

B. Baltes-Götz

Einführung in das Programmieren mit C# 2.0

2007.12.06

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 © 2007; URT

Vorwort

Dieses Manuskript entstand als Begleitlektüre zum C# - Einführungskurs, den das Universitäts-Rechenzentrum Trier (URT) über zwei Semester (WiSe 2006/2007 - SoSe 2007) angeboten hat, sollte aber auch für das Selbststudium geeignet sein.

Lerninhalte und -ziele

C# ist eine von der Firma Microsoft für das so genannte .NET – Framework entwickelte, mittlerweile von der European Computer Manufacturers Association (ECMA) standardisierte Programmiersprache, die auf den Vorbildern Java und C++ aufbaut, aber auch etliche Weiterentwicklungen bietet. Es ist zu erwarten, dass die .NET - Plattform bei der Softwareentwicklung unter Windows in den nächsten Jahren stark an Bedeutung zunehmen wird. Auf dem EDV-Arbeitsmarkt ist bereits jetzt ein deutliches Interesse an .NET und an der hier bevorzugten Programmiersprache C# festzustellen. Durch das Mono-Projekt der Linux-Szene sind .NET-Programme keinesfalls auf die Windows-Welt beschränkt.

Der Kurs 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 zur Verfügung stehen, auf dem das *.NET – Framework* der Firma Microsoft in einer Version ab 2.0 installiert ist. Außerdem empfiehlt sich die Verwendung einer Entwicklungsumgebung, z.B. *Microsoft Visual Studio 2005 Express Edition* oder *Sharp Develop 2*. Die erwähnte Software ist kostenlos verfügbar, kann also auch auf einem Privat-PC ohne Investitionen genutzt werden. Nähere Hinweise zur Installation und zur Verwendung der Software folgen in Abschnitt 2.

Dateien zum Kurs

Die aktuelle Version dieses Manuskripts ist zusammen mit den behandelten Beispielen und Lösungsvorschlägen 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 mir zu wenig Zeit für eine sorgfältige Kontrolle des Texts, so dass einige Fehler und Mängel verblieben sein dürften. Entsprechende Hinweise sind sehr willkommen!

Inhaltsverzeichnis

1	EINLEITUNG	1
1.1	Beispiel für die objektorientierte Softwareentwicklung mit C#	1
1.1.1	Objektorientierte Analyse und Modellierung	1
1.1.2	Objektorientierte Programmierung	5
1.1.3	Algorithmen	7
1.1.4	Startklasse und Main()-Methode	8
1.1.5	Ausblick auf Anwendungen mit graphischer Benutzerschnittstelle	10
1.1.6	Zusammenfassung	10
1.2	Das .NET – Framework	11
1.2.1	Installation	12
1.2.2	C#-Compiler und MSIL	12
1.2.3	Assemblies und Metadaten	14
1.2.4	CLR und JIT-Compiler	17
1.2.5	Namensräume und FCL	18
1.2.6	Zusammenfassung	20
1.3	Übungsaufgaben zu Kapitel 1	21
2	WERKZEUGE ZUM ENTWICKELN VON C# - PROGRAMMEN	23
2.1	C# - Entwicklung mit Texteditor und Kommandozeilen-Compiler	23
2.1.1	Editieren	23
2.1.2	Übersetzen in MSIL	25
2.1.3	Ausführen	28
2.1.4	Programmfehler beheben	28
2.2	.NET – SDK	31
2.3	Integrierte Entwicklungsumgebungen	32
2.3.1	Microsoft Visual Studio 2005 Professional	32
2.3.1.1	Terminal-Server – Umgebung	32
2.3.1.2	Konfiguration beim ersten Start	33
2.3.1.3	Eine erste GUI-Anwendung	35
2.3.2	Microsoft Visual C# 2005 Express Edition	39
2.3.2.1	Installation	39
2.3.2.2	Ein erstes Konsolen-Projekt	43
2.3.2.3	Referenzen und Ausgabetyp setzen	45
2.4	SharpDevelop 2.1	47
2.5	Übungsaufgaben zu Kapitel 2	48
3	ELEMENTARE SPRACHELEMENTE	49
3.1	Einstieg	49
3.1.1	Aufbau von einfachen C# - Programmen	49
3.1.2	Syntaxdiagramme	50
3.1.2.1	Klassendefinition	51
3.1.2.2	Methodendefinition	52
3.1.2.3	Eigenschaftsdefinition	52
3.1.3	Hinweise zur Gestaltung des Quellcodes	53
3.1.4	Kommentare	54
3.1.5	Namen	54
3.1.6	Übungsaufgaben zu Abschnitt 3.1	56

3.2	Ausgabe bei Konsolenanwendungen	56
3.2.1	Unformatierte Ausgabe einer (zusammengesetzten) Zeichenfolge	56
3.2.2	Formatierte Ausgabe	57
3.2.3	Übungsaufgaben zu Abschnitt 3.2	58
3.3	Variablen und elementare Datentypen	58
3.3.1	Klassifikation von Datentypen und Variablen	60
3.3.2	Elementare Datentypen	61
3.3.3	Vertiefung: Darstellung reeller Zahlen im Speicher	63
3.3.3.1	Binäre Gleitkommadarstellung	63
3.3.3.2	Dezimale Gleitkommadarstellung	64
3.3.4	Variablendeklaration, Initialisierung und Wertzuweisung	66
3.3.5	Blöcke und Deklarationsbereiche für lokale Variablen	67
3.3.6	Konstanten	68
3.3.7	Literale	69
3.3.7.1	Ganzzahlliterale	69
3.3.7.2	Literale für reelle Zahlen	70
3.3.7.3	bool-Literale	71
3.3.7.4	char-Literale	72
3.3.7.5	Zeichenkettenliterale	72
3.3.8	Übungsaufgaben zu Abschnitt 3.3	73
3.4	Einfache Techniken für Benutzereingaben	74
3.4.1	Via Konsole	74
3.4.2	Via InputBox	75
3.5	Operatoren und Ausdrücke	77
3.5.1	Arithmetische Operatoren	78
3.5.2	Methodenaufrufe	80
3.5.3	Vergleichsoperatoren	81
3.5.4	Logische Operatoren	84
3.5.5	Vertiefung: Bitorientierte Operatoren	85
3.5.6	Typumwandlung (Casting)	86
3.5.7	Zuweisungsoperatoren	88
3.5.8	Konditionaloperator	89
3.5.9	Auswertungsreihenfolge	90
3.5.10	Übungsaufgaben zu Abschnitt 3.5	91
3.6	Über- und Unterlauf bei numerischen Variablen	93
3.6.1	Überlauf bei Ganzzahltypen	93
3.6.2	Unendliche und undefinierte Werte bei den Typen float und double	94
3.6.3	Überlauf beim Typ decimal	96
3.6.4	Unterlauf bei den Gleitkommatypen	96
3.7	Anweisungen	97
3.7.1	Überblick	97
3.7.2	Bedingte Anweisung und Verzweigung	98
3.7.2.1	if-Anweisung	98
3.7.2.2	switch-Anweisung	103
3.7.3	Wiederholungsanweisungen	107
3.7.3.1	Zählergesteuerte Schleife (for)	108
3.7.3.2	Iterieren über Kollektionen (foreach)	109
3.7.3.3	Bedingungsabhängige Schleifen	110
3.7.3.4	Endlosschleifen	112
3.7.3.5	Schleifen(durchgänge) vorzeitig beenden	112
3.7.4	Übungsaufgaben zu Abschnitt 3.7	114

