den unter Windows verfügbaren Schrifttechnologien werden in WinForms-Anwendungen „nur“ **TrueType** und **OpenType** unterstützt. Im Vergleich zu diesen modernen Konturschrift-Technologien sind die fehlenden Bitmap- und Vektortechniken sehr unattraktiv, so dass wohl kaum jemand diese Altlasten aus den Windows-Gründerzeiten vermisst.

Seit Windows XP dominieren die 1997 von Adobe und Microsoft gemeinsam vorgestellten **OpenType**-Schriften. Es handelt sich im Wesentlichen um eine Kombination bzw. Integration älterer Konturschrift-Technologien:

- **TrueType** (gemeinsam von Apple und Microsoft für MacOS und Windows entwickelt)
- **Type 1** – Format der von Adobe entwickelten Seitenbeschreibungssprache **Postscript**

Zu einem Schriftdesign existieren in der Regel mehrere **Schnitte** (Regulär, *Kursiv*, **Fett**, **Fett Kursiv**). Jeder Schnitt stellt eine eigene Schriftart dar und erscheint folglich im Fenster des zuständigen Applets der Windows-Systemsteuerung als eigener Eintrag, z.B.:



Die folgenden Schriftarten (und noch viele andere) sind bei praktisch jeder Windows-Installation vorhanden:

- Times New Roman, **Times New Roman Fett**, *Times New Roman Kursiv*, **Times New Roman Fett Kursiv**
- Arial, **Arial Fett**, *Arial Kursiv*, **Arial Fett Kursiv**
- Courier New, **Courier New Fett**, *Courier New Kursiv*, **Courier New Fett Kursiv**
- Symbol (z.B. Σ, ψ, μ)
- WebDings (z.B. ♀, ♀, ♀)
- WingDings (z.B. ☺, ☹, ☹)

Wie Sie bereits aus mehrfacher Erfahrung wissen, ist im .NET – Framework ein Objekt der Klasse **Font** gekennzeichnet durch:

- **ein Schriftdesign**

In den verschiedenen Überladungen des **Font**-Konstruktors kann das Schriftdesign per **String**-Objekt oder über ein **FontFamily**-Objekt (siehe unten) angegeben werden. Die Konstruktion:

```
Font font = new Font("Arial", 12, FontStyle.Bold)
```

ist letztlich eine Abkürzung für:

```
FontFamily ffar = new FontFamily("Arial");
Font font = new Font(ffar, 12, FontStyle.Bold);
```

- **eine Schriftgröße**

Die so genannte **Gewiertgröße** einer Schriftart wird in der Einheit *Punkt* angegeben:

$$1 \text{ Punkt} = \frac{1}{72} \text{ Zoll}, 1 \text{ Zoll} = 2,54 \text{ cm}$$

- **einen Schriftschnitt bzw. -stil**

Den Schriftschnitt bzw. stil wählt man über die Werte der Enumeration **FontStyle**:

Bold	Fett
Italic	Kursiv
Regular	Normal
Strikeout	Durchgestrichen
Underline	Unterstrichen

Die einzelnen Werte sind über den bitweisen Oder-Operator kombinierbar, z.B.:

```
FontStyle style = FontStyle.Italic | FontStyle.Bold;
```

Ein **Font**-Objekt kann nicht geändert werden, und seine Eigenschaften (z.B. **FontStyle**) bieten dementsprechend nur einen lesenden Zugriff. Wird zu einem vorhandenen **Font**-Objekt ein alternativer Schnitt benötigt, so muss ein neues **Font**-Objekt erzeugt werden, wobei die Konstruktor-Überladung mit **Font**-Parameter Schreibarbeit spart, z.B.:

```
Font font = new Font("Arial", 12, FontStyle.Bold);
```

```
font = new Font(font, FontStyle.Italic);
```

Um die offensichtliche Zusammengehörigkeit der Schriftarten mit identischem Design abzubilden, kennt das .NET-Framework die Klasse **FontFamily**. Zur Objekt-Konstruktion kann man u.a. einen Familiennamen angeben, z.B.:

```
FontFamily ffar = new FontFamily("Arial");
```

Bei einem **FontFamily**-Objekt kann man per **IsStyleAvailable()**-Methode anfragen, ob ein bestimmter Schriftschnitt vorhanden ist, so dass sich Ausnahmefehler durch fehlende Schnitte vermeiden lassen.

Über die statische **FontFamily**-Eigenschaft **Families** lässt sich ein Array mit allen **FontFamily**-Objekten ansprechen, die dem aktuellen **Graphics**-Objekt zugeordnet sind, z.B.:

```
FontFamily[] ff = FontFamily.Families;
```

Im folgenden Programm wird aus jeder Familie im **FontFamily**-Array **ff** ein **Font**-Objekt mit Größe 16 und Schriftstil (**FontStyle.Italic** | **FontStyle.Bold**) (falls vorhanden) in den **Font**-Array **fonts** übernommen (siehe Formularkonstruktor):

```
using System.Windows.Forms;
using System.Drawing;
using System;

class FontFamilyDemo : Form {
    Font[] fonts;
    int nFonts;
```

```

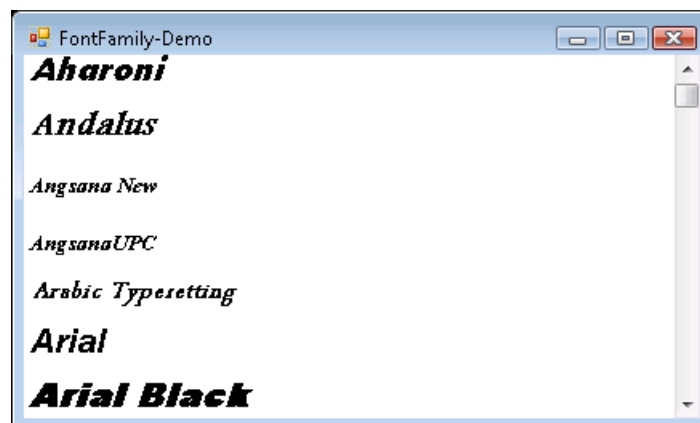
FontFamilyDemo() {
    Text = "FontFamily-Demo";
    BackColor = SystemColors.Window;
    ClientSize = new Size(400, 300);
    AutoScroll = true;
    FontFamily[] ff = FontFamily.Families;
    fonts = new Font[ff.Length];
    FontStyle style = FontStyle.Italic | FontStyle.Bold;
    int size = 16;
    int vs = 0, i = 0;
    foreach (FontFamily aff in ff)
        if (aff.IsStyleAvailable(style)) {
            fonts[i] = new Font(aff, size, style);
            vs += fonts[i].Height + 5;
            i++;
        }
    nFonts = i;
    Panel pan = new Panel();
    pan.Height = vs;
    pan.Dock = DockStyle.Top;
    pan.Paint += new PaintEventHandler(PanelOnPaint);
    pan.Parent = this;
}

void PanelOnPaint(object sender, PaintEventArgs e) {
    int ok = 0;
    for (int i = 0; i < nFonts; i++) {
        e.Graphics.DrawString(fonts[i].Name, fonts[i],
                               SystemBrushes.WindowText, 0, ok);
        ok += fonts[i].Height + 5;
    }
}

[STAThread]
static void Main() {
    Application.Run(new FontFamilyDemo());
}
}

```

Zweck des Programms ist die Anzeige einer rollbaren Liste mit allen in einem bestimmten Stil verfügbaren Schriftarten:



Zu jeder Schriftart soll in einer eigenen Zeile ihr Name (verfügbar in der **Font**-Eigenschaft **Name**) per **DrawString()**-Aufruf mit dem passenden **Font**-Objekt geschrieben werden:

```
e.Graphics.DrawString(fonts[i].Name, fonts[i], SystemBrushes.WindowText, 0, ok);
```

Weil mit einer längeren Liste zu rechnen ist, benötigt das Programm eine vertikale **Bildlaufleiste**, die auf folgende Weise mit geringem Aufwand realisiert wird:

- Auf dem Formular wird ein Steuerelement der Klasse **Panel** untergebracht, das als Zeichenfläche für die **DrawString()**-Aufrufe dient. Sein **Paint**-Ereignis wird in der Methode **PanelOnPaint()** behandelt (siehe oben).
- Besitzt die **AutoScroll**-Eigenschaft eines **ScrollableControl**-Objekts (z.B. eines Formulars) den Wert **true**, dann ergänzt das Laufzeitsystem automatisch Bildlaufleisten, wenn die untergeordneten Steuerelemente nicht vollständig in den Klientenbereich passen.
- Um die vertikale Ausdehnung der Liste mit allen Schriftzügen zu berechnen, werden die **Height**-Eigenschaftswerte der beteiligten **Font**-Objekte plus jeweils 5 Pixel Abstand aufaddiert (siehe Formularkonstruktor).

Sind zahlreiche Schriftarten vorhanden, zeigt die Rollbalkenlösung ein zähes Verhalten. Bei der systematischen Beschäftigung mit dem Thema *Rollbalken* werden wir eine deutlich performantere Lösung entwickeln (siehe Abschnitt 16.4).

Font-Objekte belegen GDI-Ressourcen (vgl. Abschnitt 15.2), so dass für selbst erzeugte und später obsolet gewordene Exemplare die **Dispose()**-Methode aufgerufen werden sollte.

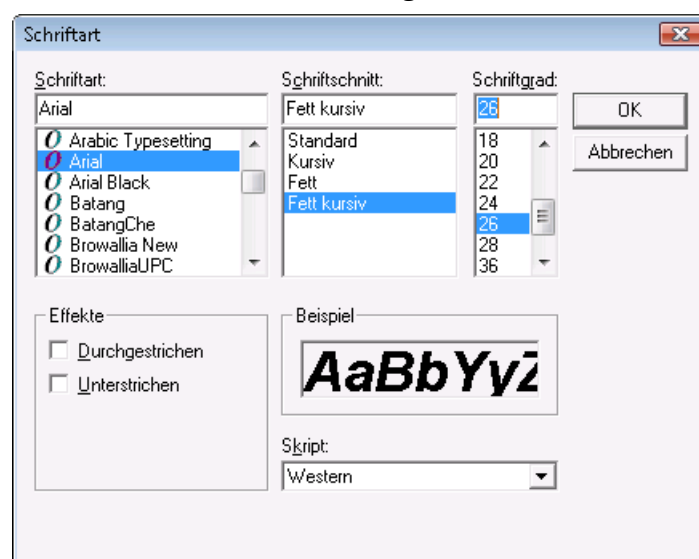
Über schreibgeschützte statische Eigenschaften der Klasse **System.Drawing.SystemFonts** sind vordefinierte Schriften im aktuellen Windows-Design verfügbar, z.B.:

Eigenschaft	Verwendung
CaptionFont	Text auf den Titelleisten von Fenstern
DefaultFont	Standardschriftart für Dialogfelder und Formulare
MenuFont	Menüs
StatusFont	Text auf der Statusleiste

Bei ihrer Verwendung braucht man sich um GDI-Ressourcen und um Konsistenz mit anderen Teilen der Benutzeroberfläche keine Gedanken zu machen.

15.7.2 Standarddialog zur Schriftauswahl

Analog zum Einsatz des Farbauswahl-Standarddialogs (siehe Abschnitt 15.5.1) kann man dem Benutzer über ein Objekt der Klasse **FontDialog** (aus dem Namensraum **System.Windows.Forms**) Gelegenheit bieten, auf vertraute Weise seine Lieblingsschrift zu wählen:



Es bedarf keiner großen Anstrengung, diesen Standarddialog zur Schriftauswahl zu präsentieren und sein Ergebnis auszuwerten, wie die Behandlungsmethode zum **Click**-Ereignis des Befehlschalters im Beispielprogramm



zeigt:

```
protected void ButtonOnClick(object sender, EventArgs e) {
    if (fd.ShowDialog() == DialogResult.OK) {
        Font oldFont = lbProbe.Font;
        lbProbe.Font = fd.Font;
        lbProbe.Text = fd.Font.Name;
        oldFont.Dispose();
    }
}
```

Nach dem Einsatz seiner **ShowDialog()**-Methode informiert das **FontDialog**-Objekt per Rückgabewert über die vom Benutzer verwendete Terminierungsmethode (**OK** oder **Cancel**) sowie ggf. per **Font**-Eigenschaft über die gewählte Schriftart. Zur Schonung der GDI-Ressourcen (vgl. Abschnitt 15.2) wird für das obsolekte **Font**-Objekt die **Dispose()**-Methode aufgerufen.

Wenn man das gelieferte **Font**-Objekt nach seiner (in Punkten zu 1/72 Zoll) gemessenen Größe befragt,

```
MessageBox.Show(fd.Font.SizeInPoints.ToString());
```

stellt man aber Abweichungen zum **Schriftgrad** fest, den der Benutzer im Standarddialog zu einer beliebigen Schriftart (Open Type oder True Type) gewählt hat, z.B.¹

Vom Benutzer gewählter Schriftgrad	SizeInPoints des gelieferten Font -Objekts
10	9,75
12	12
14	14,25

Diese Ergebnisse entstanden bei einer angenommenen Bildschirmauflösung von 96 DPI unter Windows XP (SP 3) oder Windows Vista Business (SP 2) mit installiertem .NET - Framework 3.5. Bei höherer Auflösung fallen die Abweichungen etwas kleiner aus. Offenbar wird der vom Benutzer gewünschte Schriftgrad (Abk.: t) nach folgender Formel

$$x = \frac{t}{72} 96$$

von der Maßeinheit Punkt unter Berücksichtigung der Bildschirmauflösung in Pixel (Abk.: x) umgerechnet, z.B.:

$$13,3 = \frac{10}{72} 96$$

Jetzt findet bedauerlicherweise eine Rundung statt:

$$13,3 \rightarrow 13$$

Das fehlerhafte Pixel-Zwischenergebnis wird vor dem Erstellen des **Font**-Objekts nach der Formel

¹ Auf dieses Problem hat dankenswerterweise Herr Lei Yu, ein besonders aufmerksamer und engagierter Student, hingewiesen. Zur Aufklärung der Hintergründe hat die befragte MSDN-Hotline wertvolle Beiträge geleistet.

$$t = \frac{x}{96} 72$$

wieder in einen Punktwert umgerechnet, z.B.

$$9,75 = \frac{13}{96} 72$$

Weil der vom Benutzer gewählte Schriftgrad bei der ersten Transformation durch 3 dividiert wird,

$$\frac{96}{72} = \frac{4}{3}$$

tritt kein Fehler auf, wenn der Schriftgrad ein Vielfaches von 3 ist. Ferner ergibt sich, dass der vom **FontDialog**-Objekt gelieferte **SizeInPoints**-Wert nur einen Fehler von $\pm 0,25$ enthalten kann, so dass eine erneute Rundung den vom Benutzer gewünschten Schriftgrad liefert:

```
lbProbe.Font = new Font(fd.Font.FontFamily,
    (float) Math.Round(fd.Font.SizeInPoints), fd.Font.Style);
```

Zunächst ist die FCL - Klasse **FontDialog** für das Problem verantwortlich. Diese verlässt sich aber wohl auf das Betriebssystem (also auf das GDI+), und der Fehler dürfte auch bei Verwendung anderer Windows-APIs auftreten (z.B. bei MFC-Programmen in C++).

Unter Verwendung der später zu behandelnden Druckfunktionalität unseres Editors (siehe Abschnitt 21) lässt sich (dank Druckvorschau bequem und ohne Papierverschwendung) nachweisen, dass aus dem diskutierten **FontDialog**-Problem Unterschiede bei der Druckausgabe resultieren. In der folgenden Tabelle ist für die Schriftgrade 10 und 14 angegeben, wie viele Zeilen (bei bestimmten, hier irrelevanten Randbreiten) mit bzw. ohne Korrektur des **FontDialog**-Fehlers auf eine Seite passen:

	Vom Benutzer gewählter Schriftgrad	
	10	14
ohne Korrektur	81	58
mit Korrektur	79	59

Aufgrund des Fehlers kann die Zahl der Zeilen pro Seite steigen oder fallen.

Speichert man den Text im RTF-Format in eine Datei werden die **Font**-Abweichungen automatisch wieder durch Runden beseitigt. Im RTF-Format wird die Schriftgröße in Halbpunkten angegeben, und bei beim Speichern resultiert z.B. aus 9,75 oder 10 Punkten jeweils die Größenangabe **fs20**, wie ein Blick auf den Dateianfang bestätigt:

```
{\rtf1\ansi\deff0{\fonttbl{\f0\fnl\fcharset0 Lucida Console;}}
\viewkind4\uc1\pard\lang1031\f0\fs20 1\par
. . .
```

Weitere Informationen zur Klasse **FontDialog** folgen in Abschnitt 18.3 im Kontext mit den übrigen Windows-Standarddialogen (zur Farb- und Dateiauswahl).

Den vollständigen Quellcode des Beispielsprogramms finden Sie im Ordner

... \BspUeb\GDI+\Textausgabe\FontDialog

15.7.3 Zeichen setzen

Die **DrawString()**-Methode zur Ausgabe einer Zeichenfolge unter Verwendung einer bestimmten Schriftart mit einem bestimmten Pinsel an einer bestimmten Stelle auf der Oberfläche eines Steuerelementes kennt gleich sechs Überladungen:

- **public void DrawString(String, Font, Brush, PointF);**
- **public void DrawString(String, Font, Brush, RectangleF);**
- **public void DrawString(String, Font, Brush, PointF, StringFormat);**
- **public void DrawString(String, Font, Brush, RectangleF, StringFormat);**
- **public void DrawString(String, Font, Brush, float, float);**
- **public void DrawString(String, Font, Brush, float, float, StringFormat);**

Sie unterscheiden sich durch die Technik zur Spezifikation der Ausgabestelle (per **PointF**-Objekt, **RectangleF**-Objekt oder über Koordinaten) sowie durch die An- bzw. Abwesenheit eines **StringFormat**-Objekts mit Formatierungsattributen.

Über die verfügbaren Schriftarten haben wir uns bereits ausreichend informiert. Als Pinsel wird man in der Regel ein **SolidBrush**-Objekt verwenden (vgl. Abschnitt 15.5.3), wobei die Wahl der Farbe eine kurze Überlegung wert ist. Sinnvoller als Schwarz kann die vom Benutzer zu beeinflussende Farbe **SystemColors.WindowText** sein. Dann verwendet man am besten gleich den Pinsel **SystemBrushes.WindowText** (siehe Abschnitt 15.5.3) und eine passende Hintergrundfarbe, z.B. **SystemColors.Window**.

Bei der Textpositionierung über ein **PointF**-Objekt oder ein Koordinatenpaar aus zwei **float**-Werten resultiert eine *einzige* Ausgabezeile, wenn die Zeichenfolge keinen Zeilenumbruch enthält. Mit dem Text

```
string text = "The quick brown fox jumps over the lazy dog.";
```

liefert der Aufruf

```
g.DrawString(text, font, SystemBrushes.WindowText, 10, 10);
```

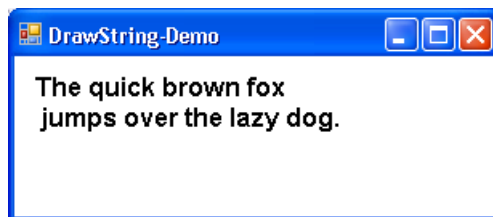
folgendes Ergebnis:



Der Text

```
string text = "The quick brown fox\n jumps over the lazy dog.";
```

wird vom selben Aufruf so ausgegeben:



Definiert man den Ausgabeort per **RectangleF**-Objekt, dann bietet **DrawString()** sogar einen automatischen Zeilenumbruch. Der Text

```
string text = "»Ich habe ja gesagt, dass er Karlsson heißt und oben "  
+ "auf dem Dach wohnt«, sagte Lillebror. »Was ist denn da "  
+ "Komisches dran? Die Leute dürfen doch wohl wohnen, "  
+ "wo sie wollen!«";
```

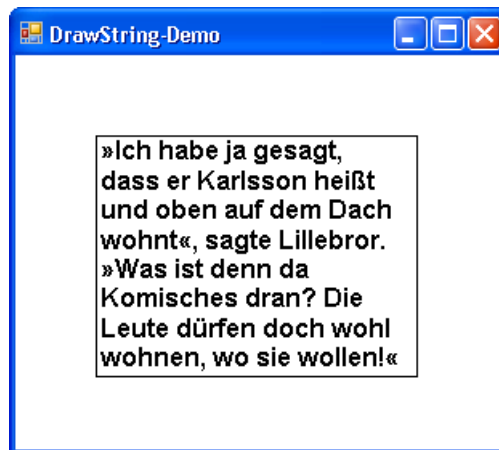
wird vom **DrawString()**-Aufruf

```
g.DrawString(text, font, SystemBrushes.WindowText, rect);
```

unter Verwendung des Rechtecks

```
Rectangle rect = new Rectangle(50, 50, 200, 150);
```

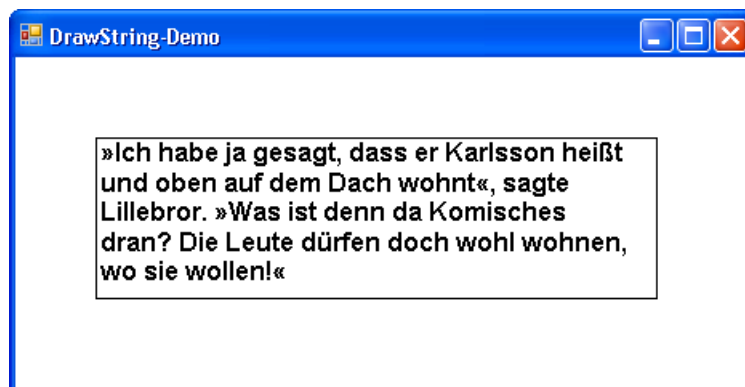
so umgebrochen:



Mit dem Rechteck

```
Rectangle rect = new Rectangle(50, 50, 350, 100);
```

erhält man:



Mit den Eigenschaften eines **StringFormat**-Objekt lässt sich das Verhalten eines **DrawString()**-Aufrufs auf vielfältige Weise modifizieren. Das folgende Exemplar

```
StringFormat sf = new StringFormat();
```

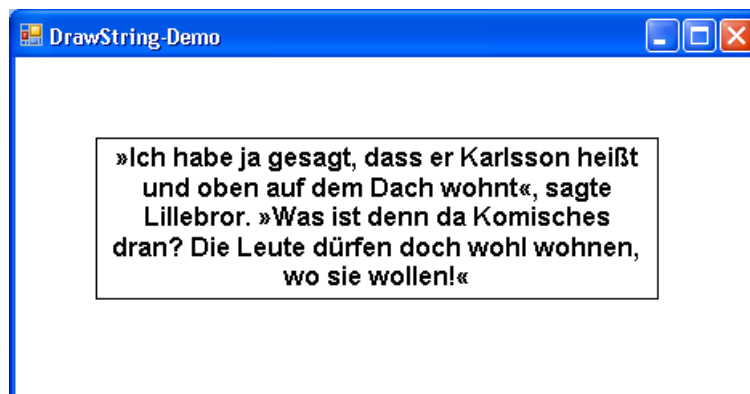
soll für eine horizontal und vertikal zentrierte Textausgabe sorgen:

```
sf.Alignment = StringAlignment.Center;  
sf.LineAlignment = StringAlignment.Center;
```

Von der **StringFormat**-Eigenschaft **Alignment** hängt die *horizontale* Textausrichtung innerhalb einer Zeile ab. Die **StringFormat**-Eigenschaft **LineAlignment** entscheidet über die *vertikale* Ausrichtung des gesamten Textblocks innerhalb eines begrenzenden Rechtecks. Der Aufruf

```
g.DrawString(text, font, SystemBrushes.WindowText, rect, sf);
```

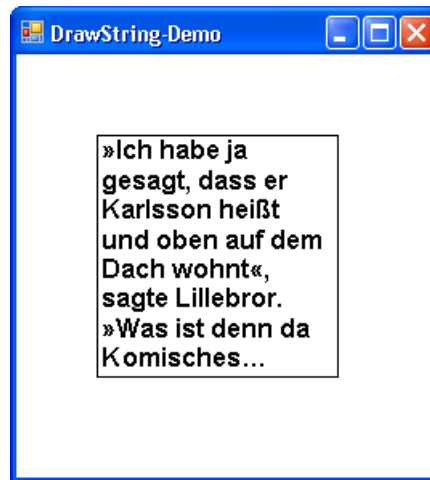
liefert mit dem letzten Textbeispiel folgende Ausgabe:



Über die **StringFormat**-Eigenschaft **Trimming** legt man fest, wie ein Text gekürzt werden soll, wenn er nicht vollständig in das Ausgaberechteck passt. Mit der folgenden Anweisung

```
sf.Trimming = StringTrimming.EllipsisWord;
```

sorgt man dafür, dass der Text gekürzt und hinter dem letzten vollständig passenden Wort ein Auslassungszeichen (...) eingefügt wird:



15.7.4 Maß halten

Bei der Textausgabe werden gelegentlich genaue Angaben zum horizontalen und vertikalen Platzbedarf einer bestimmten Zeichenfolge bei Verwendung einer bestimmten Schriftart benötigt.

15.7.4.1 *Schrifthöhe und Zeilenabstand*

In diesem Abschnitt geht es um den erforderlichen vertikalen Pixel-Abstand bei der Bildschirmausgabe von aufeinander folgender Textzeilen, wobei die Größe der beteiligten Schriftarten in der Einheit *Punkt* (= 1/72 Zoll) angegeben wird. Zwar kann **DrawString()** Text mit automatischem Zeilenumbruch und automatischer Zeilenabstandsrechnung in ein Rechteck schreiben, doch oft erzwingt z.B. eine tabellarische Layout-Vorgabe oder ein Schriftartenwechsel die Verwendung mehrerer **DrawString()**-Aufrufe mit zugehörigen Positionsberechnungen.

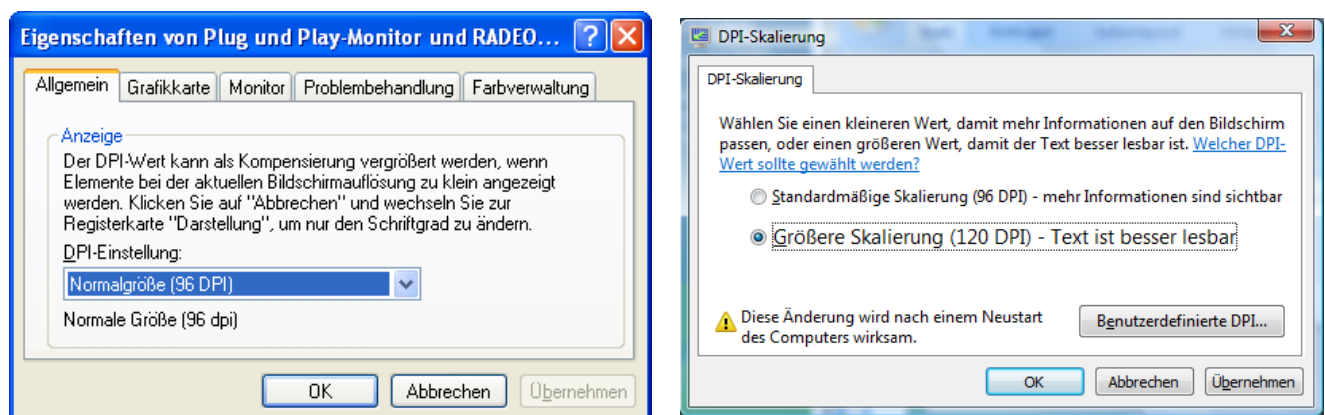
Bei den folgenden Überlegungen ist auch die Annahme des Betriebssystems zur Bildschirmauflösung (gemessen in *Dots Per Inch*, *DPI*) relevant, die vom Benutzer unter Windows XP nach

Systemsteuerung > Anzeige > Einstellungen > Erweitert

bzw. unter Windows Vista nach

Systemsteuerung > Anpassung > Schriftgrad anpassen (DPI)

beeinflusst werden kann:

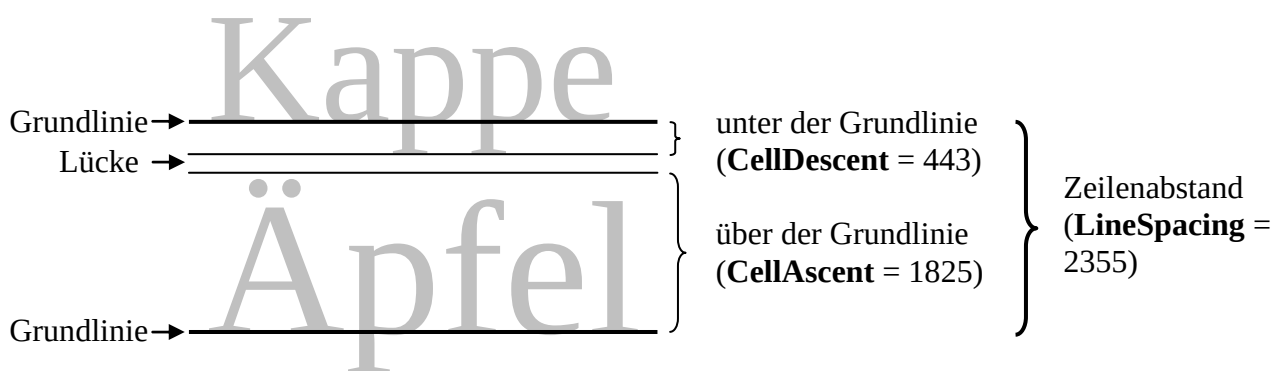


Die aktive Ein- (bzw. Unter-) stellung lässt sich über die **DpiY**-Eigenschaft eines zur Bildschirm- ausgabe befähigten **Graphics**-Objektes ermitteln (siehe unten). In Abschnitt 15.8.1 über metrische Maßeinheiten (an Stelle von Pixeln) werden wir uns mit der potentiellen Ungenauigkeit dieser Unterstellung beschäftigen.

Über die Instanzmethoden

- **GetEmHeight**(FontStyle)
- **GetCellAscent**(FontStyle)
- **GetCellDescent**(FontStyle)
- **GetLineSpacing**(FontStyle)

der Klasse **FontFamily** erhält man zum angegebenen Schnitt einer Schriftfamilie vier Höhenangaben in schriftgradunabhängigen Entwurfseinheiten, z.B. bei **Times New Roman** mit **FontStyle.Regular**:

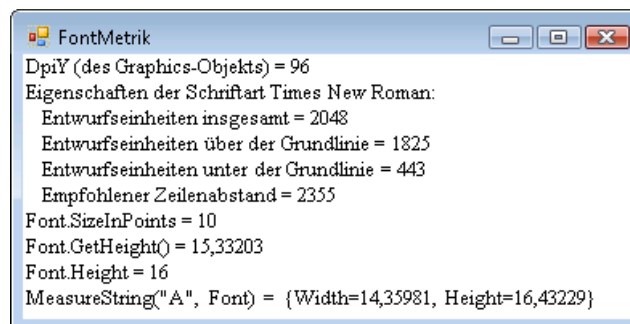


In der Abbildung ist die **EmHeight** des Schriftschnitts von 2048, die als Referenzgröße für die anderen Angaben fungiert, *nicht* darstellbar.

Aus dem Quotienten aus **LineSpacing** und **EmHeight**, dem Schriftgrad in Punkten sowie der vertikalen Auflösung des Ausgabegerätes lässt sich der empfohlene Zeilenabstand in Pixeln berechnen. Beim Schriftgrad 10 (Zeichenhöhe 10/72 Zoll) und einem Bildschirm mit 96 Pixeln pro Zoll ergibt sich als empfohlener Zeilenabstand:

$$\frac{2355}{2048} \frac{10}{72} 96 = 15,33203$$

Über die **Font**-Methode **GetHeight()** wird bei den beschriebenen Voraussetzungen genau dieser Zeilenabstand ermittelt, der als Empfehlung der Schriftdesigner gelten kann:



Die **Font**-Eigenschaft **Height** liefert eine korrespondierende, auf die nächst größere Ganzzahl gerundete und ausschließlich für den Bildschirm geeignete Zeilenabstandsempfehlung.

Von der **Graphics**-Methode **MeasureString()** wird (als **Height**-Eigenschaft des **SizeF**-Rückgabewerts) eine Angabe zur maximalen vertikalen Ausdehnung einer Schrift beigesteuert, die oft über dem empfohlenen Zeilenabstand liegt und nicht genutzt werden sollte (im Beispiel: 16,43).

Im Projektordner

... \BspUeb\GDI+\Textausgabe\FontMetrik

finden Sie ein Programm, das die eben diskutierten Größen per **DrawString()** ausgibt.

15.7.4.2 Textbreite

Das horizontale Positionieren einzelner Wörter in einer auszugebenden Textzeile erledigt meist die Methode **DrawString()**, wobei für den gesamten Text dieselbe Schriftart verwendet wird, z.B.:



Sollen aber z.B. in einer Zeile verschiedene Schriftstile auftauchen, sind mehrere **DrawString()**-Aufrufe und entsprechende Positionsrechnungen erforderlich.

Im folgenden Programm wird die **Graphics**-Methode **MeasureString()** dazu verwendet, um die Breite einer Textausgabe und damit die sinnvolle Startposition der nächsten Textausgabe zu berechnen:

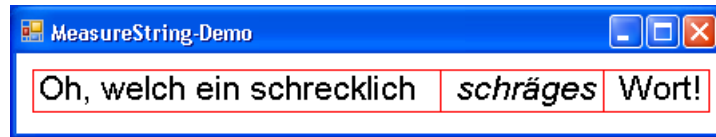
```
using System.Windows.Forms;
using System.Drawing;
using System;
class MeasureStringDemo : Form {
    Brush brush = SystemBrushes.WindowText;
    Font[] fonts;
    string[] texte;

    MeasureStringDemo() {
        Text = "MeasureString-Demo";
        BackColor = SystemColors.Window;
        ClientSize = new Size(440, 50);
        FontFamily ffar = new FontFamily("Arial");
        Font fontRegular = new Font(ffar, 16, FontStyle.Regular);
        Font fontItalic = new Font(ffar, 16, FontStyle.Italic);
        fonts = new Font[] {fontRegular, fontItalic, fontRegular};
        texte = new string[] {"Oh, welch ein schrecklich", " schräges", " Wort!"};
    }

    protected override void OnPaint(PaintEventArgs e) {
        Graphics g = e.Graphics;
        PointF pkt = new PointF(10, 10);
        SizeF size;
        for (int i = 0; i < 3; i++) {
            g.DrawString(texte[i], fonts[i], brush, pkt);
            size = g.MeasureString(texte[i], fonts[i]);
            g.DrawRectangle(Pens.Red, pkt.X, pkt.Y, size.Width, size.Height);
            pkt.X += size.Width;
        }
    }

    [STAThread]
    static void Main() {
        Application.Run(new MeasureStringDemo());
    }
}
```

Man erhält als Rückgabewert ein **SizeF**-Objekt mit Größenangaben in der aktuellen Maßeinheit. Allerdings ist nicht nur die Höhe (siehe Abschnitt 15.7.4.1), sondern auch die Breite leicht übertrieben, so dass zwischen horizontal aufeinander folgenden **DrawString()**-Ausgaben unschöne Lücken entstehen, wenn man sich bei der Positionierung an diesem **SizeF**-Objekt orientiert, z.B.:



Das Ergebnis wird präziser, wenn bei **DrawString()** und bei **MeasureString()** ein spezielles **StringFormat**-Objekt assistiert, das von der statischen **StringFormat**-Eigenschaft **GenericTypographic** geliefert wird.

```
StringFormat sf = StringFormat.GenericTypographic;
```

Man muss seine **FormatFlags**-Eigenschaft um den Wert **MeasureTrailingSpaces** aus der Enumeration **StringFormatFlags** erweitern, damit Leerzeichen am Ende einer Zeichenfolge bei der Breitenberechnung Berücksichtigung finden, z.B.:

```
protected override void OnPaint(PaintEventArgs e) {
    Graphics g = e.Graphics;
    PointF pkt = new PointF(10, 10);
    SizeF size;
    StringFormat sf = StringFormat.GenericTypographic;
    sf.FormatFlags |= StringFormatFlags.MeasureTrailingSpaces;
    for (int i = 0; i < 3; i++) {
        g.DrawString(texte[i], fonts[i], brush, pkt, sf);
        size = g.MeasureString(texte[i], fonts[i], pkt, sf);
        g.DrawRectangle(Pens.Red, pkt.X, pkt.Y, size.Width, size.Height);
        pkt.X += size.Width;
    }
}
```

Trotz aller Mühen überschätzt **MeasureString()** nach wie vor die Breite, wobei der (absolute) Fehler offenbar mit der Zeichenfolgenlänge wächst, z.B.:



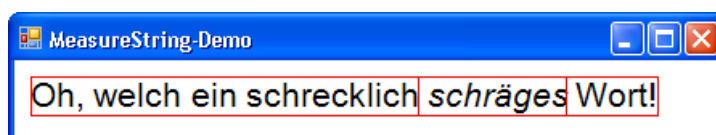
Eine weitere Verbesserung stellt sich ein, wenn man über die **Graphics**-Eigenschaft **TextRenderingHint** das **Antialiasing** einschaltet, z.B.

```
g.TextRenderingHint = System.Drawing.Text.TextRenderingHint.AntiAlias;
```

Nun kommen **DrawString()** und **MeasureString()** bei der horizontalen Ausdehnung fast zusammen, wobei allerdings das Antialiasing wegen des damit verbundenen Unschärfeeindrucks nicht unbedingt gefallen muss:



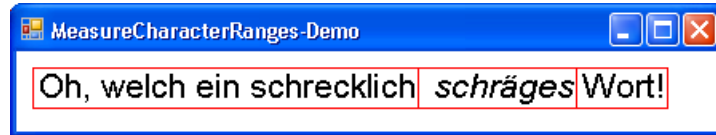
Bei einer LCD-Anzeige sollte statt des gewöhnlichen Antialiasing über den **TextRenderingHint**-Wert **ClearTypeGridFit** die hochwertigere **ClearType**-Technik verwendet werden, die auch bei der Breitenmessung nochmals besser abschneidet:



Im Internet wird das Genauigkeitsproblem der **MeasureString()**-Methode intensiv diskutiert. Wer kein Antialiasing nicht verwenden möchte, findet auf der Webseite

<http://www.codeproject.com/cs/media/measurestring.asp>

eine von *Pierre Arnaud* vorgeschlagene Lösung zur Berechnung der Textbreite, welche auf der **Graphics**-Methode **MeasureCharacterRanges()** basiert und im Beispiel folgendes Ergebnis produziert:.



Als Alternative zu der problematischen **Graphics**-Methode **MeasureString()** wird für Bildschirmausgaben vielfach die statische Methode **MeasureText()** der seit .NET 2.0 verfügbaren Klasse **TextRenderer** im Namensraum **System.Windows.Forms** empfohlen. Dann sollte man aber die *Ausgabe* mit der zugehörigen **TextRenderer**-Methode **DrawText()** bewerkstelligen, die leider *nicht* zum Drucken taugt.

Während die Klasse **Graphics** das GDI+ - Textrendering benutzt, verwendet die Klasse **TextRenderer** für diesen Zweck das ältere GDI (ohne Plus).

Seit .NET 2.0 können viele FCL-Steuerelemente zum Rendern ihrer Texte alternativ die Klasse **Graphics** oder die Klasse **TextRenderer** verwenden. Die vom Visual Studio 2008 bei neuen WinForms-Projekten per Voreinstellung in die **Main()**-Methode der Startklasse aufgenommene Anweisung

```
Application.SetCompatibleTextRenderingDefault(false);
```

(vgl. Abschnitt 9.6.4) sorgt übrigens dafür, dass per Voreinstellung auf die Kompatibilität mit .NET 1.x verzichtet und die neue .NET 2 - Technik verwendet wird (, die eigentlich eine Rückkehr zum Altbewährten ist).

15.8 Transformationen

Bisher haben wir uns auf Bildschirmausgaben beschränkt und dabei das Standardkoordinatensystem mit der Maßeinheit Pixel verwendet. Soll ein Programm verschiedene Bildschirmauflösungen berücksichtigen, alternative Ausgabegeräte (z.B. Drucker) unterstützen und/oder besonderer Ansprüche an maßhaltige Ausgaben erfüllen (z.B. in CAD-Anwendungen), dann sind metrische Maßeinheiten unvermeidlich (siehe Abschnitt 15.8.2). Bei ihrem Einsatz werden die in Zeichenmethoden angegebenen **Seitenkoordinaten** bei der Ausgabe vom .NET -Framework umgerechnet in **Gerätekoordinaten**. Es werden z.B. hinreichend viele Pixel eingeschwärzt, damit die von Ihnen gewünschte schwarze Linie mit einer Länge von 5 cm auf dem Bildschirm erscheint. In Abschnitt 15.8.1 geht es um kleine Fehler, die durch unvollständiges Wissen über die tatsächliche Größe eines Geräte-Pixels entstehen können.

Oft ist ein Koordinatensystem wünschenswert, das sich vom System der Seitenkoordinaten der zu bemalenden Steuerelementoberfläche (mit dem Ursprung links oben und einer waagrecht verlaufenden X-Achse) unterscheidet, z.B. durch einen verschobenen Ursprung und/oder durch eine Drehung. In Abschnitt 15.8.3 beschäftigen wir uns damit, wie man eine solche Welttransformation vereinbart, damit man anschließend in Zeichenmethoden die Koordinaten der eigenen Welt verwenden und dem .NET - Framework die Übersetzung in die Seitenkoordinaten überlassen kann.

Wir müssen also insgesamt drei Koordinatensysteme unterscheiden:

- **Weltkoordinaten**

In den Zeichenmethoden werden Weltkoordinaten verwendet. Sie unterscheiden sich von den Seitenkoordinaten der zu bemalenden Steuerelementoberfläche, sobald eine Welttransformation aktiv ist.

- **Seitenkoordinaten**

Dies sind die logischen Koordinaten der Normalwelt. Sie unterscheiden sich von den Bildschirmkoordinaten, sobald eine andere Maßeinheit als Pixel verwendet wird.

- **Gerätekoordinaten**

Diese beziehen sich auf die (mehr oder weniger fixierte) Pixelmatrix des Ausgabegeräts.

Ist weder eine Welt- noch eine Seitentransformation im Spiel, sind die Koordinatensysteme identisch.

15.8.1 Auflösungserscheinungen

Während ein Druckertreiber sehr genau weiß, wie groß seine Zeichenfläche (also der bedruckbare Bereich eines Blattes) ist, hat das Betriebssystem nur eine unsichere Vermutung über die Größe der Bildschirmfläche. Windows kennt die folgenden, vom Benutzer beeinflussbaren Einstellungen (vgl. Abschnitt 15.7.4.1):

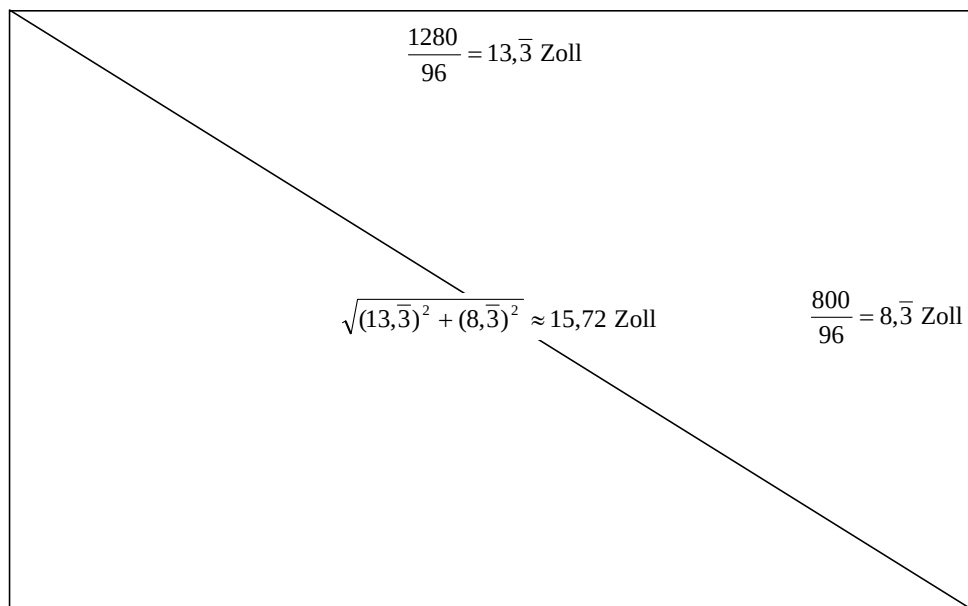
- *Anzahl* der Bildpunkte pro Zeile bzw. Spalte
- Bildschirmauflösung in der Einheit DPI (*Dots Per Inch*)

Daraus ist die Größe des Bildschirms leicht zu errechnen.

Beim LCD-Monitor eines 15,4 Zoll - Laptops wurden folgende Werte festgestellt:

- Bildpunkte: 1280×800
- DPI-Einstellung: 96

Windows geht also von einer Bildschirmdiagonale mit ca. 15,72 Zoll aus:

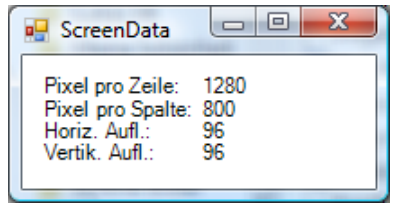


Die tatsächliche Bildschirmdiagonale des Monitors beträgt (wie vom Hersteller angegeben und durch Messung bestätigt) 15,4 Zoll, was einer Auflösung von 98 DPI entspricht. In diesem Fall stimmt die Annahme des Betriebssystems relativ gut mit der Realität überein, und eine per **DrawLine()** mit der metrischen Längenangabe (siehe unten) von 5 cm ausgegebene Linie erreicht auf dem Monitor mit ca. 4,9 cm fast die korrekte Länge. Stellt der Laptop-Benutzer aber den DPI-Wert

120 ein (vgl. Abschnitt 15.7.4.1), dann produziert derselbe **DrawLine()**-Aufruf eine Linie mit 6,25 cm Länge.

	angenommener DPI-Wert	
	96	120
Verfälschungsfaktor	$96/98 \approx 0,98$	$120/98 = 1,25$
Länge einer 5 cm -Linie	$\approx 4,9$ cm	6,25 cm

Im folgenden Quellcodesegment wird demonstriert, wie man die Anzahl der horizontalen bzw. vertikalen Pixel auf dem primären Bildschirm ermitteln kann. Von welchen Auflösungen ein **Graphics**-Objekt (mit einem Bildschirm oder auch mit einem Drucker verbunden) ausgeht, erfährt man über seine Eigenschaften **DpiX** und **DpiY**:

Quellcodefragment	Ausgabe
<pre>protected override void OnPaint(PaintEventArgs e) { Graphics g = e.Graphics; g.DrawString("Pixel pro Zeile:\t" + Screen.PrimaryScreen.Bounds.Width.ToString() + "\nPixel pro Spalte:\t" + Screen.PrimaryScreen.Bounds.Height + "\nHoriz. Aufl.:\t" + g.DpiX.ToString() + "\nVertik. Aufl.:\t" + g.DpiY.ToString(), Font, SystemBrushes.WindowText, 10, 10); }</pre>	

Bei der Druckausgabe, mit der wir uns bald beschäftigen, werden Positions- und Größenangaben per Voreinstellung in Einheiten von 1/100 Zoll (also metrisch) interpretiert, so dass z.B. die Anweisung

```
g.DrawLine(Pens.Black, 0, 10, 100, 10);
```

auf *jedem* Drucker eine Linie von exakt 2,54 cm (= 1 Zoll) Länge erzeugt.

Weil ein Drucker per Voreinstellung als 100-DPI-Gerät behandelt wird, und Monitore häufig mit ca. 100 DPI arbeiten, genügt für einfache Zwecke die Orientierung an dieser Standardauflösung.

15.8.2 Metrische Maßeinheiten (Seitentransformation)

Wir haben bisher ausschließlich den Bildschirm als Ausgabegerät verwendet, wobei für Positions- und Größenangaben die voreingestellte Maßeinheit Pixel akzeptiert wurde. Das .NET - Framework unterstützt aber auch diverse *metrische* Maßeinheiten (z.B. Millimeter). Sollen Graphikproduktionen mit exakten Positionen und Größen in der Druckausgabe landen, dann sind metrischen Maßeinheiten unvermeidlich. In vielen Fällen sind diese auch bei der (reinen) Bildschirmausgabe von Vorteil.

Für eine Änderung der Maßeinheit über die **Graphics**-Eigenschaft **PageUnit** stehen folgende Werte der Enumeration **GraphicsUnit** zur Verfügung:

Wert	Bedeutung
Display	Bildschirm: identisch mit Pixel Drucker: 1 Einheit = 1/100 Zoll (Voreinstellung)
Pixel	1 Einheit = 1 Pixel (Voreinstellung beim Bildschirm)
Point	1 Einheit = 1/72 Zoll
Inch	1 Einheit = 1 Zoll
Document	1 Einheit = 1/300 Zoll
Millimeter	1 Einheit = 1 Millimeter

Über die **Graphics**-Eigenschaft **PageScale** lässt sich die Schrittweite verfeinern oder vergrößern. Nach den folgenden Anweisungen versteht das **Graphics**-Objekt alle Positionen und Größen in der Einheit 1/10 Millimeter:

```
g.PageUnit = GraphicsUnit.Millimeter;
g.PageScale = 0.1f;
```

Mit den **Graphics**-Eigenschaften **PageUnit** und **PageScale** wird definiert, wie die in Ausgabeanweisungen enthaltenen **Seitenkoordinaten** in **Gerätekoordinaten** umzurechnen sind. Bei obigen Einstellungen und (**DpiX** = **DpiY** = 96) resultiert z.B. die folgende **Seitentransformation**:

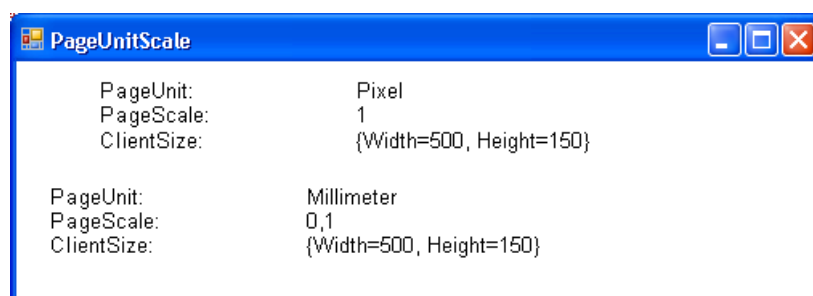
Ausgabeanweisung	Seitenkoordinaten der Punkte	Gerätekoordinaten der Punkte
<code>g.DrawLine(Pens.Black, 100, 100, 600, 100);</code>	(100, 100) (600, 100)	(38, 38) (227, 38)

Lässt man **Graphics**-Eigenschaften **PageUnit** und **PageScale** unverändert, stimmen Seiten- und Gerätekoordinaten überein (keine Seitentransformation).

Weil die per **PageUnit** und **PageScale** definierte Seitentransformation mit einem **Graphics**-Objekt vereinbart wird, hat sie z.B. keinen Einfluss auf die Formulareigenschaft **ClientSize**. Im folgenden **Paint**-Handler wird u.a. die **ClientSize**-Eigenschaft vor und nach einer Seitentransformation durch einen **DrawString()**-Aufruf mit der horizontalen X-Koordinate 50 protokolliert:

```
protected override void OnPaint(PaintEventArgs e) {
    Graphics g = e.Graphics;
    g.DrawString("PageUnit:\t\t" + g.PageUnit.ToString() +
        "\nPageScale:\t\t" + g.PageScale.ToString() +
        "\nClientSize:\t\t"+ClientSize.ToString(),
        new Font("Arial", 10), SystemBrushes.WindowText, 50, 10);
    g.PageUnit = GraphicsUnit.Millimeter;
    g.PageScale = 0.1f;
    g.DrawString("PageUnit:\t\t" + g.PageUnit.ToString() +
        "\nPageScale:\t\t" + g.PageScale.ToString() +
        "\nClientSize:\t\t"+ClientSize.ToString(),
        new Font("Arial", 10), SystemBrushes.WindowText, 50, 200);
}
```

Die Seitentransformation verändert zwar die horizontale Position der **DrawString()**-Ausgabe, aber nicht den **ClientSize**-Wert:



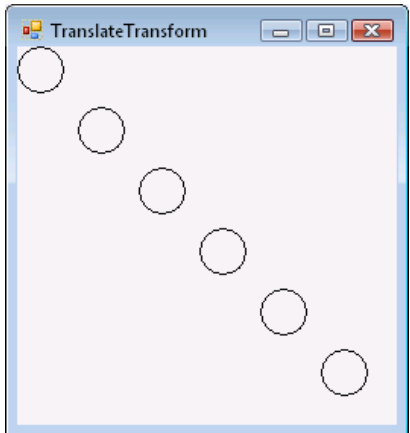
Das Beispielprogramm in Abschnitt 15.8.1 demonstriert, wie die ungefähre *metrische* Größe des Klientenbereichs ermittelt werden kann.

15.8.3 Welttransformationen

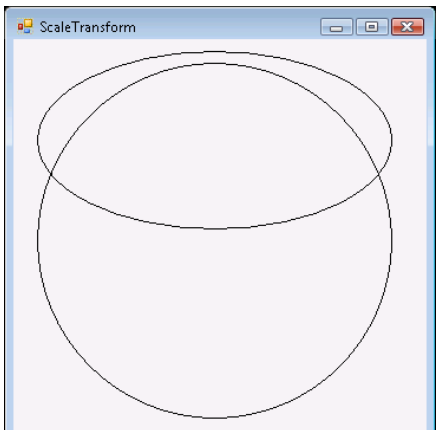
Über eine so genannte *Welttransformation* legt man einen alternativen Bezugsrahmen für die Koordinaten in den Ausgabeanweisungen fest. Man kann ...

- den Nullpunkt verschieben,
- für die X- und oder die Y-Achse eine Streckung bzw. Dehnung vereinbaren,
- oder die Zeichenebene um den Nullpunkt rotieren.

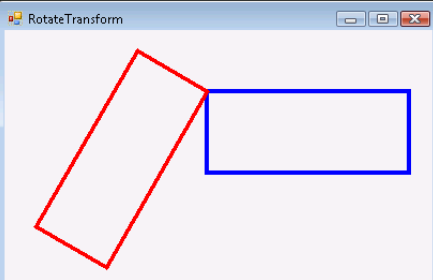
Mit der **Graphics**-Methode **TranslateTransform()** lässt sich der Ursprung des Koordinatensystems aus seiner linken oberen Ecke locken, z.B.:

Quellcodefragment	Ausgabe
<pre>protected override void OnPaint(PaintEventArgs e) { Graphics g = e.Graphics; for (int i = 0; i < 6; i++) { g.DrawEllipse(Pens.Black, 0, 0, 30, 30); g.TranslateTransform(40, 40); } }</pre>	

Die mit der **Graphics**-Methode **ScaleTransform()** möglichen Streckungen und Stauchungen des Koordinatensystems wirken ähnlich wie eine Änderung der Maßeinheit, wobei jedoch beide Richtungen der Ebene *unterschiedlich* behandelt werden können. Um einen runden Kopf mit elliptischem Hut zu zeichnen, wird im folgenden **Paint**-Handler zweimal derselbe **DrawEllipse()**-Aufruf verwendet, wobei zwischenzeitlich eine Skalierungstransformation stattfindet:

Quellcodefragment	Ausgabe
<pre>protected override void OnPaint(PaintEventArgs e) { Graphics g = e.Graphics; g.DrawEllipse(Pens.Black, 20, 20, 300, 300); g.ScaleTransform(1, 0.5f); g.DrawEllipse(Pens.Black, 20, 20, 300, 300); }</pre>	

Mit der **Graphics**-Methode **RotateTransform()** kann man eine Figur drehen, ohne Koordinatenneuberechnungen durchführen zu müssen, z.B.:

Quellcodefragment	Ausgabe
<pre>protected override void OnPaint(PaintEventArgs e) { Graphics g = e.Graphics; g.PageUnit = GraphicsUnit.Millimeter; g.TranslateTransform(50, 15); g.DrawRectangle(Pens.Blue, 0, 0, 50, 20); g.RotateTransform(120); g.DrawRectangle(Pens.Red, 0, 0, 50, 20); }</pre>	

Nach Ausgabe des blauen Rechtecks per **DrawRectangle()** wird das Koordinatensystem um 120 Grad *im Uhrzeigersinn* gedreht. Anschließend wird das rote Rechteck gezeichnet, wobei sich im **DrawRectangle()**-Aufruf nur die Farbe des **Pen**-Objekts ändert. Als Maßeinheit werden Millimeter verwendet, und der Ursprung des Koordinatensystems wird mit **TranslateTransform()** verschoben.

Im letzten Beispiel ist sowohl eine Seitentransformation vorhanden (Verwendung einer metrischen Maßeinheit), als auch eine Welttransformation (Verschiebung und Rotation). Die in **Graphics**-Zeichenbefehlen enthaltenen Positionen und Größen werden auf dem Weg zu Pixelwerten für den Bildschirm also zweimal transformiert (vgl. Petzold 2002, S. 247):

- Welttransformation zur Berücksichtigung der Verschiebung und der Rotation
- Seitentransformation zur Berücksichtigung der Maßeinheit

Mit Grundkenntnissen in linearer Algebra lassen sich unter Verwendung von Objekten der Klasse **Matrix** aus dem Namensraum **System.Drawing.Drawing2D**) sehr flexible Transformationen konstruieren (z.B. Scherungen).

Mit der **Graphics**-Methode **ResetTransform()** lassen sich alle aktiven Welttransformationen abschalten.

15.9 Rastergrafik

Mit der Klasse **Bitmap** unterstützt das .NET – Framework die Darstellung und Bearbeitung von Rastergrafiken, die z.B. aus einer Datei importiert werden können:

```
image = new Bitmap("land.jpg");
```

Das folgende Fenster zeigt links das per **DrawImage()**

```
g.DrawImage(image, 10, 10);
```

gemalte Original und rechts die per

```
ra = new RectangleF(280, 10, 1.5f * image.Width, image.Height);
g.DrawImage(image, ra);
```

horizontal gedehnte Variante:



Zur Definition der gewünschte Position und Größe kommt bei der zweiten Ausgabe eine **RectangleF**-Instanz zum Einsatz.

Wie der komplette Quellcode zum Beispiel demonstriert, sollte man das Laden des Bildes keinesfalls in der **Paint**-Ereignisbehandlungsmethode vornehmen, sondern z.B. im Formularkonstruktor:

```
using System;
using System.Windows.Forms;
using System.Drawing;
class BitmapDemo : Form {
    Bitmap image;
    RectangleF ra;

    BitmapDemo() {
        Text = "Bitmap";
        ClientSize = new Size(680, 280);
        image = new Bitmap("land.jpg");
        ra = new RectangleF(280, 10, 1.5f * image.Width, image.Height);
    }

    protected override void OnPaint(PaintEventArgs e) {
        Graphics g = e.Graphics;
        g.DrawImage(image, 10, 10);
        g.DrawImage(image, ra);
    }

    [STAThread]
    static void Main() {
        Application.Run(new BitmapDemo());
    }
}
```

Beim empfehlenswerten Verzicht auf eine Pfadangabe wird die im **Bitmap**-Konstruktor angegebene Datei im Verzeichnis mit dem Assembly erwartet (z.B. in **...bin\Debug**). Später werden Sie eine Möglichkeit kennen lernen, Bitmap-Dateien als *Ressourcen* in ein Assembly aufzunehmen.

Im .NET - Framework werden folgende Rastergrafikdateiformate unterstützt:

- BMP (Bitmap)
- GIF (Graphics Interchange Format)
- JPEG (Joint Photographic Experts Group)
- EXIF (Exchangeable Image File)
- PNG (Portable Network Graphics)
- TIFF (Tag Image File Format)

Während das obige Bild (mit 256×256 - Pixelmatrix) vom manipulierten Seitenverhältnis sogar profitiert hat, ist die Verzerrung bei Einpassung eines (z.B. vom Benutzer via Dateistandarddialog gelieferten) Bildes in ein Rechteck mit festgelegter Größe in der Regel zu vermeiden. Im folgenden Beispiel (Rechteck wie oben, 2166×1141 Pixelmatrix) ist mit Beschwerden der Hauptdarsteller¹ zu rechnen:



Im folgenden Quellcodesegment

```
int bmh = image.Width;
int bmv = image.Height;
float faktor = Math.Min(((float)ClientSize.Width) / bmh,
                        ((float)ClientSize.Height) / bmv);

if (faktor < 1.0)
    ra = new RectangleF(0, 0, faktor*bmh, faktor*bmv);
else
    ra = new RectangleF(0, 0, bmh, bmv);
```

wird zur Einpassung auf beide Achsen derselbe Faktor angewendet, der sich als minimaler Quotient aus den korrespondierenden Pixelzahlen von Bild und Zielrechteck ergibt:

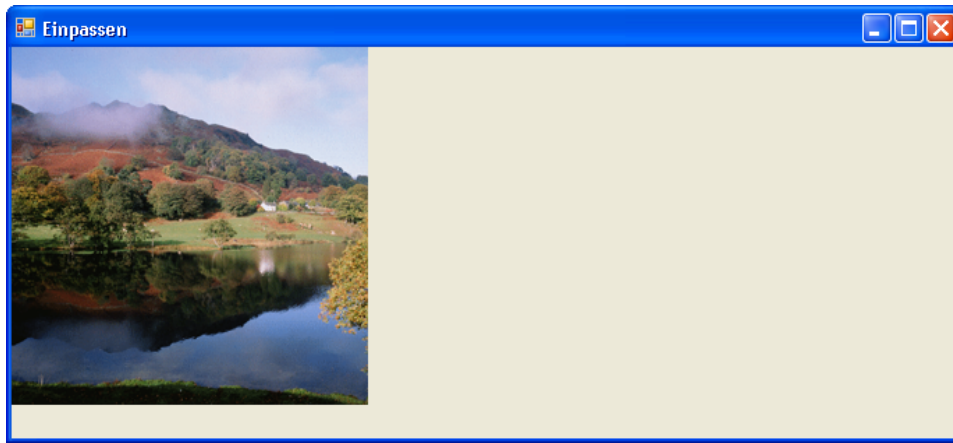
$$\text{Min} \left(\frac{\text{Zeilenpixel im Zielrechteck}}{\text{Zeilenpixel im Bild}}, \frac{\text{Spaltenpixel im Zielrechteck}}{\text{Spaltenpixel im Bild}} \right)$$

Nun unterbleibt die unschmeichelhafte Dehnung:



Auf ein Vergrößern von Bildern wird im Beispiel explizit verzichtet (siehe **if**-Anweisung):

¹ Zu sehen sind ein Berner Sennenhund (*Bodo*) und eine braune Sennenhündin (*Emma*).



Zwei Methoden der Klasse **Bitmap** haben wir schon in Abschnitt 9.4.2.3 zum Bemalen von **Button**-Objekten verwendet:

- **public Color GetPixel(int x, int y)**
ermittelt die Farbe eines Pixels
- **public void MakeTransparent(Color Transparentfarbe)**
setzt eine Farbe des Bitmap-Objekts transparent

Zur Bearbeitung eines **Bitmap**-Objekts stehen u.a. die folgenden Methoden zur Verfügung:

- **public Color SetPixel(int x, int y)**
setzt die Farbe eines Pixels
- **public void RotateFlip(RotateFlipType Rotations-Kipp-Typ)**
erlaubt u.a. das horizontale Spiegeln eines Bildes, z.B.:



Man kann sich mit der statischen **Graphics**-Methode **FromImage()** ein **Graphics**-Objekt zu einem Bild besorgen und hat dann alle in Abschnitt 15 beschriebenen Zeichenmethoden zur Verfügung.

Mit dem folgenden Quellcodefragment

```
Pen pen = new Pen(Color.Red, 20);
Graphics g = Graphics.FromImage(image);
g.DrawRectangle(pen, 1450, 50, 700, 650);
```

wird die braune Sennenhündin hervorgehoben:



Über die **Bitmap**-Methode **Save()** lässt sich ein Bild in eine Datei sichern.

Über die Klasse **System.Drawing.Imaging.Metafile**, die wie **Bitmap** von der abstrakten Klasse **Image** abstammt, unterstützt das .NET -Framework auch die Bearbeitung von vektorbasierten Grafiken (siehe Luis & Strasser 2008, S. 859).

15.10 Pfade, Zeichen- und Anzeigebereiche

15.10.1 Grafikpfade

In der **OnPaint**-Methode des folgenden Programms

```
using System.Windows.Forms;
using System.Drawing;
using System;

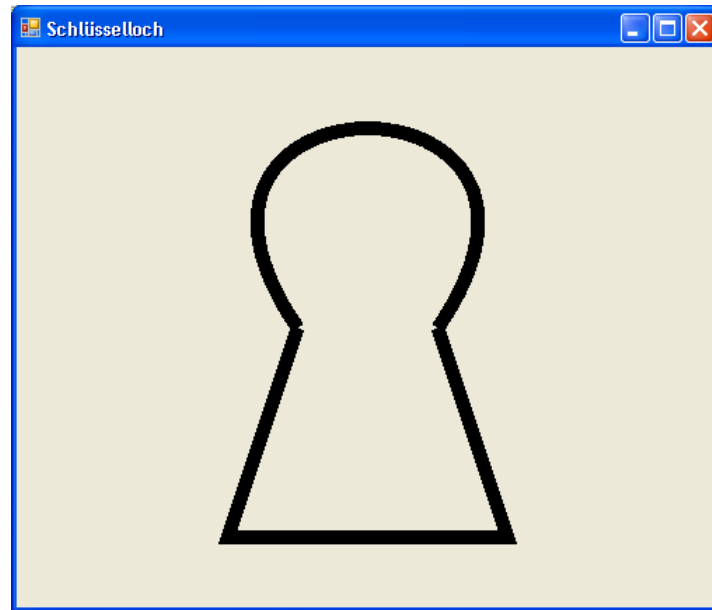
class Schluesselloch : Form {
    Pen stift = new Pen(Color.Black, 10);
    Point ul, ur, p0, p1, p2, p3;
    Point[] lz;

    Schluesselloch() {
        Text = "Schlüsselloch";
        ClientSize = new Size(500, 400);
        ul = new Point(150, 350);
        ur = new Point(350, 350);
        p0 = new Point(200, 200);
        p1 = new Point(70, 10);
        p2 = new Point(430, 10);
        p3 = new Point(300, 200);
        lz = new Point[] { p0, ul, ur, p3 };
    }

    protected override void OnPaint(PaintEventArgs e) {
        e.Graphics.DrawLines(stift, lz);
        e.Graphics.DrawBezier(stift, p0, p1, p2, p3);
    }

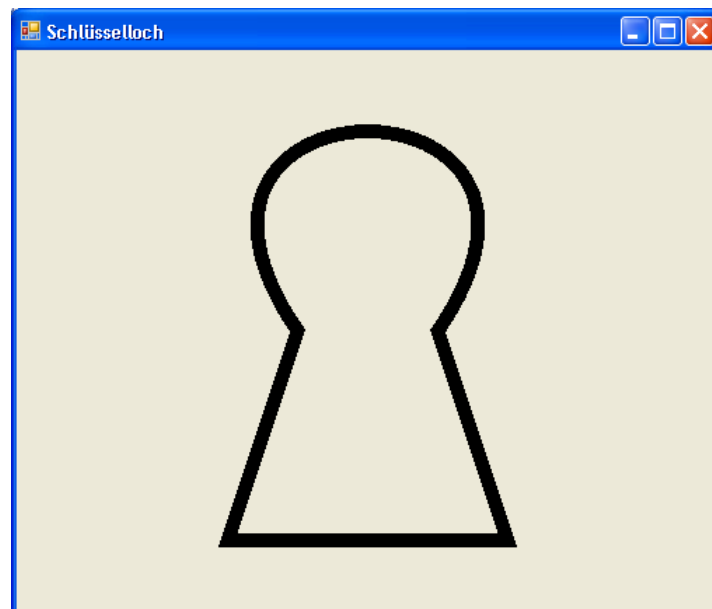
    [STAThread]
    static void Main() {
        Application.Run(new Schluesselloch());
    }
}
```

wird mit den **Graphics**-Methoden **DrawLines()** und **DrawBezier()** (siehe Abschnitt 15.6) ein Schlüsselloch gezeichnet:



Der **DrawLines()**-Aufruf ist nicht nur bequemer als drei einzelne **DrawLine()**-Aufrufe, sondern sorgt zusätzlich für „ansatzfreies“ Zeichnen an den Verbindungspunkten. An den Übergängen zwischen dem Linienzug und der Bézier-Kurve sind jedoch störende Lücken zu erkennen. Zur eleganten Lösung derartiger Probleme enthält das .NET-Grafiksystem die Klasse **GraphicsPath** (Namenraum **System.Drawing.Drawing2D**). Man übergibt einem **GraphicsPath**-Objekt eine Sequenz von Linien und Kurven und verwendet es dann als Parameter der **Graphics**-Methode **DrawPath()**, welche den gesamten Pfad mit perfekten Übergängen zeichnet.

Um im Beispielprogramm das folgende Ergebnis zu erzielen,



kann man so vorgehen:

- Aus **OnPaint()** zu streichen:
`e.Graphics.DrawLines(stift, lz);`
`e.Graphics.DrawBezier(stift, p0, p1, p2, p3);`

- Instanzvariable vom Typ **GraphicsPath** deklarieren und Objekt erstellen:

```
using System.Drawing.Drawing2D;

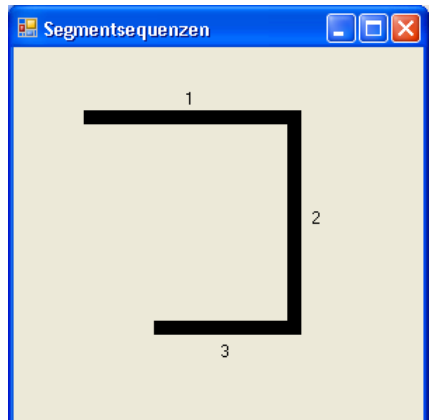
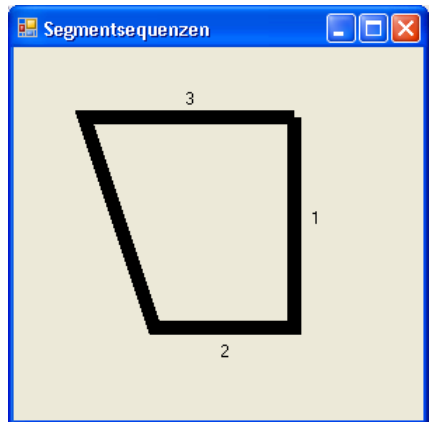
GraphicsPath gp = new GraphicsPath();
```
- Im Formularkonstruktor werden die **GraphicsPath**-Methoden **AddLines()** und **AddBezier()** verwendet, um die benötigten Segmente in den Pfad aufzunehmen:

```
gp.AddLines(lz);
gp.AddBezier(p3, p2, p1, p0);
gp.CloseFigure();
```

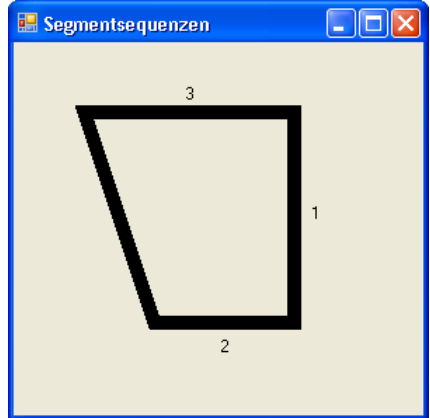
Wozu der **CloseFigure()**-Aufruf dient, und warum die Bézier-Punkte im Vergleich zur Methode **DrawBezier()** in umgekehrter Reihenfolge stehen, ist gleich zu erfahren.
- Jetzt fehlt nur noch die **Graphics**-Zeichenmethode **DrawPath()** in **OnPaint()**:

```
e.Graphics.DrawPath(stift, gp);
```

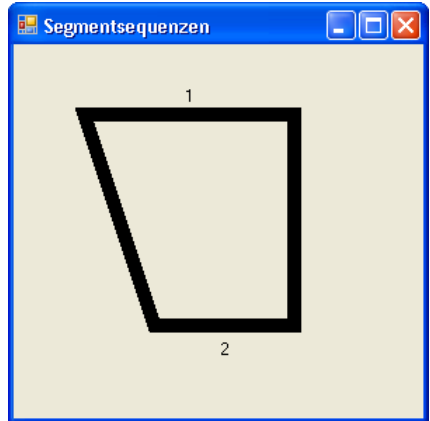
Ein **Pfad** kann aus mehreren **Figuren** bestehen, die sich wiederum aus beliebig vielen **Segmenten** zusammensetzen, welche jeweils mehrere **Punkte** und die (abhängig vom Typ) dadurch definierten Linien bzw. Kurven enthalten. Wir haben eben eine Schlüsselochfigur aus zwei Segmenten aufgebaut (ein Linienzug mit vier Punkten und ein Bézier-Spline) und dabei den Startpunkt des zweiten Segments identisch mit dem Endpunkt des Vorgängers gewählt. Generell werden zwei nacheinander in eine Figur eingefügte Segmente automatisch durch Linien zwischen dem Endpunkt des früheren und Startpunkt des späteren Segments verbunden. Daraus folgt, dass eine Figur trotz Verwendung derselben Segmente in Abhängigkeit von der Einfügereihenfolge sehr verschieden aussehen kann, z.B.:

Quellcodefragment	Ausgabe
<pre>gp.AddLine(50, 50, 200, 50); gp.AddLine(200, 50, 200, 200); gp.AddLine(200, 200, 100, 200);;</pre>	
<pre>gp.AddLine(200, 50, 200, 200); gp.AddLine(200, 200, 100, 200); gp.AddLine(50, 50, 200, 50);</pre>	

Mit der Methode **CloseFigure()** beauftragt man ein **GraphicsPath**-Objekt, die aktuelle Figur abzuschließen. Wie der kleine Defekt in der oberen rechten Ecke der letzten Figur zeigt, ist das Schließen auch dann ratsam, wenn das letzte Segment einer Figur dort endet, wo das erste startet:

Quellcodefragment	Ausgabe
<pre>gp.AddLine(200, 50, 200, 200); gp.AddLine(200, 200, 100, 200); gp.AddLine(50, 50, 200, 50); gp.CloseFigure();</pre>	

Schreibfaule können sich am Ende einer Figur ein Liniensegment sparen, weil dieses von **CloseFigure()** ohnehin ergänzt wird. Außerdem bietet das automatische Füllen von Lücken (siehe oben) noch Einsparmöglichkeiten:

Quellcodefragment	Ausgabe
<pre>gp.AddLine(50, 50, 200, 50); gp.AddLine(200, 200, 100, 200); gp.CloseFigure();</pre>	

Mit **StartFigure()** kann man eine neue Figur beginnen, ohne die aktuelle zu schließen. Während **CloseFigure()** die aktuell im Aufbau befindliche Figur schließt, bearbeitet **CloseAllFigures()** alle offenen Figuren.

Bisher bestanden unsere **GraphicsPath**-Beispiele einer einzigen Figur. Im folgenden Programm

```
using System;
using System.Windows.Forms;
using System.Drawing;
using System.Drawing.Drawing2D;

class Figuren : Form {
    Pen pen = new Pen(Color.Black, 5);
    GraphicsPath path = new GraphicsPath();

    Figuren() {
        Text = "Figuren";
        ClientSize = new Size(400, 250);
        path.AddBezier(100, 150, 150, 250, 400, 200, 250, 50);
        path.CloseFigure();
        path.AddEllipse(250, 100, 15, 15);
        path.AddLine(200, 200, 180, 150);
    }
}
```



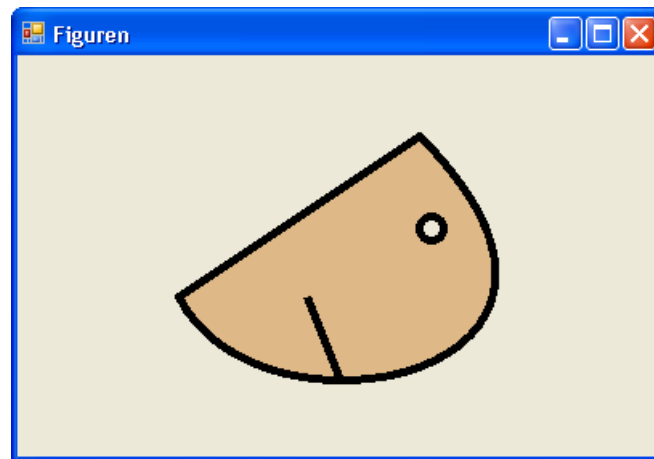
```
protected override void OnPaint(PaintEventArgs e) {
    e.Graphics.FillPath(Brushes.BurlyWood, path);
    e.Graphics.DrawPath(pen, path);
}

[STAThread]
static void Main() {
    Application.Run(new Figuren());
}
}
```

wird ein Exemplar aus drei Figuren erstellt:

- Die erste Figur startet mit einem Bézier-Spline und wird per **CloseFigure()** durch eine Linie komplettiert, so dass sie aus zwei Segmenten besteht.
- Durch die Methode **AddEllipse()** wird eine zweite Figur begonnen und *automatisch* abgeschlossen. Analog verhalten sich die anderen **GraphicsPath**-Methoden zum Erstellen *geschlossener* Figuren, z.B. **AddRectangle()**, **AddPolygon()**.
- Mit **AddLine()** startet also im Beispiel die dritte Figur, die als einziges Segment eine Linie enthält und ohne Abschluss bleibt.

Verwendet in den **Graphics**-Methoden **DrawPath()** und **FillPath()** liefert der Pfad folgendes Ergebnis:



Weitere **GraphicsPath**-Methoden zum Bestücken eines Pfads sind:

- **AddArc()**
Fügt einen Bogen ein
- **AddBeziers()**
Fügt eine Serie von Bézier-Splines ein
- **AddCurve()**
Fügt einen kardinalen Spline hinzu

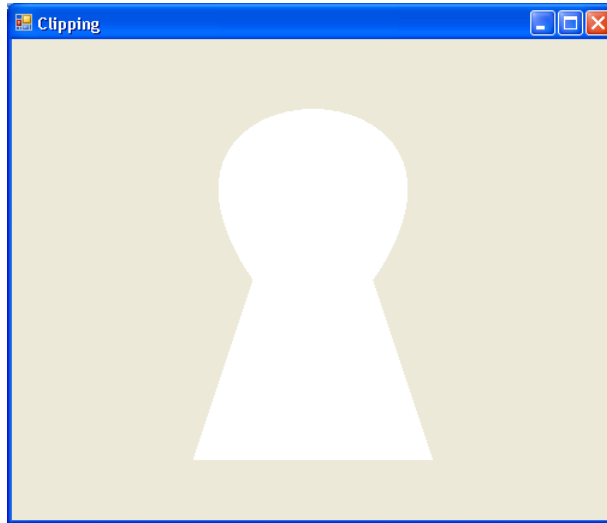
Bögen und kardinale Splines werden bei der Aufnahme in einen Pfad automatisch in Bézier-Splines gewandelt. In Petzold (2002, S. 597ff) findet sich eine zahlreiche weitere Optionen zum Erzeugen und Transformieren von Grafikpfaden.

15.10.2 Zeichenbereiche

Einen Pfad kann man nicht nur zeichnen oder füllen, sondern über die **Graphics**-Methode **SetClip()** auch zur Definition eines Zeichenbereichs (engl. *clip area*) verwenden, auf den die Grafikausgabe anschließend beschränkt werden soll. Als Beispiel verwenden wir den Schlüssellochpfad aus Abschnitt 15.10.1:

```
g.SetClip(gp);  
g.Clear(Color.White);
```

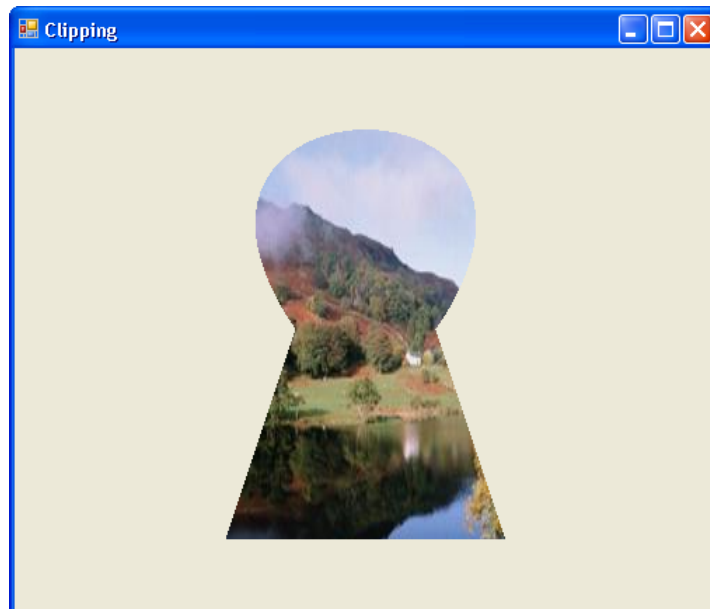
Mit der **Graphics**-Methode **Clear()** setzt man die gesamte zugängliche Zeichenfläche auf eine bestimmte Farbe. Im Beispiel wirkt sich die Methode nur auf das Innere des Schlüssellochs aus:



Mit der Clipping-Technik lassen sich beliebig geformte (per Pfad definierte) Bereiche von Bitmaps anzeigen, z.B.:

```
Bitmap image = new Bitmap("land.jpg");  
g.SetClip(gp);  
g.DrawImage(image, new RectangleF(100, 50,  
                                image.Width, 1.5f * image.Height));
```

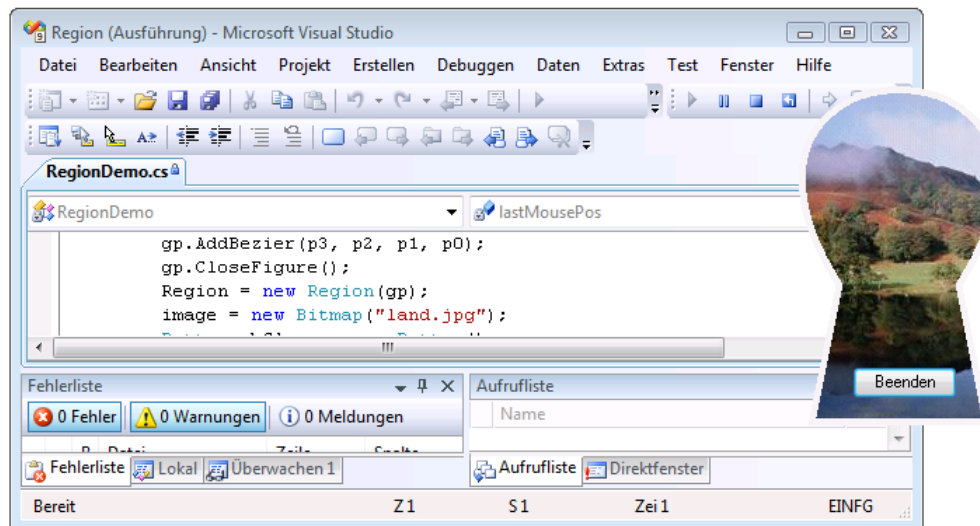
Bei dieser tollen Aussicht, kann man sich den Blick durch das Schlüsselloch nicht verkneifen:



Eine mit den Zeichenbereichen verwandte Technik ist uns schon in Abschnitt 15.3.3 über die programminterne Fensterrenovierung durch Aufruf der **Control**-Methode **Invalidate()** begegnet. Dieser Methode kann man aus Performanzgründen als Parameter einen Bereich übergeben, auf den sich die Renovierung beschränken soll. Dabei ist zur Bereichsdefinition neben der selbstverständlichen **Rectangle**-Option auch ein Objekt der gleich zu behandelnden Klasse **Region** erlaubt, deren Konstruktor u.a. ein **GraphicsPath**-Objekt akzeptiert.

15.10.3 Anzeigebereiche

Mit der eben vorgestellten **Graphics**-Methode **SetClip()** legt man einen Clipping-Bereich fest, auf den sich das angesprochene **Graphics**-Objekt anschließend bei seinen Ausgaben beschränkt. Mit der **Control**-Eigenschaft **Region**, die auf ein Objekt der gleichnamigen Klasse zeigt, lässt sich ein Fensterbereich festlegen, den das Betriebssystem ausschließlich anzeigen soll. Oft definiert man die **Region**-Fläche durch eine **Rectangle**-Instanz, doch über ein **GraphicsPath**-Objekt sind auch andere Gestalten möglich. Im folgenden Beispiel erhält ein Formular über seine **Region**-Eigenschaft ein sehr individuelles Erscheinungsbild:



Im Formularkonstruktor wird ein **Region**-Objekt unter wesentlicher Verwendung eines **GraphicsPath**-Objekts in Schlüssellochform (vgl. Abschnitt 15.10.1) erstellt und dann der **Region**-Eigenschaft des Formulars zugewiesen:

```
RegionDemo() {
    Size = new Size(500, 400);
    Point ul = new Point(170, 300);
    Point ur = new Point(300, 300);
    Point p0 = new Point(190, 200);
    Point p1 = new Point(70, 10);
    Point p2 = new Point(400, 10);
    Point p3 = new Point(280, 200);
    Point[] lz = new Point[] {p0, ul, ur, p3};
    gp = new GraphicsPath();
    gp.AddLines(lz);
    gp.AddBezier(p3, p2, p1, p0);
    gp.CloseFigure();
    Region = new Region(gp);
    image = new Bitmap("land.jpg");
    Button cbClose = new Button();
    cbClose.Top = 230;
    cbClose.Left = 194;
    cbClose.Text = "Beenden";
    cbClose.Parent = this;
    cbClose.Click += new EventHandler(ButtonOnClick);
    stift = new Pen(SystemColors.ButtonFace, 10);
    shiftX = -((Width - ClientSize.Width)/2);
    shiftY = -(Height - ClientSize.Height + shiftX);
}
```

Weil das Formular durch die **Region**-Definition seine Titelzeile samt Standardschaltflächen verloren hat, muss über Ereignismethoden für Beweglichkeit und eine Terminierungsmöglichkeit gesorgt werden. Während die **Click**-Behandlungsmethode zum Befehlsschalter

```
void ButtonOnClick(object sender, EventArgs e) {
    Close();
}
```

keiner weiteren Erläuterung bedarf, sind zur Bewegungstechnik einige Hinweise angebracht. Bei einem Mausklick werden die Bildschirmkoordinaten der Klickposition gespeichert:

```
protected override void OnMouseDown(MouseEventArgs e) {
    lastMousePos = PointToScreen(new Point(e.X, e.Y));
}
```

Eine Mausbewegung wird der Anwendung von Windows nur dann gemeldet, wenn sie über dem Anzeigebereich stattgefunden hat. Die zuständige Ereignisbehandlungsmethode prüft, ob die linke Maustaste gedrückt ist, und verschiebt ggf. die linke obere Ecke des Formulars. Dabei ergibt sich der Bewegungsvektor aus der Differenz zwischen der aktuellen Mausposition und der zuletzt gespeicherten:

```
protected override void OnMouseMove(MouseEventArgs e) {
    if (e.Button == MouseButtons.Left) {
        Point actMousePos = PointToScreen(new Point(e.X, e.Y));
        Left += actMousePos.X - lastMousePos.X;
        Top += actMousePos.Y - lastMousePos.Y;
        lastMousePos = actMousePos;
    }
}
```

Schließlich ist noch ein Detailproblem zu lösen. Bei der **Region** - Definition werden die Koordinaten (im **GraphicsPath**-Objekt) relativ zur linken oberen Ecke des *Formulars* interpretiert. In **Graphics**-Zeichenmethoden beziehen sich die Koordinaten hingegen auf die linke obere Ecke des *Klientenbereichs*. Dies wird in **OnPaint()** durch eine Verschiebung des Ursprungs per **TranslateTransform()** kompensiert, damit der per **DrawPath()** gezeichnete Rand des Schlüssellocks an der richtigen Stelle erscheint:

```
protected override void OnPaint(PaintEventArgs e) {
    Graphics g = e.Graphics;
    g.SmoothingMode = SmoothingMode.HighQuality;
    g.TranslateTransform(shiftX, shiftY);
    g.DrawImage(image, new RectangleF(120, 60, image.Width, image.Height));
    g.DrawPath(stift, gp);
}
```

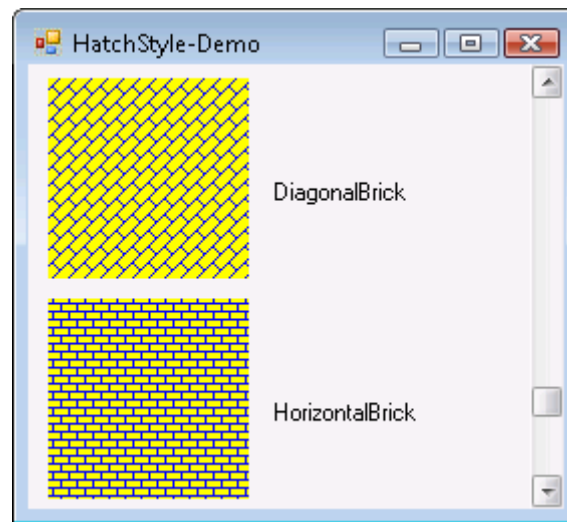
Bei der Verschiebung müssen die Breiten von Fensterrand und Titelzeile berücksichtigt werden. Weile diese von der Windows-Version abhängen könnten, werden sie durch Größenvergleich von Formular und Klientenbereich ermittelt (siehe Fensterkonstruktor).

Den vollständigen Quellcode des Beispielsprogramms finden Sie im Ordner

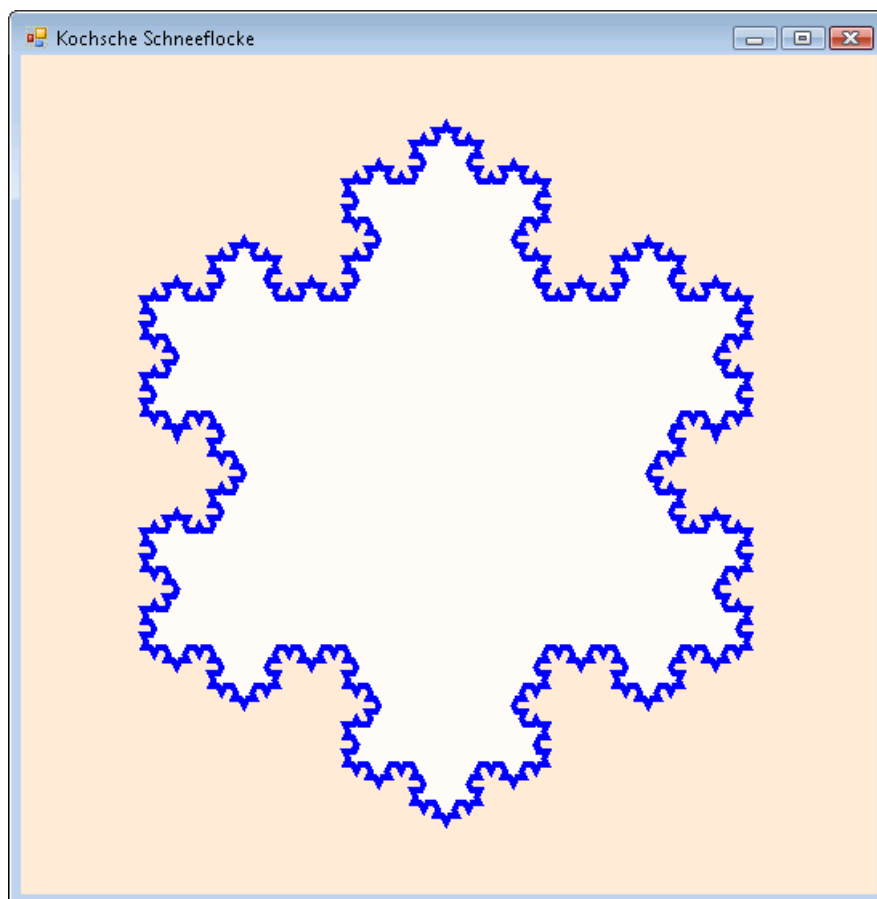
...BspUeb\GDI+\Pfade, Zeichen- und Anzeigebereiche\Region

15.11 Übungsaufgaben zu Kapitel 15

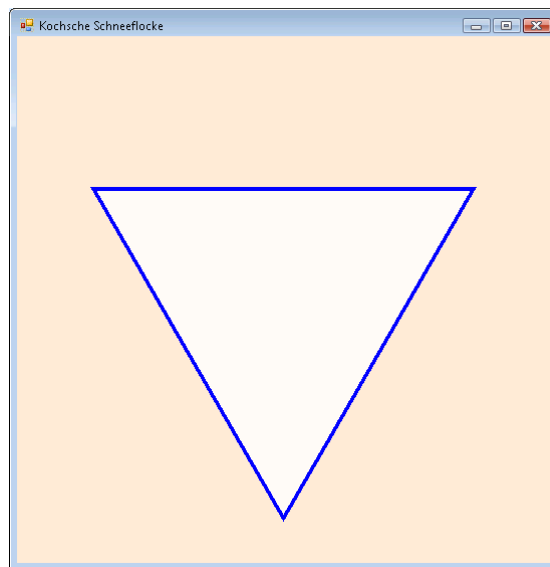
1) Erstellen Sie ein Programm, das die für **HatchBrush**-Objekte verfügbaren Muster demonstriert, z.B. mit folgender Bedienoberfläche:



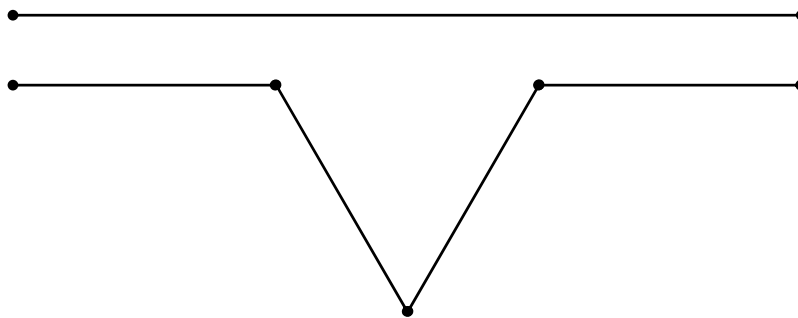
2) Erstellen Sie ein Programm, das die Kochsche Schneeflocke in jeder beliebigen Komplexität zeichnet, z.B. mit folgender Ausgabe:



Man startet mit einem gleichseitigen Dreieck



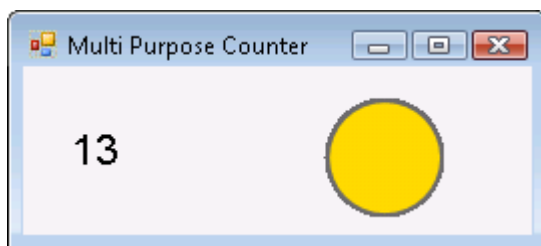
und ersetzt bei jedem Komplexitätsschritt alle geraden Linien durch eine Polylinie nach folgendem Schema:



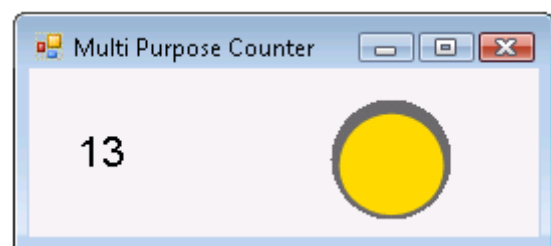
Das mittlere Drittel der Linie wird zu einem gleichseitigen Dreieck ausgeformt.

3) Realisieren Sie mit Hilfe der Klassen **GraphicsPath** und **Region** eine eigene **Button**-Ableitung (vgl. Abschnitt 9.4.2) mit einem *runden* Design, z.B.:

Normalzustand



gedrückt



16 Weitere Standardkomponenten für WinForms-Anwendungen

In diesem Abschnitt werden weitere Komponenten bzw. Steuerelemente aus dem Namensraum **System.Windows.Forms** vorgestellt. Es handelt sich meist um simple Bedienelemente, die in jedem größeren Programm zum Einsatz kommen (z.B. Kontrollkästchen, Optionsfelder). Eine Ausnahme bildet das mächtige **RichTextBox**-Steuerelement, das sich als Basis für einen relativ kompletten Editor nach dem Vorbild der Windows-Zugabe *WordPad* eignet. Wir führen das **RichTextBox**-Steuerelement in Abschnitt 16.5 ein und vertiefen seine Behandlung mehrfach in späteren Abschnitten. Neben Steuerelementen mit der Kompetenz zu selbständiger Benutzerinteraktion wird mit dem **Timer** auch eine Komponente vorgestellt, die im Hintergrund ihren Dienst verrichtet.

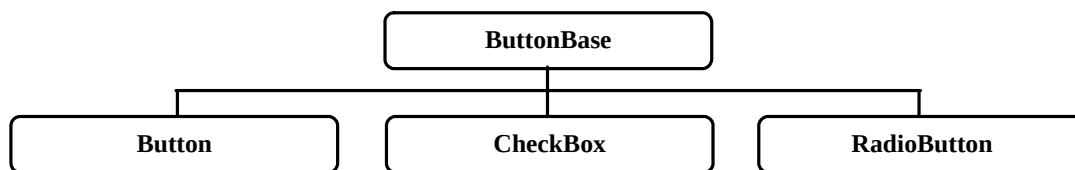
Wer sich beim Einsatz von Windows-Bedienelementen streng an die Vorgaben der Firma Microsoft halten möchte an, kann das kostenlos auf einem Webserver angebotenen Dokument [Official Guidelines for User Interface Developers and Designers](#) (Microsoft 2007b) konsultieren.

16.1 Kontrollkästchen und Optionsfelder

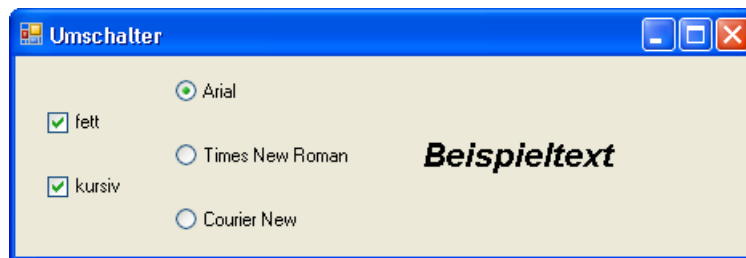
In diesem Abschnitt werden zwei Umschalter vorgestellt:

- Für **Kontrollkästchen** steht die Klasse **CheckBox** zur Verfügung.
- Für ein **Optionsfeld** verwendet man Objekte der Klasse **RadioButton**.

Beide Klassen stammen gemeinsam mit der schon in Abschnitt 9.4.2 vorgestellten Klasse **Button** von der abstrakten Basisklasse **System.Windows.Forms.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:



Die beiden Kontrollkästchen (`cbBold` und `cbItalic` genannt) erlauben das separate Ein- bzw. Ausschalten der Schriftattribute **fett** und **kursiv**. Sie werden im Formularkonstruktor folgendermaßen definiert:¹

¹ Im Beispielprogramm wird die so genannte *Ungarische Notation* zur Bezeichnung der Instanzvariablenamen 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.

```

Umschalter() {
    ClientSize = new Size(460, 125);
    Text = "Umschalter";

    lblTextbeispiel = new Label();
    lblTextbeispiel.Text = "Beispieltext";
    lblTextbeispiel.Font = new Font("Arial", 16);
    lblTextbeispiel.Location = new Point(250, 50);
    lblTextbeispiel.Width = 200;
    Controls.Add(lblTextbeispiel);

    chkBold = new CheckBox();
    chkBold.Text = "fett";
    chkBold.Location = new Point(20, 30);
    chkBold.Width = 60;
    Controls.Add(chkBold);

    chkItalic = new CheckBox();
    chkItalic.Text = "kursiv";
    chkItalic.Location = new Point(20, 70);
    chkItalic.Width = 60;
    Controls.Add(chkItalic);

    EventHandler eh = new EventHandler(CheckBoxOnClick);
    chkBold.Click += eh;
    chkItalic.Click += eh;

    . . .
}

```

Als **EventHandler** wird für beide Kontrollkästchen dasselbe Delegatenobjekt verwendet, das auf der Methode `CheckBoxOnClick()` basiert:

```

protected void CheckBoxOnClick(object sender, EventArgs e) {
    if (chkBold.Checked)
        lblTextbeispiel.Font = new Font(lblTextbeispiel.Font,
            lblTextbeispiel.Font.Style | FontStyle.Bold);
    else
        lblTextbeispiel.Font = new Font(lblTextbeispiel.Font,
            lblTextbeispiel.Font.Style & ~FontStyle.Bold);
    if (chkItalic.Checked)
        lblTextbeispiel.Font = new Font(lblTextbeispiel.Font,
            lblTextbeispiel.Font.Style | FontStyle.Italic);
    else
        lblTextbeispiel.Font = new Font(lblTextbeispiel.Font,
            lblTextbeispiel.Font.Style & ~FontStyle.Italic);
}

```

Die Schriftattribute werden durch eine bitweise Verknüpfung ihrer **FontStyle**-Ausprägung mit dem aktuellen Schriftstil ein- bzw. ausgeschaltet:

- Durch die bitweise Per OR-Operation werden die zum Attribut gehörigen Bits eingeschaltet.
- Durch die bitweise Negation (~) und die anschließende bitweise AND-Operation werden die zum Attribut gehörigen Bits abgeschaltet.

Der Bequemlichkeit halber werden GDI-Ressourcen verschwendend (vgl. Abschnitt 15.2) bei jedem Aufruf der Methode `CheckBoxOnClick()` zwei neue **Font**-Objekte erzeugt, jeweils ausgehend von der aktuellen Schriftart der **Label**-Komponente.

Hat die **AutoCheck**-Eigenschaft eines **CheckBox**- oder **RadioButton**-Objekts den voreingestellten Wert **true**, dann wechselt das Steuerelement bei einem Mausklick oder beim äquivalenten Tastaturkommando automatisch (ohne Beteiligung einer **Click**-Ereignisbehandlungsmethode) seinen Mar-

kierungszustand und den Wert seiner **Checked**-Eigenschaft. Im Beispiel wird dieser Komfort genutzt, und die **Click**-Behandlungsmethoden greifen nur lesend auf die **Checked**-Eigenschaft zu.

Alle unmittelbar zu einem Container gehörenden **RadioButton**-Objekte werden von der CLR als *Gruppe* behandelt, wobei nur *ein* Mitglied eingerastet sein kann (Wert **true** bei der Eigenschaft **Checked**). Im Umschalter-Beispielprogramm fungiert das Hauptfenster als gruppierender Container:

```
Umschalter() {
    . . .

    rbArial = new RadioButton();
    rbArial.Text = "Arial";
    rbArial.Location = new Point(100, 10);
    rbArial.Checked = true;
    Controls.Add(rbArial);

    rbTimesNewRoman = new RadioButton();
    rbTimesNewRoman.Text = "Times New Roman";
    rbTimesNewRoman.Location = new Point(100, 50);
    rbTimesNewRoman.Width = 190;
    Controls.Add(rbTimesNewRoman);

    rbCourierNew = new RadioButton();
    rbCourierNew.Text = "Courier New";
    rbCourierNew.Location = new Point(100, 90);
    Controls.Add(rbCourierNew);

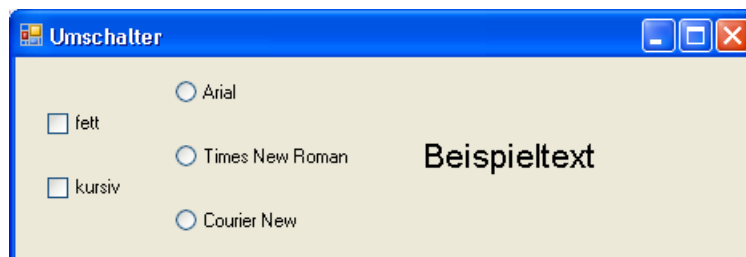
    eh = new EventHandler(RadioButtonOnClick);
    rbArial.Click += eh;
    rbTimesNewRoman.Click += eh;
    rbCourierNew.Click += eh;
}
```

Sollen auf einem Formular *mehrere* **RadioButton**-Gruppen erscheinen, ist jeweils ein eigener Sub-Container erforderlich, der z.B. per **GroupBox**- oder **Panel**-Steuerelement realisiert werden kann (siehe FCL – Dokumentation).

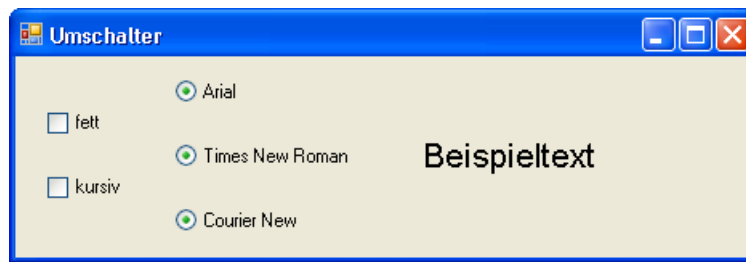
Dass *initial* genau *ein* Objekt einer **RadioButton**-Gruppe markiert ist, muss vom Programmierer durch passendes Vorbesetzen *einer* **Checked**-Eigenschaft sichergestellt werden, z.B.:

```
rbArial.Checked = true;
```

Hat bei *keinem* Gruppenmitglied die **Checked**-Eigenschaft den Wert **true**, resultiert ein gewohnt startendes, aber doch nutzbares Programm:



Haben *mehrere* Gruppenmitglieder beim Programmstart den **Checked**-Wert **true**, dann resultiert ein erratisches Verhalten:



Im Beispielprogramm wird für alle **RadioButton**-Objekte ein gemeinsamer **EventHandler** verwendet:

```
protected void RadioButtonOnClick(object sender, EventArgs e) {
    if (rbArial.Checked)
        lbTextbeispiel.Font = new Font("Arial", lbTextbeispiel.Font.Size,
                                         lbTextbeispiel.Font.Style);
    else
        if (rbTimesNewRoman.Checked)
            lbTextbeispiel.Font = new Font("Times New Roman", lbTextbeispiel.Font.Size,
                                             lbTextbeispiel.Font.Style);
        else
            lbTextbeispiel.Font = new Font("Courier New", lbTextbeispiel.Font.Size,
                                             lbTextbeispiel.Font.Style);
}
```

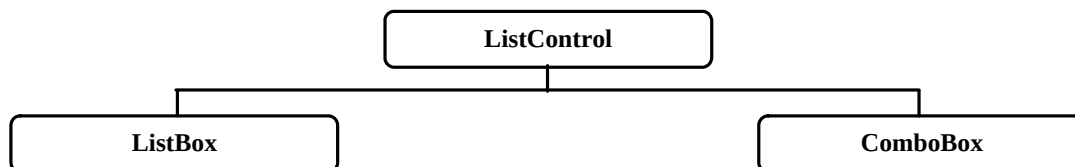
Der Bequemlichkeit halber wird GDI-Ressourcen verschwendend (vgl. Abschnitt 15.2) bei jedem Aufruf der Methode ein neues **Font**-Objekte erzeugt, wobei Größe und Stil von der aktuellen Schriftart der **Label**-Komponente übernommen werden.

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

...\\BspUeb\\WinForms\\Steuerelemente\\Umschalter

16.2 Listen- und Kombinationsfelder

In diesem Abschnitt werden die Steuerelementklassen **ListBox** und **ComboBox** vorgestellt, die von der abstrakten Klasse **System.Windows.Forms.ListControl** abstammen:

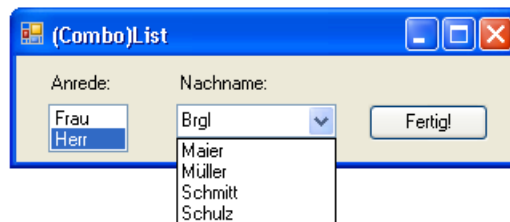


Ein **ListBox**-Steuerelement präsentiert eine Liste von Elementen, die per Mausklick oder geschickte Tastaturbedienung ausgewählt werden können.

Das **ComboBox**-Steuerelement bietet eine Kombination aus einem einzeiligen Textfeld und einer Liste. Um seine Wahl zu treffen, hat der Benutzer *zwei* Möglichkeiten:

- 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:



In diesem einfachen Beispielprogramm spielen die Unterschiede zwischen den Klassen **ListBox** und **ComboBox** keine Rolle:

```
using System;
using System.Windows.Forms;
using System.Drawing;

class ComboList : Form {
    ListBox listBoxAnrede;
    ComboBox listBoxNachname;

    ComboList() {
        listBoxAnrede = new ListBox();
        listBoxAnrede.Location = new Point(20, 30);
        listBoxAnrede.Size = new Size(50, 30);
        listBoxAnrede.Items.AddRange(new String[2] {"Frau", "Herr"});
        listBoxAnrede.SelectedIndex = 0;
        Controls.Add(listBoxAnrede);

        comboBoxNachname = new ComboBox();
        comboBoxNachname.Location = new Point(100, 30);
        comboBoxNachname.Width = 100;
        comboBoxNachname.Items.AddRange(
            new String[4] {"Müller", "Maier", "Schulz", "Schmitt"});
        comboBoxNachname.Sorted = true;
        Controls.Add(comboBoxNachname);

        . . .
    }
}
```

Die Eigenschaft **Items** zeigt auf ein Objekt der inneren Klasse **ListBox.ObjectCollection** bzw. **ComboBox.ObjectCollection** das die Liste der Elemente enthält und diverse Verwaltungsmethoden bietet, z.B.:

- **public int Add(Object element)**
Die Liste wird um *ein* Element erweitert. Zwar ist syntaktisch ein beliebiges Objekt zugelassen, doch kommt letztlich nur eine Zeichenfolge in Frage. Hat die **Sorted**-Eigenschaft des Steuerelements den Wert voreingestellten **false**, wird das neue Element am Ende der Liste angehängt, anderenfalls wird es gemäß Sortierreihenfolge eingefügt. Über den Rückgabewert erfährt man die Position des Neulings.
- **public void AddRange(Object[] elemente)**
Mit der **AddRange()**-Methode fügt man einem Array von Zeichenfolgen in eine Liste ein, z.B.:
`listBoxAnrede.Items.AddRange(new String[2] {"Frau", "Herr"});`
Hat die **Sorted**-Eigenschaft des Steuerelements den Wert voreingestellten **false**, werden die neuen Elemente am Ende der Liste angehängt, anderenfalls werden sie gemäß Sortierreihenfolge eingefügt.
- **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.:

```
listBoxAnrede.SelectedIndex = 0;
```

Über dieselbe Eigenschaft lässt sich auch der Indexwert zur aktuellen Auswahl feststellen. Bei einem **ComboBox**-Steuerelement ist allerdings die **String**-Eigenschaft **Text** weit relevanter, z.B.:

```
comboBoxNachname.Text
```

Sie steht auch bei einem **ListBox**-Steuerelement zur Verfügung.

Über den **Sorted**-Eigenschaftswert **true** beauftragen wir ein **ListBox**- bzw. **ComboBox**-Objekt, seine Elemente stets sortiert zu halten, z.B.:

```
comboBoxNachname.Sorted = true;
```

Dies wirkt sich nicht nur auf die Anzeige aus, sondern auch auf die Vergabe von Indexpositionen beim Einfügen von Elementen (siehe oben).

Im Beispielprogramm sind die durchaus zahlreich vorhandenen **ListBox**- bzw. **ComboBox**-Ereignisse (z.B. **TextChanged**) nicht von Interesse. Allerdings wird in der **Click**-Ereignismethode zum **Button**-Objekt

```
protected void FertigOnClick(object sender, EventArgs e) {
    String nachname;
    if (comboBoxNachname.Text.Length > 0)
        nachname = comboBoxNachname.Text;
    else
        nachname = " ... Äh";
    MessageBox.Show("Guten Tag, " + listBoxAnrede.Text + " " +
        nachname, Text, MessageBoxButtons.OK);
}
```

auf die **Text**-Eigenschaft der Listen-Steuerelemente zugegriffen, um einen freundlichen Gruß formulieren zu können:



Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

...\\BspUeb\\WinForms\\Steuerelemente\\Listen\\ComboList

In einem weiteren Beispielprogramm sollen folgende Techniken demonstriert werden:

- Mehrfachauswahl (nur bei **ListBox**-Objekten verfügbar)
- Dynamische Veränderung von Listen

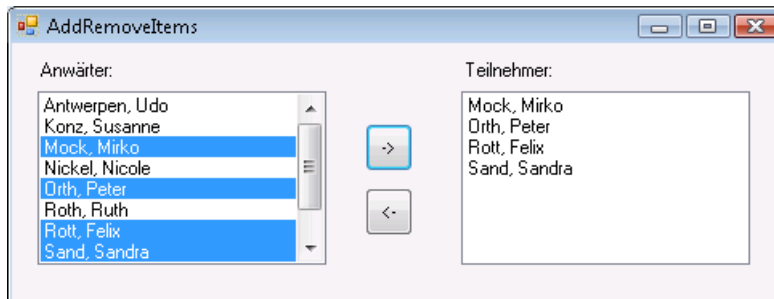
Wir erstellen und befüllen wie eben ein **ListBox**-Objekt mit Zeichenfolgen:

```
listeAnwaerter = new ListBox();
listeAnwaerter.Parent = this;
listeAnwaerter.Location = new Point(16, 30);
listeAnwaerter.Size = new Size(180, 120);
listeAnwaerter.SelectionMode = SelectionMode.MultiExtended;
listeAnwaerter.Items.AddRange(new String[10] {"Rott, Felix", "Konz, Susanne",
    "Roth, Ruth", "Schorn, Kurt", "Orth, Peter", "Antwerpen, Udo",
    "Thor, Theo", "Nickel, Nicole", "Sand, Sandra", "Mock, Mirko"});
listeAnwaerter.Sorted = true;
```

Ein zweites **ListBox**-Objekt wird ohne Inhalt angelegt:

```
listeTeilnehmer = new ListBox();
listeTeilnehmer.Parent = this;
listeTeilnehmer.Location = new Point(280, 30);
listeTeilnehmer.Size = new Size(180, 120);
listeTeilnehmer.SelectionMode = SelectionMode.MultiExtended;
```

Im Beispielprogramm können aus der linken Liste mit Anwärtern einzelne Kandidaten flexibel ausgewählt und in die rechte Liste mit Teilnehmern transportiert werden:



Aus der Teilnehmerliste können Personen auch wieder entfernt werden.