4	KLASSEN UND OBJEKTE	117
4.1	Überblick, historische Wurzeln, Beispiel	117
4.1.1	Einige Kernideen und Vorzüge der OOP	117
4.1.1.1	Datenkapselung	117
4.1.1.2	Vererbung	118
4.1.1.3	Realitätsnahe Modellierung	119
4.1.2	Strukturierte Programmierung und OOP	119
4.1.3	Auf-BruCh zu echter Klasse	120
4.2	Instanzvariablen (Felder)	122
4.2.1	Deklaration mit Wahl der Schutzstufe	122
4.2.2	Konstante und schreibgeschützte Instanzvariablen	123
4.2.3	Deklarationsbereich und Ablage im Hauptspeicher	124
4.2.4	Initialisierung	124
4.2.5	Zugriff auf Instanzvariablen	125
4.3	Methoden	126
4.3.1	Methodendefinition	127
4.3.1.1	Formalparameter	128
4.3.1.2	Modifikatoren	131
4.3.1.3	Rückgabewerte und return-Anweisung	131
4.3.1.4	Methodenrumpf	132
4.3.2	Methodenaufruf und Aktualparameter	132
4.3.3	Methoden überladen	133
4.4	Objekte	134
4.4.1	(Wieder)verwendung von Klassen bzw. Objekten	134
4.4.2	Referenzvariablen deklarieren	135
4.4.3	Objekte erzeugen	136
4.4.4	Objekte initialisieren über Konstruktoren	137
4.4.5	Abräumen überflüssiger Objekte durch den Garbage Collector	140
4.4.6	Objekt-Referenzen als Wertparameter	140
4.4.7	Rückgabewerte vom Referenztyp	141
4.4.8	this als Referenz auf das aktuelle Objekt	142
4.5	Eigenschaften	142
4.6	Klassenbezogene Member	143
4.6.1	Felder und Eigenschaften	144
4.6.2	Methoden	145
4.6.3	Konstruktoren	146
4.6.4	Klassen	147
4.6.5	Wiederholung zur Kategorisierung von Variablen	147
4.7	Vertiefungen zum Thema Methoden	148
4.7.1	Rekursive Methoden	148
4.7.2	Operatoren überladen	149
4.8	Aggregieren und Schachteln von Klassen	151
4.8.1	Aggregation	151
4.8.2	Innere Klassen	153
4.9	Verfügbarkeit von Klassen und Klassenbestandteilen	154

4.10	Bruchrechnungsprogramm mit Winforms-Benutzerschnittstelle	155
4.10.1	Projekt mit Vorlage Windows-Anwendung anlegen	155
4.10.2	Steuerelemente aus der Toolbox übernehmen	156
4.10.3	Steuerelemente gestalten	157
4.10.4	Eigenschaften der Steuerelemente ändern	157
4.10.5	Der Quellcode-Generator	158
4.10.6	Assembly mit der Bruch-Klasse einbinden	159
4.10.7	Ereignisbehandlungsmethoden anlegen	160
4.11	Übungsaufgaben zu Kapitel 4	161
5	WEITERE .NETTE TYPEN	165
5.1	Strukturen	165
5.1.1	Referenz- versus Wertsemantik	165
5.1.2	Unterschiede zwischen Klassen und Strukturen	168
5.1.3	Zur Eignung von Strukturen im Bruchrechnungs-Projekt	168
5.1.4	Strukturen im Common Type System der .NET – Plattform	169
5.2	Boxing und Unboxing	170
5.3	Arrays	171
5.3.1	Array-Referenzvariablen deklarieren	172
5.3.2	Array-Objekte erzeugen	172
5.3.3	Arrays benutzen	174
5.3.4	Beispiel: Beurteilung des .NET - Pseudozufallszahlengenerators	174
5.3.5	Initialisierungslisten	176
5.3.6	Objekte als Array-Elemente	176
5.3.7	Mehrdimensionale Arrays	177
5.3.7.1	Rechteckige Arrays	177
5.3.7.2	Mehrdimensionale Arrays mit unterschiedlich langen Elementen	178
5.3.8	Die Kollektionsklasse ArrayList	179
5.4	Zeichenketten	181
5.4.1	Die Klasse String für konstante Zeichenkette	181
5.4.1.1	String als Write Once - Klasse	181
5.4.1.2	Elementare Operationen mit Strings	182
5.4.2	Die Klasse StringBuilder für veränderliche Zeichenketten	185
5.5	Enumerationen	186
5.6	Vertiefung: Die Indexer der Klassen und Strukturen	187
5.7	Übungsaufgaben zu Kapitel 5	189
6	VERERBUNG UND POLYMORPHIE	193
6.1	Das Common Type System (CTS) des .NET – Frameworks	195
6.2	Definition einer abgeleiteten Klasse	196
6.3	Der Zugriffsmodifikator protected	197
6.4	base-Konstruktoren und Initialisierungs-Sequenzen	198
6.5	Geerbte Methoden durch spezialisierte Varianten verdecken	199
6.6	Geerbte Felder verdecken	201
6.7	Überschreiben von Methoden (Polymorphie)	202

6.8	Abstrakte Methoden und Klassen	204
6.9	Versiegelte Methoden und Klassen	205
6.10	Übungsaufgaben zu Kapitel 6	206
7	INTERFACES	209
7.1	Interfaces definieren	210
7.2	Interfaces implementieren	211
7.3	Interfaces als Referenzdatentypen	213
7.4	Übungsaufgaben zu Kapitel 7	214
8	EINSTIEG IN DIE WINFORMS-PROGRAMMIERUNG	215
8.1	Ein Projektrahmen zum Üben	216
8.2	Elementare Klassen im Windows Forms - Framework	219
8.2.1	Anwendungsfenster und die Klasse Form	220
8.2.2	Windows-Nachrichten und die Klasse Application	222
8.3	Delegaten und Ereignisse	224
8.3.1	Delegaten	224
8.3.1.1	Delegatentypen definieren	225
8.3.1.2	Delegatenobjekte erzeugen und aufrufen	225
8.3.1.3	Delegatenobjekte kombinieren	227
8.3.2	Ereignisse	227
8.3.2.1	Behandlungsmethoden registrieren	228
8.3.2.2	Ereignisse anbieten	230
8.4	Steuerelemente	232
8.4.1	Einsatz von Steuerelementen am Beispiel des Labels	233
8.4.1.1	Steuerelemente erzeugen und konfigurieren	234
8.4.1.2	Ab in den Container	235
8.4.1.3	Ereignisbehandlungsmethoden	236
8.4.2	Befehlsschalter	237
8.4.2.1	Beispiel	237
8.4.2.2	Eingabefokus und Standardschaltfläche	239
8.4.2.3	Bitmaps auf Schaltflächen	240
8.4.3	Textfelder	241
8.4.4	Spezielle Layout-Techniken	243
8.4.4.1	Z-Anordnung	243
8.4.4.2	Verankern	244
8.4.4.3	Andocken	245
8.4.4.4	TableLayoutPanel	247
8.4.4.5	Automatische Positions- und Größenanpassungen an die Schriftart des Formulars	248
8.5	Zusammenfassung	251
8.6	Übungsaufgaben zu Kapitel 8	252
9	AUSNAHMEBEHANDLUNG	253
9.1	Unbehandelte Ausnahmen	253

9.2	Ausnahmen abfangen	255
9.2.1	Die try-catch - Anweisung	255
9.2.2	Programmablauf bei der Ausnahmebehandlung	257
9.2.3	Vergleich mit der traditionellen Ausnahmebehandlung	259
9.2.4	Diagnostische Ausgaben	260
9.3	Ausnahme-Klassen im .NET - Framework	261
9.4	Ausnahmen werfen (throw)	262
9.5	Ausnahmen definieren	263
9.6	Übungsaufgaben zu Kapitel 9	264
10	ATTRIBUTE	266
10.1	Attribute vergeben	266
10.2	Attribute auswerten	267
10.3	Attribute definieren	268
10.4	Attribute für Assemblies und Module	268
10.5	Übungsaufgaben zu Kapitel 10	269
11	EIN-/AUSGABE ÜBER DATENSTRÖME	271
11.1	Datenströme aus Bytes	271
11.1.1	Das Grundprinzip	271
11.1.2	Wichtige Methoden und Eigenschaften der Basisklasse Stream	272
11.1.3	Schließen von Datenströmen	273
11.1.4	Ausnahmefehler abfangen	274
11.1.5	FileStream	275
11.1.5.1	Öffnungsmodus	276
11.1.5.2	Zugriffsmöglichkeiten für den eigenen Prozess	276
11.1.5.3	Optionen für gemeinsamen Zugriff	277
11.2	Verarbeitung von Daten mit höherem Typ	277
11.2.1	Schreiben und Lesen im Binärformat	278
11.2.2	Schreiben und Lesen im Textformat	280
11.2.3	Serialisieren von Objekten und Instanzen	283
11.3	Verwaltung von Dateien und Verzeichnissen	286
11.3.1	Dateiverwaltung	287
11.3.2	Ordnerverwaltung	288
11.3.3	Überwachung von Ordnern	289
11.4	Übungsaufgaben zu Kapitel 11	290
12	THREADS	292
12.1	Threads erzeugen	293
12.2	Threads synchronisieren	297
12.2.1	Die lock-Anweisung	297
12.2.2	Die Klasse Monitor	299
12.2.3	Koordination per Wait() und Pulse()	299
12.3	Threads abbrechen	300