Weil die **ListBox**-Eigenschaft **SelectionMode** bei beiden Listen den Wert **MultiExtended** erhält, können auf Windows-üblich Weise (z.B. mit Hilfe der Umschalt- oder **Strg**-Taste) bequem aufeinander folgende oder auch separierte Elemente gewählt werden.

In der Ereignisroutine zu den beiden Schaltflächen (cmdRein, cmdRaus)

```
void ButtonOnClick(object sender, EventArgs e) {
    if (sender == cmdRein) {
        bool found;
        foreach (object anw in listeAnwaerter.SelectedItems) {
            found = false;
            foreach (object teiln in listeTeilnehmer.Items)
                if (anw == teiln)
                    found = true;
            if (!found)
                listeTeilnehmer.Items.Add(anw);
        }
    } else {
        while (listeTeilnehmer.SelectedItems.Count > 0)
            listeTeilnehmer.Items.Remove(listeTeilnehmer.SelectedItems[0]);
    }
}
```

werden die markierten Elemente über die **ListBox**-Eigenschaft **SelectedItems** angesprochen, die auf ein Objekt der inneren Klasse **ListBox.SelectedObjectCollection** zeigt. Offenbar zeigt **SelectedItems** große Ähnlichkeiten mit der bereits bekannten Eigenschaft **Items**, die auf eine Liste mit *allen* Elemente zeigt.

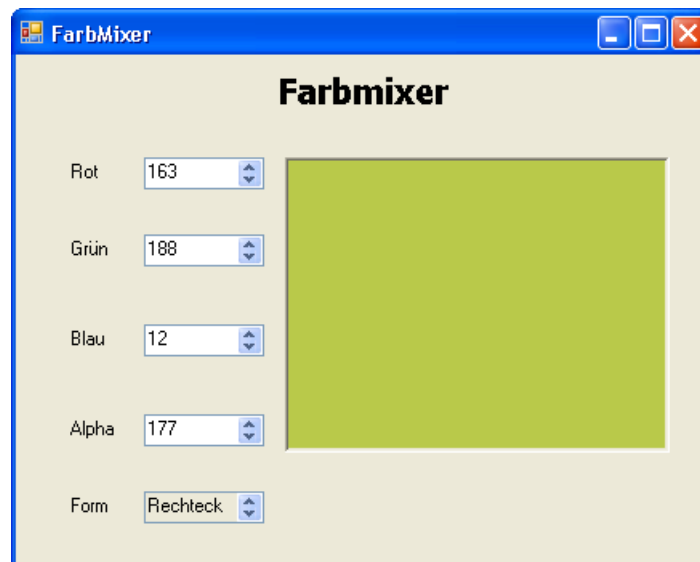
Beim Transport von links nach rechts achtet die Methode `ButtonOnClick()` darauf, dass in der Teilnehmerliste keine Dubletten entstehen.

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

...**BspUeb\WinForms\Steuerelemente\Listen\AddRemoveItems**

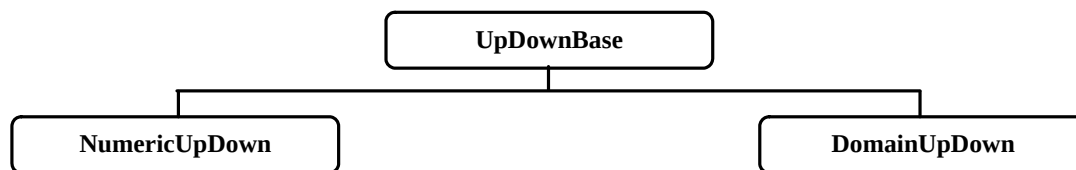
16.3 UpDown-Regler

Die auf modernen Programmoberflächen häufig anzutreffenden **UpDown**-Steuerelemente (alias: *Drehregler*, engl. *spin control*) erlauben dem Benutzer wie **ComboBox**-Objekte alternativ die Auswahl aus einer Liste oder die Eingabe von Werten, z.B.:



Es ist aber grundsätzlich nur *eine* Listenoption zu sehen, wobei man über kleine Aufwärts- und Abwärtsschalter wie bei einem vertikalen Rollbalken in Minimalausführung zwischen den Optionen wechseln kann.

In der WinForms-Bibliothek finden sich eine für numerische Werte geeignete und eine für Zeichenfolgenwerte konzipierte **UpDown**-Variante, die von der gemeinsamen (abstrakten) Basisklasse **System.Windows.Forms.UpDownBase** abstammen:



Im Beispielprogramm kann man über vier Steuerelemente der Klasse **NumericUpDown** eine Farbe definieren. Dabei sind über **NumericUpDown**-Eigenschaften festgelegt:

- **Minimum**
Als minimal erlaubter Wert bleibt die Voreinstellung 0 in Kraft.
- **Maximum**
Maximal wird der Wert 255 erlaubt.
- **Value**
Initial wird der Wert 127 angezeigt.

Der Benutzer hat folgende Möglichkeiten, das **ValueChanged**-Ereignis eines **NumericUpDown**-Reglers auszulösen:

- Er kann der Wert in- oder dekrementieren ...
 - per Mausklick auf den Aufwärts- oder Abwärtsschalter
 - mit den vertikalen Pfeiltasten bei einem **UpDown**-Regler mit Eingabefokus
- Hat die **ReadOnly**-Eigenschaft den (voreingestellten) Wert **false** hat, kann eine Zahl im zulässigen Bereich eingegeben und per **Enter**-Taste quittiert werden.

Im Beispielprogramm veranlasst die **ValueChanged**-Ereignisbehandlungsmethode per **Refresh()**-Aufruf eine sofortige Renovierung des **Panel**-Steuerelements mit der Farbprobe, so dass die registrierte **Paint**-Behandlungsmethode eine neue Farbe gemäß den Reglerständen mixt und aufträgt:

```

using System;
using System.Drawing;
using System.Windows.Forms;

class UpDown : Form {
    Panel panel;
    NumericUpDown numUpDownRot, numUpDownBlau, numUpDownGruen, numUpDownAlpha;
    DomainUpDown domainUpDown;

    UpDown() {
        . . .

        numUpDownRot = new NumericUpDown();
        numUpDownRot.Location = new Point(80, 64);
        numUpDownRot.Size = new Size(70, 20);
        numUpDownRot.Maximum = 255;
        numUpDownRot.Value = 127;
        Controls.Add(numUpDownRot);
        numUpDownRot.ValueChanged +=
            new System.EventHandler(NumericUpDownOnValueChanged);
        . . .

        domainUpDown = new DomainUpDown();
        domainUpDown.Items.Add("Ellipse");
        domainUpDown.Items.Add("Rechteck");
        domainUpDown.Location = new Point(80, 272);
        domainUpDown.Size = new Size(70, 20);
        domainUpDown.ReadOnly = true;
        domainUpDown.SelectedIndex = 0;
        Controls.Add(domainUpDown);
        domainUpDown.SelectedItemChanged +=
            new System.EventHandler(DomainUpDownOnSelectedItemChanged);
    }

    void PanelOnPaint(object sender, PaintEventArgs e) {
        Graphics g = e.Graphics;
        Panel pan = (Panel)sender;
        Brush brush = new SolidBrush(Color.FromArgb((int)numUpDownAlpha.Value,
            (int)numUpDownRot.Value, (int)numUpDownGruen.Value,
            (int)numUpDownBlau.Value));
        if (domainUpDown.SelectedIndex == 0) {
            g.FillEllipse(brush, 0, 0, pan.ClientRectangle.Width,
                pan.ClientRectangle.Height);
        } else
            g.FillRectangle(brush, 0, 0, pan.ClientRectangle.Width,
                pan.ClientRectangle.Height);
        brush.Dispose();
    }

    void NumericUpDownOnValueChanged(object sender, System.EventArgs e) {
        panel.Refresh();
    }

    void DomainUpDownOnSelectedItemChanged(object sender, System.EventArgs e) {
        panel.Refresh();
    }

    [STAThread]
    static void Main() {
        Application.EnableVisualStyles();
        Application.Run(new UpDown());
    }
}

```

Über ein **DomainUpDown**-Steuerelement erlaubt das Beispielprogramm, die Form der gefärbten Fläche zu beeinflussen. Bei der Steuerelement-Konfiguration werden folgende Eigenschaften angesprochen:

- Wie bei einem **ListBox**- oder **ComboBox**-Objekt zeigt die Eigenschaft **Items** auf eine Liste der enthaltenen Elemente. Diese Liste wird über ein Objekt der internen Klasse **DomainUpDown.DomainUpDownItemCollection** realisiert, die von **ArrayList** abstammt und die üblichen Methoden zum Verwalten der Listenelemente bietet (z.B. **Add()**, **AddRange()**, **Remove()**).
- Der **ReadOnly**-Eigenschaftswert **true** hindert die Benutzer daran, Eigenkreationen als Wert einzutippen. In anderen Einsatzfällen ist die Eingabemöglichkeit sicher sinnvoll, wobei die **Text**-Eigenschaft die eingegebene oder ausgewählte Zeichenfolge enthält.
- Über die **SelectedIndex**-Eigenschaft wird für die Anzeige eines initialen Werts gesorgt. Außerdem ist diese Eigenschaft relevant für die **Paint**-Ereignismethode des **Panel**-Steuerelements.

Um über Wertveränderungen zu informieren, bietet ein **DomainUpDown**-Objekt die Ereignisse **SelectedItemChanged** und **TextChanged** an.

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

...\BspUeb\WinForms\Steuerelemente\UpDown

16.4 Rollbalken

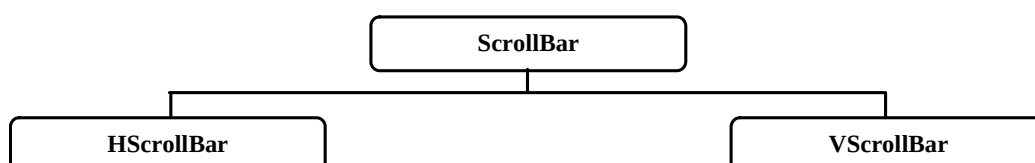
Viele Steuerelemente werden automatisch mit Rollbalken ausgestattet (z.B. **ListBox**-Objekte) oder können leicht über eine Eigenschaft (z.B. **ScrollBars** bei der Klasse **TextBox**) mit diesen Bedienelementen versorgt werden. Solche Lösungen erfüllen ihren Zweck und ersparen uns lästigen Aufwand.

Besitzt die **AutoScroll**-Eigenschaft eines **ScrollableControl**-Objekts (z.B. eines Formulars oder **Panel**-Objekts) den Wert **true**, dann ergänzt das Laufzeitsystem automatisch Bildlaufleisten, wenn die untergeordneten Steuerelemente nicht vollständig angezeigt werden können. Bei einem Beispielprogramm zur Klasse **FontFamily** haben wir uns vom **AutoScroll**-Angebot allerdings zu einer wenig performanten Lösung verleiten lassen (siehe Abschnitt 15.7.1). Dort werden die Namen aller auf dem lokalen PC installierten Schriftarten im Originallayout jeweils in einer eigenen Zeile auf ein **Panel**-Steuerelement geschrieben. Über den automatischen Bildlauf kann das gewünschte Segment des überlangen **Panel**-Steuerelements in den sichtbaren Bereich befördert werden. Dass bei jeder Rollaktion das **Panel**-Element komplett neu zu erstellen ist, erschwert allerdings bei zahlreich vorhandenen Schriftarten die Verwendung des Rollbalkens.

Wir ersetzen nun das überlange **Panel**-Objekt und die **AutoScroll**-Funktion des Formulars durch eine neue Lösung:

- Das Formular erhält ein explizites **VScrollBar**-Steuerelement.
- In der **OnPaint**-Methode des Formulars wird nur das sichtbare Segment der Schriftartenliste geschrieben.

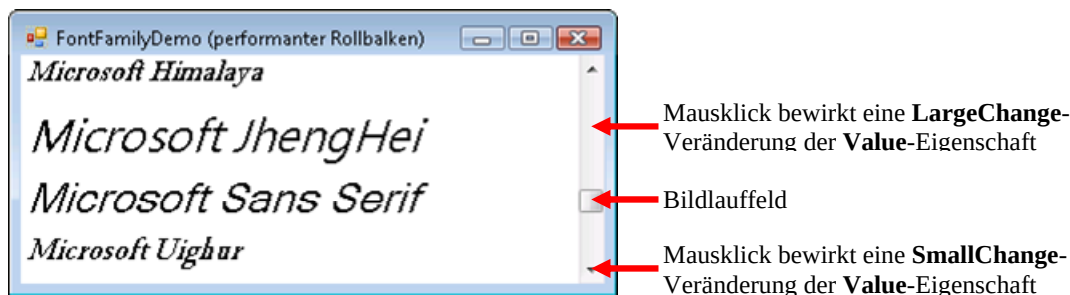
Die Klassen **VScrollBar** und **HScrollBar** bieten über ihre Eigenschaften und Ereignisse erheblich mehr Flexibilität als eine **AutoScroll**-Lösung. Sie stammen von der gemeinsamen abstrakten Basisklasse **System.Windows.Forms.ScrollBar** ab:



Zwar kann man die **ScrollBar**-Abkömmlinge frei positionieren, dimensionieren und färben, doch wird man sie in der Regel mit dem voreingestellten Erscheinungsbild per **DockStyle**-Wert am passenden Fensterrand anbringen. Im Fensterkonstruktor des performanzoptimierten **FontFamily**-Demonstrationsprogramms wird eine vertikale Bildlaufleiste am rechten Fensterrand angedockt:

```
scrollBar = new VScrollBar();
scrollBar.Parent = this;
scrollBar.Dock = DockStyle.Right;
scrollBar.Minimum = 0;
scrollBar.Maximum = vSpace;
scrollBar.SmallChange = vStep;
scrollBar.LargeChange = vStep * 5;
scrollBar.Scroll += new ScrollEventHandler(VScrollBarOnScroll);
```

Über die **ScrollBar**-Eigenschaften **Minimum** und **Maximum** ordnet man den extremen Bildlaufpositionen jeweils einen Wert zu. Im Beispiel wird das Minimum auf Null und das **Maximum** auf die Gesamthöhe der untereinander geschriebenen Textproben gesetzt. Die aktuelle Position des Bildlauffelds auf dem Rollbalken ist per **Value**-Eigenschaft zu ermitteln und auch zu modifizieren. Welche Wertveränderungen der Benutzer bei einem Klick auf einen der Pfeile am Ende des Rollbalkens bzw. auf eine Balkenzone zwischen Bildlauffeld und Pfeil bewirkt, legt man über die **SmallChange**- bzw. **LargeChange**-Eigenschaft fest.



Die maximale **Value**-Ausprägung beträgt:

Maximum - LargeChange + 1

Von einer Verschiebung des Bildlauffelds kann man sich über die Ereignisse **ValueChanged** und **Scroll** informieren lassen. Das **ValueChanged**-Ereignis tritt ein, wenn die **Value**-Eigenschaft entweder durch den Benutzer oder per Programm geändert wird. Eine mögliche Behandlung besteht im **FontFamily**-Beispiel darin, per **Refresh()**-Aufruf die **OnPaint**-Methode des Formulars dazu zu bringen, die sichtbaren Schriftproben neu zu malen:

```
scrollBar.ValueChanged += new EventHandler(VScrollBarOnValueChanged);

void VScrollBarOnValueChanged(object sender, EventArgs e) {
    Refresh();
}
```

Das **Scroll**-Ereignis tritt ein, wenn das Bildlauffeld durch eine Maus- oder Tastaturaktion verschoben wurde, jedoch *nicht* bei einer **Value**-Veränderung durch das Programm. Der **Scroll**-Ereignisbehandlungsmethode wird beim Aufruf ein informatives **ScrollEventArgs**-Objekt übergeben. Seiner **Type**-Eigenschaft ist zu entnehmen, ob die **Value**-Veränderung im Rahmen einer noch nicht abgeschlossenen Verschiebungsaktion geschah. In diesem Fall kann man aus Performanzgründen auf eine Ereignisbehandlung verzichten, z.B.:

```
scrollBar.Scroll += new ScrollEventHandler(VScrollBarOnScroll);

void VScrollBarOnScroll(object sender, ScrollEventArgs e) {
    if (e.Type == ScrollEventType.EndScroll)
        Refresh();
}
```

Es empfiehlt sich, auch den Bildlauf über das **Scroll-Rad** der Maus zu unterstützen. Dazu ist lediglich eine simple Ereignismethode zu erstellen

```
void FormOnMouseWheel(object sender, MouseEventArgs e) {
    if (e.Delta > 0)
        scrollBar.Value = Math.Max(scrollBar.Minimum,
                                   scrollBar.Value + scrollBar.SmallChange);
    else
        scrollBar.Value = Math.Min(scrollBar.Maximum - scrollBar.LargeChange + 1,
                                   scrollBar.Value + scrollBar.SmallChange);
    Refresh();
}
```

und beim **MouseWheel**-Ereignis des Formulars zu registrieren:

```
MouseWheel += new EventHandler(FormOnMouseWheel);
```

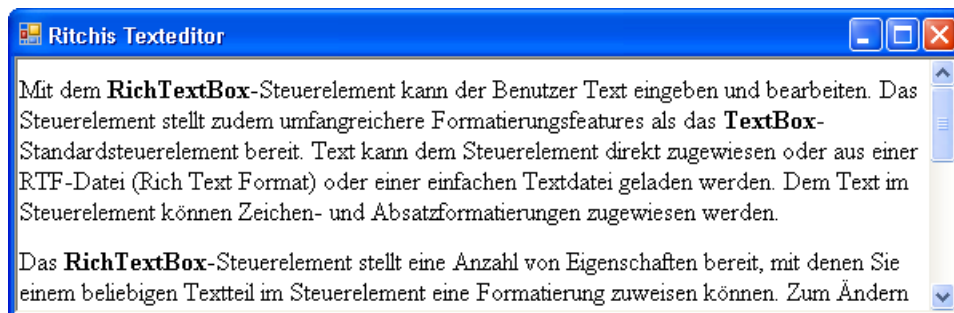
Über die **MouseEventArgs**-Eigenschaft **Delta** erfahren wir, ob der Benutzer das Mausexplorer vor (**Delta** = 120) oder zurück gedreht hat (**Delta** = -120).

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

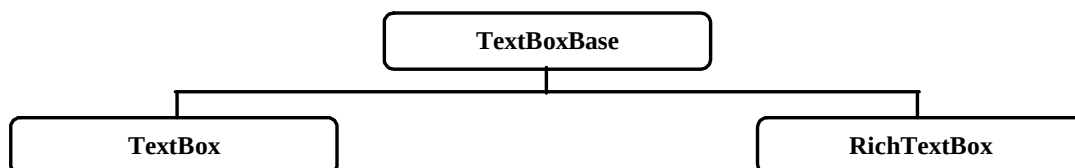
...\BspUeb\WinForms\Steuerelemente\ScrollBar

16.5 Ein RTF-Editor auf RichTextBox-Basis

Das **RichTextBox**-Steuerelement implementiert einen brauchbaren Texteditor, der sogar das RTF-Format (*Rich Text Format*) beherrscht:



Es stammt wie das bereits in Abschnitt 9.4.3 vorgestellte **TextBox**-Steuerelement von der Klasse **TextBoxBase** ab:



Für eine Rohversion des **RichTextBox**-Editors, die immerhin z.B. schon den Datenaustausch via Zwischenablage und eine mehrstufige Un(Re)do-Funktion beherrscht, muss man erstaunlich wenig Aufwand betreiben:

```
using System;
using System.Windows.Forms;

class RichTextBoxEinfach : Form {
    RichTextBoxEinfach() {
        Text = "Ritchis Texteditor (einfach)";
        RichTextBox editor = new RichTextBox();
        editor.Dock = DockStyle.Fill;
        editor.AcceptsTab = true;
        Controls.Add(editor);
    }
}
```

```

[STAThread]
static void Main() {
    Application.Run(new RichTextBoxEinfach());
}

```

Mit dem Wert **DockStyle.Fill** für die **Dock**-Eigenschaft wird dafür gesorgt, dass die **RichTextBox**-Komponente den gesamten Klientenbereich des Formulars belegt.

Mit dem **AcceptsTab**-Wert **true** sorgen wir dafür, dass die **RichTextBox**-Komponente Tabulatorzeichen per Tastatur entgegen nehmen. Dies klappt per Voreinstellung *nicht*, weil die Tabulatorteste von der übergeordneten **ContainerControl**-Komponente zur Fokusverwaltung verwendet wird (vgl. Abschnitt 9.2.1).

Das Einfüllen des obigen Beispieltexes (mit Auszeichnungen) gelingt vorläufig nur über die Zwischenablage. Mit etwas Detailarbeit lässt sich auf der **RichTextBox**-Grundlage jedoch eine zur Windows-Zugabe WordPad äquivalente Anwendung erstellen. Wir werden die Funktionalität des Editors in späteren Abschnitten erheblich ausbauen.

16.6 Timer-Komponente

Über die in Abschnitt 13.7 vorgestellte Klasse **Timer** im Namensraum **System.Threading** kann man die CLR beauftragen, eine Methode regelmäßig in einem eigenen Thread auszuführen. Es ist aber ein gewisser Aufwand zu treiben, wenn Arbeitsergebnisse eines solchen Threads sich auf die Steuerelemente der Benutzeroberfläche auswirken sollen.

Im Namensraum **System.Windows.Forms** ist ebenfalls eine Klasse namens **Timer** vorhanden, die als (nicht-visuelle) Komponente ereignisorientiert arbeitet und sehr bequem in eine GUI-Anwendung einzubinden ist. Die per **Timer**-Komponente angestoßenen Methodenaufrufe laufen jedoch im GUI-Thread ab, so dass nur Methoden mit geringem Rechenzeitbedarf in Frage kommen, damit die Pogrammoberfläche bedienbar bleibt (vgl. Abschnitt 13.7).

Die **Timer**-Komponente produziert mit einstellbarer Regelmäßigkeit **Tick**-Ereignisse und bringt somit die bei diesem Ereignis registrierten Methoden ins Spiel. Allerdings kann man sich auf die exakte Taktung *nicht* verlassen, weil eine per Timer ausgelöste Ereignisbearbeitung eine bereits laufende Ereignisverarbeitung *nicht* unterbricht. Mit einem **int**-Wert für die **Timer**-Eigenschaft **Intervall** legt man den gewünschten Abstand zwischen zwei **Tick**-Ereignissen in Millisekunden fest. Die CLR verhindert, dass *mehrere* unerledigte **Tick**-Ereignisse in der Warteschlange auflaufen.

Im folgenden Beispielprogramm mit Editor (vgl. Abschnitt 16.5) wird für eine per Standardkonstruktor erzeugte **Timer**-Komponente das Auslöseintervall auf 1000 Millisekunden gesetzt und eine Behandlungsmethode registriert, welche die aktuelle Zeit in der Titelzeile anzeigt:

```

using System;
using System.Windows.Forms;

class TimerKomponente : Form {
    TimerKomponente() {
        RichTextBox editor = new RichTextBox();
        editor.Dock = DockStyle.Fill;
        Controls.Add(editor);

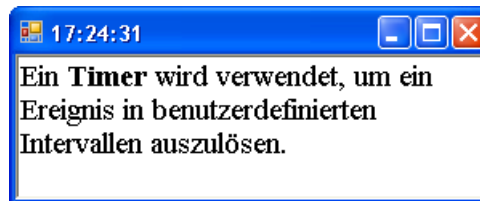
        Timer tim = new Timer();
        tim.Interval = 1000;
        tim.Tick += new EventHandler(TimerOnTick);
        tim.Start();
    }
}

```

```
void TimerOnTick(object sender, EventArgs e) {  
    Text = DateTime.Now.ToLongTimeString();  
}  
  
[STAThread]  
static void Main() {  
    Application.Run(new TimerKomponente());  
}  
}
```

Damit ein **Timer** tatsächlich damit beginnt, Ereignisse zu feuern, muss seine **Start()**-Methode aufgerufen oder seine **Enabled**-Eigenschaft auf den Wert **true** gesetzt werden. Soll er eine Pause einlegen, ruft man seine **Stop()**-Methode auf oder setzt seine **Enabled**-Eigenschaft auf den Wert **false**.

Die aktuelle Zeit liefert im Beispiel eine Instanz der Struktur **DateTime**, welche über die statische Eigenschaft **Now** dieser Struktur angesprochen wird. Zur benutzerfreundlichen Zeitanzeige wird im Beispielprogramm die **DateTime**-Instanzmethode **ToLongTimeString()** verwendet:



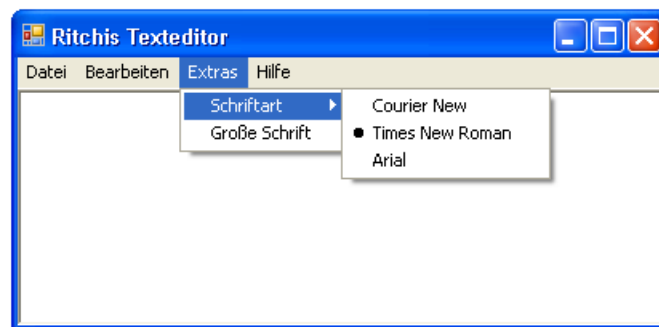
Wir haben die **Now**-Instanz der **DateTime**-Struktur schon mehrfach zur Zeitmessung verwendet und dabei per **Ticks**-Eigenschaft die Anzahl der 100-Nanosekunden-Intervalle seit dem 1. Januar 1, 00:00:00 Uhr, abgerufen (10^9 Nanosekunden = 1 Sekunde).

17 Menüs

Wir erweitern das in Abschnitt 16.5 begonnene Editor-Projekt um ein Menü:



Dabei werden die mit .NET 2.0 hinzu gekommenen WinForms-Klassen **MenuStrip**, **ContextMenuStrip** und **ToolStripMenuItem** verwendet, die im Vergleich zur Vorgängerlösung aus .NET 1.x



eine erweiterte Funktionalität bieten (insbesondere bei der Positionierung). Wer die traditionelle Lösung bevorzugt, findet eine Beschreibung in Abschnitt 0.

17.1 Wichtige Klassen und Begriffe

Das Hauptmenü (die Menüleiste) eines Formulars realisiert man durch ein Objekt der Klasse **MenuStrip**, das in der Regel am oberen Containerrand angedockt wird:

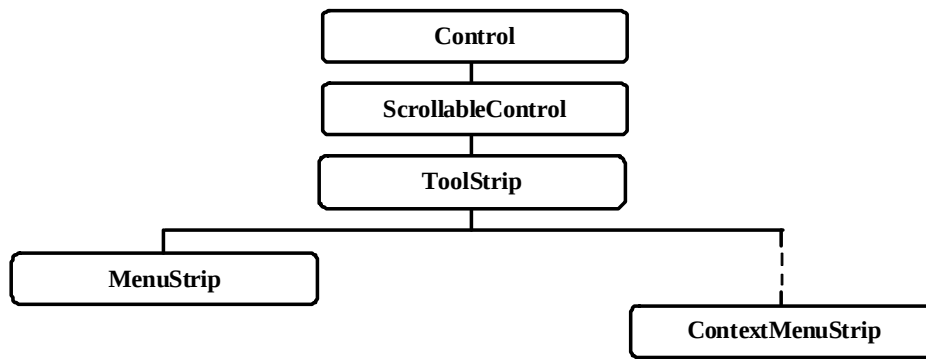
```
MenuStrip mainMenu = new MenuStrip();
mainMenu.Dock = DockStyle.Top;
Controls.Add(mainMenu);
```

Mit den folgenden Anweisungen werden die **ToolStripMenuItem**-Komponenten mitFile, mitEdit, mitTools und mitHelp erzeugt und in das Hauptmenü eingefügt:

```
ToolStripMenuItem mitFile, mitTools, mitHelp;

mitFile = new ToolStripMenuItem("&Datei");
mitEdit = new ToolStripMenuItem("&Bearbeiten");
mitTools = new ToolStripMenuItem("&Extras");
mitHelp = new ToolStripMenuItem("&Hilfe");
mainMenu.Items.AddRange(new ToolStripItem[]
    {mitFile , mitEdit, mitTools, mitHelp});
```

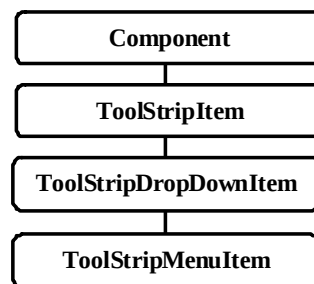
Vor der weiteren Beschäftigung mit dem Quellcode des Beispiels sollten wir uns um einige Erbschaftsangelegenheiten kümmern und wichtige Begriffe klären. Die Klasse **MenuStrip** stammt wie die später zur Realisation eines Kontextmenüs (siehe Abschnitt 17.5) verwendete Klasse **ContextMenuStrip** von der für Symbolleisten zuständigen Klasse **ToolStrip** ab, so dass wir eigentlich die Symbolleisten vor den Menüs behandeln sollten:



Bei einem klassischen Windows-Programm sind aber Menüs elementarer und für die Bedienung wichtiger als die (in Kapitel 20 behandelten) Symbolleisten.¹

Von ihrer gemeinsamen Basisklasse **ToolStrip** erben **MenuStrip** und **ContextMenuStrip** u.a. die Eigenschaft **Items**, die auf eine Liste mit **ToolStripItem**-Objekten zeigt. Diese Liste (ein Objekt aus der Klasse **ToolStripItemCollection**) verfügt u.a. über die Methoden **Add()** und **AddRange()** zur Aufnahme von Items. Offenbar ist die **Items**-Eigenschaft der Klasse **ToolStrip** ähnlich einzusetzen wie die **Controls**-Eigenschaft der Klasse **Control**.

Die Klasse **ToolStripMenuItem** stammt *nicht* von **Control** ab, verfügt aber über eine ähnliche Ausstattung mit Eigenschaften (z.B. **BackColor**, **ForeColor**, **Font**) und Ereignissen (z.B. **Click**, **Paint**, **MouseDown**):



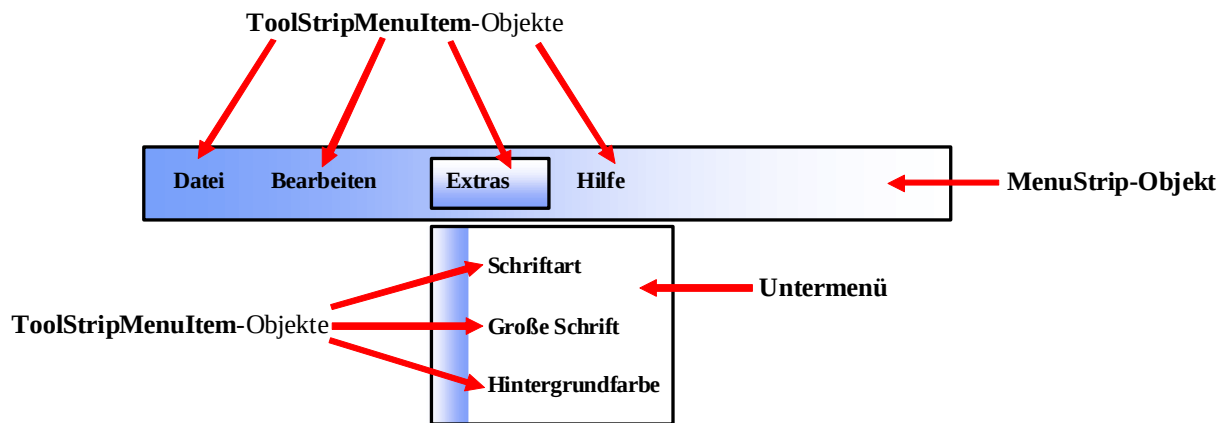
Ein Objekt der Klasse **ToolStripMenuItem** kann eine Liste mit untergeordneten Objekten derselben Klasse enthalten, wobei ...

- in der Klasse **ToolStripMenuItem** eine Eigenschaft **DropDownItems** vorhanden ist,
- die auf ein Objekt der Klasse **ToolStripItemCollection** zeigt,
- das eine Liste mit **ToolStripItem**-Objekten enthält.

Also kann sowohl **MenuStrip** als auch **ToolStripMenuItem** eine Kollektion von untergeordneten **ToolStripMenuItem**-Objekten enthalten, wobei die zur Ansprache verfügbaren Eigenschaften unterschiedliche Namen tragen (**Items** bzw. **DropDownItems**).

Wird ein **ToolStripMenuItem** mit gefüllter Liste vom Benutzer ausgewählt (z.B. per Maus), dann erscheint ein rechteckiges Fenster mit einem **Untermenü**, z.B.:

¹ Neuerdings zeigt die Firma Microsoft allerdings (z.B. beim Internet-Explorer und bei Windows-Vista) die Tendenz, das Hauptmenü nur noch als Zusatzoption für traditionsbewusste Anwender zu betrachten.



Man kann das Untermenü aufgrund seines Verhaltens beim Auftritt als **DropDown-Menü** bezeichnen, was ja auch in Typ- und Eigenschaftsnamen zum Ausdruck kommt (siehe oben). Es existiert keine Klasse für Untermenüs, weil *jedes* **ToolStripMenuItem**-Objekt eine Kollektion untergeordneter Objekte desselben Typs (und damit ein Untermenü) aufnehmen kann.

Neben gewöhnlichen Items (**ToolStripMenuItem**-Objekten) kann ein Menü (**MenuStrip**-Objekt) auch Objekte aus den Klassen **ToolStripTextBox** und **ToolStripComboBox** aufnehmen. Bei diesen selten genutzten Klassen handelt es sich um Varianten der Klassen **TextBox** und **ComboBox** (siehe Abschnitt 9.4.3 bzw. 16.2) mit Optimierungen für den Einsatz in einem Menü.

17.2 Menüitems erzeugen und konfigurieren

Soll ein **ToolStripMenuItem**-Objekt ein **Untermenü** aufnehmen, eignet sich ein Konstruktoraufwurf mit **String**-Parameter für den anzuzeigenden Text, z.B.:

```
mitFile = new ToolStripMenuItem("&Datei");
```

Allmählich wird es Zeit, die Bedeutung des (optionalen) &-Zeichens in den **ToolStripMenuItem**-Beschriftungen zu erklären. Es legt eine **Zugriffstaste** fest, die im laufenden Programm ...

- (spätestens nach Betätigen der **Alt**-Taste) durch Unterstreichen hervorgehoben wird,



- das **Click**-Ereignis zum Menüitem auslöst, wenn sie nach der **Alt**-Vorschalttaste gedrückt wird, wobei ggf. das zugehörige Untermenü erscheint.

Um das „&“-Zeichen als gewöhnlichen Bestandteil der Itembeschriftung zu verwenden, schreibt man es doppelt.

Steht ein **ToolStripMenuItem**-Objekt für einen **Menübefehl**, dann ist ein alternativer Konstruktor zu verwenden und neben der Beschriftung auch eine **Click**-Ereignisbehandlungsmethode anzugeben, z.B.:

```
mitInfo = new ToolStripMenuItem("&Info", null,
    new EventHandler(MitInfoOnClick));
mitHelp.DropDownItems.Add(mitInfo);
```

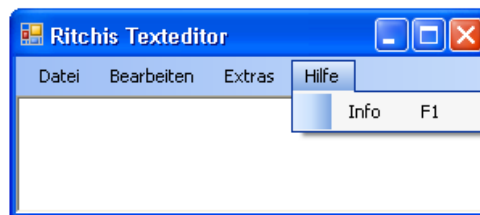
Im zweiten Parameter der gewählten Konstruktor-Überladung kann man ein Symbol zum Menüitem angeben (siehe unten).

Über die Eigenschaft **ShortcutKeys** lässt sich eine Tastenkombination zum Auslösen des Menübefehls vereinbaren, wobei die Enumeration **Keys** eine reichhaltige Auswahl an Bezeichnungen für Tasten(kombinationen) bietet, z.B.:

```
mitInfo.ShortcutKeys = Keys.F1;
```

In Beispiel hat der Benutzer also zwei Möglichkeiten, die Methode `MitInfoOnClick()` per Tastatur aufzurufen:

- **Alt, H, I**
Alt aktiviert die Tastatursteuerung des Menüs, **H** ruft das Untermenü zum Hauptmenüitem **Hilfe** auf und **I** startet den Ereignishandler zum Untermenüitem **Info**.
- **F1**
Die Taste(nkombination) zu einem Item erscheint im aufgeklappten Untermenü, sofern die **ToolStripMenuItem**-Eigenschaft **ShowShortcutKeys** den voreingestellten Wert **true** besitzt, z.B.



Um eine Taste(nkombination) „ohne“ zugehöriges Menüitem mit einer Behandlungsmethode zu verbinden, setzt man die **Visible**-Eigenschaft des Menüitems auf den Wert **false**, z.B.:

```
mitInfo.Visible = false;
```

Weil ein Menüitem wiederum Menüitems aufnehmen kann, erstellt man leicht ein **mehrstufiges Menü**. Im folgenden Segment des Fensterkonstruktors zum Beispielprogramm wird zunächst ein Menüitem namens `mitFont` erzeugt und dem Hauptmenüitem `mitTools` untergeordnet:

```
ToolStripMenuItem mitTools,
    mitFont, mitCourierNew, mitTimesNewRoman, mitArial, mitCurrentFont;

mitFont = new ToolStripMenuItem("&Schriftart");
mitTools.DropDownItems.Add(mitFont);
```

Dann entstehen die Menüitems `mitCourierNew`, `mitTimesNewRoman` und `mitArial`, die in der **DropDownItems**-Kollektion von `mitFont` landen:

```
EventHandler fontHandler = new EventHandler(MitFontOnClick);

mitCourierNew = new ToolStripMenuItem("&Courier New", null, fontHandler);
mitFont.DropDownItems.Add(mitCourierNew);

mitTimesNewRoman = new ToolStripMenuItem("&Times New Roman", null,
    fontHandler);
mitTimesNewRoman.Checked = true;
mitFont.DropDownItems.Add(mitTimesNewRoman);

mitArial = new ToolStripMenuItem("&Arial", null, fontHandler);
mitFont.DropDownItems.Add(mitArial);
```

Ein vorhandenes Untermenü wird im laufenden Programm automatisch durch einen nach rechts gerichteten Pfeil angezeigt:



Mit der Eigenschaft **Checked** (Voreinstellung: **false**) wird die Aktivierung eines Items ermittelt oder festgelegt, z.B. bei `mitTimesNewRoman` (siehe unten).

Über ein **ToolStripSeparator**-Objekt fügt man eine horizontale Trennlinie in ein Untermenü ein, z.B.:

```
mitTools.DropDownItems.Add(new ToolStripSeparator());
```

Im Beispielprogramm stehen die drei Items im Untermenü **Extras > Schriftart** für eine Gruppe sich gegenseitig ausschließender Optionen und verwenden denselben **Click**-Handler:

```
protected void MitFontOnClick(object sender, EventArgs e) {
    mitCurrentFont.Checked = false;
    mitCurrentFont = (ToolStripMenuItem)sender;
    mitCurrentFont.Checked = true;

    if (sender == mitCourierNew) {
        editor.Font = (mitSize.Checked ? fontCN16 : fontCN12);
    } else
    if (sender == mitTimesNewRoman) {
        editor.Font = (mitSize.Checked ? fontTNR16 : fontTNR12);
    } else {
        editor.Font = (mitSize.Checked ? fontA16 : fontA12);
    }
}
```

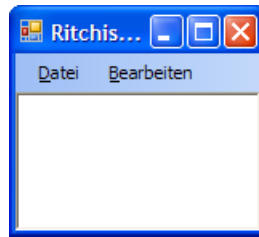
Das **RichTextBox**-Steuerelement erhält die gewünschte Schriftart, wobei ein eventueller Benutzerwunsch nach großen Schriften zu berücksichtigen ist, was unter Verwendung des Konditionaloperators geschieht.

Mit den ersten Anweisungen der Methode `MitFontOnClick()` wird für eine korrekte Aktivierung des Menüitems zur aktuell gewählten Schrift über die Eigenschaft **Checked** gesorgt. Leider existiert für Items mit Optionsfeld-Logik keine Gruppenbildung mit dem Effekt, dass genau *ein* Item aktiviert ist. Im Beispielprogramm sorgt folgende Technik (übernommen von Petzold 2002, S. 572ff) für stets korrekte Markierungen:

- Die **ToolStripMenuItem**-Referenz `mitCurrentFont` zeigt stets auf das aktuell aktivierte Schriftarten-Menüitem.
- In der Methode `MitFontOnClick()` wird zunächst die aktuelle Aktivierung aufgehoben. Dann erhält `mitCurrentFont` die Adresse des vom Benutzer gewählten Items, und schließlich wird dieses Item aktiviert.

Über die in Abschnitt 17.3 vorzustellenden Symbole zu Menüitems könnte man auch die Optionsfeld-*Optik* herstellen. Dieser Aufwand lohnt sich aber kaum, weil die Benutzer mittlerweile an Häkchen als Ersatz für Radioknöpfe gewöhnt sind.

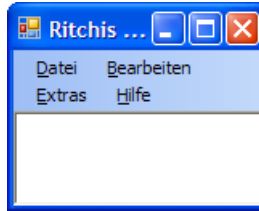
Bei unzureichender Formularbreite verschwinden per Voreinstellung Items, die in der Menüleiste keinen Platz mehr finden, z.B.:



Grundlage für dieses Verhalten ist die Ausprägung **ToolStripLayoutStyle.HorizontalStackWithOverflow** der **ToolStrip**-Eigenschaft **LayoutStyle**. Mit der alternativen Ausprägung **Flow**

```
mainMenu.LayoutStyle = ToolStripLayoutStyle.Flow;
```

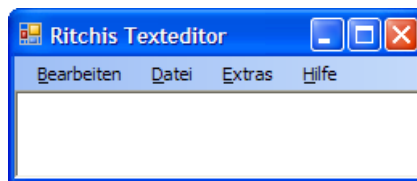
erhält man bei horizontalem Platzmangel einen „Zeilenumbruch“:



Über die **ToolStrip**-Eigenschaft **AllowItemReorder**

```
mainMenu.AllowItemReorder = true;
```

kann man dem Benutzer erlauben, bei gedrückter **Alt**-Taste per Maus die Reihenfolge der Menüitems zu verändern, z.B.:



Allerdings sollte man die individuelle Anordnung mit den in Abschnitt 19 behandelten Techniken auch zwischen den Sitzungen speichern

17.3 Symbole für Menüitems

Beim **Bearbeiten**-Untermenü mit den Items

```
ToolStripMenuItem mitEdit, mitCut, mitCopy, mitPaste, mitDel;
```

sind die Anwender daran gewöhnt, dass in dem senkrechten Streifen am linken Rand des Untermenüs Symbole zu den Items erscheinen:



Man kann diese Symbole folgendermaßen realisieren:

- Bitmap-Dateien mit einer (16 × 16)-Pixelmatrix erstellen
Wer sich nicht selbst künstlerisch betätigen will, findet geeignete Bitmap-Dateien im Zubehör vieler Entwicklungsumgebungen oder im Internet. Anwender der Microsoft Visual Stu-

dio C# 2008 Express Edition haben nach der Registrierung (z.B. über das Item **Produkt registrieren** im Hilfemenü der Entwicklungsumgebung) Zugang zum *Visual Studio 2005¹ Registration Benefits Portal*. Von dort kann mit der *IconBuffet Studio Edition Icon Suite* eine Sammlung mit zahlreichen professionellen und lizenzfreien Symbolen bezogen werden.

- **Bitmap**-Objekte erstellen und eine Transparentfarbe festlegen, z.B.:

```
Bitmap bmp = new Bitmap("cut.bmp");
bmp.MakeTransparent(bmp.GetPixel(0, 0));
```

- In den **ToolStripMenuItem**-Konstruktoren das **Bitmap**-Objekt als Parameter übergeben, z.B.:

```
mitCut = new ToolStripMenuItem("&Ausschneiden", bmp, MitCutOnClick);
```

Dem als Beispiel verwendeten Menüitem **Bearbeiten > Ausschneiden** wird noch die übliche Tastenkombination zugeordnet:

```
mitCut.ShortcutKeys = Keys.Control | Keys.X;
```

Analog werden auch die restlichen Items im **Bearbeiten**-Untermenü des Beispielsprogramms erzeugt:

```
bmp = new Bitmap("copy.bmp");
bmp.MakeTransparent(bmp.GetPixel(0, 0));
mitCopy = new ToolStripMenuItem("&Kopieren", bmp, MitCopyOnClick);
mitCopy.ShortcutKeys = Keys.Control | Keys.C;
bmp = new Bitmap("paste.bmp");
bmp.MakeTransparent(bmp.GetPixel(0, 0));
mitPaste = new ToolStripMenuItem("&Einfügen", bmp, MitPasteOnClick);
mitPaste.ShortcutKeys = Keys.Control | Keys.V;
bmp = new Bitmap("del.bmp");
bmp.MakeTransparent(bmp.GetPixel(0, 0));
mitDel = new ToolStripMenuItem("&Löschen", bmp, MitDeleteOnClick);
mitDel.ShortcutKeys = Keys.Delete;
```

Schließlich wird das per **DropDownItems**-Eigenschaft ansprechbare **ToolStripItemCollection**-Objekt im Menüitem `mitEdit` über die Methode **AddRange()** beauftragt, die neu erstellten Items aufzunehmen:

```
mitEdit.DropDownItems.AddRange(new ToolStripItem[]
    {mitCut, mitCopy, mitPaste, mitDel});
```

In den **Click**-Ereignisbehandlungsmethoden zu den **Bearbeiten**-Menüitems kann man auf die **RichTextBox**-Funktionalität zurückgreifen, z.B.:

```
protected void MitCutOnClick(object sender, EventArgs e) {
    editor.Cut();
}
protected void MitDeleteOnClick(object sender, EventArgs e) {
    if (editor.SelectionLength==0 && editor.SelectionStart < editor.TextLength)
        editor.Select(editor.SelectionStart, 1);
    editor.SelectedText = "";
}
```

Indem man die **RichTextBox**-Eigenschaft **SelectedText** auf die leere Zeichenfolge setzt,

```
editor.SelectedText = "";
```

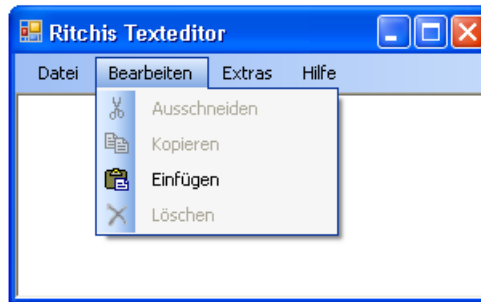
löscht man den ausgewählten Text.

Im Rahmen einer Übungsaufgabe (siehe Abschnitt 17.8) sollen Sie das **Bearbeiten**-Menü unseres Editors noch um einige Funktionen erweitern (Rückgängig, Wiederholen, Alles markieren).

¹ Dies ist kein Tippfehler. Ein *Visual Studio 2008 Registration Benefits Portal* scheint es nicht zu geben.

17.4 Verfügbarkeit von Menüitems dynamisch anpassen

Ist ein Menüitem momentan nicht nutzbar (z.B. **Bearbeiten > Kopieren** ohne markierten Text), dann sollte seine **Enabled**-Eigenschaft auf **false** gesetzt werden. Es wird dann abgeblendet dargestellt, und sein **Click**-Ereignis ist nicht auslösbar, z.B.:



Über das **ToolStripMenuItem**-Ereignis **DropDownOpening**, das vor dem Öffnen eines Untermenüs eintritt, kann man für eine rechtzeitige **Enabled**-Aktualisierung sorgen, z.B. bei unserem **Bearbeiten**-Untermenü:

```
protected void MitEditOnDropDownOpening(object sender, EventArgs e) {
    if (editor.SelectionLength > 0) {
        mitCut.Enabled = true;
        mitCopy.Enabled = true;
        mitDel.Enabled = true;
    } else {
        mitCut.Enabled = false;
        mitCopy.Enabled = false;
        mitDel.Enabled = false;
    }
    if (Clipboard.ContainsText()) {
        mitPaste.Enabled = true;
    } else {
        mitPaste.Enabled = false;
    }
}
```

Über die Methode **ContainsText()** der Klasse **Clipboard** sorgen wir dafür, dass unser Editor ausschließlich Text (also z.B. keine Bilder) aus der Windows-Zwischenablage übernimmt. Bei der Kooperation mit der Zwischenablage kommt unser Programm mit der COM-Technologie von Windows in Kontakt, so dass nach Abschnitt 11.1 die **Main()**-Methode das Attribut **STAThread** benötigt:

```
[STAThread]
static void Main() {
    Application.EnableVisualStyles();
    Application.Run(new RitchisTexteditor());
}
```

Wir dekorieren seit Abschnitt 11.1 einer Anweisung von Microsoft folgend in allen WinForms-Programmen die **Main()**-Methode mit diesem Attribut, obwohl bei vielen Programmen eine Unterlassung keine erkennbaren Folgen hat. Beim Einsatz der Klasse **Clipboard** ist das Attribut aber unverzichtbar.

Die Methode zur Verfügbarkeits-Aktualisierung der **Bearbeiten**-Menüitems wird beim **DropDownOpening**-Ereignis des Hauptmenü-Items **mitEdit** registriert:

```
mitEdit.DropDownOpening += MitEditOnDropDownOpening;
```

Mit Hilfe der **ToolStripItemCollection**-Methoden zum Einfügen oder Entfernen von Menüitems (**Add()**, **AddRange()**, **Clear()**, **Remove()** etc.) kann man in einem **DropDownOpening**-Handler die Liste der Menüitems kontextspezifisch anpassen.

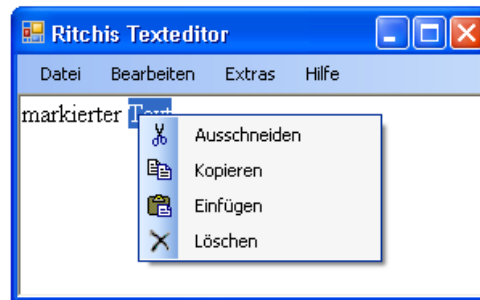
Über das **DropDownOpened**-Ereignis kann man z.B. dafür sorgen, dass zum gerade aktivierten Menüitem eine Erläuterung in der Statuszeile erscheint (siehe unten).

Den vollständigen Quellcode zum aktuellen Entwicklungsstand des Beispielprogramms finden Sie im Ordner

...\BspUeb\WinForms\Menüs\DotNet 2\Hauptmenü

17.5 Kontextmenü

Unser Editor (genauer: seine **RichTextBox**-Komponente) sollte noch ein Kontextmenü mit den Bedienoptionen des **Bearbeiten**-Menüs erhalten, z.B.



Bei der Erstellung des Kontextmenüs zu einem Steuerelement kann man genauso vorgehen wie bei der Erstellung eines Hauptmenüs, wobei aber ein Container aus der Klasse **ContextMenuStrip** zu verwenden ist, z.B.:

```
ContextMenuStrip contextMenu = new ContextMenuStrip();
```

Weil ein **ToolStripMenuItem** nicht zu zwei Containern gehören kann, benötigen wir im Beispiel zu jeder **Bearbeiten**-Funktion zwei **ToolStripMenuItem**-Objekte, eines für das Hauptmenü und eines für das Kontextmenü:

```
ToolStripMenuItem mitCut, mitCopy, mitPaste, mitDel,
                  cmitCut, cmitCopy, cmitPaste, cmitDel;
```

Das mit **ToolStripMenuItem**-Objekten ausgestattete Kontextmenü

```
contextMenu.Items.AddRange(
    new ToolStripMenuItem[] {cmitCut, cmitCopy, cmitPaste, cmitDel});
```

wird der **ContextMenuStrip**-Eigenschaft des Steuerelements zugewiesen, z.B.:

```
editor.ContextMenuStrip = contextMenu;
```

Über das **ContextMenuStrip**-Ereignis **Opening**, das vor dem Öffnen des Kontextmenüs eintritt, kann man die **Enabled**-Eigenschaften der enthaltenen Menüitems rechtzeitig aktualisieren oder das Kontextmenü passend zusammenstellen, z.B.:

```
contextMenu.Opening += ContextMenuOnOpening;
.
.
protected void ContextMenuOnOpening(object sender, CancelEventArgs e) {
    if (editor.SelectionLength > 0) {
        cmitCut.Enabled = true;
        cmitCopy.Enabled = true;
        cmitDel.Enabled = true;
    } else {
        cmitCut.Enabled = false;
        cmitCopy.Enabled = false;
        cmitDel.Enabled = false;
    }
}
```

Über die **Cancel**-Eigenschaft des als Aktualparameter übergebenen **CancelEventArgs**-Objekts lässt sich der Auftritt des Kontextmenüs blockieren. Wenn das Kontextmenü unseres Editors nur bei vorhandener Textmarkierung sinnvoll wäre, könnten wir einen nutzlosen Auftritt so verhindern:

```
protected void ContextMenuOnOpening(object sender, CancelEventArgs e) {
    if (editor.SelectionLength == 0)
        e.Cancel = true;
}
```

Mit Hilfe der **ContextMenuStrip**-Methode **Show()** kann man ein Kontextmenü, das mit keinem Steuerelement verbunden sein muss, programmgesteuert an einer frei wählbaren Bildschirmposition auftauchen zu lassen.

Den vollständigen Quellcode zum aktuellen Entwicklungsstand des Beispielprogramms finden Sie im Ordner

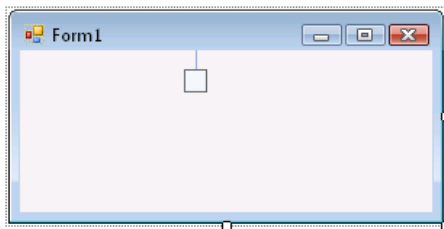
...\BspUeb\WinForms\Menüs\DotNet 2\Kontextmenü

17.6 Unterstützung durch die Entwicklungsumgebungen

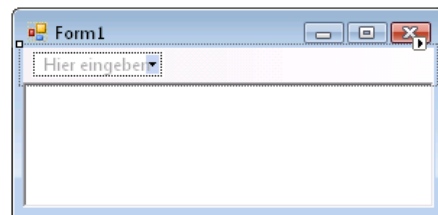
Um die exzellente Unterstützung von Microsofts Entwicklungsumgebungen (Visual C# Express Edition bzw. Visual Studio 2008) bei der Erstellung von Menüs kennen zu lernen, starten wir ein WinForms-Projekt und erstellen die Bedienoberfläche eines einfachen Editors im RAD-Modus (*Rapid Application Development*).

Befördern Sie im Formulardesigner aus der **Toolbox** ein **ToolStrip** - Objekt auf das Formular:

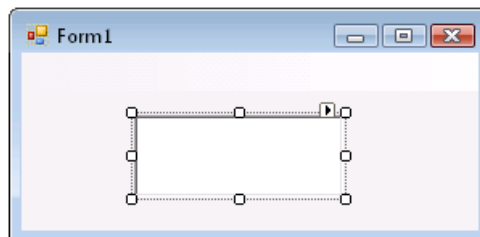
Platzierungshinweis



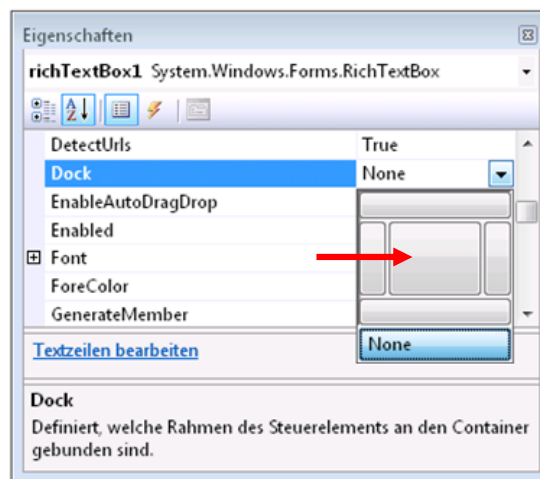
Ergebnis (Menüleiste markiert)



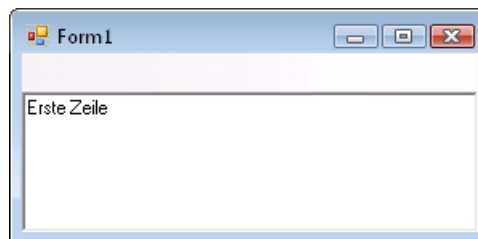
Ergänzen Sie eine **RichTextBox** - Komponente,



und wählen Sie im Eigenschaftsfenster zur **Dock**-Eigenschaft der Editorkomponente den Wert **DockStyle.Fill**:



Der entstehende Editor verhält sich wunschgemäß:



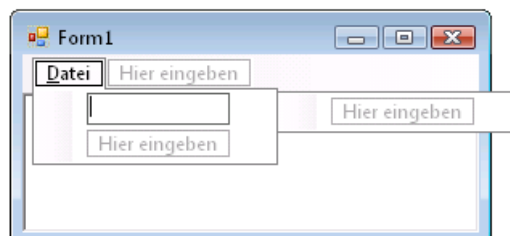
Eine Inspektion des automatisch vom Formular designer in der Datei **Form1.Designer.cs** erstellten Quellcodes zeigt, dass die **RichTextBox**- und die **ToolStrip**-Komponente in der gemäß Abschnitt 9.4.4.3 korrekten Reihenfolge als Top-Dock - Konkurrenten in die Steuerelementliste des Formulars aufgenommen worden sind:

```
this.Controls.Add(this.richTextBox1);
this.Controls.Add(this.menuStrip1);
```

Wird die notwendige **Controls**-Reihenfolge von Dock-Konkurrenten beim Einfügen per Formular designer angewendet, überlagert die Symbolleiste (**DockStyle.Top**) den Editor (**DockStyle.Fill**), und die ersten Textzeilen bleiben hinter der Symbolleiste verborgen. Sachwissen und konzentriertes Arbeiten führen leider zur falschen Reihenfolge der **Controls**-Aufrufe. Man muss also entweder naiv das oberste Steuerelement zuerst einfügen oder wissen, dass im Designer in der „falschen“ Reihenfolge vorzugehen ist.

Der Formular designer unterstützt die Menübestückung (ohne weitere Überraschungseffekte) durch ein Kombiwerkzeug:

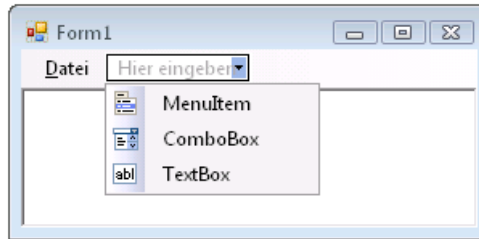
- Ein neues Menüitem entsteht, sobald man seine Beschriftung an Stelle des Platzhaltertextes **Hier eingeben** schreibt, z.B.:



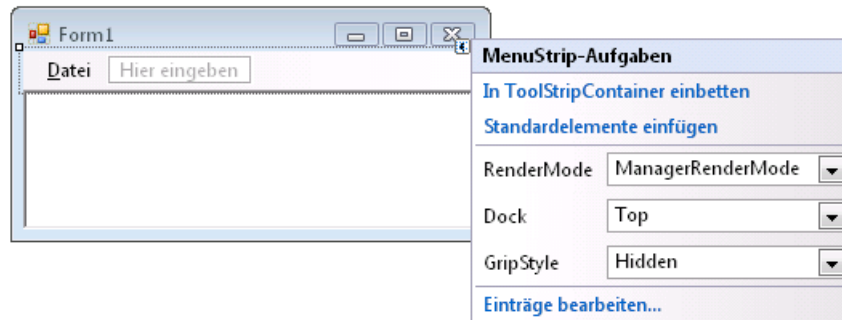
Neben einem vorhandenen Hauptmenüitem oder auch in ein Untermenü lassen sich weitere Items einfügen.

- Mit einem Bindstrich an Stelle einer Itembeschriftung erreicht man eine Horizontale Trennlinie.

- Über den Pfeil am rechten eines Platzhalters ist ein DropDown-Menü verfügbar, mit dem sich neben Menüitems auch Objekte aus den Klassen **ToolStripTextBox** und **ToolStripComboBox** einfügen lassen:



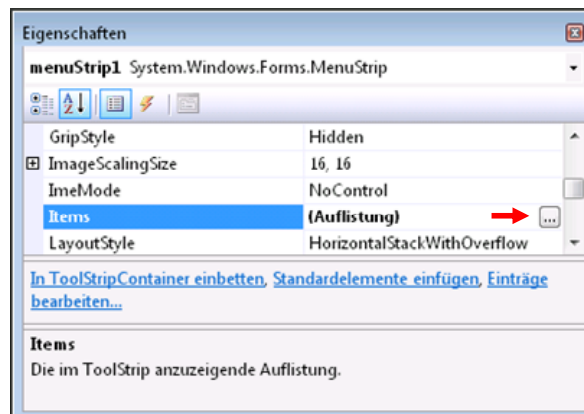
- Über den kleinen Pfeil am oberen rechten Rand der Menüleiste erhält man ein lokales Eigenschaftsfenster mit **MenuStrip-Aufgaben**:



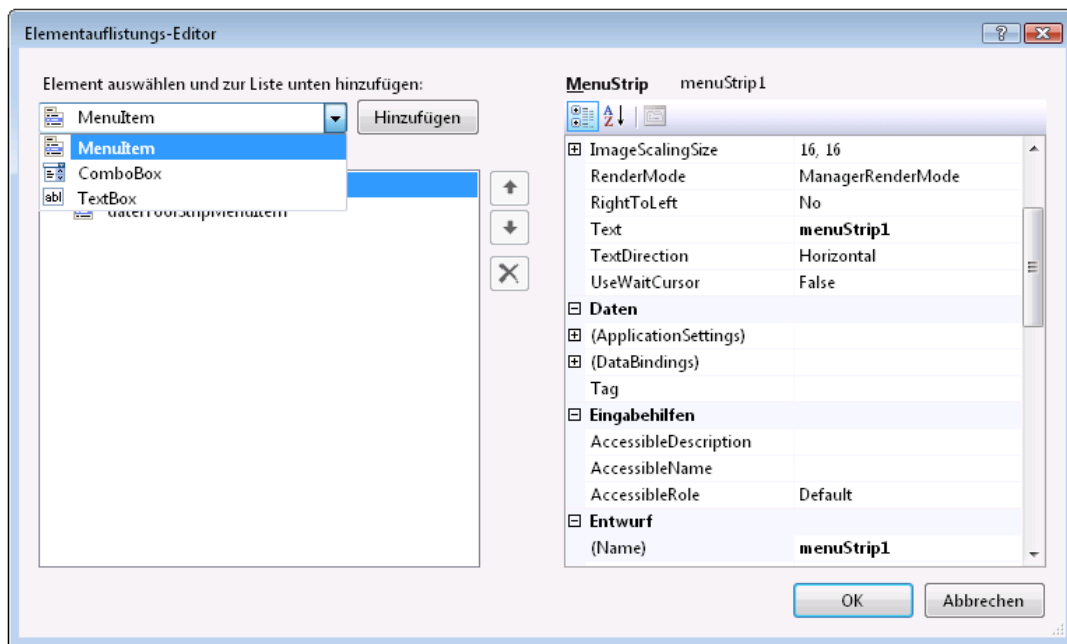
Hier lässt sich z.B. das Einbetten der Menüleiste in einen **ToolStripContainer** anfordern (siehe Abschnitt 20.1.2). Über **Elemente bearbeiten** erreicht man den **Elementauflistungs-Editor** (siehe unten). Über **Standardelemente einfügen** erhält man mit einem Mausklick eine komplette Standardmenüleiste (siehe unten).

Eine eingefügte Komponente kann man per Eigenschaftsfenster bequem konfigurieren.

Über den Erweiterungsschalter in der markierten **Items**-Zeile des Eigenschaftsfensters zur Menüleiste

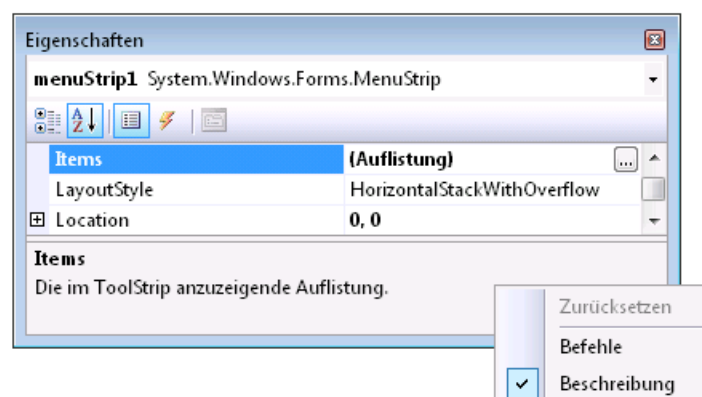


ist mit dem **Elementauflistungs-Editor** ein weiteres Hilfsmittel zur Symbolleistungsgestaltung erreichbar:

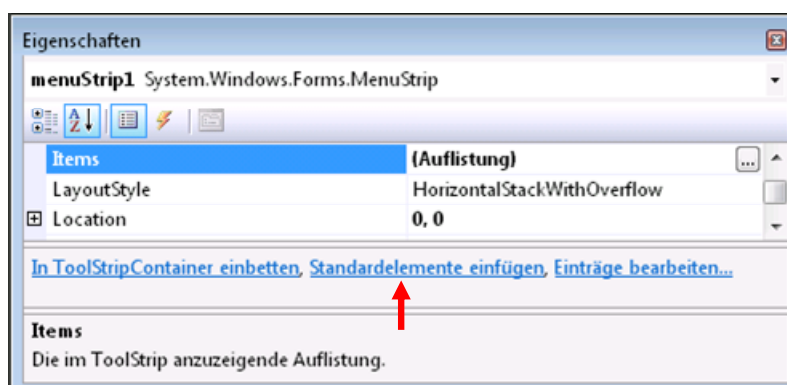


Hier lassen sich Steuerelemente aus allen zulässigen Klassen bequem einfügen, positionieren und konfigurieren.

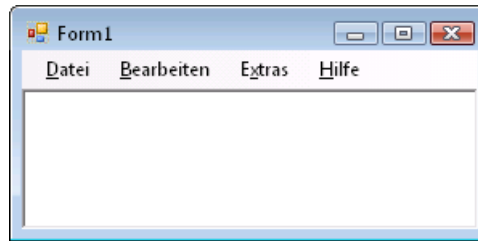
Als Clou bietet das Eigenschaftsfenster zur Menüleiste in der Befehlszone am unteren Rand, die eventuell zunächst per Kontextmenü



aktiviert werden muss, über den Link **Standardelemente einfügen** die Möglichkeit, mit einem einzigen Mausklick alle Standardkomponenten auf die Menüleiste zu setzen:



Dieselbe Funktion ist auch über das oben beschriebene lokale Eigenschaftsfenster mit **MenuStrip-Aufgaben** zu erreichen. Das Ergebnis ist beeindruckend:



können unvorsichtige Zielprogramme bei der Übernahme von Zwischenablageninhalten in Schwierigkeiten

17.7 Die Menütechnik aus .NET 1

In diesem Abschnitt wird die aus Kompatibilitätsgründen weiterhin unterstützte Menütechnik aus .NET 1 vorgestellt. Wer auf frei eine bewegliche und dynamisch auf horizontalen Platzmangel reagierende Menüleiste verzichten kann, spart im Vergleich zur neuen, ToolStrip-basierten Lösung einige Zeilen Quellcode. Ein weiterer Vorteil besteht darin, dass eine .NET 1 - Menüleiste nicht zum Klientenbereich des Fensters gehört, also nicht mit dortigen Anzeigen kollidieren kann. Dies wird vom folgenden Beispielprogramm mit Grafikausgabe

```
using System;
using System.Windows.Forms;
using System.Drawing;

class Kollision : Form {
    public Kollision() {
        Text = "Kollision";
        Size = new Size(220, 100);
        ResizeRedraw = true;

        this.Menu = new MainMenu();
        MenuItem mitFile, mitEdit, mitTools, mitHelp;
        mitFile = Menu.MenuItems.Add("&Datei");
        mitEdit = Menu.MenuItems.Add("&Bearbeiten");
        mitTools = Menu.MenuItems.Add("&Extras");
        mitHelp = Menu.MenuItems.Add("&Hilfe");
    }
    protected override void OnPaint(PaintEventArgs e) {
        Pen stift = new Pen(Color.Black, 2);
        e.Graphics.DrawLine(stift, 0, 0, ClientSize.Width, ClientSize.Height);
    }
    [STAThread]
    static void Main() {
        Application.EnableVisualStyles();
        Application.Run(new Kollision());
    }
}
```

demonstriert:

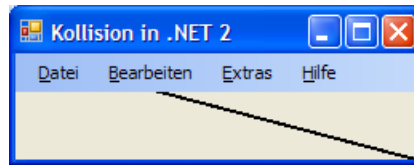


Die .NET 2 - Lösung für dasselbe Hauptmenü ist etwas aufwändiger

```
MenuStrip mainMenu = new MenuStrip();
Controls.Add(mainMenu);
ToolStripMenuItem mitFile, mitEdit, mitTools, mitHelp;
mitFile = new ToolStripMenuItem("&Datei");
mitEdit = new ToolStripMenuItem("&Bearbeiten");
mitTools = new ToolStripMenuItem("&Extras");
```

```
mitHelp = new ToolStripMenuItem("&Hilfe");
mainMenu.Items.AddRange(
    new ToolStripItem[] { mitFile, mitEdit, mitTools, mitHelp });
```

und führt zu einer Kollision im Klientenbereich:



Natürlich ist dieses Problem leicht zu beheben, z.B. durch die Grafikausgabe auf der Oberfläche eines Steuerelements aus der Klasse **Panel** statt auf der Oberfläche des Formulars.

17.7.1 Wichtige Klassen und Begriffe

Bei der .NET 1 - Lösung wird das Hauptmenü (die Menüleiste) eines Formulars durch ein Objekt der Klasse **MainMenu** realisiert, das der **Form**-Eigenschaft **Menu** zugewiesen wird:

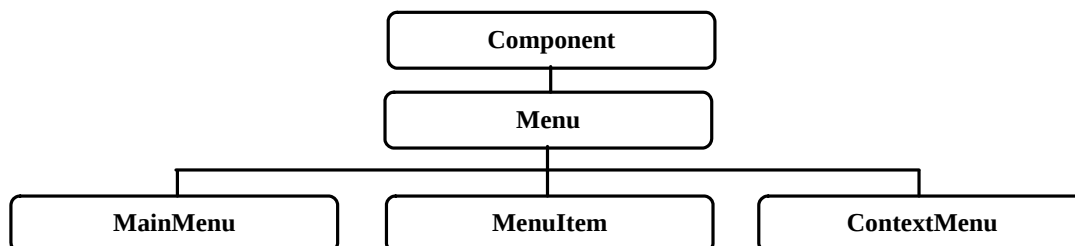
```
this.Menu = new MainMenu();
```

Mit den folgenden Anweisungen werden die **MenuItem**-Komponenten **mitDatei**, **mitExtras** und **mitHilfe** erzeugt und in das Hauptmenü des Formulars eingefügt:

```
MenuItem mitFile, mitEdit, mitTools, mitHelp;

mitFile = Menu.MenuItems.Add("&Datei");
mitEdit = Menu.MenuItems.Add("&Bearbeiten");
mitTools = Menu.MenuItems.Add("&Extras");
mitHelp = Menu.MenuItems.Add("&Hilfe");
```

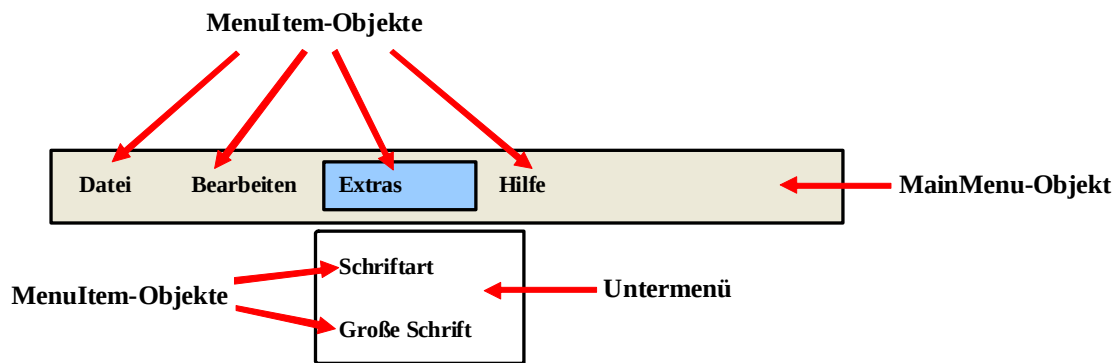
Vor der weiteren Beschäftigung mit dem Quellcode des Beispiels werden Erbschaftsangelegenheiten erläutert und wichtige Begriffe erklärt. Die Klassen **MainMenu**, **ContextMenu** (siehe Abschnitt 17.7.3) und **MenuItem** stammen von der gemeinsamen (abstrakten) Basisklasse **Menu** ab:



Menu stammt *nicht* von **Control** ab, so dass die Eigenschaften dieser Klasse wie **BackColor**, **ForeColor**, **Font** nicht zur Verfügung stehen. Um die Farbe oder Schriftart eines Menüs per Programm zu ändern, muss einiger Aufwand betrieben und die *Owner Draw* - Technik verwendet werden (siehe z.B. Petzold 2002, S.587ff). Benutzer können hingegen diese Merkmale relativ leicht per Windows-Systemsteuerung über die Eigenschaften der Anzeige modifizieren.

Von ihrer gemeinsamen Basisklasse erben **MainMenu** und **MenuItem** u.a. die Eigenschaft **MenuItems**, die auf eine Liste mit **MenuItem**-Objekten zeigt. Diese Liste (ein Objekt aus der **Menu**-internen Klasse **MenuItemCollection**) verfügt u.a. über die mehrfach überladene und komfortable **Add()**-Methode zum Erzeugen von Menüitems. Offenbar ist die **MenuItems**-Eigenschaft der Klasse **Menu** ähnlich einzusetzen wie die **Controls**-Eigenschaft der Klasse **Control**.

Wird ein **MenuItem**-Objekt mit vorhandener Kollektion von untergeordneten **MenuItem**-Objekten vom Benutzer ausgewählt (z.B. per Maus), dann erscheint ein rechteckiges Fenster mit einem Untermenü:



Es existiert keine spezielle Klasse für Untermenüs, weil *jedes* **Menu**-Objekt eine Kollektion untergeordneter **MenuItem**-Objekte (und damit ein Untermenü) aufnehmen kann.

17.7.2 Menüitems erzeugen

Statt wie in obigen Beispielen **MenuItem**-Objekte implizit durch einen Aufruf der **Menu**-Methode **Add()** zu erzeugen, kann man sie mit erweiterter Gestaltungsmöglichkeit auch über einen expliziten Konstruktor-Aufruf erstellen, z.B.:

```
MenuItem mitInfo;
...
mitInfo = new MenuItem("&Info",
    new EventHandler(MitInfoOnClick),
    Shortcut.F1);
mitHelp.MenuItems.Add(mitInfo);
```

Hier wird neben der **Text**-Eigenschaft des Menüitems (mit &-Kennzeichnung der Zugriffstaste, vgl. Abschnitt 17.2) auch ein **EventHandler** für sein **Click**-Ereignis und eine auslösende Taste(nkombination) zum **Click**-Ereignis vereinbart, wobei die Enumeration **Shortcut** eine reichhaltige Auswahl an Bezeichnungen für Tasten(kombinationen) bietet.

Weil ein Menüitem wiederum Menüitems aufnehmen kann, erstellt man leicht ein mehrstufiges Menü. Im folgenden Segment des Fensterkonstruktors zum Beispielprogramm wird zunächst per **Add()**-Aufruf ein Menüitem namens **mitFont** erzeugt und dem Hauptmenüitem **Extras** untergeordnet:

```
MenuItem mitFont, mitCourierNew, mitTimesNewRoman, mitArial, mitCurrentFont;
...
mitFont = mitTools.MenuItems.Add("&Schriftart");
```

Dann entstehen die Menüitems **mitCourierNew**, **mitTimesNewRoman** und **mitArial**, die in der Item-Kollektion von **mitFont** landen:

```
EventHandler fontHandler = new EventHandler(MitFontOnClick);
mitCourierNew = new MenuItem("&Courier New", fontHandler);
mitCourierNew.RadioCheck = true;
mitFont.MenuItems.Add(mitCourierNew);
mitTimesNewRoman = new MenuItem("&Times New Roman", fontHandler);
mitTimesNewRoman.Checked = true;
mitTimesNewRoman.RadioCheck = true;
mitCurrentFont = mitTimesNewRoman;
mitFont.MenuItems.Add(mitTimesNewRoman);
mitArial = new MenuItem("&Arial", fontHandler);
mitArial.RadioCheck = true;
mitFont.MenuItems.Add(mitArial);
```

Die im Quellcode auftauchenden **Menu**-Eigenschaften **Checked** und **RadioCheck** werden gleich behandelt. Mit der Referenzvariablen **mitCurrentFont** wird der Quellcode zum Umschalten zwischen den Schriftarten vereinfacht (siehe unten).

Das Untermenü zum Hauptmenüitem **Bearbeiten**:

```
mitUndo = mitEdit.MenuItems.Add("&Rückgängig", new
EventHandler(MitUnRedoOnClick));
mitUndo.Shortcut = Shortcut.CtrlZ;
mitRedo = mitEdit.MenuItems.Add("&Wiederholen", new
EventHandler(MitUnRedoOnClick));
mitRedo.Shortcut = Shortcut.CtrlY;
mitEdit.MenuItems.Add("-");
mitCut = mitEdit.MenuItems.Add("&Ausschneiden", new
EventHandler(MitCutOnClick));
mitCut.Shortcut = Shortcut.CtrlX;
mitCopy = mitEdit.MenuItems.Add("&Kopieren", new
EventHandler(MitCopyOnClick));
mitCopy.Shortcut = Shortcut.CtrlC;
mitPaste = mitEdit.MenuItems.Add("&Einfügen", new
EventHandler(MitPasteOnClick));
mitPaste.Shortcut = Shortcut.CtrlV;
mitDel = mitEdit.MenuItems.Add("&Löschen", new
EventHandler(MitDeleteOnClick));
mitDel.Shortcut = Shortcut.Del;
mitEdit.MenuItems.Add("-");
mitSelectAll = mitEdit.MenuItems.Add("Alles &markieren", new
EventHandler(MitSelectAllOnClick));
mitSelectAll.Shortcut = Shortcut.CtrlA;
```

Weil keine **Add()**-Überladung mit **Shortcut**-Parameter existiert, wird die **MenuItem**-Eigenschaft **Shortcut** in einer separaten Anweisung versorgt. In den **Click**-Ereignisbehandlungsmethoden zu den **Bearbeiten**-Menüitems kann man sich meist auf den Aufruf von **RichTextBox**-Methoden beschränken, z.B.:

```
protected void MitCutOnClick(object sender, EventArgs e) {
    editor.Cut();
}
```

Durch ein Menüitem mit Bindestrich als Wert der **Text**-Eigenschaft fügt man eine horizontale Trennlinie in ein Untermenü ein, z.B.:

```
mitEdit.MenuItems.Add("-");
```

Die drei Items im Menü **Extras > Schriftart** stehen für eine Gruppe sich gegenseitig ausschließender Optionen und verwenden denselben **Click**-Handler:

```
protected void MitFontOnClick(object sender, EventArgs e) {
    mitCurrentFont.Checked = false;
    mitCurrentFont = (MenuItem)sender;
    mitCurrentFont.Checked = true;

    if (sender == mitCourierNew) {
        editor.Font = (mitSize.Checked ? fontCN16 : fontCN12);
    } else
    if (sender == mitTimesNewRoman) {
        editor.Font = (mitSize.Checked ? fontTNR16 : fontTNR12);
    } else {
        editor.Font = (mitSize.Checked ? fontA16 : fontA12);
    }
}
```

Das **RichTextBox**-Steuerelement erhält die gewünschte Schriftart, wobei ein eventueller Benutzerwunsch nach großen Schriften zu berücksichtigen ist, was unter Verwendung des Konditionaloperators geschieht.

Mit den ersten Anweisungen der Methode wird für eine korrekte Markierung des Menüitems zur aktuell gewählten Schrift gesorgt. Über seine **Checked**-Eigenschaft lässt sich ein Menüitem als

gewählt markieren. Ein benutzerfreundliches Programm wählt dabei einen adäquaten Markierungsstil über die Eigenschaft **RadioCheck**:

- **Menüitems zum Ein- bzw. Ausschalten eines Merkmals**
Bei einem Menüitem zum Ein- bzw. Ausschalten eines Merkmals eignet sich der voreingestellte **RadioCheck**-Wert **false**, wobei der eingeschaltete Zustand durch ein Häkchen markiert wird. Im Beispielprogramm zeigt die Menüoption **Extras > Große Schrift** dieses Verhalten.
- **Menüitems zur Wahl aus einer Menge sich gegenseitig ausschließender Alternativen**
Dient eine Gruppe von Menüitems zur Wahl aus einer Menge sich gegenseitig ausschließender Alternativen, sollten alle **RadioCheck**-Eigenschaften auf **true** gesetzt werden. Dann führt der **Checked**-Wert **true** zu einem Markierungsstil wie bei Optionsfeldern. Im Beispielprogramm zeigen die Items im Untermenü zu **Extras > Schriftart** dieses Verhalten.

Leider existiert für Items mit Optionsfeld-Logik keine Gruppenbildung mit dem Effekt, dass genau *ein* Item markiert ist. Im Beispielprogramm sorgt folgende Technik (übernommen von Petzold 2002, S. 572ff) für stets korrekte Markierungen:

- Die **MenuItem**-Referenz `mitCurrentFont` zeigt stets auf das aktuell markierte Schriftartenitem.
- In `MitFontOnClick()` wird zunächst die aktuelle (alte) Markierung aufgehoben. Dann erhält `mitCurrentFont` die Adresse des vom Benutzer gewählten Items, und schließlich wird dieses Item markiert.

Ist ein Menüitem momentan nicht nutzbar (z.B. **Bearbeiten > Kopieren** ohne markierten Text), dann sollte seine **Enabled**-Eigenschaft auf **false** gesetzt werden. Es wird dann abgeblendet dargestellt, und sein **Click**-Ereignis ist nicht auslösbar. Über das **MenuItem**-Ereignis **Popup**, das vor dem Öffnen eines Untermenüs eintritt, kann man für eine rechtzeitige **Enabled**-Aktualisierung sorgen, z.B.:

```
mitEdit.Popup += new EventHandler(MitEditOnPopup);
...
protected void MitEditOnPopup(object sender, EventArgs e) {
    if (editor.CanUndo) {
        mitUndo.Enabled = true;
    } else {
        mitUndo.Enabled = false;
    }
    if (editor.CanRedo) {
        mitRedo.Enabled = true;
    } else {
        mitRedo.Enabled = false;
    }
    if (editor.SelectionLength > 0) {
        mitCut.Enabled = true;
        mitCopy.Enabled = true;
        mitDel.Enabled = true;
    } else {
        mitCut.Enabled = false;
        mitCopy.Enabled = false;
        mitDel.Enabled = false;
    }
    if (Clipboard.ContainsText()) {
        mitPaste.Enabled = true;
    } else {
        mitPaste.Enabled = false;
    }
    if (editor.TextLength > 0) {
        mitSelectAll.Enabled = true;
    } else {
        mitSelectAll.Enabled = false;
    }
}
```

Über die Methode **ContainsText()** der Klasse **Clipboard** sorgen wir dafür, dass unser Editor ausschließlich Text (also z.B. keine Bilder) aus der Windows-Zwischenablage übernimmt.

Mit Hilfe der **MenuItemCollection**-Methoden zum Einfügen oder Entfernen von Menüitems (**Add()**, **Clear()**, **Remove()** etc.) gelingt es, per **Popup**-Handler ein Untermenü komplett neu aufzubauen, um eine Liste von Menüitems stets aktuell zu halten

Über das **Select**-Ereignis kann man z.B. dafür sorgen, dass zum gerade markierten Menüitem eine Erläuterung in der Statuszeile erscheint (siehe unten).

Um eine Taste(nkombination) „ohne“ zugehöriges Menüitem mit einer Behandlungsmethode zu verbinden, setzt man die **Visible**-Eigenschaft des Menüitems auf den Wert **false**.

17.7.3 Kontextmenü

Das Kontextmenü zu einem Steuerelement wird in .NET 1 - Technik über ein Objekt der Klasse **ContextMenu** realisiert (siehe Abstammungsdiagramm in Abschnitt 17.7.1). Dem Konstruktor übergibt man ein Array mit **MenuItem**-Objekten. So lässt sich im Editorprojekt zur **RichTextBox**-Komponente ein Kontextmenü



mit den Bedienoptionen des **Bearbeiten**-Menüs realisieren:

```
MenuItem mitUndo, mitRedo, mitCut, mitCopy, mitPaste, mitDel, mitSelectAll,
    cmitUndo, cmitRedo, cmitCut, cmitCopy, cmitPaste, cmitDel, cmitSelectAll;

cmitUndo = new MenuItem("&Rückgängig", new EventHandler(MitUnRedoOnClick));
cmitRedo = new MenuItem("&Wiederholen", new EventHandler(MitUnRedoOnClick));
cmitCut = new MenuItem("&Ausschneiden", new EventHandler(MitCutOnClick));
cmitCopy = new MenuItem("&Kopieren", new EventHandler(MitCopyOnClick));
cmitPaste = new MenuItem("&Einfügen", new EventHandler(MitPasteOnClick));
cmitDel = new MenuItem("&Löschen", new EventHandler(MitDeleteOnClick));
cmitSelectAll = new MenuItem("Alles &markieren",
    new EventHandler(MitSelectAllOnClick));
MenuItem[] cmi = {cmitUndo, cmitRedo, new MenuItem("-"),
    cmitCut, cmitCopy, cmitPaste, new MenuItem("-"), cmitSelectAll};
```

Weil ein **MenuItem** nicht zu zwei Containern gehören kann, benötigen wir im Beispielpogramm zu jeder **Bearbeiten**-Funktion zwei **MenuItem**-Objekte, eines für das Hauptmenü und eines für das Kontextmenü.

Das **ContextMenu**-Objekt wird der **Control**-Eigenschaft des **RichTextBox**-Steuerelements zugewiesen:

```
editor.ContextMenu = new ContextMenu(cmi);
```

Über das **ContextMenu**-Ereignis **Popup**, das vor dem Öffnen des Kontextmenüs eintritt, kann man die **Enabled**-Eigenschaften der enthaltenen Menüitems rechtzeitig aktualisieren oder das Kontextmenü passend zusammenstellen, z.B.:

```
editor.ContextMenu.Popup += MitEditOnPopup;
```

Den vollständigen Quellcode mit der .NET 1 - Menülösung zum Editorprojekt finden Sie im Ordner
...\BspUeb\WinForms\Menüs\DotNet 1

17.8 Übungsaufgaben zu Kapitel 17

1) Nehmen Sie an unserem Editorprojekt ausgehend vom Entwicklungsstand in

...\BspUeb\WinForms\Menüs\DotNet 2\Kontextmenü

folgende Änderungen vor:

- Erweitern Sie das **Bearbeiten**-Untermenü:



Die zusätzlich benötigten **Click**-Ereignishandler sind mit Hilfe der **RichTextBox**-Methoden **Undo()**, **Redo()** und **SelectAll()** leicht zu realisieren. Setzen Sie die **Enabled**-Eigenschaft der Menüitems je nach Verfügbarkeit auf **true** oder **false**. Definieren Sie Tastenkombinationen zum Starten der **Click**-Ereignismethoden. Verwenden Sie bei Bedarf bitweise Oder-Verknüpfungen von zwei **Keys**-Werten, z.B.:

```
mitUndo.ShortcutKeys = Keys.Control | Keys.Z;
```

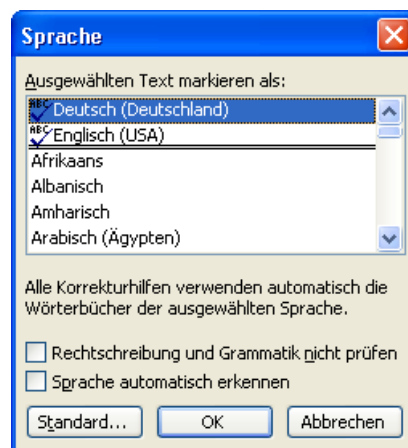
- Verbessern Sie analog auch das Kontextmenü (allerdings ohne **ShortcutKeys**):



18 Dialogfenster

Nur die wenigsten Windows-Anwendungen beschränken sich auf ein einziges Formular. Auch in unseren (möglichst einfach gestrickten) Beispielprogrammen kamen gelegentlich zusätzliche Formulare zum Einsatz, wobei wir uns aber auf **MessageBox.Show()**-Aufrufe und Standarddialoge beschränkt haben. Wir beschäftigen uns nun mit der Verwendung von selbst entworfenen Dialogfenstern und erweitern anschließend unser Wissen über die Standarddialoge.

Bei WinForms-Programmen werden auch Dialogfenster durch Objekte einer aus **System.Windows.Forms.Form** abgeleiteten Klasse realisiert; es gibt also keine spezielle Basisklasse für Dialogfenster. Der Begriff *Dialogfenster* steht für die Verwendung von (zusätzlichen) **Form**-Objekten zur Kommunikation mit dem Benutzer (zur Präsentation oder Erfassung von Informationen). In den *Official Guidelines for User Interface Developers and Designers* ordnet Microsoft die Dialogfenster (zusammen mit den Nachrichtenfestern und Werkzeugpaletten) den *sekundären Fenstern* zu (Microsoft 2007b, S. 217). Typische Kennzeichen von Dialogfenstern sind die unveränderliche Formlargoße und die sparsame Ausstattung der Titelseite mit Symbolschaltern, z.B. beim folgenden Dialog aus MS-Word 2003:



Die Initiative für den Auftritt eines Dialogfensters für eine spezielle Kommunikationsaufgabe kann vom Benutzer ausgehen (z.B. durch Wahl eines Menüitems) oder vom Programm (z.B. Nachfrage vor dem Löschen einer Datei).

18.1 Modale Dialogfenster

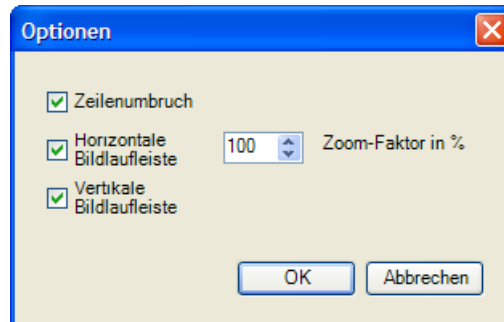
Die meisten Dialogfenster sind vom *modalen* Typ, so dass die restlichen Fenster der Anwendung bis zum Schließen des Dialogs keine Benutzereingaben entgegen nehmen. Einem modalen Dialog ist also die ungeteilte Aufmerksamkeit der Benutzer gewiss. Dies war z.B. bei den Standarddialogen zur Schriftart- oder Farbauswahl zu beobachten. In der Regel kann der Benutzer zwar zu anderen Programmen wechseln, doch die Anwendung „hinter“ dem modalen Dialog ist von allen Maus- oder Tastaturereignissen abgeschottet. Einige Ereignisse werden aber auch bei abgeschotteten Fenstern bzw. Steuerelementen ausgelöst:

- **Paint**-Ereignisse, die zum Neuaufbau des Klientenbereichs auffordern
- **Tick**-Ereignisse von **Timer**-Objekten

Wie gleich zu sehen sein wird, resultiert das modale Verhalten eines Dialogs *nicht* aus irgendwelchen **Form**-Eigenschaftsausprägungen, sondern aus dem Auftritt über einen **ShowDialog()**-Aufruf (im Unterschied zum **Show()**-Aufruf bei nicht-modalen Dialogen). Weil sich die geplante modale Verwendung eines Dialogs in der Regel auf seine Ausstattung mit Bedienelementen auswirkt (z.B. **OK**-Schalter), ist die Rede von *modalen Dialogen* aber doch gerechtfertigt.

18.1.1 Konfiguration

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 unseren seit Abschnitt 16.5 sukzessive aufgebauten Editor realisiert:



Folgende Einstellungen des Editors können im Optionen-Dialog modifiziert werden:

- **Zeilenumbruch**
Mit diesem Kontrollkästchen wird die **RichTextBox**-Eigenschaft **WordWrap** gesetzt. Beim Wert **true** wird der Text im Steuerelement automatisch umgebrochen.
- **Bildlaufleisten**
Mit den beiden Kontrollkästchen legt der Benutzer den Wert der **RichTextBox**-Eigenschaft **ScrollBars** (vom Enumerationstyp **RichTextBoxScrollBars**) fest:

		Vertikale Bildlaufleiste	
		an	aus
Horizontale Bildlaufleiste	an	Both	Horizontal
	aus	Vertical	None

Die Bildlaufleisten erscheinen nur bei Bedarf, so dass die horizontale Leiste bei eingeschaltetem Zeilenumbruch *nie* erscheint.

- **Zoom-Faktor**
Dieser **UpDown**-Regler wirkt auf die **RichTextBox**-Eigenschaft **ZoomFactor**, die (bei einem erlaubten Wert zwischen 1/64 und 64) die angezeigte Größe der Schrift verändert. Wir arbeiten im Optionen-Dialog mit Prozentangaben, so dass sich erlaubte Werte von 64 bis 640 ergeben,

Im Konstruktor eines modalen Dialogs finden sich in der Regel einige Besonderheiten im Vergleich zu den bisher gewohnten Fensterkonstruktoren:

- Über die booleschen Eigenschaften **HelpBox** (Voreinstellung **false**), **MaximizeBox** und **MinimizeBox** (Voreinstellung jeweils **true**) sorgt man für eine bescheidene Ausstattung der Titelzeile mit Standardschaltflächen, z.B.

```
MaximizeBox = false;
MinimizeBox = false;
```
- Über folgenden Wert für die Eigenschaft **FormBorderStyle** erreicht man einen fixierten Rahmen ohne Systemmenü-Symbolschalter (am linken Rand der Titelzeile):

```
FormBorderStyle = FormBorderStyle.FixedDialog;
```
- Meist sind Schaltflächen mit der Beschriftung (und Bedeutung) **OK** bzw. **Abbrechen** vorhanden, und man sorgt über die **Form**-Eigenschaften **AcceptButton** bzw. **CancelButton** dafür, dass die **Click**-Ereignisse dieser Schalter von der **Eingabe**- bzw. **Esc**-Taste ausgelöst werden (vgl. Abschnitt 9.4.2.2):

```
AcceptButton = cmdOK;
CancelButton = cmdCancel;
```

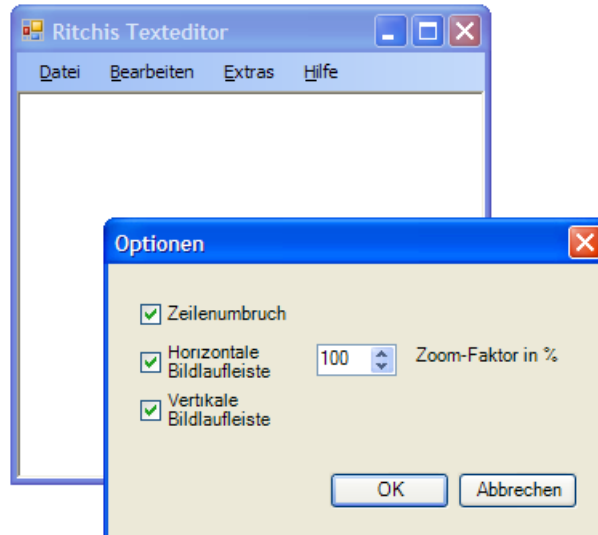
In der Regel wird ein Dialogfenster auf gleich näher zu beschreibende Weise über diese Schalter beendet, wobei die aufrufende Methode die **DialogResult**-Eigenschaftsausprägung des auslösenden Schalters als Rückgabewert erhält:

```
cmdOK.DialogResult = DialogResult.OK;
cmdCancel.DialogResult = DialogResult.Cancel;
```

Man setzt die Standardschaltflächen oft in die untere rechte Ecke eines Dialogfensters.

Der Optionsdialog unseres Editors soll ...

- beim ersten Auftritt eine Position in Relation zum Hauptfenster wählen



- und bei späteren Auftritten an der zuletzt vom Benutzer gewählten Position erscheinen.

Dieses Verhalten wird folgendermaßen realisiert:

- Während einer Editor-Sitzung kommt stets dasselbe **OptionsDialog**-Objekt zum Einsatz. Zwischen zwei Einsätzen des Fensters bleibt seine Desktop-Position in der **Form**-Eigenschaft **Location** erhalten.
- Im **OptionsDialog**-Konstruktor wird eine Initialposition in Relation zum Anwendungsfenster gewählt:

```
Location = Owner.DesktopLocation +
    new Size(Owner.ClientSize.Width / 5, Owner.ClientSize.Height / 2);
```

Soll ein Formular eine individuelle Startposition an Stelle der Windows-Voreinstellung erhalten, dann muss seine Eigenschaft **StartPosition** auf den Wert

FormStartPosition.Manual gesetzt werden. Im **OptionsDialog**-Konstruktor findet sich daher die Zeile:

```
StartPosition = FormStartPosition.Manual;
```

Wer damit einverstanden ist, dass der Dialog stets an derselben Position in sinnvoller Orientierung zum elterlichen Fenster erscheint, kommt mit folgender Ortsangabe aus:

```
StartPosition = FormStartPosition.CenterParent;
```

Ansonsten bietet der **OptionsDialog**-Konstruktor die mittlerweile vertrauten Steuerelement-Gestaltungsarbeiten:

```
public OptionsDialog(Form owner) {
    Text = "Optionen";
    MaximizeBox = false;
    MinimizeBox = false;
    FormBorderStyle = FormBorderStyle.FixedDialog;
    ClientSize = new Size(310, 170);
    StartPosition = FormStartPosition.Manual;
    Owner = owner;
}
```

```

Location = Owner.DesktopLocation +
    new Size(Owner.ClientSize.Width/5, Owner.ClientSize.Height/2);

Button cmdOK = new Button();
cmdOK.Text = "OK";
cmdOK.Parent = this;
cmdOK.Location = new Point(138, 130);
AcceptButton = cmdOK;
cmdOK.DialogResult = DialogResult.OK;

Button cmdCancel = new Button();
cmdCancel.Text = "Abbrechen";
cmdCancel.Parent = this;
cmdCancel.Location = new Point(218, 130);
CancelButton = cmdCancel;
cmdCancel.DialogResult = DialogResult.Cancel;

chkWordWrap = new CheckBox();
chkWordWrap.Text = "Zeilenumbruch";
chkWordWrap.Location = new Point(20, 20);
chkWordWrap.Parent = this;

chkHorBar = new CheckBox();
chkHorBar.Text = "Horizontale Bildlaufleiste";
chkHorBar.Location = new Point(20, 50);
chkHorBar.Width = 100;
chkHorBar.Parent = this;

chkVertBar = new CheckBox();
chkVertBar.Text = "Vertikale Bildlaufleiste";
chkVertBar.Location = new Point(20, 80);
chkVertBar.Parent = this;

updwnZoomFactor = new NumericUpDown();
updwnZoomFactor.Location = new Point(130, 50);
updwnZoomFactor.Size = new Size(50, 50);
updwnZoomFactor.Minimum = 64;
updwnZoomFactor.Increment = 5;
updwnZoomFactor.Maximum = 640;
updwnZoomFactor.Value = 100;
updwnZoomFactor.Parent = this;

Label lbZoomFactor = new Label();
lbZoomFactor.Location = new Point(190, 50);
lbZoomFactor.Text = "Zoom-Faktor in %";
lbZoomFactor.Parent = this;
}

```

18.1.2 Datenaustausch mit dem aufrufenden Formular

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 chkWordWrap.Checked;}
    set {chkWordWrap.Checked = 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.

Durch den Einsatz von öffentlichen Eigenschaften ist unser Dialogfenster analog zu den Standarddialogen (siehe Abschnitt 18.3) zu verwenden. Eine gegen das Datenkapselungsprinzip der objekt-

orientierten Programmierung (vgl. Abschnitt 4.1.1.1) verstoßende Alternative zur Definition öffentlicher Eigenschaften bestünde darin, für die informationshaltigen Steuerelemente des Dialogfensters öffentliche Instanzvariablen zu verwenden (siehe z.B. Luis & Strasser 2008, S. 629).

Um den Optionsdialog im Hauptfenstermenü anzubieten, wird das **Extras**-Untermenü um ein Item erweitert (vgl. Abschnitt 17):

```
mitOptions = new ToolStripMenuItem("&Optionen...", null, MitOptionenOnClick);
mitTools.DropDownItems.Add(mitOptions);
```

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.

18.1.3 Auftritt und Abgang

Ein modaler Dialog wird über die **Form**-Methode **ShowDialog()** angezeigt, z.B. in der **Click**-Behandlungsmethode des zuständigen Menüitems in unserem Editor:

```
protected void MitOptionenOnClick(object sender, EventArgs e) {
    if (optionsDialog == null) {
        optionsDialog = new OptionsDialog();
    }
    optionsDialog.ZoomFactor = editor.ZoomFactor;
    optionsDialog.ScrollBars = editor.ScrollBars;
    optionsDialog.WordWrap = editor.WordWrap;
    if (optionsDialog.ShowDialog() == DialogResult.OK) {
        editor.WordWrap = optionsDialog.WordWrap;
        editor.ScrollBars = optionsDialog.ScrollBars;
        editor.ZoomFactor = optionsDialog.ZoomFactor;
    }
}
```

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 Enumerationstyp **DialogResult**, welches Verhalten des Benutzers zum Schließen des Fensters geführt hat (siehe unten).

In der Methode **MitOptionenOnClick()** des Beispielprogramms werden nur nach Beendigung des Optionen-Dialogfensters mit dem Rückgabewert **DialogResult.OK** die dortigen Eigenschaften ins Anwendungsfenster übernommen.

Über den **ShowDialog()**-Rückgabewert erfahren wir den Wert der **DialogResult**-Eigenschaft des Dialogfenster-Objekts mit der gleichnamigen Enumeration als Datentyp. Wird dieser **Form**-Eigenschaft ein von **DialogResult.None** verschiedener Wert zugewiesen, resultieren bei einem modal ausgeführten Dialogfenster folgende Konsequenzen:

- Das Dialogfenster verschwindet von der Bildfläche, wobei das zugehörige Objekt samt Fenster-Ressourcen aber weiterhin existiert.
- Die Methode **ShowDialog()** kehrt zurück.

Dank diverser Automatisierungen ist es selten nötig, die terminierende Wertzuweisung explizit vorzunehmen:

- Klickt der Benutzer auf die Schließen-Schaltfläche der Dialogfenster-Titelzeile, erhält **DialogResult** den Wert **Cancel**.
- Mit der **Button**-Eigenschaft **DialogResult** legt man einen Wert fest, den die **DialogResult**-Eigenschaft des elterlichen Formulars bei einem **Click** auf einen Schalter erhalten soll. Somit kann man sich den **Click**-Handler für eine Schaltfläche sparen, wenn lediglich ein Be-

enden des Dialogs mit bestimmtem **DialogResult**-Wert bezweckt ist. Folgende Werte der Enumeration **DialogResult** bewirken eine Rückkehr des **ShowDialog()**-Aufrufs:

DialogResult-Rückgabewert	Übliche Beschriftung der auslösenden Schaltfläche
OK	OK
Cancel	Abbrechen
Abort	Abbrechen
Retry	Wiederholen
Ignore	Ignorieren
Yes	Ja
No	Nein

Im Beispiel setzt der Schalter `cmdOK` den Wert **DialogResult.OK** und der Schalter `cmdCancel` den Wert **DialogResult.Cancel**.

Unser Editor verwendet nur *ein* **OptionsDialog**-Objekt und erstellt es nur bei Bedarf in der **Click**-Behandlungsmethode zum Menüitem **Extras > Optionen**. Im Vergleich zur Objektkreation bei jedem Aufruf des Menübefehls sparen wir Speicher und Rechenzeit. Außerdem merkt sich das Dialogfeld ohne unser Zutun seine letzte Position.

Beim beschriebenen Abgang eines Dialogfelds wird vom .NET - Framework die **Control**-Methode **Hide()** aufgerufen. Sollte das Dialogfeld irgendwann obsolet geworden sein, kann über einen expliziten **Close()**-Aufruf für die frühzeitige Freigabe seiner Windows-Ressourcen gesorgt werden.

Den vollständigen Quellcode zum aktuellen Entwicklungsstand des Editors finden Sie im Ordner **...\BspUeb\WinForms\Dialogfenster\Modal\Optionendialog**

18.1.4 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 btnApply = new Button();
btnApply.Text = "Übernehmen";
btnApply.Width = 90;
btnApply.Parent = this;
btnApply.Location = new Point(200, 130);
btnApply.Click += new EventHandler(ButtonApplyOnClick);

```

- 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.:

```
protected void ButtonApplyOnClick(object sender, EventArgs ea) {
    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.:

```

protected void OptionsDialogOnApply(object sender, EventArgs e) {
    editor.WordWrap = optionsDialog.WordWrap;
    editor.ScrollBars = optionsDialog.ScrollBars;
    editor.ZoomFactor = optionsDialog.ZoomFactor;
}

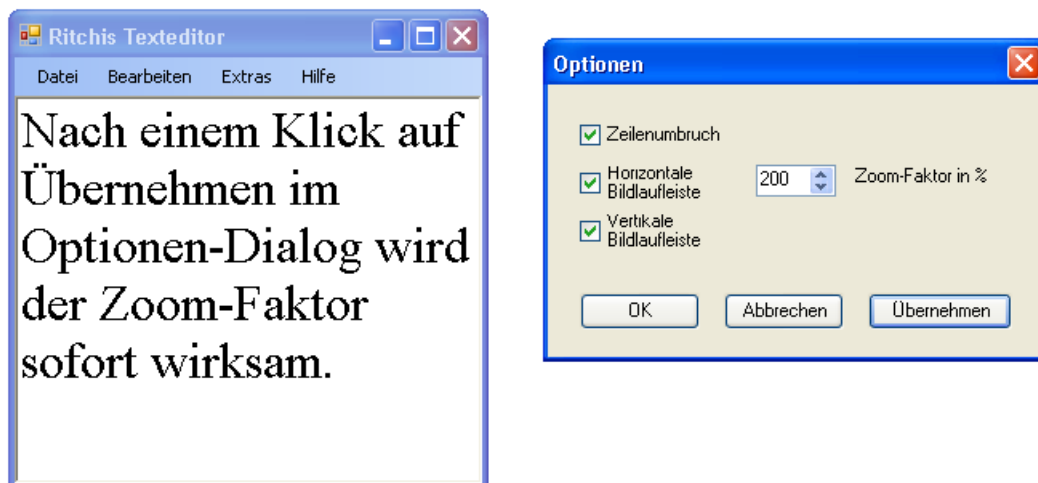
```

Hier übernimmt man aus den öffentlichen Eigenschaften des Dialogfelds alle Einstellungen, die sich auf die Anzeige des Hauptfensters auswirken.

- Diese Methode wird beim **Apply**-Ereignis des Dialogfenster-Objekts registriert, z.B.:

```
optionsDialog.Apply += new EventHandler(OptionsDialogOnApply);
```
- Sind die vom Dialog übernommenen Informationen relevant für Grafikausgaben des Hauptfensters, dann sollte das Hauptfenster am Ende der Methode **OptionsDialogOnApply** per **Invalidate()**-Aufruf für ungültig erklärt werden, damit der **Paint**-Handler aktiv wird (vgl. Abschnitt 15.3.3). Im Beispiel ist dies nicht erforderlich.

Unser Editor erlaubt nun z.B. eine bequeme Auswahl des Zoom-Faktors:



Beim Verlassen des Dialogs mit **Abbrechen** sollen nach den Design-Empfehlungen der Firma Microsoft (2007b, S. 213) alle *noch nicht übernommenen* Einstellungsänderungen verworfen werden. Die bereits per **Übernehmen** in Kraft gesetzten Einstellungsänderungen sollen aber erhalten bleiben.

Den vollständigen Quellcode zum aktuellen Entwicklungsstand des Editors finden Sie im Ordner

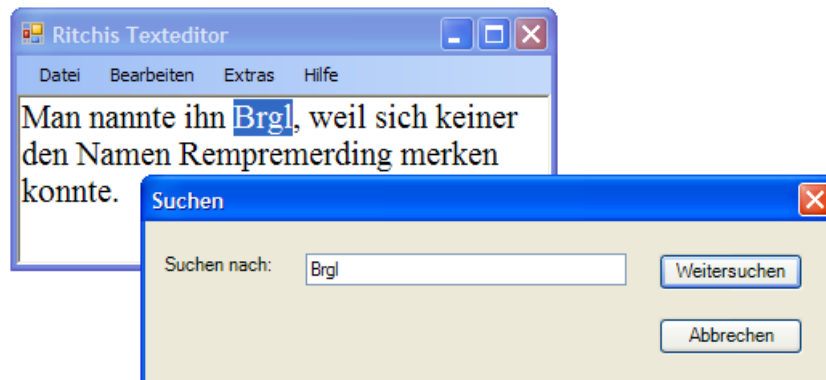
...\BspUeb\WinForms\Dialogfenster\Modal\Übernehmen

18.2 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 2007b, S. 217). Im aktuellen Abschnitt geht es also um die generelle (nicht durch Modalität eingeschränkte) Kommunikation zwischen den Fensterobjekten einer Anwendung.

18.2.1 Konfiguration

Unser seit Abschnitt 16.5 im Aufbau befindlicher Editor soll nun mit einem einfachen Suchdialog ausgestattet werden:



Wir definieren dazu die von **Form** abstammende Klasse `SearchDialog`:

```
using System;
using System.Windows.Forms;
using System.Drawing;
using System.ComponentModel;

class SearchDialog : Form {
    TextBox tbSearchText;

    public event EventHandler FindNext;

    public SearchDialog(Form owner) {
        Text = "Suchen";
        MaximizeBox = false;
        MinimizeBox = false;
        ShowInTaskbar = false;
        FormBorderStyle = FormBorderStyle.FixedDialog;
        ClientSize = new Size(430, 100);
        StartPosition = FormStartPosition.Manual;
        Location = ActiveForm.DesktopLocation +
            new Size(ActiveForm.ClientSize.Width/5, ActiveForm.ClientSize.Height/2);
        Owner = owner;

        Label lblSearch = new Label();
        lblSearch.Text = "Suchen nach:";
        lblSearch.Parent = this;
        lblSearch.Location = new Point(10, 20);
        lblSearch.Width = 80;

        tbSearchText = new TextBox();
        tbSearchText.Parent = this;
        tbSearchText.Location = new Point(100, 20);
        tbSearchText.Size = new Size(200, Font.Height);
    }
}
```



```

        Button cmdNext = new Button();
        cmdNext.Text = "Weitersuchen";
        cmdNext.Parent = this;
        cmdNext.Location = new Point(320, 20);
        cmdNext.Width = 90;
        AcceptButton = cmdNext;
        cmdNext.Click += new EventHandler(ButtonNextOnClick);

        Button cmdCancel = new Button();
        cmdCancel.Text = "Abbrechen";
        cmdCancel.Parent = this;
        cmdCancel.Location = new Point(320, 60);
        cmdCancel.Width = 90;
        CancelButton = cmdCancel;
        cmdCancel.Click += new EventHandler(ButtonCancelOnClick);
    }

    . . .
}

```

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).

Weil in der Windows-Taskleiste nur Hauptfenstern erscheinen sollten, wird die Eigenschaft **ShowInTaskbar** auf den Wert **false** gesetzt:

```
ShowInTaskbar = false;
```

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, setzt man die **Owner**-Eigenschaft des Dialogs auf das Hauptfenster der Anwendung, z.B. über einen Konstruktor-Parameter:

```
Owner = owner;
```

Wie der in Abschnitt 18.1 entwickelte Optionsdialog soll der Suchdialog ...

- beim ersten Auftritt eine Position in Relation zum Hauptfenster wählen,
- bei späteren Auftritten die vorherige Position verwenden.

Dieses Verhalten wird folgendermaßen realisiert:

- Während einer Editor-Sitzung kommt stets dasselbe **SearchDialog**-Objekt zum Einsatz. Zwischen zwei Einsätzen wird das Fenster nicht geschlossen, sondern lediglich versteckt, wobei es seine Position natürlich nicht vergisst. Was genau zu tun ist, erfahren Sie in Abschnitt 18.2.2.
- Um die Position beim ersten Auftritt festzulegen, wird im **SearchDialog**-Konstruktor dieselbe Technik gewählt wie im **OptionsDialog**-Konstruktor (siehe Abschnitt 18.1.1).

Um den Suchdialog im Hauptfenstermenü anzubieten, wird im **Bearbeiten**-Untermenü das Item **Suchen** ergänzt (vgl. Abschnitt 17):

```

ToolStripMenuItem mitEdit, mitFind;

. . .
bmp = new Bitmap("find.bmp");
bmp.MakeTransparent(bmp.GetPixel(0, 0));
mitFind = new ToolStripMenuItem("&Suchen", bmp, MitFindOnClick);
mitFind.ShortcutKeys = Keys.Control | Keys.F;

```

In der zugehörigen **Click**-Behandlungsmethode wird nötigenfalls ein Objekt der Klasse `SearchDialog` erzeugt:

```
protected void MitFindOnClick(object sender, EventArgs e) {
    if (editor.Text.Length == 0)
        return;
    if (searchDialog == null) {
        searchDialog = new SearchDialog();
        searchDialog.FindNext += new EventHandler(SearchDialogOnFindNext);
    }
    searchDialog.Show();
}
```

Trotz Item-Desaktivierung in der **DropDownOpening**-Behandlungsmethode zum **Bearbeiten**-Untermenü

```
if (editor.Text.Length > 0) {
    mitFind.Enabled = true;
} else {
    mitFind.Enabled = false;
}
```

kann `MitFindOnClick()` auch bei leerer Editor-Komponente über die Tastenkombination **Strg+F** aufgerufen werden. Eine entsprechende Abfrage verhindert aber den Auftritt des Suchdialogs. In `MitFindOnClick()` enthaltene Details zum Auftritt des Suchdialogs (siehe Abschnitt 18.2.2) und zu seiner Kommunikation mit dem Hauptfenster (siehe Abschnitt 18.2.3) werden später erläutert.

Im Konstruktor des Hauptfensters wird mit folgender Anweisung

```
editor.HideSelection = false;
```

dafür gesorgt, dass eine Textmarkierung auch dann sichtbar bleibt, wenn ein anderes Fenster (speziell: der Suchdialog) aktiv ist. Beim Durchsuchen des Textes soll nämlich eine Trefferstelle durch Markieren gekennzeichnet werden (siehe Screenshot am Anfang des Abschnitts).

18.2.2 Auftritt und Abgang

Anstelle der zur Anzeige von *modalen* Dialogen erforderlichen **Form**-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 **Suchen**:

```
suchDialog.Show();
```

Sie macht das Dialogfenster sichtbar und kehrt dann sofort zurück.

Die zum Beenden von *modalen* Dialogen wichtige **Form**-Eigenschaft **DialogResult** (siehe Abschnitt 18.1.3) spielt bei nicht-modalen Dialogen *keine* Rolle. Letztere werden häufig über die **Form**-Methode **Close()** im Rahmen einer Ereignisbehandlungsmethode beendet, z.B.:

```
protected void ButtonCancelOnClick(object sender, EventArgs ea) {
    Close();
}
```

Auch bei einem Mausklick auf die Schließen-Schaltfläche in der Titelzeile eines nicht-modalen Dialogfensters wird per Voreinstellung die **Close()**-Methode ausgeführt. Es ist zu beachten, dass diese Methode gründlich aufräumt und alle mit einem Dialogfenster verbundenen Ressourcen freigibt, sodass dieses Fenster anschließend nicht mehr mit **Show()** angezeigt werden kann.

In unserem Editor soll aber nur *ein* Objekt der Klasse `SearchDialog` (samt zugehörigem Fenster) verwendet werden. Neben einem kleinen (vermutlich nicht spürbaren) Spareffekt gewinnen wir den Vorteil, dass die Fensterposition des Dialogfensters ohne Aufwand zwischen zwei Auftritten

erhalten bleibt. Daher *verstecken* wir das Suchfenster per **Hide()**-Aufruf, wenn es vorübergehend nicht mehr benötigt wird, statt es zu schließen:

```
protected void ButtonCancelOnClick(object sender, EventArgs ea) {
    HideSearchDialog();
}

protected void HideSearchDialog() {
    tbSearchText.Focus();
    tbSearchText.SelectAll();
    Owner.Activate();
    Hide();
}
```

Mit

```
Owner.Activate();
```

wird sichergestellt, dass nach dem Verschwinden des Dialogfensters dessen Besitzer (also das Editor-Hauptfenster) das aktive (im Vordergrund liegende und den Eingabefokus besitzende) Fenster ist. Außerdem sorgen wir im Hinblick auf den möglichen weiteren Auftritt des Suchdialogs dafür, dass sein Textfeld den Eingabefokus erhält und vorhandener Text markiert wird. Dies erleichtert eine Änderung der Suchzeichenfolge.

Damit das Dialogfenster auch bei einem Mausklick auf das Schließkreuz seiner Titelzeile nicht zu Schaden kommt, erweitern wir die Klassendefinition um folgende **OnClosing()**-Überschreibung:

```
protected override void OnClosing(CancelEventArgs cea) {
    cea.Cancel = true;
    HideSearchDialog();
}
```

Mit dem Wert **true** für die **CancelEventArgs**-Eigenschaft **Cancel** verhindert man das Schließen des Fensters durch die CLR.

18.2.3 Kommunikation mit dem Anwendungsfenster

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 18.2.1 war die **Click**-Behandlungsmethode zum Editor-Menüitem **Bearbeiten > Suchen** zu sehen. Dort wird nötigenfalls ein **SearchDialog**-Objekt erzeugt und die Editor-Methode **SearchDialogOnFindNext()** beim **FindNext**-Ereignis des Suchdialogs registriert:

```
searchDialog.FindNext += new EventHandler(SearchDialogOnFindNext);
```

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;

protected void ButtonNextOnClick(object sender, EventArgs ea) {
    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 `SearchDialogOnFindNext()` ermittelt der Editor die Suchzeichenfolge und ruft seine Methode `FindText()` auf:

```
protected void SearchDialogOnFindNext(object sender, EventArgs e) {
    searchText = ((SearchDialog)sender).SearchText;
    FindNext();
}
```

In der Methode `FindText()`

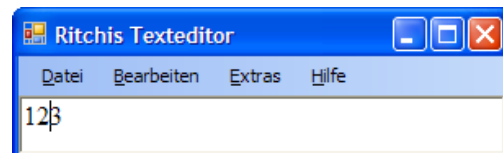
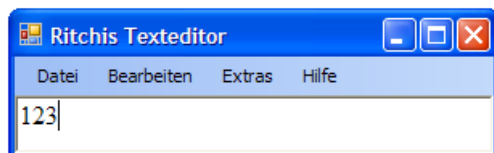
```
protected void FindText() {
    int start = editor.SelectionStart + editor.SelectionLength;
    int hitPosition;
    if (start >= editor.Text.Length ||
        (hitPosition = editor.Find(searchText, start, RichTextBoxFinds.None)) < 0) {
        MessageBox.Show("'" + searchText + "' kann nicht gefunden werden.",
            searchDialog.Text);
    }
}
```

wird die eigentliche Sucharbeit von der **RichTextBox**-Methode **Find** verrichtet.

Es ist erforderlich (und auch üblich), mit der Suche *hinter* dem Ende der aktuellen Markierung zu starten (ab `editor.SelectionStart + editor.SelectionLength`). Weil **Find()** eine Trefferstelle per Voreinstellung markiert, käme die Methode bei einer Suche ab Markierungsanfang nicht über den ersten Treffer hinaus. Die Zusatzbedingung

```
start >= editor.Text.Length
```

verhindert, dass **Find()** bei einer Startposition hinter dem Textende am Textanfang weitersucht und so ein inkonsistentes Verhalten zeigt. Ohne Zusatzbedingung würde in der linken Situation die „2“ gefunden, in der rechten aber nicht:



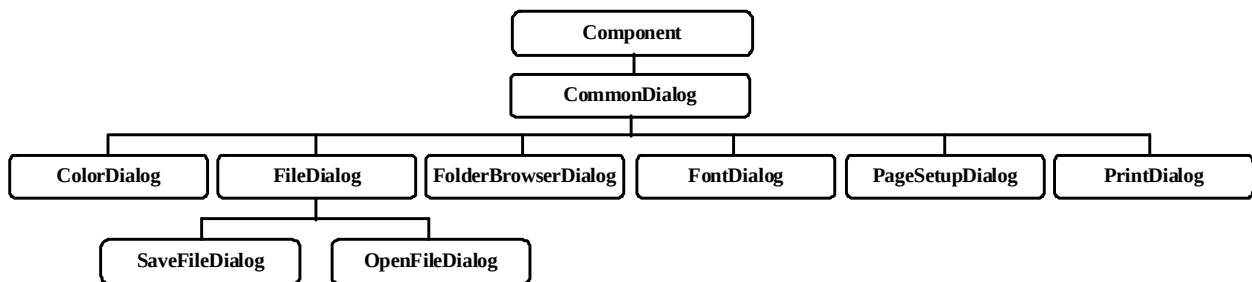
Über den dritten Parameter mit dem Datentyp **RichTextBoxFinds** lässt sich die Arbeitsweise der Suchmethode beeinflussen:

RichTextBoxFinds-Wert	Beschreibung
WholeWord	Es kommen nur ganze Wörter (durch Leerzeichen begrenzt) als Trefferstellen in Frage.
None	Keine Einschränkung der Treffer auf ganze Wörter.
MatchCase	Bei Trefferstellen muss auch die Groß- und Kleinschreibung mit der Suchzeichenfolge übereinstimmen.
NoHighlight	Die voreingestellte Markierung einer Trefferstelle wird abgeschaltet.
Reverse	Es wird vom Textende bis zu der vom zweiten Parameter definierten Position gesucht.

Den vollständigen Quellcode zum aktuellen Entwicklungsstand des Editors finden Sie im Ordner
... \BspUeb \WinForms \Dialogfenster \Nicht-Modal

18.3 Standarddialoge

Im Sinn einer konsistenten Bedienoberfläche und einer einfachen Anwendungsentwicklung bietet Windows seinen Benutzern und Entwicklern für Routineaufgaben wie das Öffnen einer Datei oder die Auswahl einer Schriftart etc. Standarddialoge an, die natürlich auch in der WinForms-Bibliothek verfügbar sind:



Wir haben die WinForms-Standarddialoge zur Schriftart- bzw. Farbauswahl bereits mehrfach eingesetzt und sind dabei mit minimalem Lernaufwand zu brauchbaren Ergebnissen gekommen. In diesem Abschnitt folgen ergänzende Hinweise zu den beteiligten Klassen **FontDialog** bzw. **ColorDialog**. Außerdem lernen wir die Klassen **SaveFileDialog** und **OpenFileDialog** zum Speichern bzw. Öffnen von Dateien kennen. Mit den Klassen **PrintDialog**, **PageSetupDialog** und **PrintPreviewDialog** (Basisklasse: **Form**) beschäftigen wir uns im Abschnitt 21 über das Drucken. Die Klasse **FolderBrowserDialog** unterstützt das Erstellen und/oder Auswählen eines Ordners (z.B. als voreingestellter Ablageort für Dokumente).

Die Standarddialoge werden stets *modal* per **ShowDialog()** aufgerufen. Eine alternative Methode zum nicht-modalen Aufruf fehlt in den zugehörigen Klassen.

Als Anwendungsbeispiel dient auch im aktuellen Abschnitt der Editor auf **RichTextBox**-Basis, den wir seit Abschnitt 16.5 allmählich zu einem praxistauglichen Programm entwickeln.

18.3.1 FontDialog

Mit dem Schriftauswahldialog haben wir uns schon in Abschnitt 15.7.2 ausführlich beschäftigt. Ist mit dem mehrfachen Einsatz zu rechnen, sollte man Ressourcen schonend ein *einziges* **FontDialog**-Objekt per Instanzvariable verwalten, z.B. folgendermaßen im Konstruktor des Hauptfensters deklariert:

```
FontDialog fontDialog;
```

Zur Konfiguration eines Schriftauswahldialogs stehen etliche Eigenschaften zur Verfügung, z.B.:

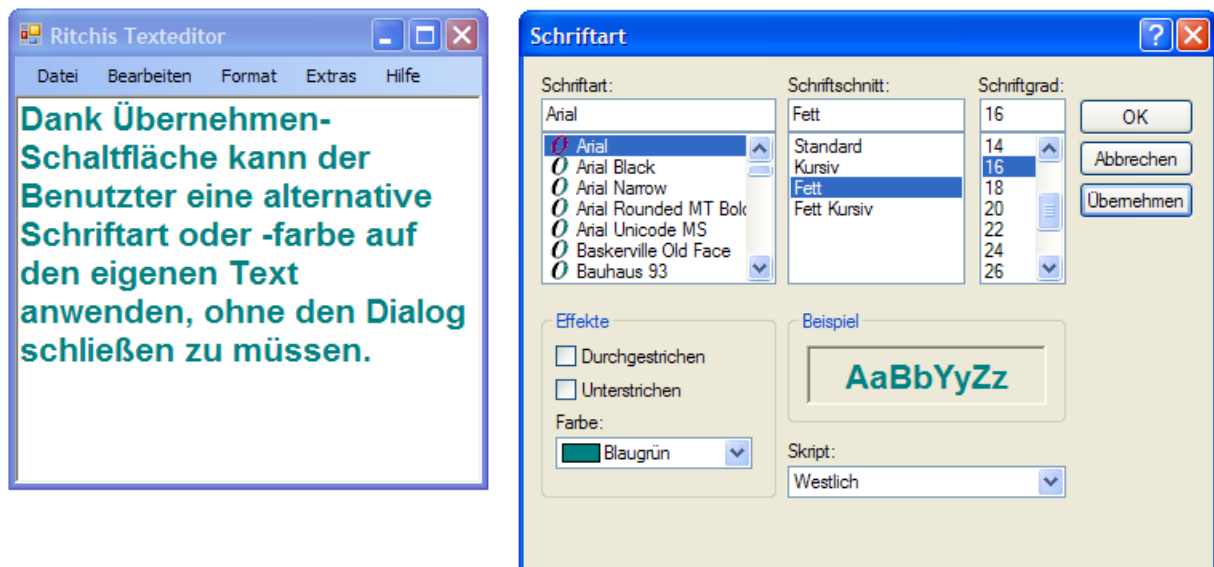
- Über die **Font**-Eigenschaft werden wir im Editor-Beispiel dafür sorgen, dass der Dialog mit der aktuellen Schriftart der **RichTextBox**-Komponente startet.
- Mit der **ShowColor**-Eigenschaft wird festgelegt, ob es eine Möglichkeit zur Auswahl der Schriftfarbe geben soll.
- Erhält die **ShowApply**-Eigenschaft den Wert **true**, dann erscheint im Standarddialog die Schaltfläche **Übernehmen** (vgl. Abschnitt 18.1.4). In diesem Fall muss eine geeignete Behandlungsmethode beim **Apply**-Ereignis des Standarddialogs registriert werden, weil sich die Benutzer ansonsten über die Nutzlosigkeit des **Übernehmen**-Schalters beklagen.
- Mit der Eigenschaft **ShowEffects** (Voreinstellung **true**) entscheidet man über den Auftritt der Kontrollkästchen für das Durch- bzw. Unterstreichen.
- Per **ShowHelp** (Voreinstellung **false**) legt man fest, ob ein **Hilfe**-Schalter erscheinen soll, für den ggf. aber auch eine Behandlungsroutine registriert werden muss.

In unserem Editor wird das **FontDialog**-Objekt in der **Click**-Behandlungsmethode zum Menüitem **Schriftart** zum Einsatz gebracht. Nötigenfalls wird es zuvor erzeugt und initialisiert:

```
protected void MitFontOnClick(object sender, EventArgs e) {
    if (fontDialog == null) {
        fontDialog = new FontDialog();
        fontDialog.Font = editor.Font;
        fontDialog.ShowColor = true;
        fontDialog.Color = editor.ForeColor;
        fontDialog.ShowApply = true;
        fontDialog.Apply += new EventHandler(FontDialogOnApply);
    }
    if (fontDialog.ShowDialog() == DialogResult.OK) {
        Font oldFont = editor.Font;
        editor.Font = new Font(fontDialog.Font.FontFamily,
            (float)Math.Round(fontDialog.Font.SizeInPoints),
            fontDialog.Font.Style);
        oldFont.Dispose();
        editor.ForeColor = fontDialog.Color;
    }
}
```

Nur beim Rückgabewert **DialogResult.OK** werden die Eigenschaften **Font** und **Color** vom Standarddialog in die passenden Hauptfenster-Eigenschaften übernommen. Mit dem Grund für die umständliche **Font**-Übernahme haben wir uns in Abschnitt 15.7.2 beschäftigt. Um GDI-Ressourcen zu schonen (vgl. Abschnitt 15.2), wird für das obsolekte **Font**-Objekt die **Dispose()**-Methode aufgerufen.

Dank **Übernehmen**-Schaltfläche kann der Benutzer eine alternative Schriftart oder -farbe auf den Text anwenden, ohne den Schriftart-Dialog schließen zu müssen:



Der **Apply**-Ereignishandler übernimmt die Eigenschaftswerte vom Standarddialog:

```
protected void FontDialogOnApply(object sender, EventArgs e) {
    Font oldFont = editor.Font;
    editor.Font = new Font(fontDialog.Font.FontFamily,
        (float)Math.Round(fontDialog.Font.SizeInPoints), fontDialog.Font.Style);
    oldFont.Dispose();
    fontDialog.Font.Dispose();
    editor.ForeColor = fontDialog.Color;
}
```

In früheren Versionen unseres Editors dienten die Schriftart-Menüitems hauptsächlich dazu, Menüoptionen zu demonstrieren. Für die nun integrierte Standardlösung wählen wir auch den üblichen Ort im Menü: **Format > Schriftart**:


```
ToolStripMenuItem mitFormat, mitFont;

mitFont = new ToolStripMenuItem("&Schriftart...", null, MitFontOnClick);
mitFormat.DropDownItems.Add(mitFont);
```

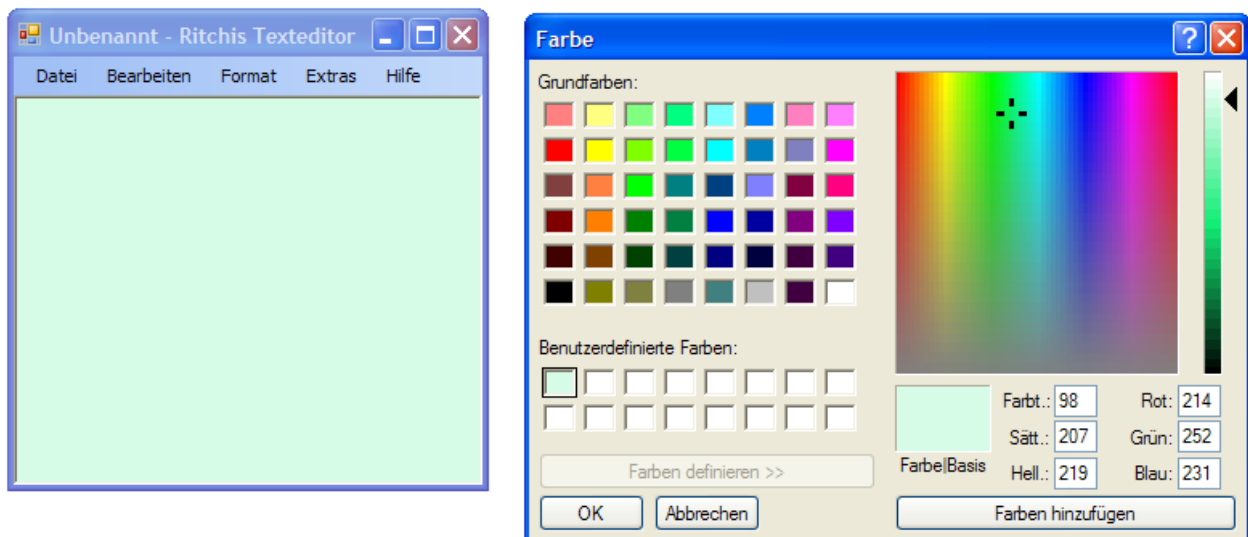
Beim Speichern im RTF-Format (siehe Abschnitt 18.3.3) wird die eingestellte Schriftart (**RichText-Box-Eigenschaft Text**) zum Bestandteil des Dokuments, so dass ein **Format**-Untermenü mit **Schriftart**-Item sinnvoll ist.

18.3.2 ColorDialog

Über das **Extras**-Menü soll unser Editor mit Hilfe eines **ColorDialog**-Objekts die Möglichkeit zur Wahl einer Hintergrundfarbe bieten. Wie beim Schriftartendialog sorgen wir auch beim Farbauswahldialog dafür, dass pro Editor-Einsatz nur *ein* **ColorDialog**-Objekt verwendet wird, verwaltet über eine Instanzvariable der Editor-Klasse

```
ColorDialog colorDialog;
```

Weil die Option zum Mischen benutzerdefinierter Farben eingeräumt werden soll, spricht neben dem Ressourcen-Spareffekt ein noch gewichtiger Grund für die Verwendung eines *einzigen* **ColorDialog**-Objekts. Hat ein Benutzer im erweiterten Farbdialog mehrere (maximal 16) Farben definiert, will er diese Kreationen zweifellos bei allen weiteren Aufrufen des Dialogs zur direkten Auswahl unverändert vorfinden, z.B.:



Statt umständlich einen Array mit den 16 Farbdefinitionen zu speichern und zu reaktivieren, erreicht man denselben Gedächtniseffekt durch Verwendung eines einzigen **ColorDialog**-Objekts. Soll das Gedächtnis die aktuelle Sitzung überdauern, ist allerdings ein Speichern der Farbinformationen (z.B. in einer XML-Konfigurationsdatei oder in der Windows-Registrierdatenbank) unvermeidlich (siehe unten).

Für die Konfiguration vor dem (ersten) Einsatz kommen beim Farbauswahlstandarddialog folgende Eigenschaften in Frage:

- Per **Color**-Eigenschaft setzt man die initiale Farbwahl.
- Über die Eigenschaft **AllowFullOpen** (Voreinstellung: **true**) wird geregelt, ob der Benutzer eigene Farben mischen darf.
- Mit der Eigenschaft **FullOpen** (Voreinstellung: **false**) legt man fest, ob der Farbauswahldialog schon beim Start den Farbmischer anzeigt.
- Per **ShowHelp** (Voreinstellung **false**) legt man fest, ob ein **Hilfe**-Schalter erscheinen soll, für den ggf. aber auch eine Behandlungsroutine erstellt und registriert werden muss.

Eine **Übernehmen**-Schaltfläche wird vom Farbwahl-Standarddialog *nicht* unterstützt.

In unserem Editor wird das **ColorDialog**-Objekt in der **Click**-Behandlungsmethode zum renovieren Menüitem **Extras > Hintergrundfarbe** per **ShowDialog()** zum Einsatz gebracht. Nötigenfalls wird es zuvor erzeugt und initialisiert:

```
protected void MitBackColorOnClick(object sender, EventArgs e) {
    if (colorDialog == null) {
        colorDialog = new ColorDialog();
        colorDialog.Color = editor.BackColor;
    }
    if (colorDialog.ShowDialog() == DialogResult.OK) {
        editor.BackColor = colorDialog.Color;
    }
}
```

18.3.3 SaveFileDialog

Vor dem (ersten) Einsatz eines Objekts aus der Klasse **SaveFileDialog** zum Ermitteln einer Zieldatei für das Speichern wird man in der Regel über seine **Filter**-Eigenschaft festlegen, zwischen welchen Dateitypen der Benutzer wählen kann. In unserem Editor-Beispiel sollen Text- und RTF-Dateien unterstützt werden:

```
String filter = "Textdokument (*.txt)|*.txt|RTF-Dokument (*.rtf)|*.rtf";
```

Über die ganzzahlige Eigenschaft **FilterIndex** wird der initiale Typ festgelegt und die Wahl des Benutzers übermittelt, wobei die Zählung ausnahmsweise bei **1** beginnt.

Weitere Gestaltungsmöglichkeiten für **SaveFileDialog**-Objekte:

- Mit der Eigenschaft **AddExtension** (Voreinstellung **true**) legt man fest, ob an einen Dateinamen nötigenfalls die zum aktuell gewählten Dateityp gehörige Namenserweiterung automatisch angehängt werden soll.
- Per Voreinstellung ändert der Benutzer durch sein Navigationsverhalten im **SaveFileDialog** das aktuelle Verzeichnis der Anwendung. Um dies zu verhindern, muss man die Eigenschaft **RestoreDirectory** (Voreinstellung **false**) auf den Wert **true** setzen.
- Hat die Eigenschaft **CreatePrompt** (Voreinstellung **false**) den Wert **true**, wird der Benutzer ggf. gefragt, ob eine nicht vorhandene Zieldatei erstellt werden soll.
- Hat die Eigenschaft **OverwritePrompt** den voreingestellten Wert **true**, wird der Benutzer ggf. gefragt, ob eine bereits vorhandene Zieldatei überschrieben werden soll.
- Per **ShowHelp** (Voreinstellung **false**) legt man fest, ob ein **Hilfe**-Schalter erscheinen soll, für den ggf. aber auch eine Behandlungsroutine erstellt und registriert werden muss.

Die folgende Methode ist im Editor-Beispiel für die Menüitems **Datei > Speichern** und **Datei > Speichern unter** zuständig. Beim einfachen Speichern wird sie nur dann tätig, wenn sich der Text geändert hat, wobei die **RichTextBox**-Eigenschaft **Modified** wertvolle Dienste leistet:

```
protected void MitSaveOnClick(object sender, EventArgs e) {
    if (sfDialog == null) {
        sfDialog = new SaveFileDialog();
        sfDialog.Filter = filter;
    }
    bool sas = (sender == mitSaveAs);
    if (!(sas || editor.Modified))
        return;
}
```



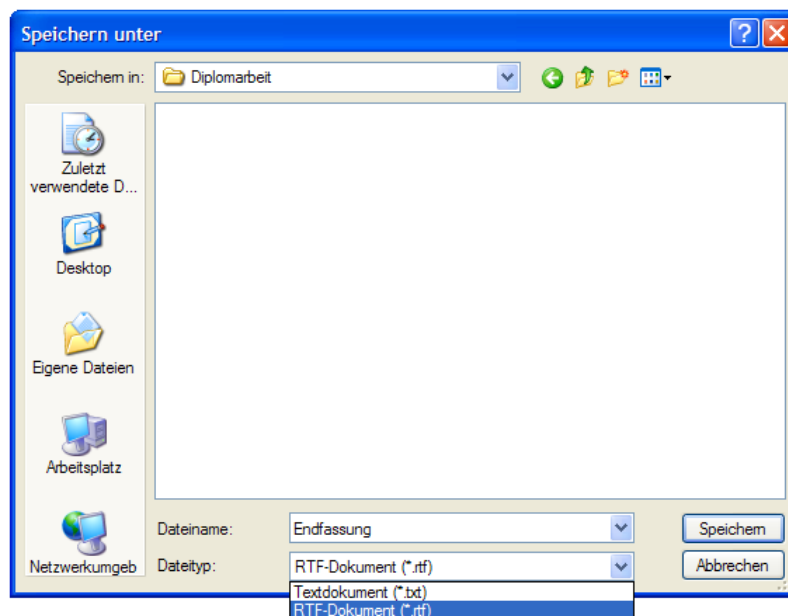
```

if (fileName == null || sas) {
    if (sfDialog.ShowDialog() != DialogResult.OK)
        return;
    fileName = sfDialog.FileName;
    fileType = sfDialog.FilterIndex;
    Text = Path.GetFileName(fileName) + " - " + appName;
}
FileStream fs = null;
try {
    fs = new FileStream(fileName, FileMode.Create);
    if (fileType == 1)
        editor.SaveFile(fs, RichTextBoxStreamType.PlainText);
    else
        editor.SaveFile(fs, RichTextBoxStreamType.RichText);
    editor.Modified = false;
} catch (Exception ex) {
    MessageBox.Show("Fehler beim Speichern in die Datei " +
        fileName + "\n" + ex.Message, Text);
} finally {
    fs.Close();
}
}

```

Die wesentliche Arbeit leistet die **RichTextBox**-Methode **SaveFile()**, die per Parameter über den vom Benutzer gewählten Dateityp informiert werden muss (**RichTextBoxStreamType.PlainText** oder **RichTextBoxStreamType.RichText**).

Ist beim Aufruf über das Menüitem **Datei > Speichern**, das Sie in einer Übungsaufgabe ergänzen sollen (siehe Abschnitt 18), bereits ein Dateiname in der Instanzvariablen `fileName` gespeichert, dann wird kein **SaveFileDialog** benötigt. Ansonsten verhält sich die Methode wie bei **Datei > Speichern unter**:



Damit die **SaveFileDialog**-Methode **ShowDialog()** im Debug-Modus ausgeführt werden kann, muss die **Main()**-Methode unbedingt mit dem Attribut **STAThread** dekoriert werden, das wir seit Abschnitt 11 einer Microsoft-Empfehlung folgend bei WinForms-Anwendungen generell verwenden:

```

[STAThread]
static void Main() {
    Application.EnableVisualStyles();
    Application.Run(new RitchisTexteditor());
}

```

Ein **RichTextBox**-Steuerelement unterstützt in seinen Methoden **SaveFile()** und **LoadFile()** (siehe unten) die folgenden, durch Werte der Enumeration **RichTextBoxStreamType** identifizierten Dateiformate, wobei sich unser Editor auf **PlainText** und **RichText** beschränkt:

RichTextBoxStreamType-Wert	Beschreibung
PlainText	Es entsteht eine Textdatei, wobei OLE-Objekte (<i>Object Linking and Embedding</i>) durch Leerzeichen ersetzt werden.
RichNoOleObjs	Es entsteht eine RTF-Datei, wobei OLE-Objekte (<i>Object Linking and Embedding</i>) durch Leerzeichen ersetzt werden.
RichText	Es entsteht eine RTF-Datei.
TextTextOleObjs	Es entsteht eine Textdatei mit einer Textdarstellung von OLE-Objekten.
UnicodePlainText	Es entsteht eine UNICODE-Textdatei, wobei OLE-Objekte durch Leerzeichen ersetzt werden.

18.3.4 OpenFileDialog

Mit einem Standarddialog aus der Klasse **OpenFileDialog** erfragt man beim Benutzer den Namen einer vorhandenen Datei. Zur Vorbereitung eines **OpenFileDialog**-Aufrufs ist analog zum Vorgehen bei **SaveFileDialog** (siehe Abschnitt 18.3.3) in der Regel ein Filter-String mit den unterstützten Dateitypen zu erstellen, z.B.:

```
String filter = "Textdokument (*.txt)|*.txt|RTF-Dokument (*.rtf)|*.rtf";
```

Über die ganzzahlige Eigenschaft **FilterIndex** wird der initiale Typ festgelegt und die Wahl des Benutzers übermittelt, wobei die Zählung ausnahmsweise bei **1** beginnt.

Weitere Gestaltungsmöglichkeiten für **OpenFileDialog**-Objekte:

- Mit der Eigenschaft **AddExtension** (Voreinstellung **true**) legt man fest, ob an einen Dateinamen nötigenfalls die zum aktuell gewählten Dateityp gehörige Namenserverweiterung automatisch angehängt werden soll.
- Hat die Eigenschaft **Multiselect** (Voreinstellung **false**) den Wert **true**, kann der Benutzer *mehrere* Dateien auswählen.
- Per Voreinstellung ändert der Benutzer durch sein Navigationsverhalten im **OpenFileDialog** das aktuelle Verzeichnis der Anwendung. Um dies zu verhindern, muss man die Eigenschaft **RestoreDirectory** (Voreinstellung **false**) auf den Wert **true** setzen.
- Hat die Eigenschaft **CheckFileExists** den voreingestellten Wert **true**, wird der Benutzer gewarnt, wenn die angegebene Datei nicht existiert.
- Per **ShowHelp** (Voreinstellung **false**) legt man fest, ob ein **Hilfe**-Schalter erscheinen soll, für den ggf. aber auch eine Behandlungsroutine erstellt und registriert werden muss.

In der **Click**-Behandlungsmethode zum Menüitem **Datei > Öffnen** unseres Editors leistet die **RichTextBox**-Methode **LoadFile()** die wesentliche Arbeit. Sie wird per Parameter über den vom Benutzer gewählten Dateityp informiert:

```

protected void MitOpenOnClick(object sender, EventArgs e) {
    if (ofDialog == null) {
        ofDialog = new OpenFileDialog();
        ofDialog.Filter = filter;
    }
    if (UnsavedText("Wollen Sie trotzdem eine Datei öffnen " +
        "und den aktuellen Text ersetzen?"))
        return;
    if (ofDialog.ShowDialog() == DialogResult.OK) {
        FileStream fs = null;
        try {
            fs = new FileStream(ofDialog.FileName, FileMode.Open,
                FileAccess.Read, FileShare.Read);
            fileType = ofDialog.FilterIndex;
            if (fileType == 1)
                editor.LoadFile(fs, RichTextBoxStreamType.PlainText);
            else
                editor.LoadFile(fs, RichTextBoxStreamType.RichText);
            fileName = ofDialog.FileName;
            Text = Path.GetFileName(fileName) + " - " + appName;
            editor.Modified = false;
        } catch (Exception ex) {
            MessageBox.Show("Fehler beim Öffnen der Datei " +
                fileName + "\n" + ex.Message, Text);
        } finally {
            fs.Close();
        }
    }
}

```

Zunächst wird geprüft, ob im Editor ungesicherter Text beim Öffnen eines anderen Dokuments gefährdet ist. Diese Prüfung erledigt die folgende Methode unserer Editor-Klasse mit Hilfe der **RichTextBox**-Eigenschaft **Modified**:

```

protected bool UnsavedText(string question) {
    if (editor.Modified && editor.Text.Length > 0)
        if (MessageBox.Show("Es ist ungesicherter Text vorhanden.\n" + question,
            appName, MessageBoxButtons.YesNoCancel,
            MessageBoxIcon.Exclamation,
            MessageBoxDefaultButton.Button2) != DialogResult.Yes)
            return true;
        else
            return false;
    return false;
}

```

Gegen Datenverlust beim Beenden unseres Editors hilft die folgende Behandlungsmethode zum **Form**-Ereignis **FormClosing**:

```

void FormOnClosing(object sender, CancelEventArgs cea) {
    if (UnsavedText("Wollen Sie das Programm trotzdem beenden?"))
        cea.Cancel = true;
    else
        Application.Exit();
}

```

Damit die **OpenFileDialog**-Methode **ShowDialog()** im Debug-Modus ausgeführt werden kann, muss die **Main()**-Methode mit dem Attribut **STAThread** dekoriert werden (vgl. Abschnitt 11):

```
[STAThread]
static void Main() {
    Application.EnableVisualStyles();
    Application.Run(new RitchisTexteditor());
}
```

Den vollständigen Quellcode zum aktuellen Entwicklungsstand des Editors finden Sie im Ordner
...\BspUeb\WinForms\Dialogfenster\Standardddialoge

18.4 Übungsaufgaben zu Kapitel 18

1) Erweitern Sie unseren Editor ausgehend vom Entwicklungsstand in

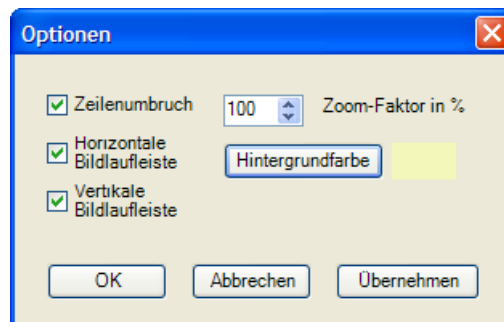
...\BspUeb\WinForms\Dialogfenster\Standardddialoge

um das **Bearbeiten**-Menüitem **Weitersuchen** (mit Tastaturbefehl **F3**), wobei die letzte Suchzeichenfolge erneut verwendet werden soll.

Außerdem fehlen im **Datei**-Menü noch die Standarditems **Neu** und **Speichern** (im Unterschied zu **Speichern unter**).

Die in Abschnitt 18.3.4 vorgestellte Methode `UnsavedText()` sollte auch in der Behandlungsmethode zum Menüitem **Datei > Neu** dazu verwendet werden, den Benutzer vor Textverlusten zu warnen.

2) Die Modifikation der Editor-Hintergrundfarbe sollte eigentlich über den Optionsdialog erfolgen (statt über das Menüitem **Extras > Hintergrundfarbe**), z.B.:



Nehmen Sie an Ihrer Lösung zu Aufgabe 1 die nötigen Änderungen vor.

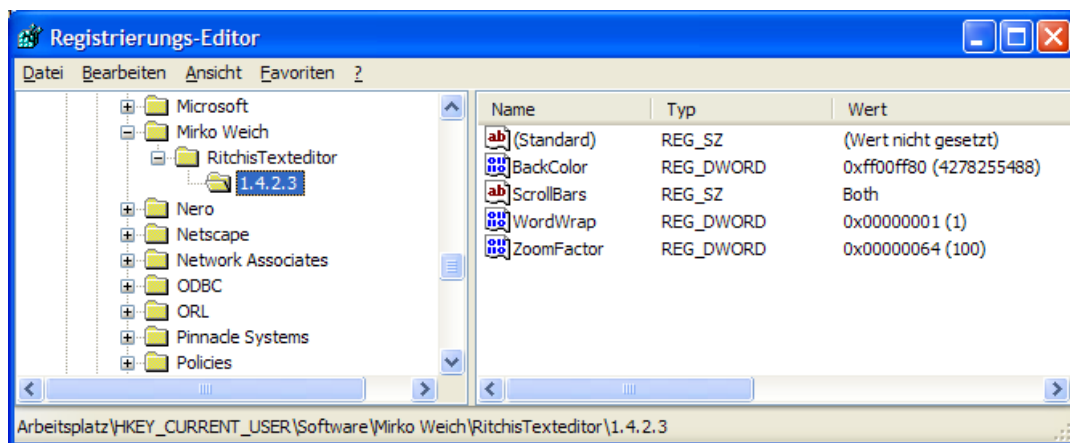
19 Anwendungs- und Benutzereinstellungen

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 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 modernen XML-Format (*Extended Markup Language*). Hier ist eine gekürzte Variante der XML-Konfigurationsdatei zu unserem Editor zu sehen:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <sectionGroup name="userSettings"
      type="System.Configuration.UserSettingsGroup, System, Version=2.0.0.0,
        Culture=neutral, PublicKeyToken=b77a5c561934e089" >
      <section name="RitchisTexteditorSettings"
        type="System.Configuration.ClientSettingsSection, System, Version=2.0.0.0,
          Culture=neutral, PublicKeyToken=b77a5c561934e089"
        allowExeDefinition="MachineToLocalUser" requirePermission="false" />
    </sectionGroup>
    <sectionGroup name="applicationSettings"
      type="System.Configuration.ApplicationSettingsGroup, System,
        Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" >
      <section name="RitchisTexteditorSettings"
        type="System.Configuration.ClientSettingsSection, System, Version=2.0.0.0,
          Culture=neutral, PublicKeyToken=b77a5c561934e089"
        requirePermission="false" />
    </sectionGroup>
  </configSections>
```

¹ Im **Windows**-Ordner einer XP-Installation fanden sich 297 Exemplare.

```

<userSettings>
  <RitchisTexteditorSettings>
    <setting name="BackColor" serializeAs="String">
      <value>Silver</value>
    </setting>
  </RitchisTexteditorSettings>
</userSettings>
<applicationSettings>
  <RitchisTexteditorSettings>
    <setting name="AppName" serializeAs="String">
      <value>Ritchis Texteditor</value>
    </setting>
  </RitchisTexteditorSettings>
</applicationSettings>
</configuration>

```

Im Abschnitt 19 werden die beiden aktuellen Techniken zur Einstellungsverwaltung (XML-Konfigurationsdateien, Registrierungsdatenbank) bei unserem Editor-Projekt eingesetzt.

19.1 Konfigurationsdateien im XML-Format

Das .NET - Framework kennt ab Version 2.0 XML-Konfigurationsdateien für ...

- einzelne Anwendungen
- einzelne Bibliotheken im Global Assembly Cache (GAC)
Über die so genannten *Herausgeberrichtliniendateien* informiert z.B. Kühnel (2008, Abschnitt 6.4.1).

- 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. in:

C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\CONFIG

Die .NET - Versionen 3.0 und 3.5 legen keine eigene Maschinenkonfigurationsdatei an, verwenden also offenbar die Konfigurationsdatei der Version 2.0.

Wir beschränken uns auf Anwendungskonfigurationsdateien.

19.1.1 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)
Anwendungsbezogene Einstellungen (z.B. eine Datenbankverbindungszeichenfolge) können *vom Programm* nur gelesen werden. Änderungen erfordern das Bearbeiten der Konfigurationsdatei. Bei benutzerbezogenen Einstellungen (z.B. Hintergrundfarbe in einem Editor) 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 *Wurzel-element* 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="BackColor" serializeAs="String">
  <value>White</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="RitchisTexteditorSettings"
  type="System.Configuration.ClientSettingsSection, System, Version=2.0.0.0,
  Culture=neutral, PublicKeyToken=b77a5c561934e089"
  allowExeDefinition="MachineToLocalUser" requirePermission="false" />
```

Bisher wurden die .NET - Konfigurationselemente zur Illustration des XML-Dateiformats verwendet. Nun kommen wir zur speziellen Struktur einer typischen .NET - Konfigurationsdatei mit anwendungs- *und* benutzerbezogenen Einstellungen:

- Das Wurzelement **<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 benutzer- bzw. anwendungsbezogenen 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="RitchisTexteditorSettings" ... />
</sectionGroup>
```


enthält Attribute für den Namen (im Beispiel: z.B. `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 Attribute die zuständige Klasse, z.B.:

```
<section name="RitchisTexteditorSettings"
  type="System.Configuration.ClientSettingsSection, System, Version=2.0.0.0,
    Culture=neutral, PublicKeyToken=b77a5c561934e089"
  allowExeDefinition="MachineToLocalUser" requirePermission="false" />
```

In Abschnitt 19.1.3 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 also ein Element namens `<RitchisTexteditorSettings>`, z.B.:

```
<userSettings>
  <RitchisTexteditorSettings>
    .
    .
    .
  </RitchisTexteditorSettings>
</userSettings>
```

- Das benutzer- bzw. anwendungsbezogene `<RitchisTexteditorSettings>` - Element enthält für jede Einstellung ein `<setting>` - Element, z.B.:

```
<RitchisTexteditorSettings>
  <setting name="BackColor" serializeAs="String">
    <value>Silver</value>
  </setting>
</RitchisTexteditorSettings>
```

- Ein `<setting>` - Element enthält
 - Attribute für den Namen und die Serialisierungsmethode einer Einstellung
 - ein `<value>` - Element für den Wert der Einstellung

Das Erstellen von XML-Konfigurationsdateien wird in der Regel durch ein Programm erledigt (z.B. durch eine Entwicklungsumgebung, siehe Abschnitt 19.1.7). 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 (siehe Abschnitt).

19.1.2 Ablage im Dateisystem

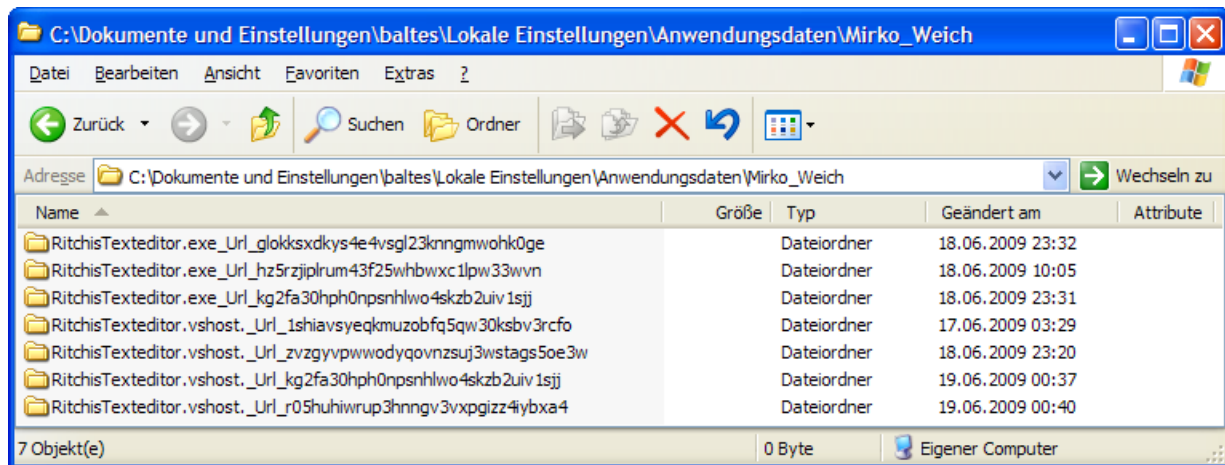
Vom Bezug einer Einstellung hängt der Speicherort ab. Anwendungsbezogene Einstellungen werden in einer Datei verwaltet, die sich im selben Ordner befindet wie das Exe-Assembly und dessen Namen um den Suffix „**.config**“ erweitert übernimmt. Bei einem Exe-Assembly mit dem Namen **prog.exe** heißt die 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.

Weil Dateien mit XML-Format in der Regel Klartext enthalten, ist im Zusammenhang mit Datenbankverbindungszeichenfolgen der Hinweis angebracht, dass SQL-Kontodaten (insbesondere) Passwörter *nicht* in eine Anwendungskonfigurationsdatei gehören.

Die Datei **user.config** mit den benutzerbezogenen Einstellungen landet im Ordner:

```
<Profilordner>\<Hersteller>\<Assembly-Name>_<Evidenz-Typ>_<Evidenz-Hash>\<Version>
```


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 XP:



Im Beispiel lautet der Assembly-Name **RitchisTexteditor.exe**; Hersteller und Version wurden in den folgenden Attribut-Deklarationen festgelegt (vgl. Abschnitt 11.4):

```
[assembly: AssemblyCompany("Mirko Weich")]
[assembly: AssemblyVersion("1.4.2.3")]
```

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 19.1.4).

Laut MSDN-Webseite (Microsoft Developer Network)

<http://msdn.microsoft.com/en-us/library/8eyb2ct1.aspx>

befindet sich die Datei **user.config** bei einem lokalen Windows-Benutzerprofil in einem Ordner, welcher der Eigenschaft **LocalUserAppDataPath** der Klasse **Application** zu entnehmen ist:

Settings File Locations

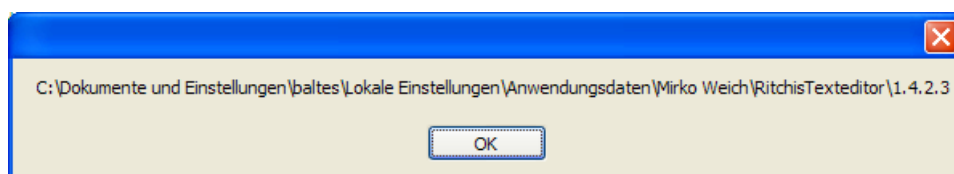
The location of the *app.exe.config* and *user.config* files will differ based on how the application is installed. For a Windows Forms-based application copied onto the local computer, *app.exe.config* will reside in the same directory as the base directory of the application's main executable file, and *user.config* will reside in the location specified by the

Application.LocalUserAppDataPath property.

Ein Zugriff auf die **Application**-Eigenschaft **LocalUserAppDataPath**

```
MessageBox.Show(Application.LocalUserAppDataPath.ToString());
```

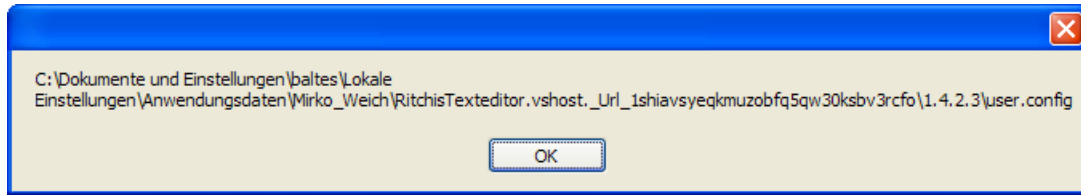
bewirkt im Editor-Beispiel die Erstellung des erwarteten Ordners:



Leider wird die Datei **user.config** jedoch *nicht* hier angelegt. Den tatsächlichen Konfigurationsordner kann man im Programm über die Klasse **ConfigurationManager**

```
Configuration config = ConfigurationManager.OpenExeConfiguration(
    ConfigurationUserLevel.PerUserRoamingAndLocal);
MessageBox.Show(config.FilePath);
```

in Erfahrung bringen:¹



Leider wird der Ordner mit den lokalen Einstellungen auch bei Benutzern einer Windows-Domäne mit einem wandernden (engl.: *roaming*) Windows-Benutzerprofil zur Ablage der Datei **user.config** verwendet. Damit werden die auf einem Rechner vorgenommenen Einstellungen bei der Abmeldung *nicht* auf den Profilserver übertragen und stehen nach der Anmeldung bei einem anderen Rechner nicht zur Verfügung.

Während die benutzerbezogene Einstellungsdatei **user.config** vom zuständigen Objekt in der **Save()**-Methode angelegt wird, muss die anwendungsbezogene Datei **prog.exe.config** vom Software-Hersteller ausgeliefert werden, wobei man aber die Erstellung der Entwicklungsumgebung überlassen kann (siehe Abschnitt 19.1.7).

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

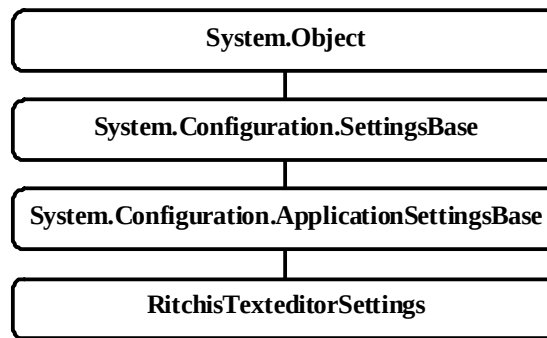
- das Element bzw. die Sektionsgruppe **<userSettings>**
Hier befinden sich die Voreinstellungen für Benutzer, die (noch) keine Konfigurationsdatei **user.config** haben.
- das Element bzw. die Sektionsgruppe **<applicationSettings>**
Hier befinden sich anwendungsbezogene Einstellungen (z.B. Datenbankverbindungszeichenfolgen).

Die Datei **user.config** enthält das Element bzw. die Sektionsgruppe **<userSettings>** mit benutzerbezogenen Einstellungen.

19.1.3 Die Klasse **ApplicationSettingsBase**

Die FCL-Klassen zur Verwaltung von Anwendungseinstellungen befinden sich im Namensraum **System.Configuration**. In WinForms-Anwendungen ist zur Einstellungsverwaltung eine anwendungsspezifische Klasse aus **ApplicationSettingsBase** abzuleiten, die bei unserem Editorprojekt den Namen **RitchisTexteditorSettings** erhalten soll:

¹ Bei Verwendung der Klasse **ConfigurationManager** benötigt der Compiler (das Projekt) einen Verweis auf das Bibliotheks-Assembly **System.Configuration.dll**.



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 auftauchen darf (vgl. Abschnitt 19.1.1). 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 Eigenschaft definiert, um einen einfachen Zugriff auf die Einstellungen zu ermöglichen. In unserem Editorprojekt sollen die Einstellungen des Optionsdialogs (vgl. Abschnitt 18.1) abgespeichert werden. Außerdem soll der Administrator einer Installation die Möglichkeit erhalten, die Titelzeile der Anwendung beliebig festzulegen, so dass benutzer- und anwendungsbezogene Einstellungen vorliegen:

```

using System.Configuration;
using System.Windows.Forms;
using System.Drawing;
class RitchisTexteditorSettings : ApplicationSettingsBase {
    [UserScopedSettingAttribute()]
    [DefaultSettingValueAttribute("True")]
    public bool WordWrap {
        get {
            return ((bool)(this["WordWrap"]));
        }
        set {
            this["WordWrap"] = value;
        }
    }
    [UserScopedSettingAttribute()]
    [DefaultSettingValueAttribute("White")]
    public Color BackColor {
        get {
            return ((Color)(this["BackColor"]));
        }
        set {
            this["BackColor"] = value;
        }
    }
    . . .
    [ApplicationScopedSettingAttribute()]
    public string AppName {
        get {
            return ((string)(this["AppName"]));
        }
    }
}

```

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.:

```
DefaultSettingValueAttribute("White")
```

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, z.B.:

```
get {
    return ((Color)(this["BackColor"]));
}
set {
    this["BackColor"] = value;
}
```

Zu den Extra-Kompetenzen der Klasse **ApplicationSettingsBase** im Vergleich zu ihrer Basisklasse 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 19.1.6)

19.1.4 Lesen und Speichen von Einstellungen

Für den Zugriff auf die in Konfigurationsdateien vorgefundenen Einstellungen benötigen wir ein Objekt unserer **ApplicationSettingsBase**-Ableitung **RitchisTexteditorSettings**. Weil es in mehreren Methoden der Editor-Klasse angesprochen werden soll, deklarieren wir eine Instanzvariable:

```
class RitchisTexteditor : Form {
    RitchisTexteditorSettings applicationSettings;
    . . .
}
```

Für die Übernahme von Einstellungswerten in Programmvariablen eignet sich eine Behandlungsmethode zum Formularereignis **Load**, z.B.:

```
void FormOnLoad(object sender, EventArgs e) {
    applicationSettings = new RitchisTexteditorSettings();
    try {
        editor.WordWrap = applicationSettings.WordWrap;
        editor.ScrollBars = applicationSettings.ScrollBars;
        editor.ZoomFactor = applicationSettings.ZoomFactor;
        editor.BackColor = applicationSettings.BackColor;
        appName = applicationSettings.AppName;
        Text = "Unbenannt - " + appName;
    } catch {
        MessageBox.Show("Fehler beim Lesen der Einstellungen\n" +
            "Beim Beenden können keine Einstellungen gespeichert werden.",
            appName, MessageBoxButtons.OK, MessageBoxIcon.Error);
        applicationSettings = null;
    }
}
```

Mit der **try-catch-finally** - Ausnahmebehandlung werden Störungen bei der Initialisierung des Konfigurationssystems abgefangen (z.B. durch eine blockierte Datei **user.config**). In diesem Fall erhält die Referenzvariable **applicationSettings** den Wert **null**, so dass beim Verlassen der Anwendung das Problem festgestellt und auf ein Sichern der Einstellung verzichtet werden kann (siehe unten). An dieser Stelle sollte man eigentlich zwischen Problemen mit der Datei **prog.exe.config** und Problemen mit der Datei **user.config** differenzieren.

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 **FormClosing**. Unser Editor enthält bereits eine solche Methode, die sich bisher auf die Vermeidung von Textverlusten beschränkt hat. Nun befördern wir zusätzlich benutzerbezogene Einstellungen in die korrespondierenden Eigenschaften des Einstellungsverwaltungs-Objekts und verlangen per **Save()**-Aufruf das Sichern in die Datei **user.config**:

```
void FormOnClosing(object sender, CancelEventArgs cea) {
    if (UnsavedText())
        cea.Cancel = true;
    else {
        if (applicationSettings != null) {
            applicationSettings.WordWrap = editor.WordWrap;
            applicationSettings.ScrollBars = editor.ScrollBars;
            applicationSettings.ZoomFactor = editor.ZoomFactor;
            applicationSettings.BackColor = editor.BackColor;
            applicationSettings.ClientSize = this.ClientSize;
            applicationSettings.Location = this.Location;
            try {
                applicationSettings.Save();
            } catch {
                MessageBox.Show("Fehler beim Sichern der Einstellungen",
                                appName, MessageBoxButtons.OK, MessageBoxIcon.Error);
            }
        }
        Application.Exit();
    }
}
```

Wenn beim Anwendungsstart das Konfigurationssystem nicht initialisiert werden konnte (siehe oben), wird kein Versuch zum Sichern der Einstellungen gemacht.

Die von **ApplicationSettingsBase** abgeleitete Einstellungsklasse kümmert sich nicht selbst um das Speichern und Laden von Einstellungen, sondern verwendet für diese Aufgaben einen **SettingsProvider**, 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).

19.1.5 Bindung von Einstellungen an Eigenschaften von Komponenten

Einstellungen beziehen sich häufig auf Eigenschaften von Komponenten, und meist sind die betroffenen Klassen in der Lage, ihre Eigenschaften so mit Einstellungen (Eigenschaften einer Einstellungsklasse) zu verbinden, dass der Informationsaustausch in beiden Richtungen automatisch ablaufen kann. In unserem Beispiel ersetzen wir den lesenden Zugriff auf die `ZoomFactor`-Einstellung in der Methode `FormOnLoad()`

```
editor.ZoomFactor = applicationSettings.ZoomFactor;
```

und den schreibenden Zugriff in der Methode `FormOnClosing()`

```
applicationSettings.ZoomFactor = editor.ZoomFactor;
```

durch folgende Datenbindung in der Methode `FormOnLoad()`

```
editor.DataBindings.Add(new Binding("ZoomFactor",  
    applicationSettings, "ZoomFactor"));
```

Wie in Abschnitt 19.1.7 zu sehen sein wird, ist die Datenbindungstechnik mit Hilfe der Entwicklungsumgebungen sehr bequem zu realisieren.

Einige Komponenten (z.B. **ToolStripItem**) unterstützen allerdings keine Datenbindungen, und bei der **Form**-Eigenschaft **ClientSize** rät Microsoft wegen der Gefahr eines unvorhersehbaren Verhaltens von der Datenbindungstechnik ab.¹

19.1.6 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 19.1.5).
SettingChanging	Das Ereignis tritt <i>vor</i> 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 <i>nach</i> 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:

¹ Siehe <http://msdn.microsoft.com/en-us/library/k4s6c3a0.aspx>

```

applicationSettings.SettingChanging +=
    new SettingChangingEventHandler(AppSettOnSettChanging);

void AppSettOnSettChanging(object sender, SettingChangingEventArgs e) {
    if (e.SettingName.Equals("ZoomFactor")) {
        if ((float) e.NewValue > 3.0)
            e.Cancel = true;
        else
            MessageBox.Show("Der neue Zoom-Faktor ist " + e.NewValue);
    }
}

```

Größere Werte der **RichTextBox**-Eigenschaft **ZoomFactor** könnten weiter mit Gültigkeit für die aktuelle Sitzung per Optionsdialog gewählt, aber nicht mehr in die Datei **user.config** gespeichert werden.

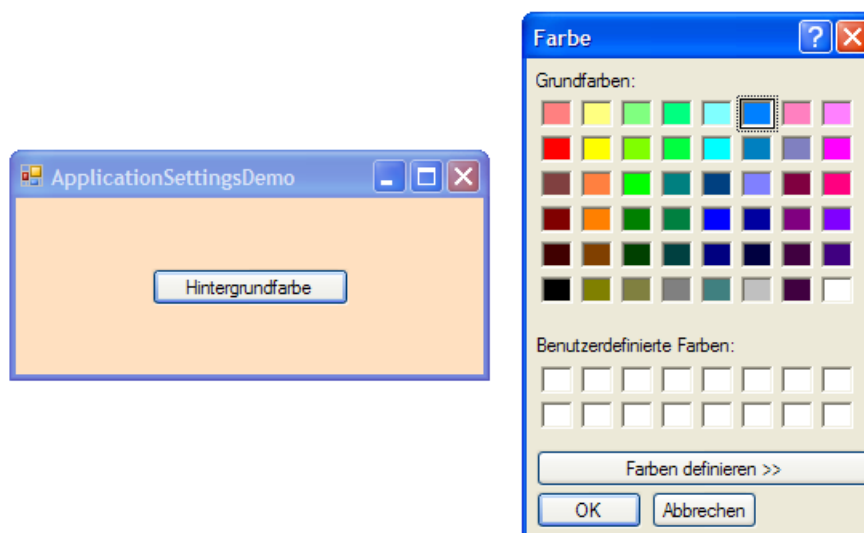
Bei aktiver Datenbindung zwischen der **RichTextBox**-Eigenschaft **ZoomFactor** und der Einstellung (Einstellungsklassen-Eigenschaft) **ZoomFactor** (vgl. Abschnitt 19.1.5) demonstriert die **SettingChanging**-Behandlungsmethode durch das Anzeigen einer MessageBox, wann ein Data Binding - Wertabgleich stattfindet. In unserem Beispiel erhält die **ZoomFactor**-Einstellung ihren neuen Wert erst beim Schließen des Anwendungsfensters (siehe Abschnitt 19.1.4), für eine Validierung mit Benutzeraufklärung bzw. Rückfrage also etwas spät.

19.1.7 Unterstützung durch die Entwicklungsumgebungen

Microsofts Entwicklungsumgebungen unterstützen die Einstellungsverwaltung, indem sie ...

- ein Formular zur bequemen Definition von Einstellungen bieten,
- die gem. Abschnitt 19.1.3 erforderliche **ApplicationSettingsBase**-Ableitung automatisch erstellen,
- das Binden von Einstellungen an die Eigenschaften von Komponenten per Eigenschaftsfenster erleichtern.

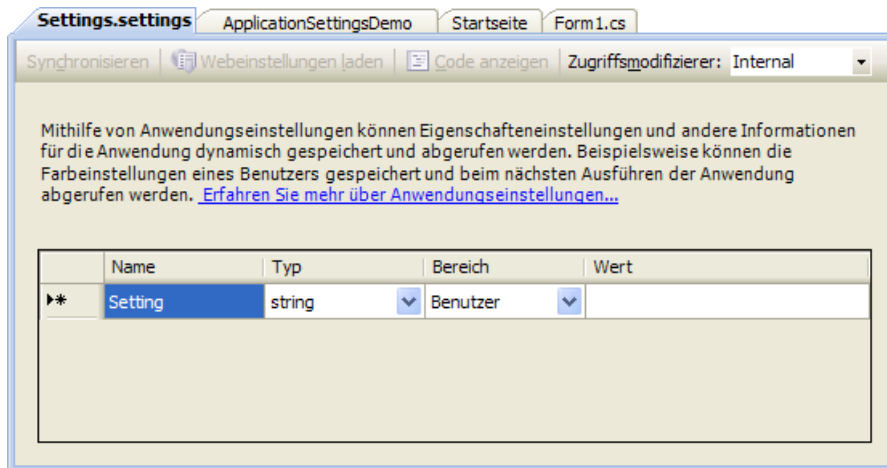
Um die Einstellungsunterstützung der Entwicklungsumgebung kennen zu lernen, erstellen wir ein einfaches WinForms-Projekt mit folgender Bedienoberfläche:



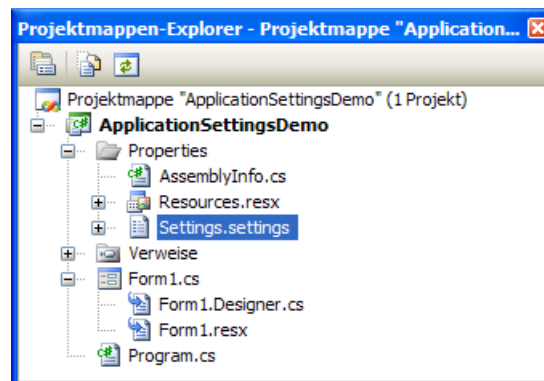
Die Benutzer können per **ColorDialog**-Objekt (siehe Abschnitt 18.3.2) die Hintergrundfarbe des Formulars wählen:


```
private void button1_Click(object sender, EventArgs e) {
    ColorDialog colorDialog = new ColorDialog();
    if (colorDialog.ShowDialog() == DialogResult.OK) {
        this.BackColor = colorDialog.Color;
    }
}
```

Wir legen eine benutzerbezogene Einstellung zur Speicherung der Hintergrundfarbe und eine anwendungsbezogene Einstellung für den Titel des Programms an. Öffnen Sie das Einstellungsformular



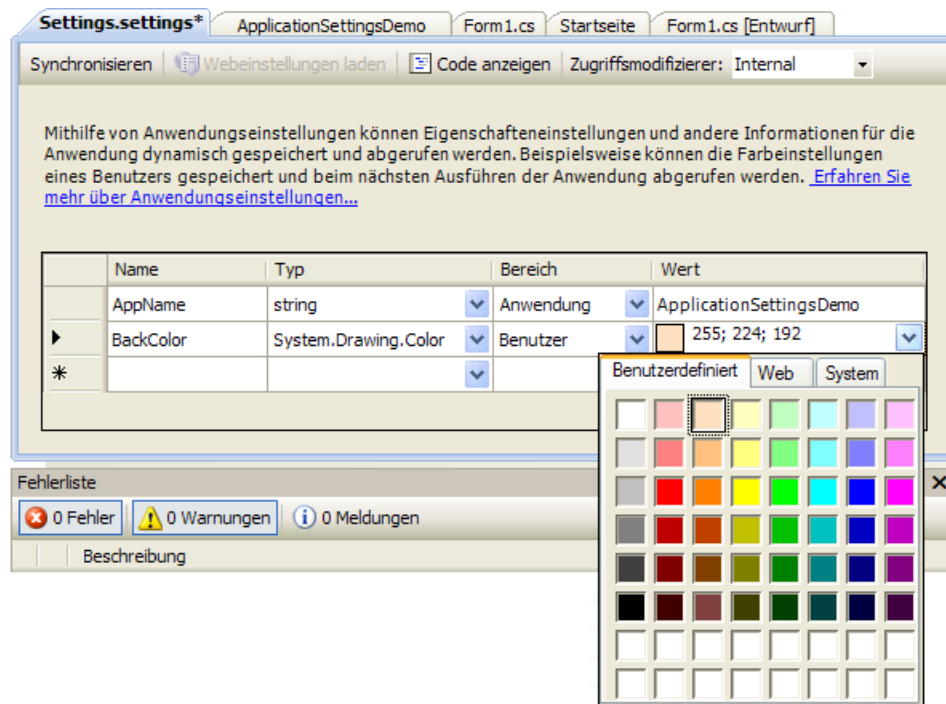
per Doppelklick auf den Eintrag **Properties > Settings.settings** im Projektmappen-Explorer:



Alternativ ist das Formular auch über den folgenden Menübefehl zu erreichen:

Projekt > Eigenschaften > Einstellungen

Wir legen nun für die geplanten Einstellungen Namen, Datentyp, Gültigkeitsbereich und Wert fest:



Die Entwicklungsumgebung legt aufgrund unserer Einstellungsdefinitionen im Projektordner die Datei **app.config** mit folgendem Inhalt hat:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <sectionGroup name="userSettings" type="System.Configuration.UserSettingsGroup, System,
      Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" >
      <section name="ApplicationSettingsVS.Properties.Settings"
        type="System.Configuration.ClientSettingsSection, System, Version=2.0.0.0,
        Culture=neutral, PublicKeyToken=b77a5c561934e089"
        allowExeDefinition="MachineToLocalUser" requirePermission="false" />
    </sectionGroup>
    <sectionGroup name="applicationSettings"
      type="System.Configuration.ApplicationSettingsGroup, System,
      Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" >
      <section name="ApplicationSettingsVS.Properties.Settings"
        type="System.Configuration.ClientSettingsSection, System, Version=2.0.0.0,
        Culture=neutral, PublicKeyToken=b77a5c561934e089" requirePermission="false" />
    </sectionGroup>
  </configSections>
  <userSettings>
    <ApplicationSettingsVS.Properties.Settings>
      <setting name="BackColor" serializeAs="String">
        <value>255, 224, 192</value>
      </setting>
    </ApplicationSettingsVS.Properties.Settings>
  </userSettings>
  <applicationSettings>
    <ApplicationSettingsVS.Properties.Settings>
      <setting name="AppName" serializeAs="String">
        <value>ApplicationSettingsVS</value>
      </setting>
    </ApplicationSettingsVS.Properties.Settings>
  </applicationSettings>
</configuration>
```

Achtung: Bei jedem Erstellen des Projekts werden die Dateien **prog.exe.config** und **prog.vshost.exe.config** durch die Datei **app.config** überschrieben!

Außerdem legt unsere Entwicklungsumgebung zur Verwaltung der Einstellungen die von **ApplicationSettingsBase** abgeleitete Klasse **Settings** an, deren Quellcode in der Projektdatei **...\Properties\Settings.Designer.cs** landet:

```

namespace ApplicationSettingsVS.Properties {

    [global::System.Runtime.CompilerServices.CompilerGeneratedAttribute()]
    [global::System.CodeDom.Compiler.GeneratedCodeAttribute(
        "Microsoft.VisualStudio.Editors.SettingsDesigner.SettingsSingleFileGenerator",
        "9.0.0.0")]
    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.ApplicationScopedSettingAttribute()]
        [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
        [global::System.Configuration.DefaultSettingValueAttribute("ApplicationSettingsVS")]
        public string AppName {
            get {
                return ((string)(this["AppName"]));
            }
        }

        [global::System.Configuration.UserScopedSettingAttribute()]
        [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
        [global::System.Configuration.DefaultSettingValueAttribute("255, 224, 192")]
        public global::System.Drawing.Color BackColor {
            get {
                return ((global::System.Drawing.Color)(this["BackColor"]));
            }
            set {
                this["BackColor"] = value;
            }
        }
    }
}

```

Im Vergleich zu Abschnitt 19.1.3 ist anzumerken:

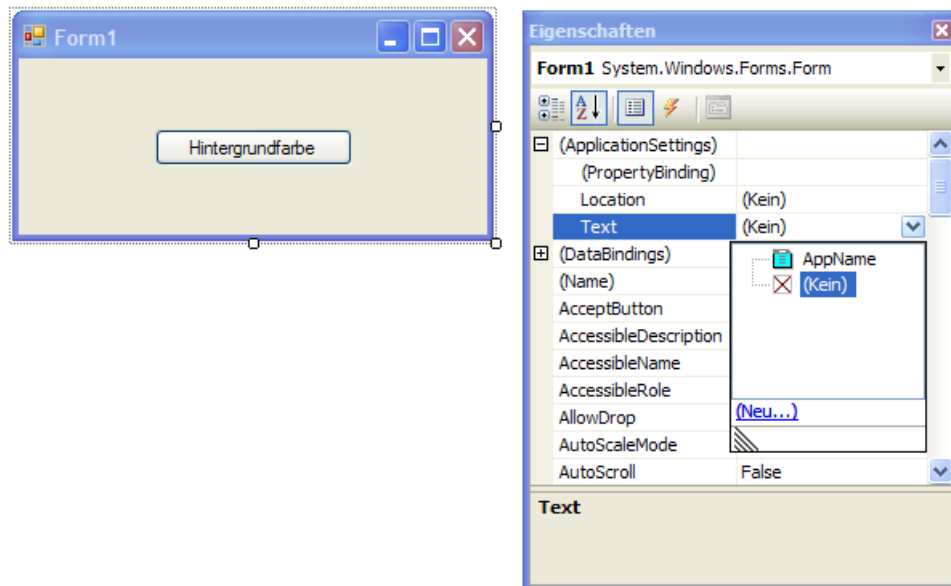
- Für die Klasse **Settings** wird ein eigener Unternamensraum definiert.
- Alle Klassennamen werden voll qualifiziert geschrieben, wobei das Präfix

global::

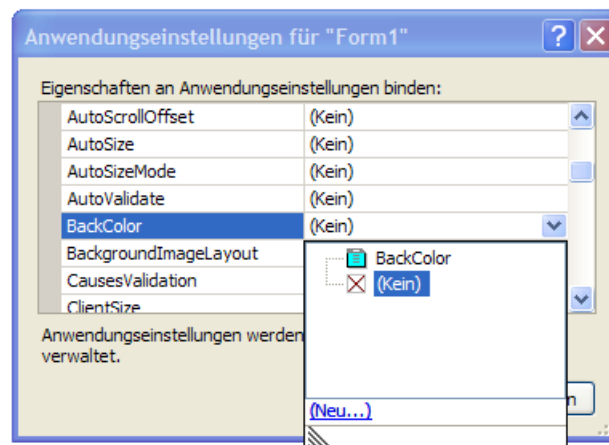
eine im globalen Namensraum beginnende Namensauflösung vorschreibt. So wird z.B. verhindert, dass eine im lokalen Projekt vorhandene Definition für den Bezeichner **System** die FCL-Namensraumbezeichnung **System** verdeckt.

- Über die statische Eigenschaft **Default** wird ein mit Hilfe der **ApplicationSettingsBase**-Methode **Synchronized()** erstelltes Objekt einer **Thread**-sicheren Hüllklasse zu **Settings** angeboten. Bei Verwendung des **Default**-Objekts werden in Multithreading-Anwendungen Inkonsistenzen bei der Einstellungsverwaltung verhindert.

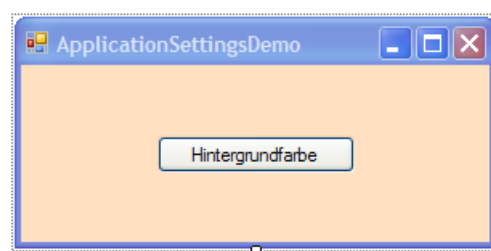
Damit die Einstellungen wirksam werden, müssen Sie den betroffenen Formulareigenschaften zugewiesen werden, was bequem im Eigenschaftsfenster über den Abschnitt **ApplicationSettings** geschehen kann. Für die **Text**-Eigenschaft ist schon ein Eintrag vorhanden, so dass wir per Drop-Down - Liste die Einstellung **AppName** zuordnen können:



Über den Schalter  in der markierten Zeile (**PropertyBinding**) binden wir mit Hilfe des folgenden Dialogs die **BackColor**-Eigenschaft des Formulars an die zugehörige Einstellung:



Der Formulardesigner zeigt nun die anwendungsbezogene **Text**-Einstellung und den Voreinstellungswert für die benutzerbezogene **BackColor**-Einstellung:



In der vom Formulkonstruktor aufgerufenen Methode **InitializeComponent()**, die von der Entwicklungsumgebung gepflegt wird, erhalten die Formulareigenschaften **BackColor** und **Text** den zugehörigen Einstellungswert:

```
private void InitializeComponent() {
    this.BackColor=global::ApplicationSettingsVS.Properties.Settings.Default.BackColor;
    this.DataBindings.Add(new System.Windows.Forms.Binding("BackColor",
        global::ApplicationSettingsVS.Properties.Settings.Default, "BackColor", true,
        System.Windows.Forms.DataSourceUpdateMode.OnPropertyChanged));
    this.Text = global::ApplicationSettingsVS.Properties.Settings.Default.AppName;
}
```

Weil die Eigenschaft **BackColor** anschließend an die zugehörige Benutzereinstellung gebunden wird, ist die vorherige Wertzuweisung wohl überflüssig.

Den zum Speichern der benutzerbezogenen Einstellungen erforderlichen Aufruf der vom **Settings**-Objekt auszuführenden **Save()**-Methode müssen wir selbst vornehmen. Man kann ihn z.B. in der Behandlungsmethode zum **Form**-Ereignis **FormClosing** unterbringen:

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e) {
    global::ApplicationSettingsVS.Properties.Settings.Default.Save();
}
```

Damit sind die beiden Einstellungen vollständig integriert.

Um die bisher fehlende Konfigurationsunterstützung für ein Projekt nachzurüsten, öffnet man sein Kontextmenü im **Projektmappen-Explorer** und wählt den Menübefehl

Hinzufügen > Neues Element > Einstellungsdatei

Anschließend steht über einem Doppelklick auf den neuen Projekteintrag **Settings1.settings** das Einstellungsformular zur Verfügung, und die Entwicklungsumgebung kümmert sich um

- die XML-Konfigurationsdateien (**app.config**, **prog.exe.config**)
- die **ApplicationSettingsBase**-Ableitung

19.1.8 Weitere Optionen der .NET - Konfigurationsdateien

In der Konfigurationsdatei **prog.exe.config** ist auch das aus .NET 1.1 stammende XML-Element **<appSettings>** erlaubt, das einfach aufgebaut ist und Schreibzugriffe auf *anwendungsbezogene* Einstellungen durch das laufende Programm erlaubt, z.B.:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key = "AppName" value = "Ritchis Texteditor" />
  </appSettings>
</configuration>
```

Diese Option ist aber in der Regel nutzlos, weil die erforderlichen Dateizugriffsrechte des Benutzers fehlen (sollten).

Sind viele anwendungsbezogene Konfigurationsoptionen in **prog.exe.config** vorhanden, kommt das Erstellen eines speziellen Konfigurations-Hilfsprogramms in Frage, um die Administration des Primärprogramms bequem und sicher zu machen. Das Hilfsprogramm kann die Datei **prog.exe.config** des Primärprogramms bei ausreichenden Benutzerrechten öffnen und verändern. Dabei sind die FCL-Klassen **Configuration**, **ExeConfigurationFileMap** und **ConfigurationManager** aus dem aus dem Namensraum **System.Configuration** von Nutzen.

19.2 Windows-Registrierungsdatenbank

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\Ritchis Texteditor\<Version>

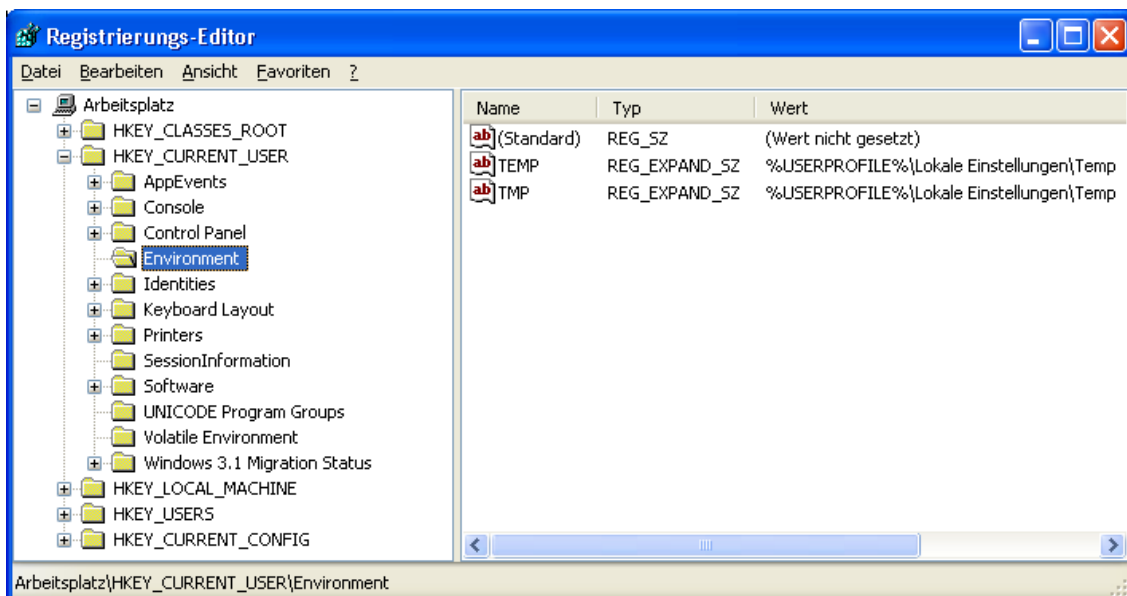
Zur Verwaltung der Einstellungen von .NET - Programmen sind eindeutig die im Abschnitt 19.2 beschriebenen Konfigurationsdateien zu bevorzugen. Das Beherrschen von Registry-Zugriffen ist aber auch für .NET - Anwendungen relevant. Indem die Einstellungsverwaltung unseres Editors

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, DWORD⁷⁰), welche die eigentlichen Daten enthalten. Im Editor-Beispiel werden wir folgende Werte einsetzen:

Name	Typ	mögliche Daten
WordWrap	REG_DWORD	0 Zeilenumbruch ist aus 1 Zeilenumbruch ist an
ScrollBars	REG_SZ ⁷¹ (Zeichenfolge)	None Keine Rollbalken Vertical Nur ein vertikaler Rollbalken Horizontal Nur ein horizontaler Rollbalken Both Beide Rollbalken
ZoomFactor	REG_DWORD	64 - 640
BackColor	REG_DWORD	32-Bit-Ganzzahl mit der Hintergrundfarbe. Die 4 Bytes enthalten die Werte der Farbkanäle (ARGB).

Zur Inspektion und Modifikation der Registry kann das Windows-Administrationswerkzeug **Regedit.exe** verwendet werden:



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_MACHINE** (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 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_MACHINE** werden administrative Benutzerrechte benötigt.

⁷⁰ Vorzeichenfreie ganze Zahl mit 32 Bit Speicherplatz

⁷¹ Die Bezeichnung SZ geht auf die Behandlung von Zeichenfolgen als Array von Char-Variablen in der Programmiersprache C zurück. Weil ein Zeichen mit dem Wert Null (Zero) das Ende einer solchen Zeichenfolge signalisiert, spricht man von einem *zero terminated string*.

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 die Daten in den Wert mit dem angegebenen Namen
- **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_MACHINE	LocalMaschine
HKEY_CURRENT_USER	CurrentUser

In unserem Editor-Projekt wird die neue Funktionalität durch Erweiterungen der Quellcodedatei **RitchisTexteditor.cs** realisiert. Zunächst wird der Namensraum **Microsoft.Win32** importiert:

```
using Microsoft.Win32;
```

Weil dieser Namensraum im stets bekannten Bibliotheks-Assembly **microsoft.win32.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("Ritchis Texteditor")]
[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 **RitchisTexteditor**,

```
string strUserKey, strWordWrap, strScrollBars, strZoomFactor, strBackColor;
```

die im Formularkonstruktor mit Werten versorgt werden:


```

strUserKey = "Software\\" +
             Application.CompanyName + "\\\" +
             Application.ProductName + "\\\" +
             Application.ProductVersion;
strWordWrap = "WordWrap";
strScrollBars = "ScrollBars";
strZoomFactor = "ZoomFactor";
strBackColor = "BackColor";

```

Zum Sichern von Einstellungen in die Registry eignet sich eine Behandlungsmethode zum Formularereignis **FormClosing**. Unser Editor enthält bereits eine solche Methode, die sich bisher auf die Vermeidung von Textverlusten beschränkt hat. Nun befördern wir zusätzlich Einstellungen in Registry-Werte:

```

void FormOnClosing(object sender, CancelEventArgs cea) {
    if (UnsavedText())
        cea.Cancel = true;
    else {
        RegistryKey userKey = null;
        try {
            userKey = Registry.CurrentUser.OpenSubKey(strUserKey, true);
            if (userKey == null)
                userKey = Registry.CurrentUser.CreateSubKey(strUserKey);
            userKey.SetValue(strWordWrap, (editor.WordWrap == true ? 1 : 0));
            switch (editor.ScrollBars) {
                case RichTextBoxScrollBars.None:
                    userKey.SetValue(strScrollBars, "None"); break;
                case RichTextBoxScrollBars.Vertical:
                    userKey.SetValue(strScrollBars, "Vertical"); break;
                case RichTextBoxScrollBars.Horizontal:
                    userKey.SetValue(strScrollBars, "Horizontal"); break;
                case RichTextBoxScrollBars.Both:
                    userKey.SetValue(strScrollBars, "Both"); break;
            }
            userKey.SetValue(strZoomFactor, (int)(editor.ZoomFactor * 100 + 0.5));
            userKey.SetValue(strBackColor, editor.BackColor.ToArgb());
        } catch {
            MessageBox.Show("Fehler beim Speichern der Einstellungen", appName,
                            MessageBoxButtons.OK, MessageBoxIcon.Error);
        } finally {
            if (userKey != null)
                userKey.Close();
        }
        Application.Exit();
    }
}

```

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**. Im Beispiel wird die **RichTextBox**-Eigenschaft **ZoomFactor**, welche den Datentyp **float** besitzt und Werte von 1/64 bis 64,0 annimmt, in der Registry durch einen **DWORD**-Wert gespeichert.

Ein **Regedit**-Bildschirmphoto mit dem von unserem Editor angelegten Registry-Key war schon am Anfang von Kapitel 19 zu sehen.

Zum Lesen der Einstellungen eignet sich eine Behandlungsmethode zum **Form-Ereignis Load**:

```
void FormOnLoad(object sender, EventArgs ea) {
    RegistryKey userKey = Registry.CurrentUser.OpenSubKey(strUserKey);
    if (userKey != null) {
        try {
            int ww = (int)userKey.GetValue(strWordWrap);
            switch (ww) {
                case 0: editor.WordWrap = false; break;
                case 1: editor.WordWrap = true; break;
            }
            string strSB = (string)userKey.GetValue(strScrollBars);
            if (strSB != null)
                switch (strSB) {
                    case "None":
                        editor.ScrollBars = RichTextBoxScrollBars.None; break;
                    case "Vertical":
                        editor.ScrollBars = RichTextBoxScrollBars.Vertical; break;
                    case "Horizontal":
                        editor.ScrollBars = RichTextBoxScrollBars.Horizontal; break;
                    case "Both":
                        editor.ScrollBars = RichTextBoxScrollBars.Both; break;
                }
            int zf = (int)userKey.GetValue(strZoomFactor);
            if (zf >= 64 && zf <= 640)
                editor.ZoomFactor = ((float)zf) / 100;
            int color = (int)userKey.GetValue(strBackColor);
            editor.BackColor = Color.FromArgb(color);
        } catch {
            MessageBox.Show("Fehler beim Laden der Einstellungen", appName,
                MessageBoxButtons.OK, MessageBoxIcon.Error);
        }
        userKey.Close();
    }
}
```

Weil **GetValue()** den Rückgabety **object** besitzt, ist eine explizite Typwandlung erforderlich.

Den vollständigen Quellcode zu der eben vorgestellten Editorvariante mit Einstellungsverwaltung per Registrierungsdatenbank finden Sie im Ordner

...\BspUeb\Einstellungen\Registry\NurOptionen

19.3 Übungsaufgaben zu Kapitel 19

1) Erweitern Sie unseren Editor ausgehend vom Entwicklungsstand in

...\BspUeb\Einstellungen\XML\ApplicationSettings

so, dass auch die Position und Klientenbereichsgröße des Anwendungsfensters beim Beenden in der benutzerspezifischen Einstellungsdatei **user.config** gesichert und beim Starten von dort gelesen werden.

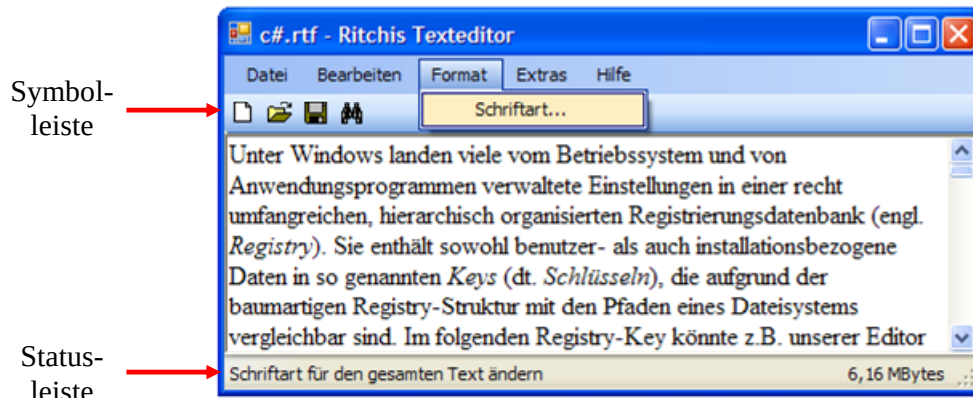
2) Erweitern Sie die Editorvariante mit Registry - basierter Einstellungsverwaltung ausgehend vom Entwicklungsstand in

...\BspUeb\Einstellungen\Registry\NurOptionen

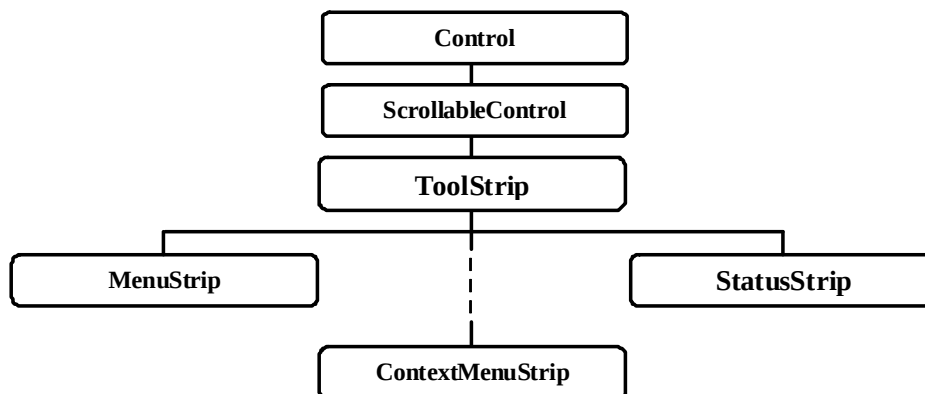
so, dass auch die Position und Klientenbereichsgröße des Anwendungsfensters beim Beenden in der Registry gesichert und beim Starten von dort gelesen werden.

20 Symbol- und Statusleiste

Nachdem wir Menüs und Dialogboxen kennen gelernt haben, sind Symbol- und Statusleisten auf dem Weg zur zeitgemäßen Bedienoberfläche eines Windows-Programms die nächsten Schritte:



Symbol- bzw. Statusleiste werden bei WinForms - Anwendungen über die Steuerelementklassen **ToolStrip** bzw. **StatusStrip** aus dem Namensraum **System.Windows.Forms** realisiert. Wie der folgende Stammbaum



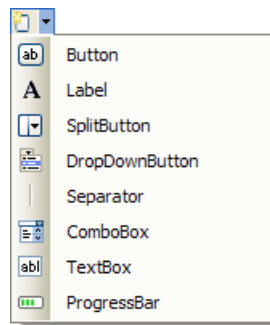
zeigt, werden ab .NET 2.0 Statusleisten (Klasse **StatusStrip**) und Menüs (Klassen **MenuStrip** bzw. **ContextMenuStrip**) als spezielle Symbolleisten behandelt.

In diesem Manuskript wird nur die aktuelle, auf der Klasse **ToolStrip** basierende WinForms-Lösung für Symbol- bzw. Statusleisten vorgestellt. Eine ausführliche Darstellung der .NET 1 - Lösung, die auf den Klassen **ToolBar** bzw. **StatusBar** basiert, finden Sie in Baltes-Götz (2007, Kap. 19).

20.1 Symbolleisten

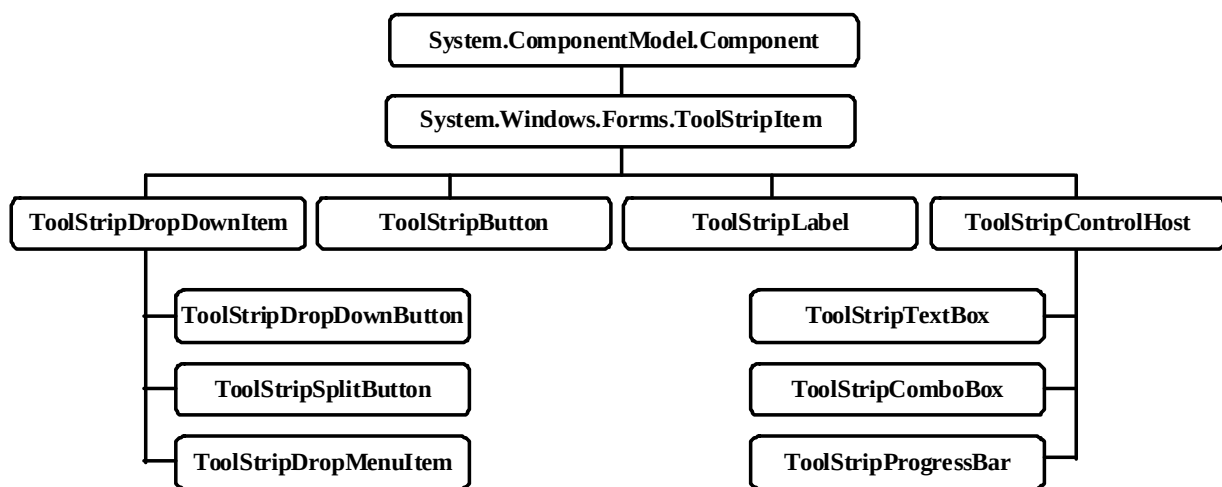
Besonders häufig benötigte Funktionen bieten Windows-Programme meist (auch) über Symbolleisten an. Dort kommen meist Symbolschalter (Klasse **ToolStripButton**) zum Einsatz. Ab der .NET - Version 2.0 sind auf einer WinForms-Symbolleiste aber auch folgende Steuerelemente möglich:

- Objekte der Klasse **ToolStripLabel** dienen zur Anzeige von Texten.
- Man kann ein per Schalter zu öffnendes DropDown-Menü anbieten (Klasse **ToolStripDropDownButton**). Häufiger kommt jedoch eine Kombination aus Befehlsschalter und Drop-Down-Menü zum Einsatz (Klasse **ToolStripSplitButton**), z.B.:



- Wie bei Menüs sind Trennlinien verfügbar (Klasse **ToolStripSeparator**).
- Texteingabefelder (Klasse **ToolStripTextBox**) und Kombinationsfelder (Klasse **ToolStripComboBox**)
- Fortschrittsbalken (Klasse **ToolStripProgressBar**)

Der folgenden Abbildung sind die Abstammungsverhältnisse zu entnehmen:



20.1.1 Do-It-Yourself - Erstellung einer Symbolleiste

Wie üblich verschaffen wir uns zunächst die früher oder später benötigten Sachkenntnisse über die beteiligten Klassen, Objekte und Methoden, um später auch die von den Entwicklungsumgebungen angebotenen Techniken und Assistenten zur Reduktion von Routine-Arbeiten kennen zu lernen.

In unserem Editorprojekt soll eine Standard-Symbolleiste



mit wichtigen Funktionen aus den Untermenüs **Datei** und **Bearbeiten** entstehen. Das Erstellen und Konfigurieren des benötigten **ToolStrip**-Objekts erledigen wir im Formularkonstruktor:

```

ToolStrip toolStripStandard = new ToolStrip();
toolStripStandard.Dock = DockStyle.Top;
mainMenu.Dock = DockStyle.Top;
Controls.Add(toolStripStandard);
Controls.Add(mainMenu);

```

Indem wir das Hauptmenü-Objekt `mainMenu` nach dem Symbolleisten-Objekt in das Formular aufnehmen, verdrängt es den **Dock**-Konkurrenten und liegt wie gewohnt unmittelbar unter der Titelzeile des Fensters (vgl. Abschnitt 9.4.4.3). Später werden wir auch verschiebbare und wahlfrei andockbare Symbolleisten realisieren. Weil die Symbolleisten-Verschiebefläche



momentan noch nutzlos ist, schalten wir sie ab. Dazu erhält die **ToolStrip**-Eigenschaft **GripStyle** den Wert **ToolStripGripStyle.Hidden**:

```
toolStripStandard.GripStyle = ToolStripGripStyle.Hidden;
```

Wir bestücken die Symbolleiste mit vier Symbolleistenschaltflächen, die Referenzvariablen auf Klassenebene erhalten, damit sie in allen Methoden angesprochen werden können (, was vorläufig nur bei der Schaltfläche **tsbFind** erforderlich ist):

```
ToolStripButton tsbNew, tsbOpen, tsbSave, tsbFind;
```

Im Fensterkonstruktor unseres Editors werden folgende Zeilen ergänzt, welche die Symbolleisten-schalter erstellen

```
tsbNew = new ToolStripButton();
tsbOpen = new ToolStripButton();
tsbSave = new ToolStripButton();
tsbFind = new ToolStripButton();
```

und in die Standard-Symbolleiste aufnehmen:

```
toolStripStandard.Items.AddRange(
    new ToolStripItem[] {tsbNew, tsbOpen, tsbSave, tsbFind});
```

Wie Sie bereits aus dem Kapitel 17 über Menüs wissen, zeigt die **ToolStrip**-Eigenschaft **Items** auf eine Liste mit **ToolStripItem**-Objekten. Diese Liste (ein Objekt aus der Klasse **ToolStripItemCollection**) verfügt u.a. über die Methoden **Add()** und **AddRange()** zur Aufnahme von Items.

Sein Symbol wird einem Schalter über die **Image**-Eigenschaft zugeordnet, wobei meist Rastergrafiken mit einer (16 × 16)-Pixelmatrix zum Einsatz kommen. Im Editorprojekt können wir die Symbole zu den Menüitems verwenden, z.B.:

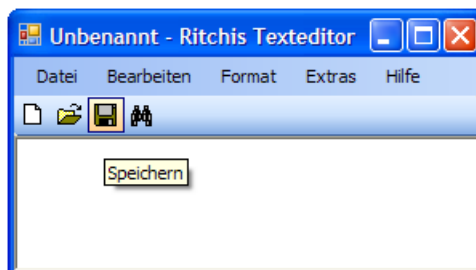
```
Bitmap bmp = new Bitmap("new.bmp");
bmp.MakeTransparent(bmp.GetPixel(0, 0));

tsbNew.Image = bmp;
```

Auch bei der **Click**-Ereignisbehandlung zu den Symbolschaltern können wir uns auf das Registrieren bereits vorhandener Methoden zu korrespondierenden Menüitems beschränken, z.B.:

```
tsbNew.Click += new System.EventHandler(MitNewOnClick);
```

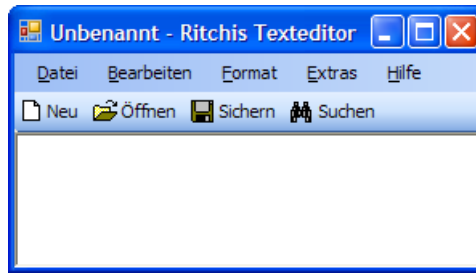
Der Klarheit halber wird man die Piktogramme in der Regel durch Quick-Info - (Tool-Tip -) Texte ergänzen. Weil Disketten mittlerweile veraltet, vielen Benutzern nur noch aus dem Museum bekannt und außerdem zum Speichern kaum geeignet sind, ist z.B. beim traditionellen Symbol für das Speichern ein erläuternder Quick-Info - Text erforderlich:



Dazu ist zu jedem **ToolStripButton**-Objekt ein **ToolTipText**-Eigenschaftswert festzulegen, z.B.:

```
tsbSave.ToolTipText = "Speichern";
```

Alternativ oder zusätzlich kann man Texte *auf* den Symbolleistenschaltflächen unterbringen, z.B.:



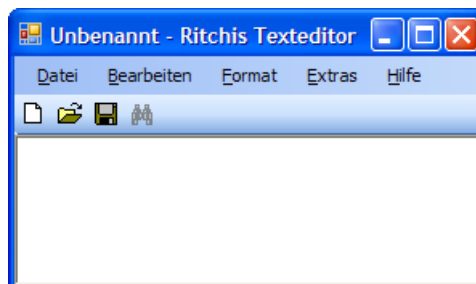
Dazu ist zu jedem **ToolStripButton**-Objekt ein **Text**-Eigenschaftswert festzulegen, z.B.:

```
tsbSave.Text = "Speichern";
```

ToolStripButton-Objekte besitzen wie **ToolStripMenuItem**-Objekte eine **Enabled**-Eigenschaft, um ihre Dienstbereitschaft festzustellen oder festzulegen. Allerdings existiert kein Analogon zum **DropDownOpening**-Ereignis der Klasse **ToolStripMenuItem**. Schließlich ist eine Symbolleiste in der Regel permanent sichtbar. In unserem Editor-Projekt sollte das **ToolStripButton**-Objekt **tsbFind** nur dann aktiv sein, wenn Text vorhanden ist. Dies wird über das **RichTextBox**-Ereignis **TextChanged** realisiert:

```
protected void RichTextBoxOnTextChanged(object sender, EventArgs ea) {
    double d = editor.TextLength;
    if (editor.TextLength > 0)
        tsbFind.Enabled = true;
    else
        tsbFind.Enabled = false;
}
```

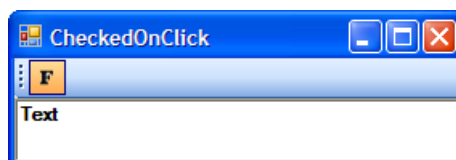
Zusätzlich setzen wir die **tsbFind**-Eigenschaft **Enabled** initial auf den Wert **false**, so dass der Symbolleistenschalter generell bei fehlendem Text deaktiviert ist:



Den vollständigen Quellcode des Editors im bisher vorgestellten Entwicklungszustand finden Sie im Ordner

...\\BspUeb\\WinForms\\Steuerelemente\\Symbolleisten\\Einfach

Gelegentlich soll ein Symbolschalter bei einem Mausklick analog zu einem Kontrollkästchen zwischen dem neutralen und dem eingerasteten Zustand wechseln. Mit dem **F**-Schalter im folgenden Programm wird die Fett-Auszeichnung für den gesamten Text der **RichTextBox**-Komponente ein- bzw. ausgeschaltet:



Das **ToolStripButton**-Objekt

```
ToolStripButton tsbBold;
```

erhält neben weiteren Konfigurationen den Wert **true** für die Eigenschaft **CheckedOnClick**:

```
tsbBold = new ToolStripButton();
tsbNew.ToolTipText = "Fett";
Bitmap bmp = new Bitmap("bold.bmp");
bmp.MakeTransparent(bmp.GetPixel(0, 0));
tsbBold.Image = bmp;
tsbBold.CheckOnClick = true;
tsbBold.CheckStateChanged += new
    System.EventHandler(TsbBoldOnCheckStateChanged);
```

Bei einer Zustandsänderung des Symbolschalters `tsbBold` (Ereignis **CheckStateChanged**) wird durch die zuständige Ereignismethode über bitorientierte Operatoren in der **RichTextBox**-Schriftart die Fett-Auszeichnung ein- bzw. ausgeschaltet:

```
protected void TsbBoldOnCheckStateChanged(object sender, EventArgs e) {
    if (tsbBold.Checked)
        editor.Font = new Font(editor.Font, editor.Font.Style | FontStyle.Bold);
    else
        editor.Font = new Font(editor.Font, editor.Font.Style & ~FontStyle.Bold);
}
```

Mit etwas erhöhtem Aufwand lässt sich über die **RichTextBox**-Eigenschaft **SelectionFont** übrigens auch das von Editoren gewohnte Verhalten eines Fett-Umschalters realisieren.

20.1.2 Bewegliche Leisten und ToolStripContainer

Bewegliche **ToolStrip**-Komponenten (z.B. Symbolleisten) realisiert man mit Hilfe eines Steuerelement aus der Klasse **ToolStripContainer**. Es enthält im Zentrum ein Objekt der Klasse **ContentPanel** für den Inhalts- bzw. Arbeitsbereich des Formulars, ansprechbar über die gleichnamige **ToolStripContainer**-Eigenschaft. Hier findet die **RichTextBox**-Komponente unseres Editors ihr neues Zuhause:

```
ToolStripContainer tsc = new ToolStripContainer();
tsc.ContentPanel.Controls.Add(editor);
```

An den vier Seiten befinden sich **ToolStripPanel**-Objekte, ansprechbar über die Eigenschaften **TopToolStripPanel**, **LeftToolStripPanel**, **BottomToolStripPanel** und **RightToolStripPanel**. Wir platzieren die Symbolleiste und das Hauptmenü unseres Editors in dieser Reihenfolge in den **TopToolStripPanel**-Container:

```
tsc.TopToolStripPanel.Controls.Add(toolStripStandard);
tsc.TopToolStripPanel.Controls.Add(mainMenu);
```

Dabei ist die **Dock**-Eigenschaft der Steuerelemente irrelevant. Wir streichen die folgende Zeile,

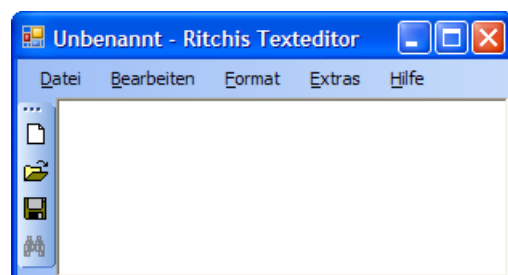
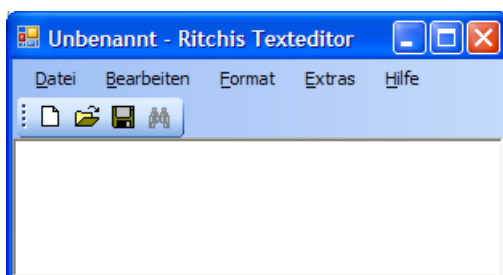
```
toolStripStandard.GripStyle = ToolStripGripStyle.Hidden;
```

weil die Symbolleisten-Verschiebefläche nun benötigt wird.

Der **ToolStripContainer** wird mit **DockStyle.Fill** auf das Formular platziert:

```
tsc.Dock = DockStyle.Fill;
Controls.Add(tsc);
```

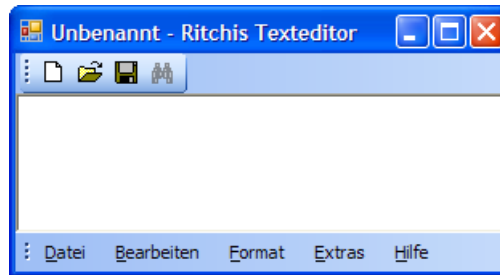
Nun kann der Benutzer die Symbolleiste an allen vier Seiten des Formulars andocken, z.B.:



Macht man die Verschiebefläche zum Hauptmenü sichtbar,

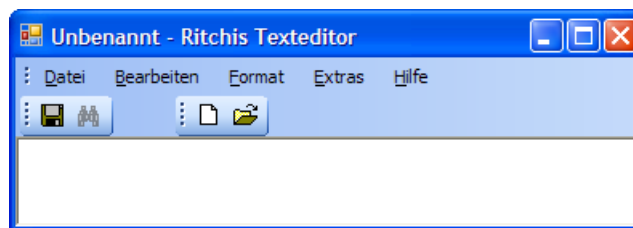
```
mainMenu.GripStyle = ToolStripGripStyle.Visible;
```

kann es ebenfalls an eine beliebigen Seite andockt werden, z.B.:



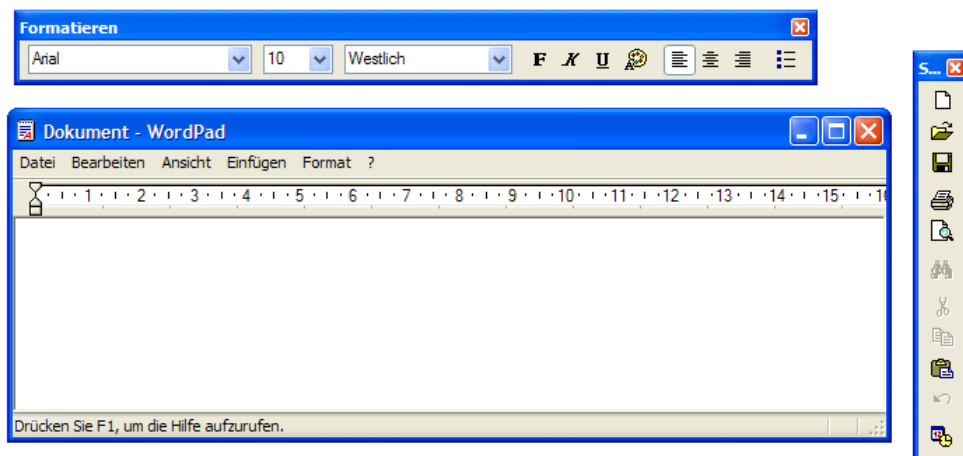
Der **ToolStripGripStyle.Visible** nicht mit dem **ToolStripLayoutStyle.Flow** kombinierbar, der für einen „Zeilenumbruch“ bei horizontalem Platzmangel sorgt.

Werden mehrere Symbolleisten erstellt und in das nördliche **ToolStripPanel**-Objekt aufgenommen, können diese vom Benutzer flexibel platziert werden, z.B.:



Man könnte die Leistenpositionen beim Verlassen des Programms in der Konfigurationsdatei **user.config** speichern (siehe Abschnitt 19).

Frei schwebende Leisten, wie sie hier bei der Windows-XP - Zugabe WordPad zu sehen sind,



lassen sich mit der Klasse **ToolStripContainer** *nicht* realisieren.

Den vollständigen Quellcode des Editors im bisher vorgestellten Entwicklungszustand finden Sie im Ordner

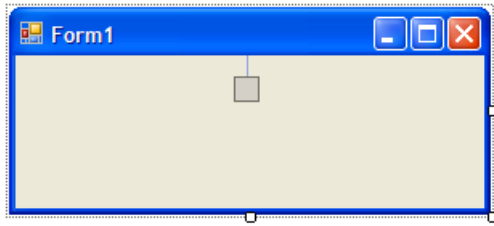
...\\BspUeb\\WinForms\\Steuerelemente\\Symbolleisten\\Beweglich

20.1.3 Unterstützung durch die Entwicklungsumgebungen

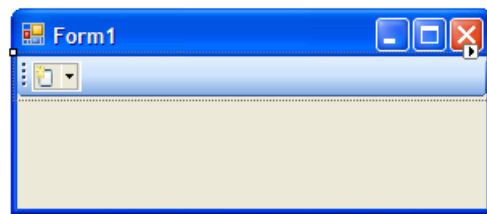
Um die exzellente Symbolleistenunterstützung von Microsofts Entwicklungsumgebungen (Visual C# Express Edition bzw. Visual Studio 2008) kennen zu lernen, starten wir ein WinForms-Projekt und erstellen die Bedienoberfläche eines einfachen Editors im RAD-Modus (*Rapid Application Development*).

Befördern Sie im Formulardesigner aus der **Toolbox** ein **ToolStrip** - Objekt auf das Formular:

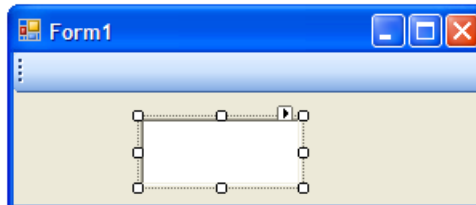
Platzierungshinweis



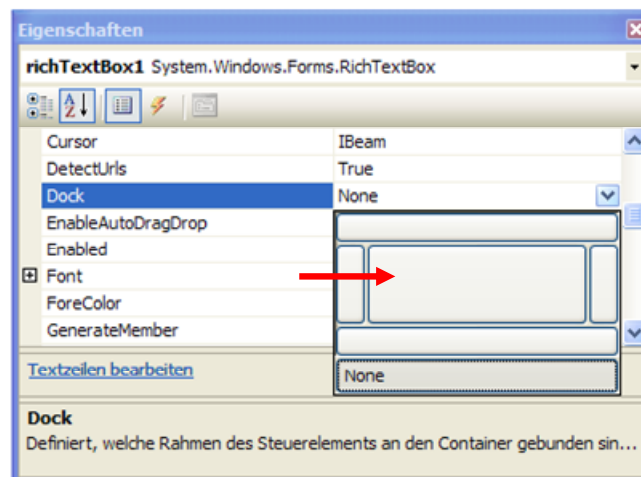
Ergebnis



Ergänzen Sie eine **RichTextBox** - Komponente,



und wählen Sie im Eigenschaftsfenster zur **Dock**-Eigenschaft der Editorkomponente den Wert **DockStyle.Fill**:



Der entstehende Editor verhält sich wunschgemäß:

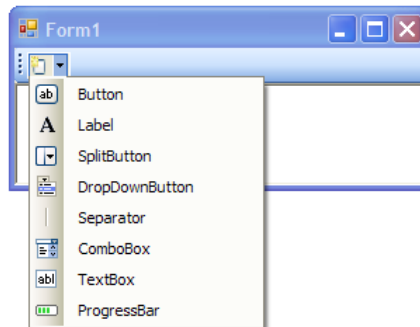


Eine Inspektion des automatisch vom Formulardesigner in der Datei **Form1.Designer.cs** erstellten Quellcodes zeigt, dass die **RichTextBox**- und die **ToolStrip**-Komponente in der gemäß Abschnitt 9.4.4.3 korrekten Reihenfolge als Top-Dock - Konkurrenten in die Steuerelementliste des Formulars aufgenommen worden sind:

```
this.Controls.Add(this.richTextBox1);
this.Controls.Add(this.toolStrip1);
```

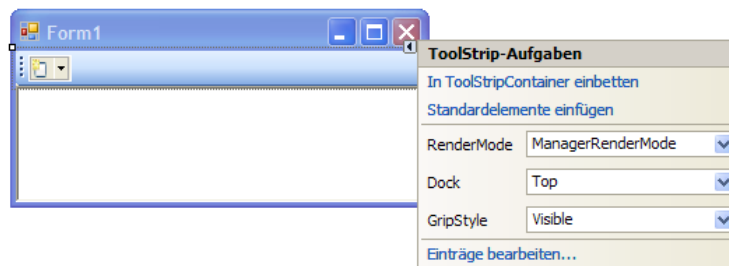
Wird die notwendige **Controls**-Reihenfolge von Dock-Konkurrenten beim Einfügen per Formulardesigner angewendet, überlagert die Symbolleiste (**DockStyle.Top**) den Editor (**DockStyle.Fill**), und die ersten Textzeilen bleiben hinter der Symbolleiste verborgen. Sachwissen und konzentriertes Arbeiten führen leider zur falschen Reihenfolge der **Controls**-Aufrufe. Man muss also entweder naiv das oberste Steuerelement zuerst einfügen oder wissen, dass im Designer in der „falschen“ Reihenfolge vorzugehen ist.

Der Formulardesigner unterstützt die Symbolleistenbestückung (ohne weitere Überraschungseffekte) durch ein Kombiwerkzeug vom Typ **ToolStripSplitButton**. Per Mausklick auf die linke Seite wird ein neues **ToolStripButton**-Objekt eingefügt. Über einen Mausklick auf die rechte Seite (mit Pfeil) öffnet man ein DropDown-Menü, das alle zulässigen Symbolleisten-Komponenten anbietet:

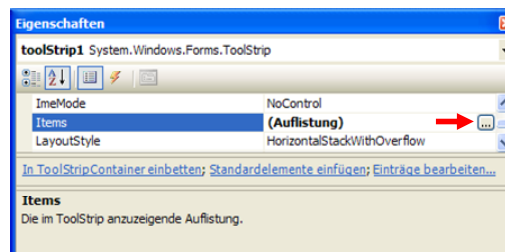


Eine eingefügte Komponente kann man per Eigenschaftsfenster bequem konfigurieren.