12.4	Thread-Lebensläufe	302
12.4.1	Scheduling und Prioritäten	302
12.4.2	Zustände von Threads	303
12.5	Deadlocks	304
12.6	Timer	305
12.7	Übungsaufgaben zu Kapitel 12	307
13	NETZWERKPROGRAMMIERUNG	309
13.1	Wichtige Konzepte der Netzwerktechnologie	309
13.1.1	Das OSI-Referenzmodell	310
13.1.2	Zur Funktionsweise von Protokollstapeln	312
13.1.3	Optionen zur Netzwerkprogrammierung in C#	313
13.2	Internet - Ressourcen per Request/Response – Modell nutzen	314
13.2.1	Statische Webinhalte anfordern	314
13.2.2	Datei-Download per HTTP - Protokoll	316
13.2.3	Dynamische erstellte Webseiten per GET oder POST anfordern	318
13.2.3.1	Überblick	318
13.2.3.2	Arbeitsablauf	319
13.2.3.3	GET	320
13.2.3.4	POST	322
13.3	Internet-Adressen ermitteln	323
13.4	Socket-Programmierung	323
13.4.1	TCP-Server	324
13.4.2	TCP-Klient	326
13.4.3	Simultane Bedienung mehrerer Klienten	328
13.5	Übungsaufgaben zu Abschnitt 13	329
14	GRAPHIKAUSGABE MIT DEM GDI+	333
14.1	Die Klasse Graphics	333
14.2	GDI-Ressourcen schonen	334
14.3	Fensterrenovierungsmethoden	335
14.3.1	PaintEventHandler	335
14.3.2	OnPaint() überschreiben	336
14.3.3	Fensteraktualisierung per Programm anfordern	337
14.3.4	Ereignisse und On-Methoden	337
14.4	Positionen und Größen	339
14.4.1	Standardkoordinatensystem	339
14.4.2	Point und Size	339
14.4.3	Rectangle	341
14.5	Farben und Zeichenwerkzeuge	342
14.5.1	Farben	343
14.5.2	Pen	344
14.5.3	Brush	346
14.6	Linien, Kurven und Flächen	348
14.6.1	Einfache Linien und Figuren	348
14.6.2	Splines	350
14.6.3	Flächen	352

14.7	Text malen	354
14.7.1	Schriftarten und -familien	354
14.7.2	Standarddialog zur Schriftauswahl	357
14.7.3	Zeichen setzen	358
14.7.4	Maß halten	361
14.7.4.1	Schrifthöhe und Zeilenabstand	361
14.7.4.2	Textbreite	363
14.8	Transformationen	365
14.8.1	Auflösungserscheinungen	365
14.8.2	Metrische Maßeinheiten	366
14.8.3	Welttransformationen	368
14.9	Rastergrafik	369
14.10	Pfade, Zeichen- und Anzeigebereiche	370
14.10.1	Pfade	370
14.10.2	Zeichenbereiche	373
14.10.3	Anzeigebereiche	374
14.11	Übungsaufgaben zu Abschnitt 14	376
15	WEITERE STANDARDKOMPONENTEN FÜR WINFORMS-ANWENDUNGEN	377
15.1	Kontrollkästchen und Optionsfelder	377
15.2	Listen- und Kombinationsfelder	380
15.3	UpDown-Regler	382
15.4	Rollbalken	385
15.5	Timer-Komponente	387
15.6	Ein RTF-Editor auf RichTextBox-Basis	388
16	MENÜS	391
16.1	Erbschaftsangelegenheiten und wichtige Begriffe	391
16.2	Menüitems erzeugen	392
16.3	Ereignisse und Eigenschaften von Menüitems	393
16.4	Kontextmenüs	395
16.5	Übungsaufgaben zu Abschnitt 16	397
17	DIALOGBOXEN	399
17.1	Modale Dialoge	399
17.1.1	Konfiguration	399
17.1.2	Auftritt und Abgang	403
17.1.3	Übernehmen Sie keinen Aberglauben	404
17.2	Nicht-modale Dialoge	406
17.2.1	Konfiguration	406
17.2.2	Auftritt und Abgang	409
17.2.3	Kommunikation mit dem Anwendungsfenster	409

17.3	Standarddialoge	411
17.3.1	FontDialog	411
17.3.2	ColorDialog	413
17.3.3	SaveFileDialog	414
17.3.4	OpenFileDialog	416
17.4	Übungsaufgaben zu Abschnitt 17	418
18	EINSTELLUNGEN IN DER WINDOWS-REGISTRY SPEICHERN	419
18.1	Die Windows-Registrierungsdatenbank	419
18.2	Die FCL-Klassen RegistryKey und Registry	420
18.3	Übungsaufgaben zu Abschnitt 18	422
19	STATUSZEILE UND SYMBOLLEISTEN	423
19.1	Statuszeile	423
19.1.1	Jeder an seinen Platz	423
19.1.2	Statuszeilensegmente	425
19.2	Symbolleisten	429
19.3	Übungsaufgaben zu Abschnitt 19	432
20	DRUCKEN	433
20.1	Wichtige Klassen	433
20.1.1	Die Klasse PrinterSettings zur Verwaltung von Druckereinstellungen	433
20.1.2	Die Klasse PageSettings zur Verwaltung von Seiteneigenschaften	434
20.1.3	Die Klasse PrintDocument zur Verwaltung von Druckaufträgen	435
20.1.3.1	Eigenschaften und Methoden	436
20.1.3.2	Ereignisse	438
20.1.3.3	Beispiel	438
20.1.4	Der Standarddialog Drucken	440
20.2	Druckfunktionalität für den Editor auf RichTextBox-Basis	442
20.2.1	Beteiligte Objekte	443
20.2.2	BeginPrint - Behandlung	444
20.2.3	PrintPage - Behandlung	445
20.2.3.1	Ausgaberechteck ermitteln	445
20.2.3.2	Textausgabe mit DrawString()	447
20.3	Weitere Standarddialoge zur Unterstützung der Druckausgabe	449
20.3.1	Seitenansicht	449
20.3.2	Seiteneinrichtung	451
20.4	Übungsaufgaben zu Abschnitt 20	453
21	ZWISCHENABLAGE, ZIEHEN UND ABLEGEN	455
21.1	Verbesserte RTF-Unterstützung im Editorprojekt	455
21.2	Windows-Zwischenablagenformate	456
21.3	Ein Zwischenablagen-Inspektor	457

21.4	Daten in die Zwischenablage schreiben	459
21.4.1	Formatspezifische Set-Methoden der Klasse Clipboard	459
21.4.2	Die Clipboard-Methode SetDataObject()	460
21.5	Daten aus der Zwischenablage abrufen	462
21.5.1	Formatspezifische Get-Methoden der Klasse Clipboard	462
21.5.2	Die Clipboard-Methode GetDataObject()	463
21.6	Ziehen und Ablegen	465
21.6.1	Die Rolle der Zielanwendung	465
21.6.2	Die Rolle der Quellanwendung	469
21.7	Übungsaufgaben zu Abschnitt 21	471
22	RESSOURCEN	473
22.1	Ressourcendateien und Generatoren	473
22.1.1	RESX-Datei	473
22.1.2	TXT-Datei für Zeichenfolge-Ressourcen	475
22.1.3	RESOURCES-Datei	475
22.2	Ressourcen verwenden	476
22.3	Ressourcen-Verwaltung mit der Visual C# 2005 Express Edition	477
22.4	Übungsaufgaben zu Abschnitt 22	480
23	INTEROPERABILITÄT MIT SOFTWARE-ALTLASTEN	481
23.1	Funktionen in nativen DLLs nutzen	481
23.1.1	DLLs importieren	481
23.1.2	Parameter-Marshaling	481
23.1.3	Win32-Fehlercode ermitteln	483
23.2	Kooperation mit der COM-Technologie	483
23.2.1	Einige Begriff und Ideen aus der COM-Technologie	484
23.2.1.1	Typbibliotheken und Interfaces	484
23.2.1.2	COM-Server und -Klienten	484
23.2.1.3	Registrierung	485
23.2.2	COM-Server in .NET - Anwendungen nutzen	485
23.2.2.1	Proxy-Assembly mit Runtime Callable Wrapper	485
23.2.2.2	COM-Server mit Visual Basic erstellen	486
23.2.2.3	COM-Server in C# verwenden	491
24	VARIABLE ANSICHTEN	497
24.1	Aufteilungssteuerelemente	497
24.1.1	Splitter	497
24.1.2	SplitContainer	500
24.2	Baumansichten	502
24.2.1	Konstruktion	502
24.2.2	Datenerfassung	504
24.2.3	Verzeichnisbaum	505
24.2.4	Bilder für die Knoten	506
24.2.5	Ereignisse	508
24.2.6	Explorer Light	508

24.3	Listen und Tabellen	510
24.3.1	Konstruktion	511
24.3.2	Datenerfassung	512
24.3.3	Ereignisse	513
24.3.4	Verzeichniseinträge auflisten	513
24.3.5	Bilder für die Listenitems	515
24.3.6	Spezielle Sortierung	516
24.3.7	Explorer Midrange	517
24.4	Übungsaufgaben zu Abschnitt 24	519
25	DATENBANKPROGRAMMIERUNG MIT ADO.NET	521
25.1	Relationale Datenbanken	521
25.1.1	Tabellen	521
25.1.2	Beziehungen zwischen Tabellen	523
25.2	SQL	524
25.2.1	Überblick	524
25.2.2	Spalten abrufen	525
25.2.3	Fälle auswählen über die WHERE-Klausel	525
25.2.4	Daten aus mehreren Tabellen zusammenführen	525
25.2.5	Abfrageergebnis sortieren	526
25.2.6	Auswertungsfunktionen	526
25.2.7	Daten aggregieren	526
25.3	Microsoft SQL Server 2005 Express Edition	526
25.3.1	Eigenschaften	526
25.3.2	Ergänzende Installationen	527
25.3.2.1	Microsoft SQL Server Management Studio Express	527
25.3.2.2	Beispieldatenbank Northwind	528
25.3.3	Konfiguration	529
25.3.3.1	Rechteverwaltung	529
25.3.3.2	Netzwerkzugriff via TCP/IP erlauben	533
25.4	ADO.NET	534
25.4.1	Überblick	534
25.4.1.1	Verbindungsloses versus verbindungsorientiertes Arbeiten	534
25.4.1.2	Provider	534
25.4.1.3	ADO.NET - Namensräume	535
25.4.2	Die Connection-Klassen	536
25.4.2.1	Verbindungszeichenfolge	536
25.4.2.2	Eigenschaften, Ereignisse und Methoden	539
25.4.3	Die Command-Klassen	540
25.4.3.1	Eigenschaften und Methoden	540
25.4.3.2	Parametrisierte Abfragen	541
25.4.4	Die DataAdapter - Klassen	542
25.4.5	Die Klassen DataSet, DataTable, DataRow und DataColumn	544
25.4.6	Datenbank-Beziehungen in ein DataSet übernehmen	545
25.4.7	Datenbank-Update	546
25.4.8	Zusammenspiel der Klassen beim verbindungslosen Arbeiten	547
25.4.9	Einsatz des DataReaders	549
25.5	Datenbankunterstützung im Visual Studio 2005	550
25.5.1	Datenquellen	550
25.5.2	Anwendungsbeispiel	551
25.5.3	Datenbanken mit dem Visual Studio 2005 bearbeiten	557
25.5.3.1	Tabellen anzeigen und bearbeiten	559
25.5.3.2	Datenbankdiagramm erstellen	560
25.5.3.3	Beziehungen definieren	561

26	ANHANG	563
26.1	Operatorentabelle	563
26.2	Lösungsvorschläge zu den Übungsaufgaben	564
	Kapitel 1 (Einleitung)	564
	Kapitel 2 (Werkzeuge zum Entwickeln von C# - Programmen)	566
	Kapitel 3 (Elementare Sprachelemente)	567
	Abschnitt 3.1 (Einstieg)	567
	Abschnitt 3.2 (Ausgabe bei Konsolenanwendungen)	567
	Abschnitt 3.3 (Variablen und elementare Datentypen)	567
	Abschnitt 3.5 (Operatoren und Ausdrücke)	568
	Abschnitt 3.7 (Anweisungen)	569
	Kapitel 4 (Klassen und Objekte)	570
	Kapitel 5 (Weitere .NETte Typen)	570
	Kapitel 6 (Vererbung und Polymorphie)	571
	Kapitel 7 (Interfaces)	572
	Kapitel 8 (Einstieg in die WinForms-Programmierung)	573
	Kapitel 9 (Ausnahmebehandlung)	573
	Kapitel 10 (Attribute)	573
	Kapitel 11 (Ein-/Ausgabe über Datenströme)	573
	Kapitel 12 (Threads)	574
	Kapitel 13 (Netzwerkprogrammierung)	574
	Kapitel 14 (Graphikausgabe mit dem GDI+)	574
	Kapitel 16 (Menüs)	575
	Kapitel 17 (Dialogboxen)	575
	Kapitel 19 (Statuszeile und Symbolleisten)	575
	Kapitel 20 (Drucken)	575
	Kapitel 21 (Zwischenablage, Ziehen und Ablegen)	575
	Kapitel 24 (Variable Ansichten)	575
	LITERATUR	577
	STICHWORTREGISTER	579

1 Einleitung

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 werden einige Grundbegriffe der Informatik tangiert, die nicht ausführlich behandelt werden können, im weiteren Verlauf des eher praxisorientierten Kurses aber auch nicht vorausgesetzt werden.

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.

Dazu muss das Programm ...

- den betroffenen Gegenstandsbereich *modellieren*
Beispiel: In einem Programm zur Verwaltung einer Spedition sind z.B. Fahrer, Fahrzeuge, Servicestationen, Zielpunkte etc. und die kommunikativen Prozesse (Arbeitsabläufe) 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 Ziele einer Modellierung durch **Klassen** und deren Beziehungen. Man identifiziert die beteiligten **Objektsorten** und definiert für sie jeweils eine **Klasse**. Diese ist gekennzeichnet durch

- **Merkmale (Instanz- bzw. Klassenvariablen, Felder)**
Technisch handelt es sich bei einer Variablen um einen benannten Speicherplatz, der Werte bestimmten Typs (z.B. Zahlen, Zeichen) aufnehmen kann. 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).

- **Handlungskompetenzen (Methoden)**

In den Methoden sind die oben angesprochenen Algorithmen realisiert. Analog zu den Merkmalen gibt es Methoden, die von individuellen Objekten (alias: Instanzen) einer Klasse ausgeübt werden (z.B. das Kürzen bei Brüchen) und Methoden, die von der Klasse selbst ausgeführt werden (z.B. ein neues Objekt erstellen). Die Beziehung zwischen Klassen bzw. Objekten basiert auf der Anforderung und Ausführung von Methoden.

Eine Klasse ist ...

- einerseits **Bauplan für konkrete Objekte**, die im geplanten Programmen je nach Bedarf erzeugt und mit der Ausführung bestimmter Methoden beauftragt werden,
- kann aber auch selbst aktiv werden (Methoden ausführen und aufrufen).

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.