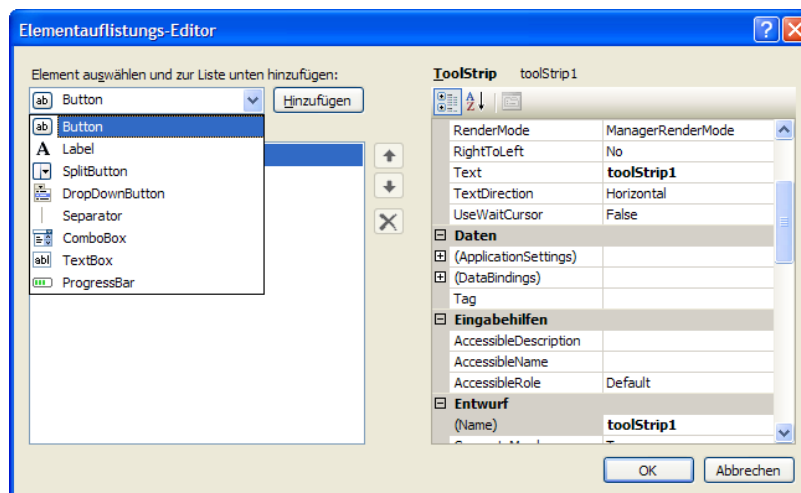
Bei markierter Symbolleiste ist über ein Pfeilsymbol in der rechten oberen Ecke ein Formular mit **ToolStrip-Aufgaben** zu erreichen, das z.B. die Einbettung der Leiste in einen **ToolStripContainer** (vgl. Abschnitt 20.1.2) anbietet:



Über den Link **Einträge bearbeiten** im Aufgabenformular oder den Erweiterungsschalter in der markierten **Items**-Zeile des Eigenschaftsfensters zur Symbolleiste

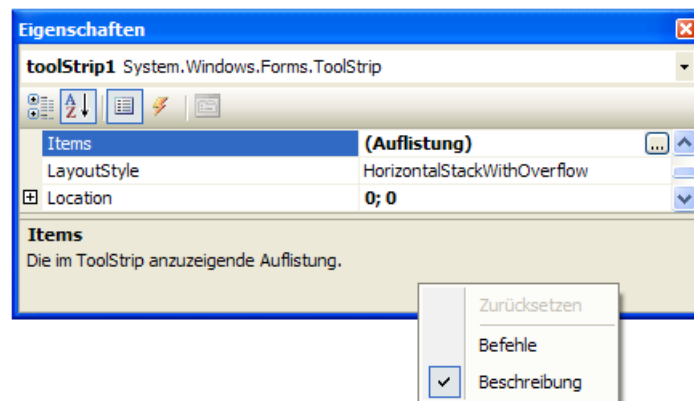


ist mit dem **Elementauflistungs-Editor** ein weiteres Hilfsmittel zur Symbolleistengestaltung erreichbar:

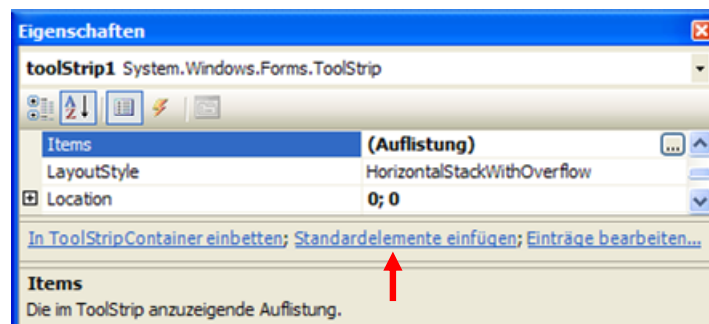


Hier lassen sich Steuerelemente aus allen zulässigen Klassen bequem einfügen, positionieren und konfigurieren.

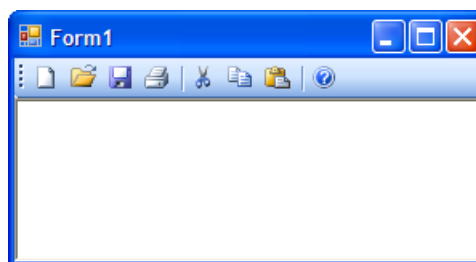
Als Clou bietet das Eigenschaftsfenster in der Befehlszone am unteren Rand, die eventuell zunächst per Kontextmenü



aktiviert werden muss, über den Link **Standardelemente einfügen** die Möglichkeit, mit einem einzigen Mausklick alle Standardkomponenten auf die aktuell bearbeitete Symbolleiste zu setzen:



Ein Link zum Einfügen der Standardelemente findet sich auch im oben beschriebenen Formular mit **ToolStrip-Aufgaben**. Das Ergebnis ist beeindruckend:

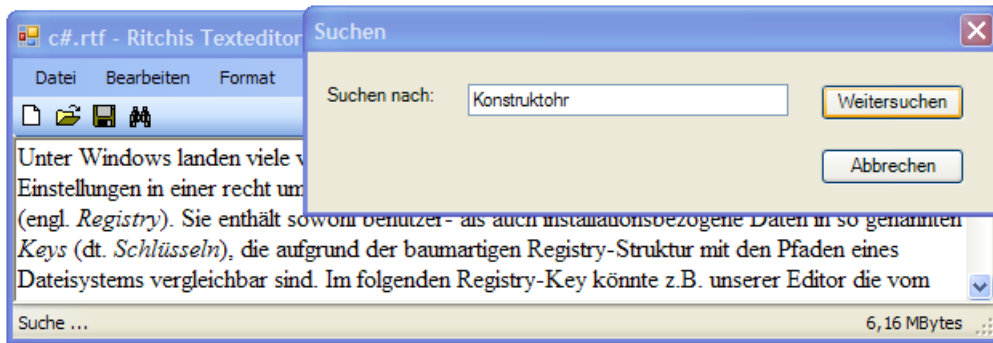


20.2 Statusleiste

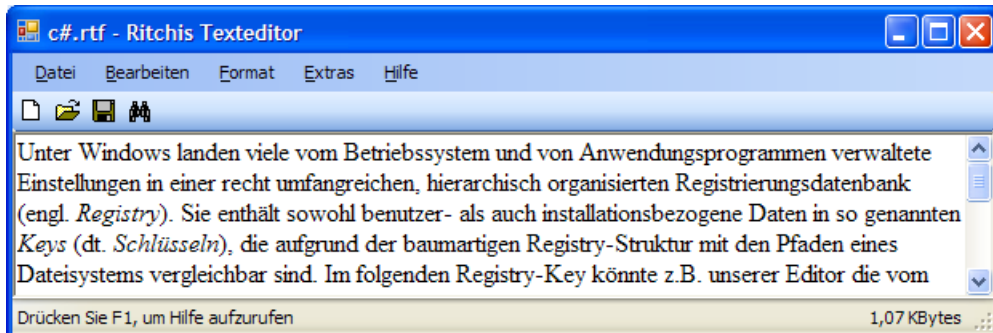
Viele Windows-Programme benutzen eine Statuszeile am unteren Fensterrand, um wichtige Informationen über den Programmzustand anzuzeigen. Unser Editor auf **RichTextBox**-Basis erhält nun eine Statusleiste mit folgenden Bestandteilen:

- eine kontextabhängige Informationsanzeige am linken Rand
- eine Anzeige der aktuellen Textlänge am rechten Rand

Im Infosegment soll z.B. ggf. angezeigt werden, dass der Editor momentan mit einem Suchauftrag beschäftigt ist:



Liegen keine besonderen Informationen an, erscheint ein Hinweis auf das (noch unterentwickelte) Hilfesystem:



Wie am Anfang von Abschnitt 20 zu sehen war, stammt die zur Realisation einer Statusleiste empfehlenswerte Klasse **StatusStrip** von der für Symbolleisten zuständige Klasse **ToolStrip** ab. Bei der Bestückung mit Komponenten aus der **ToolStripItem**-Familie bietet die Klasse **StatusStrip** im Vergleich zu ihrer Basisklasse (siehe Abschnitt 20.1) eine eingeschränkte Wahl:

- **ToolStripStatusLabel**
Objekte dieser von **ToolStripLabel** abstammenden Klasse dienen zur Anzeige von Texten.
- **ToolStripDropDownButton**
Diese Klasse realisiert einen Schalter zum Öffnen eines DropDown-Menüs
- **ToolStripSplitButton**
Hier wird ein Kombiwerkzeug geboten bestehend aus einem Befehlsschalter auf der linken Seite und einem Schalter zum Öffnen eines DropDown-Menüs auf der rechten Seite.
- **ToolStripProgressBar**
Diese Klasse realisiert einen Fortschrittsbalken.
- **ToolStripSeparator**
Wie bei Menü- und Symbolleisten sind Trennlinien verfügbar.

Wir erzeugen im Fensterkonstruktor ein **StatusStrip**-Objekt und zwei **ToolStripStatusLabel**-Objekte:

```
StatusStrip statusStrip = new ToolStrip();
sblInfo = new ToolStripStatusLabel();
sblTextSize = new ToolStripStatusLabel();
```

Damit die **ToolStripStatusLabel**-Objekte in allen Methoden ansprechbar sind, benötigen wir Referenzvariablen auf Klassenebene:

```
ToolStripStatusLabel sblInfo, sblTextSize;
```

Das **StatusStrip**-Objekt wird in die Steuerelement-Kollektion des Formulars aufgenommen

```
Controls.Add(statusStrip);
```

und findet selbständig seinen Platz am unteren Fensterrand.

Die **StatusStrip**-Eigenschaft **Items** zeigt auf ein Objekt der Klasse **ToolStripItemCollection** zur Verwaltung der Statuszeilenkomponenten, das u.a. die Methoden **Add()** und **AddRange()** beherrscht:

```
statusStrip.Items.AddRange(new ToolStripItem[] {sblInfo, sblTextSize});
```

Weil das **ToolStripStatusLabel**-Objekt zur Infoanzeige *zuerst* aufgenommen wird, erscheint es am linken Rand der Statusleiste. Es richtet seinen Text linksbündig aus und informiert initial über das (noch kaum vorhandene) Hilfesystem:

```
sblInfo.TextAlign = ContentAlignment.MiddleLeft;
sblInfo.Text = "Drücken Sie F1, um Hilfe aufzurufen";
```

Wir erlauben der Infoanzeige mit dem Wert **true** für die Eigenschaft **Spring**, wie eine elastische Feder (dt. Übersetzung von *Spring*) den jeweils in horizontaler Richtung verfügbaren Platz einzunehmen:

```
sblInfo.Spring = true;
```

Damit wird das **ToolStripStatusLabel**-Objekt zur Anzeige der Textgröße an den rechten Rand der Statusleiste geschoben. Immerhin darf es seine horizontale Ausdehnung an den Platzbedarf anpassen:

```
sblTextSize.AutoSize = true;
sblTextSize.TextAlign = ContentAlignment.MiddleRight;
sblTextSize.Text = "0 Bytes";
```

Zur Aktualisieren der Textlängenanzeige erweitern wir die beim **RichTextBox**-Ereignis **TextChanged** registrierte Methode **RichTextBoxOnTextChanged()**, die wir schon zur Aktualisierung der Verfügbarkeit von Symbolleisten-Schaltflächen verwendet haben (siehe Abschnitt 20.1.1):

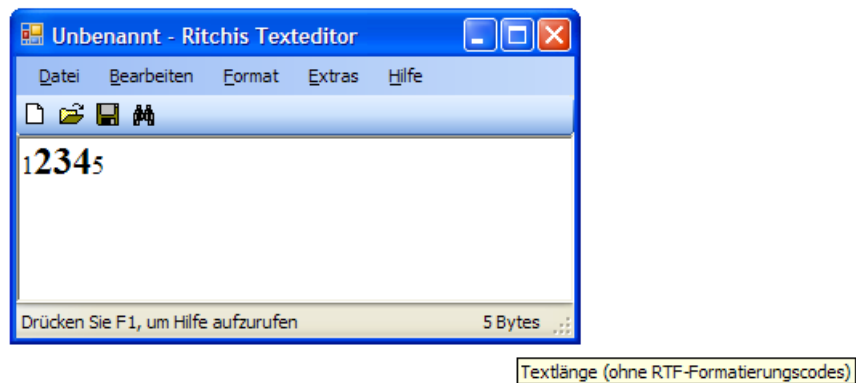
```
editor.TextChanged += new EventHandler(RichTextBoxOnTextChanged);

protected void RichTextBoxOnTextChanged(object sender, EventArgs ea) {
    double d = editor.TextLength;
    if (editor.TextLength > 0)
        tsbFind.Enabled = true;
    else
        tsbFind.Enabled = false;

    if (d < 1024)
        sblTextSize.Text = String.Format("{0:f0}", d) + " Bytes ";
    else {
        d /= 1024;
        if (d < 1024)
            sblTextSize.Text = String.Format("{0:f2}", d) + " KBytes ";
        else {
            d /= 1024;
            if (d < 1024)
                sblTextSize.Text = String.Format("{0:f2}", d) + " MBytes ";
            else {
                d /= 1024;
                sblTextSize.Text = String.Format("{0:f2}", d) + " GBytes ";
            }
        }
    }
}
```

Sollte sich die (im GUI-Thread ausgeführte) **TextChanged**-Ereignismethode einmal als Leistungsbremse erweisen, kann man die Statusleistenaktualisierungen über ein **Timer**-Objekt realisieren (siehe Abschnitte 13.7 und 16.6).

Über die **ToolStripStatusLabel**-Eigenschaft **ToolTipText** lässt sich der Komfort unseres Editors noch steigern:



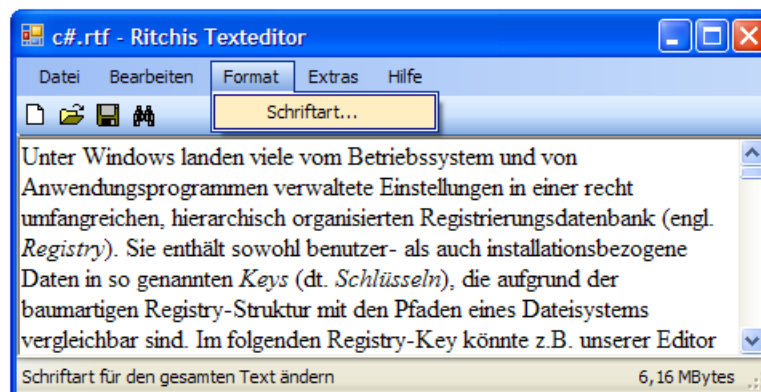
Mit der folgenden Anweisung

```
sblTextSize.ToolTipText = "Textlänge (ohne RTF-Formatierungscodes)";
```

sorgen wir dafür, dass beim kurzen Verharren des Mauszeigers über der Textlängenanzeige ein Quick-Info - Text darüber informiert, dass eventuell vorhandene RTF-Codes (z.B. für Textauszeichnungen) *nicht* in die Berechnung eingehen. Zusätzlich muss die Statusleiste noch gebeten werden, die Anzeige von Quick-Infos für ihre Items zu unterstützen:

```
statusStrip.ShowItemToolTips = true;
```

Ein weiterer Standardservice für Windows-Anwender ist die Anzeige einer kurzen Erläuterung zum aktuell gewählten Menüitem in der Statusleiste, z.B.:



Bei der Realisation ist zunächst dafür zu sorgen, dass nach dem Verlassen des Menüs der Statusleisten-Infotext wieder denselben Inhalt hat wie beim Öffnen. Dazu erstellen wir die folgenden Ereignismethoden

```
protected void MainMenuOnActivate(object sender, EventArgs ea) {
    infoTextBak = sblInfo.Text;
}
protected void MainMenuOnDeactivate(object sender, EventArgs ea) {
    sblInfo.Text = infoTextBak;
    infoTextBak = "";
}
```

und registrieren sie beim Ereignis **MenuActivate** bzw. **MenuDeactivate** des **MenuStrip**-Objekts zum Hauptmenü:

```
mainMenu.MenuActivate += MainMenuOnActivate;
mainMenu.MenuDeactivate += MainMenuOnDeactivate;
```

Die Erläuterung zu einem Menüitem soll in der Statusleiste erscheinen, sobald der Mauszeiger über dem Item verharret, wobei das **Control**-Ereignis **MouseHover** ausgelöst wird. Wir verwenden eine für *alle* Menüitems zuständige Ereignismethode:

```
protected void MenuItemOnMouseHover(object sender, EventArgs ea) {
    if (sender == mitNew)
        sblInfo.Text = "Neues Dokument beginnen";
    else if (sender == mitOpen)
        sblInfo.Text = "Dokument öffnen";

    . . .

    else if (sender == mitOptions)
        sblInfo.Text = "Optionen wählen";
    else if (sender == mitInfo)
        sblInfo.Text = "Informationen zum Programm anzeigen";
}
```

Sobald der Mauszeiger ein Menüitem verlässt, wird das **Control**-Ereignis **MouseLeave** ausgelöst, und die bei diesem Ereignis registrierte Methode ersetzt den aktuellen Infotext durch die leere Zeichenfolge, sofern das **MenuStrip**-Ereignis **MenuDeactivate** noch nicht abgearbeitet ist (erkennbar an der leeren Zeichenfolge in `infoTextBak`, siehe oben):

```
protected void MenuItemOnMouseLeave(object sender, EventArgs ea) {
    if (infoTextBak != "")
        sblInfo.Text = "";
}
```

Die beiden Ereignismethoden sind beim **MouseHover**- bzw. **MouseLeave**-Ereignis sämtlicher Untermenüitems zu registrieren, was zum Glück keinen großen Schreibaufwand verursacht:

```
EventHandler mitMouseHoverHandler = new EventHandler(MenuItemOnMouseHover);
EventHandler mitMouseLeaveHandler = new EventHandler(MenuItemOnMouseLeave);
foreach (ToolStripMenuItem mit in mainMenu.Items)
    foreach (ToolStripItem umit in mit.DropDownItems) {
        umit.MouseHover += mitMouseHoverHandler;
        umit.MouseLeave += mitMouseLeaveHandler;
    }
```

Bei einer komplizierteren Menüstruktur steigt der Registrierungsaufwand nur unwesentlich an.

Einen weiteren Pluspunkt könnte unser Editor durch Anzeige des Einfüge- bzw. Überschreibmodus in der Statusleiste sammeln.

Den vollständigen Quellcode des Editors im bisher vorgestellten Entwicklungszustand finden Sie im Ordner

...\\BspUeb\\WinForms\\Steuerelemente\\Statuszeile

20.3 Übungsaufgaben zu Kapitel 20

1) Erweitern Sie bei unserem Editor ausgehend vom Entwicklungsstand in

...\\BspUeb\\Statusleiste

die Symbolleiste um Schalter zu den **Bearbeiten**-Menüitems **Rückgängig**, **Wiederholen**, **Ausschneiden**, **Kopieren** und **Einfügen**.

21 Drucken

Wir machen uns nun daran, unseren Editor mit einer Druckfunktion auszustatten, die u.a. folgende Leistungsmerkmale bieten soll:

- Das Drucken kann über per Menüitem und per Symbolleistenschalter angefordert werden. Nach Anforderung per Menüitem erscheint der Standarddialog zum Drucken und bietet die üblichen Optionen (z.B. einen flexibel wählbaren Druckumfang). Nach Anforderung per Symbolschalter werden die aktuell gültigen Einstellungen zum Drucken verwendet.
- Der Standarddialog zur Seitenansicht (Druckvorschau) ist per Menüitem und über einen funktionsgleichen Symbolleistenschalter erreichbar.
- Der Standarddialog zur Seiteneinrichtung ist per Menüitem verfügbar.

21.1 Wichtige Datentypen

Bei der Verwaltung von Druckaufträgen spielen Objekte der Klasse **PrintDocument** die zentrale Rolle. Diese verwenden zur Verwaltung von Drucker- bzw. Seiteneinstellungen Objekte aus den Klassen **PrinterSettings** bzw. **PageSettings**. Alle genannten Klassen stammen aus dem Namensraum **System.Drawing.Printing**. Auch aus dem vorgeordneten Namensraum **System.Drawing** sind viele Typen beteiligt, z.B. die wichtige Klasse **Graphics**. Generell ist der Abschnitt 15 über die *Grafikausgabe mit dem GDI+* höchst relevant für unser aktuelles Thema. Bei der Realisation einer bequemen und flexiblen Druckfunktion werden sich die Standarddialog-Klassen **PrintDialog**, **PageSetupDialog** und **PrintPreviewDialog** aus dem Namensraum **System.Windows.Forms** als nützlich erweisen (für Programmierer und Anwender). Außerdem sind noch weitere Typen im Spiel, z.B. die Klasse **PrintPageEventArgs** als Parametertyp für Behandlungsmethoden zum wichtigen **PrintDocument**-Ereignis **PrintPage**. Wir beschäftigen uns zunächst mit den Klassen **PrinterSettings** und **PageSettings**.

21.1.1 Die Klasse **PrinterSettings** zur Verwaltung von Druckereinstellungen

Ein Objekt der Klasse **PrinterSettings** enthält die beim Drucken relevanten Geräteeinstellungen. Über etliche Eigenschaften kann man einen Drucker wählen sowie dessen Einstellungen ermitteln und/oder modifizieren, z.B.:

PrinterSettings-Eigenschaft	Beschreibung
PrinterName	Typ: String Zugriff: get, set Über diese Eigenschaft kann der zugeordnete Drucker abgefragt oder geändert werden. Per Voreinstellung kommt der Standarddrucker des angemeldeten Benutzers zum Einsatz. Wird der Name eines anderen Druckers eingetragen, ändern sich die übrigen PrinterSettings -Eigenschaften dementsprechend.
IsValid	Typ: bool Zugriff: get Zum Wert false kommt es z.B. dann, wenn PrinterName keinen Namen eines im aktuellen System verfügbaren Druckers enthält.
DefaultPageSettings	Typ: PageSettings Zugriff: get Beim Erzeugen eines PrinterSettings -Objekts wird stets ein PageSettings -Objekt (siehe Abschnitt 21.1.2) mit den voreingestellten Seiteneigenschaften des Druckers erstellt. Bei einer validen Änderung der PrinterName -Eigenschaft wird das PageSettings -Objekt aktualisiert.
SupportsColor	Typ: bool Zugriff: get

PrinterSettings-Eigenschaft	Beschreibung
	Gibt an, ob Farbdrucke möglich sind
CanDuplex	Typ: bool Zugriff: get Gibt an, ob beidseitiges Drucken möglich ist
Duplex	Typ: Duplex Zugriff: get, set Gibt an, ob derzeit beidseitiges Drucken eingestellt ist
LandscapeAngle	Typ: int Zugriff: get Bei verfügbarer Landscape-Orientierung lautet der Wert 90 oder 270, anderenfalls 0. Man erhält den Drehwinkel des Querformats gegenüber dem Hochformat.
PaperSizes	Typ: PrinterSettings.PaperSizeCollection Zugriff: get Zeigt auf eine Kollektion mit den unterstützten Papiergrößen, jeweils beschrieben durch ein PaperSize -Objekt
PaperSources	Typ: PrinterSettings.PaperSourceCollection Zugriff: get Zeigt auf eine Kollektion mit den verfügbaren Papierquellen, jeweils beschrieben durch ein PaperSource -Objekt
PaperResolutions	Typ: PrinterSettings.PrinterResolutionCollection Zugriff: get Zeigt auf eine Kollektion mit den unterstützten Auflösungen, jeweils beschrieben durch ein PaperResolution -Objekt
PrintRange	Typ: PrintRange Zugriff: get, set Ist dem Benutzer im Standarddialog Drucken (siehe Abschnitt 21.1.4) eine flexible Auswahl des Druckumfangs erlaubt, enthält die Eigenschaft PrintPage (vom gleichnamigen Aufzählungstyp mit den Werten Allpages , CurrentPage , Selection , und SomePages) seine Entscheidung.
FromPage	Typ: int Zugriff: get, set Über diese Eigenschaft wird die Nummer der ersten zu druckenden Seite ermittelt bzw. festgelegt. Im Standarddialog Drucken (siehe Abschnitt 21.1.4) kann der Benutzer diese Seitennummer bestimmen.
ToPage	Typ: int Zugriff: get, set Über diese Eigenschaft wird die Nummer der letzten zu druckenden Seite ermittelt bzw. festgelegt. Im Standarddialog Drucken (siehe Abschnitt 21.1.4) kann der Benutzer diese Seitennummer bestimmen.
MinimumPage	Typ: int Zugriff: get, set Stellt oder legt die kleinste Seitennummer fest, die im Standarddialog Drucken (siehe Abschnitt 21.1.4) bei der Wahl eines Seitenbereichs als Druckumfang erlaubt ist
MaximumPage	Typ: int Zugriff: get, set Stellt oder legt die größte Seitennummer fest, die im Standarddialog Drucken (siehe Abschnitt 21.1.4) bei der Wahl eines Seitenbereichs als Druckumfang erlaubt ist

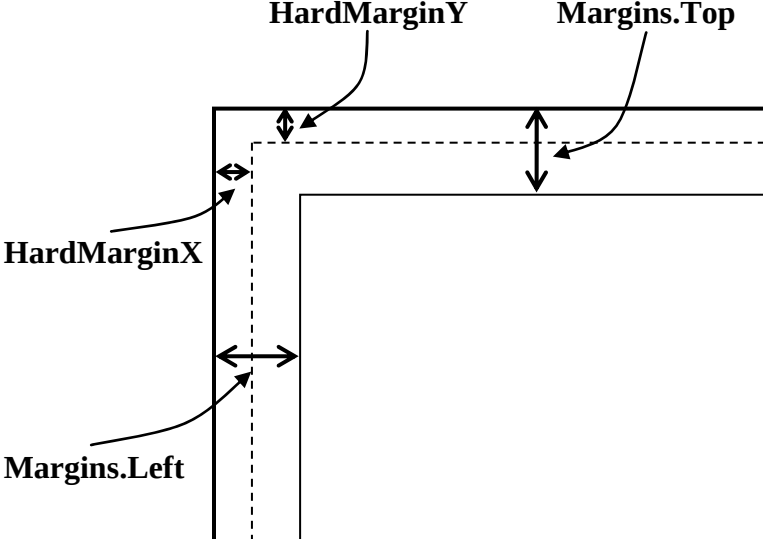
Über die statische Eigenschaft **InstalledPrinters** der Klasse **PrinterSettings** erhält man ein Objekt der internen Klasse **PrinterSettings.StringCollection** mit den Namen der verfügbaren Drucker.

Das Erstellen eines **PrinterSettings**-Objekts per Konstruktor findet eher selten statt. Meist verwendet man die in einem **PrintDocument**-Objekt (siehe Abschnitt 21.1.3) enthaltene **PrinterSettings**-Instanz. Diese ist zunächst mit dem Standarddrucker des angemeldeten Benutzers verbunden, kann aber beliebig konfiguriert werden.

21.1.2 Die Klasse **PageSettings** zur Verwaltung von Seiteneigenschaften

Einzelne Druckseiten werden durch Objekte der Klasse **PageSettings** beschrieben, die u.a. folgende Eigenschaften haben:

PageSettings-Eigenschaft	Beschreibung
PrinterSettings	Typ: PrinterSettings Zugriff: get, set Mit einem PageSettings -Objekt ist stets ein PrinterSettings -Objekt verbunden, das Initialisierungswerte für viele PageSettings -Eigenschaften liefert. Wird dem PageSettings -Konstruktor kein PrinterSettings -Objekt als Parameter übergeben, dann erzeugt er sich ein solches Objekt, das mit dem Standarddrucker verbunden ist.
Landscape	Typ: bool Zugriff: get, set Stellt oder legt fest, ob die Seite im Querformat gedruckt werden soll
PaperSize	Typ: PaperSize Zugriff: get, set Diese Eigenschaft zeigt auf ein PaperSize -Objekt, mit dem die Papiergröße in der Einheit 1/100 Zoll abgefragt oder eingestellt werden kann. Ein neuer Wert muss in der PaperSizes -Kollektion des via PrinterSettings -Wert verbundenen Druckers enthalten sein. Dabei ist die Landscape -Einstellung <i>irrelevant</i> .
Bounds	Typ: Rectangle Zugriff: get Diese Eigenschaft zeigt auf ein Rectangle -Objekt mit der Seitengröße in 1/100 Zoll, wobei die per Landscape -Eigenschaft festgelegte Seitenausrichtung berücksichtigt wird. Bei Landscape = false liefert Bounds dieselbe Größe wie PaperSize .
Margins	Typ: Margins Zugriff: get, set Über diese Eigenschaft können die vier Seitenränder ermittelt oder festgelegt werden (Maßeinheit: 1/100 Zoll, Voreinstellung: 1 Zoll an allen Seiten). Für den Zugriff auf die einzelnen Ränder enthält ein Margins -Objekt die int -Eigenschaften Bottom , Left , Right und Top . Über den Standarddialog Seiteneinrichtung (siehe Abschnitt 21.3.2) kann man dem Benutzer die Wahl der Seitenränder ermöglichen.
PrintableArea	Typ: RectangleF Zugriff: get Drucker können das Papier in der Regel nicht bis zu den Rändern bedrucken, weil z.B. Randstreifen für den Papiertransport benötigt werden. Die seit .NET 2.0 verfügbare Eigenschaft PrintableArea zeigt auf ein RectangleF -Objekt mit der bedruckbaren Seitengröße in 1/100 Zoll, wobei die per Landscape -Eigenschaft festgelegte Seitenausrichtung berücksichtigt wird.

PageSettings-Eigenschaft	Beschreibung
HardMarginX , HardMarginY	Typ: float Zugriff: get Über die seit .NET 2.0 verfügbaren Eigenschaften HardMarginX bzw. HardMarginY ermittelt man die Breite des druckerbedingten Rands in X- bzw. Y-Richtung (Einheit: 1/100 Zoll). 
PaperSource	Typ: PaperSource Zugriff: get, set Mit dieser Eigenschaft stellt oder legt man die Papierquelle fest. Über die verfügbaren Papierquellen informiert die PrinterSettings-Eigenschaft PaperSources (siehe oben).
PrinterResolution	Typ: PrinterResolution Zugriff: get, set Mit dieser Eigenschaft stellt oder legt man die Druckauflösung fest. Über die verfügbaren Auflösungen informiert die PrinterSettings-Eigenschaft PrinterResolutions (siehe oben).
Color	Typ: bool Zugriff: get, set Stellt oder legt fest, ob die Seite in Farbe gedruckt werden soll

Zwar verfügt die Klasse **PageSettings** über zwei Konstruktoren, doch verwendet man meist die in einem **PrintDocument**-Objekt (siehe Abschnitt 21.1.3) enthaltene **PageSettings**-Instanz, um Seiteneigenschaften festzulegen.

21.1.3 Die Klasse **PrintDocument** zur Verwaltung von Druckaufträgen

Beim Drucken fungiert ein Objekt der Klasse **PrintDocument** als zentrale Koordinierungsstelle, so dass seine Aufgaben zunächst einmal im Überblick dargestellt werden:

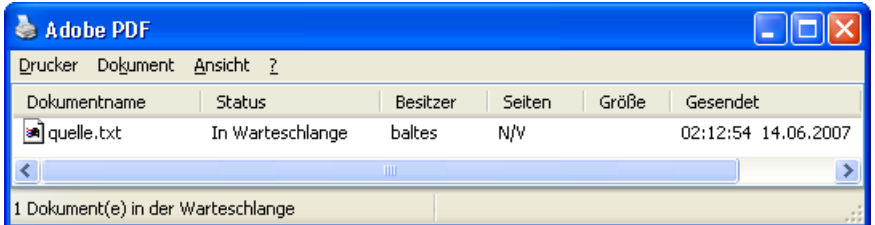
- Verwaltung der Einstellungen zum Druckauftrag
Dazu dienen diverse **PrintDocument**-Eigenschaften. Insbesondere sind ein **PrinterSettings**- und ein **PageSettings**-Objekt eingebunden.
- Starten des Druckauftrags
Dazu wird seine **Print()**-Methode aufgerufen.

- Ablaufsteuerung über Ereignisse
Die eigentliche Druckausgabe erfolgt in der Behandlungsmethode zum **PrintDocument**-Ereignis **PrintPage**. Weitere Ereignisse werden in Abschnitt 21.1.3.2 behandelt.

Obwohl es der Name **PrintDocument** suggeriert, dienen Objekte dieser Klasse *nicht* zur Aufnahme der auszugebenden Daten.

21.1.3.1 Eigenschaften und Methoden

Ein Objekt der Klasse **PrintDocument** besitzt u.a. die folgenden Eigenschaften:

PrintDocument-Eigenschaft	Beschreibung
PrinterSettings	Typ: PrinterSettings Zugriff: get, set Diese Eigenschaft zeigt auf ein Objekt aus der Klasse PrinterSettings , das vom PrintDocument -Konstruktor angelegt wird. ⁷² Es ist zunächst mit dem Standarddrucker des angemeldeten Benutzers verbunden. Bei der Druckausgabe werden die Eigenschaften dieses PrinterSettings -Objekts verwendet.
DefaultPageSettings	Typ: PageSettings Zugriff: get, set Diese Eigenschaft zeigt auf ein PageSettings -Objekt, das ebenfalls vom PrintDocument -Konstruktor angelegt wird. Bei der Druckausgabe werden per Voreinstellung die Eigenschaften dieses PageSettings -Objekts verwendet.
DocumentName	Typ: String Zugriff: get, set Über diese Eigenschaft stellt bzw. legt man den Name des Druckauftrags fest. Der Name wird auch vom Betriebssystem verwendet und z.B. in der Warteschlange eines Druckes angezeigt: 
OriginAtMargins	Typ: bool Zugriff: get, set Bei der via PrintDocument -Objekt veranlassten Druckausgabe kommt ein Graphics -Objekt ins Spiel, das wir gemäß Abschnitt 15

⁷² Den **PrintDocument**-Konstruktor kann sich ungefähr so vorstellen:

```
public PrintDocument() {
    documentname = "document"; //official default.
    printersettings = new PrinterSettings(); // use default values
    defaultpagesettings =
        (PageSettings) printersettings.DefaultPageSettings.Clone ();
    printcontroller = new StandardPrintController();
}
```

Dieses Exemplar stammt aus dem **Mono**-Projekt, der Open Source - .NET-Portierung für Linux (Version 2, siehe <http://ftp.novell.com/pub/mono/sources-stable/>).

PrintDocument-Eigenschaft	Beschreibung
	(Grafikausgabe mit dem GDI+) benutzen. Über die boolesche Eigenschaft OriginAtMargins stellt bzw. legt man fest, wo sich die Ausgangsposition (0, 0) des Graphics-Objekts befinden soll: • false (= Voreinstellung) Der Ursprung (0, 0) ist von der oberen linken Ecke des Papiers in horizontaler bzw. vertikaler Richtung um den jeweiligen druckerbedingten Rand entfernt. Die PageSettings-Eigenschaft Margins wird ignoriert. • true Der Ursprung (0, 0) ist von der oberen linken Ecke des Papiers in horizontaler bzw. vertikaler Richtung um den jeweiligen Seitenrand entfernt. Die PageSettings-Eigenschaft Margins wird also respektiert. Bei dieser Einstellung resultieren für Druckausgaben in der Randzone (z.B. Kopf- oder Fußzeilen) negative Koordinaten.

Ein Objekt der Klasse **PrintDocument** wird in der Regel explizit per Konstruktor erzeugt, z.B.

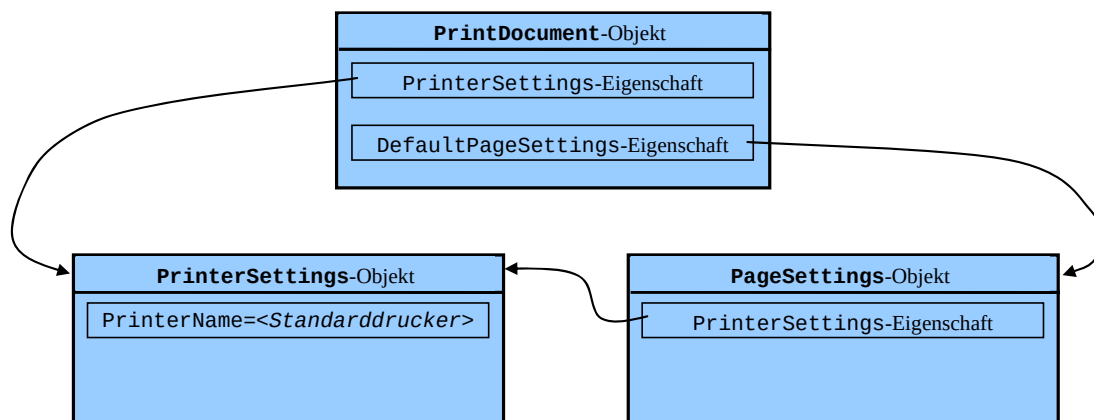
```
PrintDocument pd = new PrintDocument();
```

Es verfügt über ...

- ein „eingebautes“ **PrinterSettings**-Objekt, ansprechbar über die Eigenschaft **PrinterSettings**, das per Voreinstellung mit dem Standarddrucker des Benutzers verbunden ist. Dieses **PrinterSettings**-Objekt kann modifiziert werden, um die Druckausgabe zu beeinflussen. Insbesondere ist der zugeordnete Drucker veränderbar, z.B.:

```
pd.PrinterSettings.PrinterName = "Adobe PDF";
```
- ein „eingebautes“ **PageSettings**-Objekt, ansprechbar über die Eigenschaft **DefaultPageSettings**, das mit dem „eingebauten“ **PrinterSettings**-Objekt verbunden ist. Dieses **PageSettings**-Objekt kann modifiziert werden, um die Druckausgabe zu beeinflussen.

Man kann die Situation folgendermaßen skizzieren:



In dieser Abbildung wurde ein **PageSettings**-Objekt weggelassen, das für Verwirrung sorgen könnte: Das dem **PrinterSettings**-Objekt „untergeordnete“ Objekt mit Seitenvoreinstellungen. Man sorgt z.B. mit folgender Anweisung bei einem **PrintDocument** namens **pd** für einen Ausdruck im Querformat:

```
pd.DefaultPageSettings.Landscape = true;
```

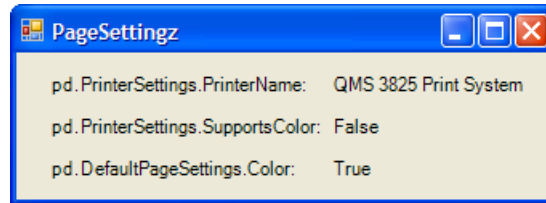
Demgegenüber bleibt die nächste Anweisung ohne Effekt, weil das falsche **PageSettings**-Objekt angesprochen wird:

```
pd.PrinterSettings.DefaultPageSettings.Landscape = true;
```

Es ist durchaus möglich, inkonsistente Einstellungen vorzunehmen, z.B. mit folgenden Zeilen

```
pd.PrinterSettings.PrinterName = "QMS 3825 Print System";
pd.DefaultPageSettings.Color = true;
```

den farbigen Druck einer Seite zu wählen, obwohl ein Schwarzweißdrucker eingestellt ist:



Um einen Druckvorgang auszulösen, ruft man die **Print()**-Methode des **PrintDocument**-Objekts auf, z.B.:

```
pd.Print();
```

Damit kommt eine Kette von Ereignissen in Gang, die anschließend beschrieben wird.

21.1.3.2 Ereignisse

Nach dem Aufruf seiner **Print()**-Methode feuert ein **PrintDocument**-Objekt nacheinander die folgenden **Ereignisse** und ruft ggf. die registrierten Behandlungsmethoden auf:⁷³

- **BeginPrint**
Dieses Ereignis erlaubt Vorbereitungen für einen Druckauftrag (z.B. Erstellen von **Font**-Objekten). Dem Handler wird als Aktualparameter ein Objekt der Klasse **PrintEventArgs** übergeben, das über seine **Cancel**-Eigenschaft einen Abbruch des Druckauftrags ermöglicht. Weitere Details zum **BeginPrint**-Handler und ein ausführliches Beispiel folgen in Abschnitt 21.2.2.
- **QueryPageSettings** (für jede einzelne Seite)
Die Ereignisbehandlungsmethode erhält beim Aufruf ein **QueryPageSettingsEventArgs**-Objekt und kann über dessen Eigenschaft **PageSettings** spezifische Einstellungen für die nächste auszudruckende Seite festlegen, z.B. die Seitenausrichtung. Über die **Cancel**-Eigenschaft des Ereignisobjekts lässt sich der Druckauftrag abbrechen. Die im Anschluss beschriebene **PrintPage**-Behandlungsmethode erhält zwar ebenfalls eine **PageSettings**-Referenz, darf diese aber nur lesend verwenden.
- **PrintPage** (für jede einzelne Seite)
Dem Handler wird als Aktualparameter ein Objekt der Klasse **PrintPageEventArgs** übergeben, das u.a. ein **Graphics**-Objekt (vgl. Abschnitt 15.1) mitbringt, das nun ein Befüllen der Seite ermöglicht. Offenbar funktioniert das **PrintDocument**-Ereignis **PrintPage** weitgehend analog zu dem bei Bildschirmausgaben essentiellen **Paint**-Ereignis der Klasse **Control** (vgl. Abschnitt 15.3). Soll eine weitere Seite ausgegeben werden, ist die Eigenschaft **HasMorePages** des **PrintPageEventArgs**-Objekts auf **true** zu setzen. Weitere Details zum **PrintPage**-Handler und ein ausführliches Beispiel folgen in Abschnitt 21.2.3.

⁷³ Genau genommen ist dabei noch ein Objekt der Klasse **PrintController** im Spiel. Man sollte diese Klasse nicht direkt einsetzen, so dass sie hier auch nicht beschrieben wird. In Abschnitt 21.3.1 kommt eine **PrintController**-Ableitung zur Sprache, die beim Erstellen der Seitenansicht hilft.

- **EndPrint**

Dieses Ereignis tritt ein, wenn der Druckauftrag abgeschlossen ist, abgebrochen wurde oder zu einer Ausnahme geführt hat. In der Behandlungsmethode kann man z.B. GDI- oder Dateisystem-Ressourcen per **Dispose()**- bzw. **Close()**-Aufruf freigeben.

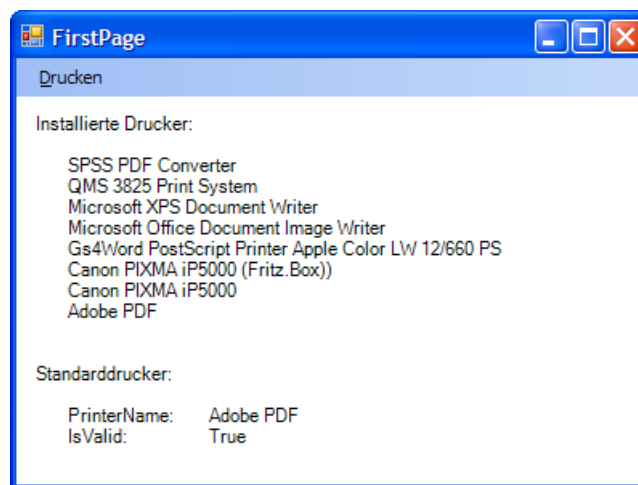
Sind keine Behandlungsmethoden für die **PrintDocument**-Ereignisse registriert, führen die beiden folgenden Zeilen

```
PrintDocument pd = new PrintDocument();
pd.Print();
```

zum Ausdruck einer leeren Seite durch den Standarddrucker des Benutzers.

21.1.3.3 Beispiel

Bevor wir die gerade angesprochenen und weitere Klassen im inzwischen relativ komplexen Editor-Projekt einsetzen, gönnen wir uns noch ein übersichtliches Beispiel. Das folgende Programm listet die verfügbaren Drucker auf und nennt den Standarddrucker:



Über das Hauptmenüitem **Drucken** können diese Informationen auch zu Papier gebracht werden.

Zunächst wird der Namensraum **System.Drawing.Printing** importiert, um seine Klassen (z.B. **PrintDocument**) im Quelltext bequem ansprechen zu können:

```
using System.Drawing.Printing;
```

An die Liste der installierten Drucker gelangt man über die statische Eigenschaft **InstalledPrinters** der Klasse **PrinterSettings**, die auf ein Objekt der internen Klasse **PrinterSettings.StringCollection** zeigt:

```
PrinterSettings.StringCollection pc = PrinterSettings.InstalledPrinters;
```

Indem wir das für eine Druckausgabe obligatorische **PrintDocument**-Objekt erzeugen, haben wir automatisch auch „eingebaute“ Objekte aus den Klassen **PrinterSettings** und **PageSettings** zur Verfügung

```
PrintDocument pd = new PrintDocument();
```

In der folgenden Methode **DrawToDevice()** werden die Informationen von einem per Parameter bestimmten **Graphics**-Objekt ausgegeben:

```

void DrawToDevice(Graphics g) {
    int vpos = mainMenue.Height + 10;
    int rand = 10;
    int indent = 20;
    g.DrawString("Installierte Drucker:", Font, brush, rand, vpos);
    for (int i = 0; i < pc.Count; i++)
        g.DrawString(pc[i], Font, brush, rand+indent, vpos + (i+2)*Font.Height);
    vpos = vpos + pc.Count*Font.Height + 4*Font.Height;
    g.DrawString("Standarddrucker:", Font, brush, rand, vpos);
    g.DrawString("PrinterName:\t"+pd.PrinterSettings.PrinterName,
        Font, brush, rand+indent, vpos + 2*Font.Height);
    g.DrawString("IsValid:  \t"+pd.PrinterSettings.IsValid.ToString(),
        Font, brush, rand + indent, vpos + 3 * Font.Height);
}

```

Diese Methode wird in der folgenden **OnPaint()**-Überschreibung des Formulars zur Bildschirm-
ausgabe

```

protected override void OnPaint(PaintEventArgs e) {
    DrawToDevice(e.Graphics);
}

```

und in der folgenden **PrintPage**-Ereignisbehandlungsmethode zur Druckausgabe genutzt:

```

void PrintDocumentOnPrintPage(object sender, PrintPageEventArgs e) {
    DrawToDevice(e.Graphics);
}

```

Der **PrintPage**-Handler wird beim entsprechenden Ereignis des **PrintDocument**-Objekts registriert:

```

pd.PrintPage += new PrintPageEventHandler(PrintDocumentOnPrintPage);

```

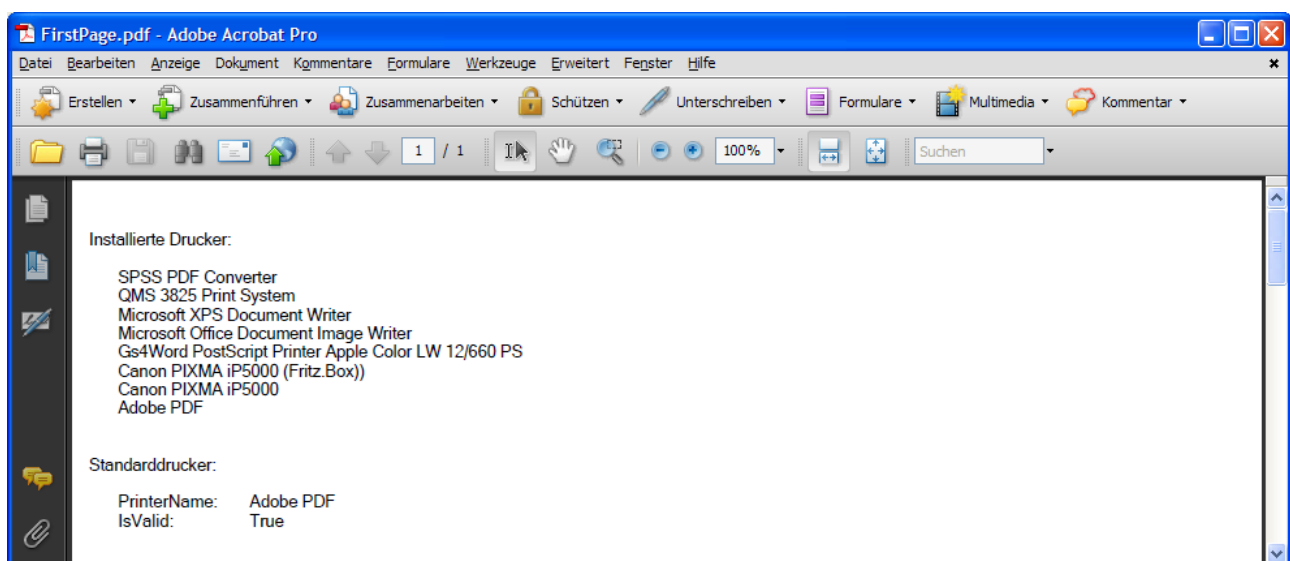
Im **Click**-Handler zum Hauptmenüitem **Drucken** wird die **Print()**-Methode des **PrintDocument**-Objekts aufgerufen, um die Druckausgabe einzuleiten:

```

void MitPrintOnClick(object sender, EventArgs e) {
    pd.Print();
}

```

Ist ein PDF-Produzent als Standarddrucker eingerichtet, dann entsteht über den Menübefehl **Drucken** folgendes PDF-Dokument:



Bei den **DrawString()**-Aufrufen in der Methode **DrawToDevice()** sind etliche Positionsangaben enthalten, die vom jeweils ausführenden **Graphics**-Objekt in der eigenen Maßeinheit (festgelegt durch die Eigenschaft **PageUnit**) verstanden werden (vgl. Abschnitt 15.8.2). Dass **DrawTo**-

`Device()` sowohl auf dem Bildschirm als auch auf dem Drucker ein passables Ergebnis liefert, liegt an den kompatiblen Maßeinheiten der beteiligten **Graphics**-Objekte:

- Der Drucker arbeitet mit der metrischen Maßeinheit von 1/100 Zoll.
- Bei der Bildschirmausgabe wird die Maßeinheit Pixel verwendet und eine Auflösung von 96 DPI unterstellt. Weil diese Annahme meist realistisch ist, entspricht 1 Pixel ungefähr 1/100 Zoll.

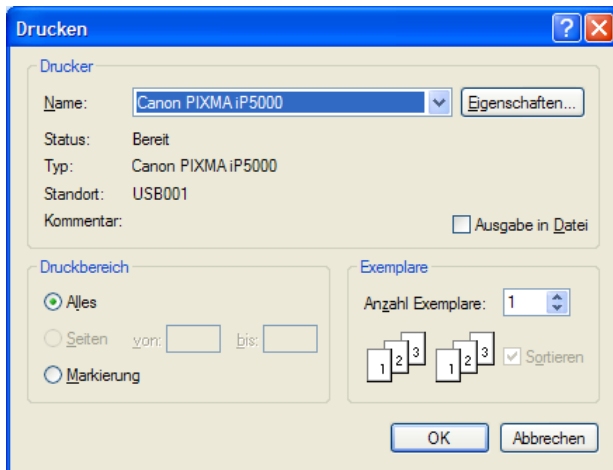
Den vollständigen Quellcode des Programms finden Sie im Ordner

...\\BspUeb\\Drucken\\FirstPage

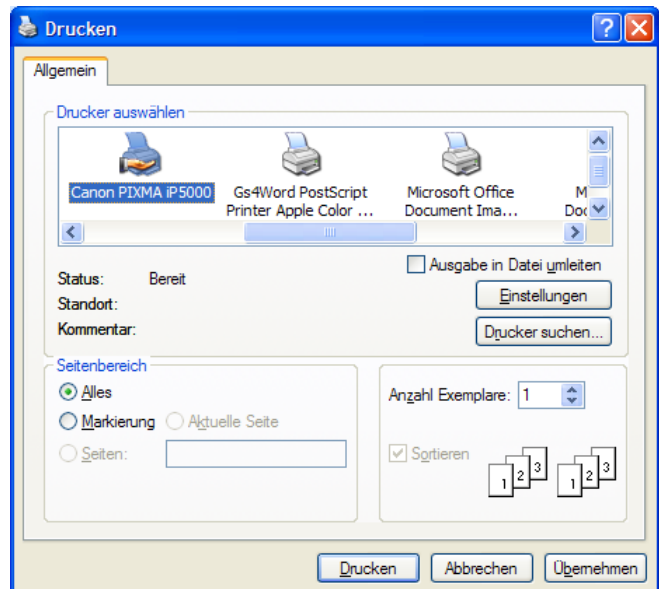
21.1.4 Der Standarddialog Drucken

Nach dem Menübefehl **Datei > Drucken** präsentieren Windows-Anwendungen meist den Standarddialog **Drucken**, der im .NET-Framework über die Klasse **PrintDialog** aus dem Namensraum **System.Windows.Forms** realisiert wird:

Einfache Version



Erweiterte Version



Hier können Benutzer ihren Druckauftrag auf vielfältige Weise beeinflussen, z.B.:

- Auswahl des **Druckers**
- Auswahl eines **Druckbereichs**
- Seitenorientierung (über den Schalter **Eigenschaften** bzw. **Einstellungen**)

Damit innerhalb eines Programmlaufs die gewählten Einstellungen erhalten bleiben, empfiehlt sich die Verwendung eines einzigen **PrintDialog**-Objekts, das über eine Instanzvariable ansprechbar ist, z.B.:

```
class RitchisTexteditor : Form {
    . . .
    PrintDocument printJob = new PrintDocument();
    PrintDialog printDialog = new PrintDialog();
    . . .
}
```

Die per Druckdialog beim Benutzer eingesammelten Informationen wirken sich auf ein **PrinterSettings**-Objekt aus, das folglich vor dem **ShowDialog()**-Aufruf festgelegt werden muss. Dazu stehen zwei **PrintDialog**-Eigenschaften zur Verfügung:

- **Document**

Bei der empfehlenswerten Technik wird ein **PrintDocument**-Objekt eingetragen, und dessen **PrinterSettings**-Eigenschaft legt das im Druckdialog zu modifizierende **PrinterSettings**-Objekt fest, z.B.:

```
printDialog.Document = printJob;
```

- **PrinterSettings**

Das im Druckdialog zu modifizierende **PrinterSettings**-Objekt wird direkt festgelegt.

Andere Eigenschaften zur Konfiguration des Dialogs:

PrintDialog-Eigenschaft	Beschreibung
AllowSelection	Typ: bool Zugriff: get, set Voreinstellung: false Soll der Benutzer als Druckumfang die aktuelle Auswahl angeben können?
AllowSomePages	Typ: bool Zugriff: get, set Voreinstellung: false Soll der Benutzer als Druckumfang einen Seitenbereich angeben können?
AllowCurrentPage	Typ: bool Zugriff: get, set Voreinstellung: false Soll der Benutzer als Druckumfang die aktuelle Seite angeben können? Damit der Wert true die gewünschte Folge hat, muss über die Eigenschaft UseEXDialog (siehe unten) der <i>erweiterte</i> Druckdialog gewählt werden.
AllowPrintToFile	Typ: bool Zugriff: get, set Voreinstellung: true Soll der Benutzer die Druckausgabe in eine Datei umleiten können?
PrintToFile	Typ: bool Zugriff: get, set Voreinstellung: false Setzt oder ermittelt den Zustand des Kontrollkästchens zur Dateiausgabe
ShowHelp	Typ: bool Zugriff: get, set Voreinstellung: false Wird die Eigenschaft auf den Wert true gesetzt, muss eine HelpRequest -Ereignisbehandlungsmethode implementiert und registriert werden.
UseEXDialog	Typ: bool Zugriff: get, set Voreinstellung: false Soll der erweiterte Druckdialog (siehe oben) verwendet werden? Mit Hilfe der zusätzlichen Schaltfläche Übernehmen kann man den zugeordneten Drucker ändern (z.B. mit Auswirkungen auf die Seitenansicht, vgl. Abschnitt 21.3.1), ohne eine Druckausgabe zu starten, wenn man den Druckdialog anschließend mit Abbrechen verlässt.

Ist dem Benutzer eine flexible Auswahl des Druckumfangs erlaubt, werden folgende Eigenschaften des eingebundenen **PrinterSettings**-Objekts relevant:

- **PrintRange**

Definition des Druckbereichs über einen Werte der Enumeration **PrintRange** (**Allpages**, **CurrentPage**, **Selection**, und **SomePages**)

- **MinimumPage** (Voreinstellung: 0)
Kleinste Seitennummer, die der Benutzer angeben kann
- **MaximumPage** (Voreinstellung: 9999)
Größte Seitennummer, die der Benutzer angeben kann
- **FromPage**
Vom Benutzer gewählte Startseite
- **ToPage**
Vom Benutzer gewählte Endseite

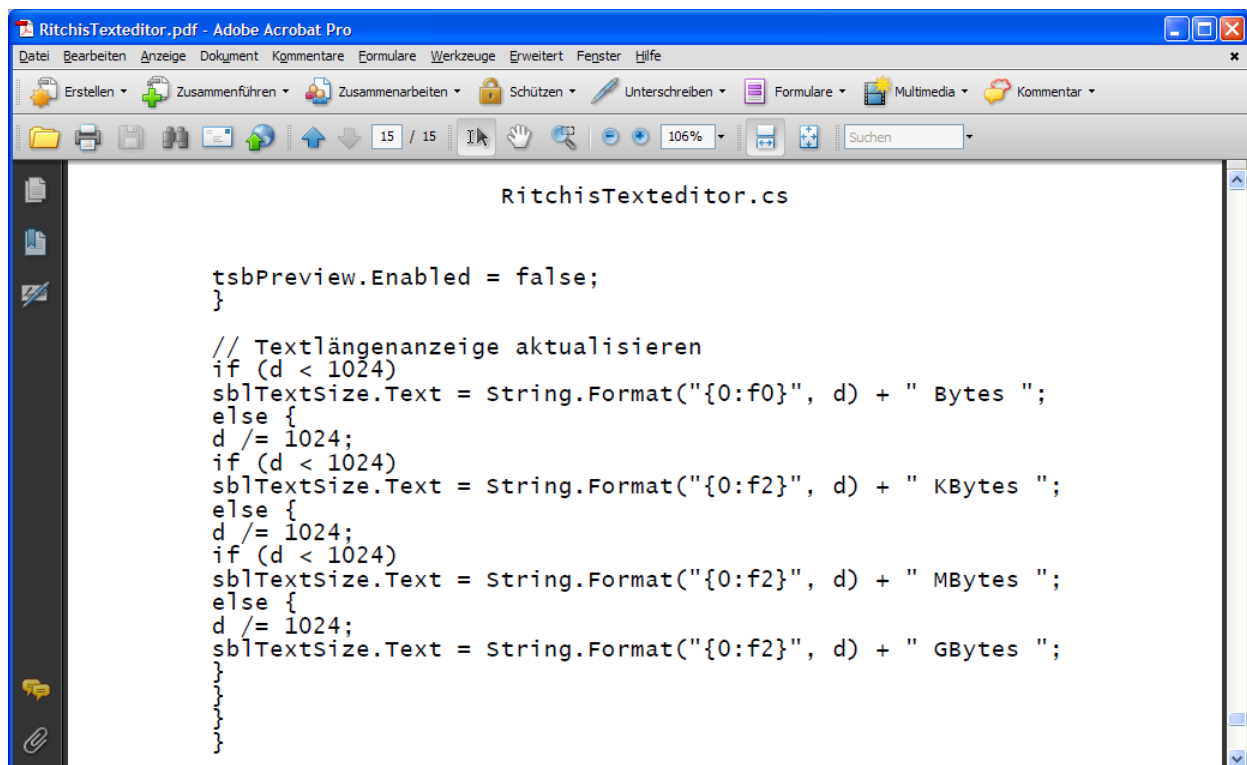
Der **PrintDialog** besorgt sich die zu präsentierenden Voreinstellungen vom **PrinterSettings**-Objekt und nimmt dort auch die vom Benutzer gewünschten Änderungen vor.

21.2 Druckfunktionalität für den Editor auf RichTextBox-Basis

Wir verwenden nun das neu erworbene Wissen dazu, unseren Editor auf **RichTextBox**-Basis ausgehend vom Entwicklungsstand in

...\BspUeb\WinForms\Steuerelemente>Statusleiste

mit einer Druckfunktion auszustatten. Dabei beschränken wir uns auf den Ausdruck von unformatiertem Text (monochrom in Schwarz), weil die Unterstützung von RTF-Textauszeichnungen oder gar von sonstigen Objekten (z.B. Bitmaps, Metafiles) den Rahmen sprengen würde. Dabei nutzen wir eine **DrawString()**-Überladung, die sich den Ausgabeort per **RectangleF**-Objekt vorschreiben lässt sogar einen automatischen Zeilenumbruch vornimmt (falls vom Benutzer gewünscht). Manche Texte lassen sich so ganz passabel zu Papier bringen, doch das PDF-Ergebnis beim Drucken des Quelltextes zur zentralen Editor-Klasse zeigt die Optimierungsreserven auf:



Neben den Farben fehlen im Vergleich zum RTF-Original vor allem die per Tabulator erzeugten Einrückungen. Bei einer Schrift mit fester Zeichenbreite könnte man Tabulatorzeichen durch eine jeweils passende (variable!) Anzahl von Leerzeichen ersetzen. Bei einer Proportionalchrift müsste man wohl die Textausgabe unterbrechen und mit einem neuen **DrawString()**-Aufruf an der Tabulatorzielposition weitermalen.

Tabulatorzeichen kann eine **RichTextBox**-Komponente übrigens per Voreinstellung nicht via Tastatur entgegen nehmen, weil die Tabulatorteste von der übergeordneten **ContainerControl**-Komponente zur Fokusverwaltung verwendet wird. Erst die folgende Eigenschaftsänderung macht eine Tabulatoreingabe per Tastatur möglich:

```
editor.AcceptsTab = true
```

Vermutlich beherrscht das **RichTextBox**-Steuerelement im .NET - Framework irgendwann einmal Methoden zum Drucken, so dass man bei dieser Komponente ohne Eigenleistung auskommt. Auf der Webseite

<http://www.microsoft.com/downloads/details.aspx?FamilyID=a79f5f90-8867-457d-93b1-00da386f6221&displaylang=en>

bietet Microsoft eine **RichTextBox**-Ableitung an, die das formatierte Drucken (WYSISYG!) über Routinen aus dem Win32-API realisiert. Aber im Abschnitt 21 geht es ja in erster Linie um die generelle .NET - Technik zum Drucken von Daten beliebiger Herkunft und nicht um den bequemsten Weg, den Text in einem **RichTextBox**-Steuerelement zu Papier zu bringen.

In den nächsten Abschnitten wird der Quellcode für die Druckunterstützung in unserem Editor auszugsweise vorgestellt. Eine vollständige Version finden Sie im Ordner

...\BspUeb\Drucken\RitchisTextEditorDrucken

21.2.1 Beteiligte Objekte

Wir verwenden in unserem Editor *ein* **PrintDocument**-Objekt, das am Anfang der Klassendefinition erzeugt und der Instanzvariablen `printJob` zugewiesen wird:

```
class RitchisTexteditor : Form {  
    . . .  
    PrintDocument printJob = new PrintDocument();  
    PrintDialog printDialog = new PrintDialog();  
    . . .  
}
```

Wie Sie bereits wissen, verfügt es über ...

- ein „eingebautes“ **PrinterSettings**-Objekt, ansprechbar über die Eigenschaft **PrinterSettings**, das per Voreinstellung mit dem Standarddrucker des Benutzers verbunden ist.
- ein „eingebautes“ **PageSettings**-Objekt, ansprechbar über die Eigenschaft **DefaultPageSettings**, das mit dem eingebauten **PrinterSettings**-Objekt verbunden ist.

Außerdem verwenden wir *ein* **PrintDialog**-Objekt, das ebenfalls am Anfang der Klassendefinition erzeugt und der Instanzvariablen `printDialog` zugewiesen wird (siehe oben).

Die folgenden Hilfsvariablen auf Klassenebene

```
string tempText;  
int fromPage, toPage, actPage;
```

dienen zur Unterstützung der Druckausgabe:

- `tempText` enthält den zu druckenden Text.
- `fromPage` erhält die Nummer der ersten auszugebenden Seite.
- `toPage` erhält die Nummer der letzten auszugebenden Seite.
- `actPage` enthält während eines Druckvorgangs die Nummer der gerade in Bearbeitung befindlichen Seite.

Das per Instanzvariable

```
ToolStripMenuItem mitPrint;
```

ansprechbare Menüitem **Datei > Drucken** wird im Fensterkonstruktor angelegt und konfiguriert:

```
Bitmap bmp = new Bitmap("print.bmp");
bmp.MakeTransparent(bmp.GetPixel(0, 0));
mitPrint = new ToolStripMenuItem("&Drucken", bmp,
    new EventHandler(MitPrintOnClick));
mitPrint.ShortcutKeys = Keys.Control | Keys.P;
mitFile.DropDownItems.Add(new ToolStripSeparator());
mitFile.DropDownItems.Add(mitPrint);
```

Analog verfahren wir mit dem zugehörigen Symbolschalter:

```
ToolStripButton tsbPrint;

tsbPrint = new ToolStripButton();
tsbPrint.Enabled = false;
tsbPrint.ToolTipText = "Drucken";
toolStripStandard.Items.AddRange(
    new ToolStripItem[] {tsbNew, tsbOpen, tsbSave, tsbPrint, tsbFind});
tsbPrint.Image = bmp; // Bitmap übernommen vom korrespondierenden Menüitem
tsbPrint.Click += new System.EventHandler(MitPrintOnClick);
```

Die Objekte für den Druckauftrag und den Standarddialog **Drucken** werden folgendermaßen im Fensterkonstruktor auf ihren Einsatz vorbereitet:

- Der Druckdialog wird über seine **Document**-Eigenschaft mit dem Druckauftrag verknüpft, um das im Druckdialog zu modifizierende **PrinterSettings**-Objekt festzulegen:

```
printDialog.Document = printJob;
```
- Der Druckdialog wird angewiesen, die Auswahl eines Seitenbereichs oder der aktuellen Seite als Druckumfang anzubieten und mit erweiterter Funktionalität aufzutreten:

```
printDialog.AllowSomePages = true;
printDialog.AllowCurrentPage = true;
printDialog.UseEXDialog = true;
```

Weil unser Editor am Bildschirm keine Seitenumbrüche und keine Seitenzahlen anzeigt, ist die Auswahl eines passenden Seitenbereichs für die Anwender nur umständlich zu realisieren (z.B. über eine vorherige Druckvorschau, vgl. Abschnitt 21.3.1).

- Es werden Behandlungsmethoden bei den Ereignissen **BeginPrint** und **PrintPage** des Druckauftrags registriert:

```
printJob.BeginPrint += new PrintEventHandler(PrintJobOnBeginPrint);
printJob.PrintPage += new PrintPageEventHandler(PrintJobOnPrintPage);
```

Die beiden Methoden werden später in eigenen Abschnitten vorgestellt.

In der Behandlungsmethode zu den **Click**-Ereignissen des Menüitems **Datei > Drucken** und des zugehörigen Symbolleistenschalters setzen wir die Druckdialog-Eigenschaft **AllowSelection** auf **true**, wenn beim Aufruf ein Textbereich markiert ist.

```
protected void MitPrintOnClick(object sender, EventArgs e) {
    printDialog.AllowSelection = (editor.SelectionLength > 0);
    if (sender == mitPrint) {
        if (printDialog.ShowDialog() == DialogResult.OK) {
            printJob.Print();
        }
    } else
        printJob.Print();
}
```

Bei einer Druckanforderung via Menüitem erhält der Benutzer per Standarddialog die Gelegenheit, die Druckeinstellungen (neu) festzulegen. Bei einer Anforderung via Symbolschalter werden die aktuellen Einstellungen verwendet.

In beiden Fällen wird die Ausgabe über die **Print()**-Methode des Druckjob-Objekts eingeleitet.

21.2.2 BeginPrint - Ereignisbehandlung

In der Behandlungsmethode zum **BeginPrint**-Ereignis des **PrintDocument**-Objekts findet eine Druckvorbereitung aufgrund der aktuellen, in der Regel vom Benutzer via Druckdialog im **PrinterSettings**-Objekt vorgenommenen Einstellungen statt:

```
protected void PrintJobOnBeginPrint(object sender, PrintEventArgs e) {
    switch (printJob.PrinterSettings.PrintRange) {
        case PrintRange.AllPages:
            tempText = editor.Text;
            fromPage = 1;
            toPage = Int32.MaxValue;
            break;
        case PrintRange.Selection:
            tempText = editor.SelectedText;
            fromPage = 1;
            toPage = Int32.MaxValue;
            break;
        case PrintRange.SomePages:
            tempText = editor.Text;
            fromPage = printJob.PrinterSettings.FromPage;
            toPage = printDialog.PrinterSettings.ToPage;
            break;
        case PrintRange.CurrentPage:
            tempText = editor.Text;
            fromPage = 1;
            toPage = Int32.MaxValue;
            break;
    }
    actPage = 1;
    if (fileName != null)
        printJob.DocumentName = Path.GetFileName(fileName);
    else
        printJob.DocumentName = Text;
}
```

Je nach Druckumfang werden die Hilfsvariablen tempText, fromPage, und toPage mit passenden Werten versorgt. Abschließend erhält der in Vorbereitung befindliche Druckauftrag noch einen Namen.

21.2.3 PrintPage - Ereignisbehandlung

Kernstück bei der Druckunterstützung eines Programms ist die Behandlungsmethode zum **PrintPage**-Ereignis des Druckauftragsobjektes.

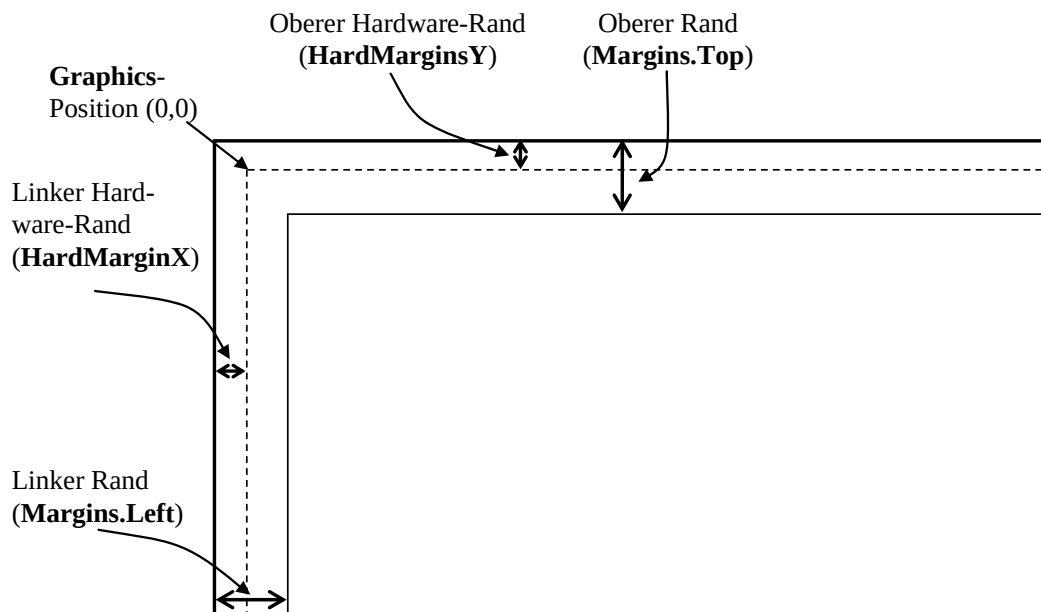
21.2.3.1 PrintPageEventArgs-Eigenschaften

Die **PrintPage**-Behandlungsmethode erhält wesentliche Informationen für ihre Tätigkeit über ein Aktualparameterobjekt vom Typ **PrintPageEventArgs**, das die folgenden **public**-Eigenschaften besitzt:

PrintPageEventArgs-Eigenschaft	Beschreibung
Graphics	Typ: Graphics Zugriff: get An dieses Objekt richten sich die Ausgabeaufträge. Seine PageUnit -Eigenschaft hat den voreingestellten Wert GraphicsUnit.Display , und weil es mit einem Drucker verbunden ist, resultiert die voreingestellte Maßeinheit 1/100 Zoll (vgl. Abschnitt 15.8.2). Als alternative PageUnit -Eigenschaft kommt z.B. GraphicsUnit.Millimeter in Frage.
Cancel	Typ: bool Zugriff: get, set Um einen Druckauftrag abubrechen, setzt ein PrintPage -Handler die Cancel -Eigenschaft auf true .
HasMorePages	Typ: bool Zugriff: get, set Voreinstellung: false Möchte der PrintPage -Handler nach Fertigstellung einer Seite zur Ausgabe einer weiteren Seite erneut aufgerufen werden, muss er die Eigenschaft HasMorePages auf true setzen.
PageSettings	Typ: PageSettings Zugriff: get Über diese Eigenschaft sind die Einstellungen der aktuellen Seite zu ermitteln (z.B. die Seitenausrichtung). Um diese Einstellungen verändern zu können (z.B. die Seitenausrichtung), muss man eine Behandlungsmethode beim PrintDocument -Ereignis QueryPageSettings registrieren (siehe Abschnitt 21.1.3.2).
PageBounds	Typ: Rectangle Zugriff: get Diese Eigenschaft informiert über die aktuelle Seitengröße unter Berücksichtigung der Seitenausrichtung (Maßeinheit: 1/100 Zoll). Eventuelle Hardwarebedingte Ränder des eingestellten Druckers sind <i>nicht</i> berücksichtigt.
MarginBounds	Typ: Rectangle Zugriff: get Diese Eigenschaft informiert über das nach Abzug der (in PageSettings.Margins definierten) Seitenränder verbleibende Rechteck (Maßeinheit: 1/100 Zoll).

21.2.3.2 Ausgaberechteck ermitteln

Unsere Ausgabeaufträge richten sich an das im **PrintPageEventArgs**-Objekt enthaltene **Graphics**-Objekt, welches per Voreinstellung auf den *bedruckbaren Papierbereich* eingestellt ist, so dass also die Ausgabeposition (0, 0) von der oberen linken Ecke des Papiers in horizontaler und vertikaler Richtung um den jeweiligen druckerbedingten Rand entfernt ist:



Über die **PrintDocument**-Eigenschaft **OriginAtMargins** kann man dafür sorgen, dass der **Graphics**-Ursprung von der oberen linken Ecke des Papiers in horizontaler bzw. vertikaler Richtung um den jeweiligen Seitenrand entfernt ist (siehe Abschnitt 21.1.3), so dass für Druckausgaben in der Randzone (z.B. Kopf- oder Fußzeilen) negative Koordinaten resultieren. Wir bleiben bei der Voreinstellung und müssen bei Positionen die druckerbedingten Ränder subtrahieren, um die vom Benutzer gewünschten Ränder einzuhalten.

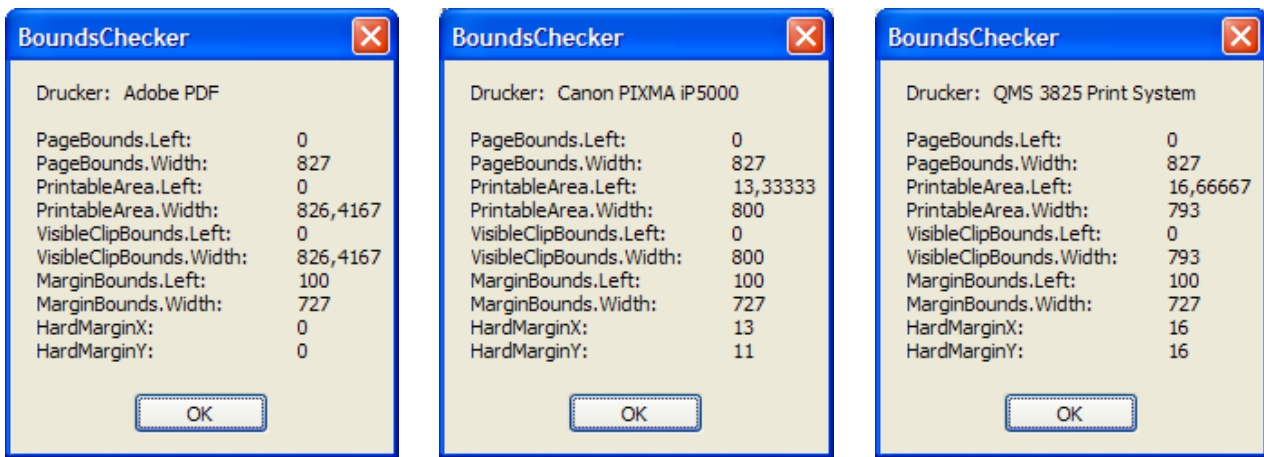
Wir kennen in der **PrintPage**-Behandlungsmethode:

- Seitengröße unter Berücksichtigung der Seitenausrichtung
Diese liefert die **PrintPageEventArgs**-Eigenschaft **PageBounds**, welche den **Bounds** – Wert des eingebundenen **PageSettings**-Objekts durchreicht.
- Position und Größe der *randbereinigten* Seite
Die **PrintPageEventArgs**-Eigenschaft **MarginBounds**, welche auf den Eigenschaften **Bounds** und **Margins** des eingebundenen **PageSettings**-Objekts basiert, liefert die Seitenposition und -größe unter Berücksichtigung der Ränder.
- Größe der bedruckbaren Seite
Die **VisibleClipBounds**-Eigenschaft des im **PrintPageEventArgs**-Objekt enthaltenen **Graphics**-Objekts zeigt ebenso wie die **PrintableArea**-Eigenschaft des eingebundenen **PageSettings**-Objekts auf eine **RectAngleF**-Instanz, welche die bedruckbare Seite beschreibt. Allerdings beziehen sich die **PrintableArea**-Angaben auf das Papier und sind folglich gegenüber den **VisibleClipBounds**-Angaben um die druckerbedingten Ränder verschoben (siehe unten).

In einer Übungsaufgabe (siehe Abschnitt 21.4) sollen Sie für einen Druckauftrag ausgeben:

- erste horizontale Koordinate und Breite der Seite
- erste horizontale Koordinate und Breite der bedruckbaren Seite
- erste horizontale Koordinate und Breite der randbereinigten Seite

Wie die folgenden Beispielausgaben zeigen, hängt die Größe der bedruckbaren Seite vom Ausgabegerät ab:



Während der Pseudodruckertreiber **Adobe PDF** (zum Erzeugen von PDF-Dateien) die gesamte Seite „bedruckt“, benötigen die beiden echten Drucker (**Canon PIXMA iP 5000** und **QMS 3825**) einen horizontalen Hardware-Rand von 27/100 bzw. 34/100 Zoll. Die voreingestellte Ausgabeposition (0, 0) des **Graphics**-Objekts, das unsere Ausgabeanweisungen ausführt, ist vom linken und vom oberen Rand des Papiers unterschiedlich weit entfernt, beim **QMS 3825 Print System** z.B. ca. 4 mm ($0,1666667 \cdot 25,4$).

Mit Hilfe der verfügbaren Informationen erzeugen wir für den Editor ein **RectangleF**-Objekt, das zum Begrenzen der Textausgabe per **Graphics**-Objekt auf einer Seite geeignet ist:

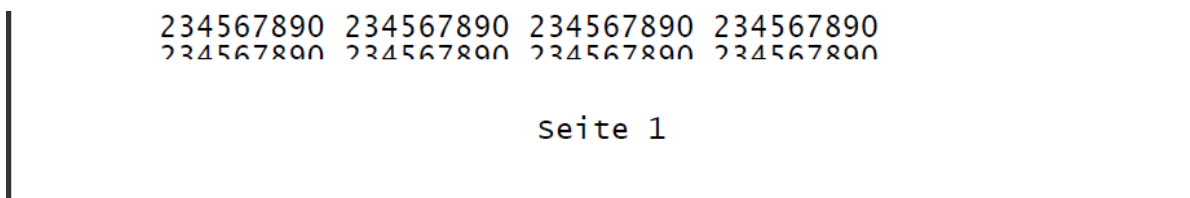
```
rectPage = new RectangleF(e.MarginBounds.Left - e.PageSettings.HardMarginX,
    e.MarginBounds.Top - e.PageSettings.HardMarginY,
    Math.Min(e.MarginBounds.Width, g.VisibleClipBounds.Width),
    Math.Min(e.MarginBounds.Height, g.VisibleClipBounds.Height));
```

Um die linke obere Ecke des Ausgaberechtecks korrekt in Bezug den **Graphics**-Ursprung (0, 0) zu positionieren, werden die gewünschten Seitenränder um die Hardware-Ränder gemindert. Die eingestellte Ausgabebreite bzw. -höhe (`e.MarginBounds.Width` bzw. `e.MarginBounds.Height`) wird übernommen, wenn sie nicht größer als die bedruckbare Breite bzw. Höhe des Ausgabegeräts ist (`g.VisibleClipBounds.Width` bzw. `g.VisibleClipBounds.Height`).

Nun wird die Höhe des Ausgaberechtecks so gekürzt,

```
lineSpacing = editor.Font.GetHeight(g);
completeLines = (int)(rectPage.Height / lineSpacing);
rectPage.Height = completeLines * lineSpacing;
```

dass am unteren Seitenrand keine unvollständigen Zeilen



erscheinen, was ansonsten zumindest bei Ausgabe mit automatischem Wortumbruch (siehe unten) passieren kann.

Weil der **GetHeight()**-Aufruf die aktuelle Maßeinheit des **Graphics**-Objektes verwendet (= 1/100 Zoll), bleiben alle Größenangaben konsistent.

Wenn im oberen bzw. unteren Randbereich der Seite genügend Platz ist, wird eine Kopfzeile mit dem Dateinamen und eine Fußzeile mit der Seitennummer geschrieben. Eine Diskussion des relativ langweiligen Quellcodes zur Berechnung der zugehörigen Ausgaberechtecke ersparen wir uns an dieser Stelle.

21.2.3.3 Textausgabe mit DrawString()

Bei der Textausgabe verlassen wir uns auf die **Graphics**-Methode **DrawString()**, die besonders kompetent in der Überladung mit Ausgaberechteck und **PrintFormat**-Objekt agiert:

- Beachtung der Ränder
- Automatischer Wortumbruch

Für jede zu bedruckende Seite ermitteln wir ein Ausgaberechteck nach obigen Überlegungen und lassen es von **DrawString()** unter Beachtung der Einstellungen in einem **StringFormat**-Objekt füllen:

```
g.DrawString(tempText, editor.Font, Brushes.Black, rectPage, sfText);
```

Weil generell der Pinsel **Brushes.Black** zum Einsatz kommt, ist z.B. beim Drucken eines vom Visual Studio farblich gestalteten und via Zwischenablage in unseren Editor übernommenen Quelltextes nicht mit einem farbigen Druckergebnis zu rechnen. Hier sollte noch einmal daran erinnert werden, dass wir mit dem GDI+ arbeiten, also die volle Kontrolle über und die volle Verantwortung für das Druckergebnis haben.

Über die **FormatFlags** des **StringFormat**-Objekts sorgen wir für eine Ausgabe mit bzw. ohne Wortumbruch in Abhängigkeit von der aktuellen **WordWrap**-Einstellung der **RichTextBox**-Komponente, welche der Benutzer via **Optionen**-Dialog beeinflussen kann. Über die **Trimming**-Eigenschaft des **StringFormat**-Objekts legen wir fest, wie am Ende einer Zeile bzw. am Ende eines Ausgaberechtecks der nicht mehr passende Text behandelt wird:

- Bei automatischem Zeilenumbruch endet der Text auf einer Seite mit dem letzten vollständig passenden Wort.
- Bei einer Druckausgabe *ohne* Zeilenumbruch endet jede Zeile mit dem letzten passenden Buchstaben, wobei ggf. drei Punkte für den abgeschnittenen Text stehen.

In der folgenden Passage der **PrintPage**-Behandlungsmethode werden die besprochenen Einstellungen vorgenommen:

```
StringFormat sfText = new StringFormat();
if (editor.WordWrap)
    sfText.Trimming = StringTrimming.Word;
else {
    sfText.FormatFlags |= StringFormatFlags.NoWrap;
    sfText.Trimming = StringTrimming.EllipsisCharacter;
}
```

Hat sich der Benutzer für einen Seitenbereich als Ausgabeumfang entschieden, muss erst bis zur Startseite vorgeblättert werden, wobei sich die **Graphics**-Methode **MeasureString()** als sehr nützlich erweist:⁷⁴

```
while (actPage < firstPage && tempText.Length > 0) {
    g.MeasureString(tempText, editor.Font, rectPage.Size, sfText,
        out nrChars, out nrLines);
    if (!editor.WordWrap)
        nrChars = NumberOfCharsInBlock(tempText, nrLines);
    tempText = tempText.Substring(nrChars);
    actPage++;
}
```

⁷⁴ Die in Abschnitt 15.7.4.2 diskutierten **MeasureString()**-Probleme bei der Breitenberechnung für kurze, aufeinander folgende und individuell formatierte Zeichenfolgen kommen hier *nicht* zum Tragen.

Wir erfahren über einen Ausgabeparameter, wie viele Zeichen der aktuellen `tempText`-Zeichenfolge im Ausgaberechteck Platz finden und streichen über die **String**-Methode **Substring()** entsprechend viele Zeichen am Anfang des Hilfstexts. Bei abgeschaltetem Zeilenumbruch liefern die **MeasureString()**-Ausgabeparameter jedoch zu wenig Informationen: Wir erfahren zwar exakt, wie viele Zeilen geschrieben würden, doch ist die Anzahl der darin enthaltenen Zeichen unbekannt, weil die abgeschnittenen Zeichen nicht mitgezählt werden. Abhilfe schafft die folgende, weitgehend von Petzold (2002, S. 906) übernommene Methode:

```
int NumberOfCharsInBlock(string text, int nrLines) {
    int last = 0;
    for (int i = 0; i < nrLines; i++) {
        last = text.IndexOf("\n", last) + 1;
        if (last == 0)
            return text.Length;
    }
    return last;
}
```

Beachten Sie:

- Die Methode wird nur bei abgeschaltetem Zeilenumbruch aufgerufen, so dass **MeasureString()** die Anzahl der per **NewLine**-Sequenz abgeschlossenen Ausgabezeilen ermittelt.
- Endet die letzte Ausgabezeile der Seite *nicht* mit der **NewLine**-Sequenz, passte der gesamte Drucktext auf die Seite.

Stellt sich heraus, dass kein Text mehr auszugeben ist, bricht **PrintPage**-Handler den Druckauftrag ab und gibt die Kontrolle zurück:

```
if (tempText.Length == 0) {
    e.Cancel = true;
    return;
}
```

Anderenfalls wird eine Textseite ausgegeben:

```
g.DrawString(tempText, editor.Font, Brushes.Black, rectPage, sfText);
```

Hat sich der Benutzer für die aktuelle Seite als Ausgabeumfang entschieden, wird zunächst so lange vorgeblättert, bis sich die aktuelle Position des Editors (definiert durch die Position der Einfügemarke oder das Ende einer vorhandenen Markierung) auf der nächsten Ausgabeseite befindet.

Wenn im oberen bzw. unteren Randbereich der Seite genügend Platz für eine Kopf- bzw. Fußzeile ist, werden diese jetzt geschrieben, z.B.:

```
if (rectPage.Top > lineSpacing) {
    StringFormat sfKF = new StringFormat();
    sfKF.Alignment = StringAlignment.Center;
    RectangleF rectHead = rectPage;
    rectHead.Height = lineSpacing;
    rectHead.Y = Math.Max((e.PageSettings.Margins.Top - lineSpacing) / 2
        - e.PageSettings.HardMarginY, 0);
    g.DrawString(printJob.DocumentName, editor.Font, Brushes.Black,
        rectHead, sfKF);
}
```

Nach Ausgabe einer Seite kommt zur Klärung der Frage, ob noch weiterer Text vorhanden ist, wiederum das Gespann aus **MeasureString()** und **NumberOfCharsInBlock()** zum Einsatz:

```

g.MeasureString(tempText, editor.Font, rectPage.Size, sfText,
                out nrChars, out nrLines);
if (!editor.WordWrap)
    nrChars = NumberOfCharsInBlock(tempText, nrLines);
tempText = tempText.Substring(nrChars);
actPage++;
if (tempText.Length > 0 && actPage <= toPage)
    e.HasMorePages = true;

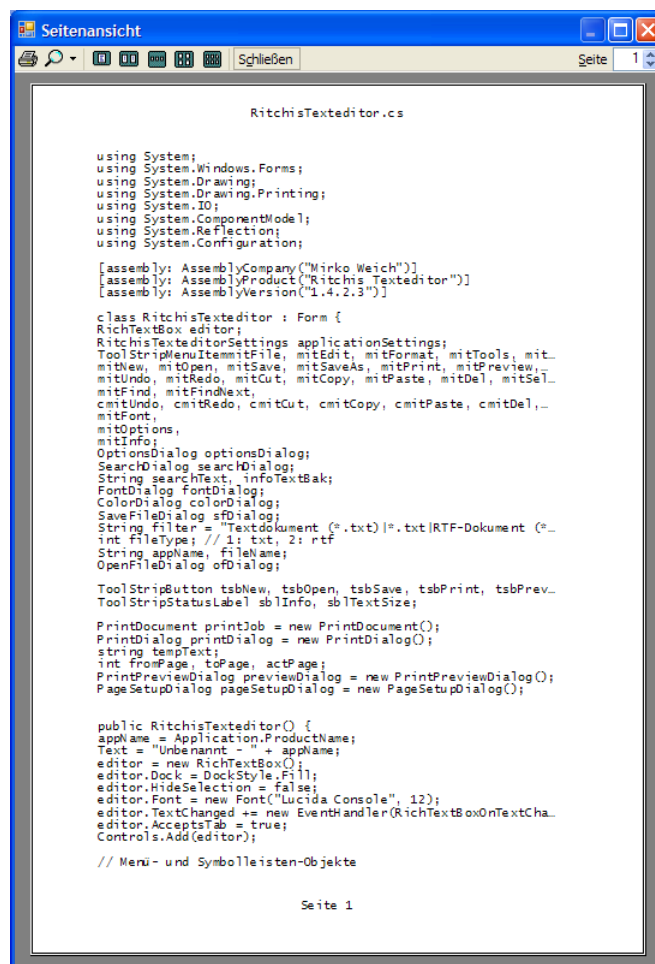
```

Indem der **PrintPage**-Handler die **HasMorePages**-Eigenschaft des übergebenen **PrintPageEventArgs**-Objekts auf **true** setzt, signalisiert er dem Druckauftrag, dass er zur Ausgabe einer weiteren Seite erneut aufgerufen werden möchte.

21.3 Weitere Standarddialoge zur Unterstützung der Druckausgabe

21.3.1 Seitenansicht

Ist erst einmal ein **PrintPage**-Handler erstellt, kann ohne großen Zusatzaufwand mit Hilfe der Klasse **PrintPreviewDialog** aus dem Namensraum **System.Windows.Forms** auch ein Seitenansicht-Standarddialog erstellt werden, der natürlich auch bei unserem Editor nicht fehlen soll:



Der **PrintPage**-Handler wird dazu genutzt, Bilder zu den einzelnen Ausgabeseiten zu erzeugen, die anschließend vom Seitenansicht-Standarddialog präsentiert werden.

Analog zum Vorgehen beim Standarddialog **Drucken** erzeugen wir in unserer Editorklasse ein per Instanzvariable ansprechbares **PrintPreviewDialog**-Memberobjekt

```
class RitchisTexteditor : Form {
    . . .
    PrintPreviewDialog previewDialog = new PrintPreviewDialog();
    . . .
}
```

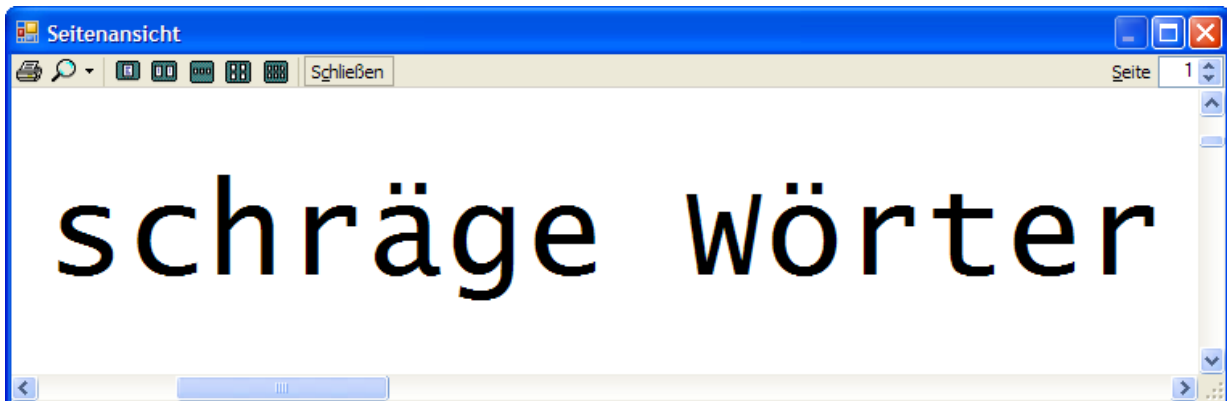
und stellen im Fensterkonstruktor die erforderliche Verknüpfung zum **PrintDocument**-Objekt her:

```
previewDialog.Document = printJob;
```

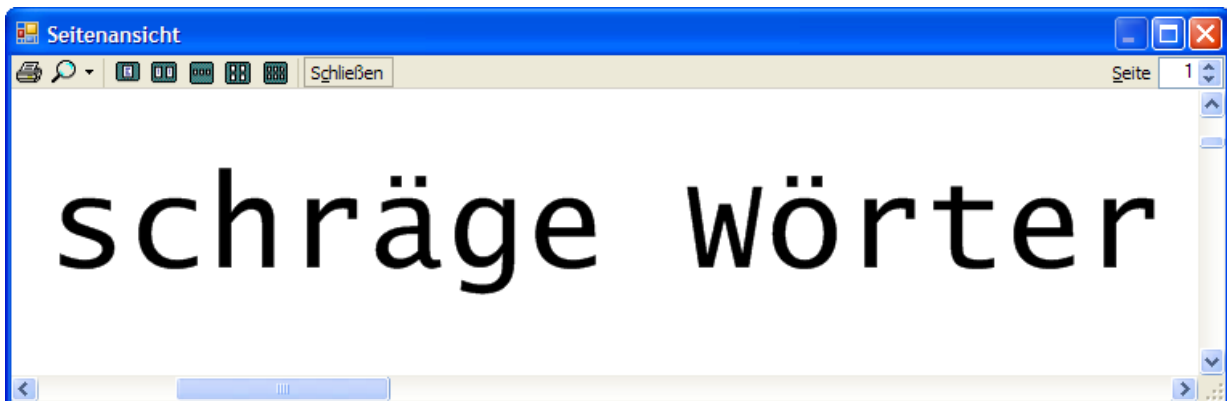
Über die **PrintPreviewDialog**-Eigenschaft **UseAntiAlias**

```
previewDialog.UseAntiAlias = true;
```

lässt sich die Kantenglättung (vgl. Abschnitt 15.6.1) für die Druckvorschau aktivieren, um Treppeneffekte bei schrägen Linien



zu reduzieren:



Um den Effekt bei der Textdarstellung nachzuweisen, wurde im Beispiel der maximale Seitenansicht-Vergrößerungsfaktor fünf eingestellt. Eine relevante Verzögerung beim Erstellen der Druckvorschau durch die Kantenglättung war *nicht* festzustellen.

Wir erstellen das neue Menüitem **Datei > Seitenansicht** sowie einen funktionsgleichen Symbolleistenschalter zum Starten der folgenden Ereignisbehandlungsmethode:

```
protected void MitPreviewOnClick(object sender, EventArgs e) {
    tempText = editor.Text;
    fromPage = 1;
    toPage = Int32.MaxValue;
    previewDialog.ShowDialog();
}
```

Unabhängig von den vorherigen Einstellungen im **PrintDialog** bzw. in den **PrinterSettings** sollen die Hilfsvariablen `tempText`, `fromPage`, `toPage` bei der Seitenansicht stets für eine vollständige Textausgabe sorgen.

Für die eigentliche Seitenvorschau ist ein Steuerelement aus der Klasse **PrintPreviewControl** verantwortlich. Man kann ein solches Objekt auch unabhängig vom Standarddialog einsetzen, um eine individuellere Seitenansicht zu realisieren. Z.B. kann die aktivierte Seitenansicht (wie in MS-WordPad) im Anwendungsfenster an Stelle der normalen Textanzeige erscheinen.

Es fällt übrigens auf, dass der Seitenaufbau durch ein **PrintPreviewDialog**-Objekt in Abhängigkeit vom zugeordneten Drucker recht lahm sein kann. Einige durch wiederholtes Messen stabil geschätzte Zeiten zum Erstellen der Druckvorschau für ein 20-seitiges Dokument:

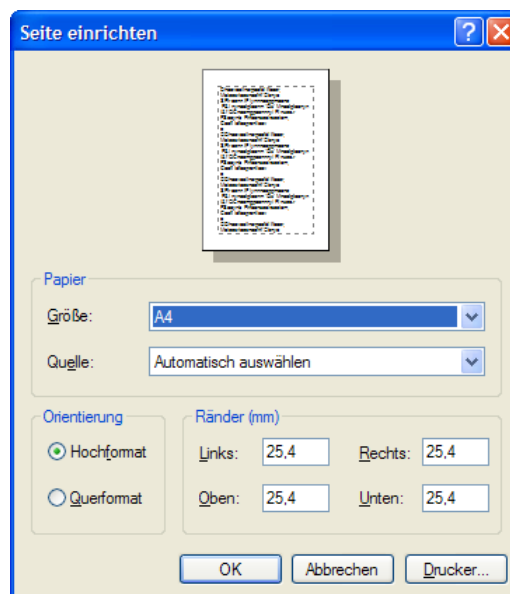
Drucker(treiber)	mittlere Zeit für den Aufbau der Druckvorschau
Adobe PDF	22 Sekunden
Canon PIXMA iP 5000	4 Sekunden
QMS 3825 Print System	4 Sekunden

Vielleicht sind Sie daran interessiert, was ein **PrintPreviewDialog**-Objekt hinter der Bühne beim Aufruf seiner **ShowDialog()**-Methode tut:

- Es setzt die **PrintController**-Eigenschaft des angeschlossenen Druckauftrags (Wert der **PrintPreviewDialog**-Eigenschaft **Document**) auf ein **PreviewPrintController**-Objekt. Dieses Objekt erzeugt Bilder und liefert die zugehörigen **Graphics**-Objekte an die **PrintPage**-Ereignisbehandlungsmethode, während der standardmäßige **PrintController** ein **Graphics**-Objekt für den eingestellten Drucker übergibt.
- Es ruft die **Print()**-Methode des Druckauftrags auf, was pro Seite zu einem Aufruf des **PrintPage**-Handlers führt.
- Es zeigt die fertigen Bilder in einem Formular mit etlichen Bedienelementen an.

21.3.2 Seiteneinrichtung

Zum normalen Bedienungskomfort eines Editors gehört auch der Standarddialog zur Seiteneinrichtung, der im .NET-Framework durch die Klasse **PageSetupDialog** aus dem Namensraum **System.Windows.Forms** realisiert wird, z.B.:



Analog zum Vorgehen bei den anderen Standarddialogen im Editorprojekt erzeugen wir *ein* per Instanzvariable ansprechbares **PageSetupDialog**-Objekt:

```
class RitchisTexteditor : Form {
    . . .
    PageSetupDialog pageSetupDialog = new PageSetupDialog();
    . . .
}
```

Wir verbinden es im Fensterkonstruktor mit unserem **PrintDocument**-Objekt und legen damit fest, welche **PageSettings**-Einstellungen durch den Dialog modifiziert werden sollen:

```
pageSetupDialog.Document = printJob;
```

Die Verbindung zu einem **PageSettings**-Objekt kann alternativ über die **PageSettings**-Eigenschaft des **PageSetupDialog**-Objekts hergestellt werden.

Über den per Voreinstellung verfügbaren Schalter **Drucker** im **PageSetupDialog** ist ein **PrintDialog** erreichbar. Folglich benötigt der **PageSetupDialog** ein **PrinterSettings**-Objekt, um es an den **PrintDialog** weiterreichen zu können. Ist ein **PrintDocument** zugewiesen, wird dessen **PrinterSettings**-Objekt verwendet. Alternativ kann man der **PageSetupDialog**-Eigenschaft **PrinterSettings** ein Objekt zuweisen. In unseren Editor vermeiden wir Aufwand und Inkonsistenzen mit folgender Strategie:

- Es wird jeweils genau *ein* Objekt aus den Klassen **PrintDocument**, **PrintDialog** und **PageSetupDialog** verwendet.
- Die beiden letztgenannten Objekte erfahren über das **PrintDocument**-Objekt, welche **Printer**- bzw. **PageSettings** sie modifizieren sollen.

Bei der Einsatzvorbereitung des **PageSetupDialog**-Objekts muss unbedingt seine **EnableMetric**-Eigenschaft auf den Wert `true` gesetzt werden:

```
pageSetupDialog.EnableMetric = true;
```

Ansonsten verhält sich der Standarddialog wenig sinnvoll:

- Beim Öffnen des Dialogs wird falsch von der Druckereinheit (1/100 Zoll) in Millimeter umgerechnet, so dass um den Faktor 2,54 verkleinerte Ränder in den Textfeldern landen.
- Weil beim Quittieren des Dialogs mit **OK** *kein* komplementärer Fehler passiert, werden die geminderten Ränder zum **PageSettings**-Objekt übertragen.

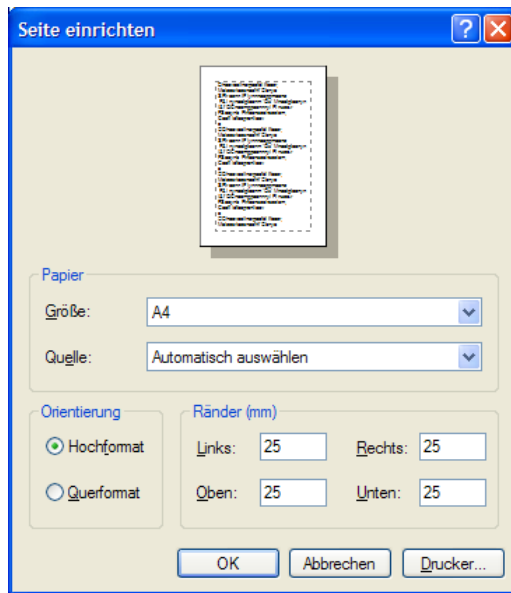
Wir erstellen im Editor das neue Menüitem **Datei > Seite einrichten** zum Starten der folgenden Ereignisbehandlungsmethode:

```
protected void MitPageSetupOnClick(object sender, EventArgs e) {
    pageSetupDialog.ShowDialog();
}
```

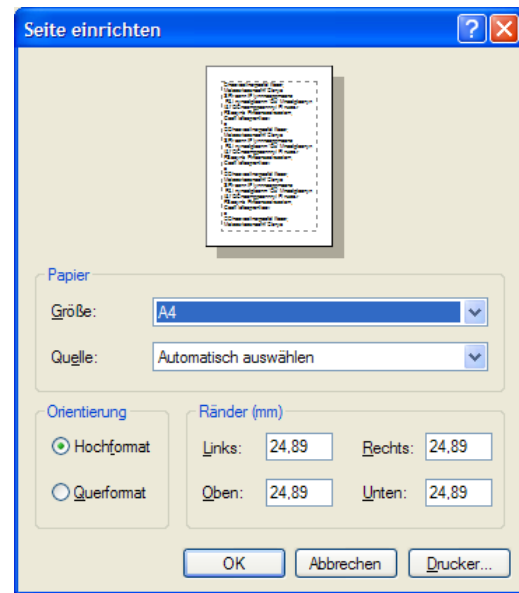
Es ist keine Auswertung des **ShowDialog()**-Rückgabewerts und keine Verarbeitung der erfragten Informationen erforderlich, weil der Seiteneinrichtungsdialog die mit **OK** quitierten Einstellungen selbständig auf das angebundene **PageSettings**- bzw. **PrinterSettings**-Objekt anwendet und auch auf das **Abbrechen** durch den Benutzer geeignet reagiert.

Die Randeinstellungen sind mit der **Margins**-Eigenschaft des im **PrintDocument**-Objekt integrierten **PageSettings**-Objekt verbunden (siehe Abschnitt 21.1.2). Weil die Randbreiten in der Maßeinheit 1/100 Zoll ganzzahlig gespeichert werden, sind exakte Angaben in *Millimeter* in der Regel *nicht* möglich. Sind z.B. 25 Millimeter als Randbreite gewünscht, resultieren als bestmögliche Näherung 98/100 Zoll (= 24,89 mm):

Vom Benutzer gewünscht



Resultat (nach erneutem Öffnen des Dialogs)

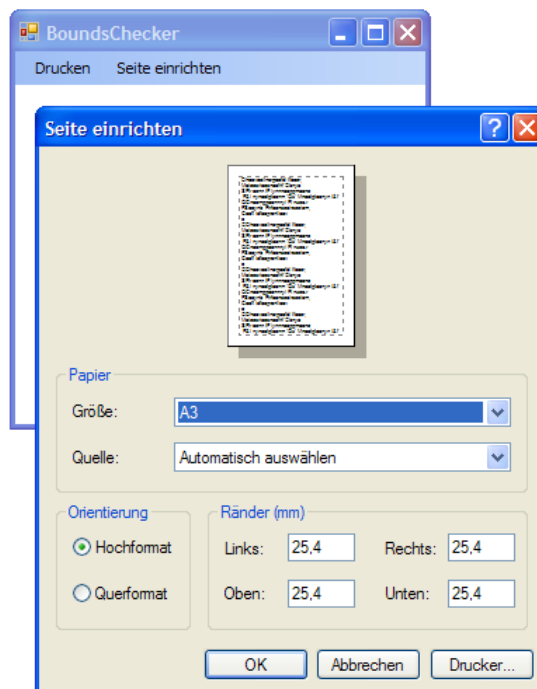


21.4 Übungsaufgaben zu Kapitel 21

1) Erstellen Sie ein Programm, das für die **PageSettings** zu einen Druckauftrag folgende Einstellungen ausgibt:

- erste horizontale Koordinate und Breite der Seite
- erste horizontale Koordinate und Breite der bedruckbaren Seite
- erste horizontale Koordinate und Breite der randbereinigten Seite

Das Ausgabegerät und die Seiteneinstellungen sollten über einen **PageSetupDialog** wählbar sein, z.B.:



Um faktische Ausdrücke zu vermeiden, sollten Sie im **PrintPage**-Handler die **Cancel**-Eigenschaft des **PrintEventArgs**-Objekts auf den Wert **true** setzen.

22 Zwischenablage, Ziehen und Ablegen

Mit Hilfe der Windows-Zwischenablage können Anwender auf bequeme Weise Daten zwischen Windows-Anwendungen austauschen oder auch innerhalb einer Anwendung übertragen. In Abhängigkeit von den beteiligten Programmen werden dabei unterschiedliche Datenformate unterstützt (z.B. Zeichenfolgen, Bitmaps, Metadateien). Markierte Daten werden per (Kontext)menü oder Tastaturbefehl (**Strg+C** bzw. **Strg+X**) in die Zwischenablage kopiert und beim Ausschneiden anschließend am alten Ort gelöscht. Viele Quellanwendungen legen die Daten sogar in mehreren Formaten in der Zwischenablage ab (z.B. einfachen und formatierten Text). Beim ebenfalls per (Kontext)menü oder Tastaturbefehl (**Strg+V**) anzufordernden Einfügen prüft die Zielanwendung, ob sie eines der verfügbaren Formate verarbeiten kann.

Neben der systemweit verfügbaren (von allen Programmen nutzbaren) Windows-Zwischenablage werden gelegentlich auch anwendungsinterne Zwischenspeicher realisiert, mit denen wir uns aber *nicht* beschäftigen werden.

In der FCL-Klasse **TextBoxBase** ist eine Zwischenablagen-Unterstützung bereits enthalten, so dass man sich beim Einsatz entsprechender Steuerelemente auf die Nutzung der vorhandenen Methoden **Cut()**, **Copy()** und **Paste()** beschränken kann, was wir bei unserem **RichTextBox**-basierten Editor z.B. in den **Click**-Behandlungsmethoden zu den Menüitems **Bearbeiten > Kopieren** und **Bearbeiten > Einfügen** auch getan haben:

```
protected void MitCopyOnClick(object sender, EventArgs e) {  
    if (editor.SelectionLength > 0) {  
        editor.Copy();  
    }  
}  
  
protected void MitPasteOnClick(object sender, EventArgs e) {  
    editor.Paste();  
}
```

Wir werden nun aber aus didaktischen Gründen im Editorprojekt vorübergehend mit den zugrunde liegenden und allgemein verwendbaren FCL-Klassen für den Datenaustausch via Windows-Zwischenablage arbeiten.

Ein solcher Datenaustausch kann zwischen *verschiedenen* Anwendungen, aber auch *innerhalb* einer Anwendung stattfinden. Speziell im zuletzt genannten Fall können auch sehr spezielle Datenformate transportiert werden, z.B. Objekte aus selbst definierten Klassen. Die Windows-Zwischenablage ist also keinesfalls auf den Transfer von Texten und Bildern beschränkt, und bei der Nutzung des vollen Funktionsumfangs kann man sich nicht auf die vorgefertigten Methoden von FCL - Steuerelementen beschränken.

Im aktuellen Abschnitt wird auch die Unterstützung für das Ziehen und Ablegen von Daten per Maus (engl.: *Drag & Drop*) behandelt, wobei teilweise dieselben Klassen wie beim Zwischenablagentransfer zum Einsatz kommen.

22.1 Verbesserte RTF-Unterstützung im Editorprojekt

Wir wollen die im zugrunde liegenden **RichTextBox**-Steuerelement enthaltene RTF-Kompetenz besser ausnutzen und die Wahl einer Schriftart, -auszeichnung und/oder -farbe für die *markierte Textpassage* erlauben. Dazu beschränken wir die Auswirkungen des Schriftartenstandarddialogs (zu öffnen mit dem Menübefehl **Format > Schriftart**) mit Hilfe der **RichTextBox**-Eigenschaften **SelectionFont** und **SelectionColor** auf die aktuelle Auswahl:

```
protected void MitFontOnClick(object sender, EventArgs e) {
    if (fontDialog.ShowDialog() == DialogResult.OK) {
        editor.SelectionFont = new Font(fontDialog.Font.FontFamily,
            (float)Math.Round(fontDialog.Font.SizeInPoints), fontDialog.Font.Style);
        editor.SelectionColor = fontDialog.Color;
    }
}

protected void FontDialogOnApply(object sender, EventArgs e) {
    editor.SelectionFont = new Font(fontDialog.Font.FontFamily,
        (float)Math.Round(fontDialog.Font.SizeInPoints), fontDialog.Font.Style);
    editor.SelectionColor = fontDialog.Color;
}
```

Über das Motiv für den ungewöhnlichen Aufwand bei der Entgegennahme des **Font**-Objekts vom Schriftarten-Standarddialog informiert Abschnitt 15.7.2. Später sollte der Bedienkomfort des Editors durch Symbolleistenkomponenten (vgl. Abschnitt 20.1) noch gesteigert werden, z.B. durch

- **ToolStripButton**-Schalter für die Schriftattribute **fett** und *kursiv*
- **ToolStripComboBox**-Elemente für die Schriftart und den Schriftgrad

Weil unser Editor *formatierten* Text nun selbst erzeugen kann, eignet er sich besser zur Demonstration der Multiformat-Zwischenablage durch die Ablage von einfachem *und* formatiertem Text (siehe Abschnitt 22.4.2).

Beachten Sie bitte, dass die in Abschnitt 21 realisierte Druckfunktion mit der Standardschrift der **RichTextbox**-Komponente arbeitet:

```
g.DrawString(tempText, editor.Font, Brushes.Black, rectPage, sfText);
```

Die Schriftspezifika einzelner Textpassagen bleiben also ohne Einfluss auf das Drucken, werden jedoch beim Speichern im RTF-Format unterstützt.

22.2 Windows-Zwischenablagenformate

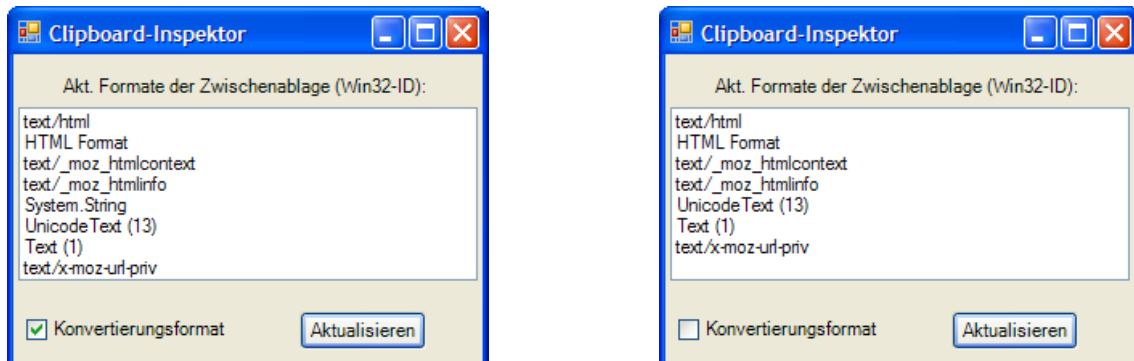
Damit ein Datenaustausch via Windows-Zwischenanlage gelingt, müssen die verwendeten Datenformate und deren Bezeichnungen bei allen beteiligten Programmen bekannt sein. Grundsätzlich kann man zwischen **allgemein bekannten** und **benutzerdefinierten** Formaten unterscheiden, wobei die Allgemeingültigkeit aber auf eine Software-Technologie (z.B. Win32, MFC, .NET) eingeschränkt sein kann. Das .NET - Framework kennt 21 allgemeine Formate, die bequem über statische und öffentliche **String**-Felder der Klasse **DataFormats** ansprechbar sind. Von diesen 21 Formaten sind 16 im Win32-API über eine **Clipboard ID** bekannt, was den Datenaustausch zwischen verschiedenen Software-Technologien erleichtern sollte. In der folgenden Tabelle finden Sie die im .NET-Framework allgemein bekannten Formate samt Clipboard ID (falls vorhanden):

Name	Win32-ID	Bedeutung
Text	1	Text in 8-Bit-ANSI-Kodierung
Bitmap	2	Geräteabhängiges Bitmap-Format
MetafilePict	3	Traditionelles Metafile-Format
SymbolicLink	4	Dateisystemeintrag, der auf eine Datei in einem anderen Verzeichnis verweist
Dif	5	DIF-Format für Tabellenkalkulationsdaten (<i>Data Interchange Format</i>)
Tiff	6	TIFF-Bitmapformat (<i>Tagged Image File Format</i>)
OemText	7	Text in 8-Bit-ASCII-Kodierung
Dib	8	Geräteunabhängiges Bitmap-Format (<i>Device Independent Bitmap</i>)

Name	Win32-ID	Bedeutung
Palette	9	Farbpalette zur gemeinsamen Verwendung mit geräteunabhängigen Bitmaps mit 256 Farben
PenData	10	Format für Stift-Eingabegeräte
Riff	11	RIFF-Audioformat
WaveAudio	12	Wave-Audioformat
UnicodeText	13	Text in Unicode-Kodierung
EnhancedMetafile	14	erweitertes Windows - Metafile-Format
FileDrop	15	Liste von Dateien, z.B. per Explorer mit Strg+C angelegt
Locale	16	Windows-Kulturformat für Textdaten in der Zwischenablage
CommaSeparatedValue		Austauschformat für Tabellenkalkulations- und Datenbankprogramme
Html		HTML-Format
Rtf		RTF-Format
Serializable		Beliebiges WinForms-Objekt
StringFormat		Ergebnis bei Ablage von String -Objekten durch .NET - Anwendungen

22.3 Ein Zwischenablagen-Inspektor

Wir erstellen als technisches Hilfsmittel ein Programm, das die aktuell in der Zwischenablage verfügbaren Formate anzeigt, wobei optional auch die automatisch daraus generierbaren .NET - Typen erscheinen. Nachdem der WWW-Browser Firefox 3.5 eine HTML-Passage in die Zwischenablage geschrieben hat, zeigt der Inspektor z.B. folgende Formate an:



Vom Quellcode des Zwischenablagen-Inspektors ist momentan nur die **Click**-Behandlungsmethode zum Befehlsschalter von Interesse:

```
class ClipboardInspector : Form {
    ListBox libClipboard;
    CheckBox cbAutoConv;
    . . .
    protected void AktualisierenOnClick(object sender, EventArgs ea) {
        libClipboard.Items.Clear();
        DataFormats.Format clipFormat;
        int id;
        String[] sF = Clipboard.GetDataObject().GetFormats(cbAutoConv.Checked);
```

```

    for (int i = 0; i < sF.Length; i++) {
        clipFormat = DataFormats.GetFormat(sF[i]);
        id = clipFormat.Id;
        if (1 <= id && id <= 16)
            libClipboard.Items.Add(clipFormat.Name + " (" + id + ")");
        else
            libClipboard.Items.Add(clipFormat.Name);
    }
}
}
}

```

Den vollständigen Quellcode des Programms finden Sie im Ordner

...\BspUeb\Zwischenablage, Ziehen und Ablegen\ClipboardInspector

Weil der **Clipboard**-Inspektor möglichst früh als Hilfsmittel zur Verfügung stehen soll, müssen wir bei seiner Implementierung dem Stoff etwas vorgreifen und die statische Methode **GetDataObject()** der Klasse **Clipboard** verwenden, die eigentlich erst in Abschnitt 22.5.2 zu behandeln ist. Sie liefert ein Objekt, das die Zwischenablagendaten in allen verfügbaren Formaten enthält und die Schnittstelle **IDataObject** erfüllt. Folglich beherrscht das Objekt die folgenden Überladungen der Methode **GetFormats()**:

```

public String[] GetFormats()
public String[] GetFormats(bool mitKonvertierungen)

```

Sie liefern einen **String**-Array mit den Namen der vorhandenen Formate, z.B.:

```
string[] sF = Clipboard.GetDataObject().GetFormats(cbAutoConv.Checked);
```

Mit dem booleschen Parameter in der zweiten Überladung entscheidet man darüber, ob auch die möglichen Konvertierungen von Zwischenablageinhalten in FCL - Typen angezeigt werden sollen. Bei der parameterfreien ersten Überladung ist diese Konvertierung aktiviert. Ein Vergleich der obigen Bildschirmfotos zeigt, dass beim Abschalten der Konvertierung das Format **System.String** verschwindet.

Mit der statischen Methode **GetFormat()** der Klasse **DataFormats** erhält man zu einem Formatnamen oder zu einer Format-ID ein Objekt der internen Klasse **DataFormats.Format**:

```

public static Dataformats.Format GetFormat(string formatName)
public static Dataformats.Format GetFormat(int formatID)

```

Dieses Objekt hält in seinen Eigenschaften **Name** und **ID** beide Identifikationsmöglichkeiten bereit. Bei den Clipboard IDs beschränken wir uns auf die in allen relevanten Windows-Versionen definierten Werte von 1 bis 16.

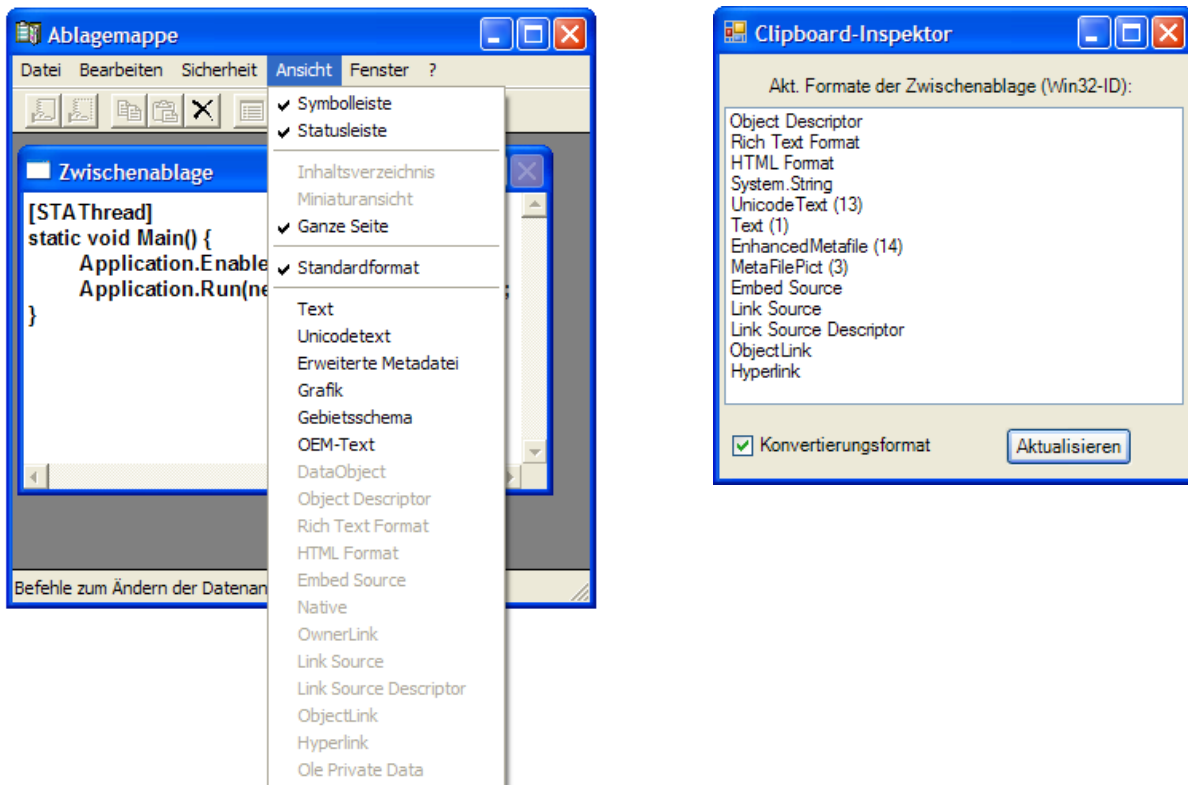
Damit die in Kapitel 22 vorgestellten Methoden zum Zwischenablagenzugriff und zum Datenaustausch per Drag & Drop keine Ausnahmen werfen, muss die **Main()**-Methode das Attribut **STAThread** erhalten (vgl. Abschnitt 11.1):

```

[STAThread]
static void Main() {
    Application.EnableVisualStyles();
    Application.Run(new ClipboardInspector());
}

```

Zur Inspektion der Zwischenablage kann auch das Windows-Werkzeug **clipbrd.exe** dienen, das man am einfachsten über das Windows-Startmenü auf die Beine bringt. Es liefert eine Liste der vorhandenen Zwischenablagenformate (via Ansichtsmenü) und zu manchen Formaten auch eine Inhaltsanzeige. Im folgenden Beispiel ist das Verhalten der beiden Inspektoren bei identischem Zwischenablageninhalt zu sehen:



Unser Inspektor informiert auch über .NET-Datentypen, die durch automatische Konvertierung erstellt werden können.

22.4 Daten in die Zwischenablage schreiben

Die Klasse **Clipboard** im Namensraum **Systems.Windows.Forms** bietet diverse statische Methoden zur Ablage von Daten in der Windows-Zwischenablage und kann mit ihrer Methode **Clear()** auch den aktuellen Zwischenablageninhalt löschen:

```
Clipboard.Clear();
```

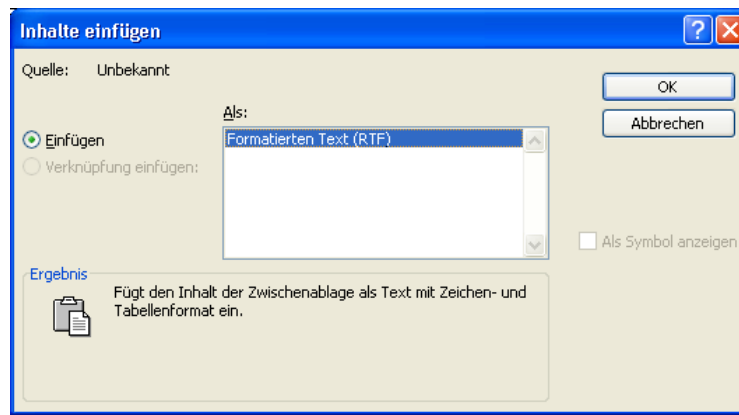
22.4.1 Bequeme Clipboard-Methoden für Routineaufgaben

In diesem Abschnitt werden etliche mit .NET 2.0 eingeführte **Clipboard**-Methoden vorgestellt, die eine bequeme Datenablage in der Zwischenablage ermöglichen und die meisten Routineaufgaben abdecken.

Die folgende **Click**-Behandlungsmethode **MitCopyOnClick()** zum Menüitem **Bearbeiten > Kopieren** unseres Editors verwendet (wie zu Beginn des Kapitels angekündigt) an Stelle der **RichTextBox**-Methode **Copy()** die statische **Clipboard**-Methode **SetText()**-Methode, um den markierten Text des **RichTextBox**-Steuerelements (inklusive Formatierungen) in die Zwischenablage zu kopieren:

```
protected void MitCopyOnClick(object sender, EventArgs e) {
    if (editor.SelectionLength > 0)
        Clipboard.SetText(editor.SelectedRtf, TextDataFormat.Rtf);
}
```

Nach Ausführung der Methode erkennt MS-Word das Zwischenablagenformat korrekt:



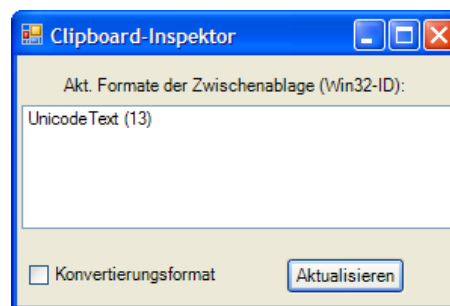
In der Klasse **Clipboard** sind die folgenden formatspezifischen Methoden zum Erstellen von Zwischenablagendaten verfügbar:

- **public static void SetText(String text)**

Je nach Windows-Version entsteht das Zwischenablageformat **Unicode Text** (ab Windows 2000) oder **Text**. Wie mit unserem **Clipboard**-Inspektor (siehe Abschnitt 22.3) leicht geprüft werden kann, führt unter Windows XP die folgende Anweisung

```
Clipboard.SetText("Text");
```

zum Zwischenablageformat **Unicode Text**:



- **public static void SetText(String text, TextDataFormat format)**

Mit dem zweiten Parameter vom Enumerationstyp **TextDataFormat** deklariert man das zu erzeugende Zwischenablageformat. Es sind folgende Werte erlaubt:

- **CommaSeparatedValue**
- **Html**
- **Rtf**
- **Text**
- **UnicodeText**

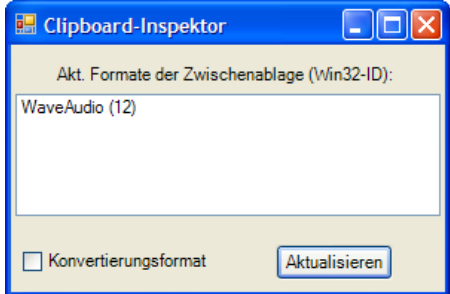
- **public static void SetImage(Image bitmap)**

Im der Zwischenablage entstehen die Formate **Bitmap** und **System.Drawing.Bitmap**:

Quellcode-Fragment	Clipboard-Inspektor
<pre>Bitmap image = new Bitmap("eins.bmp"); Clipboard.SetImage(image);</pre>	

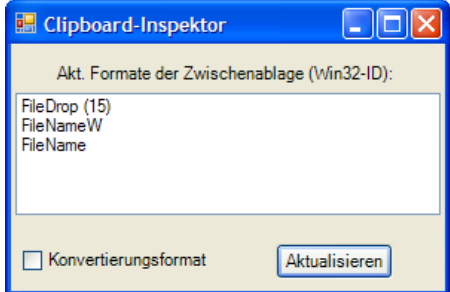
- **public static void SetAudio(Stream audioStream)**
public static void SetAudio(byte[] audioBytes)

Im der Zwischenablage entsteht das Format **WaveAudio**:

Quellcode-Fragment	Clipboard-Inspektor
<pre>byte[] b = new byte[] {1, 2, 3}; Clipboard.SetAudio(b);</pre>	

- **public static void SetFileDropList(StringCollection pfadnamen)**

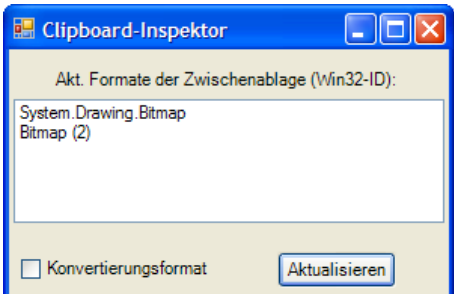
In der Zwischenablage landet u.a. eine Liste von Dateinamen (Format **FileDrop**):

Quellcode-Fragment	Clipboard-Inspektor
<pre>StringCollection sc = new StringCollection(); sc.Add("Eins"); sc.Add("Zwei"); Clipboard.SetFileDropList(sc);</pre>	

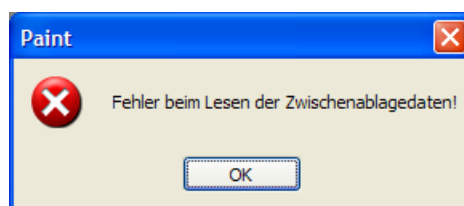
Etwas allgemeiner definiert, aber trotzdem bequem nutzbar ist die **Clipboard**-Methode **SetData()**:

public static void SetData(String format, Object daten)

Mit dem ersten Parameter legt man das gewünschte Zwischenablageformat fest, wobei man die Formatnamen über die statischen Felder der Klasse **DataFormats** angeben sollte (siehe Abschnitt 22.2). Derart schamlose Lügen

Quellcode-Fragment	Clipboard-Inspektor
<pre>Clipboard.SetData(DataFormats.Bitmap, 13);</pre>	

können unvorsichtige Zielprogramme bei der Übernahme von Zwischenablageninhalten in Schwierigkeiten bringen:



Für die bisher vorgestellten, mit .NET 2.0 eingeführten **Clipboard**-Methoden gilt:

- In der Zwischenablage entsteht eine *Kopie* der Daten. Gleich wird eine Möglichkeit behandelt, Objektreferenzen in der Zwischenablage abzulegen.
- Nach dem Programmende stehen die Zwischenablagendaten anderen Programmen weiter zur Verfügung. Dies ist bei der anschließend behandelten Methode **SetDataObject()** *nicht* unbedingt der Fall.

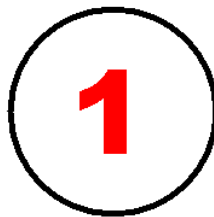
22.4.2 Die Clipboard-Methode SetDataObject() für spezielle Aufgaben

Die seit .NET 1.0 verfügbare **Clipboard**-Methode **SetDataObject()** überträgt Instanzen von Werttypen sowie beliebige Objekte aus serialisierbaren Klassen (vgl. Abschnitt 12.2.3) in die Zwischenablage und bietet einige spezielle Optionen:

- Ablage per Referenz
- befristete Ablage (bis zum Ende des Quellprogramms)
- Simultane Ablage in verschiedenen Formaten
- Ablage in selbstdefinierten Formaten

Es sind zwei Überladungen vorhanden:

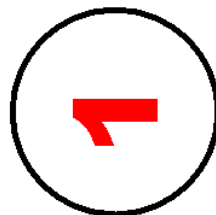
- **public static void SetDataObject(Object data)**
Instanzen von Werttypen und **String**-Objekte werden in die Zwischenablage *kopiert*. Von Objekten landet dort eine **Referenz**, so dass beim späteren Zwischenablagenzugriff der aktuelle Objektzustand geliefert wird. Das folgende Objekt vom Typ **Bitmap**



wird per Referenz in die Zwischenablage übertragen und anschließend im Quellprogramm rotiert:

```
Bitmap image = new Bitmap("eins.bmp");
Clipboard.SetDataObject(image);
image.RotateFlip(RotateFlipType.Rotate90FlipXY);
```

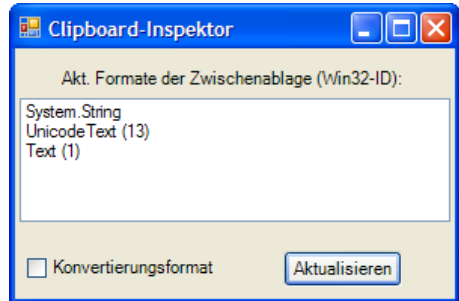
Ein Konsument bezieht folglich via Zwischenablage folgendes Bild:



Beim Beenden des Programms werden die Daten (Objekt-Referenz oder Wert- bzw. **String**-Kopie) aus der Zwischenablage entfernt. Unter Windows XP ist allerdings zu beobachten, dass erfolgreich abgerufene Bitmap-Daten *nach* Beenden des Quellprogramms weiter verfügbar sind.

- **public static void SetDataObject(Object data, bool kopie)**
Mit dem Wert **true** für den booleschen Parameter wird angeordnet, dass der abgelegte Inhalt nach Programmende anderen Anwendungen weiterhin zur Verfügung stehen soll. Außerdem entsteht bei der Übertragung in die Zwischenablage stets eine *Kopie*, d.h. beim späteren Zwischenablagenzugriff wird nicht der aktuelle, sondern der abgelegte Zustand eines Objekts geliefert. Beim Wert **false** für den booleschen Parameter resultiert das eben für die erste **SetDataObject()**-Überladung beschriebene Verhalten.

Neben der Ablage per Referenz und der befristeten Ablage (bis zum Programmende) hat die Methode **SetDataObject()** vor allem die Möglichkeit zur **simultanen Ablage mehrerer Formate** zu bieten. Diese Kompetenz wird spontan dazu verwendet, Daten mit einem .NET - Typ zusätzlich in kompatiblen, für andere Software-Technologien verdaulichen Formaten abzulegen. Wenn Sie z.B. ein **String**-Objekt per **SetDataObject()** in die Zwischenablage befördern, dann landen dort Daten in den Formaten **System.String**, **Text** und **UnicodeText**:

Quellcode-Fragment	Clipboard-Inspektor
<pre>Clipboard.SetDataObject("Text");</pre>	

In Abschnitt 22.4.1 wurde gezeigt, dass der folgende **SetText()**-Aufruf RTF-Daten in die Zwischenablage schreibt:

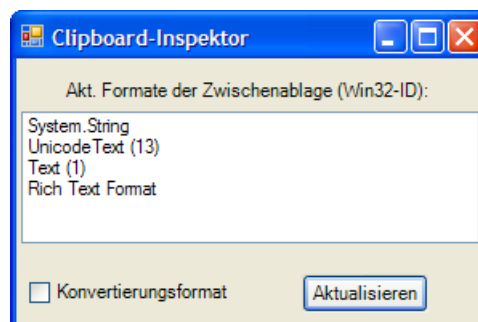
```
Clipboard.SetText(editor.SelectedRtf, TextDataFormat.Rtf);
```

Um sowohl RTF-Text als auch gewöhnlichen Text in die Zwischenablage zu übertragen und durch geeignete Formatdeklaration nutzbar zu machen, benötigen wir die Methode **SetDataObject()** und ein Objekt der Klasse **DataObject**. Dieses Objekt dient als Container für Zwischenablagendaten in mehreren Formaten und beherrscht zur Aufnahme der Daten dieselben **Set**-Methoden wie die Klasse **Clipboard** (siehe Abschnitt 22.4.1).

Ist eine **DataObject**-Instanz durch mehrere **Set**-Aufrufe mit allen benötigten Zwischenablagendaten bzw. -formaten ausgestattet, wird sie per **SetDataObject()**-Aufruf in die Zwischenablage befördert, z.B.:

```
protected void MitCopyOnClick(object sender, EventArgs e) {
    if (editor.SelectionLength > 0) {
        DataObject dataObject = new DataObject();
        dataObject.SetData(DataFormats.Rtf, editor.SelectedRtf);
        dataObject.SetData(DataFormats.Text, editor.SelectedText);
        Clipboard.SetDataObject(dataObject, true);
    }
}
```

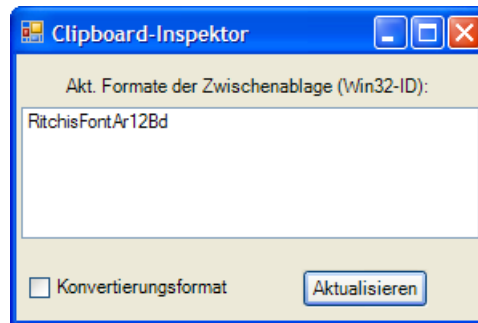
Zur Formatbezeichnung verwenden wir die in Abschnitt 22.2 beschriebenen Felder der Klasse **DataFormats**. Unser Inspektor aus Abschnitt 22.2 zeigt, dass nun alle gewünschten Formate in der Zwischenablage landen:



Mit Hilfe der Klasse **DataObject** und ihrer Methode **SetData()** gelingt auch die Ablage mit **individueller Formatbezeichnung**, z.B.:

```
Font fontAr12Bd = new Font("Arial", 12, FontStyle.Bold);
DataObject dataObject = new DataObject();
dataObject.SetData("RitchisFontAr12Bd", fontAr12Bd);
Clipboard.SetDataObject(dataObject);
```

Wie unser Inspektor zeigt, ist das Zwischenablagenformat nun für alle Eingeweihten klar erkennbar:



Beim Aufruf der Methode **SetDataObject()** kommt übrigens immer ein Objekt der Klasse **DataObject** ist Spiel. Übergibt man ein Objekt aus einer anderen Klasse, wird dieses automatisch in ein **DataObject** verpackt.

22.5 Daten aus der Zwischenablage abrufen

Auch für das Abrufen von Zwischenanlagedaten ist die Klasse **Clipboard** zuständig.

22.5.1 Bequeme Clipboard-Methoden für Routineaufgaben

In diesem Abschnitt werden etliche mit .NET 2.0 eingeführte **Clipboard**-Methoden vorgestellt, die einen bequemen Datenimport aus der Zwischenablage ermöglichen und die meisten Routineaufgaben abdecken.

Die folgende Variante der **Click**-Behandlungsmethode **MitPasteOnClick()** zum Menüitem **Bearbeiten > Einfügen** unseres Editors überprüft mit **ContainsText()**-Aufrufen, ob Textdaten im RTF-Format oder im einfachen Textformat (ANSI- oder Unicode-kodiert) vorhanden sind:

```
protected void MitPasteOnClick(object sender, EventArgs e) {
    if (Clipboard.ContainsText(TextDataFormat.Rtf))
        editor.SelectedRtf = Clipboard.GetText(TextDataFormat.Rtf);
    else
        if (Clipboard.ContainsText())
            editor.SelectedText = Clipboard.GetText();
}
```

Bei positivem Prüfergebnis wird der (RTF-)Text mit der passenden **GetText()**-Überladung aus der Zwischenablage bezogen, um den markierten (RTF-)Text des Editors damit zu ersetzen.

In der Klasse **Clipboard** sind die folgenden formatspezifischen Methoden zum Importieren von Zwischenablagendaten verfügbar, wobei mit einer korrespondierenden **Contains()**-Methode die Existenz von Daten im gewünschten Format festgestellt werden kann:

- **public static bool ContainsText()**
public static String GetText()
ContainsText() liefert den Wert **true** zurück, wenn Textdaten in der Zwischenablage vorhanden sind, ansonsten **false**. Ab Windows 2000 wird das Format **UnicodeText** erwartet, ansonsten das Format **Text**. Vorhandener Text lässt sich mit **GetText()** in ein **String**-Objekt befördern. Ist kein geeignetes Zwischenablagenformat vorhanden (**ContainsText()**-Rückgabe **false**), liefert **GetText()** eine leere Zeichenfolge (das Referenzziel der statischen **String**-Eigenschaft **Empty**).

- **public static bool ContainsText(TextDataFormat format)**
public static String GetText(TextDataFormat format)

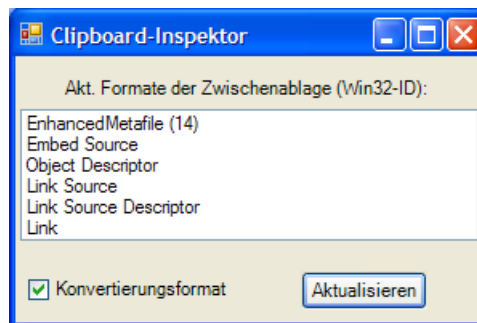
ContainsText() liefert den Wert **true** zurück, wenn Textdaten mit dem gewünschten Format in der Zwischenablage vorhanden sind, ansonsten **false**. Vorhandener Text im gewünschten Format lässt sich mit **GetText()** in ein **String**-Objekt befördern. Es sind folgende Werte vom Enumerationstyp **TextDataFormat** erlaubt:

- **CommaSeparatedValue**
- **Html**
- **Rtf**
- **Text**
- **UnicodeText**

Ist kein geeignetes Zwischenablagenformat vorhanden, liefert **GetText()** eine leere Zeichenfolge (**String.Empty**).

- **public static bool ContainsImage()**
public static Image GetImage()

ContainsImage() liefert den Wert **true** zurück, wenn die Zwischenablage Bitmap-Daten enthält, ansonsten **false**, z.B. auch in der folgenden Situation mit Metafile-Daten in der Zwischenablage:



Aus vorhandenen Bitmap-Daten lässt sich mit **GetImage()** ein **Bitmap**-Objekt erstellen. Fehlen geeignete Daten, dann liefert **GetImage()** die **null**-Referenz.

- **public static bool ContainsAudio()**
public static Stream GetAudioStream()
ContainsAudio() liefert den Wert **true** zurück, wenn die Zwischenablage Daten im **WaveAudio**-Format enthält, ansonsten **false**. Sind **WaveAudio**-Daten vorhanden, dann liefert **GetAudioStream()** ein **Stream**-Objekt mit Audiodaten.
- **public static bool ContainsFileDropList()**
public static StringCollection GetFileDropList()
ContainsFileDropList() liefert den Wert **true** zurück, wenn die Zwischenablage Daten im **FileDrop**-Format des Windows-Explorers enthält, ansonsten **false**. Sind **FileDrop**-Daten vorhanden, dann erstellt **GetFileDropList()** ein **StringCollection**-Objekt mit den Dateinamen.

Etwas allgemeiner definiert, aber trotzdem bequem nutzbar sind die folgenden Methoden:

- **public static bool ContainsData(String format)**
public static Object GetData(String format)

ContainsData() liefert den Wert **true** zurück, wenn die Zwischenablage Daten im angegebenen Format enthält, ansonsten **false**. Sind Daten im gewünschten Format vorhanden, dann werden sie von **GetData()** in ein passendes Objekt gepackt.

22.5.2 Die Clipboard-Methode **GetDataObject()**

Die schon seit .NET 1.0 vorhandene **Clipboard**-Methode **GetDataObject()** macht etwas mehr Aufwand beim Abruf von Zwischenablageinhalten, ist aber wegen ihrer Formatflexibilität gelegentlich von Nutzen. Außerdem lohnt sich die Beschäftigung mit dieser Methode, weil sie beim Ziehen und Ablegen (siehe Abschnitt 22.6) eine zentrale Rolle spielt. **GetDataObject()** liefert ein Objekt, das *sämtliche* aktuell in der Zwischenablage verfügbaren Formate enthält. Dieses Objekt stammt aus einer Klasse, welche die Schnittstelle **IDataObject** erfüllt.⁷⁵ Folglich können wir das Objekt mit einer von den folgenden Überladungen der Methode **GetDataPresent()** befragen, ob Daten mit einem bestimmten Format vorhanden sind:

```
public bool GetDataPresent(String format)
public bool GetDataPresent(Type typ)
public bool GetDataPresent(String format, bool autoConvert)
```

So wird die in Abschnitt 22.5.1 vorgeführte Methode **MitPasteOnClick()** mit den Mitteln von .NET 1.0 realisiert:

```
protected void MitPasteOnClick(Object sender, EventArgs e) {
    IDataObject dataObject = Clipboard.GetDataObject();
    if (dataObject.GetDataPresent(DataFormats.Rtf))
        editor.SelectedRtf = (String)dataObject.GetData(DataFormats.Rtf);
    else
        if (dataObject.GetDataPresent(typeof(String)))
            editor.SelectedText = (String)dataObject.GetData(typeof(String));
}
```

Im Beispiel werden beiden ersten **GetDataPresent()**-Überladungen vorgeführt (Formatspezifikation per **String** bzw. per **Type**-Objekt). Sollen bei der Existenzprüfung mögliche Konvertierungen unberücksichtigt bleiben, verwendet man die dritte Überladung und setzt den booleschen Parameter *autoConvert* auf **false**.

Zur Entnahme von Daten mit bestimmtem Format verwendet man eine von den folgenden Überladungen der **IDataObject**-Methode **GetData()**, wobei das Format wie bei **GetDataPresent()** durch seinen Namen oder durch einen .NET - Datentyp beschrieben werden kann:

```
public Object GetData(String format)
public Object GetData(Type typ)
public Object GetData(String format, bool autoConvert)
```

Der folgenden Tabelle ist zu entnehmen, welcher .NET - Datentyp **GetData()** abhängig vom angeforderten (und auch verfügbaren) Zwischenablagenformat resultiert:

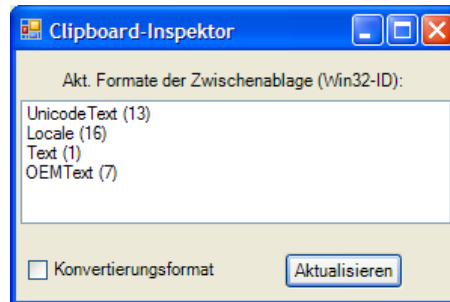
angefordertes Zwischenablageformat	resultierender FCL – Typ
Text, OemText, UnicodeText, SymbolicLink, DataInterchangeFormat, HTML Format, Rich Text Format	System.String (bei HTML Format und Rich Text Format mit Formatierungs-Tags)
Bitmap, DeviceIndependentBitmap	System.Drawing.Bitmap
Enhanced Metafile	System.Drawing.Imaging.Metafile
MetaFilePict, PenData, RiffAudio, WaveAudio, Palette, Csv	System.IO.MemoryStream
FileDrop	Array mit String -Elementen

Weil **GetData()** den Rückgabetyt **Object** besitzt, ist zur weiteren Verarbeitung in der Regel eine explizite Typumwandlung erforderlich, z.B.

⁷⁵ Nach Petzold (2002, S. 1018) handelt es sich in der Regel um ein Objekt der Klasse **DataObject**, die wir bereits kennen gelernt haben.

```
String s = (string) dataObject.GetData(DataFormats.StringFormat);
```

Fehlt das angeforderte Format, wird eine automatische Konvertierung versucht, wenn nicht der *autoConvert*-Parameterwert **false** dagegen spricht. Nachdem man z.B. per Windows-Notepad einen Text in die Zwischenablage befördert hat, liegen dort folgende Formate vor:



In dieser Lage liefert folgender **GetData()**-Aufruf ein **String**-Objekt:

```
IDataObject dataObject = Clipboard.GetDataObject();
dataObject.GetData(DataFormats.StringFormat);
```

Es resultiert hingegen ein **null**-Objekt, wenn man das automatische Konvertieren verbietet:

```
dataObject.GetData(DataFormats.StringFormat, false);
```

Ein **null**-Objekt resultiert immer dann, wenn das angeforderte Format nicht verfügbar und eine Konvertierung entweder unmöglich oder verboten ist.

Eine von **GetDataObject()** gelieferte **IDataObject**-Instanz spielt übrigens auch bei unserem **Clipboard**-Inspektor eine zentrale Rolle (siehe Abschnitt 22.3). In der folgenden Anweisung liefert es einen **String**-Array mit den Namen der Formate, die in der Zwischenablage vorhanden oder aus dortigen Inhalten durch automatische Konvertierung zu gewinnen sind:

```
String[] sF = Clipboard.GetDataObject().GetFormats(cbAutoConv.Checked);
```

Bei der Einsatzversion unseres Editors greifen wir in **MitCopyOnClick()** und **MitPasteOnClick()** nach erfolgreichem Studium der **Clipboard**-Methoden nun wieder auf die **RichTextBox**-Methoden **Copy()** und **Paste()** zurück, die bequemer sind und noch weitere Datenformate unterstützten (z.B. Bitmaps, Metadateien).

22.6 Ziehen und Ablegen

Benutzerfreundliche Windows-Anwendungen beherrschen auch den Datenaustausch durch Ziehen und Ablegen (Drag & Drop) mit der Maus, wobei Daten von einer *Quelle* bereitgestellt und von einem *Ziel* verarbeitet werden. Je nach Sachlage und Entwicklerfleiß unterstützt eine Anwendung von den beiden Rollen keine, eine oder beide.

Die von **System.Windows.Forms.Control** abstammenden Klassen sind durch entsprechende Methoden, Eigenschaften und Ereignisse für beide Rollen vorbereitet, so dass sich ein Datenaustausch zwischen Anwendungen und *innerhalb* einer Anwendung realisieren lässt. Unser Editor-Projekt ist zur Demonstration weniger gut geeignet, weil die zentrale Klasse **RichTextBox** bereits eine relativ komplette (per Voreinstellung aber nicht aktivierte) Drag & Drop - Kompetenz besitzt. Daher verwenden wir zunächst ein spezielles Beispiel, bevor wird die Drag & Drop - Funktionalität des Editors aktivieren und ergänzen.

Meist können die Benutzer über Maus-, Umschalt und **Strg**-Tasten ihre Wünsche näher spezifizieren und z.B. durch die Kombination (linke Maustaste + Umschalttaste + **Strg**-Taste) signalisieren, dass eine Verknüpfung angelegt werden soll.

Es lassen sich Daten in diversen Formaten übertragen (z.B. Texte, Bitmaps, Metadateien), und mit Hilfe des Windows-Explorers setzt man auch eine Liste mit Dateinamen in Bewegung, um die Zielanwendung zum Öffnen der Dateien aufzufordern.

Zielanwendungen können die per Maus angebotenen Daten ähnlich in Empfang nehmen wie Daten aus der Zwischenablage.

22.6.1 Die Rolle der Zielanwendung

Wir verwenden ein simples Beispielprogramm, das Zeichenfolgen, Bitmap-Daten und Grafikdateien per Drag & Drop entgegen nehmen und anzeigen kann, z.B.:



Drag & Drop - Empfänger ist das Hauptfenster selbst. Es malt eintreffende Daten zur Sicherung gegenüber Fensterschäden in ein **Bitmap**-Objekt, das per **Paint**-Ereignismethode flächendeckend auf der Fensteroberfläche gehalten wird.

Soll ein Formular oder Steuerelement per Drag & Drop Daten entgegen nehmen, muss die von der Klasse **Control** im Namensraum **System.Windows.Forms** geerbte Eigenschaft **AllowDrop** auf **true** gesetzt werden (Voreinstellung: **false**), z.B.:

```
this.AllowDrop = true;
```

Beim Ziehen und Ablegen geht es recht ereignisreich zu. Für die Zielanwendung sind folgende Ereignisse relevant:

Control-Ereignis	Anlass und Behandlung
DragEnter	Die Maus entert bei gedrückter Taste mit Daten im Schlepp den (Klienten)bereich des Fensters bzw. Steuerelements. Jetzt kann sich eine Anwendung bereit erklären, Daten entgegen zu nehmen. Allerdings bieten dazu auch noch zahlreiche DragOver -Ereignisse reichlich Gelegenheit.
DragOver	Die mit Daten beladene Maus bewegt sich im (Klienten)bereich des Fensters bzw. Steuerelements. Spätestens jetzt muss sich das betroffene Objekt dazu bereit erklären, Daten in einem von den verfügbaren Formaten entgegen zu nehmen. Äußert sich eine Anwendung <i>nicht</i>, nimmt der Mauszeiger die Form eines Verbotsszeichens an . Anderenfalls legt die vom Drag & Drop - Ziel angekündigte Übertragungsart die neue Form des Mauszeigers fest, so dass der Benutzer eine optische Rückmeldung erhält:  Verschieben  Kopieren  Verknüpfung anlegen Der Benutzer kann per Tastenkombination (z.B. Linke Maustaste + Strg-Taste für Kopieren) die Übertragungsart vorschreiben und dabei eventuell während des Ziehens seine Meinung ändern, so dass eine DragOver-Behandlung mit Reaktionsmöglichkeit durch das Drag & Drop - Ziel angemessen ist und die alleinige DragEnter-Behandlung nicht genügt.

Control-Ereignis	Anlass und Behandlung
DragDrop	Die Taste der beladenen Maus wird über dem (Klienten)bereich des Fensters bzw. Steuerelements losgelassen, und die transportierten Daten werden einer registrierten Ereignismethode zur Verfügung gestellt. Der Zugriff erfolgt genauso wie beim Einfügen aus der Zwischenablage.
DragLeave	Die mit Daten beladene Maus verlässt den Klientenbereich.

Man kann sich meist auf die Behandlung der Ereignisse **DragOver** und **DragDrop** beschränken und entsprechende Methoden registrieren. Das gilt auch für unser Beispiel:

```
this.DragOver += FormOnDragOver;
this.DragDrop += FormOnDragDrop;
```

Einem **DragEventHandler** wird ein Objekt der Klasse **DragEventArgs** als Aktualparameter übergeben. Es besitzt folgende Eigenschaften:

Eigenschaft	Zweck
X, Y	Typ: int Zugriff: get Diese Eigenschaften enthalten die Mauskoordinaten bezogen auf den Bildschirm. In der Regel müssen sie mit der Control -Methode PointToClient() (siehe Abschnitt 15.4.1) in Klientenbereichskoordinaten umgerechnet werden.
Data	Typ: IDataObject Zugriff: get Diese Eigenschaft zeigt auf ein Objekt, das die Schnittstelle IDataObject erfüllt und folglich gemäß Beschreibung in Abschnitt 22.5.2 genutzt werden kann. Hier zeigt sich deutlich die Parallelität zur Zwischenablagentechnik.
AllowedEffect	Typ: DragDropEffects Zugriff: get Von der Quellanwendung wird über die bitweise kombinierbaren Werte der Enumeration DragDropEffects festgelegt, was die Zielanwendung mit den Daten tun darf, z.B. Copy , Move , Link .
Effect	Typ: DragDropEffects Zugriff: get, set Diese Eigenschaft ist vom selben Typ wie AllowedEffect , erlaubt aber einen schreibenden Zugriff. Durch Setzen eines Wertes signalisiert die Zielanwendung ihre Verarbeitungsabsichten, wobei sich die Gestalt des Mauszeigers entsprechend ändert. Ein gesetzter Effect -Wert bleibt bis zu einer eventuellen Änderung erhalten und wird nach dem DragDrop -Ereignis an die Quelle gemeldet, die dann z.B. beim Wert DragDropEffects.Move für das Löschen von Daten verantwortlich ist.

Eigenschaft	Zweck								
KeyState	Typ: int Zugriff: get Bei ihren Verarbeitungsabsichten orientiert sich die Ziellanwendung an den Genehmigungen in AllowedEffect und am Maus- bzw. Tastaturstatus. Folgende Bits der int-wertigen KeyState-Eigenschaft signalisieren mit einer Eins, dass die zugehörige Taste gedrückt ist: 1 Linke Maustaste 2 Rechte Maustaste 4 Umschalttaste 8 Strg-Taste 16 Mittlere Maustaste 32 Alt-Taste Die Tasten werden z.B. vom Windows-Explorer als Ziellanwendung so interpretiert: <table border="0"> <tr> <td>Linke Maustaste</td><td>Verschieben innerhalb eines Laufwerks und Kopieren zwischen Laufwerken</td></tr> <tr> <td>Rechte Maustaste</td><td>Absicht nach DragDrop per Kontextmenü erfragen</td></tr> <tr> <td>Linke Maustaste + Strg-Taste</td><td>Kopieren</td></tr> <tr> <td>Linke Maustaste + Strg-Taste + Umschalttaste</td><td>Verknüpfung anlegen</td></tr> </table>	Linke Maustaste	Verschieben innerhalb eines Laufwerks und Kopieren zwischen Laufwerken	Rechte Maustaste	Absicht nach DragDrop per Kontextmenü erfragen	Linke Maustaste + Strg -Taste	Kopieren	Linke Maustaste + Strg -Taste + Umschalttaste	Verknüpfung anlegen
Linke Maustaste	Verschieben innerhalb eines Laufwerks und Kopieren zwischen Laufwerken								
Rechte Maustaste	Absicht nach DragDrop per Kontextmenü erfragen								
Linke Maustaste + Strg -Taste	Kopieren								
Linke Maustaste + Strg -Taste + Umschalttaste	Verknüpfung anlegen								

Falls die **DragOver**-Behandlungsmethode unseres Beispielsprogramms Daten in den Formaten **Text**, **Bitmap** oder **FileDrop** im **DragEventArgs**-Ereignisobjekt vorfindet, signalisiert sie per **Effect**-Eigenschaft die Bereitschaft zum Verschieben oder zum Kopieren. Dabei sind die Genehmigungen der Quellanwendung (Eigenschaft **AllowedEffect**) und die Benutzerabsicht (**KeyState**-Eigenschaft) zu berücksichtigen:

```
private void FormOnDragOver(object sender, DragEventArgs dea) {
    if ((dea.KeyState & 1) != 1)
        return;
    if (dea.Data.GetDataPresent(DataFormats.Text) ||
        dea.Data.GetDataPresent(DataFormats.Bitmap) ||
        dea.Data.GetDataPresent(DataFormats.FileDrop)) {
        if (((dea.AllowedEffect & DragDropEffects.Move) != 0) &&
            (dea.KeyState & 8) == 0) {
            dea.Effect = DragDropEffects.Move;
        } else {
            if ((dea.AllowedEffect & DragDropEffects.Copy) != 0)
                dea.Effect = DragDropEffects.Copy;
        }
    }
}
```

Wir fühlen uns nur angesprochen, wenn die linke Maustaste gedrückt ist, was durch einen Bit-Vergleich der **DragEventArgs**-Eigenschaft **KeyState** mit dem **int**-Wert Eins zu ermitteln ist:

```
if ((dea.KeyState & 1) != 1)
    return;
```

Das per **DragEventArgs**-Parameter übergebene Objekt vom Typ **IDataObject** kann mit der Methode **GetDataPresent()** zur Anwesenheit bestimmter Formate befragt werden (vgl. Abschnitt 22.5.2), z.B.:

```
dea.Data.GetDataPresent(DataFormats.Text)
```


Bei diesen Anfragen eignen sich zur Bezeichnung der Formate die in Abschnitt 22.2 behandelten **DataFormats**-Felder.

Dasselbe Objekt mit allen verfügbaren Datenformaten steht auch der **DragDrop**-Behandlungsmethode zur Verfügung, die sich mit **GetData()**-Aufrufen bedient und die Daten nach einer Typanpassung über die passende **Graphics**-Methode in das **Bitmap**-Objekt der Anwendung malt:

```
private void FormOnDragDrop(object sender, DragEventArgs dea) {
    if (dea.Data.GetDataPresent(DataFormats.Text)) {
        String text = (string) dea.Data.GetData(DataFormats.Text);
        insertPoint = PointToClient(new Point(dea.X, dea.Y));
        gPicture.DrawString(text, Font, Brushes.Black, insertPoint);
        Refresh();
    } else if (dea.Data.GetDataPresent(DataFormats.Bitmap)) {
        intDrop = true;
        insertPicture = (Bitmap) dea.Data.GetData(DataFormats.Bitmap);
        insertPoint = PointToClient(new Point(dea.X, dea.Y));
        gPicture.DrawImage(insertPicture, insertPoint);
        Renew();
    } else if (dea.Data.GetDataPresent(DataFormats.FileDrop)) {
        String[] fns = (string[]) dea.Data.GetData(DataFormats.FileDrop);
        gPicture.DrawImage(new Bitmap(fns[0]),
                           new Rectangle(0, 0, pictureWidth, pictureHeight));
        Refresh();
    }
}
```

Die Referenzvariable **gPicture** zeigt auf ein **Graphics**-Objekt zu der **Bitmap**, die auf der Fensteroberfläche zu sehen ist. Text- oder Bitmap-Daten werden an der Klickstelle per **DrawString()** oder **DrawImage()** gemalt. Eintreffende Grafikdateien füllen den Klientenbereich der Anwendung.

22.6.2 Die Rolle der Quellanwendung

Eine Drag & Drop - Aktion wird von der Quellanwendung eingeleitet durch Aufruf der **Control**-Methode **DoDragDrop()**, die folgende Aktualparameter verlangt:

- **Object daten**
In Frage kommen:
 - o Objekte mit der Basisklasse **String**, **Bitmap** oder **Metafile**
 - o Objekte aus einer Klasse, welche eine von den Schnittstellen **ISerializable** oder **IDataObject** implementiert
- **DragDropEffects erlaubt**
Erlaubt sind die bitweise kombinierbaren Werte der Enumeration **DragDropEffects**.

Unser Beispielprogramm übergibt ein Objekt der Klasse **DataObject**, das ein **Bitmap**-Objekt mit einem rechteckigen Bildausschnitt samt Formatangabe enthält und erlaubt sowohl das Kopieren als auch das Verschieben:

```
Bitmap subPic = picture.Clone(mark, picture.PixelFormat);
dataObject.SetData(DataFormats.Bitmap, subPic);
DragDropEffects ddResult = DoDragDrop(dataObject,
    DragDropEffects.Copy | DragDropEffects.Move);
```

Im Beispiel hätten wir den Bildausschnitt auch direkt an **DoDragDrop()** übergeben können:

```
DragDropEffects ddResult = DoDragDrop(
    picture.Clone(mark, picture.PixelFormat),
    DragDropEffects.Copy | DragDropEffects.Move);
```

Die Flexibilität der Klasse **DataObject** kommt dann zum Tragen, wenn *mehrere* Datenformate oder individuelle Formatbezeichnungen erforderlich sind.

Im Beispielprogramm kann der Benutzer durch rechte Mausklicks einen Bildausschnitt markieren



oder die Markierung aufheben. Zuständig ist die **MouseDown**-Ereignismethode, wo auch der Ort des letzten linken Mausklicks festgehalten wird:

```
void FormOnMouseDown(object sender, MouseEventArgs e) {
    if (e.Button == MouseButton.Left) {
        mousePos = new Point(e.X, e.Y);
        return;
    } else
        if (e.Button != MouseButton.Right)
            return;
    if (oleck.X < 0) {
        oleck = new Point(e.X, e.Y);
    } else
        if (mark.Width < 0) {
            if ((e.X > oleck.X) && (e.Y > oleck.Y))
                mark = new Rectangle(oleck, new Size(e.X - oleck.X, e.Y - oleck.Y));
            gForm.DrawRectangle(Pens.Red, mark);
        } else {
            gForm.Clear(Color.White);
            Renew();
        }
}
```

In der **MouseMove**-Ereignismethode

```
void FormOnMouseMove(object sender, MouseEventArgs e) {
    if (!(e.Button == MouseButton.Left) || mark.Width < 0 ||
        mousePos.X < oleck.X || mousePos.X > oleck.X + mark.Width ||
        mousePos.Y < oleck.Y || mousePos.Y > oleck.Y + mark.Height)
        return;
    if (Math.Abs(mousePos.X - e.X) > SystemInformation.DragSize.Width ||
        Math.Abs(e.Y - mousePos.Y) > SystemInformation.DragSize.Height) {
        Rectangle localMark = mark;
        intDrop = false;
        Bitmap subPic = picture.Clone(mark, picture.PixelFormat);
        dataObject.SetData(DataFormats.Bitmap, subPic);
        DragDropEffects ddResult = DoDragDrop(dataObject,
            DragDropEffects.Copy | DragDropEffects.Move);
        if (ddResult == DragDropEffects.Move) {
            gPicture.FillRectangle(Brushes.White, localMark);
            if (intDrop)
                gPicture.DrawImage(insertPicture, insertPoint);
        }
        Renew();
    }
}
```

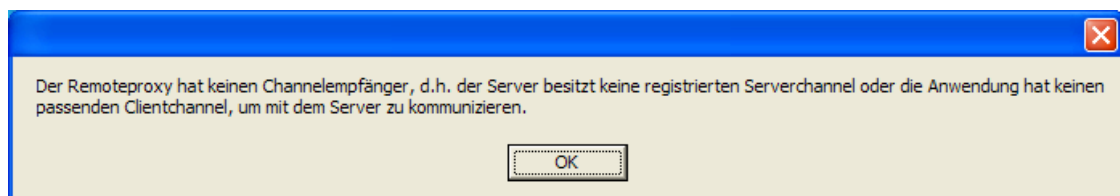
wird das Ziehen und Ablegen per **DoDragDrop()** eingeleitet, wenn ...

- eine Markierung vorhanden ist,
- die linke Maustaste in der Markierung gedrückt wurde und immer noch gedrückt ist,
- die aktuelle Mausposition in horizontaler oder vertikaler Richtung so weit von der Druckstelle entfernt ist, dass aufgrund der **SystemInformation**-Eigenschaft **DragSize** beim Benutzer eine Drag & Drop - Intention unterstellt werden kann.

Der **DoDragDrop()**-Aufruf kehrt erst dann zurück, wenn die Aktion abgeschlossen oder per **Esc**-Taste abgebrochen wurde.

Nach einer Verschiebung (Rückgabewert **DragDropEffects.Move**) ist der Bildausschnitt am alten Ort zu löschen, wobei im Fall einer anwendungsinternen Verschiebung mit Überlappung von altem und neuem Ort eine Beschädigung des neuen Orts zu vermeiden ist.

Während man problemlos per Drag & Drop Bitmap-Daten in die Anwendung hinein befördern oder anwendungsintern bewegen kann, scheitert der Transport in eine andere Anwendung. Selbst zwei Instanzen der Anwendung können keine Bitmap-Daten per Drag & Drop austauschen. Es tritt ein Ausnahmefehler mit folgender Meldung auf:



Es handelt sich vermutlich um ein generelles .NET - Problem (getestet mit der Version 3.5). Allzu gravierend ist das Problem nicht, weil das Exportieren von Bitmap-Daten per Drag & Drop generell selten praktiziert wird. Ob für das rein anwendungsinterne Bewegen von Bildausschnitten die Drag & Drop - Technik notwendig und sinnvoll ist, sei dahingestellt.

Bisher wurde das **Control**-Ereignis **QueryContinueDrag** verschwiegen, das während eines Drag & Drop - Vorgangs bei Maus- oder Tastaturaktionen gefeuert wird und für eine Drag & Drop - Quelle von Interesse sein kann. Weil unser Beispielprogramm keine Bildausschnitte exportieren kann, soll das Drag & Drop abgebrochen werden, sobald die Maus den Klientenbereich des Anwendungsfensters verlässt:

```
this.QueryContinueDrag += FormOnQueryContinueDrag;

void FormOnQueryContinueDrag(object sender, QueryContinueDragEventArgs e) {
    Point pc = PointToClient(
        new Point(Control.MousePosition.X, Control.MousePosition.Y));
    if (pc.X < 0 || pc.X > ClientSize.Width ||
        pc.Y < 0 || pc.Y > ClientSize.Height)
        e.Action = DragAction.Cancel;
}
```

Wir ermitteln und beurteilen die Klientenkoordinaten der aktuellen Mausposition, die einer **Control**-Eigenschaft zu entnehmen ist.

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

...\\BspUeb\\Zwischenablage, Ziehen und Ablegen\\DragAndDrop

22.6.3 Drag & Drop - Funktionalität für unseren Editor

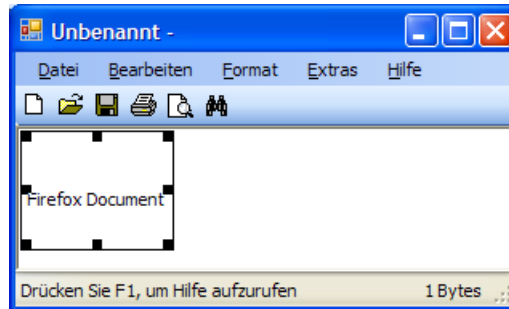
In der Klasse **RichTextBox** kann seit der .NET-Version 2.0 über die Eigenschaft **EnableAutoDragDrop** eine weit gehende Drag & Funktionalität freigeschaltet werden, was im Editor-Projekt geschehen soll:

```
editor.EnableAutoDragDrop = true;
```

Weil damit die **Control**-Eigenschaft **AllowDrop** (vgl. Abschnitt 22.6.1) beim **RichTextBox**-Steuerelement unattraktiv geworden ist, wird sie übrigens im Eigenschaftsfenster unserer Entwicklungsumgebungen nicht mehr angezeigt, obwohl sie natürlich weiter verfügbar ist.

Dank **EnableAutoDragDrop** besitzt der Editor u.a. die folgenden Drag & Drop - Kompetenzen:

- Bewegen von formatiertem Text, sowohl innerhalb der Anwendung als auch im Austausch mit anderen, entsprechend kompetenten Programmen (getestet mit MS-Word)
- Bewegen von Bitmap-Grafiken *innerhalb* der Anwendung
- Bitmap-Import aus einer geeigneten anderen Anwendung (getestet mit MS-Word). Wie aufgrund der Erfahrungen in Abschnitt 22.6.2 zu erwarten, klappt der Bitmap-Export *nicht*.
- Beim Ablegen eines Dateinamens resultiert ein OLE-Objekt, z.B.:



Per Doppelklick auf einen solchen Link lässt sich die eingebundene Datei in der Originalanwendung öffnen und bearbeiten.

An Stelle des OLE-Objekt - Imports wäre ein Öffnen von TXT- oder RTF-Dateien per Drag & Drop sinnvoller. Per **EnableAutoDragDrop** ist diese Variante aber nicht zu haben. Eine mögliche, aber mit Mühen verbundene Alternative besteht darin,

- auf die Vollautomatik zu verzichten,
- die Eigenschaft **AllowDrop** auf den **true** zu setzen,
- das Drag & Drop - Quellverhalten zu implementieren,
- die Ereignisse **DragOver** und **DragDrop** zu behandeln

Wir verzichten auf den Aufwand und damit auf das Öffnen von Dateien per Drag & Drop.

22.7 Übungsaufgaben zu Kapitel 22

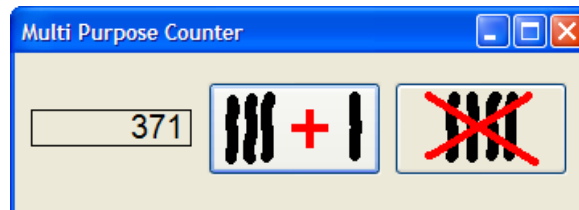
1) Wie Sie sicher wissen, kann die Windows-Zwischenablage nur *ein* Element aufnehmen. Ist es trotzdem möglich, dass sich dort z.B. gleichzeitig ein Bild und ein Text befinden?

23 Ressourcen

Neben MSIL-Code, Metadaten und einem Manifest kann ein Assembly auch Ressourcen wie Bitmaps und Zeichenfolgen enthalten. Bei der Integration von Ressourcen in Assemblies spielen integrative, aber auch separatistische Bestrebungen eine Rolle:

- Viele unserer bisherigen Beispielprogramme laden Bitmaps aus Dateien, die sich folglich im selben Verzeichnis wie das Assembly befinden müssen, wenn es nicht zu einem Ausnahmefehler kommen soll. Indem man die von einem Programm benötigten Ressourcen in sein Assembly integriert, minimiert man offenbar den Aufwand bei der Weitergabe des Programms und das und das Fehlerrisiko durch verloren gegangene Bestandteile. Die entsprechenden Techniken werden anschließend vorgestellt.
- Speziell bei Programmen für den Einsatz in verschiedenen Sprachräumen bzw. Kulturen ist es sinnvoll, Zeichenfolge-Ressourcen vom eigentlichen Code getrennt in Ressourcendateien unterzubringen. Während man sprachneutrale Ressourcen in das **Hauptassembly** des Programms integriert, bringt man sprach- bzw. kulturabhängigen Ressourcen meist in einem so genannten **Satellitenassembly** unter. Bei einer Änderung der Programmlogik ist dann nur *ein* Assembly betroffen, und eine Modifikation von kulturspezifischen Ressourcen erfordert keine Neuübersetzung des Programms, die bei großen Projekten durchaus zeitaufwändig sein kann. In diesem Manuskript können die für mehrsprachige Projekte sinnvollen Separationstechniken aus Zeitgründen nicht dargestellt werden.

Als Beispiel betrachten wir den *Multi Purpose Counter* aus Abschnitt 9.4.2.3, dessen Bitmap-Ressourcen nun aber in das Assembly integriert werden sollen. Zu Übungszwecken behandeln wir auch den Titeltext als Ressource, wenngleich daraus im Beispiel kein praktischer Nutzen entsteht.



23.1 Ressourcendateien und -generatoren

Das .NET-Framework benutzt für Ressourcen zwei äquivalente und aufeinander abbildbare Dateiformate: die (zumindest partiell) für Menschen lesbare RESX-Datei mit XML-Code und die binäre RESOURCES-Datei. Für Zeichenfolgen existiert noch ein einfach handhabbares Textdateiformat. Bei umfangreichen Ressourcen (z.B. Videos) kommt anstelle der Einbettung in die Ressourcen-Datei auch ein Verweis auf externen Dateien in Frage.

23.1.1 RESX-Datei

Die **RESX-Datei** verwendet das XML-Format, kann also prinzipiell mit einem Texteditor bearbeitet werden, wobei man komplexe Kreationen aber der FCL-Klasse **ResXResourceWriter** oder einem anderen Werkzeug überlassen sollte (siehe unten). Wir haben im Zusammenhang mit den .NET-Konfigurationsdateien einige Details zum XML-Format kennen gelernt (siehe Abschnitt 19.1.1).

In der RESX-Datei folgen auf die XML-Schemadefinition zur Beschreibung der Struktur

```

<?xml version="1.0" encoding="utf-8"?>
<root>
  <xsd:schema id="root" . . . >
    <xsd:import namespace="http://www.w3.org/XML/1998/namespace" />
    <xsd:element name="root" msdata:IsDataSet="true">
      <xsd:complexType>
        <xsd:choice maxOccurs="unbounded">
          <xsd:element name="metadata">
            . . .
          </xsd:element>
          <xsd:element name="resheader">
            . . .
          </xsd:element>
        </xsd:choice>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
</root>

```

zunächst einige **<resheader>**-Elemente mit einer Namensangabe und einem eingebetteten **<value>**-Element, die u.a. eine Versionsangabe enthalten und für das Schreiben bzw. Lesen der Ressourcen jeweils die zuständige .NET - Klasse angeben, z.B.:

```

<resheader name="resmimetype">
  <value>text/microsoft-resx</value>
</resheader>
<resheader name="version">
  <value>2.0</value>
</resheader>
<resheader name="reader">
  <value>System.Resources.ResXResourceReader, . . . </value>
</resheader>
<resheader name="writer">
  <value>System.Resources.ResXResourceWriter, . . . </value>
</resheader>

```

Es folgt für jede Ressource ein **<data>**-Element mit Namensangabe und einem eingebetteten **<value>**-Element, z.B. bei einer Zeichenfolge

```

<data name="TitelText" xml:space="preserve">
  <value>Multi Purpose Counter</value>
</data>

```

Außer bei Zeichenfolge-Ressourcen enthält das **<data>**-Element auch eine Typinformation, wobei eine .NET - Klasse und/oder ein Mimetype angegeben werden, z.B. bei einer Bitmap-Ressource:

```

<data name="Count" type="System.Drawing.Bitmap, System.Drawing"
  mimetype="application/x-microsoft.net.object.bytearray.base64">
  <value>
    Qk300gAAAAAADYAAAAoAAAAZAAAADIAAABABgAAAAAAAAAADEdGAAxAAAAA20ns20ns
    20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns
    . . .
    20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns
  </value>
</data>

```

Mit einem Objekt der Klasse **ResXResourceWriter** aus dem Namensraum **System.Resources** kann man eine RESX-Datei per Programm erstellen, z.B.:

```

using System;
using System.Drawing;
using System.Resources;

class ResXResourceWriterDemo {
    static void Main() {
        ResXResourceWriter rsxw = new ResXResourceWriter("mpc.resx");
        rsxw.AddResource("TitelText", "Multi Purpose Counter");
        rsxw.AddResource("Count", new Bitmap("count.bmp"));
        rsxw.AddResource("Reset", new Bitmap("reset.bmp"));
        rsxw.Close();
        Console.WriteLine("Fertig!"); Console.ReadLine();
    }
}

```

Zum *Lesen* einer RESX-Datei per Programm (z.B. in Ihrem Ressourcen-Editor) steht die FCL-Klasse **ResXResourceReader** bereit.

Unsere Entwicklungsumgebungen bieten eine umfangreiche Unterstützung für das Erstellen, Bearbeiten und Verwenden von RESX-Dateien (vgl. Abschnitt 23.3), so dass man in der Regel weder zu einem XML-Editor greifen, noch ein **ResXResourceWriter**-basiertes Programm erstellen wird.

23.1.2 TXT-Datei für Zeichenfolge-Ressourcen

Bei Zeichenfolgen-Ressourcen kann man eine einfach editierbare Textdatei erstellen. Aus dem *WorldCalc*-Beispiel, das im SDK zu .NET 1.1 enthalten war, stammt die folgende Datei **mystrings.de.txt**:

```

Math_Greeting = Guten Tag - Ganzzahl Taschenrechner
Math_Formula_Label = Formel und Ergebnisse:
Math_Clear_Button = Löschen
Math_Calc_Button = Berechnen
Math_Calc_Error = Ungültige Formel. Bitte versuchen Sie es noch einmal.

```

Anders als beim RESX-Format kann man sich beim TXT-Format ein direktes Editieren durchaus vorstellen. Welche Verarbeitungsschritte danach bis zur Verwendung der Ressourcen im Programm nötig sind, erfahren Sie gleich.

23.1.3 RESOURCES-Datei

Aus einer RESX- oder TXT-Datei lässt sich mit dem Konsolenprogramm **ResGen.exe**, das im kostenlos verfügbaren Windows-SDK der Firma Microsoft vorhanden ist (Ordner: ...\\Microsoft SDKs\\Windows\\v6.0A\\bin) eine äquivalente Binärdatei mit der Namensendung **.resources** erstellen, z.B.:

```

>"C:\\Programme\\Microsoft SDKs\\Windows\\v6.0A\\bin\\resgen" mpc.resx
Liest in den 3-Ressourcen von "mpc.resx".
Ressourcendatei wird geschrieben... Fertig.

```

ResGen.exe beherrscht aber auch noch andere Konvertierungen zwischen den Ressourcen-Datei-formaten:

Quellformat	mögliche Zielformate
TXT	RESX, RESOURCES
RESX (nur Zeichenfolgen)	TXT, RESOURCES
RESX	RESOURCES
RESOURCES (nur Zeichenfolgen)	TXT, RESX
RESOURCES	RESX

Mit einem Objekt der Klasse **ResourceWriter** aus dem Namensraum **System.Resources** kann man eine RESOURCES-Datei direkt per .NET-Anwendung erstellen, z.B.:

```
using System;
using System.Drawing;
using System.Resources;

class ResourceWriterDemo {
    static void Main() {
        ResourceWriter rsw = new ResourceWriter("mpc.resources");
        rsw.AddResource("TitelText", "Multi Purpose Counter");
        rsw.AddResource("Count", new Bitmap("count.bmp"));
        rsw.AddResource("Reset", new Bitmap("reset.bmp"));
        rsw.Close();
        Console.WriteLine("Fertig!"); Console.ReadLine();
    }
}
```

Die Inhalte einer RESOURCES-Datei müssen per Compiler in ein Assembly oder ein Satellitenassembly befördert werden (siehe Abschnitt 23.2).

23.2 Ressourcen verwenden

Die Inhalte einer RESOURCES-Datei werden per Compiler in ein Assembly befördert werden, z.B. bei der Übersetzung unseres Multi Purpose Counters:

```
>csc /target:winexe /res:mpc.resources mpc.cs
```

Weil das Beispielprogramm seine Bitmaps nun nicht mehr aus externen Dateien lädt, sondern auf Ressourcen im eigenen Assembly zugreift, ist der Quellcode des Fensterkonstruktors zu modifizieren:

```
ResourceManager rm = new ResourceManager("mpc",
                                           Assembly.GetExecutingAssembly());

Text = rm.GetString("TitelText");
count = new Button();
Bitmap bmp = (Bitmap)rm.GetObject("Count");
bmp.MakeTransparent(bmp.GetPixel(1, 1));
count.Image = bmp;

reset = new Button();
bmp = (Bitmap)rm.GetObject("Reset");
bmp.MakeTransparent(bmp.GetPixel(1, 1));
reset.Image = bmp;
```

Es wird ein Objekt der Klasse **ResourceManager** (Namensraum **System.Resources**) erzeugt, wobei dem gewählten Konstruktor folgende Parameter zu übergeben sind:

- **String** *namensstamm*
Es handelt sich um den Namensstamm (ohne **.resources**) der zugrunde liegenden RESOURCES-Datei.
- **Assembly** *assembly*
Im obigen Beispiel wird mit Hilfe der statischen Methode **GetExecutingAssembly()** der Klasse **Assembly** (Namensraum **System.Reflection**) auf das gerade ausgeführte Assembly verwiesen.

Mit den Methoden **GetString()** bzw. **GetObject()** beauftragt man ein **RessourceManager**-Objekt, eine Ressource vom **String**-Typ bzw. von beliebigem Typ zu liefern. Dabei ist zu beachten:

- Bei den Namen der Ressourcen ist die Groß/Klein –Schreibung relevant.
- Weil **GetObject()** den Rückgabotyp **Object** besitzt ist eine explizite Typanpassung erforderlich.

Selbstverständlich können in einem **csc**-Aufruf auch *mehrere* RESOURCES-Dateien in *ein* Assembly aufgenommen werden.

Weil Sie wohl kaum mit einem simplen Texteditor und **csc**-Kommandos Ihre Programme erstellen wollen, wird gleich beschrieben, wie man die Visual C# Express Edition beauftragt, eine RESOURCES-Datei automatisch zu erstellen und an den Compiler zu übergeben.

23.3 Ressourcen-Verwaltung mit der Visual C# 2008 Express Edition

Die Visual C# Express Edition unterstützt bzw. erledigt das Erstellen, Konvertieren und Übersetzen von RESX-Dateien. Wir beschränken uns auf die so genannte **Standardressourcendatei**, die im Hauptassembly landet und als letzte Rückfallebene dient, wenn beim Start einer .NET - Anwendung keine kulturspezifischen Ressourcen in einem **Satellitenassembly** angetroffen werden.

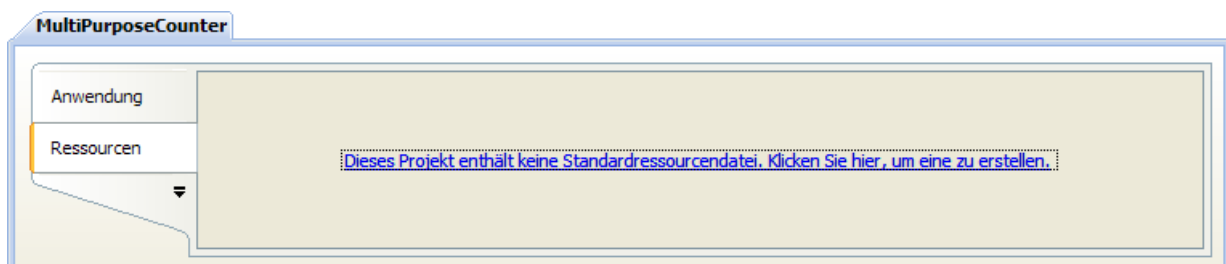
Bei einem neuen WinForms-Projekt befinden sich im Unterordner **Properties** ...

- die RESX-Datei **Resources.resx**
Mit diesem Dateityp haben wir uns in Abschnitt 23.1.1 beschäftigt.
- die Quellcodedatei **Resources.Designer.cs**
Hier erstellt die Entwicklungsumgebung eine Hilfsklasse für den bequemen Zugriff auf die Ressourcen, wobei im Wesentlichen die in Abschnitt 23.2 beschriebene Technik zum Einsatz kommt.

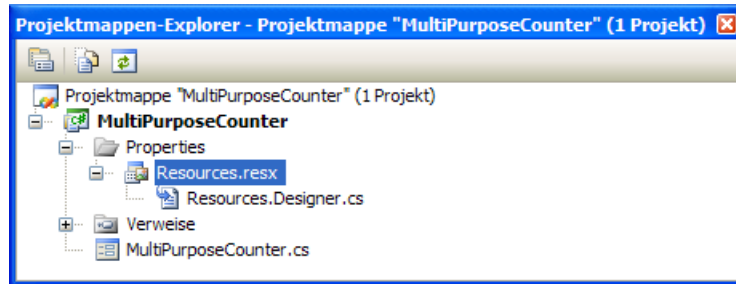
Beim Erstellen des Projekts laufen je nach Bedarf zwei Verarbeitungsschritte ab:

- Die Datei **...\Properties\Resources.resx** wird in eine Binärversion (mit Namenserverweiterung **resources**) konvertiert, die im Unterordner **...\obj\Debug** bzw. **...\obj\Release** landet (je nach Erstellungskonfiguration).
- Die **resources**-Datei wird vom Compiler in ein Assembly integriert.

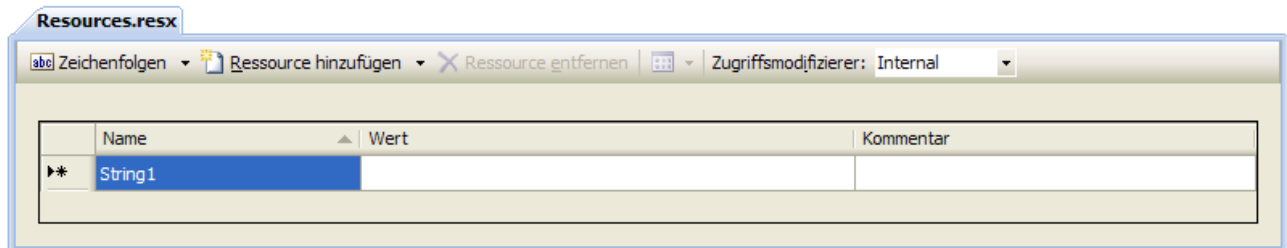
Weil wir beim Multi Purpose Counter von einer leeren Projektvorlage ausgegangen sind, ist noch keine Ressourcendatei vorhanden. Die **Ressourcen**-Registerkarte des Projekteigenschaftsformulars bietet einen bequemen Weg, um die Standardressourcendatei zu erstellen:



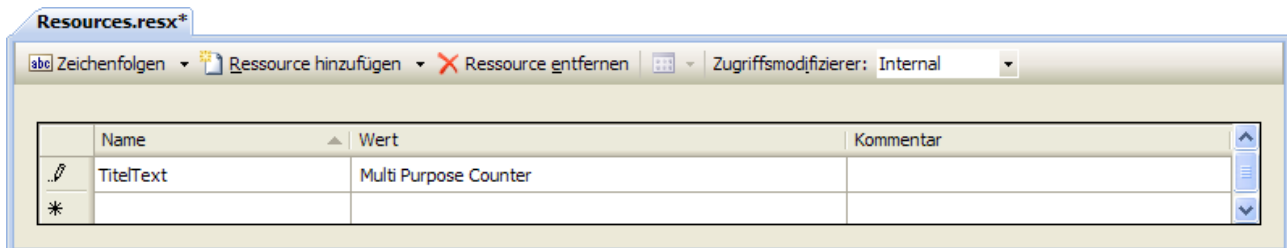
Die neuen Dateien erscheinen auch im Projektmappenexplorer:



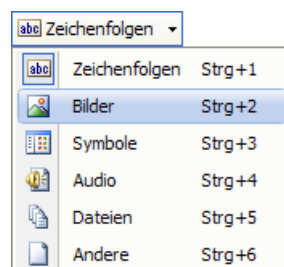
Von hier aus kann die Bearbeitung der Ressourcendatei per Doppelklick gestartet werden:



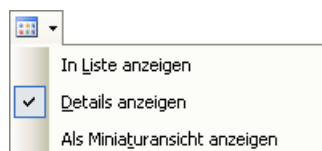
Für unser Zählprogramm tragen wir zunächst die Zeichenfolgen-Ressource ein:



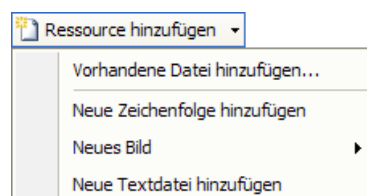
Um die Bitmap-Ressourcen zu ergänzen, wählen wir in der Liste am linken Fensterrand die Option **Bilder**



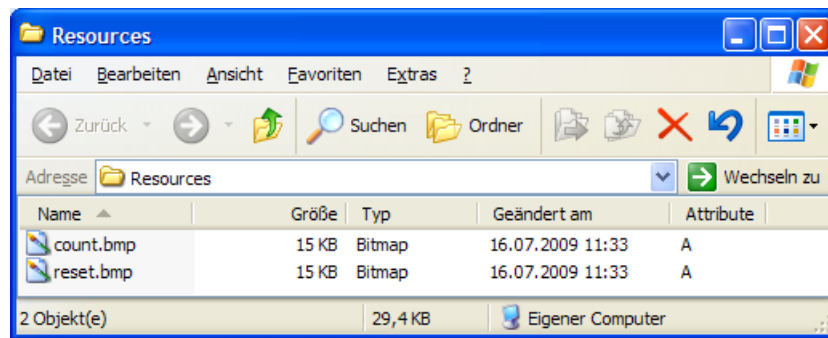
und in der -Liste die Option **Details anzeigen**:



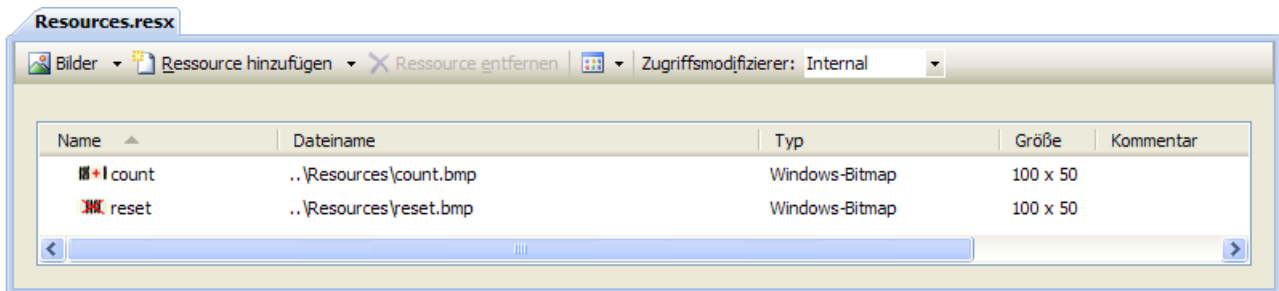
Nun wählen wir in der Liste **Ressource hinzufügen** die Option **Vorhandene Datei hinzufügen**



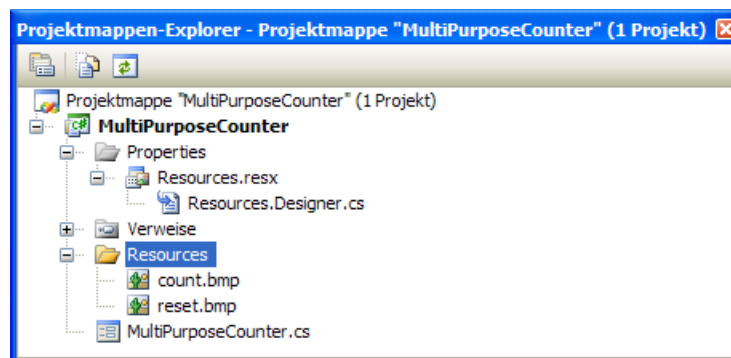
und öffnen die beiden Bitmap-Dateien. Wird ein Quellordner außerhalb des Projektdatenbaums gewählt, erstellt die Visual C# 2008 Express Edition den neuen Projekt-Unterordner **Resources**



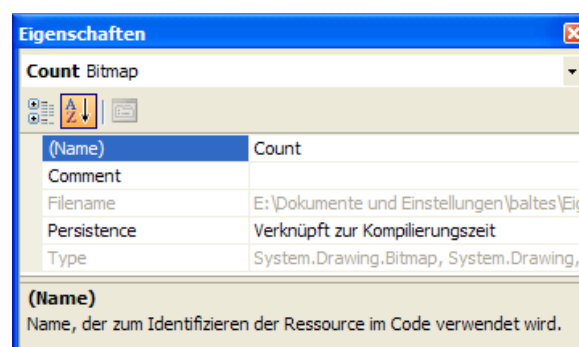
und kopiert die Bitmap-Dateien dorthin. Außerdem erscheinen die Dateien im Ressourcen-Formular:



Im Projektmappenexplorer zeigt sich nun folgendes Bild:



Ihren Quellcode-Namen erhalten die Ressourcen via Ressourcen-Formular oder Eigenschaftsfenster, z.B.:



Da wir in diesem Manuskript Methodennamen unabhängig von der Schutzstufe mit einem Großbuchstaben beginnen lassen (vgl. Abschnitt 4.3.1), und es sich bei Eigenschaften um Paare von Methoden handelt, verwenden wir abweichend von der Vorgabe der Entwicklungsumgebung für die Ressourcen-Referenzen große Anfangsbuchstaben.

In der Datei **Resources.Designer.cs** steckt eine automatisch erstellte Hilfsklasse namens **Resources**, die über statische Eigenschaften einen bequemen Zugriff auf die Ressourcen ermöglicht:

```

internal static System.Drawing.Bitmap Count {
    get {
        object obj = ResourceManager.GetObject("Count", resourceCulture);
        return ((System.Drawing.Bitmap)(obj));
    }
}

internal static System.Drawing.Bitmap Reset {
    get {
        object obj = ResourceManager.GetObject("Reset", resourceCulture);
        return ((System.Drawing.Bitmap)(obj));
    }
}

internal static string TitelText {
    get {
        return ResourceManager.GetString("TitelText", resourceCulture);
    }
}

```

In der Hilfsklasse **Resources** kommt erwartungsgemäß ein Memberobjekt der Klasse **ResourceManager** zum Einsatz (vgl. Abschnitt 23.2):

```
private static global::System.Resources.ResourceManager resourceMan;
```

Wir importieren in unserer Quellcodedatei per **using**-Direktive den Namensraum, den die Entwicklungsumgebung für die Hilfsklasse definiert hat:

```
using MPC.Properties;
```

Nun sind die Ressourcen in unserer Klassendefinition sehr bequem verwendbar:

```

Text = Resources.TitelText;

count = new Button();
Bitmap bmp = Resources.Count;
bmp.MakeTransparent(bmp.GetPixel(1, 1));
count.Image = bmp;

reset = new Button();
bmp = Resources.Reset;
bmp.MakeTransparent(bmp.GetPixel(1, 1));
reset.Image = bmp;

```

Bei der nächsten Projekterstellung resultiert ein Exe-Assembly mit integrierten Ressourcen, das ohne zusätzliche Dateien ablauffähig ist.

Den vollständigen Quellcode des Beispielsprogramms finden Sie im Ordner

...\\BspUeb\\Ressourcen\\Multi Purpose Counter\\vs

23.4 Übungsaufgaben zu Kapitel 23

1) Integrieren Sie bei unserem Editorprojekt die Bitmap-Ressourcen in das Assembly.

24 Interoperabilität mit Software-Altlasten

Durch etliche Klassen aus den Namensräumen **System.Runtime.InteropServices** bietet das .NET-Framework eine gute Integration von traditionellen Funktions-DLLs (z.B. mit dem Windows-API) und von Komponenten aus dem Component Object Model (COM). Diese Software-Altlasten bezeichnet man auch als **nicht-verwalteten Code** (engl.: *unmanaged code*), weil sie nicht innerhalb der .NET – Common Language Runtime ausgeführt werden und damit nicht von deren Sicherheitsmodell, Garbage Collection etc. profitieren. In diesem Abschnitt kann nur ein erster Eindruck von der insgesamt recht komplexen Thematik vermittelt werden.

24.1 Funktionen in nativen DLLs nutzen

Obwohl das .NET-Framework eine ziemlich vollständige Kapselung des Windows-APIs liefert, kommt doch gelegentlich der direkte Aufruf einer API-Funktion und damit die Verwendung einer Windows-DLL in Frage. Eine komplette Referenz zum Windows-API bietet Microsoft auf der folgenden Webseite

[http://msdn.microsoft.com/en-us/library/aa383749\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa383749(VS.85).aspx)

Außerdem existieren zahllose weitere native DLLs mit interessanten Funktionen.

Die **Platform Invocation Services** (Kurzname: *PInvoke*) in der CLR erlauben die Nutzung von Funktionen in DLLs mit nicht-verwalteten Code. Im Vergleich zur Verwendung von .NET - Klassen muss man dabei als Nachteile akzeptieren:

- Verzicht auf Objektorientierung, relativ unbequeme Funktionsaufrufe
- Reduzierte Sicherheit durch unverwalteten Code
- Mögliche Performanzprobleme
Neben Fixkosten von ca. 10 - 30 x86-Instruktionen pro Aufruf einer nativen Win32-Funktion schlägt noch der Aufwand der Parameter-Konvertierung (siehe Abschnitt 24.1.2) zu Buche.⁷⁶

24.1.1 DLLs importieren

In C# wird eine nicht verwaltete DLL-Funktion folgendermaßen genutzt:

- Man schreibt einen Methodenprototypen mit den Modifikatoren **static** und **extern**.
- Dieser Methodendefinition wird das Attribut **DllImport** aus dem Namensraum **System.Runtime.InteropServices** vorangestellt, wobei der Attribut-Konstruktor den DLL-Dateiname als Parameter erhält.

Im folgenden Beispiel wird auf diese Weise die Funktion **MessageBox()** aus der Win32-API-DLL **user32.dll** aufgerufen:

```
using System;
class Win32MessageBox {
    [System.Runtime.InteropServices.DllImport("user32.dll")]
    static extern int MessageBox(int hWnd, string text, string titel, int typ);

    static void Main() {
        MessageBox(0, "MessageBox - Funktion aus user32.dll", "Win32-API", 0);
    }
}
```

⁷⁶ Die Angaben stammen von der folgenden MSDN-Webseite:
<http://msdn.microsoft.com/en-us/library/ms235282.aspx>

Das Ergebnis:



24.1.2 Parameter-Marshaling

PInvoke sorgt automatisch für die richtige Übergabe von einfachen Parameter-Datentypen, wobei wir beim Aufruf einer externen Funktion natürlich mit den gewöhnlichen .NET - Datentypen arbeiten können. Im obigen Beispiel werden die **int**-Parameter unverändert durchgereicht, während die **String**-Objekte in null-terminierte C-Strings (vgl. z.B. Baltes-Götz 2003) konvertiert werden müssen, wobei unter Win9x ANSI-Code (Win32-Datentyp **LPSTR**) und ab Windows NT Unicode (Win32-Datentyp **LPWSTR**) verwendet wird.

Für die erforderlichen Typanpassungen enthält die CLR einen so genannten **Marshaler**, der die wichtigsten Datentypen aus dem Windows-API und dem Component Object Model (COM) kennt. Bei Typen mit derselben Speicherdarstellung in beiden Welten ist *kein* Marshaling erforderlich (z.B. bei den Typen **int** und **double**). Die folgende Tabelle zeigt wichtige Typzuordnungen:⁷⁷

Win32-Headerdatei wtypes.h	Datentyp in C/C++	Datentyp im Common Type System von .NET
HANDLE	void *	IntPtr, UIntPtr
BYTE	unsigned char	Byte
SHORT	short	Int16
WORD	unsigned short	UInt16
INT	int	Int32
UINT	unsigned int	UInt32
LONG	long	Int32
BOOL	long	Boolean
DWORD	unsigned long	UInt32
ULONG	unsigned long	UInt32
CHAR	char	Char
LPSTR	char *	String [in], StringBuilder [in, out]
LPCSTR	const char *	String
LPWSTR	wchar_t *	String [in], StringBuilder [in, out]
LPCWSTR	const wchar_t *	String
FLOAT	float	Single (Alias: Float)
DOUBLE	double	Double

Beim Aufruf der API-Funktion **GetWindowsDirectory()** (aus der Datei **kernel32.dll**) bildet der Marshaler den ersten Parameter (Typ: **StringBuilder**) ab und reicht den zweiten (Typ: **int**) unverändert durch:

⁷⁷ Quelle: <http://msdn.microsoft.com/de-de/library/ms235282%28VS.80%29.aspx>

```

using System;
using System.Text;
using System.Runtime.InteropServices;
class SimpleMarshaling {
    [DllImport("kernel32.dll")]
    static extern int GetWindowsDirectory(StringBuilder sb, int maxChars);
    static void Main() {
        StringBuilder s = new StringBuilder(256);
        GetWindowsDirectory(s, 256);
        Console.WriteLine("Via Win32-Aufruf:\t"+s);
        Console.WriteLine("Via Environment:\t" +
            Environment.GetEnvironmentVariable("SystemRoot"));
    }
}

```

Auch zum Ermitteln des Windows-Ordners braucht man nicht unbedingt das Windows-API:



Von der .NET - Klasse **Environment** ist diese Information über die Methode **GetEnvironmentVariable()** ebenfalls zu erhalten.

Wird eine .NET-Klasse oder -Struktur an eine DLL-Funktion übergeben, ist die Anordnung der Instanzvariablen beim Parametertransfer relevant. Mit dem Argument **LayoutKind.Sequential** des Attributs **StructLayout** wird die .NET - Runtime angewiesen, die Instanzvariablen in der Definitionsreihenfolge anzuordnen:

```
[StructLayout(LayoutKind.Sequential)]
```

Ohne Layout-Information wird das Marshaling bei einer Klasse oder Struktur per Laufzeitfehler verweigert. Im folgenden Beispiel aus Drayton et al. (2003, S. 211) wird ein .NET - Objekt an die API-Funktion **GetSystemTime()** als Parameter übergeben:

```

using System;
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Sequential)]
class SystemTime {
    public ushort wYear = 0;
    public ushort wMonth = 0;
    public ushort wDayOfWeek = 0;
    public ushort wDay = 0;
    public ushort wHour = 0;
    public ushort wMinute = 0;
    public ushort wSecond = 0;
    public ushort wMilliseconds = 0;
}

class ClassMarshaling {
    [DllImport("kernel32.dll")]
    static extern void GetSystemTime(SystemTime t);

    static void Main() {
        SystemTime t = new SystemTime();
        GetSystemTime(t);
        Console.WriteLine("Datum: " + t.wDay + "." + t.wMonth + "." + t.wYear);
    }
}

```

Bei der potentiell kniffligen Formulierung der Parameterlisten von Windows-API - Aufrufen helfen etliche Informationsquellen, z.B. das Buch von Bayer (2006, S. 33ff) und die folgende Webseite:

<http://www.pinvoke.net>

24.1.3 Windows-Fehlercode ermitteln

Viele Windows-Funktionen signalisieren den (Miss)erfolg bei der Ausführung über einen boole-
schen Rückgabewert. Wenn Sie darüber hinaus bei negativem Ausgang am Windows-Fehlercode
interessiert sind, müssen Sie ...

- im **DLLImport**-Attribut der Windows-Funktion den Konstruktor-Parameter **SetLastError**
auf den Wert **true** setzen,
- nach dem Funktionsaufruf die statischen Methode **GetLastWin32Error()** der Klasse
Marshal (im Namensraum **System.Runtime.InteropServices**) nutzen.

Im folgenden Beispiel wird die API-Funktion **MoveFile()** zum Umbenennen von Dateien benutzt,
wobei die Ursache für einen Fehler (z.B. Quelle nicht vorhanden, Zugriff verboten) festgestellt
werden soll:

```
using System;
using System.Runtime.InteropServices;

class NativeDLLs {
    [DllImport("kernel32.dll", SetLastError = true)]
    static extern bool MoveFile(string quelle, string ziel);

    static void Main() {
        string quelle = @"C:\Windows\System32\notepad.exe";
        string ziel = @"C:\Windows\System32\notebad.exe";
        if (!MoveFile(quelle, ziel))
            Console.WriteLine("Fehler " + Marshal.GetLastWin32Error() +
                              " bei MoveFile");
        else
            Console.WriteLine("MoveFile erfolgreich");
    }
}
```

Beim Versuch, mit den Rechten eines Standardbenutzers eine Datei in **C:\Windows\System32** um-
zubenennen, resultiert erwartungsgemäß der Windows-Fehler 5 (Fehler beim Zugriff):

Fehler 5 bei MoveFile

24.2 Kooperation mit der COM-Technologie

Microsoft hat beim .NET – Design großen Wert auf die bidirektionale Interoperabilität mit der
Component Object Model – Technologie gelegt, in die große Investitionen der Softwarebranche
geflossen sind und teilweise noch weiter fließen. Wenngleich für neue Projekte nachdrücklich die
.NET – Plattform empfohlen wird, ist die COM-Technik nach wie ein fester Bestandteil der aktuel-
len Windows-Betriebssysteme, und es ist noch eine längere Phase der friedlichen Koexistenz von
COM (inklusive seiner Weiterentwicklungen DCOM sowie COM+) und .NET zu erwarten.

In diesem Abschnitt soll ein erster Eindruck von der traditionellen Windows-Komponententechno-
logie vermittelt werden. Wir konzentrieren uns auf die *Verwendung* von COM-Komponenten in
.NET - Projekten. Über die ebenfalls mögliche *Entwicklung* von COM-Komponenten in C# oder
anderen .NET – Sprachen informieren z.B. Troelsen (2001) und MacBeth (2004).

24.2.1 Einige Begriffe und Ideen aus der COM-Technologie

Mit der COM-Technologie wird die Entwicklung von

- objektorientierter und
- auf binärer Ebene programmiersprachenunabhängig wiederverwendbarer

Software ermöglicht.

Wie in Abschnitt 24.1 zu sehen war, können auch die Funktionen in klassischen DLLs (z.B. mit dem Windows-API), mit gewissem Aufwand von verschiedenen Programmiersprachen genutzt werden. Im Vergleich zu dieser Technologie glänzt das Component Object Model aber durch ein objektorientiertes Programmiermodell mit Schnittstellen (interfaces), Eigenschaften (properties), Methoden und Ereignissen.

Wiederverwendbare Klassen finden sich auch z.B. in einer C++ - Klassenbibliothek, doch sind diese nur in C++ - Programmen nutzbar. Demgegenüber entstehen mit der COM-Technologie Softwarekomponenten, die von Anwendungen in beliebigen Programmiersprachen zu nutzen sind. Außerdem herrscht natürlich auch beim *Erstellen* solcher Komponenten weitgehende Freiheit in der Wahl der Programmiersprache.

In der Variante **DCOM** (*Distributed Component Object Model*) sind auch Zugriffe über Netzwerke möglich. Die mit Windows 2000 eingeführte Erweiterung **COM+** unterstützt Unternehmenslösungen mit Transaktionstechnik.

Die ausgeprägt komponentenorientierte .NET-Plattform kann in vielerlei Hinsicht als Weiterentwicklung und Vereinfachung der COM-Technologie aufgefasst werden. Durch die Fülle technischer Details im weiteren Verlauf von Abschnitt 24.2.1, die hoffentlich zur Konkretisierung und Veranschaulichung der COM-Technik beitragen, sollten Sie sich nicht beunruhigen lassen. Bei der hauptsächlich wichtigen Verwendung von COM-Objekten in C# - Programmen ist von COM-plikationen und C++ - Pointer-Akrobatik nichts mehr vorhanden.

24.2.1.1 COM-Server

Eine COM-Komponente (synonym: ein COM-Objekt), das Dienste anbietet, wird als **COM-Server** bezeichnet. Es sind folgende Typen zu unterscheiden:

- **In-Process-Server**
Sie residieren in einer DLL-Datei und werden im Adressraum des Klienten ausgeführt, so dass der Aufruf einer In-Process-Server - Methode kaum aufwendiger ist als der Aufruf einer klienteneigenen Methode.
- **Out-Of-Process-Server** (*Local Server*)
Sie residieren in einer EXE-Datei und werden in einem eigenen Adressraum ausgeführt, so dass sie bei einem Crash den Klienten nicht in Mitleidenschaft ziehen. Durch die erforderliche Inter-Prozess - Kommunikation wird allerdings die Performanz erheblich beeinträchtigt.
- **Remote Server**
Die mit Windows NT eingeführte DCOM-Softwarearchitektur macht es möglich, Out-Of-Process-Server auf einem fremden Rechner im Netzwerk ausführen zu lassen.

Jede COM-Klasse besitzt eine global eindeutige Kennzeichnung durch eine 128-Bit - Zahl (GUID, *Globally Unique Identifier*), die als CLSID (*Class ID*) bezeichnet wird.

Eine DLL- oder EXE-Datei kann einen oder auch mehrere COM-Server enthalten. Für den Klienten ist es irrelevant, ob ein COM-Server im selben Prozess existiert, in einem anderen Prozess derselben Maschine oder auf einem anderen Rechner.

24.2.1.2 COM-Schnittstellen und -Typbibliotheken

Gegenüber dem Nutzer einer Komponente (gemeint sind Softwareentwickler) werden im guten objektorientierten Sinn alle Implementierungsdetails verborgen (perfekte Datenkapselung). Er muss nur die in einer **Typbibliothek** veröffentlichten **Schnittstellen** (engl.: *Interfaces*) der Komponente kennen, die für jeweils mehrere Methoden einen Namen, eine Parameterliste und einen Rückgabe-

wert definieren. Ein COM-Server kann eine oder auch mehrere Schnittstellen implementieren. Die **Metadaten** der Typbibliothek befinden sich entweder in einer separaten Datei (mit der Namensendung **.tlb**) oder auch innerhalb der DLL- oder EXE-Datei mit der COM-Klasse (speziell beim Erstellen des Servers mit Visual Basic). Wie in Abschnitt 24.2.2.3 zu sehen sein wird, kann eine Entwicklungsumgebung die Verwendung eines COM-Servers über die zugehörige Typbibliothek sehr vereinfachen. Auch die Typbibliothek zu einer COM-Klasse besitzt eine GUID.

COM-Schnittstellen ähneln stark den .NET – Gegenständen, die wir in Abschnitt 8 behandelt haben, doch gibt es auch einige gleich zu beschreibende Besonderheiten. So besitzt jede COM-Schnittstelle (wie auch jede COM-Klasse, siehe oben) eine global eindeutige Kennzeichnung durch eine 128-Bit - Zahl (GUID), die als IID (**Interface ID**) bezeichnet wird.

Jedes COM-Objekt implementiert die Schnittstelle **IUnknown** mit den folgenden Methoden:

IUnknown-Methode	Beschreibung
QueryInterface()	Ein COM-Klient kann eine Referenz (einen Zeiger) auf eine andere Schnittstelle des Objekts anfordern. Welche Schnittstellen ein Objekt implementiert, muss der Klient dabei wissen.
AddRef()	Erhöht den Referenzzähler eines COM-Objekts, sobald ein Klient eine Schnittstellenreferenz erhält
Release()	Sobald ein Klient einen Schnittstellen-Zeiger nicht mehr benötigt, muss er ihn per Release() freigeben. Dabei wird der Referenzzähler des Objekts vermindert. Liegen keine Referenzen mehr vor, entfernt sich das COM-Objekt aus dem Speicher.

Jedes COM-Interface stammt von **IUnknown** ab, so dass beim Implementieren einer beliebigen Schnittstelle auch die drei beschriebenen Methoden einbezogen sind.

24.2.1.3 Aufruf einer COM-Methode

In der (z.B. für C++ - Programmierer noch unmittelbar erfahrbaren) COM-Technik wird beim Zugriff auf eine Methode durch einen COM-Klienten ein Zeiger (eine Referenz) auf das zugehörige Interface benötigt. Durch die API-Funktion **CoCreateInstance()** wird ein COM-Objekt über seine CLSID und ein von diesem Objekt implementiertes Interface über seine IID angesprochen, wobei das Objekt erstellt oder sein Referenzzähler erhöht wird, z.B.:

```
IDmEuroConv* pDEC;
CoCreateInstance(
    CLSID_DmEuroConv,
    NULL,
    CLSCTX_INPROC_SERVER,
    IID_IDmEuroConv,
    (void**) &pDEC);
```

Hier stehen CLSID_DmEuroConv und IID_IDmEuroConv für Variablen mit den benötigten GUIDS. Wir werden in Abschnitt 24.2.2.2 einen COM-Server namens DmEuroConv erstellen, der ein Interface namens IDmEuroConv implementiert, das über die **IUnknown**-Methoden hinaus noch die Konvertierungsmethoden ToDM() und ToEuro() zur (Euro ↔ DM) - Wandlung beinhaltet. Ein gelungener **CoCreateInstance()**-Aufruf liefert einen Pointer zum Interface, und man kann die gewünschte Methode aufrufen, z.B.:

```
double dm;
pDEC->ToDm(10.0, &dm);
```

Weil jedes COM-Interface von **IUnknown** abstammt, kann man sich per **QueryInterface()**-Aufruf (siehe oben) über die Existenz einer anderen Schnittstelle im Objekt informieren und ggf. einen entsprechenden Pointer erhalten.

24.2.1.4 Frühe und späte Bindung

Beim eben beschriebenen COM-Methodenaufruf durch einen C++ - Klienten wie auch bei der in Abschnitt 24.2.2.1 behandelten COM-Integration in das .NET - Framework spricht man von *früher Bindung*, weil der Compiler viele Informationen über die aufzurufenden Methoden besitzt, z.B. ihre Positionen im COM-Server kennt. Diese in der *Microsoft Interface Definition Language* (MIDL) formulierte Information steht dem C++ - Programm über inkludierte Dateien (oben nicht zu sehen) und dem .NET - Framework über eine Typbibliothek zur Verfügung.

Alternativ ist auch eine *späte Bindung* möglich, wobei Methoden aus dem COM-Interface **IDispatch** verwendet werden, um die interessierenden Methoden aufzurufen, von denen an Stelle der Speicheradresse lediglich der Name bekannt sein muss. Das späte Binden erhöht die Toleranz gegenüber Versionsunterschieden beim COM-Server, aber auch den Zeitaufwand beim Funktionsaufruf.

Microsoft empfiehlt für fast alle Anwendungsfälle die *frühe Bindung*⁷⁸, so dass wir uns darauf beschränken können. Wesentliche Vorteile der frühen Bindung sind:

- Bessere Performanz
- Typsicherheit
- IntelliSense-Unterstützung durch die Entwicklungsumgebungen

Weitere Informationen zur frühen bzw. späten Bindung finden Sie z.B. in Bayer (2006, S. 51), Liberty (2002, S. 597ff) oder Troelsen (2002, S. 622ff).

Unterstützt ein COM-Server auch die für das späte Binden erforderliche **IDispatch**-Schnittstelle, spricht Microsoft von einem *dualen* Interface.

24.2.1.5 Registrierung

Aus welcher ausführbaren Datei eine COM-Klasse zu laden ist, erfährt das COM-Laufzeitsystem aus der Windows-Registry (vgl. Abschnitt 19.2). Im Registry-Schlüssel⁷⁹

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID

findet sich für jede COM-Klasse ein nach ihrer CLSID benannter Unterschlüssel mit den erforderlichen Angaben. Für die in Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.** vorzustellende COM-Klasse wird z.B. der folgende Schlüssel verwendet:

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{B175D878-FEEE-4515-88CA-363F5AC07CCF}

Als Standardwert des ProgID-Teilschlüssels findet sich hier eine für Menschen lesbare Bezeichnung. Die beim Instantiieren eines COM-Objekts erforderliche CLSID kann der Klient aus der Registry ermitteln, sofern er die ProgID kennt und im Zweig HKCR unter diesem Namen ein Schlüssel mit der benötigten Information existiert, z.B.:

[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\ComSvrAtl.DmEuroConv]

In die ProgID sind im Beispiel die Namen der DLL-Datei und der COM-Klasse eingegangen. Während eine CLSID mit an Sicherheit grenzender Wahrscheinlichkeit eindeutig ist, kann man dies bei der zugehörigen ProgID nicht annehmen.

Dem TypeLib-Key ist u.a. zu entnehmen, wo die Metainformationen stecken, z.B.:

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\TypeLib\{5386F11F-21A1-4431-AAEF-DF7121F87E7E}

⁷⁸ Quelle: <http://support.microsoft.com/?scid=kb%3Ben-us%3B245115&x=12&y=7>

⁷⁹ Dieser Schlüssel ist auch ansprechbar über den Registry-Einstieg:

HKEY_CLASSES_ROOT\CLSID

Unterstützt eine COM-Klasse die späte Bindung, dann ist auch für jede implementierte Schnittstelle ein Interface-Key vorhanden, z.B.:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\Interface\{A47A69C4-FA52-4CE7-9323-08B195F4862F}
```

Mit dem Distributed Component Object Model wurde zur Rechteverwaltung noch die einer ausführbaren Datei (DLL oder EXE) zugeordnete, wiederum global eindeutige AppID samt Registry-Key eingeführt, z.B.:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\AppID\ComSvrAtl.DLL
```

Die Registry-Eintragungen zu einem COM-Objekt können über das Windows-Hilfsprogramm **regsvr32.exe** angestoßen werden, wobei die Komponente die Arbeit aber selbst zu erledigen hat und entsprechend vorbereitet sein muss, z.B.:

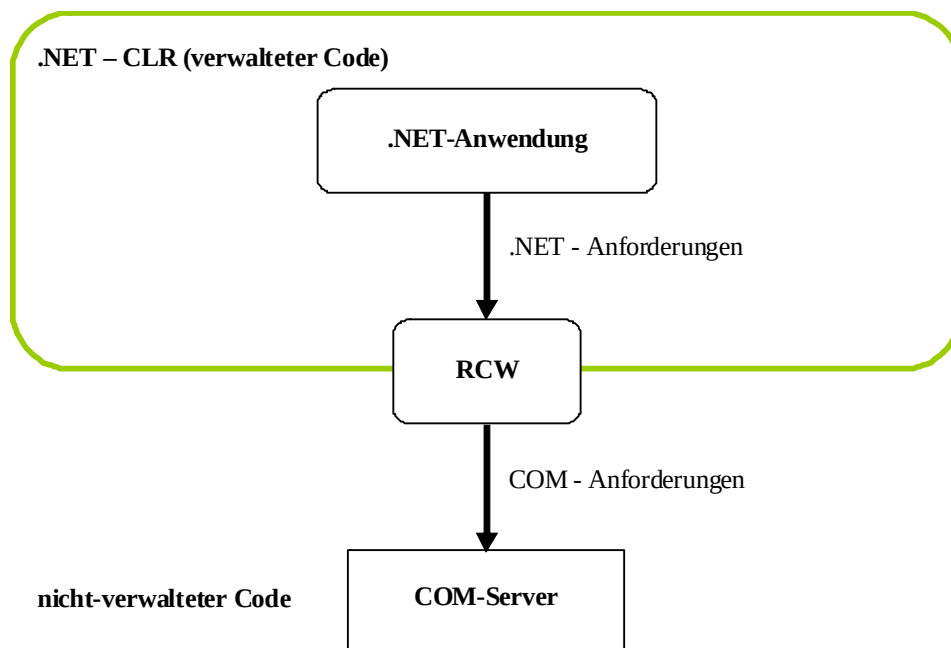
```
>regsvr32 ComSvrAtl.dll
```

Die fehlerträchtige Abhängigkeit der COM-Technik von zahlreichen Registry-Keys stellt einen zentralen Kritikpunkt dar, den das .NET-Framework überwunden hat.

24.2.2 COM-Server in .NET - Anwendungen nutzen

24.2.2.1 Proxy-Assembly mit Runtime Callable Wrapper

Um eine COM-Komponente für .NET - Software verfügbar zu machen, wird ein als **Runtime Callable Wrapper (RCW)** bezeichneter Proxy (Stellvertreter) eingesetzt, der Aufträge von .NET - Klassen entgegen nimmt, in COM-Anforderungen übersetzt und weiterleitet. Dies ist in folgender Abbildung schematisch dargestellt:



Die .NET – Anwendung kooperiert mit einem RCW-Objekt, welches seinerseits mit einem COM-Objekt zusammenarbeitet.

Wesentlicher Bestandteil des RCWs zu einer COM-Komponente ist ein DLL-Assembly, welches die Interfaces in der COM-Typbibliothek als .NET - Metadaten enthält. Zur Unterstützung der COM-Integration in das .NET - Framework sind Werkzeuge zur automatischen Erstellung solcher Proxy-Assemblies vorhanden. Im Windows-SDK ist das Hilfsprogramm

```
...\\Microsoft SDKs\\Windows\\v6.0A\\bin\\TlbImp.exe
```

enthalten, welches per Kommandozeilenparameter einige Kontrolle über das resultierende Assembly ermöglicht (z.B. Name, Signatur, Namensraum). Beim Einsatz einer .NET - Entwicklungs-

umgebung (wie Visual C# 2008 Express Edition oder SharpDevelop) erzeugt man jedoch sehr bequem ein Bibliotheks-Assembly mit dem RCW durch Hinzufügen einer Referenz auf den COM-Server (siehe Abschnitt 24.2.2.3).

24.2.2.2 COM-Server mit C++ erstellen

Der Einfachheit halber erzeugen wir einen winzigen COM-Server, der Methoden zur Währungsumrechnung ($\text{DM} \leftrightarrow \text{Euro}$) anbietet und dabei auf eine objektorientierte Programmierung verzichtet. Wir halten uns an Microsofts Empfehlung, für die Entwicklung von COM-Objekten in C++ bevorzugt die *Active Template Library* (ATL) zu benutzen.⁸⁰ Leider ist weder die ATL-Bibliothek noch die wichtigere Alternative MFC (Microsoft Foundation Classes in der kostenlosen Visual C++ 2008 Express Edition vorhanden, so dass die (für den C# - Kurs eher nebensächliche) Erstellung eines traditionellen COM-Servers mit dem Visual Studio 2008 Professional erledigt wird.

Beim Erstellen dieses Projekts habe ich einige Hinweise aus dem folgenden Tutorial

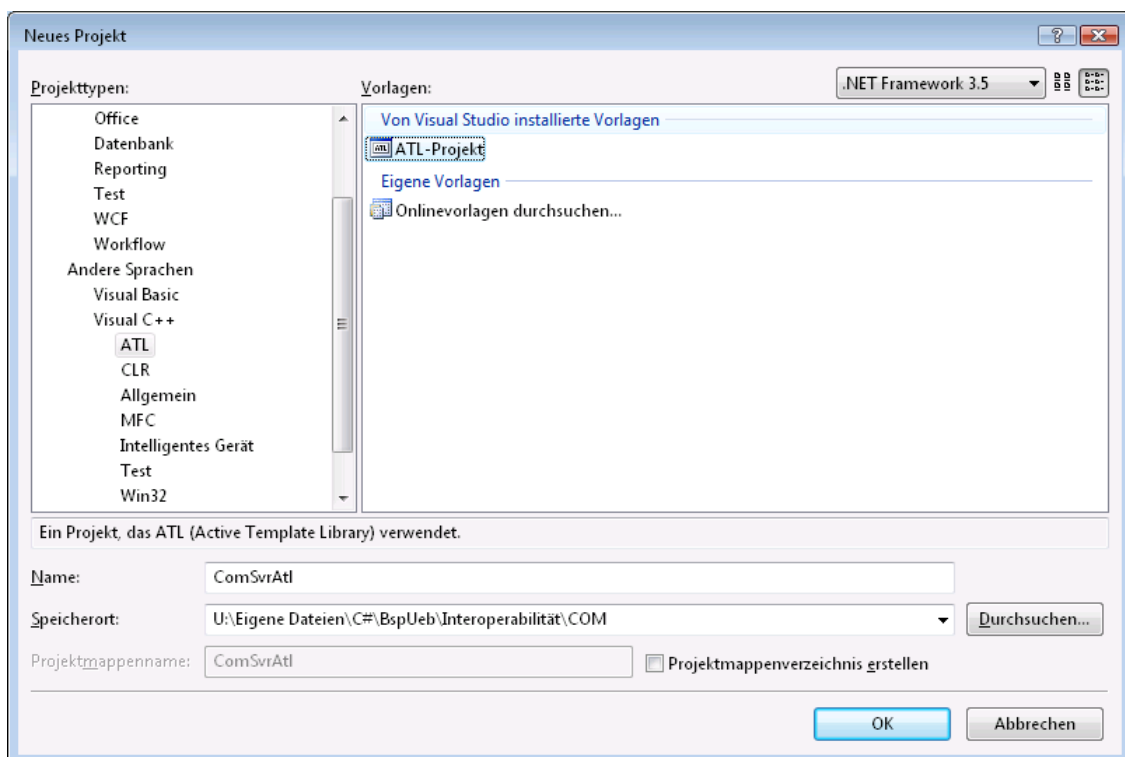
<http://www.codeproject.com/KB/COM/SimpleCOM.aspx>

von **Zhiwei Liu** übernommen, dem ich an dieser Stelle herzlich danken möchte.