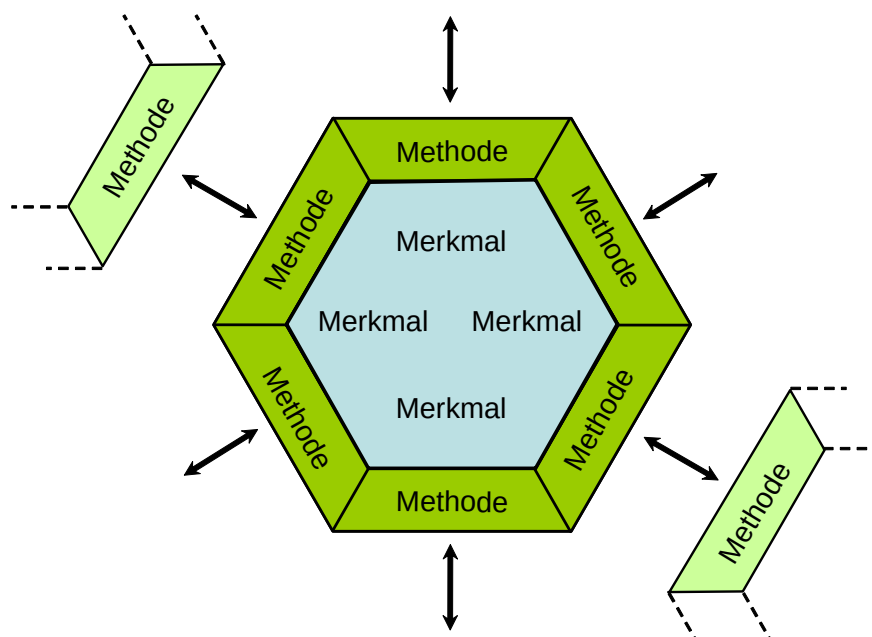
In unserem einfachen Beispiel kann man sich bei der objektorientierten Analyse vorläufig auf die Klasse der *Brüche* beschränken. Beim möglichen Ausbau des Programms kommen jedoch sicher weitere Klassen hinzu (z.B. Aufgabe, Übungsaufgabe, Testaufgabe, Trainer, Prüfer).

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

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 Abbildung zeigt, bilden die **Handlungskompetenzen** (Methoden) einer Klasse demgegenüber ihre öffentlich zugängliche **Schnittstelle** zur Kommunikation mit anderen Klassen:



Das Aufrufen einer Methode wird auch als **Nachricht** mit Handlungsauforderung interpretiert.

Aufgrund einer Klassendefinition lassen sich **Objekte** (Exemplare, Instanzen) erzeugen, die in der Lage sind, auf eine Reihe von Nachrichten mit einem bestimmten Verhalten zu reagieren. In unserem Beispiel sollte die Klasse Bruch z.B. eine Methode zum Kürzen besitzen. Dann kann einem konkreten Bruch-Objekt die Nachricht zugestellt werden, diese Methode auszuführen.

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 Signal hin kürzen, nicht unbedingt alltäglich, wenngleich möglich (z.B. als Lernspielzeug). Das objektorientierte Modellieren eines Gegenstandsbereiches ist nicht unbedingt eine direkte Abbildung der Realität, sondern eher 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:

- **Sicherheit**

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 der 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**).

Mit dem *Nutzer* einer Klasse ist im aktuellen Kontext nicht der Anwender eines fertigen Programms gemeint, sondern ein Programmierer, der die Klasse zur Softwareentwicklung verwendet. In einfachen Fällen werden Designer und Nutzer einer Klasse (zunächst) identisch sein. Manche Klassen (z.B. aus der .NET – Laufzeitbibliothek) werden jedoch von außerordentlich vielen Programmierern benutzt. Wir werden unser Beispiel noch um die Klasse Bruchrechnung 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. Es bietet sich z.B. an, zum Feld `zaehler` die *Eigenschaft* `Zaehler` (großer Anfangsbuchstabe!) zu definieren, welche dem Nutzer einer Klasse *Methoden* zum Lesen und

Setzen des Merkmals bietet, wobei er aber dieselbe Syntax wie beim direkten Zugriff auf ein Feld verwenden darf.

In der folgenden Anweisung wird z.B. der `zaehler` eines Bruch-Objektes mit dem Namen `b1` auf den Wert 4 gesetzt:

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

Während der Entwickler *Zugriffsmethoden* bereitstellt, 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 verhindern, dass sein `nenner` auf 0 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 Klassen-Designer 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()

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 in C# handelt es sich nach obigen Überlegungen eigentlich um *Zugriffsmethoden*, die aber syntaktisch wie (öffentlich verfügbare) Merkmale angesprochen werden. Hier hängt es wohl vom Adressaten eines Klassendiagramms (z.B. Entwickler-Teamkollege oder Anwender der Klasse) ab, welche Darstellung man wählt.
- Unten stehen die **Handlungskompetenzen** (Methoden).
In Anlehnung an eine in vielen Programmiersprachen (wie z.B. C#) übliche Syntax zur Methodendefinition gibt man für die Argumente eines Methodenaufrufs sowie für den Rückgabewert (falls vorhanden) den Datentyp an. Was mit dem letzten Satz genau gemeint ist, wird bald besprochen.

Für die Bruch-Klasse erhält man folgende Darstellung, wenn die Eigenschaften aus der Anwenderperspektive betrachtet werden:

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

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

Das 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 dieser Klassendefinition selbstverständlich jetzt noch fremd, doch sind die Variablendeklarationen sowie die Eigenschafts- und Methodenimplementationen als zentrale Bestandteile leicht zu erkennen:

```

using System;

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

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

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

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

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

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

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

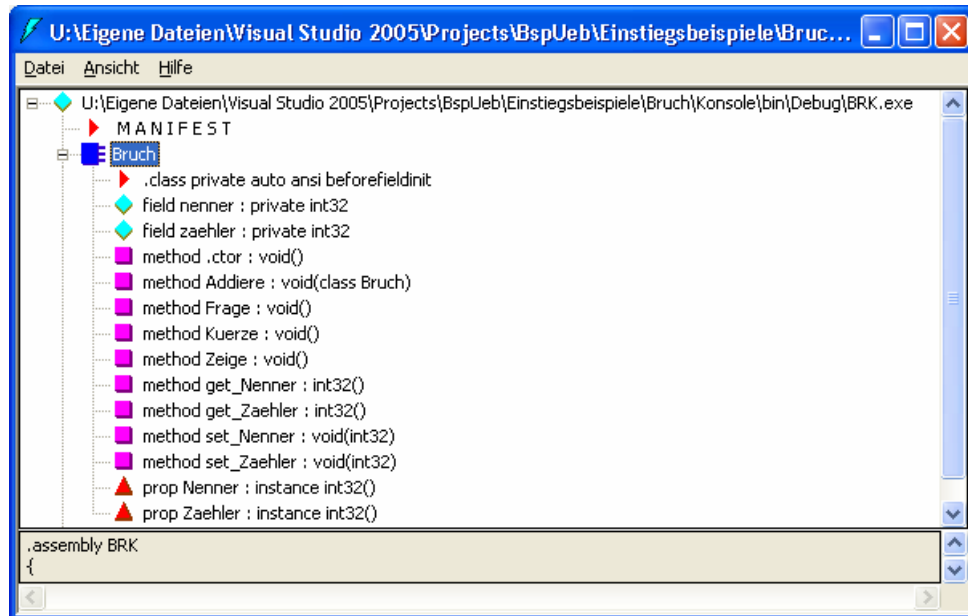
    public void Frage() {
        Console.Write("Zaehler: ");
        Zaehler = Convert.ToInt32(Console.ReadLine());
        Console.Write("Nenner : ");
        Nenner = Convert.ToInt32(Console.ReadLine());
    }
}

```

Durch die implizite **privat**-Deklaration der beiden Felder (zaehler, nenner) 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 kostenlosen .NET – SDK (*Software Development Kit*, siehe Abschnitt 2) 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

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

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

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

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

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

Folglich ist t auch ein Teiler von v :

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

Weil die Paare (u, v) und $(u, u - v)$ dieselbe Menge gemeinsamer Teiler besitzen, sind auch die größten gemeinsamen Teiler identisch.

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. Wir bringen 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 dem Ausführen 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

...\BspUeb\Einleitungsbeispiele\Bruch\Konsole\bin\Debug\

das Programm **BruchAddition.exe**.