Nach dem Menübefehl

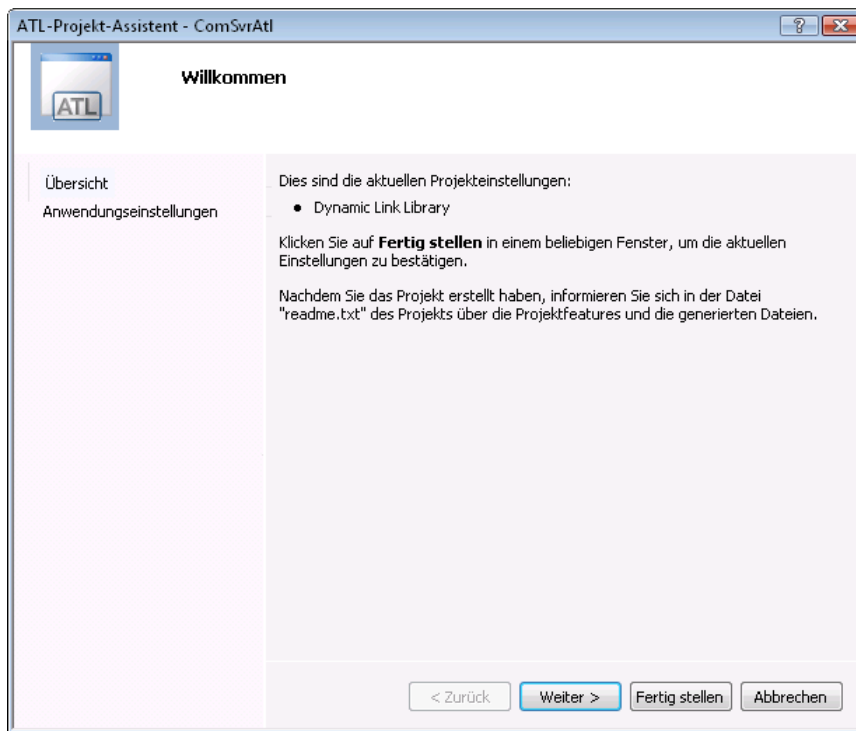
Datei > Neu > Projekt

wählen wir ein **ATL-Projekt**:

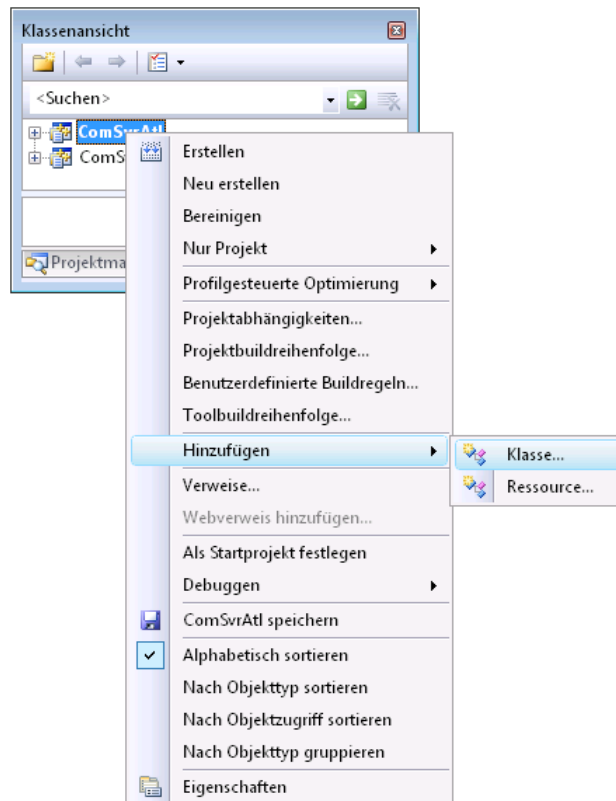


Im nächsten Dialog klicken wir auf **Fertigstellen**:

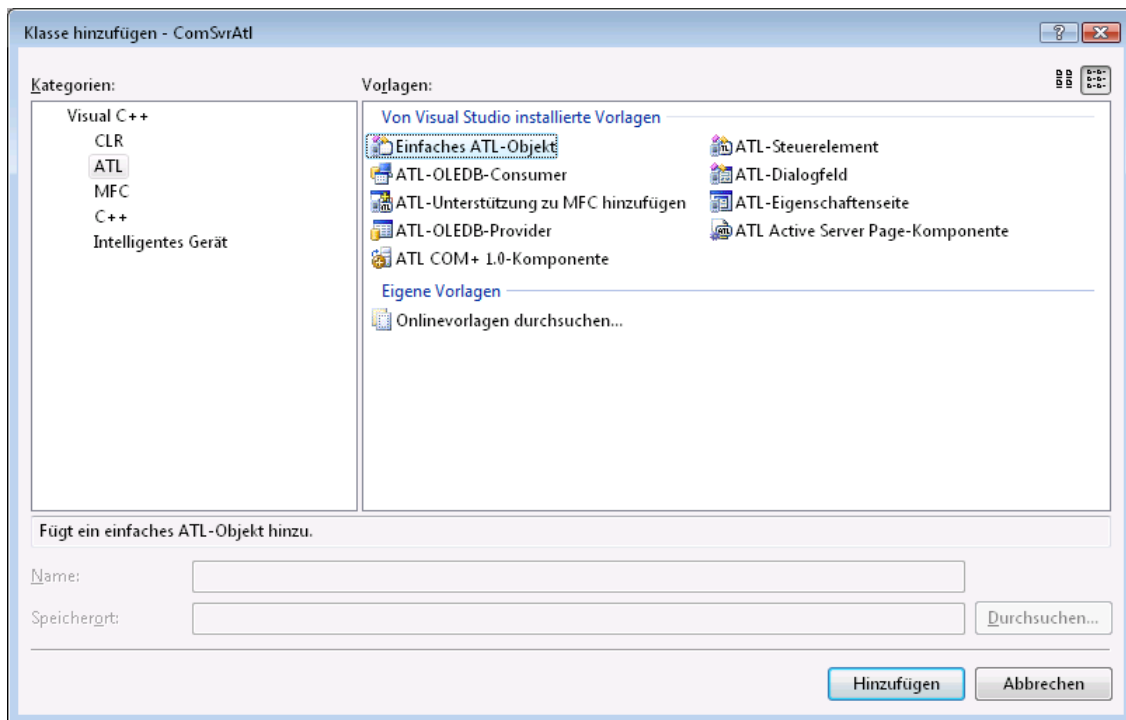
⁸⁰ Dieser Standpunkt ist den Webseiten <http://msdn.microsoft.com/en-us/library/3ax346b7.aspx> und <http://msdn.microsoft.com/en-us/library/bk8ytxz5.aspx> zu entnehmen.



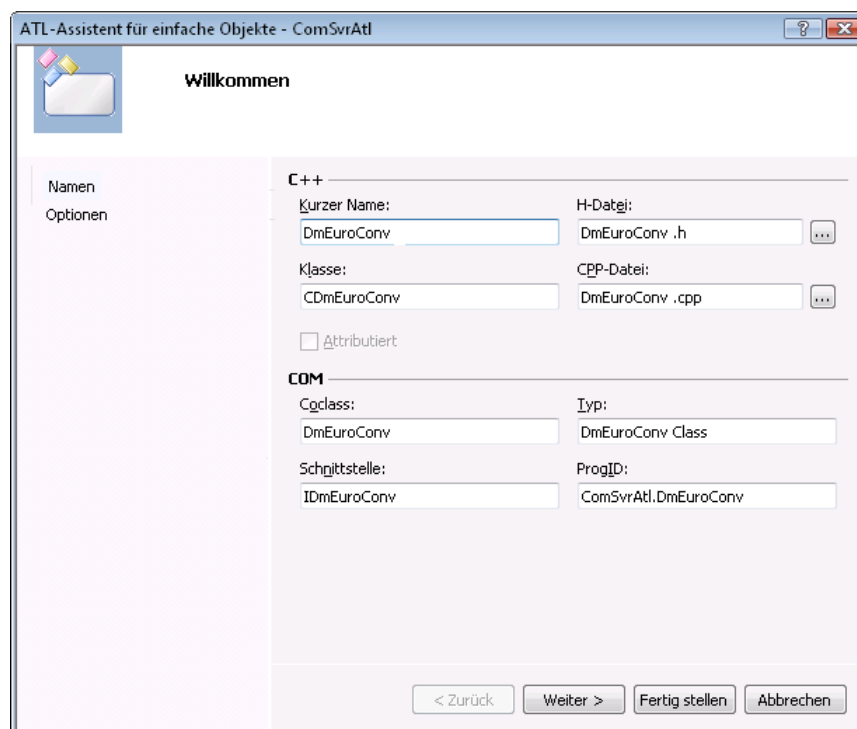
Im Projektmappenexplorer wählen wir die (notigenfalls über **Ansicht > Klassenansicht** eingeschaltete) Klassenansicht zum neuen Projekt und fügen eine Klasse hinzu:



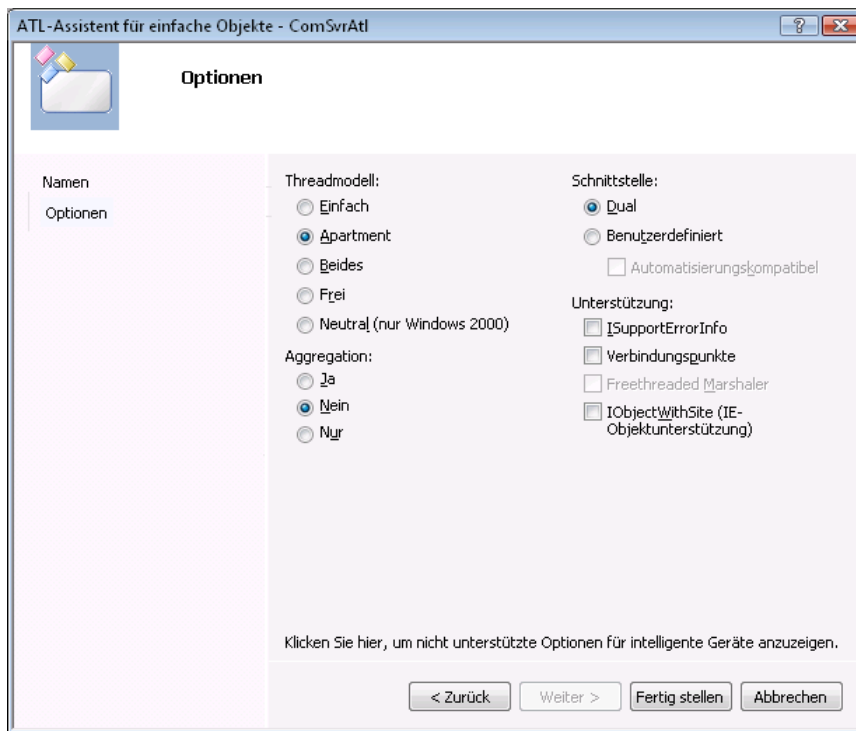
Als Vorlage für die neue Klasse verwenden wir **Einfaches ATL-Objekt**:



Mit dem ATL-Assistenten vereinbaren für die COM-Klasse den Namen `DmEuroConv` und für die COM-Schnittstelle den Namen `IDmEuroConv`:



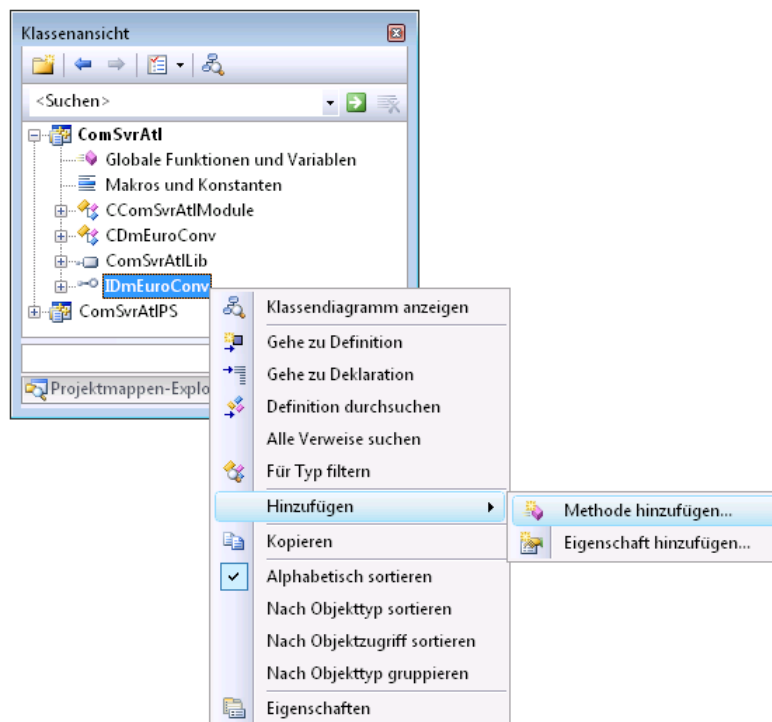
Nach einem Klick auf **Weiter** nehmen wir im nächsten Dialog



folgende Einstellungen vor:

- Schnittstellentyp **Dual**
Obwohl es für die Abschnitt 24.2.2.1 beschriebene Verwendung (frühes Binden, RCW) überflüssig ist, soll der COM-Server auch das späte Binden unterstützen, also das Interface **IDispatch** implementieren. Infolgedessen wird beim späteren Registrieren auch ein Interface-Key in der Windows-Registrierungsdatenbank entstehen.
- Keine **Aggregation**

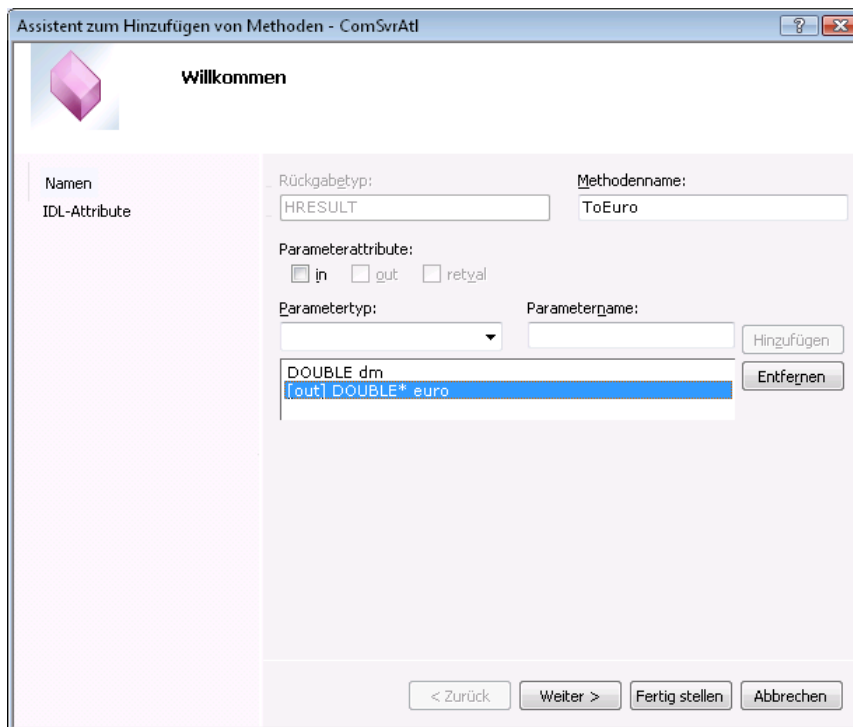
Nach dem **Fertigstellen** fügen wir per Klassenansicht zur Schnittstelle **IDmEuroConv** Konvertierungsmethoden hinzu:



Die erste Methode erhält ...

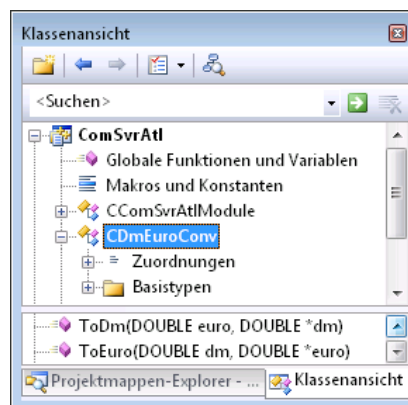
- den Namen `ToEuro()`
- den Eingabeparameter `dm` vom Typ **DOUBLE**
- den Ausgabeparameter `euro` vom Typ **DOUBLE***

Stellen Sie sich unter diesem Pointertyp der Programmiersprache C einen Referenzparameter vor.

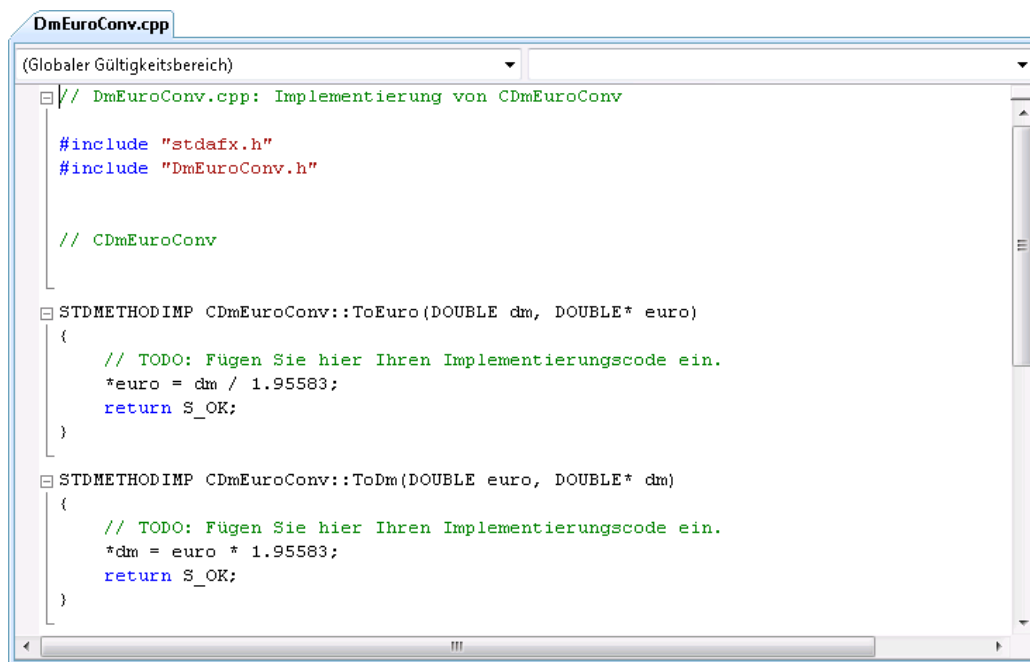


Als Rückgabtyp akzeptieren wir (nolens volens) den COM-Standardtyp **HRESULT**, der eine erfolgreiche oder gescheiterte Ausführung meldet. Wir quittieren mit **Fertigstellen** und definieren analog die Konvertierungsmethode `ToDm()`.

Bei markierter C++ - Klasse `CDmEuroConv` setzen wir in der Klassenansicht



einen Doppelklick auf eine der neuen Methoden, so dass die C++ - Quellcodedatei **DmEuroConv.cpp** im Editor geöffnet wird. Hier implementieren wir die beiden Methoden:



Mit dem Menübefehl **Erstellen > Projektmappe erstellen** oder mit dem Tastenbefehl **F6** wird das Erstellen des COM-Servers angefordert, wobei die Entwicklungsumgebung automatisch eine Registrierung des Servers versucht, die bei fehlenden Administratorrechten scheitern muss. Eventuell wollen Sie die DLL-Datei zu einem anderen Rechner kopieren und dort mit dem Windows-Hilfsprogramm **regsvr32.exe** registrieren lassen, z.B.:

>regsvr32 ComSvrAtl.dll

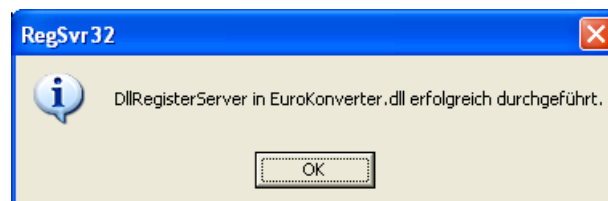
Wenn dieser Versuch trotz vorhandener Administratorrechte scheitert,



kommen u.a. folgende Ursachen in Frage:

- Auf dem Zielrechner ist kein Visual Studio 2008 Professional vorhanden, und folglich muss das *Microsoft Visual C++ 2008 Redistributable Package (x86)* installiert werden.⁸¹
- Das Visual Studio 2008 verwendet beim Erstellen per Voreinstellung die **Debug**-Konfiguration. Die resultierende DLL ist dann abhängig von Debug-Varianten andere Bibliotheken, die nur bei installiertem Visual Studio 2008 Professional vorhanden sind. Wählen Sie die **Release**-Konfiguration.

Eine erfolgreiche Registrierung wird so gemeldet:



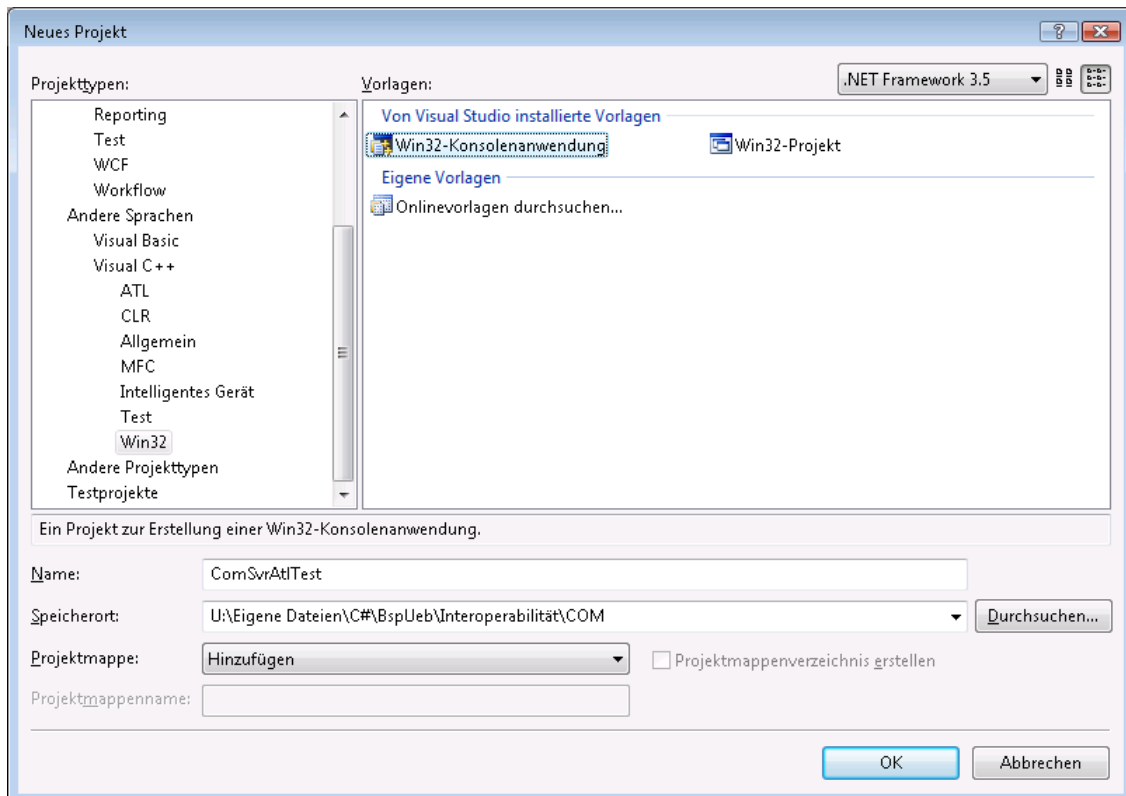
⁸¹ Verfügbar über die Webseite:

<http://www.microsoft.com/downloads/details.aspx?displaylang=de&FamilyID=9b2da534-3e03-4391-8a4d-074b9f2bc1bf>

In der Windows-Registrierungsdatenbank entstehen daraufhin die in Abschnitt 24.2.1.5 vorgestellten Schlüssel, wobei wir aus Platzgründen auf die Wiedergabe der Teilbäume verzichten. Diese sind zusammen mit dem kompletten ATL-Projekt zum COM-Server im folgenden Ordner zu finden:

...\\BspUeb\\Interoperabilität\\COM\\ComSvrAtl

Bevor wir den Einsatz des COM-Servers im .NET-Framework behandeln, soll noch seine Funktionsfähigkeit im Rahmen eines C++ - Konsolenprogramms überprüft werden, wobei die in Abschnitt 24.2.1.3 vorgestellte API-Technik zum Aufruf einer COM-Methode im Kontext demonstriert wird. Wir öffnen im Visual Studio 2008 Professional die Projektmappe mit dem COM-Server und fügen ein neues Projekt unter Verwendung der Vorlage **Win32-Konsolenanwendung** hinzu, z.B. mit dem Namen `ComSvrAtlTest`:



Diesmal soll aber keine neue Projektmappe angelegt, sondern das neue Projekt zur aktuellen Mappe hinzugefügt werden. Im **Willkommen**-Dialog des Assistenten lassen wir das Projekt gleich **fertig stellen**.

In der spontan zum Editieren geöffneten Quellcode-Datei `ComSvrAtlTest.cpp` werden zunächst per **include**-Direktive die vom MIDL-Compiler erzeugten Dateien `ComSvrAtl_i.*` mit den Interface-Definitionen zum COM-Server einbezogen:

```

#include "stdafx.h"
#include "../ComSvrAtl/ComSvrAtl_i.h"
#include "../ComSvrAtl/ComSvrAtl_i.c"
#include <iostream>
using namespace std;

int main() {
    HRESULT returnCode;
    IDmEuroConv* pDEC;
    returnCode = CoInitialize(0);
    if (SUCCEEDED(returnCode)) {
        returnCode = CoCreateInstance(
            CLSID_DmEuroConv,
            NULL,
            CLSCTX_INPROC_SERVER,
            IID_IDmEuroConv,
            (void**) &pDEC);
        if (SUCCEEDED(returnCode)) {
            double dm, euro;
            cout << "Euro <-> DM - Konvertierung\n\nEuro-Betrag:\t";
            cin >> euro;
            cin.get();
            pDEC->ToDm(euro, &dm);
            pDEC->Release();
            cout << "DM-Betrag:\t" << dm << endl;
        } else
            cout << "COM-Server konnte nicht instantiiert werden." << endl;
    } else
        cout << "COM konnte nicht initialisiert werden." << endl;
    cin.get();
    CoUninitialize();
    return 0;
}

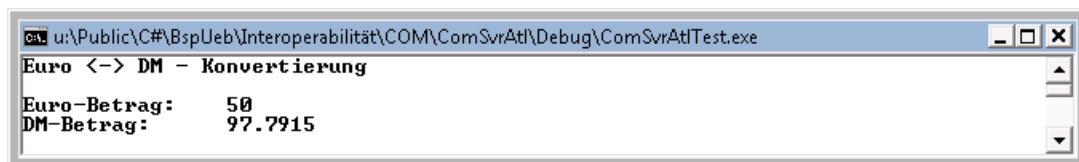
```

Die COM-Laufzeitumgebung muss mit der API-Funktion **CoInitialize()** initialisiert werden, und als sorgfältige C++ - Programmierer geben wir die belegten Ressourcen frühestmöglich mit der API-Funktion **CoUninitialize()** wieder frei. Mit der **IUnknown**-Methode **Release()** (siehe Abschnitt 24.2.1.2)

```
pDEC->Release();
```

geben wir den per **CoCreateInstance()** erworbenen Pointer zur Schnittstelle **IDmEuroConv** wieder frei, damit das COM-Objekt sich aus dem Speicher verabschieden kann, falls keine weiteren Nutzer vorliegen.

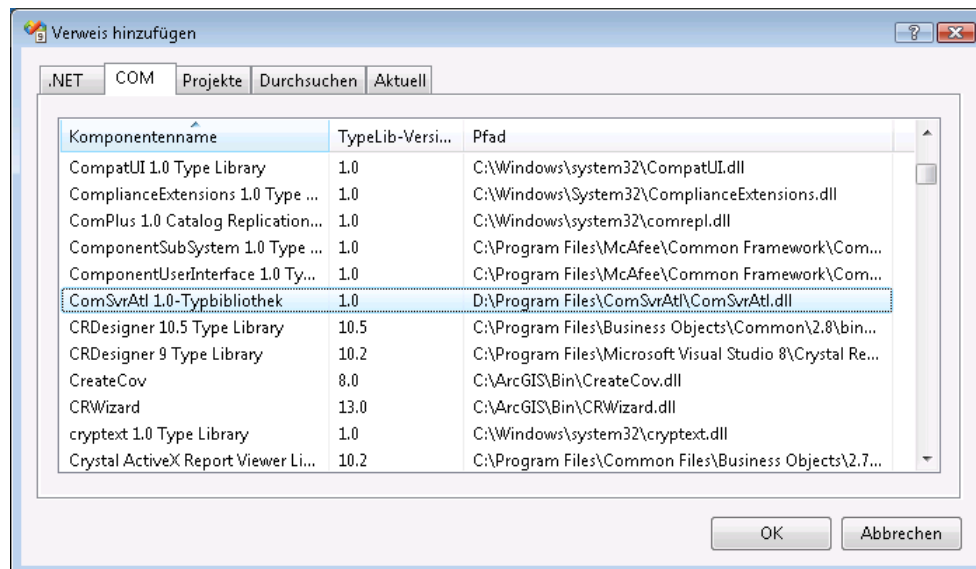
Das ausführbare Programm entsteht aufgrund der gewählten Projektmappenstruktur im Debug-Unterrordner des Server-Projekts:



Wie gleich zu sehen ist, macht die Nutzung des COM-Servers in einer .NET - Anwendung weniger Mühe.

24.2.2.3 COM-Server in C# verwenden

Wir legen ein neues Projekt nach der Verabredung in Abschnitt 9.1 (selbstgestrickte WinForms-Anwendung) an und fügen per Projektmappen-Explorer einen Verweis auf den COM-Server **DmEuroConv** in der Datei **ComSvrAtl.dll**, die ihren Namen vom Projekt übernommen hat:

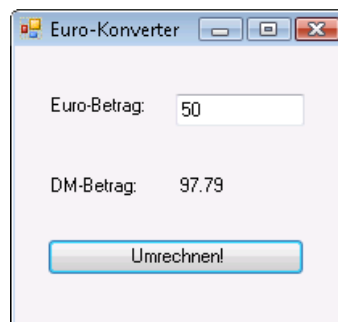


In der konkreten Situation wurde die DLL-Datei nach der temporären Übernahme von Administratorrechten zum Stationierungsort kopiert und per **regsvr32.exe** registriert.

Durch die Verweisaufnahme entsteht im Projekt-Unterordner **...\obj\Debug** das Proxy-Assembly **Interop.ComSvrAtlLib.dll** mit der Proxy-Klasse **BetragClass** im Namensraum **EuroKonverter**.

Wer mehr Kontrolle benötigt (z.B. beim Klassennamen oder Namensraum), der kann das in Abschnitt 24.2.2.1 beschriebene Windows-SDK - Werkzeug **TlbImp.exe** verwenden und auf das resultierende Proxy-Assembly einen gewöhnlichen .NET - Verweis setzen.

Im Quellcode für die folgende C# - Anwendung



sorgt fast nur die GUI-Gestaltung für Umfang:

```
using System;
using System.Windows.Forms;
using ComSvrAtlLib;

class COMClientCs : Form {
    Label lbErgebnis;
    TextBox tbEuro;
    DmEuroConvClass konverter = new DmEuroConvClass();

    public COMClientCs() {
        Height = 200; Width = 210;
        Text = "Euro-Konverter";
        FormBorderStyle = FormBorderStyle.FixedSingle;

        Label lbEuro = new Label();
        lbEuro.Text = "Euro-Betrag:";
        lbEuro.Left = 20; lbEuro.Top = 30; lbEuro.Width = 80;
        lbEuro.Parent = this;
```

```

Label lbDM = new Label();
lbDM.Text = "DM-Betrag:";
lbDM.Left = 20; lbDM.Top = 80; lbDM.Width = 80;
lbDM.Parent = this;

lbErgebnis = new Label();
lbErgebnis.Text = "";
lbErgebnis.Left = 100; lbErgebnis.Top = 80;
lbErgebnis.Parent = this;

tbEuro = new TextBox();
tbEuro.Left = 100; tbEuro.Top = 30; tbEuro.Width = 80;
tbEuro.Parent = this;

Button cbUmrechnen = new Button();
cbUmrechnen.Text = "Umrechnen!";
cbUmrechnen.Top = 120; cbUmrechnen.Left = 20; cbUmrechnen.Width = 160;
cbUmrechnen.Parent = this;
AcceptButton = cbUmrechnen;
cbUmrechnen.Click += new EventHandler(ButtonOnClick);
}

void ButtonOnClick(object sender, EventArgs e) {
    if (tbEuro.Text.Length > 0) {
        double euro;
        try {
            euro = double.Parse(tbEuro.Text);
        } catch {
            lbErgebnis.Text = "Falsche Eingabe!";
            return;
        }
        double dm;
        konverter.ToDm(euro, out dm);
        lbErgebnis.Text = dm.ToString("f2");
    }
}

[STAThread]
static void Main() {
    Application.EnableVisualStyles();
    Application.Run(new COMClientCs());
}
}

```

Der automatisch angelegte Namensraum zum RCW-Assembly wird importiert:

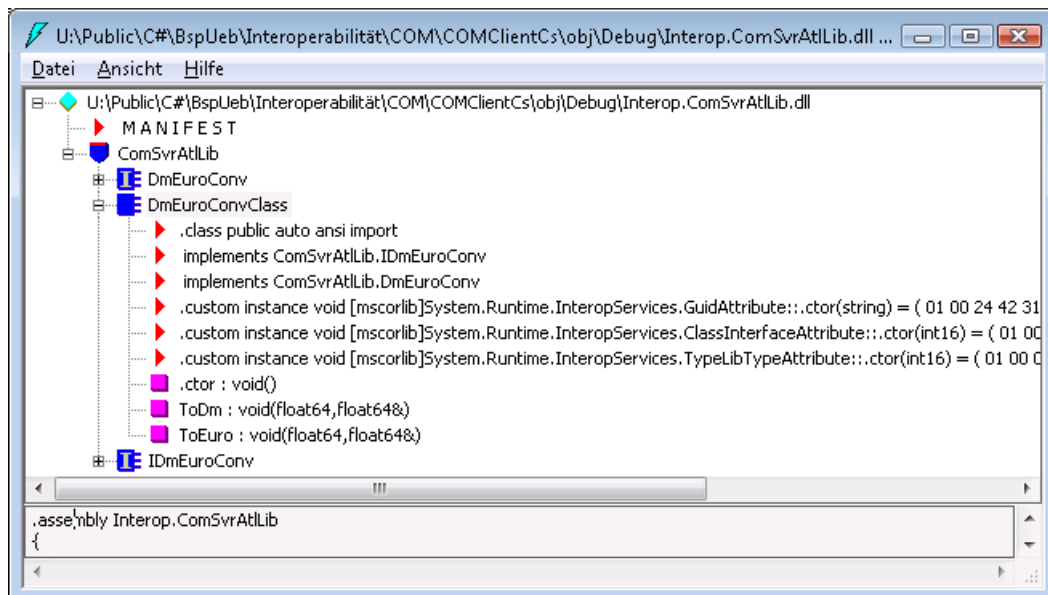
```
using ComSvrAtlLib;
```

Für die Währungskonvertierung wird ein Objekt der Proxy-Klasse `DmEuroConvClass` erzeugt, womit indirekt ein COM-Objekt entsteht:

```
DmEuroConvClass konverter = new DmEuroConvClass();
```

Um COM-Details wie den Erwerb und die Freigabe von Interface-Pointern müssen wir uns natürlich *nicht* kümmern.

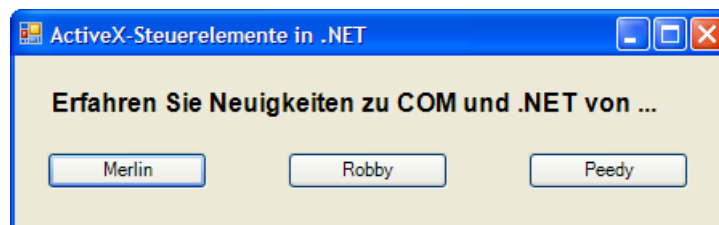
Bei Bedarf kann man ein Proxy-Assembly z.B. mit dem Windows-SDK - Hilfsprogramm **IL-Dasm.exe** inspizieren:



Unser COM-Klient nutzt in seiner `ButtonClick()`-Methode die `ToDm()`-Methode des COM-Objekts.

Wir haben bisher aus Gründen der Transparenz einen *selbst erstellten* COM-Server in einem .NET - Klienten genutzt. Allerdings sollten *neue* Komponenten möglichst in .NET – Technologie erstellt werden, und in der Praxis wird die Aufgabe dominieren, *vorhandene* COM-Server einzubinden. Wir üben diese praxisrelevante Aufgabe abschließend mit den per ActiveX-Steuerelement realisierten MS-Agent - Animationsfiguren. Eine ActiveX-Steuerelement ist ein spezieller COM-Server, der sich *auch* für den Einsatz bei Web-Applikationen eignet.

Ein C# - Projekt mit der folgenden Bedienoberfläche

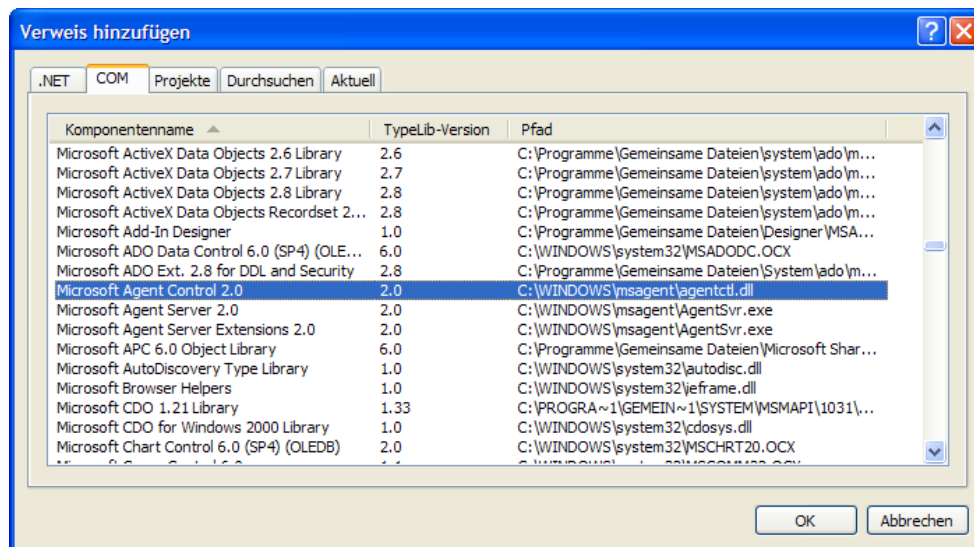


erhält in der Entwicklungsumgebung per Projektmappen-Explorer einen Verweis auf das ab Windows 2000 in der Regel installierte **Microsoft Agent Control 2.0**:⁸²

⁸² Nötigenfalls kann die Software von einem Microsoft-Server herunter geladen werden:

<http://www.microsoft.com/products/msagent/main.aspx>

Dort stehen über den bei einer Windows-Installation meist ausschließlich vorhandenen *Merlin* hinaus noch weitere Agenten zur Verfügung. Ab Windows 7 soll die Entwicklung und Unterstützung von MS-Agent allerdings eingestellt werden.



Nach dem Abschicken dieser Dialogbox entsteht im Projekt-Unterverzeichnis `...\obj\Debug` das Proxy-Assembly **Interop.AgentObjects.dll** mit dem RCW.

Die Formalklasse erhält Referenzinstanzvariablen vom Schnittstellentyp **IAgentCtlCharacterEx**, den der RCW aus der COM-Typbibliothek übernommen hat:

`IAgentCtlCharacterEx` merlin, robby, peedy;

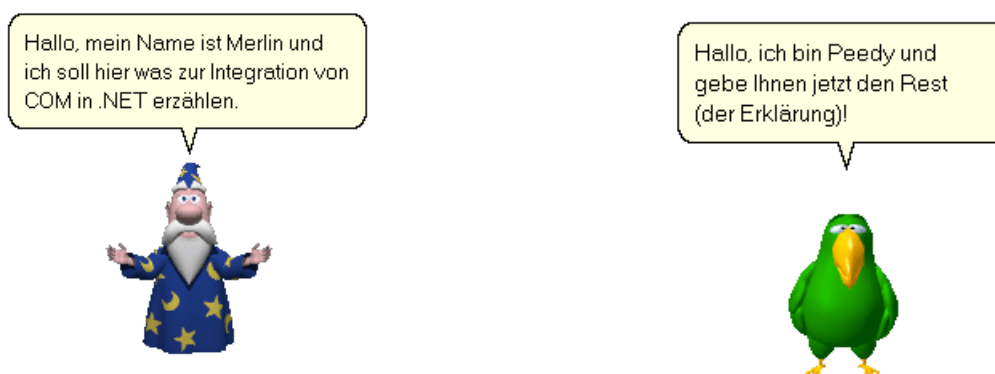
Ein Objekt der ebenfalls importierten Klasse **AgentClass** übernimmt die Aufgabe, drei Akteure (Merlin, Robby und Peedy) auf ihren Auftritt vorzubereiten:

```
AgentClass agent = new AgentObjects.AgentClass();
agent.Connected = true;
agent.Characters.Load("merlin", "merlin.acs");
merlin = agent.Characters["merlin"];
agent.Characters.Load("robby", "robby.acs");
robby = agent.Characters["robby"];
agent.Characters.Load("peedy", "peedy.acs");
peedy = agent.Characters["peedy"];
```

Die **Click**-Behandlungsmethode zu den Befehlsschaltern lässt die Puppen tanzen, z.B.:

```
merlin.Show(true);
merlin.MoveTo(200, 300, 1000);
merlin.Play("Greet");
merlin.Play("Announce");
merlin.Play("Explain");
merlin.Speak("Hallo, mein Name ist Merlin und ich soll hier was zur
Integration von COM in .NET erzählen.", null);
```

Mit etwas Liebe zum Detail gelingt vielleicht ein brauchbares Lernprogramm zur .NET - COM - Integration:



Andere COM-Server, die Sie vielleicht eher in eigenen Anwendungen verwenden wollen als die MS-Agenten, sind MS-Excel, MS - Internet Explorer, MS-Outlook, Matlab etc. Die Liste der COM-Einträge im Dialog **Verweis hinzufügen** (siehe oben) ist sehr lang!

25 Variable Ansichten

In diesem Abschnitt werden mit der Struktur- und der Listenansicht zwei mächtige Steuerelemente zur Anzeige (oder auch Erfassung) komplexer Daten behandelt. Weil sie oft (z.B. im Windows-Explorer) nebeneinander mit variabler Platzaufteilung eingesetzt werden, werden auch die Aufteilungssteuerelemente behandelt.

25.1 Aufteilungssteuerelemente

Leistungsfähige und multifunktionale Windows-Programme (z.B. Visual Studio, SharpDevelop, Outlook) bieten eine Bedienoberfläche mit mehreren rechteckigen Bereichen für bestimmte Funktionsgruppen, wobei der Benutzer durch Aufteilungssteuerelemente den verfügbaren Platz des Anwendungsfensters flexibel aufteilen kann. Sobald sich der Mauszeiger über einem Aufteilungssteuerelement befindet, erhält er eine neue Form (nämlich **Cursors.HSplit** bzw. **Cursors.VSplit**), und der Benutzer kann bei gedrückter Maustaste die Grenze verschieben.

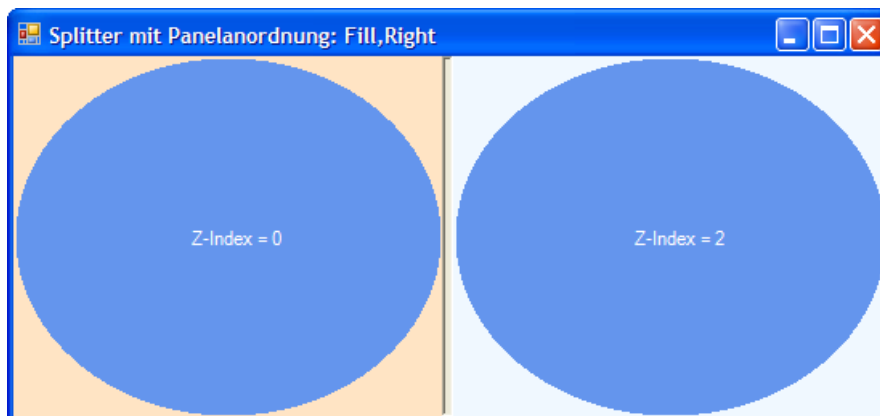
Neben dem älteren Steuerelement **Splitter** ist seit .NET 2.0 die bequemere und in der Regel zu bevorzugende Klasse **SplitContainer** verfügbar. Anschließend wird trotzdem auch die ältere Klasse vorgestellt, weil sie in vielen vorhandenen C# - Quellen anzutreffen ist und nur .NET 1.0 benötigt, also eine optimale Kompatibilität bietet

25.1.1 Splitter

Bei der Realisation von Mehrbereichsformularen mit den traditionellen **Splitter**-Steuerelementen sind folgende Gestaltungsmittel im Spiel:

- Geeignete Container-Steuerelemente (z.B. **Panel**)
- Die **Dock**-Eigenschaft der Steuerelemente
- Die Reihenfolge beim Einfügen der Steuerelemente (Z-Anordnung)

Um für zwei durch einen **Splitter** getrennte **Panel**-Steuerelemente



eine variable Platzaufteilung zu ermöglichen, kann man so vorgehen:

- Man fügt ein erstes **Panel**-Objekt mit **DockStyle.Fill** ein.
- Am rechten Rand wird ein **Splitter** angedockt, dem das ältere **Panel**-Element nach der in Abschnitt 9.4.4.3 beschriebenen Regel für **Dock**-Konkurrenten den Vorrang überlässt.
- Zuletzt wird ein weiteres **Panel**-Element rechts angedockt, vor dem die beiden älteren Steuerelemente (mit den kleineren Z-Indexnummern) zurückweichen.

Alternativ kann man die Konstruktion auch auf der linken Seite andocken. Beide Konstruktionen reagieren verschieden auf eine Änderung der horizontalen Containergröße: Diese wirkt sich jeweils auf das **Panel**-Objekt mit **DockStyle.Fill** aus.

Hinter dem obigen Fenster steckt folgendes Programm:

```

using System;
using System.Drawing;
using System.Windows.Forms;

class SplitterDemo : Form {
    SplitterDemo() {
        Text = "Splitter mit Panelanordnung: Fill, Right";
        ClientSize = new Size(300,200);

        Panel panel1 = new Panel();
        panel1.Parent = this;
        panel1.Dock = DockStyle.Fill;
        panel1.BackColor = Color.Bisque;
        panel1.Resize += new EventHandler(PanelOnResize);
        panel1.Paint += new PaintEventHandler(PanelOnPaint);

        Splitter split = new Splitter();
        split.Parent = this;
        split.Dock = DockStyle.Right;
        split.Width = 7;
        split.BorderStyle = BorderStyle.Fixed3D;

        Panel panel2 = new Panel();
        panel2.Size = new Size(100,100);
        panel2.Parent = this;
        panel2.Dock = DockStyle.Right;
        panel2.BackColor = Color.AliceBlue;
        panel2.Resize += new EventHandler(PanelOnResize);
        panel2.Paint += new PaintEventHandler(PanelOnPaint);
    }

    void PanelOnPaint(object sender, PaintEventArgs e) {
        Graphics g = e.Graphics;
        Panel pan = (Panel) sender;
        g.FillEllipse(Brushes.CornflowerBlue, 1, 1, pan.Width-2, pan.Height-2);
        for (int i = 0; i < Controls.Count; i++)
            if (sender == Controls[i])
                g.DrawString("Z-Index = "+i, Font, Brushes.White,
                    pan.Width / 2 - 24, pan.Height / 2 - 5);
    }

    void PanelOnResize(object sender, EventArgs e) {
        ((Panel) sender).Refresh();
    }

    [STAThread]
    static void Main() {
        Application.EnableVisualStyles();
        Application.Run(new SplitterDemo());
    }
}

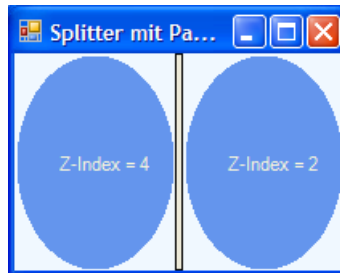
```

Für eine Aktualisierung der Anzeige bei einer Größenänderung der **Panel**-Objekte wird in diesem Programm über einen **Resize**-Handler gesorgt. In früheren Programmen haben wir bei neu definierten (aus **Form** abgeleiteten) Fensterklassen denselben Effekt über den Wert **true** für die geschützte(!) Eigenschaft **ResizeRedraw** erreicht.

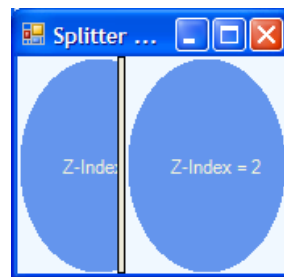
Es dürfte Ihnen keine Mühe machen, auch am linken Rand ein Paar aus **Splitter**- und **Panel**-Objekt einzubauen, so dass drei größenvariable Flächen entstehen, von denen die *mittlere* (mit **DockStyle.Fill**) auf horizontale Änderungen der Fenstergröße reagiert:



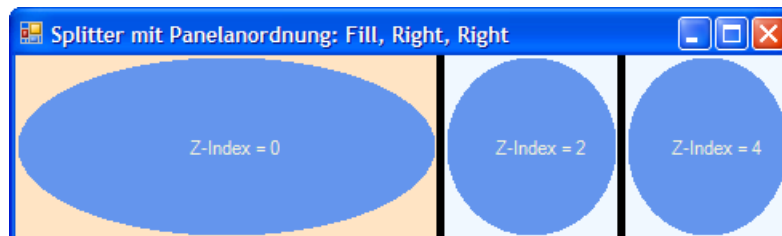
Bei zunehmender Platznot verschwindet irgendwann das **Panel** mit **DockStyle.Fill**:



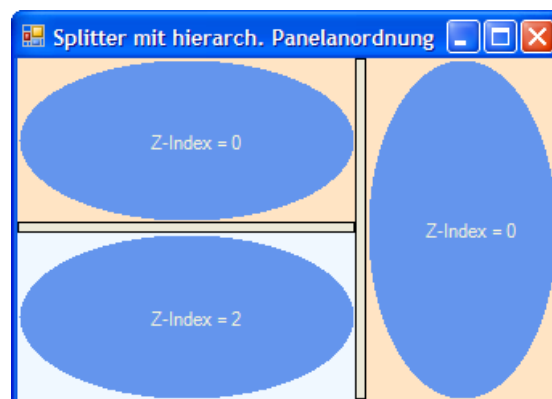
Bei weiter schrumpfender horizontaler Fensterausdehnung wird das **Panel** mit dem Z-Index 4 vom **Panel** mit dem Z-Index 2 überdeckt:



Es sind vielfältige Kombinationen von **Splitter**-separierten Steuerelementen mit den **DockStyle**-Werten **Fill**, **Left** und **Right** möglich, z.B. mit dem **DockStyle**-Wert **Fill** auf der linken Seite:



Durch Flächenaufteilungen auf mehreren Hierarchieebenen gewinnt man weitere Gestaltungsoptionen, z.B.:



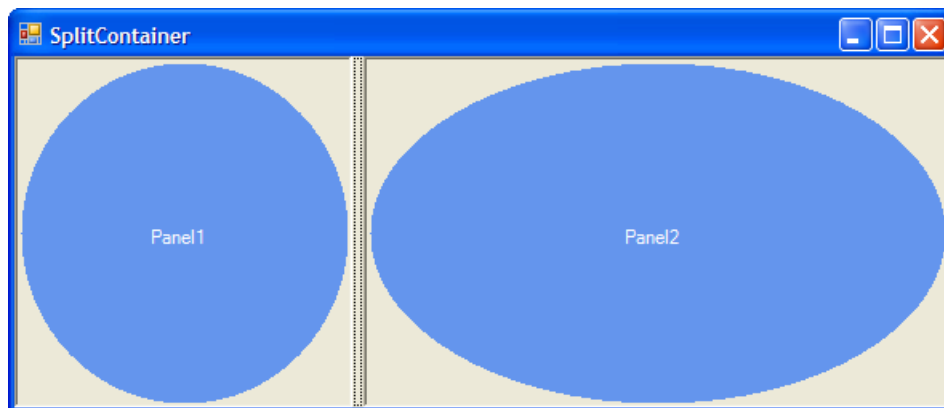
Von den Eigenschaften des **Splitter**-Steuerelements sollen noch erwähnt werden:

- **MinSize**
Diese Eigenschaft ermittelt oder setzt den minimalen Abstand zwischen dem Splitter und dem Container-Rand, an den er andockt ist.
- **MinExtra**
Diese Eigenschaft ermittelt oder setzt den minimalen Abstand zwischen dem Splitter und der dem Dock gegenüber liegenden Seite. Ist ein Splitter z.B. rechts andockt, geht es um den Abstand zum linken Containerrand oder bis zum nächstgelegenen links andockten Steuerelement.
- **SplitPosition**
Diese Eigenschaft ermittelt oder setzt den Abstand zwischen dem Splitter und dem Containerrand, an den er andockt ist.
- **BackColor**
Diese Eigenschaft ermittelt oder setzt die Hintergrundfarbe. Die Eigenschaft **ForeColor** hat *keinen* Effekt auf das Erscheinungsbild eines Splitters.
- **Width** bzw. **Height**
Breite eines vertikalen bzw. Höhe eines horizontalen Splitters
- **BorderStyle**
Mit dieser Eigenschaft wählt man einen Rahmentyp. Erlaubt sind die Werte **None**, **FixedSingle** und **Fixed3D** der Enumeration **BorderStyle**.

25.1.2 SplitContainer

Ein **SplitContainer** enthält zwei größenvariable Bereiche aus der Klasse **SplitterPanel** mit einer Aufteilungssteuerelement dazwischen, spart also Routineaufwand im Vergleich zur **Splitter-Panel** - Lösung aus dem letzten Abschnitt. Weil sich **SplitContainer** schachteln lassen und über Eigenschaften den Zugriff auf ihre Bestandteile erlauben, ist eine hohe Flexibilität vorhanden.

Beide Bereiche reagieren identisch auf Änderungen der Gesamtfläche des **SplitContainer**-Steuerelements, sofern keine Verhaltensunterschiede über Eigenschaften wie **FixedPanel** etc. (siehe unten) angeordnet werden. Zur Aufteilung eines Formulars in zwei größenvariable Bereiche



ist nur wenig Aufwand erforderlich:

```
using System;
using System.Drawing;
using System.Windows.Forms;
```

```

class SplitContainerDemo : Form {
    SplitContainer sc;

    SplitContainerDemo() {
        Text = "SplitContainer";
        ClientSize = new Size(300, 300);

        sc = new SplitContainer();
        sc.Parent = this;
        sc.Dock = DockStyle.Fill;
        sc.SplitterWidth = 7;
        sc.BorderStyle = BorderStyle.Fixed3D;
        sc.Panel1.Paint += new PaintEventHandler(PanelOnPaint);
        sc.Panel2.Paint += new PaintEventHandler(PanelOnPaint);
    }

    void PanelOnPaint(object sender, PaintEventArgs e) {
        Graphics g = e.Graphics;
        SplitterPanel pan = (SplitterPanel)sender;
        g.FillEllipse(Brushes.CornflowerBlue, 2, 2, pan.Width-7, pan.Height-7);
        if (pan == sc.Panel1)
            g.DrawString("Panel1", Font, Brushes.White,
                pan.Width / 2 - 24, pan.Height / 2 - 5);
        else
            g.DrawString("Panel2", Font, Brushes.White,
                pan.Width / 2 - 24, pan.Height / 2 - 5);
    }

    [STAThread]
    static void Main() {
        Application.EnableVisualStyles();
        Application.Run(new SplitContainerDemo());
    }
}

```

Im Unterschied zu der in Abschnitt 25.1.1 vorgestellten **Splitter**-Lösung aus .NET 1 klappt die Aktualisierung der Anzeige bei Größenänderungen auch ohne **Resize-EventHandler**.

Wichtige Eigenschaften des **SplitContainer**-Steuerelements:

- **Panel1, Panel2**
Mit diesen Eigenschaften vom Typ **SplitterPanel** erhält man Referenzen auf die beiden größenvariablen Bereiche.
- **Orientation**
Mit diesen Eigenschaften ermittelt oder setzt man die Aufteilungsrichtung, wobei die Werte **Horizontal** und **Vertical** der Enumeration **Orientation** erlaubt sind.
- **Panel1MinSize**
Diese Eigenschaft ermittelt oder setzt den minimalen Abstand des Splitters vom linken bzw. oberen Rand (je nach Orientierung).
- **Panel2MinSize**
Diese Eigenschaft ermittelt oder setzt den minimalen Abstand des Splitters vom rechten bzw. unteren Rand (je nach Orientierung).
- **SplitterDistance**
Diese Eigenschaft ermittelt oder setzt den Abstand zwischen dem Splitter und linken bzw. oberen Rand (je nach Orientierung).
- **Panel1Collapsed, Panel2Collapsed**
Mit diesen booleschen Eigenschaften kann man das zugehörige Panel verschwinden lassen bzw. seine Anwesenheit überprüfen.

- **FixedPanel**
Mit dieser Eigenschaft wird ermittelt oder festgelegt, welcher Bereich seine Größe bei einer Änderungen der Gesamtfläche des **SplitContainer**-Steuerelements beibehält, wobei die Werte **None**, **Panel1** und **Panel2** der Enumeration **FixedPanel** erlaubt sind.
- **IsSplitterFixed**
Mit diesen booleschen Eigenschaften wird ermittelt oder festgelegt, ob der Splitter fixiert ist.
- **SplitterWidth**
Diese Eigenschaft ermittelt oder setzt die Breite des Splitters.
- **BorderStyle**
Mit dieser Eigenschaft wählt man einen Rahmentyp für den Splitter. Erlaubt sind die Werte **None**, **FixedSingle** und **Fixed3D** der Enumeration **BorderStyle**.

25.2 Baumansicht

Die Präsentation von hierarchisch strukturierten Daten (z.B. Laufwerke und Verzeichnisse eines Rechners) in WinForms-Anwendungen wird durch die folgenden Klassen unterstützt:

- **TreeView**
Das als *Baum- oder Strukturansicht* bezeichnete Steuerelement zeigt eine hierarchisch organisierte Liste von Namen.
- **TreeNode**
Dieses Steuerelement repräsentiert einen einzelnen Knoten in einer Baumansicht.
- **TreeNodeCollection**
Dieses Steuerelement verwaltet eine Liste von **TreeNode**-Objekten.

25.2.1 Konstruktion

Ein **TreeView**-Steuerelement enthält beliebig viele *Stammknoten* aus der Klasse **TreeNode**, die jeweils über ein per **Nodes**-Eigenschaft ansprechbares **TreeNodeCollection**-Memberobjekt beliebig viele weitere **TreeNode**-Objekte als *Unterknoten* aufnehmen können. Weil jedes **TreeNode**-Objekt eine Kollektion von Unterknoten aufnehmen kann, entsteht eine Struktur mit beliebiger Schachtelungstiefe. Wenn Sie sich an die Konstruktion eines Menüs mit Hilfe der Klassen **MenuStrip**, **ToolStripMenuItem** und **ToolStripItemCollection** (vgl. Abschnitt 17) erinnert fühlen, liegen Sie richtig und können den folgenden Quellcode sicher mühelos verstehen:

```
using System;
using System.Windows.Forms;
using System.Drawing;

class TreeViewEinstieg : Form {
    TreeViewEinstieg() {
        Text = "TreeView - Einstieg";

        TreeView tv = new TreeView();
        tv.Parent = this;
        tv.Dock = DockStyle.Fill;

        tv.Nodes.Add("Scanner");
        tv.Nodes[0].Nodes.Add("Canon");
        tv.Nodes[0].Nodes.Add("Panasonic");
        tv.Nodes[0].Nodes.Add("Kodak");
        tv.Nodes[0].Nodes[0].Nodes.Add("M 3060");
        tv.Nodes[0].Nodes[0].Nodes.Add("M 3080");
        tv.Nodes[0].Nodes[1].Nodes.Add("X Delta");
        tv.Nodes[0].Nodes[1].Nodes.Add("Y Zeta");
        tv.Nodes[0].Nodes[2].Nodes.Add("9001");
```



```

        tv.Nodes[0].Nodes[2].Nodes.Add("9004");

        tv.Nodes.Add("Drucker");
        tv.Nodes[1].Nodes.Add("Canon");
        tv.Nodes[1].Nodes.Add("HP");
        tv.Nodes[1].Nodes.Add("Minolta");
        tv.Nodes[1].Nodes[0].Nodes.Add("PIXMA IP 4000");
        tv.Nodes[1].Nodes[0].Nodes.Add("PIXMA IP 5000");
        tv.Nodes[1].Nodes[1].Nodes.Add("QuickPix 3000");
        tv.Nodes[1].Nodes[1].Nodes.Add("PS 7");
        tv.Nodes[1].Nodes[2].Nodes.Add("Magic 2000");
        tv.Nodes[1].Nodes[2].Nodes.Add("FinePix 56");
    }

    [STAThread]
    static void Main() {
        Application.EnableVisualStyles();
        Application.Run(new TreeViewEinstieg());
    }
}

```

Per Indexzugriff auf die Unterknoten erspart man sich explizite Referenzvariablen, muss aber strikt auf die Reihenfolge der Knoten achten. Z.B. wird in folgender Anweisung

```
tv.Nodes[1].Nodes[2].Nodes.Add("Magic 2000");
```

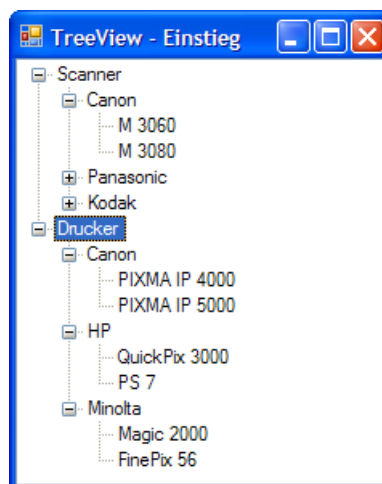
die **Add()**-Methode des 3. Unterknotens (*Minolta*) im 2. Stammknoten (*Drucker*) aufgerufen. Mit expliziten Referenzvariablen vom Typ **TreeNode** kann eventuell die Lesbarkeit des Quellcodes verbessert werden, z.B.:

```

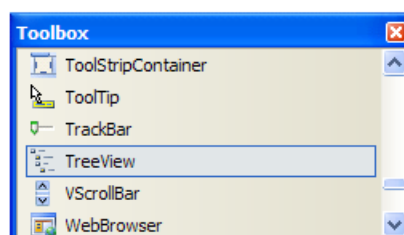
TreeNode minolta = tv.Nodes[1].Nodes[2];
minolta.Nodes.Add("Magic 2000");
minolta.Nodes.Add("FinePix 56");

```

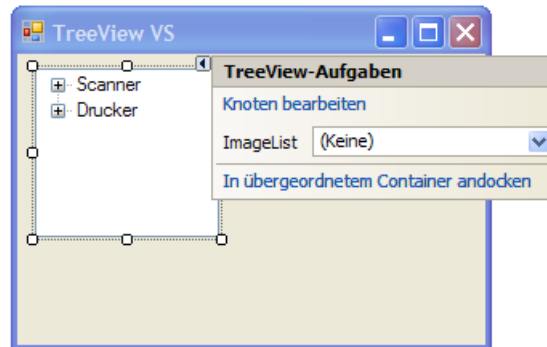
Gemessen am Programmieraufwand erhalten wir eine brauchbare Optik und Funktionalität:



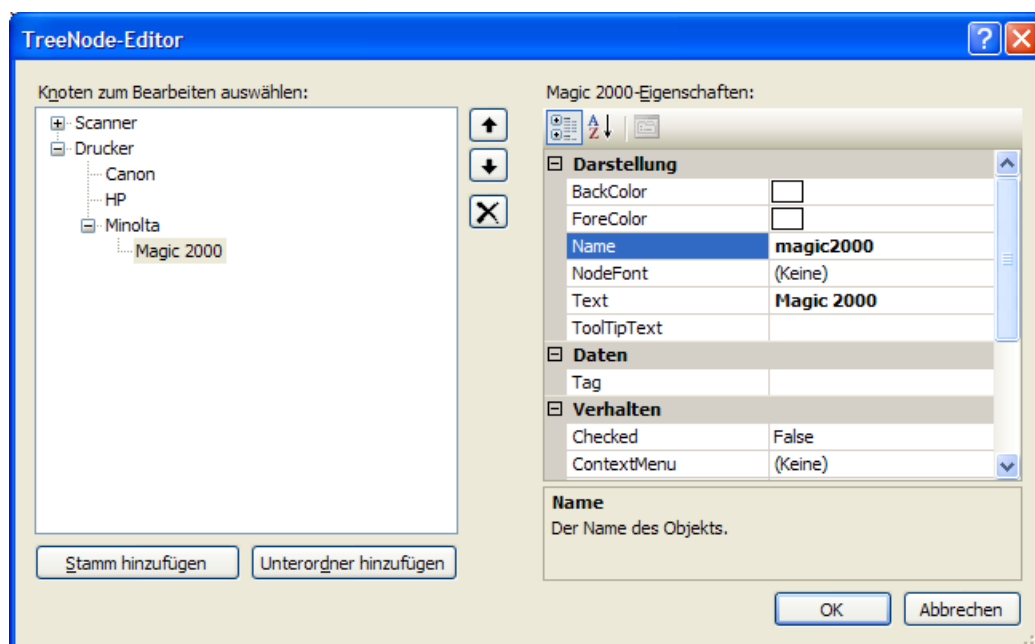
In unseren Entwicklungsumgebungen unterstützt der Formulardesigner nach dem Einfügen eines **TreeView**-Steuerelements



das Erstellen einer Baumansicht über den **TreeNode**-Editor, erreichbar über den Eintrag **Knoten bearbeiten** im **TreeView**-Aufgabenformular, das nach einem Mausklick auf den kleinen Pfeil am oberen Rand des markierten Steuerelements erscheint:



Das Anlegen, Verschieben, Löschen und Konfigurieren von Stamm- und Unterknoten gelingt auf intuitivem Weg:



25.2.2 Datenerfassung

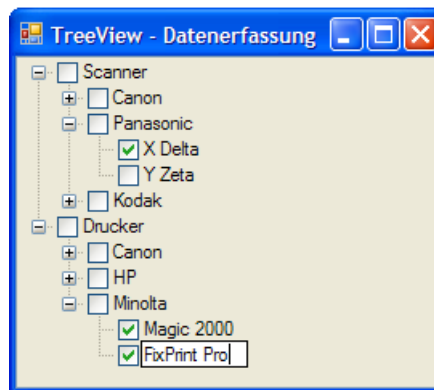
Ein **TreeView**-Objekt eignet sich nicht nur zur Präsentation, sondern auch zur Erfassung von Daten:

- Dem Benutzer stehen die üblichen Techniken zum Markieren *eines* Elements zur Verfügung.
- Um die Mehrfachauswahl zu unterstützen, blendet man per **CheckBoxes**-Eigenschaft Kontrollkästchen ein, z.B.:

```
treeview.CheckBoxes = true;
```
- Man kann per **LabelEdit**-Eigenschaft das Verändern der Knotenbezeichnungen durch den Benutzer erlauben, z.B.:

```
treeview.LabelEdit = true;
```
- Um dem Benutzer einen individuellen Baumaufbau zu gestatten, realisiert man Ereignisroutinen mit passenden Baumpflegemethoden aus den Klassen
 - o **TreeNodeCollection** (z.B. **Add()**) und
 - o **TreeNode** (z.B. **Remove()**)

Nach dem Einschalten der **TreeView**-Eigenschaften **CheckBoxes** und **LabelEdit** kann unser Programm differenzierte Informationen erfassen:



Mit den Details zur Auswertung der vom Benutzer übermittelten Informationen können wir uns aus Zeitgründen nicht beschäftigen.

25.2.3 Verzeichnisbaum

Wir betrachten nun ein besonders wichtiges Beispiel für die Baum- bzw. Strukturansicht: die Darstellung des hierarchisch organisierten Dateisystems. Weil eine Ordneransicht vermutlich in vielen Programmen benötigt wird, erstellen wir in Anlehnung an Petzold (2002, S. 932ff) die aus **TreeView** abgeleitete Klasse **DirectoryView**:

```
using System;
using System.IO;
using System.Windows.Forms;

public class DirectoryView : TreeView {
    public DirectoryView() {
        String[] drives = Directory.GetLogicalDrives();
        foreach (String s in drives) {
            TreeNode tnDrive = new TreeNode(s);
            Nodes.Add(tnDrive);
            AddDirectories(tnDrive);
            if (s == "C:\\")
                SelectedNode = tnDrive;
        }
    }

    void AddDirectories(TreeNode tn) {
        String[] dirs;
        try {
            dirs = Directory.GetDirectories(tn.FullPath);
        } catch {
            return;
        }
        foreach (String s in dirs) {
            TreeNode tnDir = new TreeNode(Path.GetFileName(s));
            tn.Nodes.Add(tnDir);
            AddDirectories(tnDir);
        }
    }
}
```

Zu Demonstrations- und Testzwecken ergänzen wir noch eine **Main()**-Methode:

```
[STAThread]
public static void Main() {
    Application.EnableVisualStyles();
    Form frm = new Form();
    frm.Text = "DirectoryViewDemo";
    DirectoryView dv = new DirectoryView();
    dv.Parent = frm;
    dv.Dock = DockStyle.Fill;
    Application.Run(frm);
}
```

Als Stammknoten treten im Beispiel die logischen Laufwerke auf, und deren Namen liefert uns die statische Methode **GetLogicalDrives()** der Klasse **Directory**, die Ihnen aus dem Abschnitt 12.3.2 über Ein-/Ausgabe per Datenstrom bekannt sein dürfte:

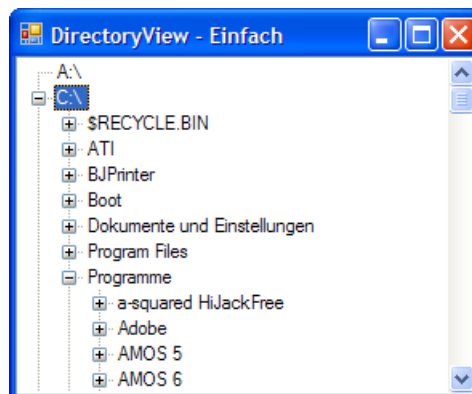
```
String[] drives = Directory.GetLogicalDrives();
```

Analog erhalten wir die Unterknoten eines Laufwerks oder Ordners mit der **Directory**-Methode **GetDirectories()**:

```
dirs = Directory.GetDirectories(tn.FullPath);
```

Weil **GetDirectories()** etliche Ausnahmen auf Lager hat, wird die Methode in einem geschützten **try**-Block aufgerufen.

Im Konstruktor wird die private Methode **AddDirectories()** zum Erstellen der Unterknoten aufgerufen. Durch den rekursiven Selbstaufufruf (vgl. Abschnitt 4.7.1) von **AddDirectory()** erhält man einen recht eleganten Algorithmus, der aber bei heute üblichen Festplattengrößen zu einer erheblichen Wartezeit führt:



Wir lösen das Performanzproblem bald durch einen bedarfsgerechten und ereignisgesteuerten Bauaufbau.

25.2.4 Bilder für die Knoten

Sicher haben Sie in der eben vorgestellten Ordneransicht die vom Windows-Explorer und vergleichbaren Programmen wohlbekannten Bilder für Laufwerke und Ordner vermisst, die nun nachgerüstet werden sollen.

Eine erste Teilaufgabe besteht darin, für jedes Laufwerk seinen Typ festzustellen (z.B. Diskettenlaufwerk, Festplatte, Netzlaufwerk). In .NET 1.1 war dazu noch die Win32-API-Funktion **GetDriveType()** erforderlich:

```
[DllImport("kernel32.dll")]
static extern uint GetDriveType(string drive);
```

Seit der .NET-Version 2.0 bietet die Klasse **DriveInfo** eine bequemere Lösung. In der erweiterten Version des **DirectoryInfo**-Konstruktors verschafft uns die statische **DriveInfo**-Methode **GetDrives()** einen Array mit **DriveInfo**-Objekten, welche die im lokalen System vorhandenen Da-

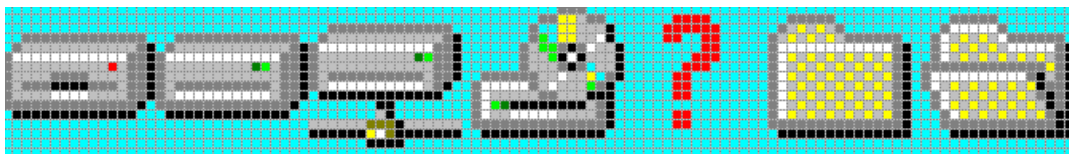
tensträger repräsentieren. Über den Typ eines Laufwerks informiert die **DriveInfo**-Eigenschaft **DriveType** mit Werten vom gleichnamigen Enumerationstyp:

```
public DirectoryView() {
    DriveInfo[] drives = DriveInfo.GetDrives();
    int imageIndex;

    ImageList = new ImageList();
    ImageList.Images.AddStrip(Image.FromFile("dirview.bmp"));
    ImageList.TransparentColor = Color.Cyan;

    foreach (DriveInfo d in drives) {
        switch (d.DriveType) {
            case DriveType.Removable: imageIndex = 0; break;
            case DriveType.Fixed: imageIndex = 1; break;
            case DriveType.Network: imageIndex = 2; break;
            case DriveType.CDRom: imageIndex = 3; break;
            default: imageIndex = 4; break;
        }
        TreeNode tnDrive = new TreeNode(d.Name, imageIndex, imageIndex);
        Nodes.Add(tnDrive);
        AddDirectories(tnDrive);
        if (d.Name == "C:\\")
            SelectedNode = tnDrive;
    }
}
```

Zur typgerechten Bebilderung der Laufwerke und Ordner wird ein **ImageList**-Objekt erstellt und der **ImageList**-Instanzeigenschaft der (aus **TreeView** abgeleiteten) Klasse **DirectoryView** zugewiesen. Die benötigte Bilderliste stammt aus dem folgenden Streifen mit sieben (16×16)-Bitmaps für die wichtigsten Laufwerkstypen sowie für geschlossene und geöffnete Ordner:

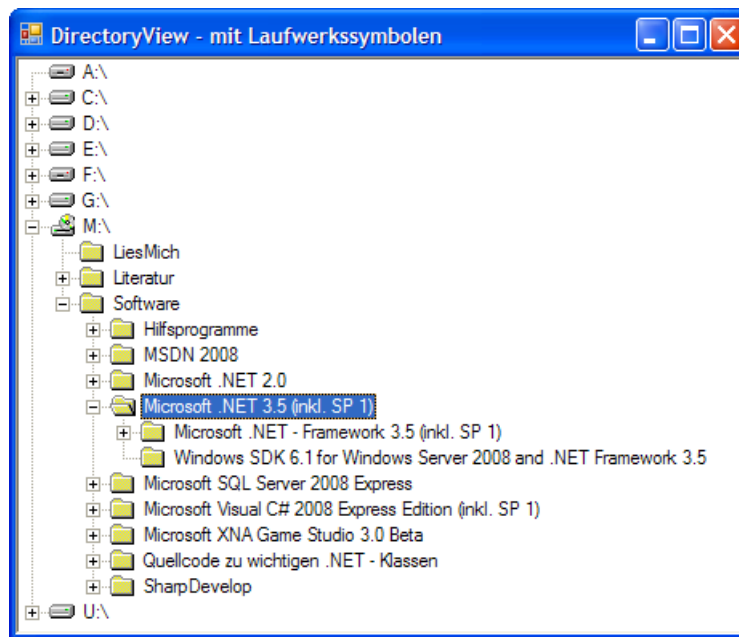


Mit der Transparenz-Deklaration für die Hintergrundfarbe des Streifens sorgen wir für eine ansprechende Optik.

Beim Erstellen der Knoten kommt nun ein alternativer Konstruktor zum Einsatz, der zum normalen und zum markierten Zustand eines Knotens jeweils ein **ImageList**-Element per Indexwert festlegt, z.B. in der folgenden Anweisung zum Erstellen eines Ordnerknotens aus der Methode **AddDirectories()**:

```
TreeNode tnDir = new TreeNode(Path.GetFileName(s), 5, 6);
```

Unser **DirectoryView**-Steuerelement besitzt nun eine akzeptable Optik:



Leider haben Diskettenlaufwerke (z.B. Laufwerk A: in obigem Bildschirmphoto) und USB-Sticks bzw. -Festplatten (z.B. Laufwerk F: in obigem Bildschirmphoto) als **DriveType** denselben Wert **Removable**, so dass wir nicht für die optische Unterscheidbarkeit sorgen können. In der WMI-Datenbank des Betriebssystems (*Windows Management Instrumentation*), die über Klassen im FCL-Namensraum **System.Management.Instrumentation** für .NET - Programme erschlossen wird, sind sicher ausführlichere Informationen vorhanden.

Anschließend wird endlich das schon in Abschnitt 25.2.3 erwähnte Performanzproblem beim Erstellen eines **DirectoryInfo**-Objekts durch einen bedarfsgerechten und ereignisgesteuerten Baumaufbau behoben.

25.2.5 Ereignisse

Während die Klasse **TreeNode** keine Ereignisse anbietet, kann man sich von einem **TreeView**-Objekt u.a. vor und nach der Markierung, Erweiterung oder Reduktion eines Knotens informieren lassen:

Ereignis	On-Methode	Delegatentyp
BeforeSelect	OnBeforeSelect()	TreeViewCancelEventHandler
BeforeExpand	OnBeforeExpand()	TreeViewCancelEventHandler
BeforeCollapse	OnBeforeCollapse()	TreeViewCancelEventHandler
AfterSelect	OnAfterSelect()	TreeViewEventHandler
AfterExpand	OnAfterExpand()	TreeViewEventHandler
AfterCollapse	OnAfterCollapse()	TreeViewEventHandler

Als Designer einer aus **TreeView** abgeleiteten Klasse haben wir bei einer Ereignisbehandlung die Wahl zwischen zwei Techniken:

- die zugehörige **On**-Methode überschreiben (siehe Abschnitt 9.5)
- eine zum jeweiligen Delegatentyp passende Methode definieren und beim Ereignis registrieren (siehe Abschnitt 9.4.1.3).

Die Nachbehandlungsmethoden erhalten beim Aufruf ein Objekt vom Typ **TreeViewEventArgs**, das u.a. per **Node**-Eigenschaft über den betroffenen Knoten informiert. Im Ereignisobjekt vom Typ **TreeViewCancelEventArgs** für die Vorbehandlungsmethoden steckt zusätzlich die schreiboffene

boolesche Eigenschaft **Cancel**. Wenn eine Behandlungsmethode diese Eigenschaft auf den Wert **false** setzt, wird die vom Benutzer geplante Aktion verhindert.

In der folgenden **OnBeforeExpand()**-Überschreibung für unsere **DirectoryView**-Klasse

```
override protected void OnBeforeExpand(TreeViewCancelEventArgs tvcea) {
    base.OnBeforeExpand(tvcea);
    BeginUpdate();
    tvcea.Node.Nodes.Clear();
    AddDirectories(tvcea.Node);
    foreach (TreeNode tn in tvcea.Node.Nodes)
        AddDirectories(tn);
    EndUpdate();
}
```

wird für einen zu expandierenden Knoten *K* zunächst die Liste der Unterknoten aktualisiert. Somit kann der Benutzer durch Reduzieren und anschließendes Erweitern eines Knotens eine Aktualisierung erzwingen. Dann werden die Unterknoten der nächsten Schachtelungsstufe ermittelt, damit das Steuerelement die Unterknoten von *K* ggf. mit einem Plussymbol ausstattet. Mit den Methoden **BeginUpdate()** und **EndUpdate()** verhindern wir vorsichtshalber ein Aktualisieren der Anzeige, während das Expandieren läuft. Bei einem Knoten mit sehr vielen Unterknoten könnte es anderenfalls zu einem Flimmern der Anzeige kommen.

Mit dem rekursiven Selbstaufruf am Ende der Methode **AddDirectories()** verschwindet unser Performanzproblem:

```
void AddDirectories(TreeNode tn) {
    String[] dirs = {""};
    try {
        dirs = Directory.GetDirectories(tn.FullPath);
    } catch {
        return;
    }
    TreeNode[] tna = new TreeNode[dirs.Length];
    for (int i = 0; i < dirs.Length; i++) {
        tna[i] = new TreeNode(Path.GetFileName(dirs[i]), 5, 6);
    }
    tn.Nodes.AddRange(tna);
}
```

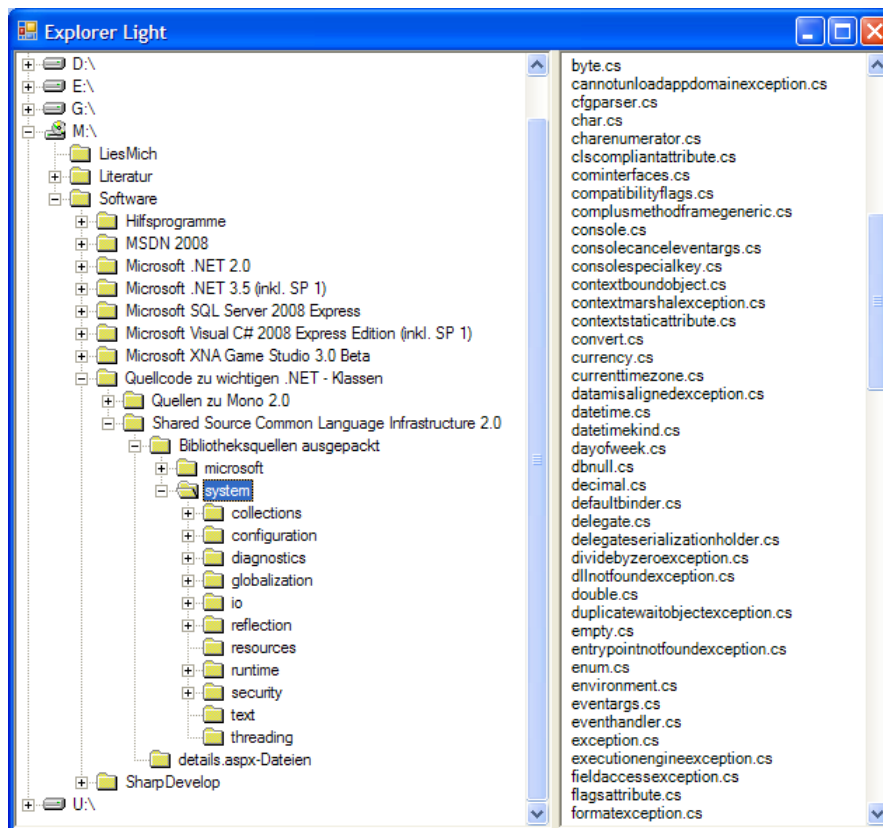
Außerdem nehmen wir nicht mehr einzelne Unterknoten auf, sondern per **AddRange()** -Aufruf jeweils komplette Arrays mit allen Unterknoten eines Knotens.

Den vollständigen Quellcode der Klasse **DirectoryView** finden Sie im Ordner

...\BspUeb\WinForms\TreeView\DirectoryView\Fertig

25.2.6 Explorer Light

Zum Abschluss der **TreeView**-Behandlung soll als Anwendungsbeispiel für unser neues Steuerelement **DirectoryView** eine Light-Variante des Windows-Explorers erstellt werden:



Bei der Raumaufteilung kommt die in Abschnitt 25.1.2 behandelte Klasse **SplitContainer** zum Einsatz:

```
using System;
using System.Drawing;
using System.Windows.Forms;
using System.IO;

class ExplorerLight : Form {
    SplitContainer sc;
    DirectoryView dv;
    Panel fileList;

    ExplorerLight() {
        Text = "Explorer Light";
        sc = new SplitContainer();
        sc.Parent = this;
        sc.Dock = DockStyle.Fill;
        sc.SplitterWidth = 5;
        sc.BorderStyle = BorderStyle.Fixed3D;

        dv = new DirectoryView();
        dv.Parent = sc.Panel1;
        dv.Dock = DockStyle.Fill;
        dv.BorderStyle = BorderStyle.None;
        dv.AfterSelect += new TreeViewEventHandler(DirectoryViewOnAfterSelect);

        sc.Panel2.AutoScroll = true;
        sc.Panel2.BackColor = SystemColors.Window;

        fileList = new Panel();
        fileList.Parent = sc.Panel2;
        fileList.Dock = DockStyle.Top;
        fileList.Height = sc.Panel2.Height;
        fileList.BackColor = SystemColors.Window;
        fileList.Paint += new PaintEventHandler(FileListOnPaint);
    }
}
```



```

void DirectoryViewOnAfterSelect(object sender, TreeViewEventArgs tvea) {
    fileList.Refresh();
}

void FileListOnPaint(object sender, PaintEventArgs e) {
    if (dv.SelectedNode == null)
        return;
    String[] files;
    try {
        files = Directory.GetFiles(dv.SelectedNode.FullPath);
    } catch {
        return;
    }
    int y = 5;
    foreach (String s in files) {
        e.Graphics.DrawString(Path.GetFileName(s), Font,
            SystemBrushes.WindowText, 5, y);
        y += Font.Height;
    }
    fileList.Height = y;
}

[STAThread]
static void Main() {
    Application.EnableVisualStyles();
    Application.Run(new ExplorerLight());
}

```

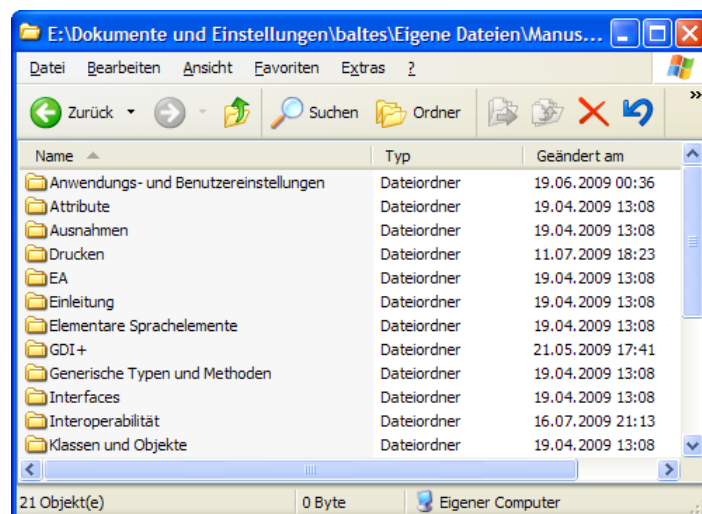
In den linken **SplitContainer**-Bereich setzen wir **DirectoryView**-Objekt mit dem **Dock**-Wert **DockStyle.Fill**. Rechts platzieren wir ein oben angedocktes **Panel**-Steuerelement mit einem **Paint**-Handler, welcher die Namen der Dateien im markierten Knoten auflistet und die **Panel**-Höhe neu berechnet. In einem **AfterSelect**-Handler für das **DirectoryView**-Objekt veranlassen wir die Aktualisierung der Dateienliste beim Wechsel der Knotenmarkierung.

25.3 Listen und Tabellen

Zum Präsentieren einer Tabelle bzw. Liste mit mehreren Spalten und einer beliebigen Anzahl von Einträgen sind die Steuerelemente **ListBox** und **ComboBox** wenig geeignet, weil die spaltenbündige Anzeige kaum gelingt. Als Alternative bietet das WinForms-Framework die Klasse **ListView**, die mit ihrer Detailansicht das Spaltenproblem löst. Wählbar über die **ListView**-Eigenschaft **View** stehen für die Listeneinträge, welche durch Objekte der Klasse **ListViewItem** repräsentiert werden, insgesamt die folgenden fünf Ansichten zur Verfügung:

View	Beschreibung
Details	Jedes ListViewItem erscheint in einer eigenen Zeile. Durch das Definieren von Spaltenköpfen über die Columns -Eigenschaft der Klasse ListView erhält man eine tabellarische Darstellung. In der ersten Spalte erscheint der Wert der ListViewItem -Eigenschaft Text . In weiteren Spalten erscheinen die Text -Eigenschaften der zu einem Item erzeugten ListViewItem.ListViewSubItem - Objekte. Ist mit einem Item ein Bild verknüpft, erscheint dieses am linken Rand der ersten Spalte.
List	Im Unterschied zur Details -Ansicht werden Spaltenköpfe und Subitems ignoriert. Bei ausreichender Fensterbreite erscheinen mehrere Spalten (mit Items) nebeneinander.
LargeIcon, SmallIcon	Für diese Darstellungsarten sollten Symbole zu den Items vorhanden sein. Diese erscheinen in voller Größe mit einer Beschriftung darunter (LargeIcon) oder in reduzierter Größe mit einer Beschriftung rechts daneben (SmallIcon).
Tile	Bei der Kachelansicht erscheinen Symbole in voller Größe mit einer Elementbezeichnung sowie Unterelementinformationen rechts daneben.

Sicher haben Sie längst bemerkt, dass die **ListView**-Klasse sehr für eine Auflistung im Stil des Windows-Explorers geeignet ist. Hier ist die Detailansicht des Programms **explorer.exe** zu sehen, das als Vorbild für unsere nächsten Projekte dient:



25.3.1 Konstruktion

Um mit den Klassen **ListView**, **ListViewItem** und **ListViewItem.ListViewSubItem** vertraut zu werden, erstellen wir zunächst ein einfaches Beispielprogramm:

```
using System;
using System.Windows.Forms;

class ListViewDemo : Form {
    ListViewDemo() {
        Text = "ListView-Demo";
        ListView lv = new ListView();
        lv.Parent = this;
        lv.Dock = DockStyle.Fill;
        lv.View = View.Details;
        lv.Sorting = SortOrder.Ascending;

        lv.Columns.Add("Name", 120, HorizontalAlignment.Left);
        lv.Columns.Add("Preis", 100, HorizontalAlignment.Right);
        lv.Columns.Add("Kalorien", 100, HorizontalAlignment.Right);
    }
}
```

```

String[] namen = {"Lecker & Gesund", "Beckerstolz",
                  "Die gute Paste", "Prima fürs Brot"};
double[] preise = {2.34, 1.78, 2.19, 0.78};
int[] kalorien = {57, 77, 121, 85};
for (int i = 0; i < namen.Length; i++) {
    ListViewItem lvi = new ListViewItem(namen[i]);
    lv.Items.Add(lvi);
    lvi.SubItems.Add(preise[i].ToString());
    lvi.SubItems.Add(kalorien[i].ToString());
}

[STAThread]
static void Main() {
    Application.EnableVisualStyles();
    Application.Run(new ListViewDemo());
}

```

Es wird ein **ListView**-Objekt erstellt und mit dem **DockStyle.Fill** in das Anwendungsfenster gesetzt. Wir wählen über die oben diskutierte **View**-Eigenschaft die Detailansicht und sorgen per **Sort**-Eigenschaft für eine aufsteigende Sortierung der Listenitems nach der ersten Spalte.

Die **ListView**-Eigenschaft **Columns** zeigt auf ein Objekt der internen Klasse **ListView.ColumnHeaderCollection**, welches eine Sammlung von Spaltenköpfen verwaltet, die durch Objekte der Klasse **ColumnHeader** dargestellt werden. Es stehen die bei Kollektionsklassen üblichen Methoden wie **Add()**, **Clear()**, **Remove()** etc. zur Verfügung.

Ein **ColumnHeader**-Objekt (Spaltenkopf) besitzt u.a. die folgenden Eigenschaften:

- **String Text**
Spaltenbeschriftung
- **int Width**
Initiale Breite der Spalte. Mit speziellen Werten erhält man eine automatische Breitenanpassung:

Width	automatische Anpassung ...
-1	an das breiteste Element in der Spalte
-2	an die Breite der Spaltenüberschrift

- **HorizontalAlignment TextAlign**
Horizontale Textausrichtung in einer Spalte

Die **ListView**-Eigenschaft **Items** zeigt auf ein Objekt der internen Klasse **ListView.ListViewItemCollection**, welches eine Sammlung von Objekten der Klasse **ListViewItem** verwaltet und die bei Kollektionsklassen üblichen Methoden bietet.

Ein Listenelement (**ListViewItem**-Objekt) besitzt u.a. die folgenden Eigenschaften:

- **String Text**
Bei mehrspaltigen Listen erscheint dieser Text in der ersten Spalte.
- **ListViewSubItemCollection.SubItems**
Die Eigenschaft **SubItems** zeigt auf ein Objekt der internen Klasse **ListViewItem.ListViewSubItemCollection**, welches eine Sammlung von Objekten der internen Klasse **ListViewItem.ListViewSubItem** verwaltet und die bei Kollektionsklassen üblichen Methoden bietet.

Listenunterelemente (**ListViewItem.ListViewSubItem**-Objekte) besitzen ebenfalls eine **Text**-Eigenschaft. Bei der Detailansicht erscheinen die **Text**-Werte der Subitems sukzessive ab der zweiten Spalte.

Das Beispielprogramm zeigt für diverse Brotaufstriche die Namen linksbündig, die Preise und Kalorienwerte hingegen rechtsbündig an, wobei die Zeilen aufsteigend nach der ersten Spalte sortiert sind:



Name	Preis	Kalorien
Beckerstolz	1,78	77
Die gute Paste	2,19	121
Lecker & Gesund	2,34	57
Prima fürs Brot	0,78	85

25.3.2 Datenerfassung

Wie ein **TreeView**-Objekt (vgl. Abschnitt 25.2.2) eignet sich auch ein **ListView**-Objekt dazu, Informationen vom Benutzer entgegenzunehmen:

- Dem Benutzer stehen die üblichen Techniken zum Markieren von Elementen zur Verfügung, wobei (anders als beim **TreeView**-Steuerelement) auch eine *Mehrfachauswahl* möglich ist. Über die **ListView**-Eigenschaft **SelectedItems**, die auf ein Objekt der internen Klasse **ListView.SelectedItemsCollection** zeigt, ist die Liste der gewählten Items ansprechbar. Alternativ kann man über die Eigenschaft **SelectedIndices** auf eine Liste mit den Indizes der ausgewählten Items zugreifen. Auswahländerungen werden über das Ereignis **SelectedItemChanged** gemeldet.
- Als alternative Möglichkeit zur Artikulation einer Mehrfachauswahl lassen sich per **CheckBoxes**-Eigenschaft Kontrollkästchen einschalten, z.B.:

```
lv.CheckBoxes = true;
```

Über die **ListView**-Eigenschaft **CheckedItems**, die auf ein Objekt der internen Klasse **ListView.CheckedListViewItemCollection** zeigt, ist die Liste der aktuell markierten Items ansprechbar, die sich durchaus von der Liste der ausgewählten Items unterscheiden kann, z.B.:



Name	Preis	Kalorien
☒ Beckerstolz	1,78	77
☐ Die gute Paste	2,19	121
☐ Lecker & Gesund	2,34	57
☒ Prima fürs Brot	0,78	85

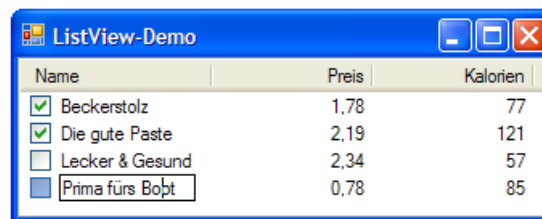
Sind Kontrollkästchen zugeschaltet, signalisiert das **ListView**-Ereignis **ItemCheck** einen Zustandswechsel.

- Man kann per **LabelEdit**-Eigenschaft das Verändern der Itembezeichnungen durch den Benutzer erlauben, z.B.:

```
lv.LabelEdit = true;
```

Den Text der Subitems kann man nicht auf diese Weise frei geben. In der FCL-Dokumentation empfiehlt Microsoft, stattdessen eine per Doppelklick aufrufbare Dialogbox einzusetzen.

Nach dem Einschalten der **ListView**-Eigenschaften **CheckBoxes** und **LabelEdit** kann unser Programm differenzierte Informationen erfassen:



Mit Details zur Auswertung der vom Benutzer übermittelten Informationen können wir uns aus Zeitgründen nicht beschäftigen.

25.3.3 Ereignisse

Analog zu den Verhältnissen bei der Baumansicht (vgl. Abschnitt 25.2.5) tritt bei der Listenansicht nur die Klasse **ListView** (nicht aber die Klasse **ListViewItem**) als Ereignismitteld in Erscheinung. Neben den bereits erwähnten Ereignissen **SelectedIndexChanged** (Auswahländerung) und **ItemCheck** (Markierungsänderung) sind auch die folgenden Ereignisse oft von Interesse:

Ereignis	Erläuterung
ItemActivate	Dieses Ereignis tritt ein, sobald der Benutzer ein Item aktiviert, um eine Bearbeitungsintention auszudrücken. Im Windows-Explorer hat das Aktivieren z.B. zur Folge, dass ein Ordnerinhalt angezeigt, ein Dokument geöffnet bzw. ein Programm gestartet wird. Das Aktivieren ist also vom einfachen Auswählen zu unterscheiden. Per Voreinstellung dient der Doppelklick zum Aktivieren. Über die Activation-Eigenschaft, die Werte vom Aufzählungstyp ItemActivation akzeptiert, kann aber ein alternatives Aktivierungsverfahren gewählt werden: OneClick Ein Mausklick genügt zum Aktivieren, wobei ein handförmiger Mauszeiger  über die aktuell angehobene Funktionalität des linken Mausklicks informiert. TwoClick Ein bereits markiertes Item kann durch einen linken Mausklick aktiviert werden. Über den zu erwartenden Effekt informieren ein handförmiger Mauszeiger  und eine optisch veränderte Markierung.
ColumnClick	Der Benutzer hat auf einen Spaltenkopf geklickt. Jetzt kann man etwa die Liste nach der Spalte sortieren (siehe Abschnitt 25.3.6).

Eine **ItemActivate**- oder **SelectedIndexChanged**-Behandlungsmethode erhält beim Aufruf nur ein gewöhnliches **EventArgs**-Objekt, jedoch ist z.B. über die **ListView**-Eigenschaft **SelectedItems** leicht zu ermitteln, welche Items ausgewählt sind. Insgesamt sind über 90 **ListView**-Ereignisse definiert, so dass sich ein Blick in die FCL-Dokumentation lohnt.

25.3.4 Dateisystem-Verzeichniseinträge auflisten

In der folgenden **ListView**-Ableitung, die teilweise auf einem Vorschlag von Petzold (2002, S. 952f) basiert, werden elementare Funktionen der Dateien- bzw. Unterordnerliste im Windows-Explorer realisiert. Es steht nur die **Details**-Ansicht zur Verfügung, wobei für jeden Eintrag der Name, die Größe (nur bei Dateien) und der Zeitpunkt der letzten Änderung erscheinen:

```

using System;
using System.IO;
using System.Windows.Forms;
using System.Threading;

public class DirEntryList : ListView {
    String strDir;

    public DirEntryList(String dir) {
        strDir = dir;
        View = View.Details;
        Sorting = SortOrder.Ascending;
        Columns.Add("Name", 120, HorizontalAlignment.Left);
        Columns.Add("Größe", 120, HorizontalAlignment.Left);
        Columns.Add("Letzte Änderung", 120, HorizontalAlignment.Left);
    }

    public String Dir {
        get {
            return strDir;
        }
        set {
            if (Directory.Exists(value))
                strDir = value;
        }
    }

    public void ShowEntries() {
        ShowEntries(strDir);
    }

    public void ShowEntries(String dir) {
        if (!Directory.Exists(dir))
            return;
        try {
            BeginUpdate();
            strDir = dir;
            Items.Clear();
            DirectoryInfo directoryInfo = new DirectoryInfo(dir);

            DirectoryInfo[] adi;
            adi = directoryInfo.GetDirectories();
            foreach (DirectoryInfo di in adi) {
                ListViewItem lvi = new ListViewItem(di.Name);
                lvi.SubItems.Add("");
                lvi.SubItems.Add(di.LastWriteTime.ToString());
                Items.Add(lvi);
            }

            FileInfo[] afi;
            afi = directoryInfo.GetFiles();
            foreach (FileInfo fi in afi) {
                ListViewItem lvi = new ListViewItem(fi.Name);
                lvi.SubItems.Add(fi.Length.ToString());
                lvi.SubItems.Add(fi.LastWriteTime.ToString());
                Items.Add(lvi);
            }
        } catch (Exception e) {
            MessageBox.Show(e.Message);
            return;
        } finally {
            EndUpdate();
        }
    }
}

```

```

override protected void OnItemActivate(EventArgs ea) {
    base.OnItemActivate(ea);
    foreach (ListViewItem lvi in SelectedItems) {
        if (lvi.SubItems[1].Text == "") // Item ist Unterordner
            ShowEntries(Path.Combine(strDir, lvi.Text));
        else
            ThreadPool.QueueUserWorkItem(new WaitCallback(StartProcess),
                                           Path.Combine(strDir, lvi.Text));
    }
}

protected void StartProcess(object path) {
    try {
        System.Diagnostics.Process.Start((String)path);
    } catch {
        return;
    }
}
}

```

Zu Demonstrations- und Testzwecken ergänzen wir noch eine **Main()**-Methode:

```

[STAThread]
static void Main() {
    Application.EnableVisualStyles();
    Form frm = new Form();
    frm.Text = "DirEntryList - Einfach";
    DirEntryList fv = new DirEntryList("C:\\");
    fv.Parent = frm;
    fv.Dock = DockStyle.Fill;
    fv.ShowEntries();
    Application.Run(frm);
}

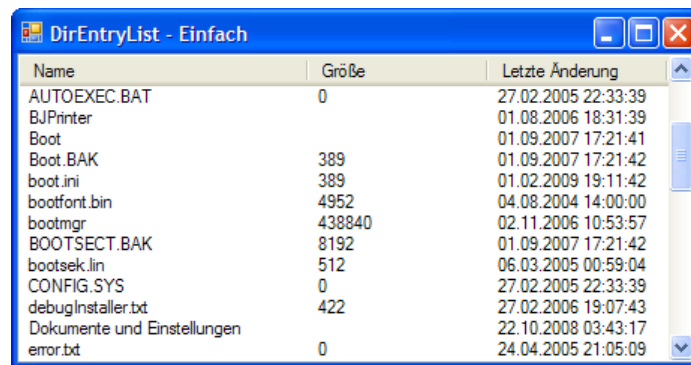
```

Über die statischen Methoden **GetFiles()** und **GetDirectories()** der Klasse **DirectoryInfo** erhalten wir einen **FileInfo**- bzw. **DirectoryInfo**-Array mit den Dateien bzw. Unterordnern im zu beschreibenden Verzeichnis. Mit den genannten Klassen zur Datei- bzw. Ordnerverwaltung haben wir schon in Abschnitt 12.3 gearbeitet.

Die überschreibende Behandlungsmethode für das Ereignis **ItemActivate** sorgt dafür, dass die aktivierte Datei mit Hilfe der statischen Methode **Start()** aus der Klasse **Process** (Namensraum **System.Diagnostics**) gestartet bzw. geöffnet wird. Weil ein Aufruf der Methode **Start()** je nach dem zu startenden Programm relativ lange dauert, soll er nicht im GUI-Thread stattfinden, um verzögerte Reaktionen der Benutzeroberfläche zu vermeiden. Daher wird jeder **Start()**-Aufruf mit Hilfe der statischen **ThreadPool**-Methode **QueueUserWorkItem()** in einem eigenen Thread aus dem verwalteten Threadpool ausgeführt.

Ist der aktivierte Verzeichniseintrag ein Ordner, dann werden seine Verzeichniseinträge aufgelistet. So gelingt der Wechsel zu einem Unterordner, während der umgekehrte Weg mangels Eintrag für das übergeordnete Verzeichnis nicht zur Verfügung steht. Es macht aber sicher keine großen Probleme, den Rückweg z.B. über einen Symbolschalter zu realisieren (wie im Windows-Explorer).

Die Bedienoberfläche des **DirEntryList**-Steuerelements ist funktional und ästhetisch noch verbesserungsbedürftig:



Name	Größe	Letzte Änderung
AUTOEXEC.BAT	0	27.02.2005 22:33:39
BJPrinter		01.08.2006 18:31:39
Boot		01.09.2007 17:21:41
Boot.BAK	389	01.09.2007 17:21:42
boot.ini	389	01.02.2009 19:11:42
bootfont.bin	4952	04.08.2004 14:00:00
bootmgr	438840	02.11.2006 10:53:57
BOOTSECT.BAK	8192	01.09.2007 17:21:42
bootsek.lin	512	06.03.2005 00:59:04
CONFIG.SYS	0	27.02.2005 22:33:39
debugInstaller.txt	422	27.02.2006 19:07:43
Dokumente und Einstellungen		22.10.2008 03:43:17
error.txt	0	24.04.2005 21:05:09

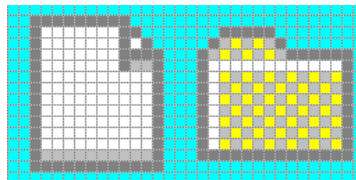
Weil Microsoft in der FCL-Dokumentation (Stand: .NET 3.5) keine Ordnung für die **GetFiles()** - Produktion garantieren möchte, hat die **ListView**-Eigenschaft **Sorting** den Wert **Sort-Order.Ascending** erhalten, was allerdings zum *gemeinsamen* Sortieren von Dateien und Ordnern führt. In Abschnitt 25.3.6 sorgen wir dafür, dass zunächst die Unterordner und danach die Dateien (jeweils sortiert) erscheinen.

25.3.5 Bilder für die Listenitems

Wie die Knoten im **TreeView**-Steuerelement (vgl. Abschnitt 25.2.4) lassen sich auch die Items im **Listview**-Steuerelement durch Bilder illustrieren. Im Konstruktor unserer **Listview**-Ableitung **DirEntryList** wird ein neues **ImageList**-Objekt erzeugt und der **Listview**-Eigenschaft **SmallImageList** zugewiesen, die auch bei der Detailansicht relevant ist:

```
SmallImageList = new ImageList();
SmallImageList.Images.AddStrip(Image.FromFile("direntry.bmp"));
SmallImageList.TransparentColor = Color.Cyan;
```

Beim zugehörigen Bitmapstreifen beschränken wir uns auf Bilder für beliebige Dateien und Ordner:

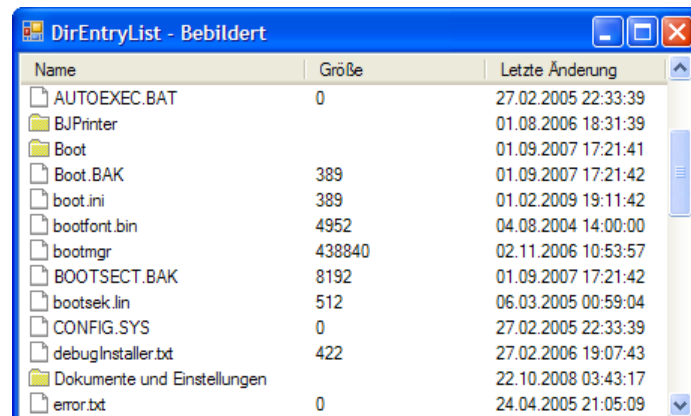


Mit der Transparenz-Deklaration für die Hintergrundfarbe des Streifens sorgen wir für eine ansprechende Optik.

Beim Erstellen der Items kommt nun ein alternativer Konstruktor zum Einsatz, der eine Beschriftung *und* (per Indexwert) ein **ImageList**-Element festlegt, z.B. bei Dateien:

```
ListViewItem lvi = new ListViewItem(fi.Name, 0);
```

Nun sind Ordner und Dateien leichter zu unterscheiden:



Name	Größe	Letzte Änderung
AUTOEXEC.BAT	0	27.02.2005 22:33:39
BJPrinter		01.08.2006 18:31:39
Boot		01.09.2007 17:21:41
Boot.BAK	389	01.09.2007 17:21:42
boot.ini	389	01.02.2009 19:11:42
bootfont.bin	4952	04.08.2004 14:00:00
bootmgr	438840	02.11.2006 10:53:57
BOOTSECT.BAK	8192	01.09.2007 17:21:42
bootsek.lin	512	06.03.2005 00:59:04
CONFIG.SYS	0	27.02.2005 22:33:39
debugInstaller.txt	422	27.02.2006 19:07:43
Dokumente und Einstellungen		22.10.2008 03:43:17
error.txt	0	24.04.2005 21:05:09

25.3.6 Spezielle Sortierung

Der Windows-Explorer zeigt per Voreinstellung die Ordner vor den Dateien an und sortiert beide Typen separat. Außerdem kann der Benutzer per Mausklick auf einen Spaltenkopf das Sortierkriterium und die Sortierrichtung ändern. Bei unserem Steuerelement `DirEntryList` ist bisher per **Sorting**-Eigenschaft ein gemeinsames, aufsteigendes Sortieren fest eingestellt. Um die vom Windows-Explorer gewohnten Sortieroptionen anzubieten, nehmen wir folgende Erweiterungen vor:

- Beim Sortieren entscheidet ein Objekt der intern definierten Klasse `DirEntryComparer` über die Anordnung von zwei **ListViewItem**-Objekten. Diese Klasse muss das Interface **IComparer** implementieren, das die öffentliche Methode **Compare()** mit der folgenden Signatur
`public int Compare(Object x, Object y)`
 vorschreibt.
- Unsere **Compare()**-Implementierung berücksichtigt die vom Benutzer gewählte Sortierspalte und -richtung. Dazu kennen `DirEntryComparer`-Objekte über eine Instanzvariable vom Typ `DirEntryList` das zu unterstützende Steuerelement.
- In der Klasse `DirEntryList` implementieren wir die öffentlichen Eigenschaften `SortCol` und `SortOrder`, an denen der `DirEntryComparer` seine Arbeitsweise orientiert.
- In die `DirEntryList`-Definition wird ein Memberobjekt der Klasse `DirEntryComparer` aufgenommen und der **ListView**-Eigenschaft **ListViewItemSorter** zugewiesen:

```
DirEntryComparer dirEntryComparer = new DirEntryComparer(this);
ListViewItemSorter = dirEntryComparer;
```

- Um dem Benutzer die Kontrolle über die Sortierung zu geben, wird in der Klasse `DirEntryList` eine Behandlungsmethode für das Ereignis **ColumnClick** implementiert

```
public void DirEntryListOnColumnClick(Object sender,
                                       ColumnClickEventArgs ccea) {
    sortCol = ccea.Column;
    sortOrder = sortOrderArray[SortCol] *= -1;
    ShowEntries();
}
```

und registriert:

```
ColumnClick += new ColumnClickEventHandler(DirEntryListOnColumnClick);
```

- Die Subitems für die Größe und den letzten Schreibzugriff erhalten bei der Konstruktion die anzuzeigende Zeichenfolge. Beim Sortieren sind aber Zeichenfolgen mit formatierten Datums- und Zeitangaben (z.B. 24.07.2009 11:45) wenig nützlich. Ähnliche Probleme treten auch beim Sortieren nach Größe auf. Daher nutzen wir die **ListViewItem**-Eigenschaft **Tag**, um jedem Eintrag ein **DirectoryInfo**- oder ein **FileInfo**-Objekt anzuheften, das vom `DirEntryComparer` einfach zu verwenden ist, z.B.:

```
order = comparer.Compare((lx.Tag as DirectoryInfo).LastWriteTime,
                          (ly.Tag as DirectoryInfo).LastWriteTime);
return del.SortOrder > 0 ? order : -order;
```

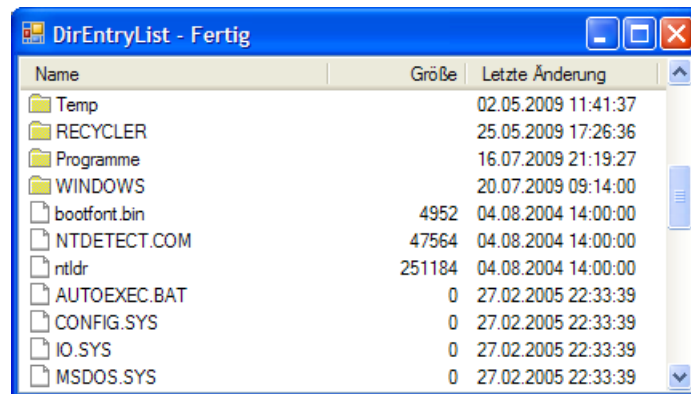
Hier wird per **Tag**-Eigenschaft auf die angehefteten **DirectoryInfo**-Objekte zugegriffen. Für den eigentlichen Vergleich verwenden wir ein Objekt der Klasse **Comparer** (Namensraum **System.Collections**):

```
Comparer comparer = new Comparer(CultureInfo.CurrentCulture);
```

Seine **Compare()**-Methode vergleicht zwei Objekte mit demselben Typ, der das Interface **IComparable** erfüllen muss. Beim Typ **String** werden die regionalen Sortierregeln berück-

sichtigt. Bei seiner Rückmeldung muss der `DirEntryComparer` die Eigenschaft **SortOrder** des `DirEntryList`-Objekts `del` berücksichtigen.

Nun kann unser Steuerelement z.B. Ordner und Dateien getrennt nach dem Änderungsdatum sortieren:



Den vollständigen Quellcode der Klasse `DirEntryList` finden Sie im Ordner
...\BspUeb\WinForms\ListView\DirEntryList\Fertig

25.3.7 Explorer Midrange

Mit dem neu entwickelten Steuerelementen `DirectoryView` (siehe Abschnitt 25.2.3ff) und `DirEntryList` (siehe Abschnitt 25.3.4ff) können wir eine verbesserte Variante unseres Explorer-Nachbaus erstellen:

```
using System;
using System.Drawing;
using System.Windows.Forms;

class ExplorerMidrange : Form {
    SplitContainer sc;
    DirectoryView dv;
    DirEntryList del;

    ExplorerMidrange() {
        sc = new SplitContainer();
        sc.Parent = this;
        sc.Dock = DockStyle.Fill;
        sc.SplitterWidth = 5;
        sc.BorderStyle = BorderStyle.Fixed3D;
        sc.FixedPanel = FixedPanel.Panel1;

        dv = new DirectoryView();
        dv.Parent = sc.Panel1;
        dv.Dock = DockStyle.Fill;
        dv.BorderStyle = BorderStyle.None;
        dv.AfterSelect += new TreeViewEventHandler(DirectoryViewOnAfterSelect);

        del = new DirEntryList(dv.SelectedNode.FullPath);
        del.Parent = sc.Panel2;
        del.Dock = DockStyle.Fill;
    }

    void DirectoryViewOnAfterSelect(object sender, TreeViewEventArgs tvea) {
        String s = tvea.Node.FullPath.Replace(@"\", @"\");
        del.ShowEntries(s);
        Text = s;
    }
}
```

```

[STAThread]
static void Main() {
    Application.EnableVisualStyles();
    Application.Run(new ExplorerMidrange());
}
}

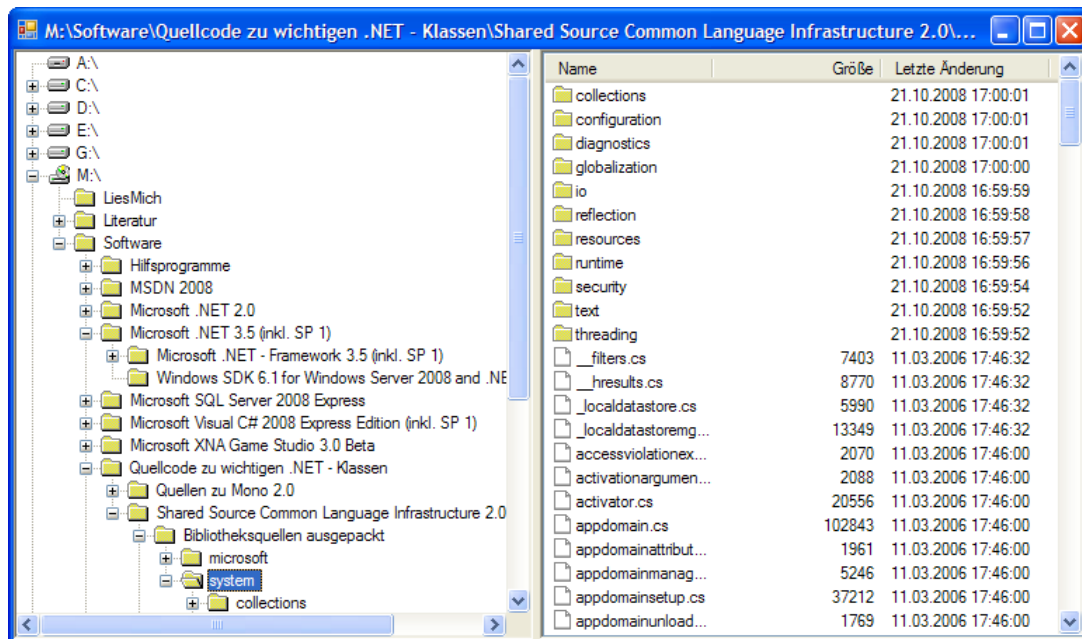
```

In der Methode `DirectoryViewOnAfterSelect()`, die nach einer Auswahländerung in der Strukturansicht die Liste der Ordnerinhalte neu aufbauen lässt, wird mit der folgenden Anweisung

```
String s = tvea.Node.FullPath.Replace(@"\", @"\");
```

eine Backslash-Verdopplung nach der Laufwerksangabe (z.B.: C:\\Perl) korrigiert.

Wir haben mit begrenztem Aufwand ein brauchbares und leicht ausbaufähiges Programm erstellt:



Zur Projektorganisation ist noch anzumerken, dass die beiden Klassen `DirectoryView` und `Dir-EntyList` gemeinsam in einem Bibliotheks-Assembly mit integrierten Ressourcen verfügbar gemacht werden sollten.

26 Datenbankprogrammierung mit ADO.NET

In diesem Abschnitt soll ein erster Eindruck von der Datenbankprogrammierung mit C# vermittelt werden. Wer ausführliche Information benötigt, wird z.B. bei Doberenz & Gewinnus (2008) fündig.

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.

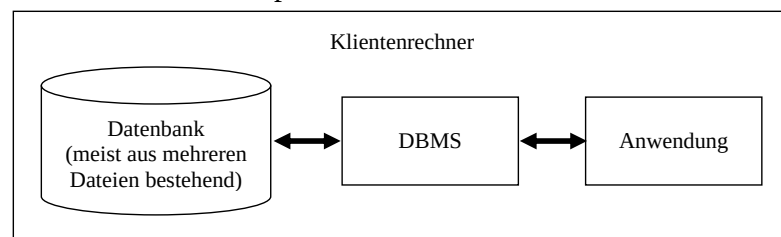
Redundanz würde sich z.B. in der Datensammlung eines Versandhauses schnell einstellen, wenn man für jede Bestellung einen Datensatz anlegen und dabei z.B. auch die Adresse des Bestellers einbeziehen würde. Sobald von einem Kunden mehrere Bestellungen vorlägen, wäre z.B. seine Adresse mehrfach vorhanden. 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:

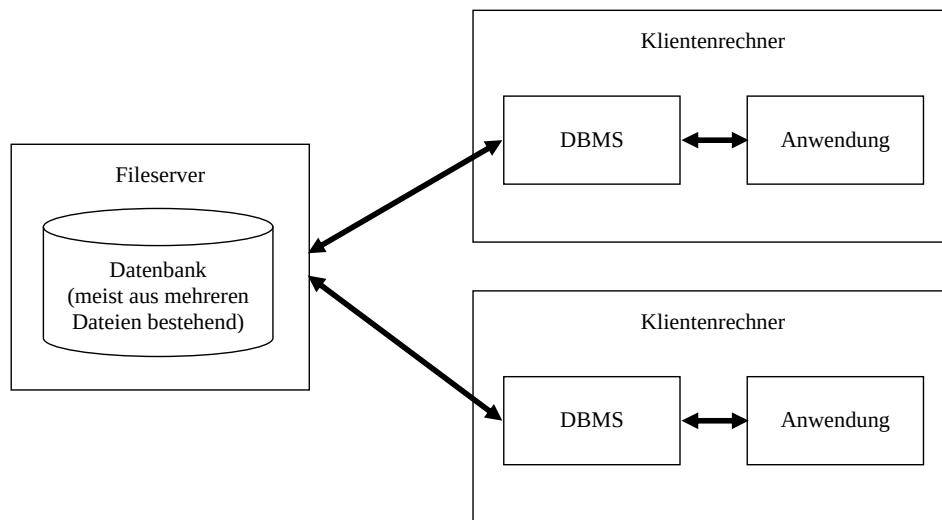
- Microsoft SQL Server (mit Express- und Compact- Edition)
- MySQL
- Oracle Database
- DB2 von IBM usw.

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:

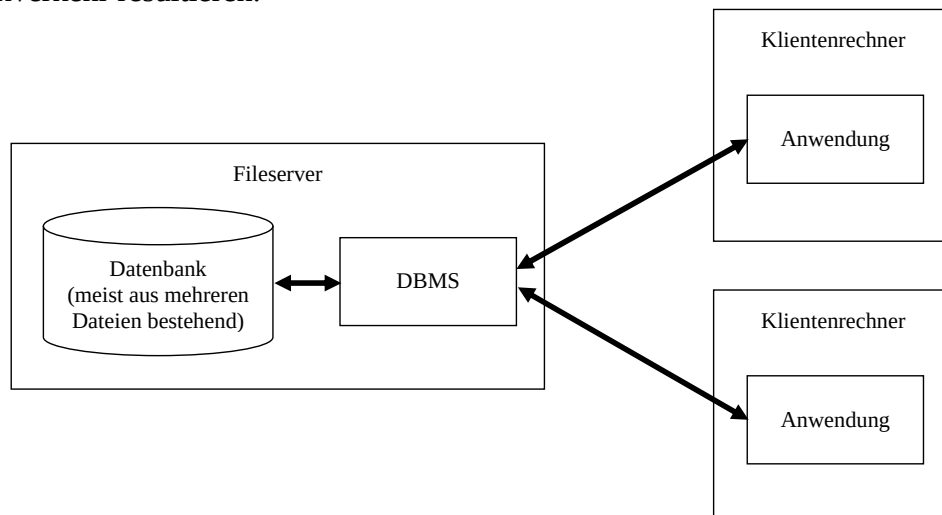
- Einzelbenutzerdatenbank
Anwendungsprogramm und DBMS laufen auf demselben Rechner, z.B. bei einer Seriendrucklösung aus den MS-Officekomponenten Access und Word.



- 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, die zusammen mit der Visual C# 2008 Express Edition ausgeliefert und (automatisch bzw. auf Wunsch) installiert werden:

- Microsoft SQL Server 2008 Express Edition
Diese Software erlaubt Client-Server - Lösungen mit wenigen (für uns irrelevanten) Einschränkungen gegenüber der kostenpflichtigen Variante.
- Microsoft SQL Server 2008 Compact Edition
Wenn nur eine lokale Lösung (ohne Mehrbenutzerbetrieb) benötigt wird, ist diese schlanke Software zu empfehlen (Plattenspeicherbedarf: 2 Megabyte).

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 sind vor allem zu nennen:

- ADO wurde für COM-basierte Anwendungen und die zugehörige Datenzugriffsschnittstelle 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.

26.1 Relationale Datenbanken

Heute arbeitet praktisch jedes DBMS mit der **relationalen Datenbankstruktur**, die sich gegenüber alternativen Bauformen (z.B. hierarchische Datenbank, Netzwerkdatenbank) weitgehend durchgesetzt hat. 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 26.2) weitgehend standardisiert. Von praktischer Bedeutung ist auch, dass unserer Entwicklungsumgebungen mit Microsofts RDBMS-Produkten exzellent kooperieren.

26.1.1 Tabellen

Bei einer relationalen Datenbank sind die Daten in **Tabellen**⁸³ angeordnet, wobei die **Zeilen** auch als **Datensätze** (engl: **records**), **Tupel** oder **Fälle** und die **Spalten** auch als **Felder**, **Attribute**, **Merkmale** oder **Variablen** bezeichnet werden:

- Jede Tabelle enthält Fälle 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. Zahl, Datum, Zeichenfolge).
 - 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.

⁸³ 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 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. In manchen Arbeiten zur relationalen Datenbanktechnik wird aber unter einer *Relation* allerdings keine *Tabelle*, sondern eine *Beziehung* zwischen zwei Tabellen verstanden (siehe unten).

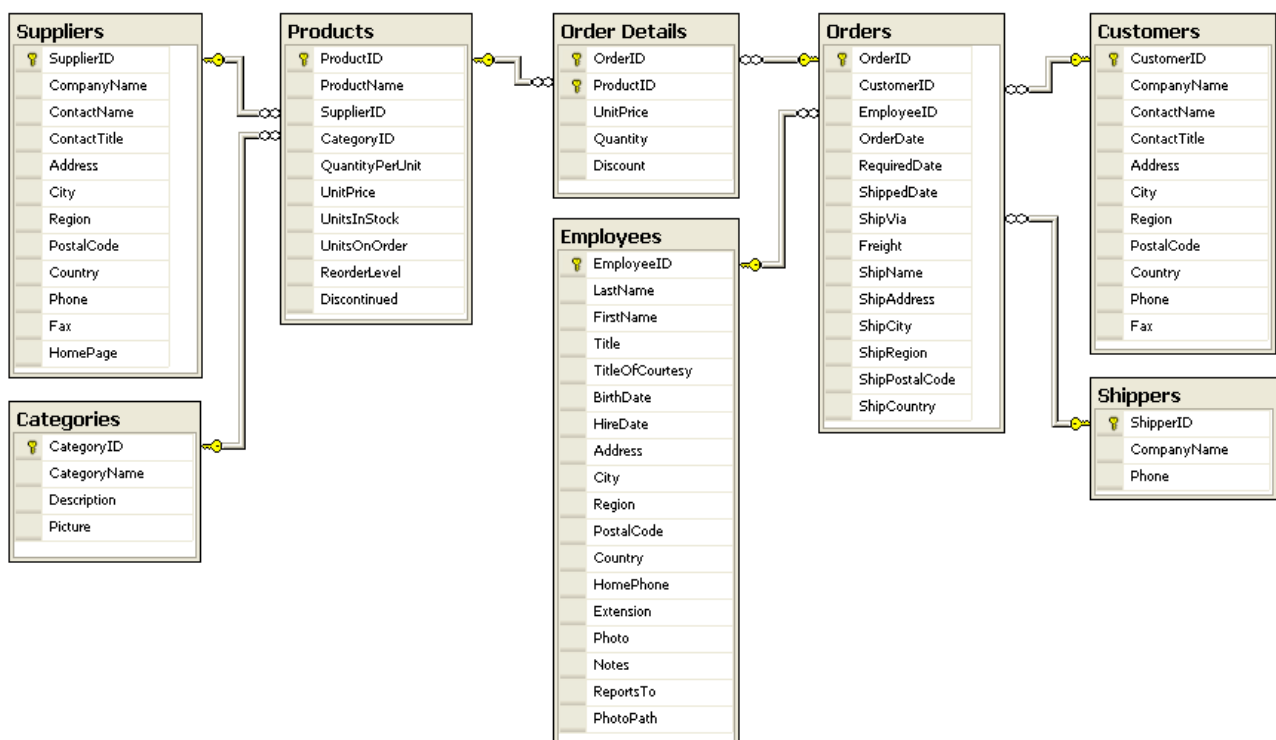
Wir betrachten in Abschnitt 26 die von Microsoft angebotene Beispieldatenbank zu einer erfundenen Firma namens **Northwind** (siehe Abschnitt 26.3.2.2 zum Bezug und zur Installation der Datenbank). In der **Northwind**-Datenbank befindet sich u.a. eine Tabelle mit dem Personal der Firma:

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):



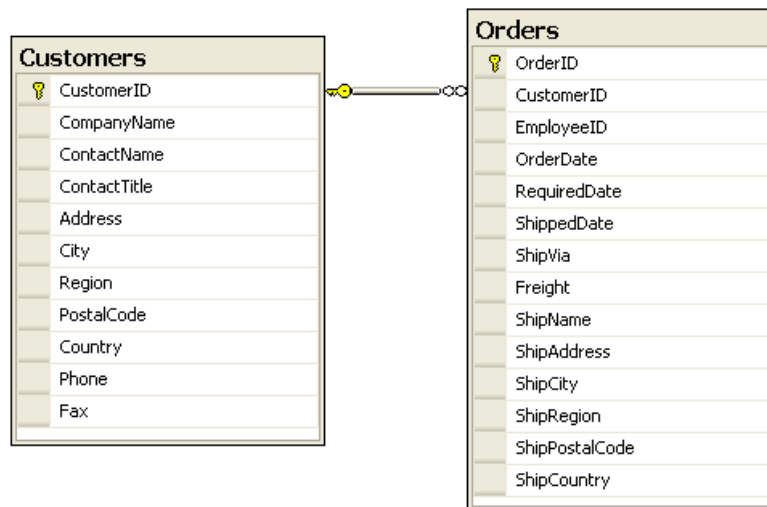
Ein redundanzfreier Datenbankentwurf erfordert bei den meisten Aufgabenstellungen das Verteilen der Daten auf *mehrere* Tabellen. Man spricht hier vom *Normalisieren* einer Datenbank. Damit durch das Verteilen der Daten keine Inkonsistenzen entstehen, müssen Beziehungen zwischen den Tabellen definiert werden.

Die eben zur Illustration eingesetzten Bildschirmfotos stammen von der Visual C# 2008 Express Edition. Wie unsere Entwicklungsumgebung als SQL Server - Frontend genutzt werden kann, erfahren Sie in Abschnitt 26.5.3.

26.1.2 Beziehungen zwischen Tabellen

Zwischen zwei Tabellen (z.B. **Customers** und **Orders**) kann über korrespondierende Felder mit bestimmten Eigenschaften (z.B. über die in beiden Tabellen vorhandenen Felder **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 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.

26.2 SQL

Ein DBMS interagiert mit anderen Softwaresystemen über die **Structured Query Language** (SQL). Microsofts SQL-Server 2008 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.

26.2.1 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 (Data Definition Language)**
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 (z.B. durch Verwendung von Beziehungen) strukturell von den Zeilen der Datenbanktabellen unterscheiden können. Oft spricht man beim SELECT-Befehl von einer *Auswahlabfrage* im Unterschied zu den anschließend behandelten *Aktionsabfragen*.
- **DML (Data Manipulation Language)**
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 26.4.6).

Auf der Webseite

<http://msdn2.microsoft.com/de-de/library/ms365303.aspx>

biete Microsoft eine T-SQL - Referenz und Lernprogramme.

Das folgende Syntaxdiagramm zum SELECT-Befehl beschränkt sich auf die anschließend besprochenen Ausdrucksmittel:

```
SELECT { * | spalten }  
FROM tabellen  
[WHERE log_ausdruck]  
[ORDER BY sortier_kriterium [ASC] DESC] [, sortier_kriterium [ASC] DESC] ...]]  
[GROUP BY gruppierungs_spalte [, gruppierungs_spalte ...]]
```

Im Unterschied zu C# ist in SQL die Groß/Klein-Schreibung irrelevant; es hat sich aber eingebürgert, die Schlüsselwörter groß zu schreiben.

26.2.2 Spalten abrufen

Will man per SELECT-Befehl *alle* Spalten einer Tabelle abrufen, ist ein Stern zu setzen (*). Mehrere einzelne Spalten sind durch ein Komma zu trennen.

In der FROM-Klausel wird die Tabelle genannt, aus der die abgerufenen Spalten stammen.

Beispiel:

```
SELECT EmployeeID, FirstName, LastName FROM Employees
```

26.2.3 Fälle auswählen über die WHERE-Klausel

Mit der WHERE-Klausel der SELECT-Anweisung kann über einen logischen Ausdruck eine Teilmenge von Fällen ausgewählt werden, z.B.:

```
SELECT EmployeeID, FirstName, LastName FROM Employees
WHERE City='London'
```

Für die Suche nach Teilzeichenfolgen bietet SQL den Vergleichsoperator **LIKE**, wobei folgende Jokerzeichen zur Verfügung stehen:

- % ersetzt ein oder mehrere beliebige Zeichen
- _ ersetzt genau ein beliebiges Zeichen

Der folgende Befehl spürt alle Personen auf, deren Nachname mit „K“ beginnt:

```
SELECT FirstName, LastName FROM Employees
WHERE LastName LIKE 'K%'
```

Leere Feldinhalte lassen sich mit dem Schlüsselwort **NULL** ansprechen, z.B.:

```
SELECT FirstName, LastName FROM Employees
WHERE Region IS NULL
SELECT FirstName, LastName FROM Employees
WHERE Region IS NOT NULL
```

Recht praktisch kann auch der Vergleichsoperator **IN** sein:

```
SELECT FirstName, LastName, City FROM Employees
WHERE City IN ('London', 'Seattle')
```

26.2.4 Daten aus mehreren Tabellen zusammenführen

Sobald die FROM-Klausel *mehrere* Tabellen enthält, muss den Namen der abgerufenen Spalten ein Tabellennamen vorangestellt werden, z.B.:

```
SELECT Customers.CompanyName, Orders.OrderDate
FROM Customers, Orders
WHERE Customers.CustomerID = Orders.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.

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
```

Oft wendet man die Operation **INNER JOIN** auf zwei Tabellen an, die in einer Master-Detail – Beziehung stehen.

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
```

26.2.5 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
```

26.2.6 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(<i>col</i>)	Zählt die Werte in der Spalte <i>col</i>
COUNT(*)	Liefert die Anzahl der Zeilen in der Tabelle
SUM(<i>col</i>)	Summiert über die Spalte <i>col</i>
AVG(<i>col</i>)	Mittelt über die Spalte <i>col</i>
MIN(<i>col</i>)	Ermittelt das Minimum in Spalte <i>col</i>
MAX(<i>col</i>)	Ermittelt das Maximum in Spalte <i>col</i>

26.2.7 Daten aggregieren

Über die Klausel **GROUP BY** kann man Untergruppen bilden, für die sich dann 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
```

26.3 Microsoft SQL Server 2008 Express Edition

26.3.1 Eigenschaften

Microsofts SQL Server 2008 Express Edition ist ein voll funktionstüchtiges, zeitlich unbefristet verwendbares Datenbankmanagementsystem (DBMS). Im Vergleich zu den kostenpflichtigen Varianten von Microsofts SQL Server 2008 bestehen folgende Einschränkungen:

- maximale Datenbankgröße 4 GB
- maximal 1 GB Hauptspeicher nutzbar
- nur 1 CPU nutzbar
- Mitwirkung bei der Datenreplikation nur als Klient
- Keine Unterstützung für Benachrichtigungs- und Analysedienste

Von der guten Integration mit der Visual C# 2008 Express Edition (siehe Abschnitt 26.5.3) haben wir u.a. folgende Vorteile:

- bequeme Datenbankprogrammierung
- Ablage von kompletten .NET - Objekten in einer Datenbank

Einer Verwendung der SQL Server 2008 Express Edition in Ihren eigenen Programmen stehen keine lizenzrechtlichen Probleme im Wege, weil Microsoft eine Verbreitung zusammen mit Ihren An-

wendungen nach einer Registrierung (<http://www.microsoft.com/sql/editions/express/default.mspx>) erlaubt.

Weil in diesem Manuskript die meisten Aussagen über Microsofts SQL-Server für *alle* Versionen der Generation 2008 gelten, sparen wir uns in der Regel die umständliche Bezeichnung *Microsoft SQL Server 2008 Express Edition* und reden kurz über den *SQL-Server*.

26.3.2 Installation

26.3.2.1 SQL Server 2008 Express with Tools

Bei der Installation unserer Entwicklungsumgebung Visual C# 2008 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 auf der Webseite

<http://www.microsoft.com/downloads/details.aspx?displaylang=de&FamilyID=7522a683-4cb2-454e-b908-e805e9bd4e28>

kostenlos angebotene Paket *SQL Server 2008 Express with Tools*, das neben der Datenbank auch das Management Studio enthält.

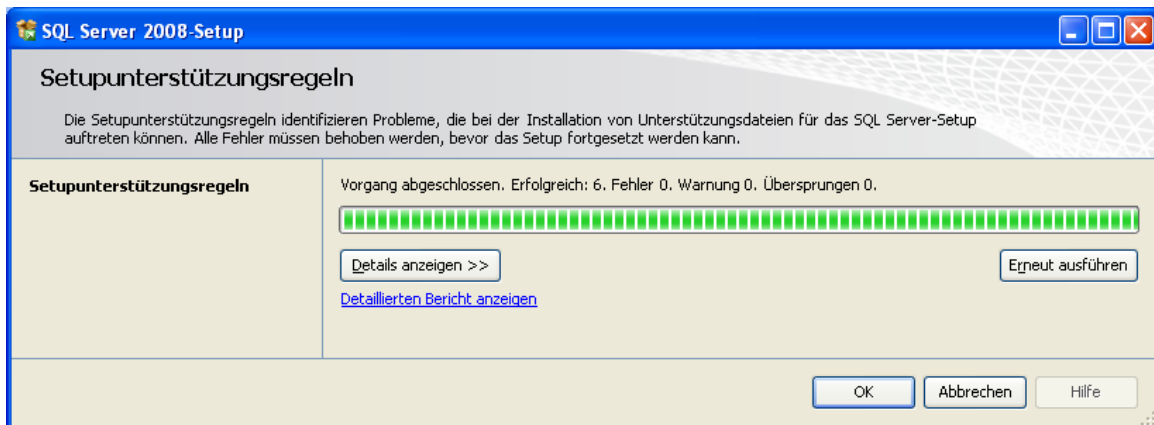
Nach einem Doppelklick auf die herunter geladene Datei **SQLEXPRWT_x86_DEU.exe** startet das **Installationscenter**:



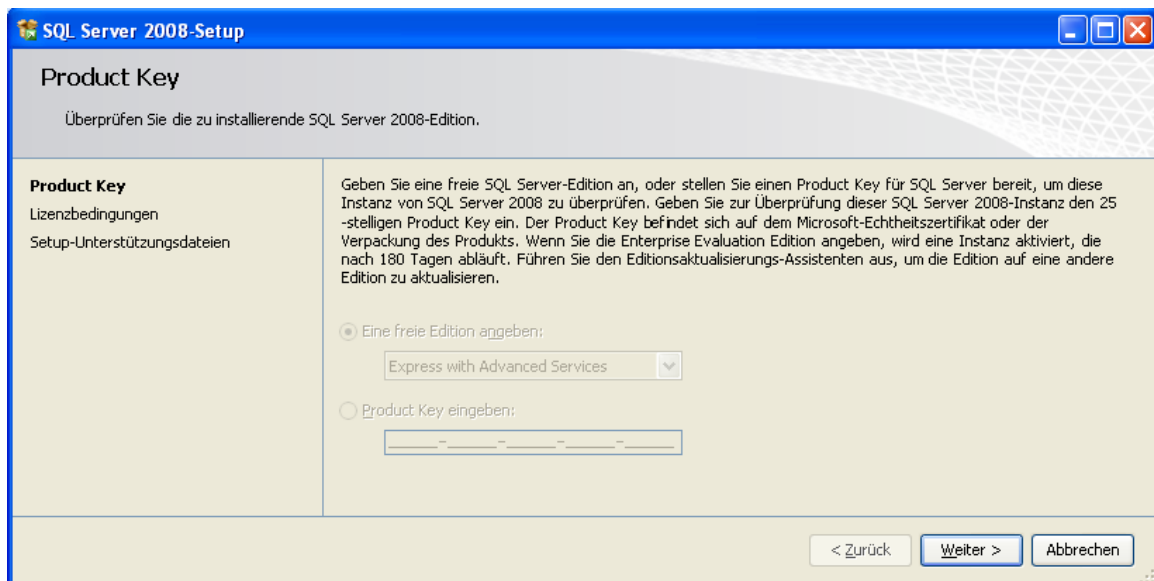
Nach einem Klick auf **Installation** erscheint der folgende Dialog mit Installationsoptionen:



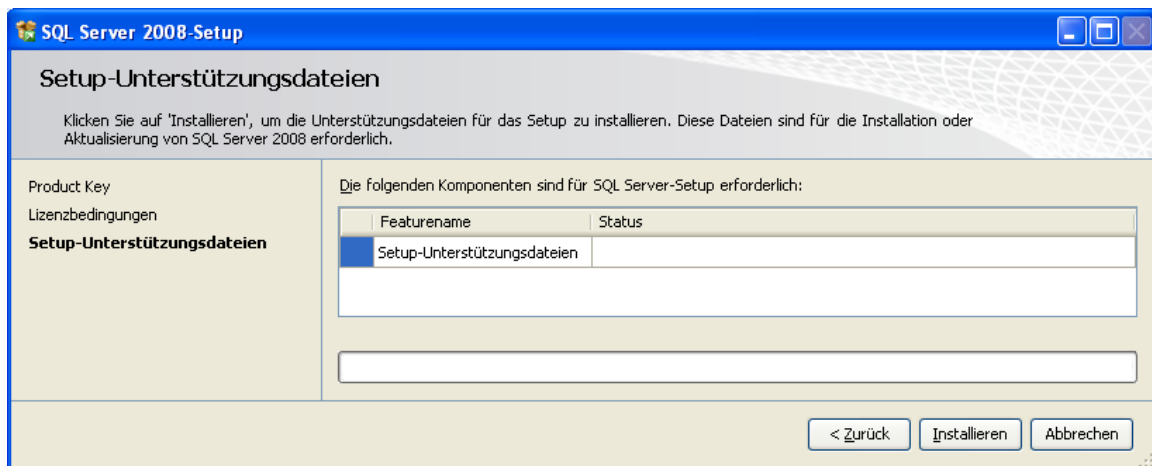
Vermutlich werden auch Sie einen **neuen eigenständigen SQL Server** installieren wollen. Wenn alle Setupunterstützungsregeln erfüllt sind, können Sie den folgenden Dialog mit **OK** quittieren:



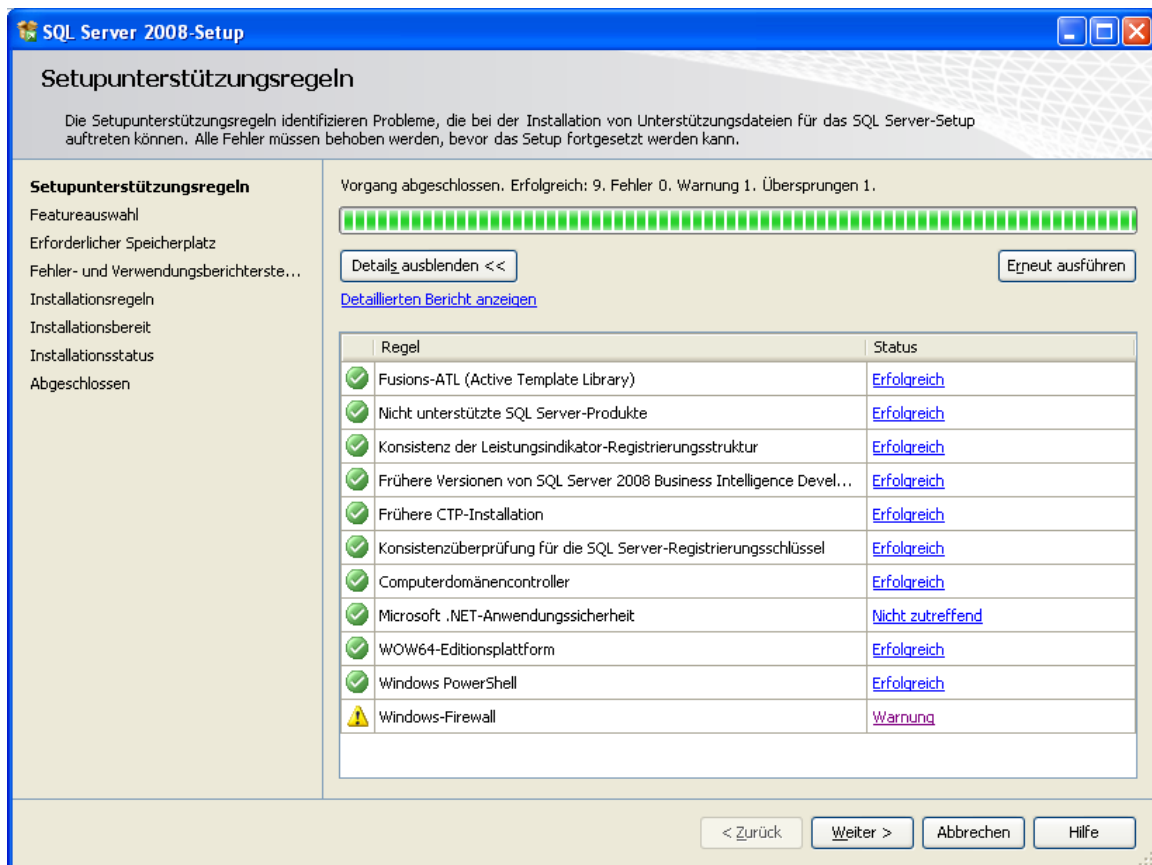
Machen Sie hier einfach **weiter**:



Akzeptieren Sie die Lizenzbedingungen, und lassen Sie die **Setup-Unterstützungsdateien** installieren:



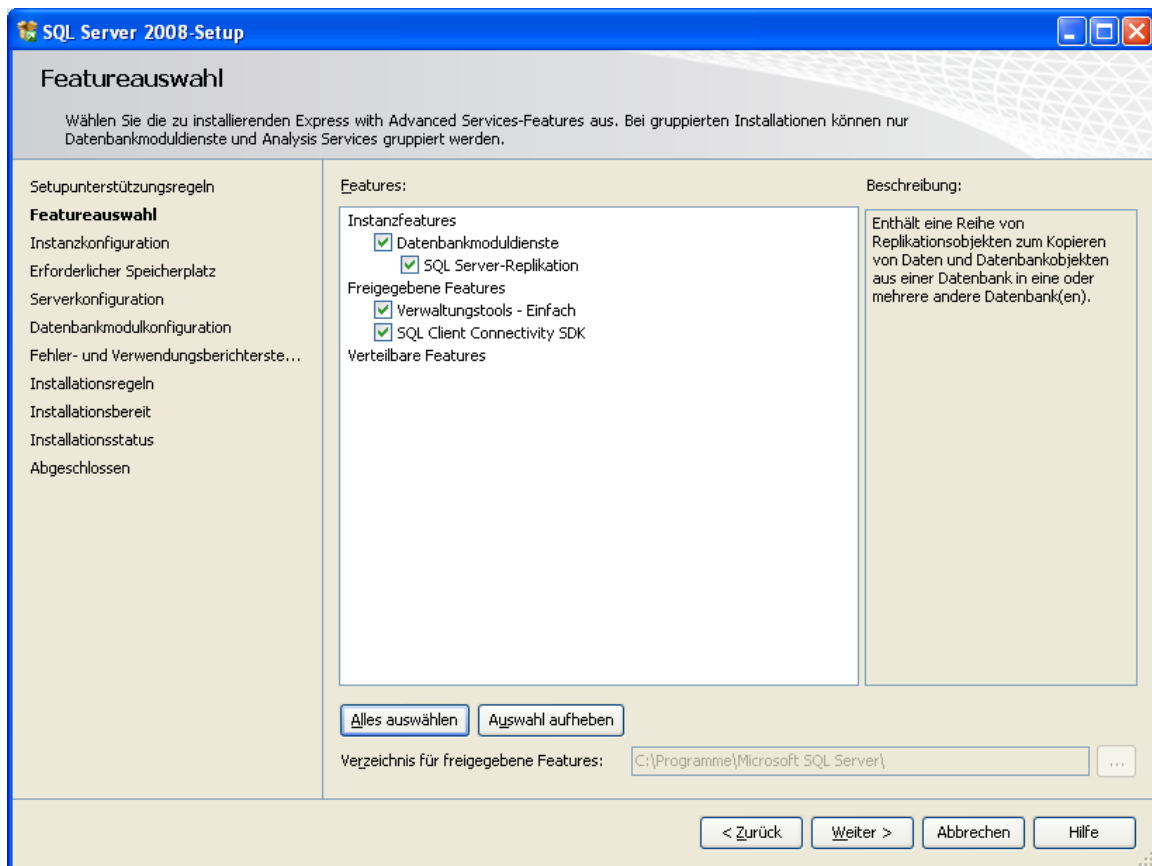
Bei der Prüfung des nächsten Pakets von Setup-Unterstützungsregeln



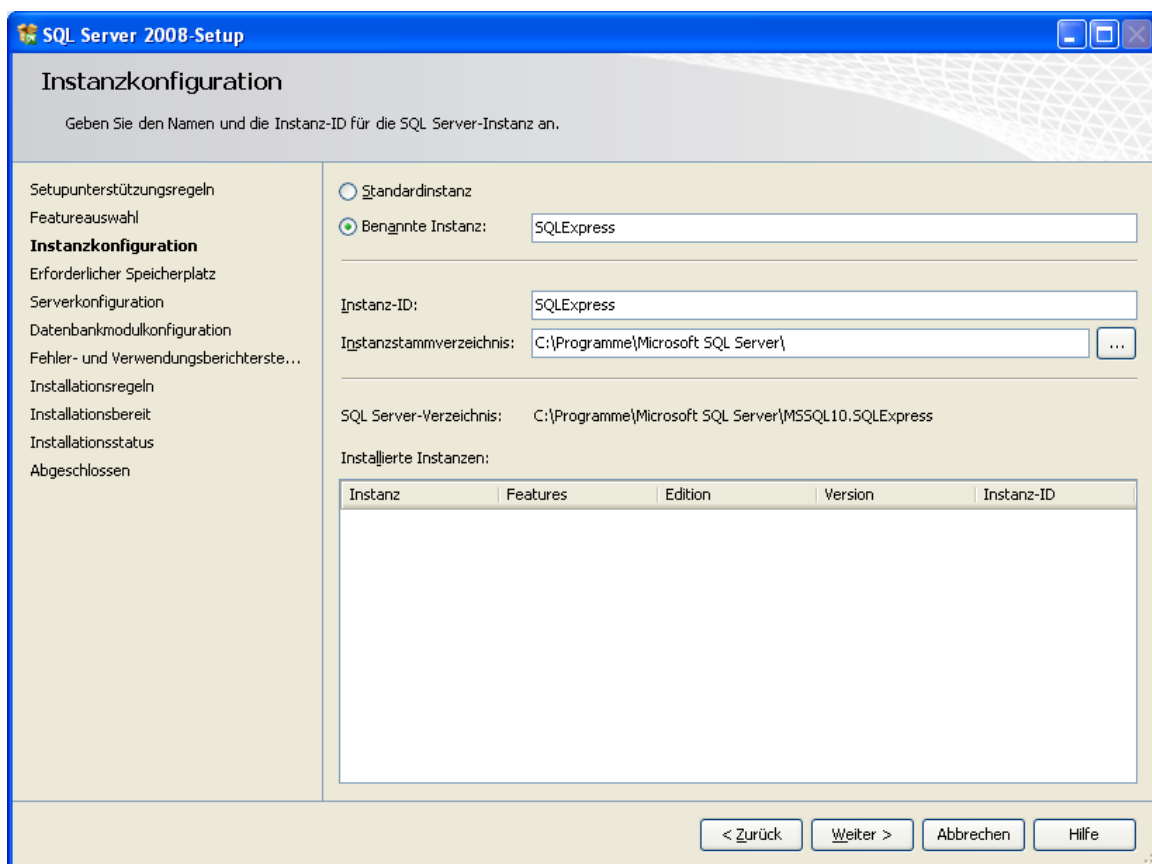
erscheint bei einem korrekt gesicherten Rechner eine Warnung:



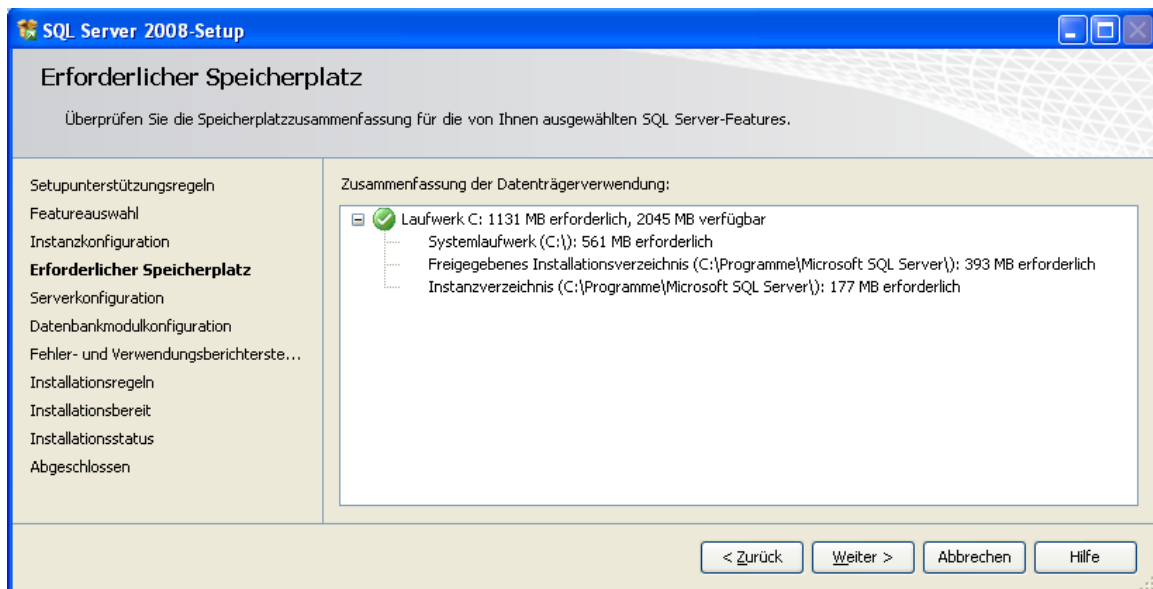
Wir machen **weiter**, und wählen im Dialog **Featureauswahl** alle Optionen:



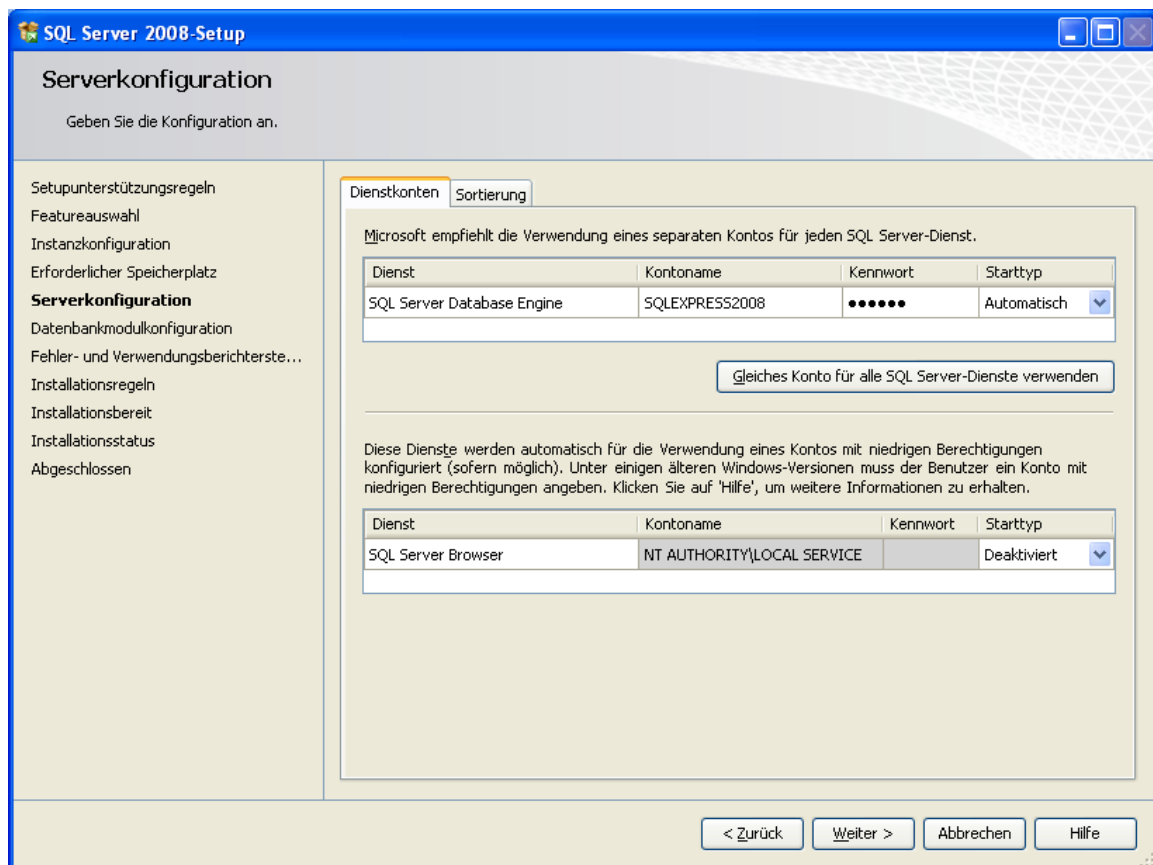
Im Dialog **Instanzkonfiguration** übernehmen wir die Voreinstellungen:



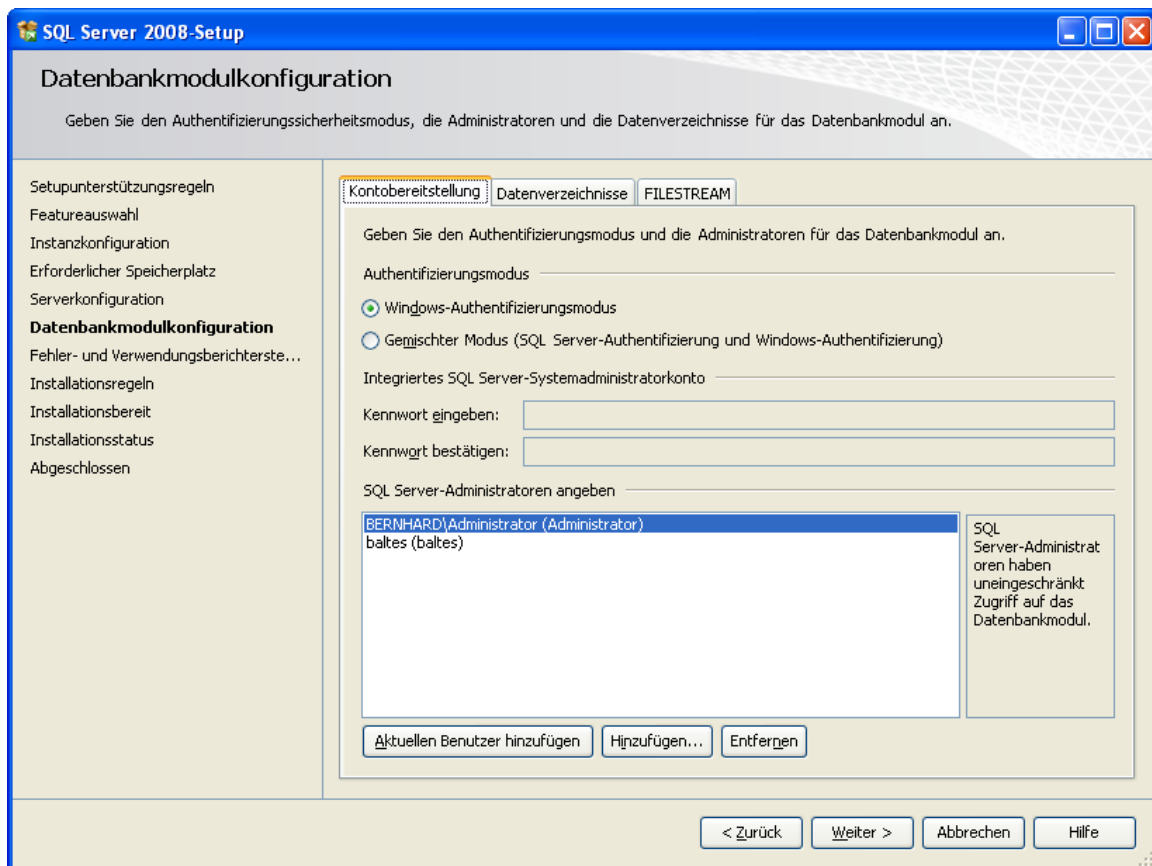
Der SQL Server 2008 Express macht sich auf der Festplatte ganz schön breit:



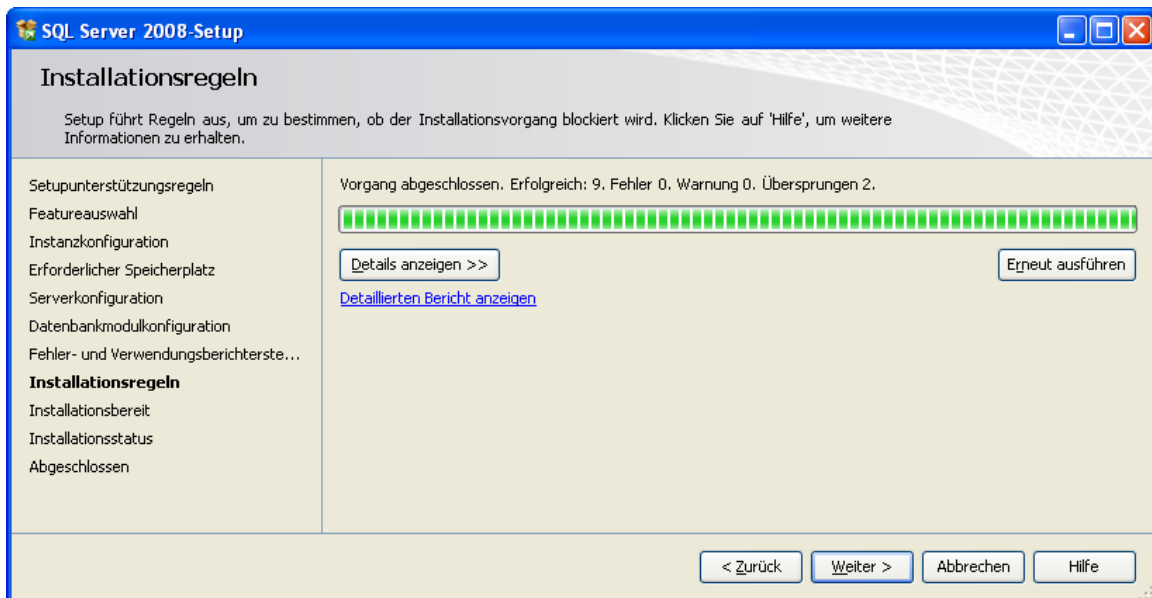
Der Datenbankdienst sollte unter einem speziellen Konto ausgeführt werden, das nur die benötigten Rechte besitzt:



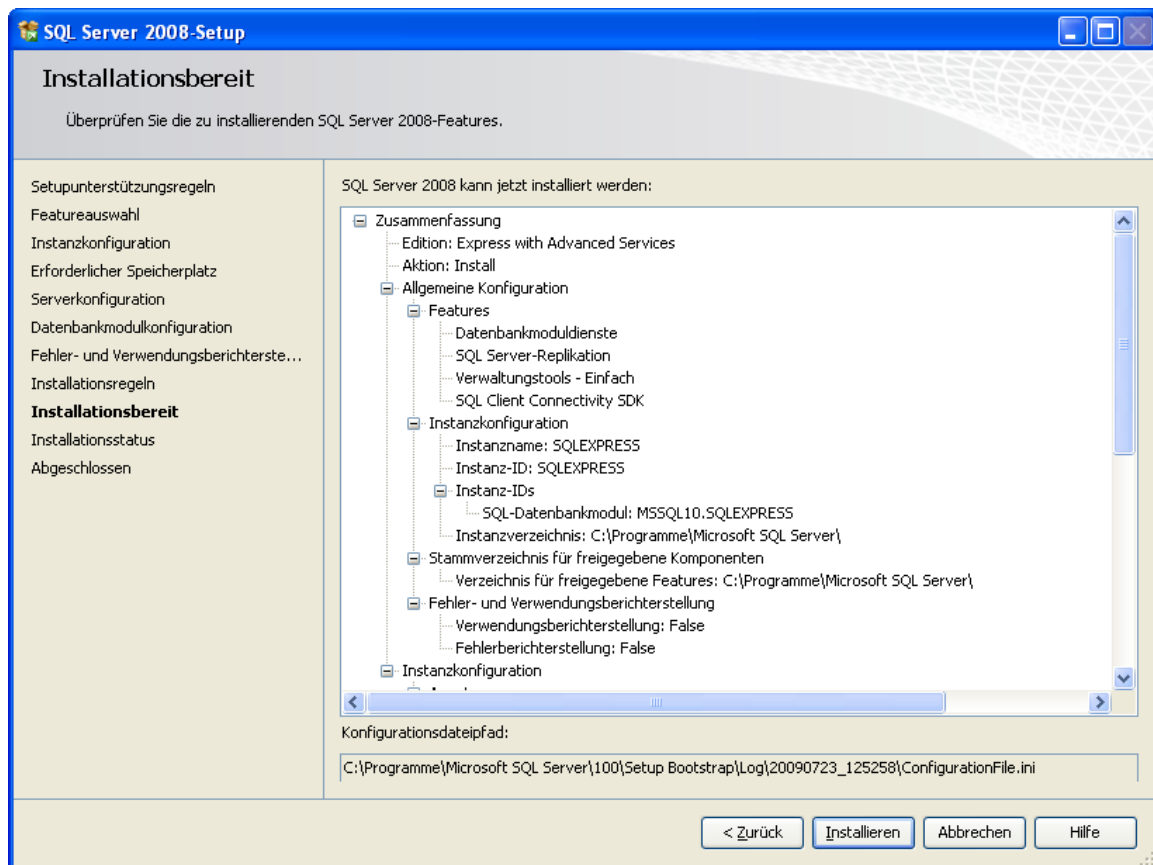
Im nächsten Dialog ist der Authentifizierungsmodus zu wählen und (mindestens) ein Server-Administrator einzutragen:



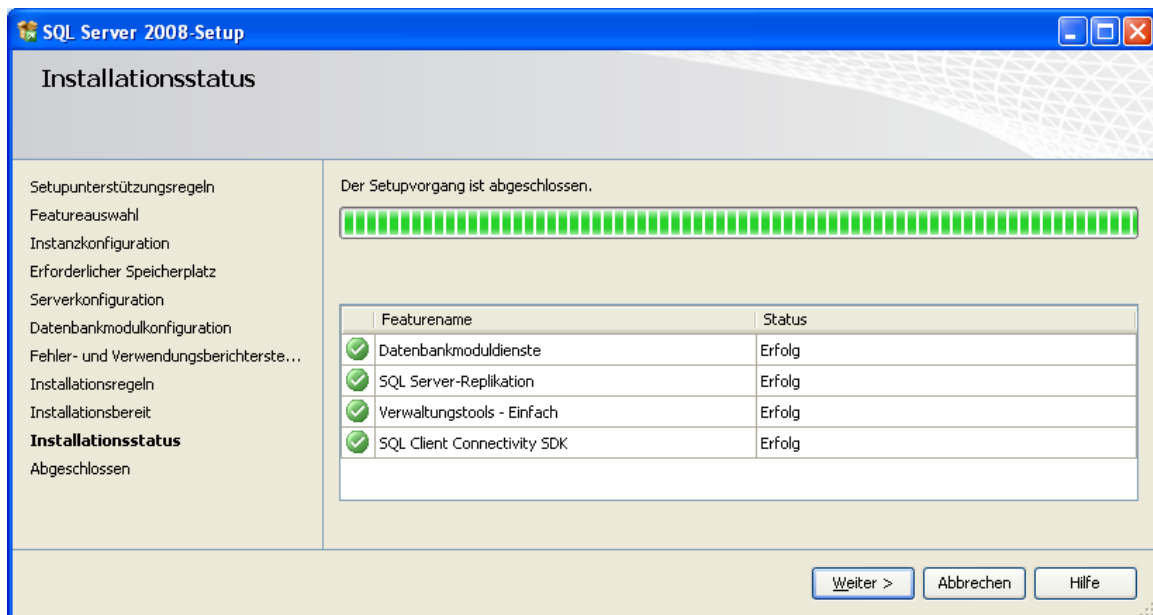
Wir verzichten auf die **Fehler- und Verwendungsberichterstellung** und nehmen mit Freude zur Kenntnis, dass alle **Installationsregeln** erfüllt sind:



Hier kann der Startschuss endlich gegeben werden:



Was lange währt, wird endlich gut:



26.3.2.2 Beispieldatenbank Northwind

Auf der Webseite

<http://www.microsoft.com/downloads/details.aspx?FamilyId=06616212-0356-46A0-8DA2-EEBC53A68034&displaylang=en>

bietet Microsoft die Beispieldatenbanken **Northwind** und **Pubs** für seine SQL Server 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**.

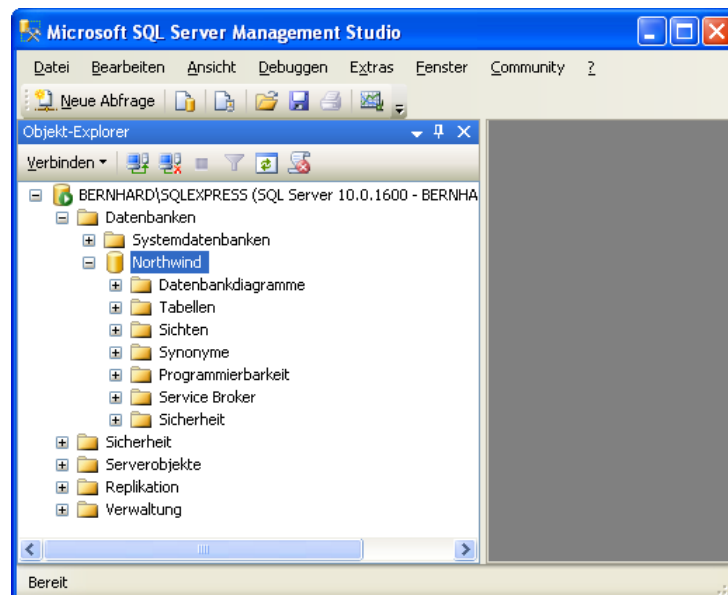
Eine vorhandene Datenbank muss dem SQL Server bekannt gemacht werden, damit er sie verwalten kann, was auf unterschiedliche Weise geschehen kann. In der Readme-Datei zu den Beispieldatenbanken sind einige Verfahren beschrieben. Wir verwenden das eben installierte Server Management Studio Express, um die Beispieldatenbank **Northwind** in die Verwaltung des SQL Servers zu integrieren:

- Starten Sie das Management Studio über den Link
SQL Server Management Studio
in der Programmgruppe **Microsoft SQL Server 2008**.
- Stellen Sie im folgenden Dialog 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 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.

26.3.3 Konfiguration

26.3.3.1 Rechteverwaltung

Bei der Verwaltung von Rechten geht es um ...

- **Authentifizierung**
Welche Benutzer, Gruppen oder Dienste können 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.⁸⁴ 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?

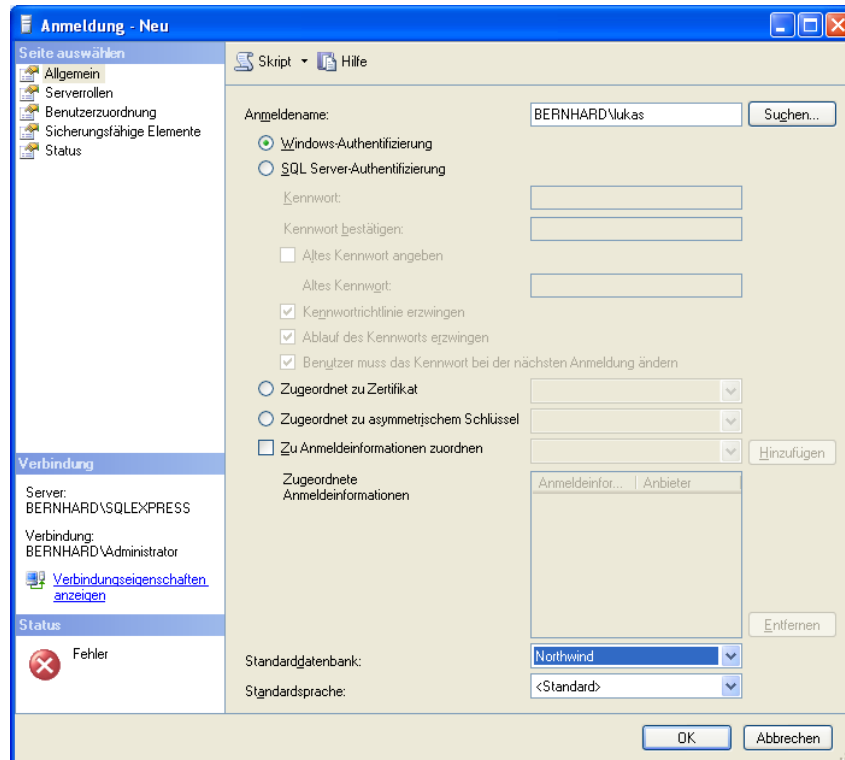
Authentifizierung

Gehen Sie folgendermaßen vor, um einem Windows-Konto den Zugriff auf den SQL Server zu erlauben:

- Starten Sie das **SQL Server Management Studio Express** mit Administratorrechten, und stellen Sie die Verbindung zum SQL Server 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.

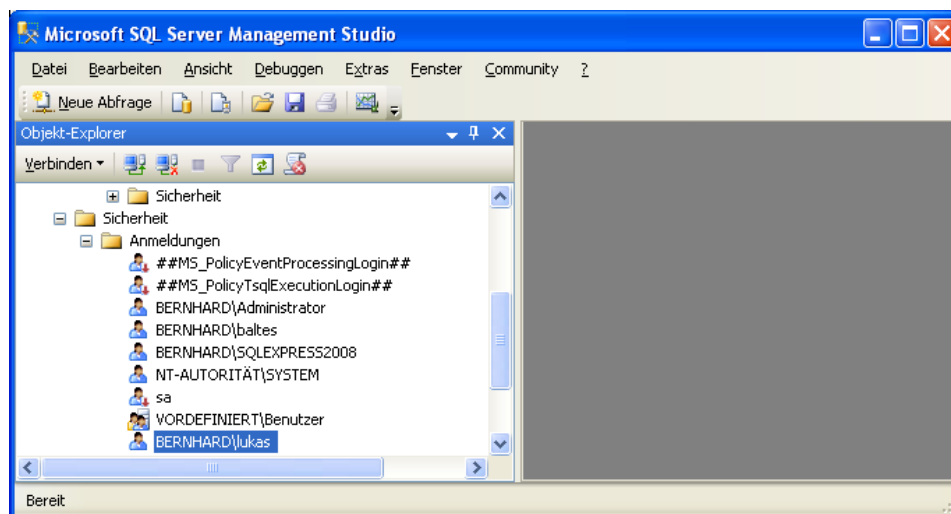
⁸⁴Die SQL Server 2008 *Express Edition* bietet die Option, für den angemeldeten Windows-Benutzer eine so genannte *Benutzerinstanz* des SQL Servers dynamisch zu erzeugen und eine private Datenbank dynamisch einzubinden (vgl. Abschnitt 26.4.2.1). Bei dieser Einsatzart ist keine vorbereitende Sicherheitskonfiguration erforderlich.

- Nach Aufnahme eines Benutzers sieht die Dialogbox **Anmeldung - Neu** so aus:



- Wählen Sie für den Benutzer eine **Standarddatenbank**.

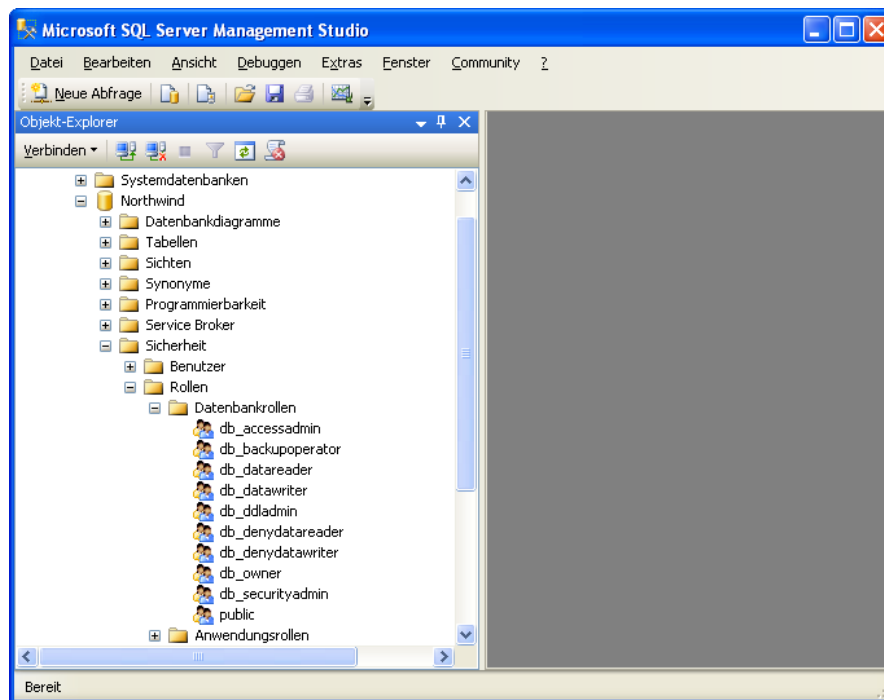
Der neue Benutzer wird im Ordner **Sicherheit > Anmeldungen** des Objekt-Explorers aufgelistet:



Autorisierung durch Rollen

Zur Verwaltung von Benutzerrechten verwendet der SQL Server **Rollen**, die den Benutzergruppen des Betriebssystems vergleichbar sind. Es gibt ...

- Serverrollen**
Diese sind vordefiniert und beziehen sich auf den gesamten SQL-Server. Im **SQL Server Management Studio Express** 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.:

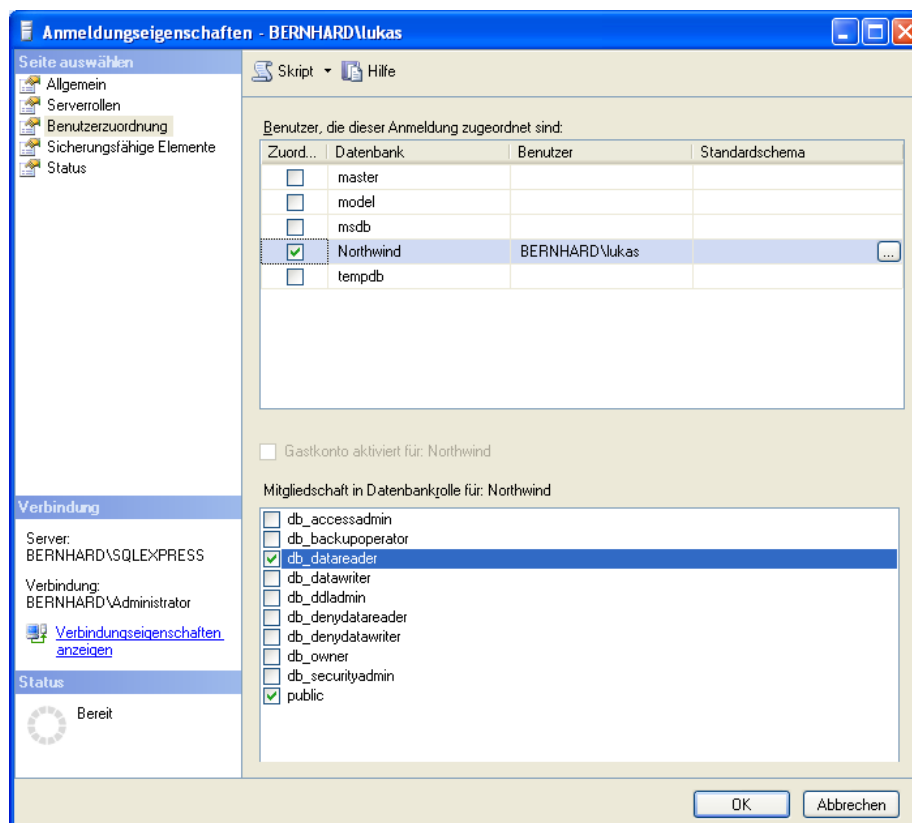


Die vordefinierten Datenbankrollen können durch Eigenkreationen ergänzt werden.

- **Anwendungsrollen**

Zu einer Datenbank können auch Anwendungsrollen definiert werden, um Zugriffe auf bestimmte Anwendungen zu beschränken.

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 **Anmeldeeigenschaften**. Wir verzichten auf die Vergabe von **Ser-verrollen**, 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.

SQL Server - Authentifizierung erlauben

In einer Netzwerkumgebung *ohne* Windows - Domänencontroller (so genannte Arbeitsgruppe) 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 - Authentifizierung zu erlauben.

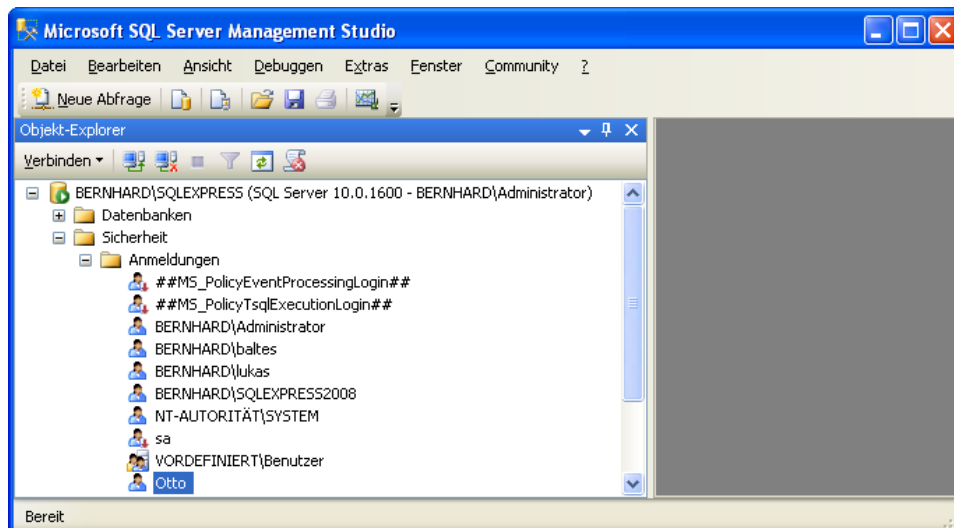
Nach der Standardinstallation unterstützt der SQL Server 2008 Express nur die Windows - Authentifizierung. Im Management Studio Express lässt sich jedoch die SQL - Authentifizierung nachträglich zuschalten:

- Wählen Sie im Kontextmenü zur SQL-Serverinstanz das Item **Eigenschaften**.
- 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.

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.

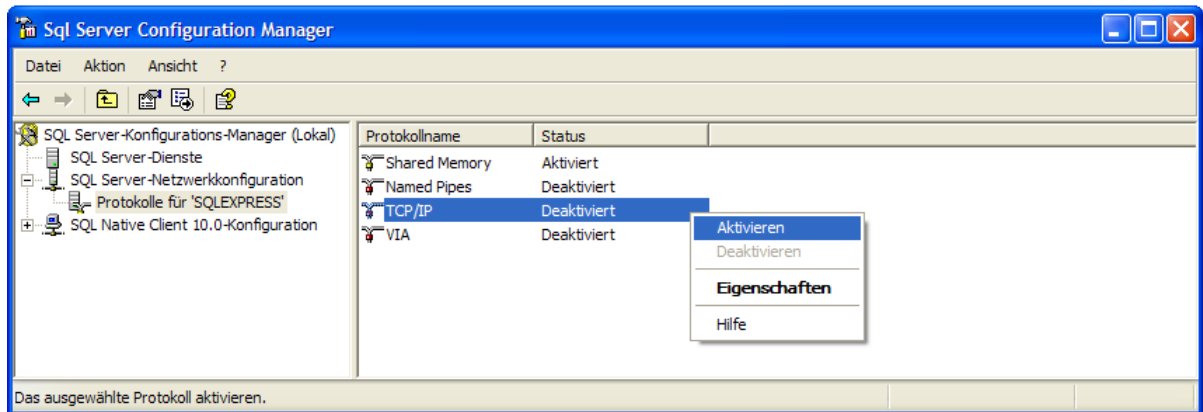
26.3.3.2 Netzwerkzugriff via TCP/IP erlauben

Per Voreinstellung steht der SQL-Server nur lokal angemeldeten Benutzern zur Verfügung, was für die Zwecke von Abschnitt 26 meist genügt. Trotzdem soll an dieser Stelle erläutert werden, wie man Netzwerkzugriff auf den SQL Server erlaubt. Dazu sind drei Maßnahmen erforderlich:

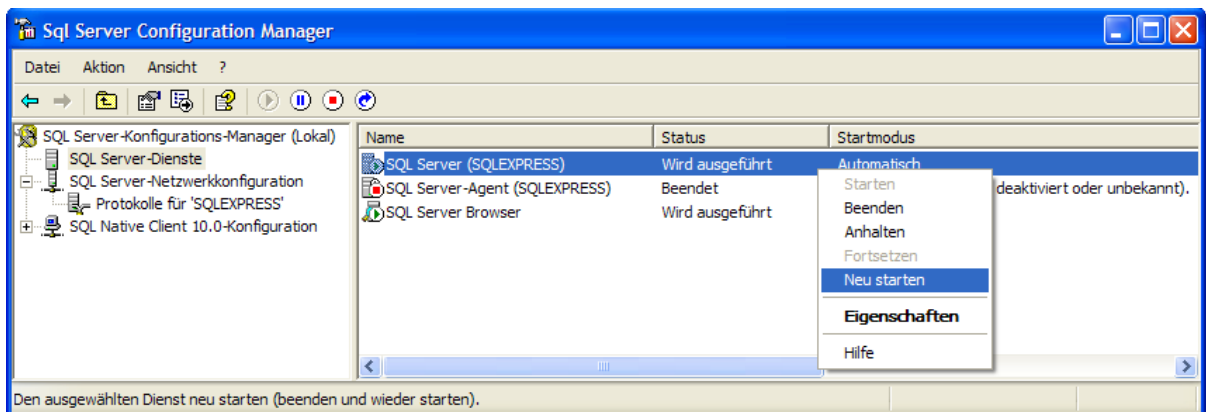
a) Netzwerkprotokoll TCP/IP aktivieren

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-Netzwerkconfiguration**.
- Aktivieren Sie im rechten Segment des Dialogs das **TCP/IP** - Protokoll per Kontextmenü:



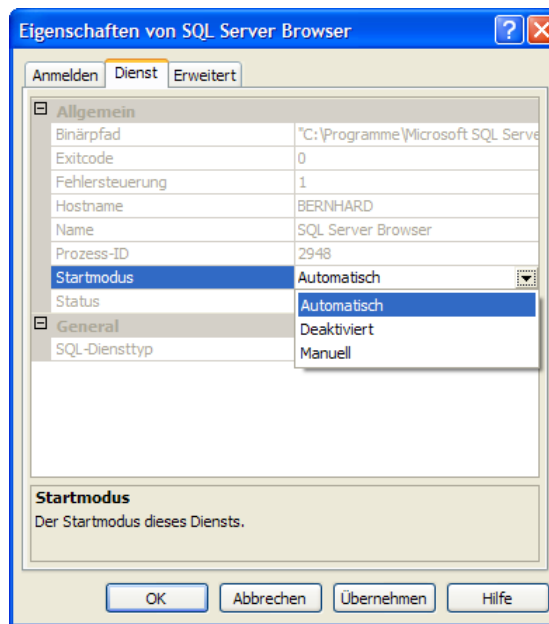
- Wählen Sie in der Baumansicht den Knoten **SQL Server-Dienste**, und starten Sie per Kontextmenü den Dienst **SQL Server (SQLEXPRESS)** neu:



b) SQL Server Browser Dienst starten

Wenn der SQL-Server ohne Angabe einer Portnummer in der Verbindungszeichenfolge (vgl. Abschnitt 26.4.2) 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 XP so geschehen kann:

- Starten Sie das Systemsteuerungs-Applet zur Verwaltung der Windows-Firewall über **Start > Systemsteuerung > Windows-Firewall**, klicken Sie auf das Registerblatt **Ausnahmen** und auf den Schalter **Programm**.
- Klicken Sie auf **Durchsuchen**, und wählen Sie das für den SQL Server Dienst zuständige Programm, z.B.:
`C:\Programme\Microsoft SQL Server\MSSQL10.SQLEXPRESS\MSSQL\Binn\sqlservr.exe`
- Quittieren Sie mit **Öffnen** und **OK**.
- Um das Sicherheitsrisiko gering zu halten, sollten Sie möglichst wenigen Rechnern erlauben, den Dienst zu benutzen. Markieren Sie das eben berechtigte Programm, klicken Sie auf **Bearbeiten** und **Bereich ändern**, und wählen Sie z.B. die Option **Nur für eigenes Netzwerk (Subnetz)**.
- Erstellen Sie analog eine Ausnahme für das Programm
`C:\Programme\Microsoft SQL Server\90\Shared\sqlbrowser.exe`
 zum SQL Server Browser - Dienst.

26.4 ADO.NET

26.4.1 Überblick

26.4.1.1 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 26.4.1.2) 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.

26.4.1.2 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 ab Version 2.0 enthält Provider für die folgenden Datenbankmanagementsysteme bzw. Datenbankschnittstellen:

- **SqlClient**
Dieser Provider unterstützt SQL-Server und ist speziell optimiert für Microsofts SQL Server ab Version 7.0.
- **SqlServerCe**
Dieser Provider unterstützt Microsofts SQL Server 2008 Compact Edition.
- **OracleClient**
Dieser Provider unterstützt die Datenbanktechnik der Firma Oracle. Weil Microsoft die Weiterentwicklung eingestellt hat⁸⁵, sollte man wohl besser auf den von Oracle kostenlos angebotenen Provider setzen.⁸⁶
- **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.

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 26.4.2)
- **SqlCommand**
Repräsentiert ein SQL-Kommando (siehe Abschnitt 26.4.3)
- **SqlDataAdapter**
Diese Klasse vermittelt zwischen der Datenbankverbindung und der lokalen **DataSet**-Objekt (siehe Abschnitt 26.4.4).
- **SqlDataReader**
Erlaubt ein lesendes, sequentielles Durchlaufen der angeschlossenen Datenbank (siehe Abschnitt 26.4.8)
- **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.

⁸⁵ Siehe Webseite: <http://blogs.msdn.com/adonet/archive/2009/06/15/system-data-oracleclient-update.aspx>

⁸⁶ Siehe Webseite: <http://www.oracle.com/technology/tech/windows/odpnet/index.html>

26.4.1.3 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 providerunabhängige 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 26.1.2).

- **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 Unternamensraum zum eingesetzten Provider importieren.

26.4.2 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. Andere Provider bieten analog benannte Klassen. Wir beschränken uns anschließend auf die Verbindung zu einem SQL-Server.

26.4.2.1 Verbindungszeichenfolge

Um die Verbindung zu einer bestimmten Datenbank herstellen zu können, benötigt ein **SqlConnection**-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 zahl-

reiche Varianten in Frage.⁸⁷ Ein **ConnectionString** zur Verbindung mit einem SQL-Server enthält in der Regel folgende Angaben:

- **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:

a) Der SQL-Server läuft auf demselben Rechner wie die Klientenanwendung

Verwenden Sie für den Wert die Syntax

Servername\Instanzname

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 Express Edition wird per Voreinstellung mit dem Instanznamen **SQLEXPRESS** installiert.

Beispiel:

Data Source=(local)\SQLEXPRESS

b) Der SQL-Server wird über ein Netzwerk per TCP/IP - Protokoll angesprochen

Ist ein DBMS auf einem anderen Rechner über das TCP/IP - Protokoll (an einem bestimmten Port) erreichbar (vgl. Abschnitt 26.3.3.2 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 26.3.3.2).

Beispiel:

Server=tcp:BBGVM01\SQLEXPRESS

- **Datenbank**

Die zu verwendende Datenbank kann über ihren logischen Namen (beim SQL-Server) oder über ihren Dateinamen angesprochen werden.

a) 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

b) AttachDbFilename bei Benutzerinstanzen

Bei dynamisch erzeugten SQL Server - Benutzerinstanzen (siehe unten) gibt man den **mdf**-Dateinamen als Wert zum Parameter **AttachDbFilename** an, z.B.:

⁸⁷ Es sind sogar spezialisierte Internet-Angebote mit Vorschlägen zur Bildung von korrekten Verbindungszeichenfolgen entstanden, z.B. <http://www.connectionstrings.com/>

```
AttachDbFilename=|DataDirectory|\NORTHWND.MDF
```

Die Datenbankdatei kann über das mit .NET 2.0 eingeführte 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 oder auf einem NAS-Server (*Network Attached Storage*) werden also nicht akzeptiert.⁸⁸

- **Benutzer - Authentifizierung**

Wie Sie schon aus Abschnitt 26.3.3.1 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
```

- **Benutzerinstanzen der Microsoft SQL Server 2008 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 26.3.3.1) 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). Um eine sehr gut portable Lösung zu erhalten, sollte man außerdem die Windows - Authentifizierung verwenden:

```
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 XP in

```
...\Lokale Einstellungen\Anwendungsdaten\Microsoft\Microsoft SQL Server Data\SQLEXPRESS
```

Benutzerinstanzen der SQL Server 2008 Express Edition eignen sich gut für Datenbanken, die eng mit einer Anwendung verbunden und nur von *einer* Person benutzt werden sollen.

⁸⁸ Diese Regel mit dem gelegentlich empfohlenen Service-Startparameter -T1807 außer Kraft zu setzen, ist mir nicht gelungen.

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: 2 Megabyte).

- **Zugriffsoptionen**

Über den Parameter **Connect Timeout** legt man fest, wie lange auf eine erfolgreiche Verbindung zum SQL Server gewartet werden soll (Voreinstellung: 15 Sekunden), z.B.:

```
Connection Timeout=3
```

Beim Datenbankzugriff über 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**.

Weitere Hinweise zur Verbindungszeichenfolge

- Die Groß-/Kleinschreibung ist irrelevant.
- Einige schreibgeschützte Eigenschaften der **Connection**-Klassen (z.B. die **Database**-Eigenschaft von **SqlConnection**-Objekten) lassen sich über den **ConnectionString** ändern.
- 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.

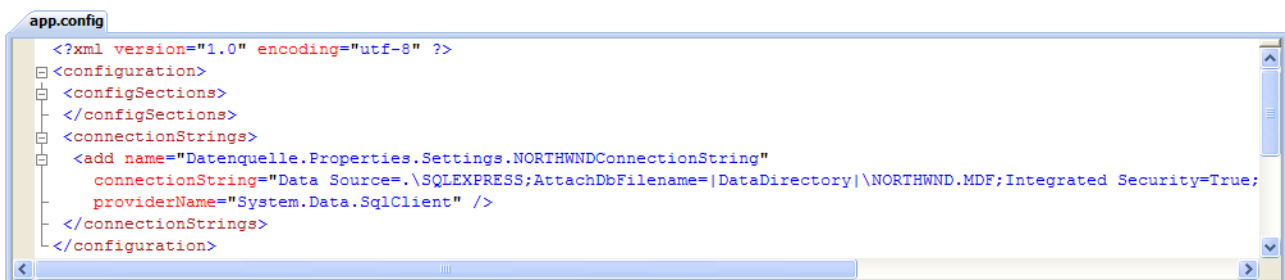
Am Anfang des aktuellen Abschnitts war eine komplette Verbindungszeichenfolge für den Zugriff auf eine lokale SQL Server - Benutzerinstanz zu sehen. Hier folgt nun ein Beispiel für den Netzwerkzugriff auf die primäre Instanz eines SQL Servers bei Verwendung der Windows - Authentifizierung:

```
dbConnection.ConnectionString = @"Server=tcp:BBGVM01\SQLEXPRESS;" +  
    "Database=Northwind;Integrated Security=true";
```

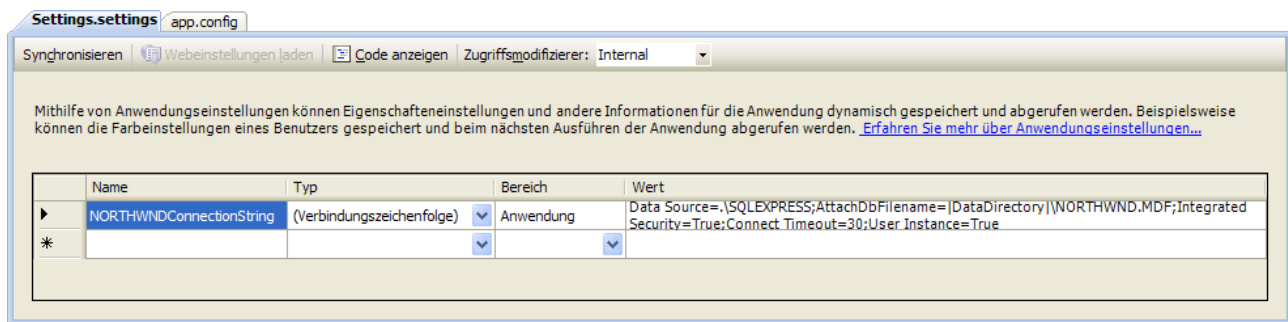
Bei der Microsoft SQL Server 2008 Compact Edition kann die Verbindungszeichenfolge besonders einfach ausfallen und sich z.B. auf die Angabe des Datenbankdateinamens beschränken:

```
dbConnection.ConnectionString = "Data Source=Northwind.sdf";
```

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 19.1). Dieses Exemplar



wurde von der Visual Studio 2008 Express Edition zu dem im Abschnitt 26.5.2 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 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.

26.4.2.2 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. **SqlDataReader**, vgl. Abschnitt 26.4.8) ist das explizite Öffnen unumgänglich. Beim verbindungslosen Arbeiten mit der Klasse **SqlDataAdapter** (siehe Abschnitt 26.4.4) 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.

26.4.3 Die Command-Klassen

Für SELECT- oder DML-Befehle (UPDATE, INSERT, DELETE) an einen Datenbankserver (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.

26.4.3.1 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 26.4.3.2 über parametrisierte Abfragen ausführlich 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 26.4.4) 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 26.4.8).
- **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();
```

26.4.3.2 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 des 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()));
        } catch {
        } 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 Liefertennummer 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 den üblichen Kolle-

tionsmethoden 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**).

26.4.4 Die DataAdapter-Klassen

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**

Beim Aufruf der **DataAdapter**-Methode **Fill()** 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 Kommandostapel übrigens *nicht* verarbeiten.

Die im folgenden Beispiel verwendete **Fill()** - Überladung erwartet den Namen des **DataSet**-Objekts und einen 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
```

trotz der eindeutigen Angabe im SELECT-Kommando den Wert **Table**. Es ist nicht unbedingt erforderlich, für die **DataSet**-Tabelle den Namen der Datenbanktabelle zu übernehmen, aber in der Regel doch empfehlenswert.

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 eine nullbasierte Datensatznummer sowie die Anzahl der gewünschten Datensätze angeben, z.B.:

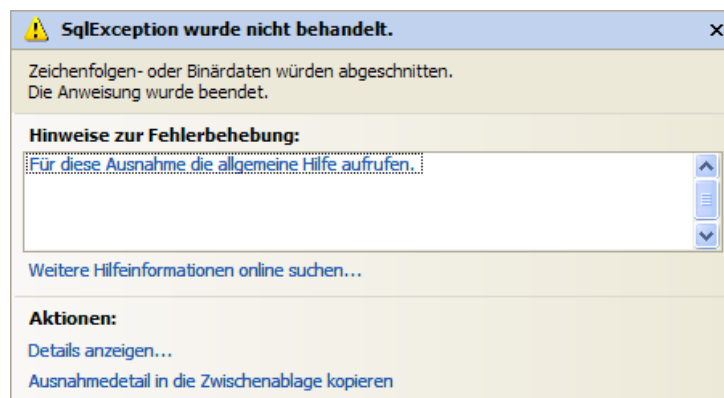
```
dbAdapter.Fill(ds, 2, 5, "Employees");
```

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 SQL Server-Datenbank zu übertragen, z.B.:

```
dbAdapter.Update(ds);
```

In Abschnitt 26.4.6 ist zu erfahren, wie die DML-Befehle per **CommandBuilder** entstehen, und was bei **DataSet**-Objekten mit mehreren Tabellen zu beachten ist.

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 so genannte *Schema* der Datenbank mit den Gültigkeitsregeln und Primärschlüsseldefinitionen zu den Tabellen in ein lokales **DataSet** zu übertragen, kann man die **DataAdapter**-Methode **FillSchema()** verwenden, z.B.:

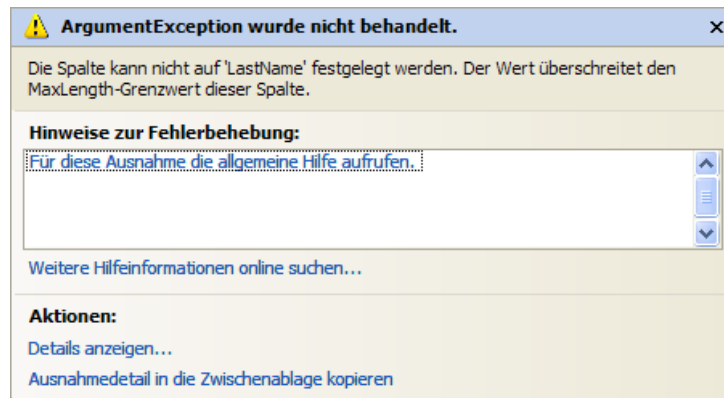
```
dbAdapter.FillSchema(ds, SchemaType.Mapped);
```

Um für aussagekräftig benannte **DataSet**-Tabellen Schemainformationen zu übernehmen, sollte man zunächst über die **TableMappings**-Eigenschaft des **DataAdapter**-Objekts für die Namenszuordnung sorgen (siehe oben) und danach im **FillSchema()**-Aufruf über den zweiten Parameter **SchemaType.Mapped** wählen. Beim alternativen Aktualparameterwert **SchemaType.Source** verwendet **FillSchema()** die Standardtabellennamen (`Table`, `Table1`, ...).

Existieren im **DataSet** noch keine **DataTable**-Objekte mit passenden Namen, dann legt **FillSchema()** diese an. Zur Aufnahme der Schemainformationen werden nötigenfalls in den Tabellen

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 zeitaufwändige Datenbankverbindung erkannt:



Statt die **DataAdapter**-Methode **FillSchema()** aufzurufen, kann man die **DataAdapter**-Eigenschaft **MissingSchemaAction** auf den Wert **AddWithKey** der Enumeration **MissingSchemaAction** setzen, z.B.:

```
dbAdapterCust.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,⁸⁹ ...

- 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 26.4.2.2).

26.4.5 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 (vgl. Abschnitt 26.4.1.1) lokale Kopien der angeforderten Datenbanktabellen samt Restriktionen und Beziehungen.

26.4.5.1 Abfrageergebnisse und Schema-Informationen aus einer Datenbank übernehmen

In der folgenden Sequenz 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;
. . .
```

⁸⁹ Siehe Webseite <http://support.microsoft.com/?scid=kb%3Ben-us%3B310128&x=10&y=7>

```

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);
dbAdapterCust.TableMappings.Add("Table", "Customers");
dbAdapterOrd = new SqlDataAdapter(selCommandOrd);
dbAdapterOrd.TableMappings.Add("Table", "Orders");

dbAdapterCust.MissingSchemaAction = MissingSchemaAction.AddWithKey;
dbAdapterOrd.MissingSchemaAction = MissingSchemaAction.AddWithKey;

try {
    dbConnection.Open();
    dbAdapterCust.Fill(ds);
    dbAdapterOrd.Fill(ds);
} 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 **DataSet**s. Im Beispielprogramm werden der Bequemlichkeit halber **DataTable**-Referenzvariable zu den beiden Tabellen angelegt. Weil wir **TableMappings** erstellt haben (vgl. Abschnitt 26.4.4), können wir die Tabellen im **DataSet** über sinnvolle Namen ansprechen.

Mit dem folgenden Methodenaufruf

```

Console.WriteLine(
    "MaxLength(Company Name)      = " + dtCustomers.Columns["Company Name"].MaxLength +
    "\nPrimärschlüssel(Customers) = " + dtCustomers.PrimaryKey[0].ColumnName);

```

kann man sich davon überzeugen, dass per **FillSchema()**-Aufruf wichtige Merkmale der Datenbanktabellen in die zugehörigen **DataTable**-Objekte 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. Dieser Kollektion fügt die **FillSchema()**-Methode ggf. ein **UniqueConstraint**-Objekt hinzu.

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().ToString(),
    dtOrders.Rows[0]["CustomerID"].GetType().ToString(),
    dtOrders.Rows[0]["ShipCountry"].GetType().ToString());
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 Feldnamen (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)*.

26.4.5.2 DataTable-Objekte modifizieren

Änderungen an den (lokalen) **DataTable**-Objekten werden später über die **DataAdapter**-Methode **Update()** (siehe Abschnitt 26.4.6) zur Datenbank übertragen.

26.4.5.2.1 Werte in vorhandenen Zeilen ändern

Um einzelne Werte einer **DataTable**-Zeile (eines **DataRow**-Objekts) zu ändern, kann man die zugehörigen Zellen per Indexer-Zugriff ansprechen (vgl. Abschnitt 5.6), z.B.:

```

DataRow dr = dtCustomers.Rows[0];
dr["Country"] = "Lummerland";
dr["CustomerID"] = "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 eine Spalte ihren Wert behalten, verwendet man an der betroffenen Stelle das **Array**-Element **null**.

26.4.5.2.2 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
ColumnChanging	Das Ereignis tritt ein, nachdem eine Wertänderung angefordert worden ist. Eine registrierte Methode erfährt den vorgeschlagenen Wert über das per Parameter übergebene DataColumnChangeEventArgs -Objekt und kann ggf. über die DataRow -Methode SetColumnError() einen später auszuwertenden Fehlerindikator setzen.
ColumnChanged	Das Ereignis tritt ein, sobald der Wert einer Zelle erfolgreich geändert worden ist.
RowChanging	Das Ereignis tritt ein, wenn bei einer Zeile ein Zellenwert oder die RowState -Eigenschaft geändert werden soll. Einer registrierten Methode wird mit dem Ereignisbeschreibungsojekt vom Typ DataRowChangeEventArgs auch eine Referenz zur betroffenen Zeile übergeben, die einen Zugriff auf die Vorschlagswerte erlaubt, so dass ggf. über die DataRow -Methode SetColumnError() ein später auszuwertender Fehlerindikator 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.ToString() +
        "\n ProposedValue: " + e.ProposedValue + ", Wert: " +
        e.Row[e.Column].ToString());
}
void CustomersOnColumnChanged(Object sender, DataColumnChangeEventArgs e) {
    DataTable table = (DataTable)sender;
    Console.WriteLine("ColumnChanged,\tSpalte: " + e.Column.Caption.ToString() +
        "\n ProposedValue: " + e.ProposedValue + ", Wert: " +
        e.Row[e.Column].ToString());
}
void CustomersOnRowChanging(Object sender, DataRowChangeEventArgs e) {
    DataTable table = (DataTable)sender;
    Console.WriteLine("RowChanging (ID: " + e.Row.ItemArray[0].ToString() +
        "), Action: " + e.Action.ToString() + ", RowState: " + e.Row.RowState);
}
void CustomersOnRowChanged(Object sender, DataRowChangeEventArgs e) {
    DataTable table = (DataTable)sender;
    Console.WriteLine("RowChanged (ID: " + e.Row.ItemArray[0].ToString() +
        "), Action: " + e.Action.ToString() + ", RowState: " + e.Row.RowState);
}

```

Sie werden bei den zugehörigen Ereignissen des **DataTable**-Objekts **dtCustomers** registriert:

```

dtCustomers.RowChanging += CustomersOnRowChanging;
dtCustomers.RowChanged += CustomersOnRowChanged;
dtCustomers.ColumnChanged += CustomersOnColumnChanged;
dtCustomers.ColumnChanging += CustomersOnColumnChanging;

```

Aus den Wertänderungen

```

dr["CompanyName"] = "Alfreds Kutterkiste";
dr["Country"] = "Hummerland";

```


resultieren diese Kontrollausgaben:

```
ColumnChanging, Spalte: CompanyName
  ProposedValue: Alfreds Kutterkiste, Wert: Alfreds Futterkiste
ColumnChanged, Spalte: CompanyName
  ProposedValue: Alfreds Kutterkiste, Wert: Alfreds Kutterkiste
RowChanging (ID: ALFKI), Action: Change, RowState: Unchanged
RowChanged (ID: ALFKI), Action: Change, RowState: Modified
ColumnChanging, Spalte: Country
  ProposedValue: Hummerland, Wert: Lummerland
ColumnChanged, Spalte: Country
  ProposedValue: Hummerland, 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**, später liegt der **RowState**-Wert **Modified** vor. Mit den möglichen **RowState**-Werten einer Zeile werden wir uns gleich 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 wird von Microsoft eine andere Vorgehensweise empfohlen (vgl. obige Tabelle), die sich z.B. im anschließend beschriebenen Bearbeitungsmodus realisieren lässt.

26.4.5.2.3 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("ColumnChanging, \tSpalte: " + e.Column.Caption.ToString() +
        "\n ProposedValue: " + e.ProposedValue + ", Wert: " +
        e.Row[e.Column].ToString());
    if (e.Column.ToString().Equals("Country"))
        if ((e.ProposedValue as String).Equals("Kummerland"))
            e.Row.SetColumnError(e.Column, "Error");
}
```

Der Bearbeitungsmodus deaktiviert **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 (vgl. oben).

26.4.5.2.4 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);
```

Bei einem späteren Datenbank-Update (siehe Abschnitt 26.4.6) erhalten die unversorgten Felder der neuen Zeile den Wert **NULL**, sofern dieser erlaubt ist.

26.4.5.2.5 DataRow-Objekt löschen

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");
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** (ein Aufnahmekandidat, siehe unten) 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);
```

Durch ein Datenbank-Update nach dem Entfernen einer **DataTable**-Zeile bleibt eine zugehörige Tabellenzeile der Datenbank unbehellig!

26.4.5.2.6 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 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 zurück versetzt werden.

Je nach Bearbeitungsstand und **RowState**-Wert existieren zudem von einer Zeile verschiedene Versionen, wobei die Existenz einer bestimmten Version über die **DataRow**-Methode **HasVersion()** mit einem Parameter vom Enumerationstyp **DataRowVersion** festgestellt werden kann:

DataRowVersion	Beschreibung												
Current	Diese Version enthält die aktuellen Werte einer Tabellenzeile.												
Original	Diese Version enthält die ursprünglichen Werte, genauer: die aktuellen Werte aus dem letzten Unchanged -Zustand der Zeile.												
Proposed	Diese Version enthält mindestens einen <i>vorgeschlagenen</i> Wert. Sie ist für Tabellenzeilen im Bearbeitungsmodus (zwischen BeginEdit() und EndEdit() bzw. CancelEdit()) sowie bei Zeilen im Zustand Detached verfügbar.												
Default	Die Standardversion hängt vom RowState-Wert ab: <table border="1"> <thead> <tr> <th>DataRowState</th><th>Default-Version ist gleich</th></tr> </thead> <tbody> <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> </tbody> </table>	DataRowState	Default-Version ist gleich	Unchanged	Original	Modified	Current	Added	Current	Detached	Proposed	Deleted	Current
DataRowState	Default-Version ist gleich												
Unchanged	Original												
Modified	Current												
Added	Current												
Detached	Proposed												
Deleted	Current												

Zum Zugriff auf eine bestimmte Zeilenversion dient eine Indexer-Überladung mit Versions-Parameter, z.B.

```
Console.WriteLine("Nach Fill():\t\t"+dr0[1]+" \tRowState: "+dr0.RowState);
dr0[1] = "Alfreds Hummerkiste";
Console.WriteLine("Nach Änderung:\t\t"+dr0[1]+" \tRowState: "+dr0.RowState);
Console.WriteLine("Original-Vers.:\t"+dr0[1, DataRowVersion.Original]);
```

Nach der Änderung unterscheiden sich die Versionen **Current** und **Original**:

Nach Fill():	Alfreds Kutterkiste	RowState: Unchanged
Nach Änderung:	Alfreds Hummerkiste	RowState: Modified
Original-Vers.:	Alfreds Kutterkiste	

Bei der Tabellenmodifikation spielen die für **DataSet**-, **DataTable**- und **DataRow**-Objekte definierten Methoden **AcceptChanges()** und **RejectChanges()** eine wichtige Rolle:

- **AcceptChanges()** wirkt sich folgendermaßen auf **DataRow**-Objekte aus:

RowState vorher	RowState nachher	Wirkung 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 26.4.5.2.3).

- **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.

26.4.5.3 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-Detail - Beziehungen (siehe Abschnitt 26.1.2). 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. 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:

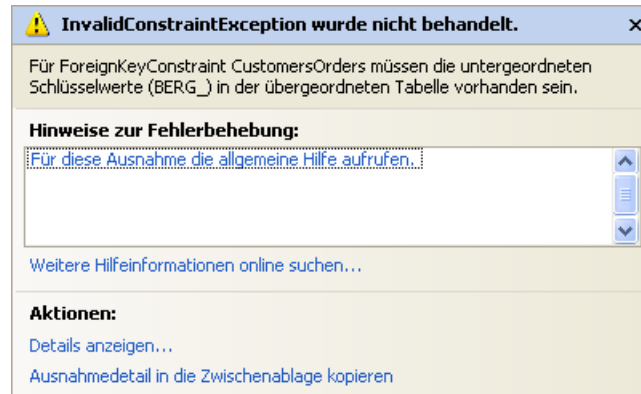
```
ParentKeyConstraint zur Beziehung "CustomersOrders":
Tabelle: Customers
Feld:    CustomerID
Typ:     System.Data.UniqueConstraint
```

```

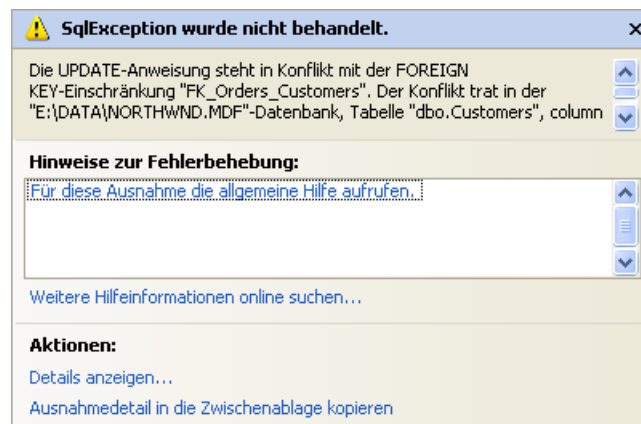
ChildKeyConstraint zur Beziehung "CustomersOrders":
Tabelle: Orders
Feld:      CustomerID
Typ:      System.Data.ForeignKeyConstraint

```

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"]);

```

26.4.6 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 SQL-Kommando (vgl. Abschnitt 26.2) an das DBMS geschickt:

- ein UPDATE-Kommando für geänderte Zeilen
- ein INSERT-Kommando für ergänzte Zeilen
- ein DELETE-Kommando für geänderte Zeilen

Alternative **Update()**-Überladungen berücksichtigen ein komplettes **DataSet**-Objekt, das mehrere Tabellen enthalten kann, oder auch nur ein einzelnes **DataRow**-Objekt.

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, zum Adapter einen passenden **CommandBuilder** zu erzeugen, der die SQL-Kommandos automatisch produziert:

```
new SqlCommandBuilder(dbAdapterCust);
```

Die **CommandBuilder**-Kreationen basieren auf dem **SelectCommand**-Objekt des **DataAdapters**, 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 26.4.4), 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 Datenbank-Tabelle 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);
dbAdapterOrd.Fill(ds);
```

Zum Aktualisieren der Datenbank sind bei der vorgeschlagenen Konstruktion zwei **Update()**-Aufrufe erforderlich:

```
void UpdateData() {
    try {
        dbConnection.Open();
        dbAdapterCus.Update(ds);
        dbAdapterOrd.Update(ds);
    } finally {
        dbConnection.Close();
    }
}
```

Analog zur **Fill()**-Methode stellt auch die **Update()**-Methode bei Bedarf kurzzeitig eine Datenbankverbindung her (vgl. Abschnitt 26.4.2.2). 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 26.4.5.2.6). Weil eine Tabelle unmittelbar nach einem **AcceptChanges()**-Aufruf nur noch Zeilen im Zustand **Unchanged** enthält, ist in dieser Situation ein **Update()**-Aufruf wirkungslos.

26.4.7 Zusammenspiel der ADO.NET - Klassen beim verbindungslosen Arbeiten

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);

        // TableMappings erleichtern die Ansprache von Tabellen
        dbAdapterCus.TableMappings.Add("Table", "Customers");
        dbAdapterOrd.TableMappings.Add("Table", "Orders");

        // Das explizite Öffnen und Schließen der Datenbankverbindung vermeidet
        // die wiederholte Ausführung der Operationen durch Fill() und FillSchema()
        try {
            dbConnection.Open();
            dbAdapterCus.Fill(ds);
            dbAdapterCus.FillSchema(ds, SchemaType.Mapped);
            dbAdapterOrd.Fill(ds);
            dbAdapterOrd.FillSchema(ds, SchemaType.Mapped);
        }
        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", "CompanyName", "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";
    }

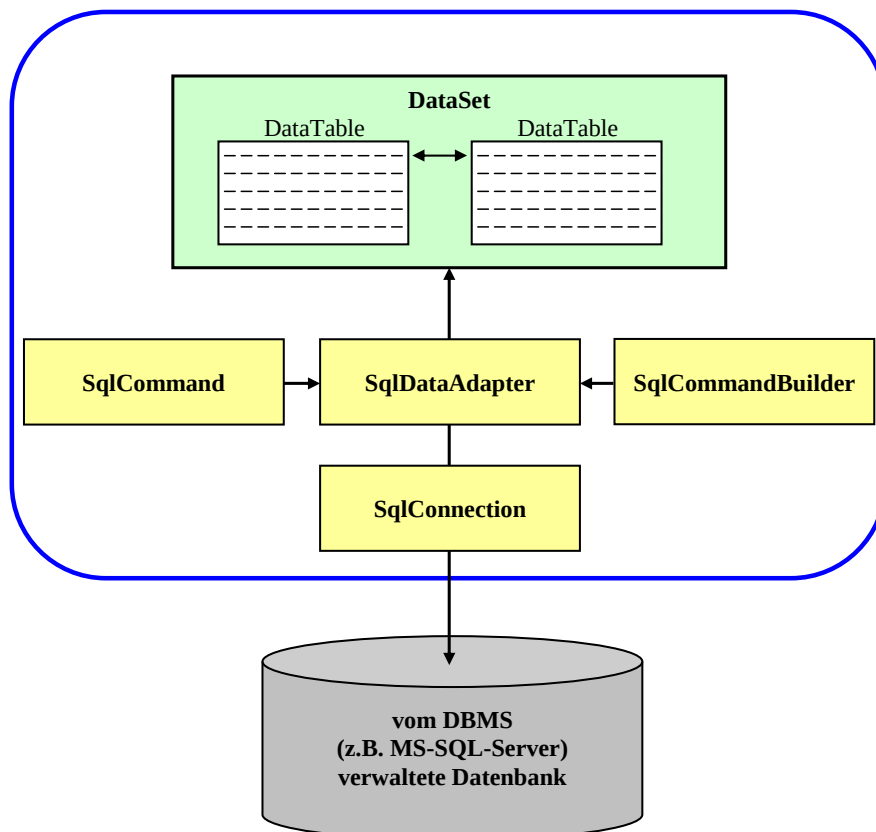
    void UpdateData() {
        try {
            dbConnection.Open();
            dbAdapterCus.Update(ds);
            dbAdapterOrd.Update(ds);
        }
        finally {
            dbConnection.Close();
        }
    }

    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:



26.4.8 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 der Klasse **DataReader** in Frage. Das folgende (mäßig sinnvolle) Beispielprogramm listet aus der **Customers**-Tabelle der **Northwind**-Datenbank alle Firmennamen auf:

```
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);

        // Verbindung zur Datenbank öffnen
        dbConnection.Open();

        SqlDataReader reader = command.ExecuteReader();

        while (reader.Read()) {
            Console.WriteLine(reader["CompanyName"]);
        }

        reader.Close();
        // Schließen der Datenbankverbindung:
        dbConnection.Close();
        Console.ReadLine();
    }
}
```

Zur Klasse **SqlDataReader** existiert kein öffentlicher Konstruktor. Im Beispiel erzeugt die **SqlCommand**-Methode **ExecuteReader()** den **SqlDataReader** und liefert eine Referenz zurück.

Beim ersten Aufruf positioniert die **Read()**-Methode den **SqlDataReader** auf die erste Datenzeile im Abfrageergebnis. Jeder weitere Aufruf inkrementiert den Zeiger oder signalisiert mit dem Rückgabewert **false**, dass keine weiteren Zeilen verfügbar sind.

Solange ein **DataReader** geöffnet ist, verwendet er das zugehörige **Connection**-Objekt exklusiv. Es wird erst durch explizites Schließen des **DataReaders** für eine andere Verwendung frei (z.B. für einen anderen **DataReader**). Dabei wird die Verbindung zur Datenbank *nicht* geschlossen. Schließt man allerdings den vom folgenden **ExecuteReader()**-Aufruf

```
reader = command.ExecuteReader(System.Data.CommandBehavior.CloseConnection);
```

gelieferten **SqlDataReader**, dann wird automatisch auch die Verbindung geschlossen.

26.5 Datenbankunterstützung durch die Entwicklungsumgebungen

26.5.1 Datenquelle

Bei der Datenbankunterstützung in der Visual C# 2008 Express Edition (oder im Visual Studio 2008 Professional) spielt das Konzept der *Datenquelle* eine zentrale Rolle. Fleißige Assistenten schreiben Quellcode ...

- zur Verwaltung von Daten aus verschiedenen Quellen (Datenbanken, Objekte, Webdienste)
- und zur Bindung der Daten an Steuerelemente.