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

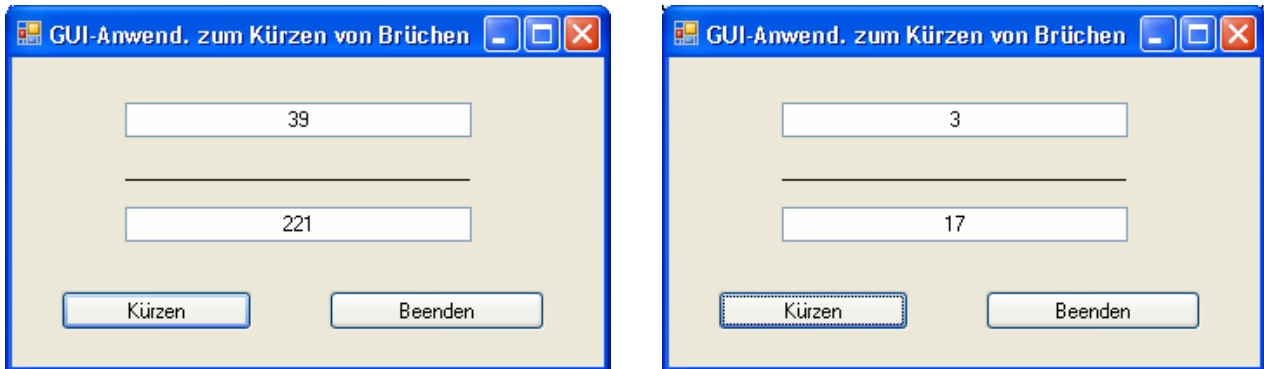
Wir haben zur Lösung der Aufgabe, ein Programm für einfache Operationen mit Brüchen zu erstellen, zwei Klassen mit folgender Rollenverteilung definiert:

- 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.
 - 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 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 Namensерweiterung **.cs**. Das .NET-Framework erlaubt aber auch Quellcodelateien 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 erarbeitet 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\Einleitungsbeispiele\Bruch\GUI\bin\Debug\`

das Programm **BruchKürzen.exe**.

1.1.6 Zusammenfassung

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:

- 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).
- 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.
- Zum Starten einer Anwendung wird eine ihrer Klassen vom Laufzeitsystem aufgefordert, die Methode **Main()** auszuüben. Ein Hauptzweck dieser Methode besteht oft darin, Objekte zu erzeugen, die anschließend miteinander (und mit dem Benutzer) kommunizieren.

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 Programme 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 Assemblersprache stellt eine für Menschen lesbare Form des Maschinencodes dar. Mit dem Assembler- bzw. Maschinenbefehl

mov eax, 4

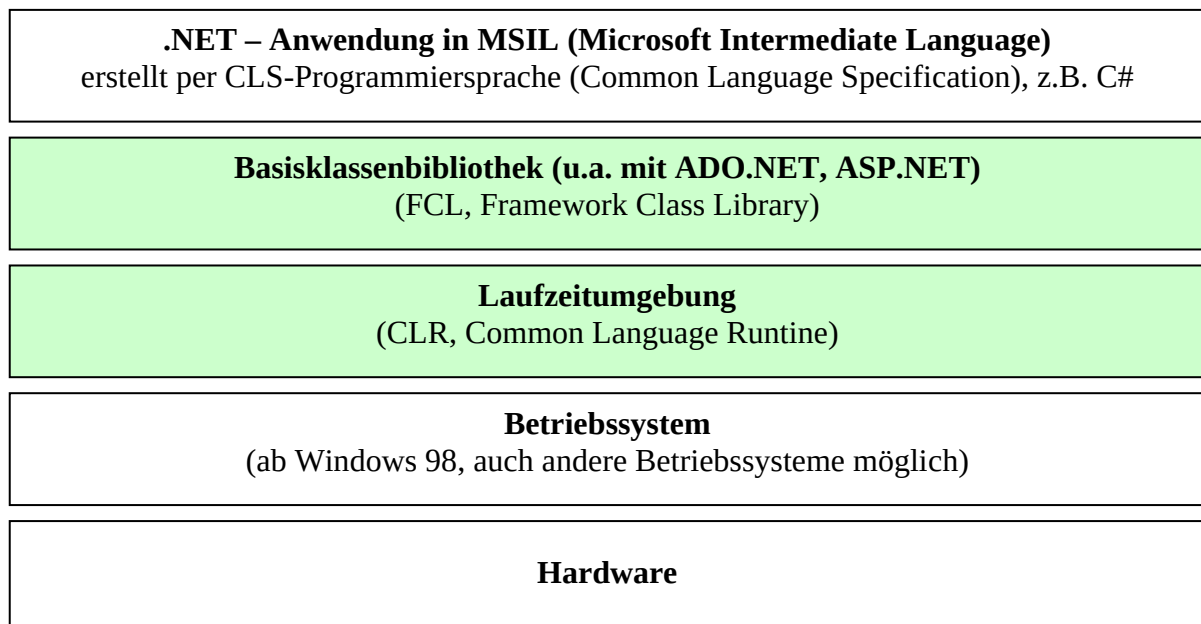
wird z.B. der Wert 4 in das EAX-Register (ein Speicherort im Prozessor) geschrieben.

Die CPU holt sich einen Maschinenbefehl nach dem anderen aus dem Hauptspeicher und führt ihn aus, heutzutage immerhin mehrere Milliarden Befehle pro Sekunde.

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 Hintergrundinformationen, die für jeden wichtig sein dürften, der Programme für das .NET – Framework entwickelt.

Beim .NET – Framework handelt es sich um eine von Microsoft entwickelte Plattform (Sammlung von Technologien), welche die bisherige Windowsprogrammierung 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:



Ab der Windows-Version Vista ist das .NET – Framework fester Systembestandteil. Bei älteren Versionen (ab Windows 98) kann es über das bei Microsoft kostenlos verfügbare **Redistributable-** Paket nachinstalliert werden, das auch mit eigenen .NET - Anwendungen vertrieben werden darf.

Vom .NET –Framework sind bisher folgende Versionen erschienen:

Version	Erscheinungsjahr
1.0	2002
1.1	2003
2.0	2005

Im Kurs verwenden wir die Version 2.0.

1.2.1 Installation

Das zum Ausführen von .NET - Programmen erforderliche Framework (mit der CLR, der Klassenbibliothek und den Compilern für C#, VB.NET etc.) in der aktuellen Version 2.0 ist in Windows Vista enthalten und kann für andere Windows-Versionen über das

Microsoft .NET Framework 2.0 Redistributable (x86)

nachinstalliert werden. Neben der für uns ausschließlich 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).

Anforderungen der x86-Version:

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

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

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

Es ist sinnvoll, anschließend das deutsche Sprachpaket zu installieren:

Microsoft .NET Framework 2.0 Language Pack Deutsch (x86)

Bei der Installation von Microsofts Visual C# 2005 Express Edition (siehe Abschnitt 2.3.1) landen Framework und Sprachpaket automatisch auf der Festplatte.

1.2.2 C#-Compiler und MSIL

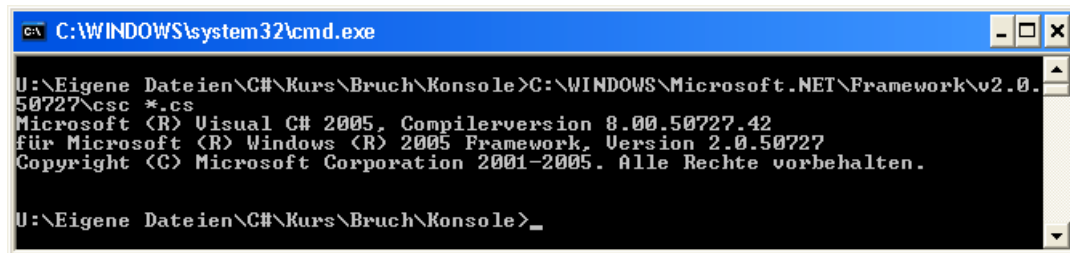
Den (z.B. mit einem beliebigen Editor verfassten) C# - Quellcode übersetzt der C# - **Compiler** im .NET - Framework in die **Microsoft Intermediate Language (MSIL)**, oft kurz als **IL** (*Intermediate Language*) bezeichnet. Wenngleich dieser Zwischencode von den heute üblichen Prozessoren noch nicht 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 Maschinensprache einer konkreten CPU geschieht *just-in-time* bei der Ausführung des Programms durch die CLR (siehe Abschnitt 1.2.4).

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



eines Konsolenfensters, dann kann ihre Übersetzung in IL-Code mit folgendem Compiler-Aufruf veranlasst werden:



Weil sich der Ordner

`C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\`

mit dem C# - Compiler per Voreinstellung nicht im Suchpfad für ausführbare Programme befindet, muss er im Aufruf angegeben werden. Der eigentliche Compiler-Auftrag, 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 Eigenschaft vom C# - Compiler erstellten **get-** und **set-**Methoden, z.B.:¹

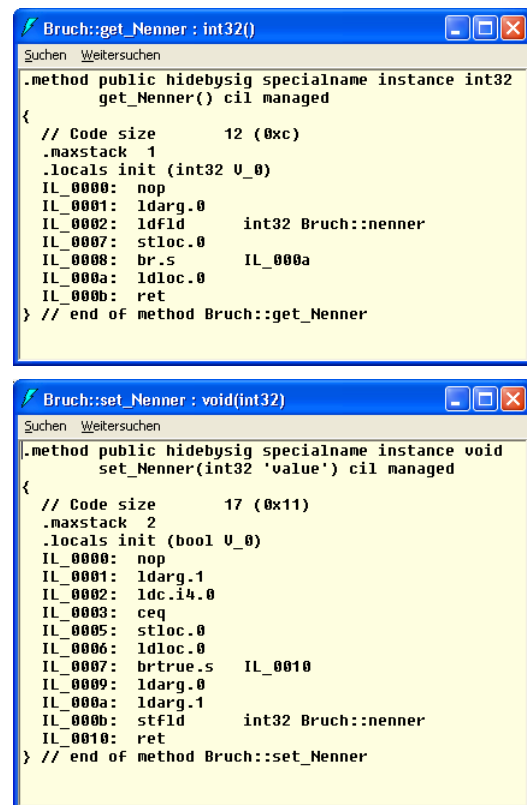
¹ Diese Ausgabe liefert das schon in Abschnitt 1.1.2 angesprochene Werkzeug **ILDasm** nach einem Doppelklick auf die interessierende Methode (siehe Seite 7).

```

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

```

C# - Compiler



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 Programmierer unter Verwendung verschiedener Sprachen Klassen für ein gemeinsames Programm erstellen. In der Java-Welt arbeiten alle Entwickler hingegen mit derselben Programmiersprache, was man als Vor- oder Nachteil betrachten kann.

Mittlerweile sind für viele Programmiersprachen MSIL-Compiler verfügbar, etliche werden sogar mit dem Redistributable-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 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 sicher gestellt.

1.2.3 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 Namensendung:

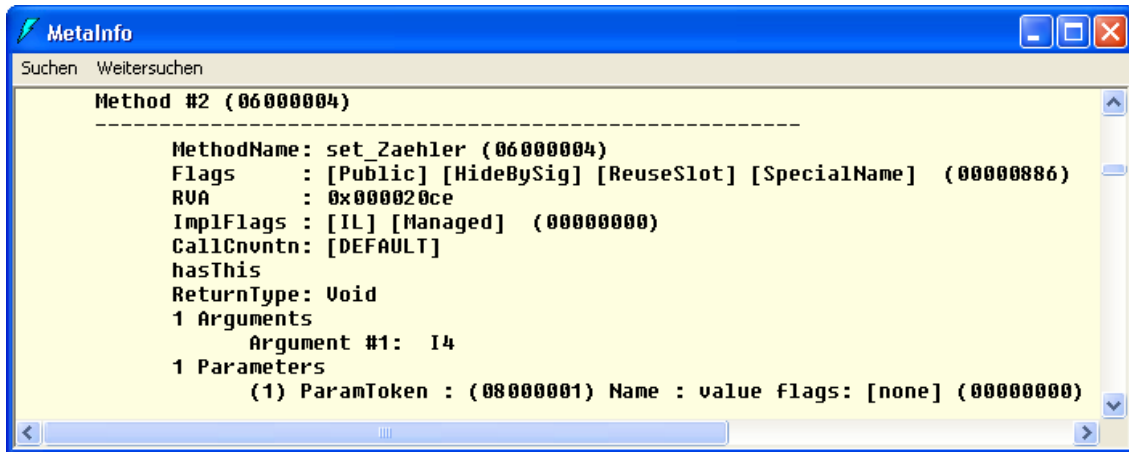
- **exe** (bei .NET – Anwendungen) oder
- **dll** (bei .NET – Bibliotheken)

Man übergibt dem Compiler im Allgemeinen mehrere Quellcodedateien mit jeweils einer Klassendefinition (siehe obiges Beispiel) und erhält *ein* Assembly. In der Konsolen-Variante unseres Bei-

spiels lassen wir vom Compiler ein Exe-Assembly mit dem Zwischencode der Klassen Bruch und BruchAddition erzeugen.

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

Im Bruchadditions-Assembly sind 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.

Unter den Metadaten eines Assemblies spielen die Informationen im so genannten **Manifest** eine herausgehobene Rolle. Hier finden sich u.a. die folgenden Informationen:

- 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.
- Unterstützte Sprache (Kultur), z.B.
`AssemblyCulture("de-ch")`
- Sicherheitsmerkmale bei signierten Assemblies (z.B. der öffentliche Schlüssel des Herausgebers)
- 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.

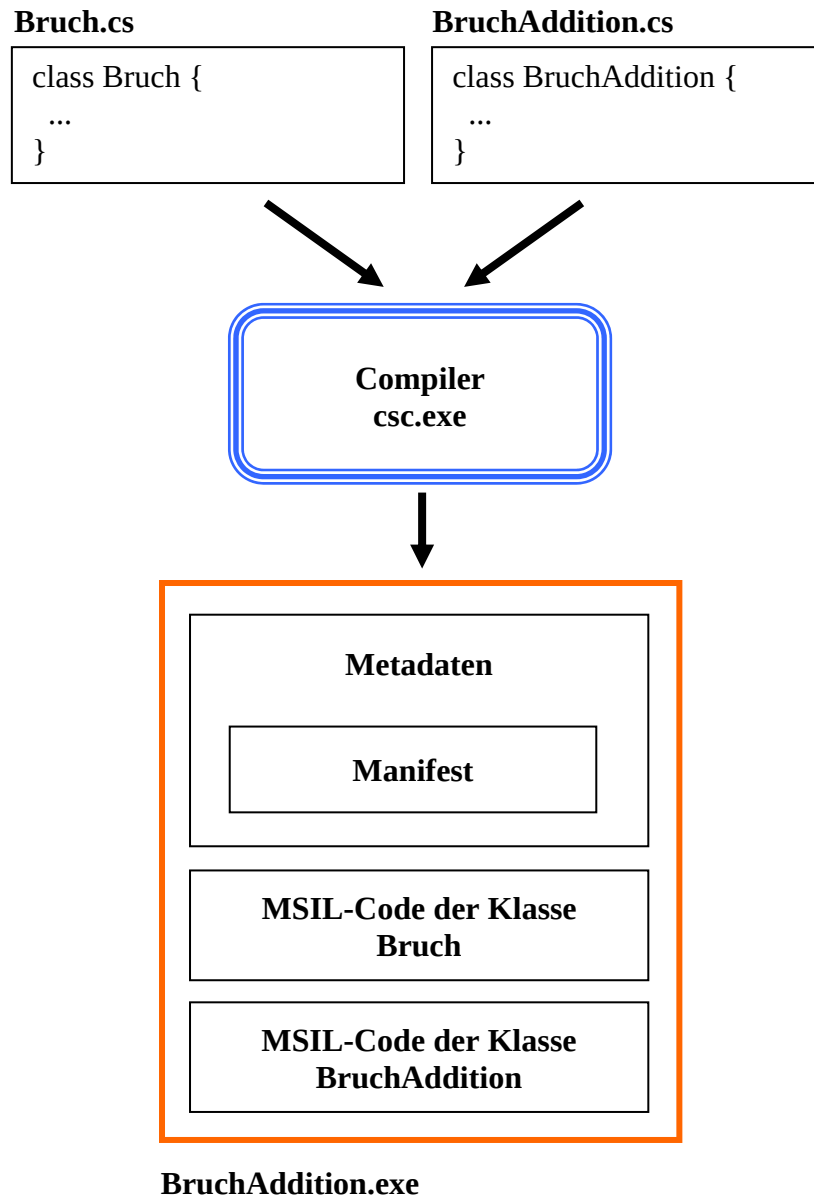
Metadaten sorgen dafür, dass die Klassen eines Assemblies unproblematisch von beliebigen .NET - Programmen genutzt werden können. Oben wurde schon erwähnt, dass mit dem .NET – Framework die ältere COM-Architektur (*Common Object Model*) abgelöst werden soll, die in den 90er Jahren des letzten Jahrhunderts 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.