Wir beschränken uns auf Datenbank-Datenquelle, wobei mehrere problemspezifisch definierte Klassen beteiligt sind:

- 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 generische Klasse **DataSet** als Basisklasse dient, z.B.:

```
public partial class NORTHWNDDataset : System.Data.DataSet {
    . . .
}
private NORTHWNDDataset northWNDDataset;

this.northWNDDataset = new WindowsApplication1.NORTHWNDDataset();
```

Im Unterschied zur generischen Klasse **DataSet** sind Schemainformationen (z.B. mit den Datentypen der Spalten) vorhanden, so dass Typfehler bei der Wertzuweisung schon vom Compiler aufgespürt werden können. Außerdem 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 von der Entwicklungsumgebung definiert worden ist:

```
public partial class OrdersDataTable : System.Data.DataTable,
    System.Collections.IEnumerable {
    . . .
}
```

Neben einer solchen **DataTable**-Ableitung entsteht zu jeder Tabelle auch noch eine **DataRow**-Ableitung, z.B.:

```
public partial class OrdersRow : System.Data.DataRow {
    . . .
}
```

Diese Klassen besitzen spezielle Eigenschaften und Methoden für die zugehörige Tabelle und alle erforderlichen Schemainformationen, z.B.:

```
this.columnOrderID.AutoIncrement = true;
this.columnOrderID.AllowDBNull = false;
this.columnOrderID.ReadOnly = true;
this.columnOrderID.Unique = true;
this.columnCustomerID.MaxLength = 5;
```

- 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;
}

```

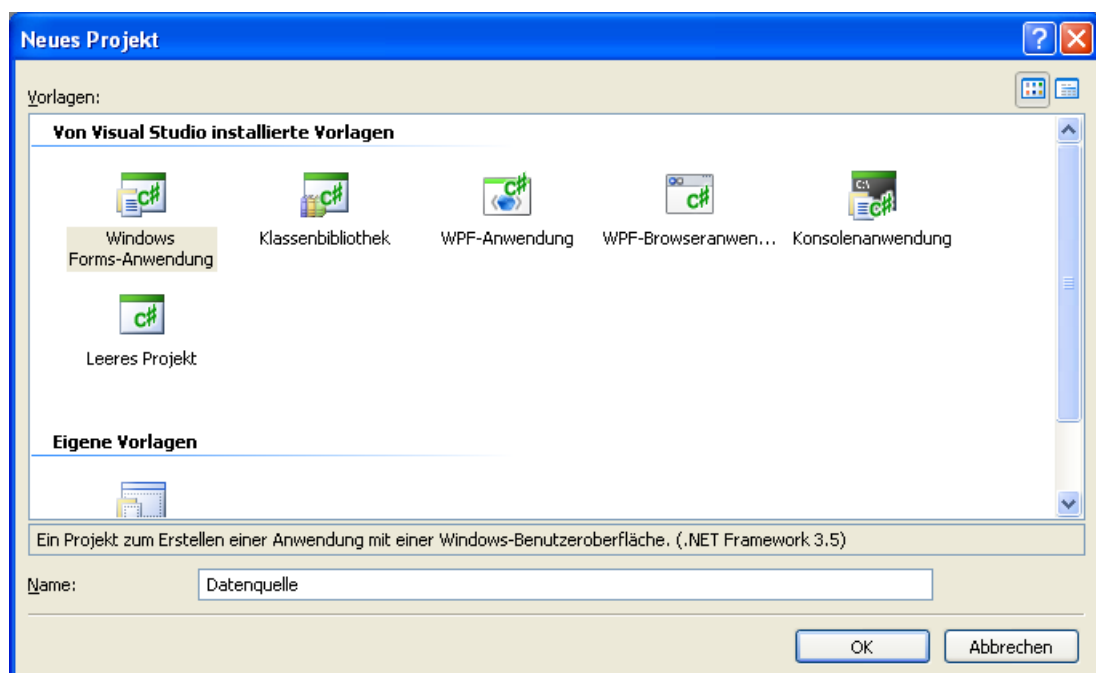
Eine Datenbank-Datenquelle im Sinn unserer Entwicklungsumgebung enthält also mindestens eine typisierte **DataSet**-Klasse und eine typisierte **DataTable**-Ableitung mit zugehöriger **TableAdapter**-Klasse.

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.

26.5.2 Anwendungsbeispiel

Wir erstellen eine WinForms-Anwendung mit Datenbankzugriff:

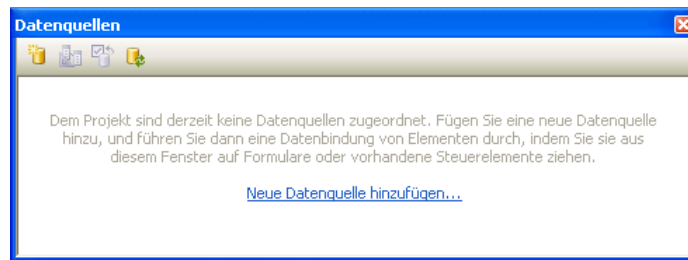
- Legen Sie ein neues Projekt auf Basis der Vorlage **Windows Forms - Anwendung** an, z.B.:



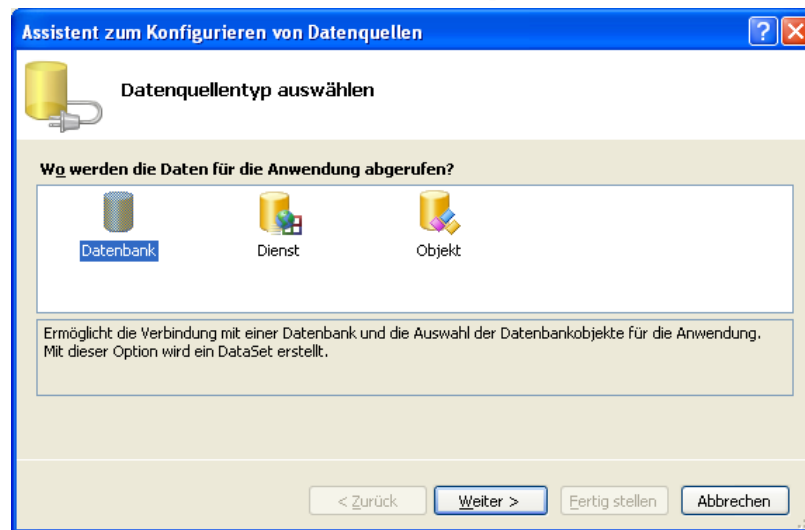
- Öffnen Sie nötigenfalls mit dem Menübefehl

Daten > Datenquellen anzeigen

das **Datenquellen**-Fenster. Es enthält einen Link, der zum Ergänzen einer neuen Datenquelle auffordert:

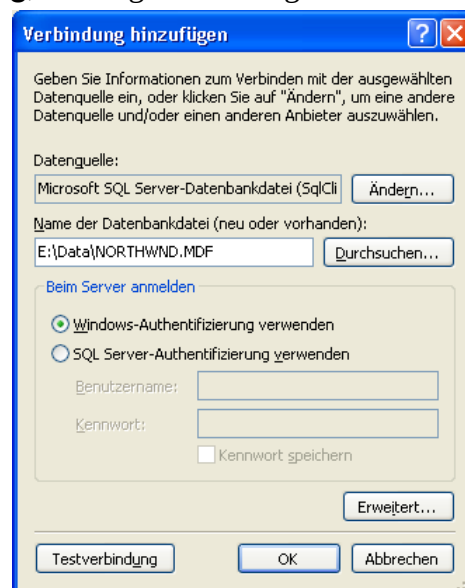


- Nach einem Klick auf diesen Link startet der Datenquellen-Konfigurationsassistent:



Wählen Sie den Datenquellentyp **Datenbank**. Mit den beiden anderen Optionen können wir uns in diesem Kurs aus Zeitgründen nicht beschäftigen.

- Klicken Sie im Dialog zur **Auswahl einer Datenverbindung** (siehe unten) auf den Schalter **Neue Verbindung**, um folgende Dialogbox zu erhalten:

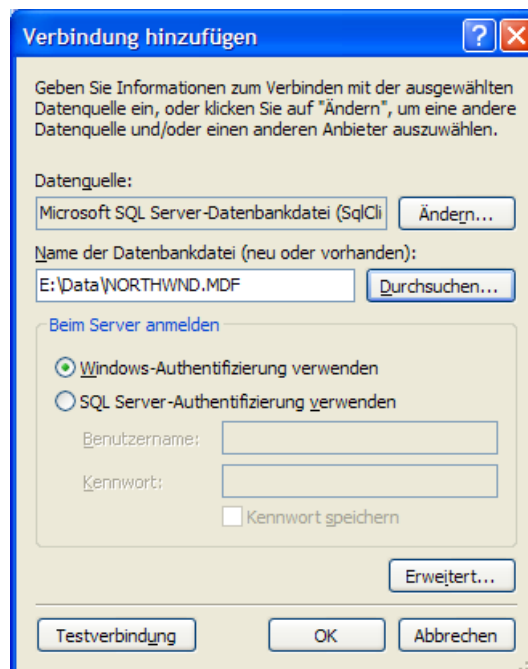


- Nach einem Klick auf den Schalter **Ändern** haben Sie (alle im Kurs vorgeschlagenen Installationen vorausgesetzt) die Wahl zwischen der Express- und der Compact-Edition von Microsofts SQL Server:



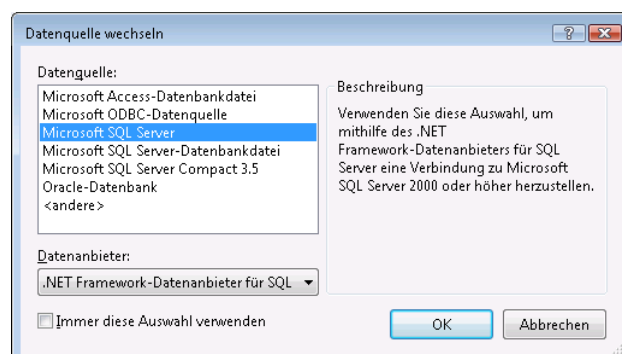
Wenn *beide* Optionen vorhanden sind, spricht einiges für die leistungsfähigere Express-Edition. Am Zusatz *Datenbankdatei* hinter dem Eintrag *Microsoft SQL Server* ist erkennbar, dass eine Benutzerinstanz des SQL-Servers eingerichtet wird (vgl. Abschnitt 26.4.2.1).⁹⁰

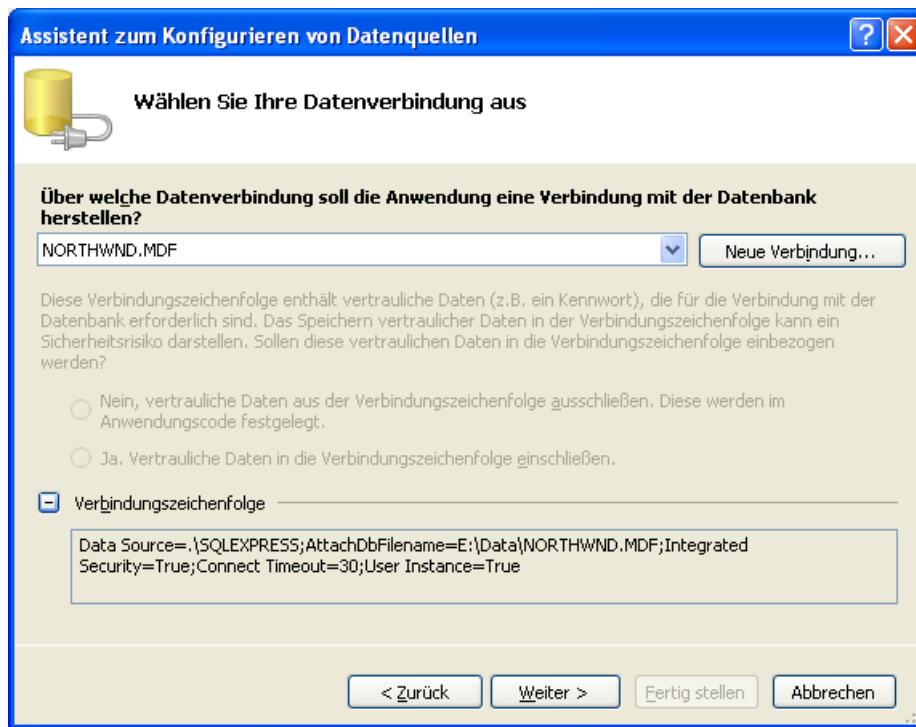
- Wählen Sie im Dialog **Verbindung hinzufügen** die Datenbankdatei **Northwnd.mdf**, und quittieren Sie mit **OK**:



- Nun kann im Dialog zur **Auswahl einer Datenverbindung** die Verbindungszeichenfolge eingesehen werden:

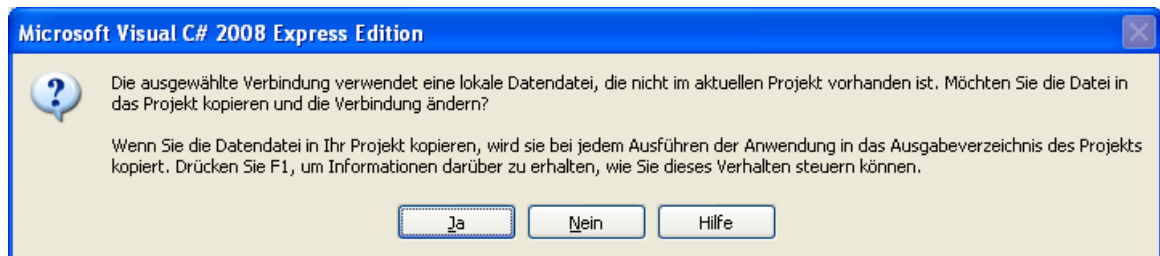
⁹⁰ Wenn Sie statt der Visual C# 2008 Express Edition die Professional-Version verwenden, stehen im Dialog **Datenquelle wechseln** mehr Optionen zur Verfügung, u.a. die Verbindung mit einem **Microsoft SQL Server** auf einem beliebigen Rechner:





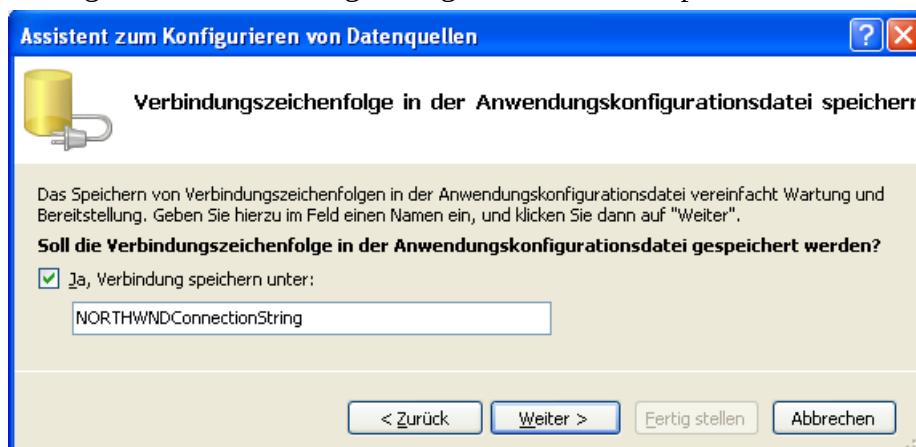
Das Ergebnis entspricht ziemlich genau unseren Erwartungen auf der Basis von Abschnitt 26.4.2.1.

- Wenn Sie **Weiter** machen, erkundigt sich die Entwicklungsumgebung nach Ihren Absichten zur Ablage der Datenbankdatei:

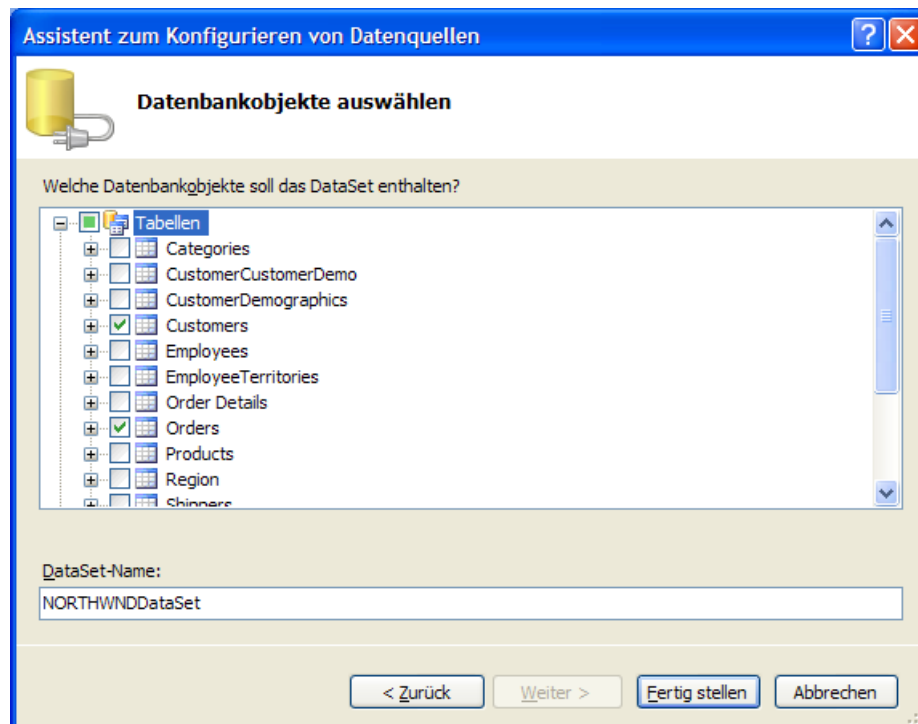


Eine Zustimmung zur projektinternen Kopie vereinfacht die Übertragung des Projekts auf andere Rechner.

- Die nächste Frage bezieht sich auf die in Abschnitt 26.4.2.1 behandelte Option, die Verbindungszeichenfolge in der Anwendungskonfigurationsdatei zu speichern:

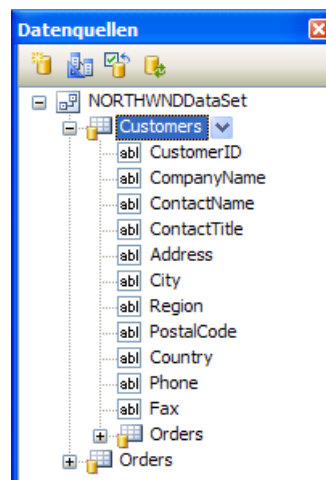


- Schließlich geht es an die Auswahl der benötigten Tabellen aus der **Northwind**-Datenbank:

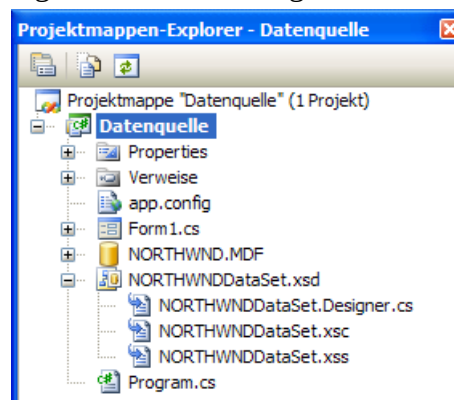


Wählen Sie die Tabellen **Customers** und **Orders**. Von der Möglichkeit, einzelne Tabellenspalten auszuwählen, machen wir keinen Gebrauch.

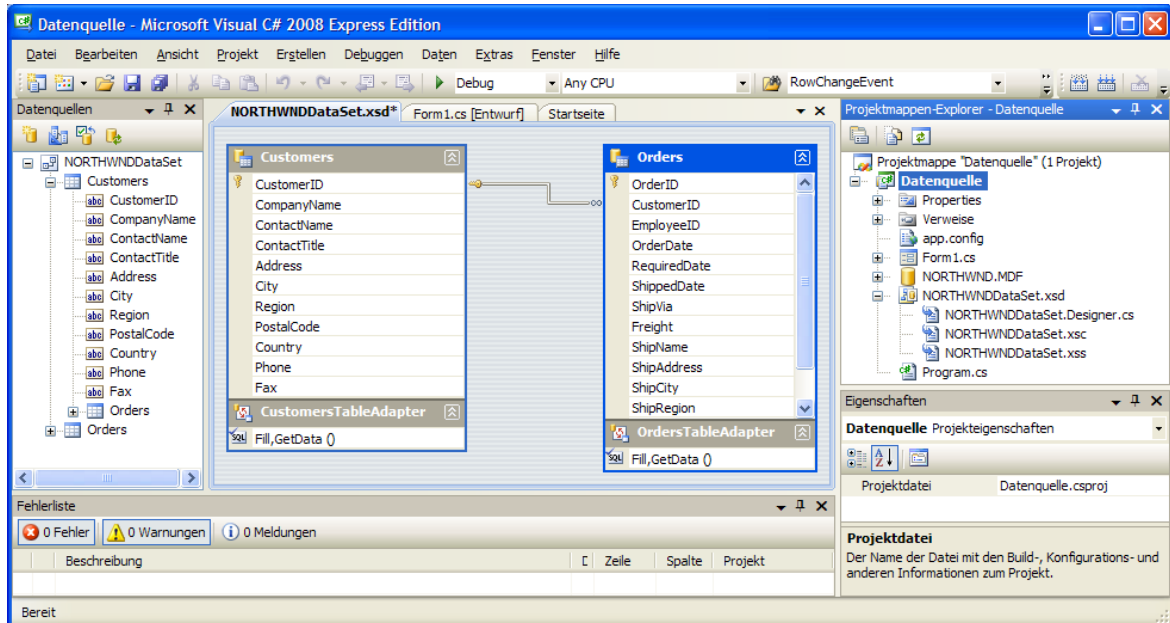
- Beenden Sie den Assistenten mit einem Klick auf den Schalter **Fertig stellen**. Im **Datenquellen**-Fenster der Entwicklungsumgebung sind die beiden ausgewählten Tabellen zu finden:



- Im Projektmappenexplorer zeigt sich nach dem Ergänzen der Datenquelle folgendes Bild:

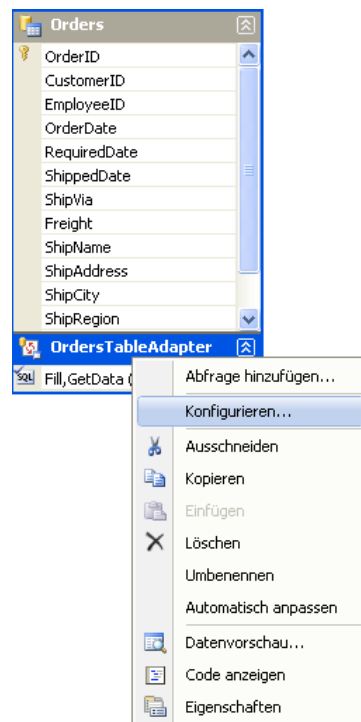


- Die Entwicklungsumgebung hat mittlerweile die in Abschnitt 26.5.1 beschriebenen Klassen für die spezifischen Datenbankbedürfnisse unserer Anwendung angelegt, die nun mit Hilfe des **DataSet-Designers** konfiguriert werden können. Wählen Sie aus dem Kontextmenü zum Datenquellenfenster das Item **DataSet mit Designer bearbeiten**.

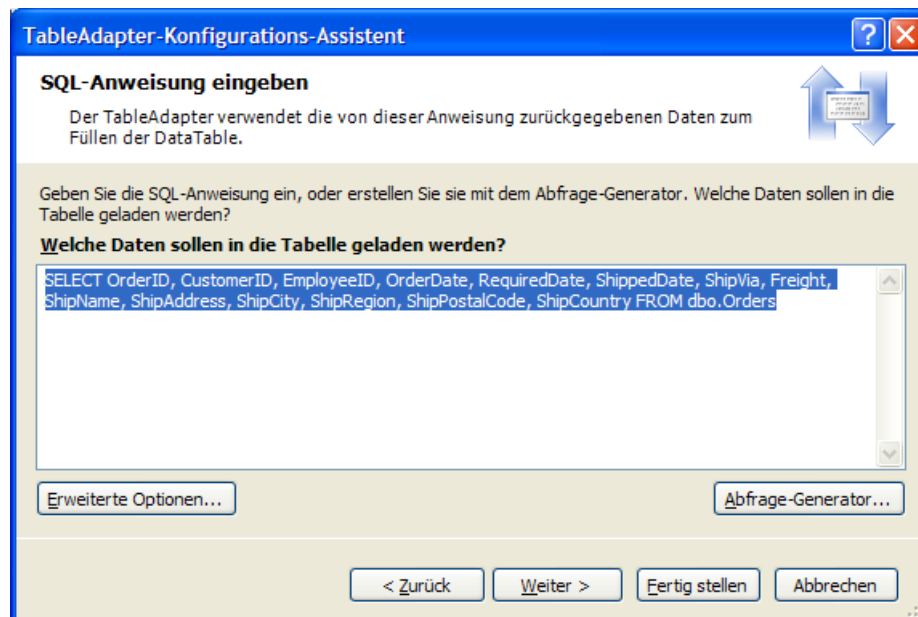


Wir können auf sehr intuitive Weise die Schemadatei **NORTHWNDDataSet.xsd** zum typisierten **DataSet** bearbeiten. Auf den ersten Blick wird klar, dass auch die Master-Details - Beziehung aus der Datenbank übernommen wurde.

- 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.:

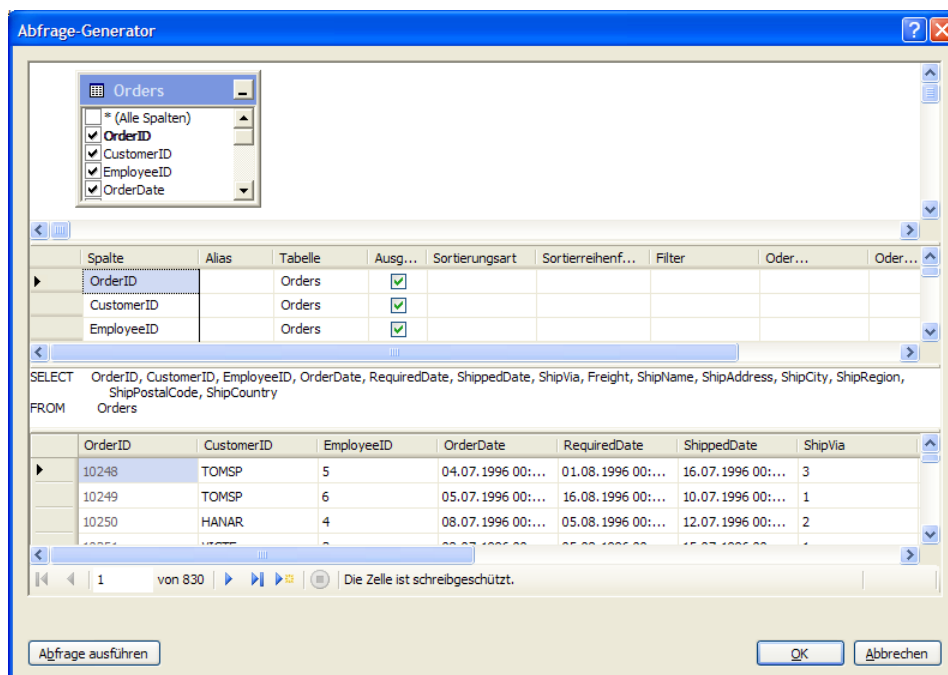


Hier ist das SELECT-Kommando zur Auswahlabfrage zu sehen und zu modifizieren:



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:

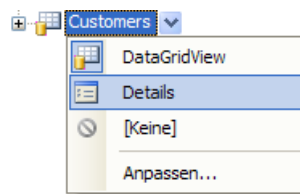


Die am unteren Rand angezeigte Vorschau auf das Abfrageergebnis erhält man über die Kontextmenü-Option **SQL ausführen**.

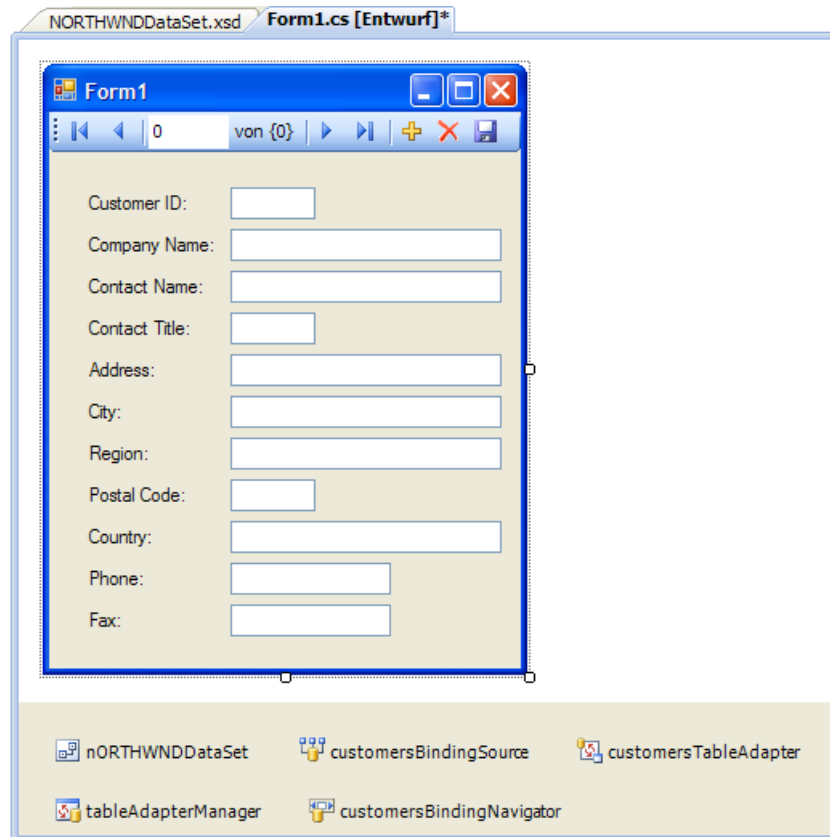
- Wir ignorieren aus Zeitgründen viele Optionen zur Assistenten-gestützten Gestaltung der **TableAdapter**-Klassen, über die z.B. die folgende Microsoft-Webseite informiert:

<http://msdn.microsoft.com/de-de/library/ms379590%28VS.80%29.aspx>

- Wir versorgen das Formular unserer Anwendung nun mit datenzentrierten Bedienelementen und verwenden dabei die besonders bequeme *Drag Once* - Technik. Öffnen Sie den Formulardesigner. 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 Formular, und korrigieren Sie bei Bedarf per Maus die **Width**-Eigenschaftsausprägungen der **TextBox**-Steuerelemente, z.B.:



- Im **Komponentenfach** unterhalb des Formulars (für die Steuerelemente ohne visuellen Auftritt) tauchen fünf Komponenten auf, von denen wir zwei aufgrund von Abschnitt 26.5.1 als Datenquellenbestandteile erwartet haben: die Objekte aus den vom Visual Studio definierten Klassen **NORTHWNDDataSet** und **CustomersTableAdapter**.

Das Objekt **customersBindingSource** aus der mit .NET 2.0 eingeführten Klasse **BindingSource** (im Namensraum **System.Windows.Forms**) sorgt für die Datenbindung der Steuerelemente:

- o Um das **BindingSource**-Objekt mit einer Datenquelle zu verbinden, wird seiner **DataSource**-Eigenschaft das typisierte **DataSet**-Objekt und seiner **DataMember**-Eigenschaft die **Orders**-Tabelle zugewiesen:

```
this.customersBindingSource.DataMember = "Customers";
this.customersBindingSource.DataSource = this.nORTHWNDDataSet;
```
- o Um die Steuerelemente mit dem **BindingSource**-Objekt zu verbinden, wird dieses in die **DataBindings**-Liste der Steuerelemente aufgenommen, z.B.:

```
this.customerIDTextBox.DataBindings.Add(
    new System.Windows.Forms.Binding("Text",
    this.customersBindingSource, "CustomerID", true));
```

Die Referenzvariable **customersBindingNavigator** zeigt auf ein Steuerelement aus der mit .NET 2.0 eingeführten Klasse **BindingNavigator** (Namensraum **System.Windows.Forms**). Diese Klasse realisiert eine Standardsymbolleiste zur Tabellennavigation und

-bearbeitung. Für die Datenbindung sorgt das eben beschriebene Objekt `customers - BindingSource`:

```
this.customersBindingNavigator.BindingSource =
    this.customersBindingSource;
```

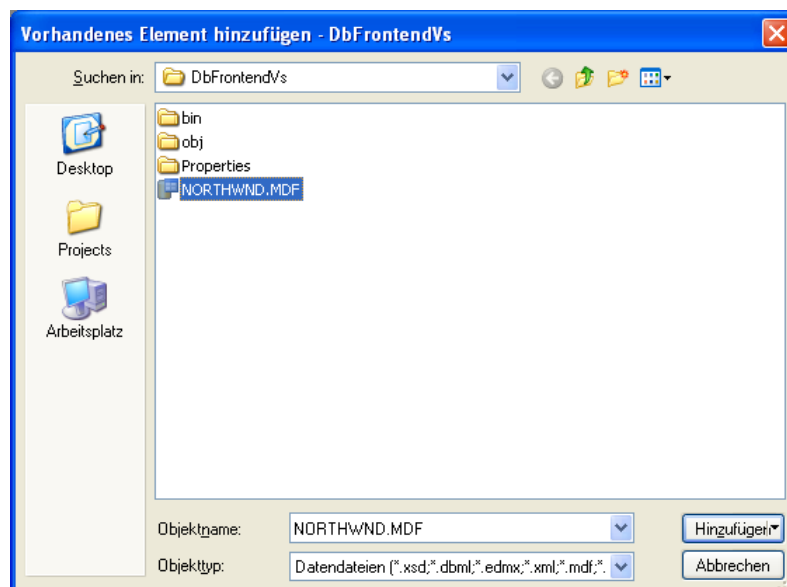
Schließlich findet sich im Komponentenfach noch ein Objekt der vom Visual Studio definierten Klasse `TableAdapterManager`. Es sorgt für referentielle Integrität beim Speichern in Tabellen, die über Beziehungen verknüpft sind.

Im aktuellen Entwicklungszustand taugt das Programm zur Verwaltung der **Customers**-Tabelle:

Es fehlt noch die Integration der **Orders** – Tabelle.

26.5.3 Datenbanken mit der Entwicklungsumgebung bearbeiten

Das Visual Studio 2008 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 Windows-Projekts. Nehmen Sie die Datenbank per Projektmappen-Explorer über das Kontextmenü des Projekts mit **Hinzufügen > Vorhandenes Element** in das Projekt auf:

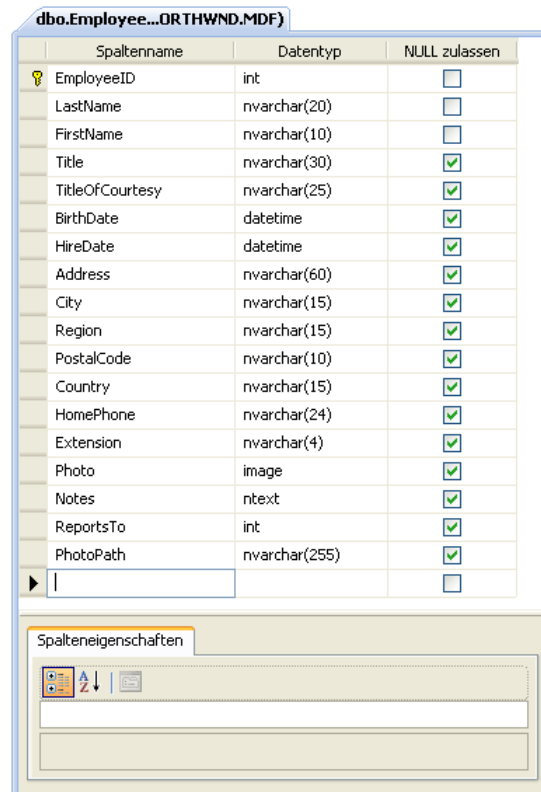


Brechen Sie den Datenquellen-Konfigurationsassistenten ab. Nach einem Doppelklick auf die Datenbankdatei im Projektmappen-Explorer erscheint der Datenbank-Explorer am linken Rand des Visual Studio - Fensters:



26.5.3.1 Tabellen anzeigen und bearbeiten

Erweitern Sie die Zweig **Tabellen** und öffnen Sie per Doppelklick die Tabelle **Employees** zur Bearbeitung in der Designer-Zone der Entwicklungsumgebung:



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.:

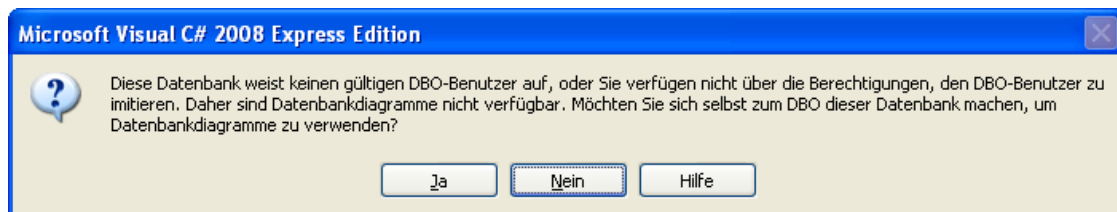
Employees: A...ORTHWND.MDF

	EmployeeID	LastName	FirstName	Title	TitleOfCourtesy	BirthDate	HireDate	Address	City	Region
▶	1	Davolio	Nancy	Sales Representative	Ms.	08.12.1948 ...	01.05.1992 ...	507 - 20th Ave. E...	Seattle	WA
	2	Fuller	Andrew	Vice President, Sales	Dr.	19.02.1952 ...	14.08.1992 ...	908 W. Capital Way	Tacoma	WA
	3	Leverling	Janet	Sales Representative	Ms.	30.08.1963 ...	01.04.1992 ...	722 Moss Bay Blvd.	Kirkland	WA
	4	Peacock	Margaret	Sales Representative	Mrs.	19.09.1937 ...	03.05.1993 ...	4110 Old Redmon...	Redmond	WA
	5	Buchanan	Steven	Sales Manager	Mr.	04.03.1955 ...	17.10.1993 ...	14 Garrett Hill	London	NULL
	6	Suyama	Michael	Sales Representative	Mr.	02.07.1963 ...	17.10.1993 ...	Coventry House...	London	NULL
	7	King	Robert	Sales Representative	Mr.	29.05.1960 ...	02.01.1994 ...	Edgeham Hollow...	London	NULL
	8	Callahan	Laura	Inside Sales Coordin...	Ms.	09.01.1958 ...	05.03.1994 ...	4726 - 11th Ave. ...	Seattle	WA
	9	Dodsworth	Anne	Sales Representative	Ms.	27.01.1966 ...	15.11.1994 ...	7 Houndstooth Rd.	London	NULL
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

1 von 9 Die Zelle ist schreibgeschützt.

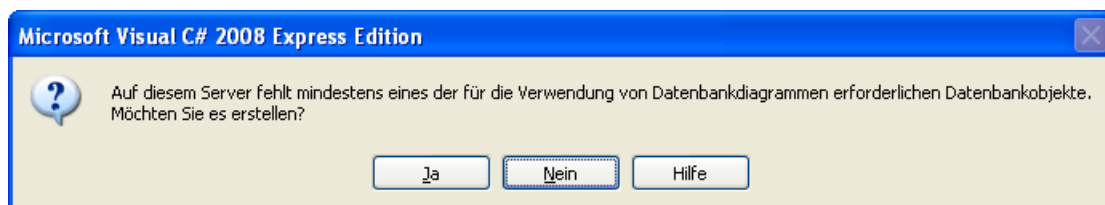
26.5.3.2 Datenbankdiagramm erstellen

Wenn Sie im Datenbank-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 (*database owner*) der Datenbank.

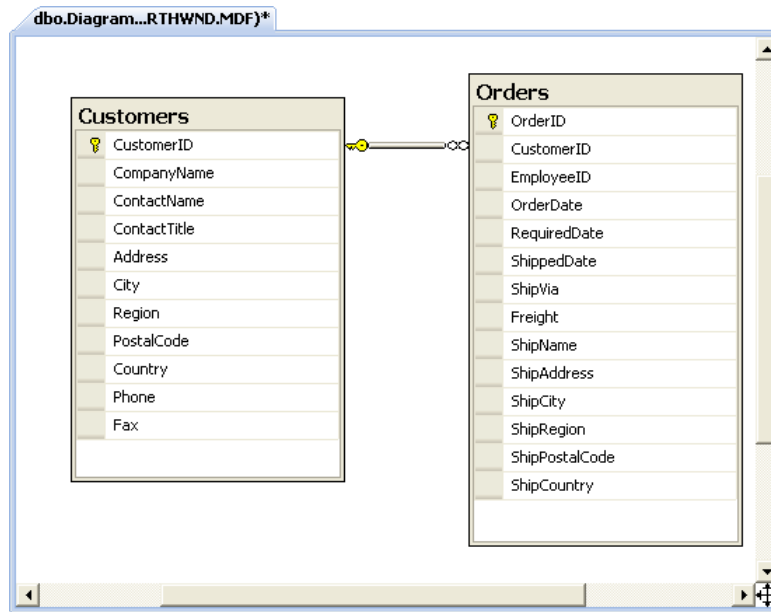
Veranlassen Sie ggf. anschließend das Erstellen fehlender Datenbankobjekte:



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:

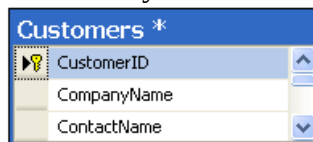


Beim Speichern des Diagramms (per **Strg-S** oder Symbolschalter angefordert) legt man seinen Namen fest.

26.5.3.3 Beziehungen definieren

Über ein Datenbankdiagramm lassen sich Beziehungen definieren:

- Primärschlüssel per Klick auf das Schlüsselssymbol 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**

Fremdschlüsselbeziehung

Beziehung (ausgewählt):
FK_Orders_Customers*

Eigenschaften für Beziehung (neu) werden bearbeitet. Die Tabellen- und Spaltenspezifikation-Eigenschaft muss ausgefüllt werden, bevor Beziehung (neu) akzeptiert wird.

(Allgemein)

☒ Tabellen- und Spaltenspezifität

Vorhandene Daten bei Erstellen: Ja

Datenbank-Designer

☒ Fremdschlüsseinschränkung: Ja

☒ Für Replikation erzwingen: Ja

☒ INSERT- und UPDATE-Spezif

Identität

(Name): FK_Orders_Customers

Beschreibung:

OK Abbrechen

jeweils mit **OK** quittieren.

27 LINQ

Die Integration einer einheitlich für diverse Datenquellen geeignete, typsicheren, an SQL angelehnten Abfragetechnik ist wohl der wichtigste Zugewinn bei C# 3.0 bzw. .NET 3.5. Leider ist die LINQ-Behandlung (*Language Integrated Query*) in diesem Manuskript noch unausgereift.

27.1 Erweiterungen der Programmiersprache C#

27.1.1 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	Ausgabe
<pre>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()); } }</pre>	<pre>Wert: 13 Typ: System.Int32 Wert: 3,14 Typ: System.Double</pre>

Die strenge Typisierung von C# wird keinesfalls gelockert, so dass sich z.B. auch ein implizit festgelegter Typ nicht mehr ändern lässt.

27.1.2 Instanz- und Listeninitialisierer

Speziell für die neuen anonymen Typen (siehe Abschnitt 27.1.3) ist die Möglichkeit von Interesse, bei der Objekterzeugung öffentliche Felder und öffentliche Eigenschaft mit Werten zu versorgen, ohne einen speziellen Konstruktor zu benötigen, z.B.:

Quellcode	Ausgabe
<pre>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); } }</pre>	<pre>13 4711</pre>

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 Objekte bei der Kreation mit Werten versorgen, arbeiten die mit C# 3.0 eingeführten Initialisierer für (generische) Listen, z.B.:

```
List<int> il = new List<int>() {1, 2, 3};
```

27.1.3 Anonyme Klassen

Aus einem Objektinitialisierer ohne Typbezeichnung resultiert eine anonyme Klasse. Genau genommen hat die Klasse durchaus einen (vom Compiler vergebenen) Namen, doch darf dieser Name im Quellcode nicht verwendet werden. Zur Deklaration der zugehörigen Referenzvariablen verwendet man das Schlüsselwort **var** (siehe Abschnitt 27.1.1), weil der Klassenname unbekannt ist, und die (syntaktisch erlaubte) Deklaration als **Object** große praktische Nachteile beim Zugriff auf die Member hätte. Als Innenausstattung enthält die anonyme Klasse eine öffentliche **get-only** - Eigenschaft für jedes Listenelement des Initialisierers, so dass im folgenden Beispiel

Quellcode-Segment	Ausgabe
<pre>var a=new{Name="Knut",Alter=53}; Console.WriteLine(a.GetType());</pre>	<pre><>f__AnonymousType0`2[System.String,System.Int32]</pre>

eine Klasse mit den Eigenschaften **Name** und **Alter** resultiert, wobei der Compiler die Datentypen (**String** und **int**) 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 anonymen Objekt-Initialisierer auch ein Eigenschafts-, Variablen- oder Parameternamen der umgebenden Methode stehen, der als Eigenschaftsname für die anonyme Klasse übernommen wird, z.B.:

Quellcode-Segment	Ausgabe
<pre>int Alter = 53; var b = new {Name = "Knut", Alter}; Console.WriteLine(b.Alter.GetType());</pre>	<pre>System.Int32</pre>

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;
```

27.1.4 Lambda-Ausdrücke

Mit den Lambda-Ausdrücken bietet C# 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);
```

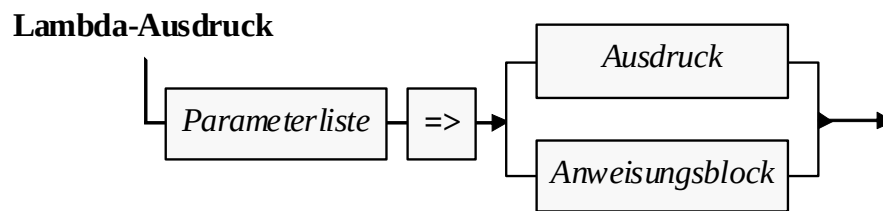
die äquivalenten Realisationen über eine anonyme Methode

```
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 Parametertypen verzichten, weil sich diese aus dem zu erfüllenden Delegationstyp 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 Rückgabotyp 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;  
};
```

27.1.5 Erweiterungsmethoden

Soll eine Klasse um zusätzliche Handlungskompetenzen erweitert werden, erstellt man üblicherweise eine abgeleitete Klasse. In C# 3.0 steht eine alternative Erweiterungstechnik zur Verfügung für Situationen, in denen das Ableiten eines neuen Typs umständlich, oder unmöglich wäre (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:

Quellcode-Segment	Ausgabe
<pre>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)); } }</pre>	9,8596

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.

Tritt eine Erweiterungsmethode in Konkurrenz mit einer typeigenen, dann gewinnt letztere.

27.2 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 27.1 eingeführten C# - Sprachergänzungen sowie neue C# - Schlüsselwörter zum Einsatz kommen.

Eine zentrale Rolle im LINQ-Framework spielen die 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-Abfragen 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 27.2 vertraut gemacht hat, wird mit Konstruktionen wie im folgenden Beispiel keine großen Probleme haben:

```
var cust = customers.
    Where(c => c.Country == "Mexico").
    Select(c => new { c.CustomerID, c.Country });
```

- **Abfrage-Syntax**

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.

27.2.1 Die from-Klausel zur Definition einer Datenquelle

Ein LINQ-Abfrageausdruck beginnt mit einer **from**-Klausel mit Angabe der Datenquelle und unterscheidet sich damit auf den ersten Blick von einer SELECT-Abfrage in SQL (vgl. Abschnitt 26.2). Indem zunächst die Datenquelle ausgewählt wird, von der die nachfolgenden Operationen ausgehen, kann die Entwicklungsumgebung mit ihrer IntelliSense-Technik das Erstellen von Abfrage-Syntax 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 Datenquelle muss die Schnittstelle **IEnumerable<T>** implementieren, und die hinter dem Schlüsselwort **from** genannte Variable hat implizit den Typ **T**.

Implementiert die Datenquelle nur die nicht-generische Schnittstelle **IEnumerable**, muss in der **from**-Klausel 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**-Entsprechung benötigt, z.B.:

```
var custIds = customers.Select(c => c.CustomerID);
```

27.2.2 Die 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 Ausgabetyt spricht. Neben dem **this**-Parameter vom zu erweiternden Typ erwartet **Select<TSource, TResult>()** einen Parameter vom Delegationstyp **Func<TSource, TResult>**. Ein Objekt dieses Typs zeigt auf eine Methode mit einem Parameter vom Typ **TSource**, die einen Wert 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 Delegationenobjekt wird über einen Lambda-Ausdruck geliefert.

So sieht die äquivalente Abfrage-Syntax 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()**- und dem dadurch erforderlichen **GetEnumerator()**-Aufruf geändert, so dass in der verzögert ausgeführten Abfrage der aktuelle Zustand enthalten ist:

Quellcode-Segment	Ausgabe
<pre>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));</pre>	<pre>BLONP SEVEN SIEBEN</pre>

In den obigen Beispielen liefert der Projektionsoperator **select** ein einzelnes Feld des Quellelementtyps, was zu einer besonders einfachen Syntax führt. Wird ein zusammengesetzter Abfrageelement-

typ benötigt, verwendet man eine anonyme Klasse (vgl. Abschnitt 27.1.3). Dieser Erweiterungsmethodenaufruf

```
var cust = customers.Select(c => new {c.CustomerID, c.Country});
```

liefert wie die äquivalente Abfrage-Syntax

```
var cust = from c in customers
           select new {c.CustomerID, c.Country};
```

Abfrageelemente mit den beiden Feldern CustomerID und Country.

27.2.3 Formulierung eines Auswahlkriteriums

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 Delegationstyp **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 Abfrage-Syntax

```
var cust = from c in customers
           where c.Country == "Mexico"
           select new {c.CustomerID, c.Country};
```

auf Kunden in Mexiko.

27.2.4 Weitere LINQ-Operationen

Über die zahlreichen weiteren LINQ-Erweiterungsmethoden bzw. -Operatoren informiert z.B. die FCL-Dokumentation zur Schnittstelle **IEnumerable<T>**.

27.3 LINQ to SQL

LINQ to SQL (früher als *DLINQ* bezeichnet) bietet neben der in Abschnitt 27.2 beschriebenen Abfragetechnik eine objekt-relationale Brücke zwischen Microsofts SQL-Servern und den Objekten einer .NET - Anwendung. Nachdem sich Microsoft bei Techniken zur Überbrückung der Kluft zwischen der dominierenden objektorientierten Softwaretechnik und dem relationalen Datenbankstandard lange zurückgehalten hat, stehen nun mit *LINQ to SQL* und dem *ADO.NET Entity Framework* gleich zwei alternative bzw. konkurrierende Lösungen zur Verfügung (siehe z.B. Schwichtenberg 2009c).

Im Zusammenhang mit ORM-Lösungen (*Object Relational Mapping*) ist oft von Entitätsklassen und Entitäten die Rede, so dass diese Begriffe kurz erläutert werden sollen. Eine Entitätsklasse steht im Objektmodell für eine Tabelle der Datenbank. Die Eigenschaften einer Entitätsklasse entsprechen den Spalten der zugehörigen Datenbanktabelle. Ein Objekt der Entitätsklasse (auch als *Entität* bezeichnet) repräsentiert eine Zeile der Datenbanktabelle.

Es stellt sich in Frage, ob man nicht auch die in Abschnitt 26.5.1 vorgestellten typisierten Klassen als ORM-Lösung betrachten kann. In der folgenden Tabelle werden (ohne jeglichen Anspruch auf Vollständigkeit) einige Eigenschaften der beiden Lösungen verglichen:

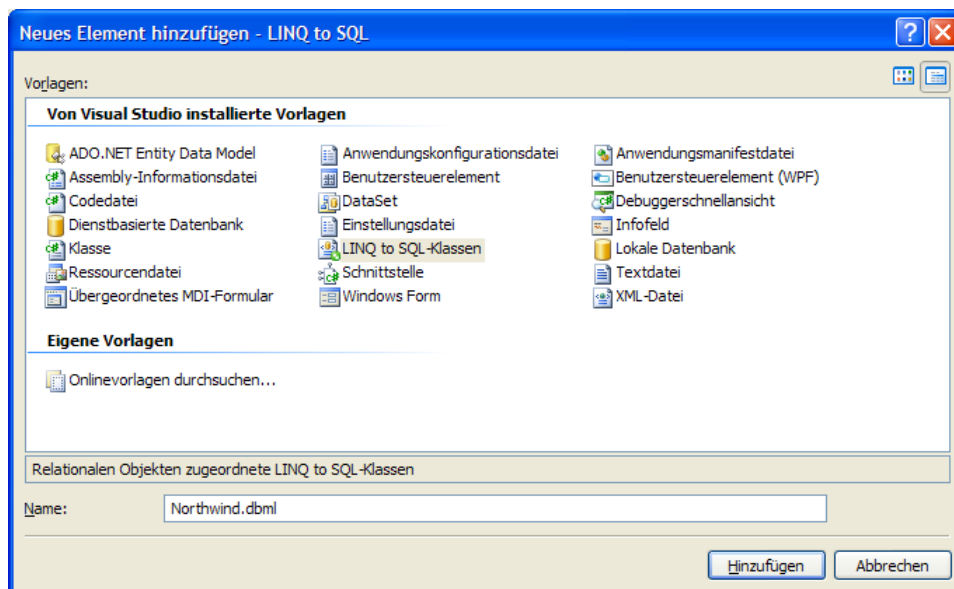
	LINQ to SQL	Typisierte Klassen
Erstellung eines Objektmodells zu einem Datenmodell	ja	ja
Erstellung eines Datenmodells zu einem Objektmodell	nein	ja
Basisklasse einer Entitätsklasse	beliebig	System.Data.DataTable
Nutzbare Datenbanken	MS SQL	alle mit ADO.NET - Provider

27.3.1 Elementare LINQ-to-SQL - Klassen und O/R - Designer

Um zu der vom SQL-Server Express verwalteten Northwind-Datenbank die LINQ-to-SQL - Klassen vom Visual C# 2008 Express erzeugen zu lassen, wählen wir vom Projekt-Kontextmenü ausgehend:

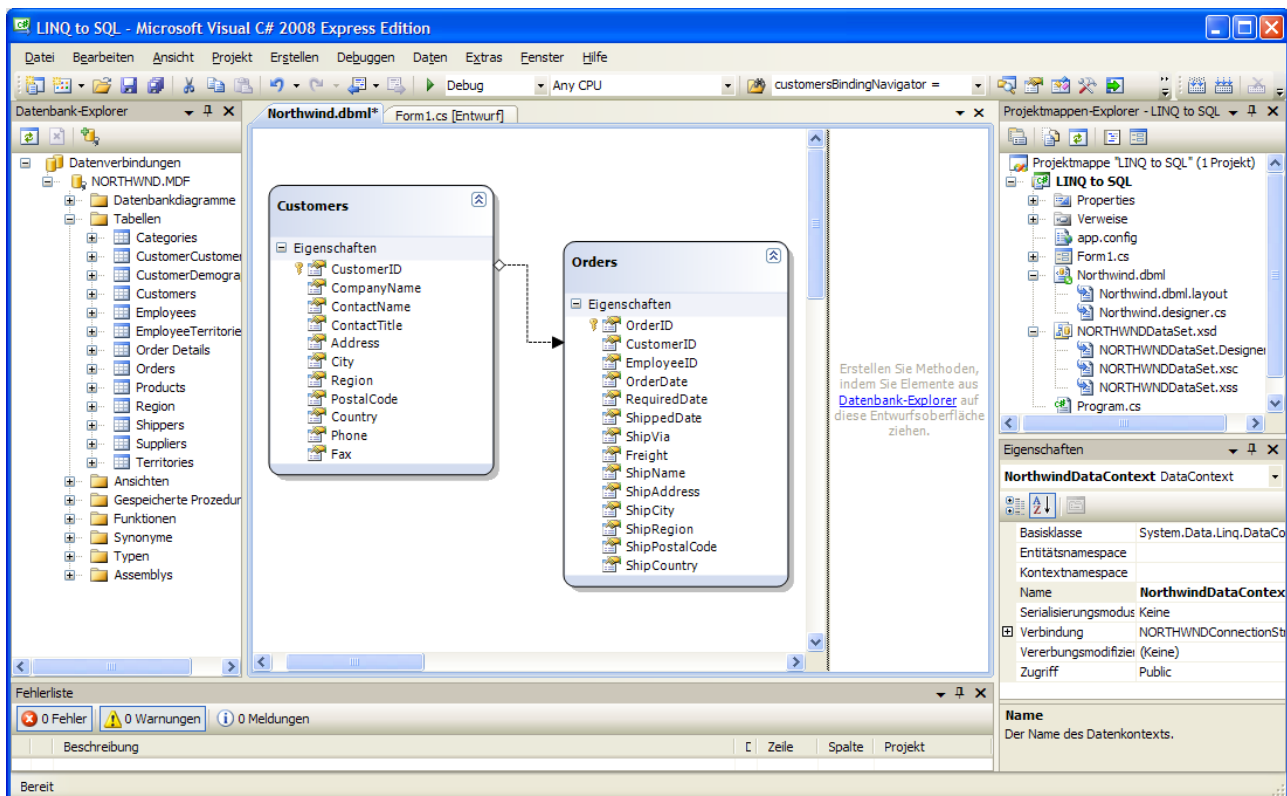
Projekt > Hinzufügen > Neues Element > LINQ to SQL - Klassen

Wie der folgende Dialog zeigt,

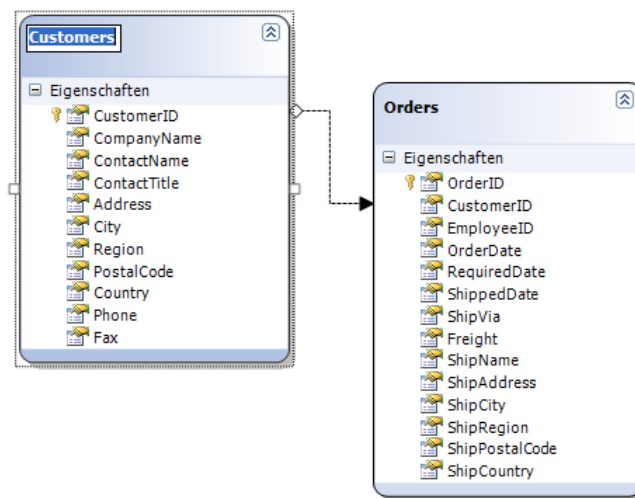


entsteht dabei (auch) eine Konfigurationsdatei im XML-Format (im Beispiel: **Northwnd.dbml**). Sie beschreibt die Datenbankstruktur, muss aber nicht direkt bearbeitet werden.

Mit Hilfe des **Datenbank-Explorer**-Fensters platzieren wir die benötigten Tabellen auf die (notwendigenfalls per Doppelklick auf die DBML-Datei zu öffnenden) Arbeitsfläche des **O/R - Designers**, z.B.:



Der Designer pluralisiert die Namen der Entitätsklassen im Vergleich zu den Tabellennamen (z.B. Consumer → Consumers), was der persönlichen Auffassung von objektorientierter Programmierer nicht unbedingt entsprechen muss. Man kann die Namen im Designer anpassen (auf der Arbeitsfläche oder im Eigenschaftsfenster):



Beim Singularisieren verschwindet aber auch der sinnvolle Plural aus den Namen der **DataContext**-Eigenschaften (siehe unten), die auf Tabellen mit Entitäten aus einer Klasse zeigen, so dass wir auf eine Änderung verzichten. Dass generell die Namen der Entitätsklassen von den Tabellennamen abweichen dürfen, schafft Flexibilität und reduziert z.B. den Aufwand bei einer Umbenennung der Datenbanktabellen.

Der O/R - DESIGNER übernimmt automatisch die **Beziehungen** zwischen den Tabellen, so dass im Beispiel die **Customers**-Objekte über die Eigenschaft **Orders** vom Auflistungstyp **EntitySet<Orders>** verfügen, welche auf ein Listenobjekt mit den Bestellungen eines Kunden zeigt:


```
[Association(Name="Customers_Orders", Storage="_Orders",
    ThisKey="CustomerID", OtherKey="CustomerID")]
public EntitySet<Orders> Orders {
    get {
        return this._Orders;
    }
    set {
        this._Orders.Assign(value);
    }
}
```

In der vom O/R - Designer gepflegten Quellcodedatei **Northwind.designer.cs** (siehe Eintrag im Projektmappen-Explorer) werden die folgenden drei Klassen definiert:

```
namespace LinqToSqlExample {
    using System.Data.Linq;

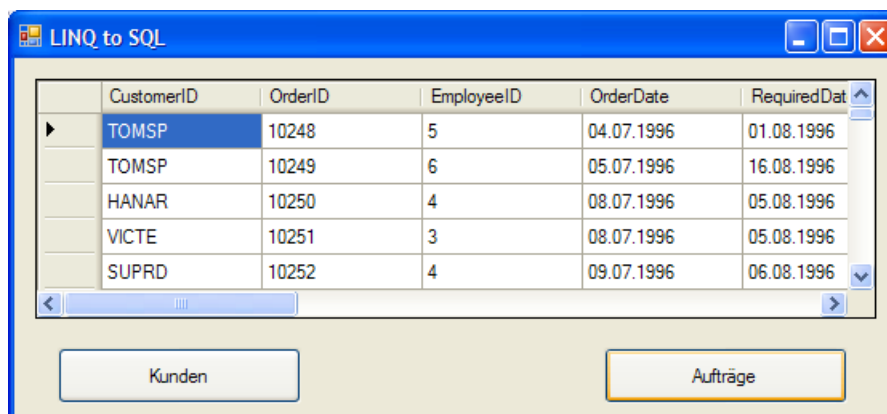
    using System;

    [System.Data.Linq.Mapping.DatabaseAttribute(Name="NORTHWND")]
    public partial class NorthwindDataContext : System.Data.Linq.DataContext
    { . . . }
    [Table(Name="dbo.Customers")]
    public partial class Customers : INotifyPropertyChanging, INotifyPropertyChanged
    { . . . }
    [Table(Name="dbo.Orders")]
    public partial class Orders : INotifyPropertyChanging, INotifyPropertyChanged
    { . . . }
}
```

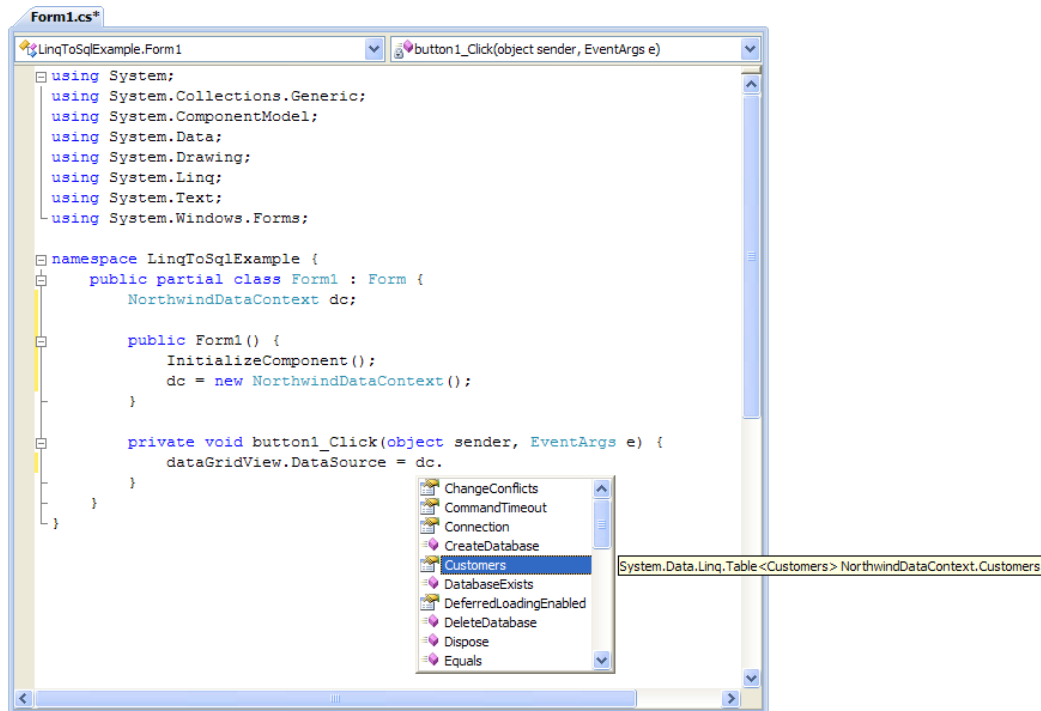
Neben den erwarteten Entitätsklassen **Customers** und **Orders** begegnet uns die von **System.Data.Linq.DataContext** abgeleitete Klasse **NorthwindDataContext**. Im LINQ to SQL - Framework spielt die Klasse **DataContext** eine vermittelnde Rolle zwischen den Entitätsklassen und der Datenbank. Ein **DataContext**-Objekt wird analog zu einem ADO.NET - **SqlConnection**-Objekt verwendet (vgl. Abschnitt 26.4.2).

Im Vergleich zum ADO.NET - Framework mit den essentiellen Klassen **DataSet**, **DataTable**, **DataRow**, **DataColumn**, **Constraint**, **DataRelation**, **SqlConnection**, **SqlDataAdapter**, **SqlCommand**, **SqlCommandBuilder** etc. wirkt das LINQ to SQL - Framework, das vom Microsoft als *Teil der ADO.NET-Technologiefamilie* bezeichnet wird, deutlich übersichtlicher. Wie zu erwarten, besitzt das LINQ to SQL - Framework allerdings nicht den kompletten ADO.NET - Funktionsumfang. Damit die Entscheidungslage für Entwickler nicht allzu simple wird, bietet Microsoft mit dem **ADO.NET Entity - Framework**, das in diesem Manuskript nicht behandelt wird (siehe z.B. Doberenz & Gewinnus, 2008, Kap 14.), eine weitere ORM-Lösung.

Wir erweitern den vom O/R - Designer verfassten Quellcode zu einem einfachen Beispielprogramm mit WinForms-Benutzerschnittstelle bestehend aus einem **DataGridView**-Steuerelement und zwei Befehlsschaltern:



Im Konstruktor der Formalklasse wird das über eine private Instanzvariable ansprechbare **DataContext**-Objekt erzeugt. In den **Click**-Ereignisbehandlungsmethoden zu den Befehlsschaltern wird der **DataSource**-Eigenschaft des **DataGridView**-Steuerelements die jeweils passende Tabelle über eine öffentliche Eigenschaft des **DataContext**-Objekts zugewiesen, wobei die Entwicklungsumgebung mit ihrer IntelliSense-Funktion kräftig unterstützt, z.B.:



27.3.2 Datenbankbearbeitung via LINQ to SQL

Am Ende der rudimentären LINQ to SQL - Behandlung sollen noch elementare Datenbankmodifikationen demonstriert werden. Dazu wird ein Konsolenprojekt verwendet, das die LINQ to SQL - Klassen zur Northwind-Datenbank per O/R - Designer erhalten hat (siehe Abschnitt 27.3.1).

27.3.2.1 Werte ändern

Die folgende Methode ändert den Firmennamen der Customers-Tabellenzeile mit dem Wert ALFKI beim Primärschlüsselfeld CustomerID:

```
static void ChangeData() {
    var cust = dc.Customers.Where(c => c.CustomerID == "ALFKI");
    if (cust.Count() > 0) {
        var alfki = cust.First();
        if (alfki.CompanyName == "Alfreds Kutterkiste")
            alfki.CompanyName = "Alfreds Futterkiste";
        else
            alfki.CompanyName = "Alfreds Kutterkiste";
        dc.SubmitChanges();
    }
}
```

Für die Datenbankaktualisierung sorgt das per **SubmitChanges()** beauftragte **NorthwindDataContext**-Objekt. Es kommen die **Enumerable**-Erweiterungsmethoden **Count()** (liefert die Anzahl der Elemente in einer Auflistung) und **First()** (liefert eine Referenz auf das erste Element einer Auflistung) zum Einsatz.

27.3.2.2 Neue Entitäten (Tabellenzeilen) anlegen

Hier wird ein neues Objekt der Entitätsklasse `Customers` erzeugt:

```
static void CreateEntity() {
    var cust = dc.Customers.Where(c => c.CustomerID == "NEWEN");
    if (cust.Count() == 0) {
        Console.WriteLine("Creating new Entity NEWEN\n");
        Customers newCust = new Customers {
            CustomerID = "NEWEN",
            CompanyName = "LINQ to SQL",
            ContactName = "Default-Contact",
            ContactTitle = "Default-Title",
            Address = "Default-Address",
            City = "Default-City",
            Region = "Default-Region",
            PostalCode = "00000",
            Country = "Germany",
            Phone = "00000",
            Fax = "00000"};
        dc.Customers.InsertOnSubmit(newCust);
        dc.SubmitChanges();
    }
}
```

Anschließend erhält das über die `NorthwindDataContext` -Eigenschaft `Customers` ansprechbare Objekt vom Typ `System.Data.Linq.Table<Customers>` den Auftrag **InsertOnSubmit()**. Für die Datenbankaktualisierung sorgt das per **SubmitChanges()** beauftragte `NorthwindDataContext`-Objekt.

27.3.2.3 Entitäten (Tabellenzeilen) löschen

Zum Löschen einer Tabellenzeile, die mit einem LINQ to SQL - Abfrageergebnis assoziiert ist, gibt man dem per `NorthwindDataContext` - Eigenschaft `Customers` ansprechbaren Objekt vom Typ `System.Data.Linq.Table<Customers>` den Auftrag **DeleteOnSubmit()**:

```
static void DeleteEntity() {
    var cust = dc.Customers.Where(c => c.CustomerID == "NEWEN");
    if (cust.Count() > 0) {
        Console.WriteLine("Deleting Entity NEWEN\n");
        dc.Customers.DeleteOnSubmit(cust.First());
        dc.SubmitChanges();
    }
}
```

Für die Datenbankaktualisierung sorgt das per **SubmitChanges()** beauftragte `NorthwindDataContext`-Objekt.

28 Anhang

28.1 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 fette horizontale Linien begrenzt.

Sie verfügen dann stets auch über dieselbe Auswertungsrichtung. Mit Ausnahme der Zuweisungsoperatoren sind alle binären Operatoren 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

$$x = y = z$$

ausgewertet als

$$x = (y = z)$$

Operator	Bedeutung
<code>x.y</code>	Member-Zugriff
<code>()</code>	Methoden- oder Delegatenaufruf
<code>[]</code>	Array-Index
<code>++, --</code>	Postinkrement bzw. -dekrement
<code>new</code>	Objekterzeugung
<code>typeof</code>	Typermittlung
<code>checked</code>	Ganzzahl-Überlaufdiagnose
<code>-</code>	Vorzeichenumkehr
<code>!</code>	Negation
<code>~</code>	Bitweise Negation
<code>++, --</code>	Präinkrement bzw. -dekrement
<code>(Typ)</code>	Typumwandlung
<code>*, /</code>	Punktrechnung
<code>%</code>	Modulo (Divisionsrest)
<code>+, -</code>	Strichrechnung

Operator	Bedeutung
+	Stringverkettung
<<, >>	Links- bzw. Rechts-Shift
>, <, >=, <=, is	Vergleichsoperatoren
==, !=	Gleichheit, Ungleichheit
&	Bitweises UND
&	Logisches UND (mit unbedingter Auswertung)
^	Exklusives bitweises ODER
^	Exklusives logisches ODER
	Bitweises ODER
	Logisches ODER (mit unbedingter Auswertung)
&&	Logisches UND (mit bedingter Auswertung)
	Logisches ODER (mit bedingter Auswertung)
??	Null-Koaleszenz
? :	Konditionaloperator
=	Wertzuweisung
+=, -=, *=, /=, %=	Wertzuweisung mit Aktualisierung
=>	Lambda-Operator

28.2 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**
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 IL-Codes beim Laden
So werden technische Defekte abgefangen.
- Sicherheitsprüfung (CAS, Code Access Security)
So werden Sicherheitsrisiken minimiert.
- Der JIT-Compiler in der CLR übersetzt den IL-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.

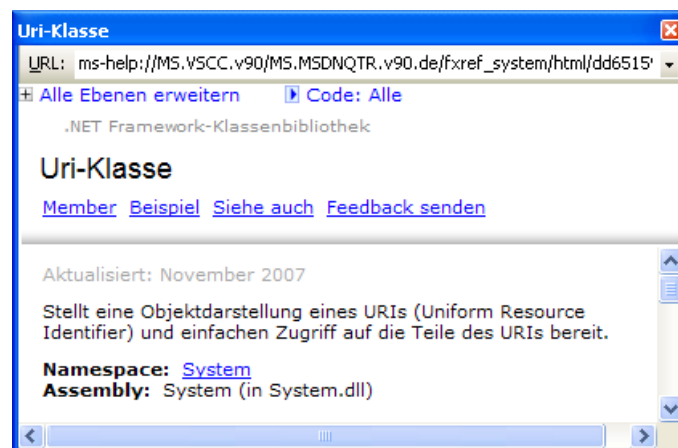
Mit dem SDK-Werkzeug **ngen.exe** kann der IL-Code eines Assemblies im Maschinencode übersetzt werden, so dass bei der Ausführung kein JIT-Compiler mehr benötigt wird. Allerdings zeigen sich dabei nur in speziellen Situationen Leistungsverbesserungen.

Aufgabe 5

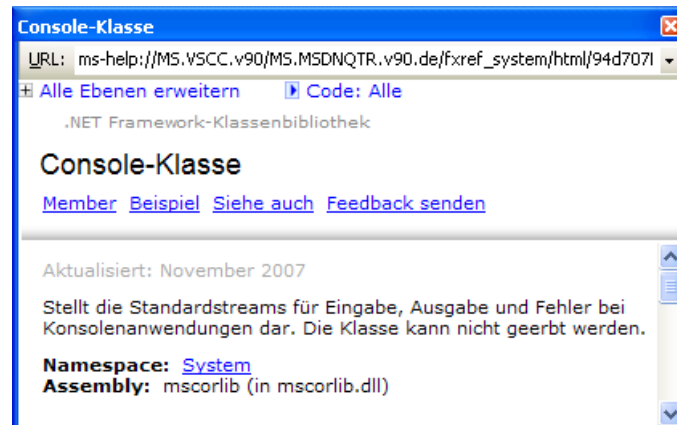
Namensräume und Assemblies sind zwei voneinander *unabhängige* Organisationsstrukturen:

- 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**, wie die folgende Ausgabe des noch vorzustellenden *Document Explorers* zeigt:



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 <i>Microsoft Intermediate Language</i> (kurz: MSIL). Diesen Zwischencode übersetzt der JIT-Compiler in der CLR in Maschinencode.
FCL	Die Standardklassenbibliothek der .NET - Plattform wird als <i>Framework Class Library</i> bezeichnet. Sie enthält für praktisch alle Routineaufgaben der Programmierung (z.B. graphische Benutzeroberflächen, Dateiverarbeitung, Netzwerkprogrammierung) ausgereifte Lösungen.
CLS	Microsoft hat unter dem Namen <i>Common Language Specification</i> einen Sprachumfang definiert, den <i>jede</i> .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 (<i>Component Object Model</i>) soll vom .NET – Framework abgelöst werden.

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 muss groß geschrieben werden.
 Der 2. Aufruf klappt: Der Modifikator **public** ist allerdings nicht erforderlich.
 Der 3. Aufruf klappt: Statt **void** ist auch der Rückgabetyt **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 Rückgabetyt **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

Der im **WriteLine()**-Parameter unmittelbar auf die Zeichenkette folgende Plus-Operator wird zuerst ausgeführt und bewirkt eine Verkettung von Zeichenfolgen, wobei die Zahl 3.3 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
<pre>using System; class Prog { static void Main() { Console.WriteLine("3,3 + 2 = " + (3.3 + 2)); } }</pre>	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
<code>6/4*2.0</code>	double	2.0	Abarbeitung mit Zwischenergebnissen: 6/4*2.0 1*2.0
<code>(int)6/4.0*3</code>	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
<code>(int)(6/4.0*3)</code>	int	4	Abarbeitung mit Zwischenergebnissen: (int)(6/4.0*3) (int)(1.5*3) (int)4.5
<code>3*5+8/3%4*5</code>	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 $2 \neq 3$ 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 $\wedge$ wird aufgrund seiner höheren Priorität vor dem Operator $\&\&$ ausgewertet.

`la2` erhält den Wert **true**, weil die runden Klammern dafür sorgen, dass der Operator $\wedge$ zuletzt ausgewertet wird.

`la3` erhält den Wert **false**

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\Exp

Aufgabe 5

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\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 3

In der Anweisung

```
return Anzahll;
```

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
Definition einer Instanzmethode mit Referenz-Wertparameter	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 Beispielpogramm 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 5.4.1.3.2).

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.

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 verschiedener Parameterliste. 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 (Generische Typen und Methoden)

Aufgabe 1

Sie finden einen Lösungsvorschlag in:

...\BspUeb\Weitere .NETte Typen\Generische Typen und Methoden\GratisIndexer

Über die Aufgabenstellung hinausgehend enthält der Lösungsvorschlag eine Erweiterung der Klasse **Person** um die Methode **CompareTo(Person p)** und um 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 Pentium 4 - CPU, Single Core, 3 GHz Taktfrequenz):

```
Zeit in Millisek. für List<int>: 15,625
Zeit in Millisek. für ArrayList: 140,625
```

Das zugehörige Programm finden Sie in:

...\BspUeb\Weitere .NETte Typen\Generische Typen und Methoden\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 keine Felder, Konstanten und statische Member erlaubt.
- **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 und können z.B. über Arrays mit entsprechenden Interface-Elementdatentypen (zusammen mit Instanzen aus anderen kompatiblen Klassen und Strukturen) verwaltet werden. Eine Klasse „erbt“ allerdings nichts von den Schnittstellen, sondern sie gibt Verpflichtungserklärungen ab und muss die entsprechenden Implementierungsleistungen erbringen.

Kapitel 9 (Einstieg in die WinForms - Programmierung)

Aufgabe 1

1. falsch
2. richtig
3. falsch
4. richtig

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\WinForms\Steuerelemente\DM2Euro

Aufgabe 3

Der Quellcode-Editor unserer Entwicklungsumgebung bietet u.a. folgende Leistungen:

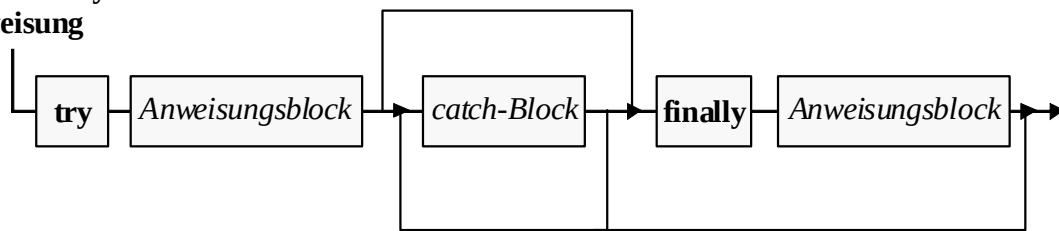
- Farbliche Unterscheidung von Programmbestandteilen
- Syntaxprüfung
- Automatische Code-Vervollständigung
- Automatische Formatierung (z.B. Einrückungen)
- Kontextsensitive Hilfe zum markierten Sprachbestandteil (Aufruf mit der Taste **F1**)
- Unterstützung bei der Umgestaltung des Quellcodes (Refaktorisierung)

Kapitel 10 (Ausnahmebehandlung)

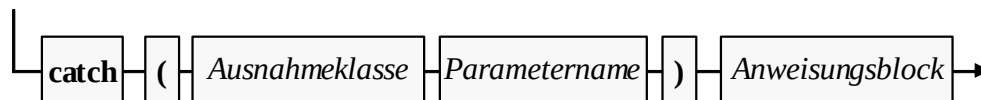
Aufgabe 1

Lösungsvorschlag:

**try-catch-finally -
Anweisung**



catch-Block



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\Ausnahmen\DuaLog\ArgumentOutOfRangeException

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Ausnahmen\EinfachStapelEx

Kapitel 11 (Attribute)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Attribute\MyAttribute

Kapitel 12 (Ein-/Ausgabe über Datenströme)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\EA\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 (Threads)

Aufgabe 1

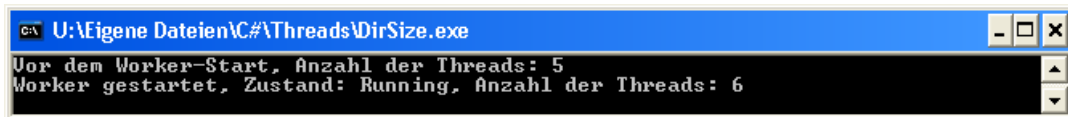
- Falsch
- Falsch
- Richtig

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Threads\DirSize

Dieses Programm zeigt in der Konsole einige Informationen über die vorhandenen Threads an, z.B.:



```
U:\Eigene Dateien\C#\Threads\DirSize.exe
Vor dem Worker-Start, Anzahl der Threads: 5
Worker gestartet, Zustand: Running, Anzahl der Threads: 6
```

Kapitel 14 (Netzwerkprogrammierung)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Netzwerk\WebBrowser

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 (Grafikausgabe mit dem GDI+)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\GDI+\Farben und Werkzeuge\HatchStyle

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\GDI+\Linien, Kurven und Flächen\Schneeflocke

Im Lösungsvorschlag wird ein **PointF**-Array für **FillPolygon()** aufgebaut. Bei der Berechnung neuer Zwischenpunkte für eine Ausgangslinie mit Startpunkt *a* (eine **PointF**-Instanz) wird eine **SizeF**-Instanz *s* gebildet und als Vektor behandelt mit:

- Richtung: wie die Ausgangslinie
- Länge: ein Drittel der Ausgangslinienlänge

Den ersten Zwischenpunkt z_1 erhält man durch die Addition (vgl. Abschnitt 15.4.2)

$$z_1 = a + s$$

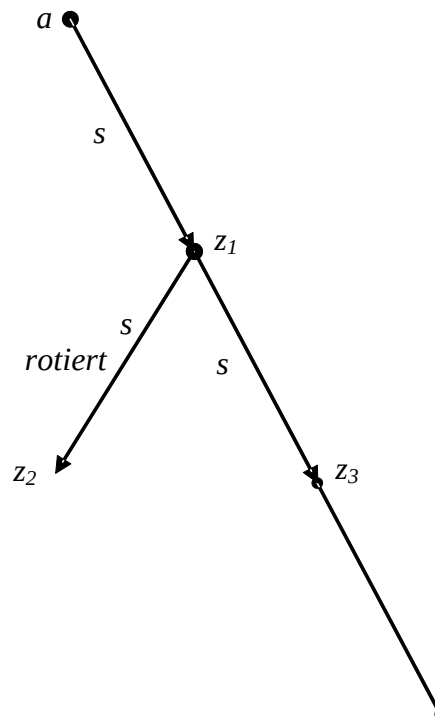
Den dritten Zwischenpunkt z_3 erhält man durch die Addition

$$z_3 = z_1 + s$$

Dann wird s um 60° gedreht, und für den zweiten Zwischenpunkt verwendet:

$$z_2 = z_1 + s$$

Wie man einen Vektor im $\mathbb{R}^2$ rotiert, wurde im Rahmen einer früheren Übungsaufgabe erläutert (siehe Abschnitt 4.11). Im Lösungsvorschlag werden die neuen Zwischenpunkte also gemäß folgender Skizze konstruiert:



Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\GDI+\Pfade und Zeichenbereiche\RoundButton

Es ist mir leider nicht gelungen, den Schalter ausrasten zu lassen, wenn der Mauszeiger bei gedrückter Taste den Bereich des Steuerelements verlässt. Der Plan, das Ereignis **MouseLeave** entsprechend zu behandeln, scheiterte, weil das Ereignis nicht ausgelöst wird, wenn eine Maustaste gedrückt ist.

Kapitel 17 (Menüs)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\WinForms\Menüs\DotNet 2\Bearbeiten

Kapitel 18 (Dialogfenster)**Aufgabe 1 und Aufgabe 2**

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\WinForms\Dialogboxen\Weitersuchen

Kapitel 19 (Anwendungs- und Benutzereinstellungen)**Aufgabe 1**

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Einstellungen\XML\MitSizeLocation

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Einstellungen\Registry\MitSizeLocation

Kapitel 20 (Symbol- und Statusleiste)**Aufgabe 1**

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\WinForms\Steuerelemente\Symbolleisten\Erweitert

Kapitel 21 (Drucken)**Aufgabe 1**

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Drucken\BoundsChecker

Kapitel 22 (Zwischenablage, Ziehen und Ablegen)**Aufgabe 1**

Mit der **Clipboard**-Methode **SetDataObject()** und einem Objekt der Klasse **DataObject** lassen sich diverse Datenformate gleichzeitig in der Zwischenablage unterbringen, wobei es für die Zusammenstellung keine Restriktionen gibt (siehe Abschnitt 22.4.2).

Kapitel 23 (Ressourcen)**Aufgabe 1**

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Ressourcen\RitchisTextEditorRes

Literatur

- Albahari, B., Drayton, P. & Merrill, B. (2002). *C# Essentials* (2nd ed.). Sebastopol, CA: O'Reilly.
- 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. (2007). *Einführung in das Programmieren mit C# 2.0*. 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. (2008). *Datenbankprogrammierung mit Visual C# 2008*. Unterschleißheim: Microsoft Press.
- 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>
- Eller, F. (2001). *C# lernen*, 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.icssharpcode.net/TechNotes/SharpDevelopCodingStyle03.pdf>
- Kühnel, A. (2008). *Visual C# 2005*. 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. (2006). *Praxisbuch Objektorientierung. Professionelle Entwurfsverfahren*. Bonn: Galileo
- Liberty, J. (2002). *Programming C#* (2nd ed.). Sebastopol, CA: O'Reilly.
- 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.
- Microsoft Inc. (2007a). *C# Language Specification. Version 3.0*. Online-Dokument: <http://download.microsoft.com/download/3/8/8/388e7205-bc10-4226-b2a8-75351c669b09/CSharp%20Language%20Specification.doc>
- Microsoft Inc. (2007b). *Official Guidelines for User Interface Developers and Designers*. Online-Dokument: <http://www.microsoft.com/downloads/details.aspx?familyid=B996E1E7-A83A-4CAE-936B-2A9D94B11BC5&displaylang=en>
- 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. (2002). *Windows-Programmierung mit C#*. Unterschleißheim: Microsoft Press.
- 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). *.NET-Oberflächen mit Windows Forms oder WPF*. Online-Dokument: <http://www.heise.de/developer/NET-Oberflaechen-mit-Windows-Forms-oder-WPF--/artikel/141990>
- Schwichtenberg, H. (2009c). *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>
- 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>
- Troelsen, A. (2002). *C# und die .NET-Plattform*. Bonn: mitp-Verlag

Stichwortregister

:
-Operator 22
+
+-Operator 65
1
15.10.1 Grafikpfad 469
A
Abfrage
 parametrisierte 698
Ablaufsteuerung 113
Abort() 383
abstract 240, 260
Abstraktion 1
AcceptButton 296, 514
AcceptChanges() 706, 708
AcceptTcpClient() 414
ACK-Bit 401
Acos() 186
Active Template Library 629
Add()
 ListBox 483
AddBezier() 471
AddLines() 471
AddRange() 483
 ListBox 483
 ToolStrip 494, 555
AddRef() 626
Aggregation 172
Aggregieren 676
Aktionsabfrage 674
Aktualisierungsoperatoren 102
Aktualparameter 149, 153
Algorithmen 8
Alignment 437
AllowDrop 606
AllowItemReorder 498
Alpha-Kanal 434
AnchorStyles 301
Anonyme Klassen 730
Anonyme Methoden 280
ANSI-Textdateien 362
Antialiasing 441, 459
Anweisung
 zusammengesetzte 113
Anweisungen 112
Anweisungsblöcke 112
Anwendungsdomäne 694
Anwendungseinstellungen 534
Anwendungsfenster 272
Anwendungskonfigurationsdatei 534, 696, 718
Anzeigebereich 475
APM 388
Append()
 StringBuilder 218
Application 272, 345, 394, 537
ApplicationException 332, 334
ApplicationScopedSettingAttribute 540

ApplicationSettingsBase 538
Apply 518
ARGB-Farbmodell 434
ArgumentOutOfRangeException 332, 337
Arithmetische Operatoren 91
Arithmetischer Ausdruck 91
Array 200, 255
 mehrdimensional 207
ArrayList 209, 245
Array-Parameter 152
as-Operator 237
Assembler 12
Assembly 15, 17
AssemblyInfo.cs 345
Assembly-Metadaten 18
Assoziativität 104
AsyncCallback 389
Asynchronous Programming Model 388
ATL 629
AttachDbFilename 693
Attribut
 XML 535
Attribute 339
Aufzählungen 218
Ausdrücke 90
Ausdrucksanweisungen 112
Ausgabe
 im Konsolenfenster 65
Ausgabeparameter 151
Ausgabetyt 30
Ausnahme 319
Auswahlabfrage 674
Auswertungsfunktionen 676
Auswertungsreihenfolge 104
Auswertungsrichtung 104
Autoboxing 200
AutoCheck 480
AutoFlush 361
AutoScroll 451, 488

B

base-Konstruktoren 230
Basisklasse 227
Baumansicht 648
Bearbeitungsmodus 705
Bedingte Anweisung 113
Befehlsschalter 293
 runde 478
Befehlszeilenargumente 120
BeginEdit() 705
BeginGetHostEntry() 413
BeginInvoke() 388
 Control 393
BeginPrint
 -Handler 581
Benutzerinstanz 694, 711, 717
Bezeichner 63
Beziehung 672
Bézier-Splines 443
Bildlaufleiste 450
Bildschirmkoordinaten 431
Binärdateien 355
binäre
 Operatoren 91
Binäre
 Gleitkommadarstellung 73

BinaryFormatter 363
 BinaryReader 356
 BinaryWriter 356
 BindingNavigator 722
 BindingSource 722
 Bindung
 COM 627
 Bitfelder 346
 Bitflags 346
 Bitmap 465
 Bitorientierte Operatoren 98
 Bitweises UND 99
 Block 78
 Blockanweisung 78, 112
 BMP 466
 Bogen 442
 bool 72
 bool-Literale 83
 BorderStyle 291
 Boxing 198
 break 119
 break-Anweisung 127
 BringToFront() 301
 Brush 438
 Brushes 440
 Button 293
 ButtonBase 479
 byte 72
 Bytecode 16

C

C++ 79
 Call Back - Routinen 267
 Camel Casing 64
 CancelButton 296, 514
 CancelEdit() 705
 CancelEventArgs 523
 CanRead 351
 CanSeek 351
 CanWrite 351
 Capacity 210
 CAS 20
 case-Marke 119
 casting 99
 Casting-Operator 100
 catch-Block 322
 CGI 407
 Change()
 Timer 392
 char 72
 char-Literale 84
 CheckBox 479
 checked
 Anweisung 108
 Compileroption 108
 Operator 107
 Checked
 Menüitem 509
 CIL 16
 Clear() 474
 ListBox 484
 ClearType 459
 ClientSize 431
 Clipboard 500, 511, 596, 597
 Clipboard ID 594
 clipbrd.exe 596
 Clipping 473
 Clone() 256
 Close() 351

StreamWriter 360
 CloseAllFigures() 472
 CloseFigure() 471
 CLR 13, 20
 CLS 13, 17, 89
 CLSID 625
 CoCreateInstance() 626
 Code Access Security 20
 Color 435
 ColorDialog 436, 527
 Columns
 ListView 659
 COM 12, 19, 341, 621, 624
 COM+ 625
 ComboBox 482
 CommandBuilder 709
 Common Gateway Interface 407
 Common Intermediate Language 16
 Common Language Runtime 13
 Common Language Specification 13, 17
 Common Type System 135, 197, 228
 Strukturen 197
 CompareTo() 248
 String 215
 Compiler 15
 COM-Server 625
 ConnectionString 692
 Console 65
 const 80, 142
 ConstraintCollection 702
 ContainerControl 291
 ContainsText() 500, 511
 ContentAlignment 291
 ContentPanel 557
 continue-Anweisung 127
 Control 291
 ControlCollection 291
 controls 267
 Controls 291
 Convert 86
 Copy()
 File 368
 Cos() 187
 CPU 12
 Create()
 File 368
 WebRequest 402
 CreateDirectory() 369
 CreateSubKey() 551
 CreateText()
 File 368
 csc 30
 csc.exe 15
 csc.rsp 50
 ctor 158
 CTS 135, 197, 228
 Strukturen 197
 CultureInfo 376
 CurrentThread() 378

D

dangling else 116
 DashCap 436, 437
 DashPattern 437
 DashStyle 436, 437
 DataColumn 701
 DataDirectory 694
 DataFormats 594, 609
 DataObject 601

- DataReader 713
- DataRelation 708
- DataRow 701
- DataRowState 706
- DataRowVersion 707
- DataSet-Designer 720
- DataTable 701
- Dateiauswahldialog 528
- Dateiverzeichnisbaum 651
- Datenbankdiagramm 725
- Datenbankmanagementsystem 669
- Datenkapselung 133, 145
- Datenquelle 713
- Datenstrom 349
- Datentyp 69
- Datentypen
 - elementare 71
- DateTime 131
- DB2 691
- DBMS 669
- DBNull 703
- DCOM 625
- Deadlocks 386
- debug
 - Compiler-Option 331
- decimal 72, 75, 82, 95, 111
- default
 - bei einem Typparameter 247, 251
 - switch 119
- Definitionstabellen 18
- Deklarationsbereich 78, 79
- Deklarative Programmierung 339
- delegate
 - Schlüsselwort 277
- Delegaten 276
 - generische 281
- Delete()
 - Directory 369
 - File 368
- DeleteValue() 550
- DeMorgan 106
- Denormalisierte
 - Gleitkommadarstellung 75
- deprecated 386
- Dialogfenster
 - modal 513
 - nicht-modal 520
- DialogResult 517
- Dictionary 248
- DictionaryEntry 331
- Directory 369, 652
- DirectoryInfo 369
- DirectoryInfo 663
- Direktive
 - using 21
- Dispose() 352, 425
- DLL-Hölle 18
- DllImport 621
- Dns 412
- DNS 412
- Do~Drag~Drop() 609
- Dock 303, 643
- Document Explorer 47
- DoEvents() 394
- Dokumentationskommentar 63
- Domain Name System 412
- DomainUpDown 488
- Doppelpufferung 428
- do-Schleife 126
- Dots Per Inch 456

- double 72, 73, 95
- Double 110
 - Epsilon 111
- DoubleBuffered 429
- Download
 - Dateien 405
- DownloadData() 410
- DownloadFile() 405
- DPI 456
- DpiY 457
- Drag & Drop 605
- Drag Once 721
- DragSize 611
- DrawArc() 442
- DrawBezier() 443
- DrawBeziers() 444
- DrawClosedCurve() 443
- DrawCurve() 443
- DrawEllipse() 441
- DrawImage() 465
- DrawLine() 440
- DrawLines() 440
- DrawPath() 471, 473
- DrawPie() 442
- DrawPolygon() 442
- DrawRectangle() 441
- DrawString() 453, 585
- DriveInfo 652
- DriveType 653
- DropDown-Menü 495
- DropDownOpening 500, 501
- Drucken 567
- Durchfall 119
- Dynamische Bindung 238

E

- Eigenschaft
 - Syntaxdiagramm 60
- Eigenschaften 4, 163
 - klassenbezogene 166
- Eingabefelder 298
- Eingabefokus 295
- Einschränkende Konvertierung 100
- einstellige
 - Operatoren 91
- Element
 - XML 534
- Elementare Datentypen 71
- Elementauflistungs-Editor 504, 560
- else-Klausel 114
- EnableAutoDragDrop 611
- Enabled
 - MenuItem 500, 510
- Encoding 362
- EndCap 436
- EndEdit() 705
- EndGetHostEntry() 413
- EndInvoke() 388
- Endlosschleife 127
- Enter()
 - Monitor 381
- Entitätsklasse 734
- Entwurfsmuster
 - Ereignisbehandlung 285
- Enumerable 732
- Enumerationen 218
- Environment 184, 623
- Epsilon 111
- Eratosthenes 223

Ereignisse 282
 ERRORLEVEL 320
 Erweiternde Konvertierungen 99
 Erweiterungsmethoden 731
 Escape-Schaltfläche 296
 Escape-Sequenzen 65, 66, 84
 Euklidischer Algorithmus 8, 130
 event 286
 Exception 319
 Exception-Handler 322
 ExecuteNonQuery() 697
 ExecuteReader() 697
 ExecuteScalar() 697
 EXIF 466
 Exists()
 Directory 369
 File 368
 Exit 184
 Exit() 320
 Klasse Environment 320
 Monitor 381
 Exitcode 320
 Exklusives logisches ODER 97
 Explizite
 Interface-Implementation 264
 extern 621

F

Families 449
 Farbauswahl
 Standarddialog 435
 Farbauswahl-
 Standarddialog 527
 Farben 434
 Farbverlauf 440
 FCL 13, 21
 Felder 2
 klassenbezogene 165
 verdeckte 235
 Festkomma 66
 FieldOffsetAttribute 347
 File 367
 FileAccess 354
 FileInfo 367, 663
 FileMode 354
 FileShare 355
 FileStream 350, 353
 FileSystemWatcher 370
 Fill()
 DataAdapter 699
 FillClosedCurve() 445
 FillEllipse() 445
 FillPath() 473
 FillPie() 445
 FillPolygon() 445
 FillRectangle() 445
 FillSchema() 700
 Filter
 SaveFileDialog 528
 finally-Block 322
 Find 524
 Find()
 DataRowCollection 703
 FindMembers() 342
 FixedDialog 514
 Flackern 428
 Fließkommazahl 73
 float 72, 73, 82, 95
 floating point number 73

Flush() 351
 Flussdiagramm 113
 Font 448
 FontDialog 451, 525
 FontFamily 449, 457
 FontStyle 449, 480
 foreach 122
 Formalparameter 149
 Formatierte Ausgabe
 im Konsolenfenster 66
 Formatierung
 von Java-Programmen 61
 FormBorderStyle 294
 FormClosing 531
 FormClosing-Ereignis 541, 551
 Formular 271
 HTML 407
 Formulardesigner 38, 178, 309
 Formulare 271
 for-Schleife 123
 Frame 398
 Framework 257
 Framework Class Library 13, 21
 Fremdschlüssel 673, 726
 Fremdschlüssel-Restriktion 673
 FromArgb() 435
 FromImage() 468
 FROM-Klausel 674
 Frühe Bindung
 COM 627
 FTP-Protokoll 406
 Füllmodus 446
 Funktions-Pointer 277

G

GAC 88
 Ganzzahlarithmetik 91
 Ganzzahliliterale 81
 Garbage Collector 160
 GDI 423
 GDI+ 423
 GenericTypographic 459
 Generische
 Delegaten 281
 Klassen 246
 Methoden 253
 Gerätekontext 423
 Gerätekoordinaten 463
 GET
 CGI-Parameter 409
 GetChildRows() 709
 GetCurrentDirectory() 369
 GetCustomAttribute() 342
 GetCustomAttributes() 343
 GetDataObject() 596, 604
 GetDataPresent() 604, 608
 GetDirectories() 369, 652, 663
 GetDrives() 652
 GetDriveType() 652
 GetEnvironmentVariable() 623
 GetExecutingAssembly() 617
 GetFiles() 369, 663
 GetHeight() 457
 GetHostEntry() 412
 GetLastError() 624
 GetLength()
 Array 208
 GetLogicalDrives() 652
 GetObject() 617

GetPixel() 468
 GetResponse() 403
 GetResponseStream 403
 GetString() 617
 GetType() 135, 198
 GetValue()
 RegistryKey 550
 Geviertgröße 449
 GIF 466
 Gleitkomma 66
 Gleitkommaarithmetik 73, 91
 Gleitkommadarstellung
 binär 73
 Gleitkommaliterale 82
 Gleitkommazahl 73
 global 22
 Global Assembly Cache 19
 Globale Variablen 71
 goto-Anweisung 119
 Graphics 423
 Graphics Device Interface 423
 GraphicsPath 470, 475
 GripStyle 555
 GROUP BY 676
 GroupBox 481
 GUI 267
 GUID 625

H

Hallo 27
 HasErrors 705
 HasVersion() 707
 HatchBrush 439
 Hauptfenster 272
 Heap 71, 143, 157
 Height 273
 Hide() 518
 Hollywood-Prinzip 267
 Host-Name 412
 HRESULT 633
 HTTP 402, 405
 HttpUtility 410
 HttpRequest 402

I

ia64 14
 ICloneable 256
 ICMP 399
 IComparable 255, 261, 665
 IComparer 665
 IDataObject 604, 607
 IDictionary 331
 IDisposable 352
 IDL 19
 IEEE-754 73
 IEnumerable<T> 732
 if-Anweisung 113
 IFormatter 363
 IID 626
 IL 15
 ILDasm.exe 638
 Image
 ToolStripButton 555
 Index
 in Datenbanktabellen 671
 Indexer 220
 IndexOf()

String 216
 IndexOutOfRangeException 203, 320
 Inflate() 433
 information hiding 133, 145
 Initialisierung 69, 77
 Initialisierungslisten 206
 InitializeComponent() 182, 312
 Inkonsistenz 669
 Inlining 164, 331
 INNER JOIN 675
 Innere Klassen 175
 In-Process-Server 625
 InputBox() 88
 Insert()
 StringBuilder 218
 InstalledPrinters 569, 574
 Instanzen 191
 Instanzinitialisierer 729
 Instanzvariablen 71, 140
 Integrated Security 694
 IntelliSense 39
 Interaction 88
 Interface 255
 Interface Definition Language 19
 Intern() 212
 Interner String-Pool 212
 Internet Control Message Protocol 399
 Interrupt() 386
 Invalidate() 428, 429
 InvalidConstraintException 709
 IN-Vergleichsoperator (SQL) 675
 Invoke()
 Control 393
 IPAddress 412
 IP-Adresse 412
 IPHostEntry 412
 IPv4 399
 IPv6 399
 IsBackground 376
IsDefined() 341, 347
 ISerializable 362
 IsInterned() 214
 IsNaN() 110
 IsNegativeInfinity() 110
 is-Operator 237
 IsPositiveInfinity() 110
 IsStyleAvailable() 449
 ItemArray 703
 Items
 ListBox 483
 ListView 659
 IUnknown 626

J

jagged arrays 208
 Java 12, 16, 112
 JIT-Compiler 20
 Join() 386
 JPEG 466

K

Kachelansicht 658
 Kapselung 133
 Kardinale
 Splines 443
 Key
 Registry 548

- Keys 496
- Kindfenster 288
- Klammern 104
- Klasse
 - Syntaxdiagramm 58
- Klassen 133
 - anonyme 730
 - innere 175
 - statische 169
- klassenbezogene
 - Eigenschaften 166
- Klassenmethode 29
- Klassenvariablen 71
- Kochs Schneeflocke 477
- Kodierung 358
- Kollektionen 124
- Kombinationsfeld 482
- Kommentar 62
- Komponentenfach 722
- Konditionaloperator 103
- Konfiguration
 - Visual Studio 40
- Konsolenfenster
 - Formatierte Ausgabe 66
- Konstanten 80, 142
- Konstruktoren 157
 - statische 168
- Kontextmenü
 - .NET 1.x 511
 - .NET 2 501
- Kontrollkästchen 479
- Kontrollstrukturen 113
- Konvertierung
 - einschränkende 100
 - erweiternde 99
- Kreissegment 442

L

- Label 289
- Lambda-Ausdrücke 730
- LayoutStyle
 - MenuStrip 498
- Leere Anweisung 112
- Left 290
- Length 164, 204, 215
 - Stream 351
- lexikographische Priorität 215
- LIFO-Stapel 247
- LIKE-Vergleichsoperator (SQL) 675
- LinearGradientBrush 440
- LineCap 437
- LineJoin 440
- Links-Shift-Operator 98
- LINQ to SQL 734
- Linux 13
- List<T> 246, 281, 427
- ListBox 482
- Listenfeld 482
- ListView 657
- ListViewItem 657
- ListViewSubItem 658
- Literale 81
- Load-Ereignis
 - bei Form 540, 552
- LoadFile()
 - RichTextBox 530
- Local Server 625
- LocalFileSettingsProvider 541
- lock 379

- Logische Operatoren 96
- Logisches ODER 97
- Logisches UND 97
- Lokale Variablen 71

M

- MAC-Adresse 398
- machine.config 534
- MacOS-X 13
- Main() 9, 57
- MainMenu 507
- MakeTransparent() 468
- managed code 24
- Manifest 18
- Margins 569
- Marshaler 622
- Maschinencode 12
- Master-Detail -
 - Beziehung 672
- Math-Klasse 93
- Matrix 465
- MaxValue
 - Double 110
- MeasureCharacterRanges() 460
- MeasureString() 458
- MeasureText() 460
- Mehrfachvererbung 230, 258
- Member 5
- MemberInfo 342
- memory leaks 160
- Menü 493
- MenuActivate 564
- Menübefehl 495
- MenuDeactivate 564
- MenuItem 507
- MenuItemCollection 507
- MenuStrip 493
- Menüzeile 493, 507
- MessageBox 49
- Metafile 469
- Methode
 - Syntaxdiagramm 60
- Methoden 145
 - Aufruf 153
 - Definition 146
 - generische 253
 - Modifikatoren 148
 - rekursive 169
 - Rückgabewert 148
 - statische 167
 - Überladen 154
 - überschreiben 238
- Microsoft Intermediate Language 13, 15
- Microsoft SQL Server 2008 Express Edition 676
- MIDL 627
- MinimumSize 302
- MissingSchemaAction 701
- Modale Dialoge 513
- Modified 528
- Modifikatoren
 - bei Methoden 148
- Modularisierung 134
- module 31
- Module 18
- Modulo 92
- Monitor 380
- Mono 13, 23, 571
- MouseDown 307, 610
- MouseEventArgs 490

MouseHover 564
MouseLeave 565
MouseMove 610
MouseWheel 490
Move()
 File 368
MoveFile() 624
MS-Agent 639
mscorlib.dll 31
MSDN 41
MSIL 13, 15
Müllsammelr 160
MulticastDelegate 277
Multiline 299
Multitasking 373
Multithreading 373
Murphy's Law 319
Mutex 380
MySQL 691

N

Namen 63
 von Klassen 139
Namensparameter 344
Namensräume 21
namespace 21
NameValueCollection 411
NaN 109
NAS 694
NDoc 63
Nebeneffekt 90
Nebeneffekte 92, 97
Negation 97
Network Attached Storage 694
NetworkStream 414, 416
Netzwerkprogrammierung 397
new-Modifikator 233
new-Operator 156, 157
NewRow() 706
Next() 205
ngen.exe 20
noconfig 50
NonSerialized 363
Normalisieren 672
Normalisierte
 Gleitkommandarstellung 74
Novell 23
null 144, 155
NULL (SQL) 675
Null-Koaleszenz - Operator 253
NullReferenceException 144
Number
 Klasse 321
NumericUpDown 486

O

O/R - Designer 735
Objektdiagramm 362
Objektinitialisierer 160
Objektserialisierung 362
Obsolete
 Attribut 339
ObsoleteAttribute 340
ODBC 691
Öffnungsmodus 354
Offset()
 Point 431

Rectangle 433
OLE 530
OLEDB 691
OLE-Objekt 612
OnBeforeExpand() 655
OnPaint() 425
OpCode 200
OpenFileDialog 530
OpenSubKey() 551
OpenType 448
operator+ 172
Operatoren 90
 Arithmetische 91
 bitorientierte 98
 logische 96
 überladen 171
 vergleichende 94
Operatorentabelle 741
Optionale Parameter 89
Optionsfeld 479
OracleClient 691
ORDER BY 676
Ordneransicht 651
ORM-Lösung 734
OSI-Modell 398
out 152
 -Compiler-Option 30
Out-Of-Process-Server 625
override 238, 239
Owner 521

P

PageScale 463
PageSettings 569
PageSetupDialog 589
PageUnit 431, 462
Paint 425
PaintEventArgs 427
Panel 451, 481
Parameter
 optionale 89
Parametrisierte Abfragen 698
params 152
Parent 291
partial 182, 311
Pascal 137
Pascal Casing 64
Peek() 361
Pen 436
Pens 438
Pfadname 368
PHP 409, 410
PInvoke 621
Platform Invocation Services 621
PNG 466
Point 431
PointToClient() 431, 607
PointToScreen() 431
Polling 390
Polymorphie 238, 263
Popup 510, 511
Port 399
Position
 Stream 350
Positionsparameter 344
POST
 CGI-Parameter 411
Postinkrement bzw. -dekrement 92
Postscript 448

Potenzfunktion 93
 Pow() 93
 Präinkrement bzw. -dekrement 92
 Preemptives Zeitscheibenverfahren 385
 Primärer Thread 375
 Primärschlüssel 671, 702
 PrimaryKey 702
 Primzahlen 128
 PrintController 573
 PrintDialog 576
 PrintDocument 570
 PrinterSettings 567
 PrintFormat 585
 PrintPage
 -Handler 581
 PrintPageEventArgs 581
 PrintPreviewControl 589
 PrintPreviewDialog 587
 Priorität 104
 Prioritäten 384
 Priority
 Thread 384
 private 141
 Privates Assembly 19
 Process 663
 Produktivität 3
 Programmablaufplan 113
 Projektion 733
 Projektmappen-Explorer 38
 properties 163
 Properties 4
 protected 232
 Provider 691
 Pseudozufallszahlengenerator 204
 Puffer
 StreamWriter 360
 Pulse()
 Monitor 381
 PulseAll() 382
 Punktoperator 145, 153

Q

QUERY_STRING 410
 QueryContinueDrag 611
 QueryInterface() 626
 QueueUserWorkItem() 663
 Quick-Info - Texte 555

R

Race Condition 379
 RAD 308
 RadioButton 479
 RadioCheck
 MenuItem 510
 Random 169, 204
 Rank 208
 Rapid Application Development 308
 Rastergrafik 465
 RCW 628
 RDBMS 671
 Read()
 SqlDataReader 713
 Stream 351
 ReadByte() 351
 ReadKey() 370
 ReadLine() 86
 readonly 142

Read-Only
 Eigenschaft 164
 ReadTimeout 418
 record 137
 Rectangle 433
 RectangleF 466
 Redistributable 13
 Redundanz 669
 Refactoring 315
 reference
 -Compiler-Option 31, 50, 88
 referentielle Integrität 673
 Referenz 48
 Referenzparameter 150
 Referenzsemantik 192
 Referenztabelle 18
 Referenztypen 69
 Referenzvariablen 155
 Reflection 341
 Reflexion 339
 Refresh() 429
 Region 475
 Registrierungsdatenbank 548
 Registry 533, 548, 627
 RegistryKey 550
 regsvr32.exe 628, 634
 RejectChanges() 708
 Rekursive Methoden 169
 Relationale Datenbanken 671
 Release() 626
 Reload() 541
 Remote Server 625
 Remove()
 StringBuilder 218
 Remove()
 ListBox 483
 Replace()
 StringBuilder 218
 Replace()
 String 216
 Reservierte Wörter 63
 ResetAbort() 384
 resgen.exe 615
 Resize()
 Array 203
 ResizeRedraw 429
 ResourceManager 616
 RESOURCES-Datei 615
 ResourceWriter 616
 Response-Datei 31
 Ressourcen 613
 Restmantis 74
 Resume() 386
 RESX-Datei 613
 ResXResourceReader 615
 ResXResourceWriter 614
 return 121, 148
 return-Anweisung 119, 148
 Returncode 328
 RichTextBox 490
 roaming profile 538
 Rollbalken 488
 Rollen 686
RotateFlip() 468
 Round-Robin 385
 Router 399
 RowState 705, 706
 RTF 453
 RTF-Format 490
 Rückgabewert 148

Run() 272
Runtime Callable Wrapper 628

S

SAN 694
Sandcastle 63
Satellitenassembly 613
Save()
 Einstellungen 541
SaveFile()
 RichTextBox 529
SaveFileDialog 528
ScaleTransform() 464
Schaltfläche 293
 runde 478
Scheduler 384
Schema einer Datenbank 700
Schleifen 122
Schließen
 von Datenströmen 351
Schnittstelle 255
Schriftarten-
 Standarddialog 525
Schriftschnitte 448
ScrollBar 488
ScrollBars 299, 514
Scroll-Ereignis 489
Scroll-Rad 490
sealed 242
Seek() 351
SeekOrigin 351
Seitenansicht 587
Seiteneinrichtung 589
Seitenkoordinaten 463
Seitentransformation 463
Sekundäre Fenster 513
Sekundäre Threads 375
Select() 421, 733, 734
SELECT-Befehl (SQL) 674
SelectedIndex 484
SelectedItems 485
SelectionColor
 RichTextBox 593
SelectionFont
 RichTextBox 593
SelectionMode
 ListBox 485
Semaphore 380
Serialisieren 362
Serializable 362
SerializationException 363
SetClip() 473
SetCurrentDirectory() 369
SetDataObject() 600
SetLastError 624
SetLastWriteTime()
 File 368
SetPixel() 468
SetText() 597
SettingChanging 542
SettingsProvider 537, 541
SetValue()
 RegistryKey 550
Shortcut 508
ShortcutKeys 496
Show() 272
ShowApply 525
ShowDialog() 517
ShowInTaskbar 521

ShowShortcutKeys 496
ShutDown() 414
Sichtbarkeitsbereich 79, 143
Sieb des Eratosthenes 223
Signatur 154, 233
Silverlight 42
Sin() 187
Singlethread-Apartment 341
Size 431
Skalarprodukt 186
Sleep() 377
Smalltalk 133
SmoothingMode 441
SOAP 366, 397
SoapFormatter 366
Socket 400, 413
Solaris 13
SolidBrush 438
Sort() 255
Sorted
 ComboBox, ListBox 484
Späte Bindung 238
 COM 627
Speicherlöcher 160
Spin-Control 485
Splines
 Bézier 443
 kardinale 443
Split() 225
SplitContainer 646
Splitter 643
SplitterPanel 647
SQL 673
 SELECT-Befehl 674
SQL Server 2008 Express Edition 676
SQL Server Browser Dienst 689
SqlClient 691
SqlCommand 697, 711
SqlCommandBuilder 711
SqlConnection 692, 711
SqlDataAdapter 699, 711
SqlDataReader 713
SqlParameter 698
SqlServerCe 691
Sqrt() 186
STA 341
Stabilität 3
Stack 71, 143, 153
 Überlauf 171
Standarddialog
 Drucken 576, 587
 Farbauswahl 435, 527
 Öffnen 530
 Schriftarten 525
 Schriftartenwahl 451
 Seiteneinrichtung 589
 Sichern 528
Standarddialoge 525
Standardereignis 314
Standardkonstruktor 157
Standardressourcendatei 617
Standardschaltfläche 296
StartCap 436
startfähige Klasse 57
Startklasse 9
StartPosition 515
StartsWith()
 String 216
starvation 385
STAThread 341, 529, 531, 596

static 165
 Statische
 Felder 165
 Klassen 169
 Konstruktoren 168
 Methoden 167
 Statische Methoden
 Verdecken 234
 StatusStrip 553, 562
 Statusvariable 329
 Steuerelemente 267
 Storage Area Network 694
 Stream 350
 streams 349
 String 211, 255
 Methoden 214
 StringBuilder 217
 StringFormat 455
 String-Pool 212
 Strings
 vergleichen 215
 verketteten 214
 Strom 349
 struct 136, 192
 StructLayoutAttribute 347
 Structured Query Language 673
 Struktogramm 171
 Strukturansicht 648
 Strukturen 191
 Strukturiertes Programmieren 136
 SubmitChanges() 738
 Substring()
 String 215
 Suspend() 386
 switch-Anweisung 117
 Symbolleisten 553
 Synchronisierter Block 380
 Synchronized() 546
 Syntaxdiagramm 58
 System.IO 349
 SystemBrushes 440
 SystemColors 436
 Systemfarben 436
 SystemFonts 451
 SystemInformation 611
 SystemPens 438

T

TabIndex 296
 TableAdapter 715
 TableAdapterManager 723
 TableLayoutPanel 304
 TableMappings 699
 target
 -Compiler-Option 30
 TCP 399
 TCP/IP
 SQL-Server 688
 TcpClient 416
 TCP-Flags 401
 TcpListener 414
 Telnet 415
 TextChanged 299, 563
 TextDataFormat 598
 Textdateien 355, 359
 Textfelder
 einzeilig 298
 mehrzeilig 299
 TextReader 358

TextRenderer 460
 TextRenderingHint 459
 TextureBrush 439
 TextWriter 358
 this 145, 154, 159, 162
 Indexer 222
 Thread 374
 ThreadAbortException 383
 ThreadInterruptedException 386
 Threadpool 373, 387
 ThreadPool 663
 ThreadPriority 384
 Threads 373
 throw 332
 throw-Anweisung 119
 TIFF 466
 Timeout
 NetworkStream 416
 Timer
 Threading 391
 WinForms 491
 Time-To-Live 401
 TlbImp.exe 628
 ToChar() 98
 ToInt32() 86
 ToLongTimeString() 492
 ToLower() 216
 ToolStrip 553, 562
 ToolStripContainer 557
 ToolStripItemCollection 494, 555, 563
ToolStripMenuItem 493
 ToolStripSeparator 497
 Tool-Tip - Texte 555
 ToolTipText 555, 556
 Top 290
 ToString()
 StringBuilder 218
 ToUpper() 216
 TranslateTransform() 464
 Transmission Control Protocol 399
 Transparentfarbe 297
 Transparenz 434
 TreeNode 648
 TreeNodeCollection 648
 TreeNode-Editor 650
 TreeView 648
 Trennzeichen 61
 Trimming 456
 TrimToSize() 210
 TrueType 448
 try-catch-finally 321
 T-SQL 673
 TTL 401
 Type 135
 Typformalparameter 247, 253
 Typisiertes DataSet 714
 Typ-Metadaten 18
 Typsicherheit 69
 Typtest-Operator 237
 Typumwandlung 99

U

Überladen
 von Methoden 154
 von Operatoren 171
 Überladung 214, 234
 Überlauf 107
 Übernehmen 518
 Überschreiben von Methoden 238

UDP 400
uint 72
ulong 72
UML 5
Umschalter 479
unäre
 Operatoren 91
Unboxing 198
unchecked-Operator 109
undefinierte Werte 109
Unendlich 109
Ungarische Notation 479
UNICODE 530
Unicode-Escape-Sequenzen 84
Unicode-Zeichensatz 63
Unified Modeling Language 5
Uniform Resource Identifier 402
Union 347
UNIX 13
unmanaged code 621
Unterlauf 111
Untermenü 494
Unterprogrammtechnik 136
Unterstrich 64
Update() 429
 DataAdapter 700
UpDown-Steuererelement 485
UploadValues 411
URI 402
URL 402
UrlDecode() 410, 411
UrlEncode() 410
URL-Kodierung 408
User Datagram Protocol 400
User Instance 694
user32.dll 621
UserAgent 403
UserScopedSettingAttribute 540
ushort 72
using
 -Direktive 21
using-
 Anweisung 352
UTF8Encoding 358
UTF8-Kodierung 361

V

Validierung
 von Einstellungen 542
value 163
Value
 Registry 549
ValueChanged-Ereignis
 NumericUpDown 486
 ScrollBar 489
ValueType 196
var 729
Variablen 67
 globale 71
 lokale 71
Variablendeklaration 77, 112
Verbindungsloser Datenzugriff 690
Verbindungszeichenfolge 692
Verbundanweisung 78, 112
Verdecken 233
Verdeckte Felder 235
Vererbung 227
Verfügbarkeit 176
Vergleich 94

Vergleichen
 von Strings 215
Vergleichsoperatoren 94
Verifikation 20
Verketteten
 von Strings 214
Verkettete Liste 221
versiegelt 242
Versiegelte
 Klassen 242
 Methoden 242
Verweis 48
virtual 238, 239
Visible 272
Vista 13
Visual C# 2008 Express Edition 41, 108, 121, 177
VScrollBar 488

W

Wahrheitstafeln 96
Wait()
 Monitor 381
WaitCallback 387
WaitSleepJoin-
 Zustand eines Threads 381
Wanderndes
 Benutzerprofil 538
WebBrowser 404, 419
WebClient 405, 411
Webdienste 397
WebDings 448
WebResponse 403
Wertparameter 150
Wertsemantik 192
Werttypen 69
Wertzuweisung 78
where
 Typrestriktion 249
WHERE-Klausel 675
while-Schleife 125
widgets 267
Width 273
Wiederholungsanweisungen 122
Windows-Registry *Siehe* Registry
Windows-SDK 7, 43, 52, 158, 615, 628
winexe 51
WinForms 267
WinFX 268
WingDings 448
WM_QUIT 275
WMI 654
WordPad 558
WordWrap 514
Write() 65, 351
WriteByte() 351
WriteLine() 65
Write-Only
 Eigenschaft 164

X

x64 14
x86 14
XML 366, 533, 534, 613
XmlSerializer 366

Z

- Z-Anordnung 300, 643
- Zeichenbereich 473
- Zeichenketten 211
- Zeichenkettenlitterale 84
- Zeilenumbruch 224
- Zeitscheibenverfahren 385
- Ziehen und Ablegen 605
- ZoomFactor 514
- Zufallszahlen 169, 204
- Zugriffsschutz 133, 176
- Zugriffstaste 495
- Zusammengesetzte
 - Anweisung 113
- Zuweisungsoperator 101
- zweistellige
 - Operatoren 91