¹ Metainformationen erhält man in **ILDasm** über die Tastenkombination **Strg+M**.

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).

In der folgenden Abbildung (nach Mössenböck 2003, S. 6) werden wesentliche Informationen über Quellcode, C#-Compiler, MSIL, Assemblies und Metadaten anhand des Bruchrechnungsbeispiels zusammen gefasst:



Um die Liste der Assembly-Dateitypen zu vervollständigen, wird bis zum Ende des aktuellen Abschnitts noch ein Aspekt behandelt, den Einsteiger gefahrlos ignorieren dürfen: 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**. Es besteht aus mehreren **Moduldateien** mit Zwischencode und zugehörigen Metadaten, wobei *eine* Moduldatei zusätzlich das Manifest des gesamten Assemblies enthält.

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 Namensweiterung **.netmodule** verwendet.

1.2.4 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. Beim Bruchrechnungsbeispiel ist dabei von einer Ver

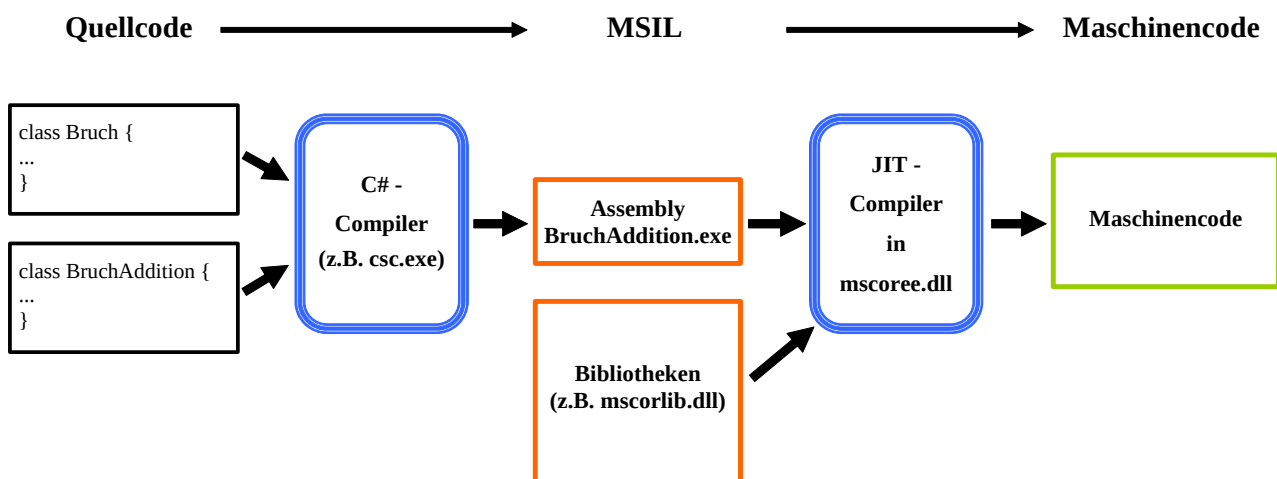
```

U:\Eigene Dateien\C#\Kurs\Bruch\Konsole\BruchAddition.exe
1. Bruch
Zaehler: 2
Nenner : 7
  2
 ----
  7

2. Bruch
Zaehler: 13
Nenner : 17
  13
 ----
  17

Summe
 125
 ----
 119
  
```

In der folgenden Abbildung ist der Weg vom Quellcode bis zum ausführbaren Maschinencode für das Bruchrechnungsbeispiel 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 weit- aus 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.

1.2.5 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 definieren meist einen eigenen Namensraum (**namespace**), 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.:

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


Um in einem Programm die Klassen eines Namensraums vereinfacht (ohne Namensraum-Präfix) ansprechen zu können, muss der Namensraum am Anfang des Quelltextes per **using**-Direktive¹ importiert werden, z.B.:

```
using System;

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

Durch diese **using**-Direktive wird dafür gesorgt, dass der Compiler jeden Klassennamen, der nicht im projekteigenen Quellcode definiert ist, den Präfix **System** voranstellt und dann die referenzierten Assemblies nach dem vervollständigten Namen durchsucht.

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 einen 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.

Einen ersten Eindruck vom Leistungsvermögen der .NET – Standardbibliothek vermittelt die folgende Auswahl ihrer Namensräume, die alle unter dem Einstieg **System** angesiedelt sind. Diese Auflistung ist für Programmier-Einsteiger allerdings von begrenztem Wert und sollte bei diesem Leserkreis keine Verunsicherung durch die große Anzahl fremder Begriffe auslösen:

Namensraum	Inhalt
System	... enthält grundlegende Basisklassen sowie Klassen für Dienstleistungen wie 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.

¹ Neben der **using**-Direktive werden wir noch die **using**-Anweisung kennen lernen. Vorläufig spielt die Unterscheidung zwischen einer Anweisung und einer Direktive für uns keine Rolle.

Namensraum	Inhalt
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.

Im kostenlosen .NET – SDK (*Software Development Kit*, siehe Abschnitt 2) ist ein mächtiges, komfortables und zudem kostenloses Werkzeug zum Explorieren der Standardbibliothek enthalten.

1.2.6 Zusammenfassung

Als Vorteile der .NET – Technologie für die Softwareentwicklung sind u.a. zu nennen:

- **Sprachintegration**
Es ist z.B. möglich, in einem VB.NET – Programm eine mit C# erstellte Klasse zu benutzen.
- **potentielle Betriebssystem-Unabhängigkeit**
Es ist möglich, die .NET-Plattform auf andere Betriebssysteme zu portieren. Im **Mono-Projekt** der Linux-Szene sind viel versprechende Entwicklungen 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 1.2.2 u.a. 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 für von .NET – Software für die ...:

- Weitergabe,
- Versionierung,
- 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 zur kompletten Beschreibung der implementierten und der referenzierten Typen.

- **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 (namespace)**

Indem man eine Klasse in einen Namensraum einfügt, ergänzt man ihren Namen um einen 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?

- Die **Main()**-Methode der Startklasse einer .NET – Anwendung muss als **static** deklariert werden.
- Die Datei mit dem Quellcode einer Klasse muss (abgesehen von der Extension) denselben Namen haben wie die Klasse.
- Unter den .NET – Programmiersprachen zeichnet sich C# durch eine besonders leistungsfähige Standardbibliothek aus.
- 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 CLR?

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

5) Welche Beziehung besteht zwischen Assemblies und Namensräumen?

6) Was bedeuten die Abkürzungen CLS, IL, COM und FCL?

7) Gönnen Sie sich ein wenig Unterhaltung bei .NET TV:

<http://www.microsoft.com/germany/msdn/nettv/default.msp>

Speziell die Folge 5 über CAS (Code Access Security) könnte interessant sein.

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 im Kurs gleich mit drei Produkten in Kontakt kommen:

- **Microsoft Visual Studio 2005 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 (z.B. Trolltech, MySQL).
- **Microsoft Visual C# 2005 Express Edition**
Dieser kostenlose Visual Studio - Ableger ist vermutlich auf Ihrem Privat-PC eine sehr gute Wahl.
- **SharpDevelop 2.1**
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 Datenbankbindung

Mit Ausnahme der Entwicklungsumgebung SharpDevelop stammt die im Kurs verwendete Software von der Firma Microsoft und kann (abgesehen vom Visual Studio) 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 Microsoft-URLs wird anschließend verzichtet, weil sie potentiell variabel 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, könnte 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 aufgefordert, ihre **Main()**-Methode auszuführen. Trotz Klassendefinition haben wir es praktisch mit einer Prozedur historischer Bauart zu tun, was für den einfachen Zweck des Programms durchaus angemessen ist. In den Abschnitten 2 und 3 werden wir solche pseudo-objektorientierten (POO-) Programme benutzen, um elementare Sprachelemente in möglichst einfacher Umgebung kennen zu lernen. Aus den letzten Ausführungen ergibt sich, dass C# zwar eine objektorientierte Programmierweise nahe legen und unterstützen, aber nicht erzwingen kann.

Das Hallo-Programm eignet sich aufgrund seiner Kürze zum Erläutern wichtiger Regeln, an die Sie sich so langsam gewöhnen müssen. Alle Themen werden aber später noch einmal systematischer und ausführlicher behandelt:

- In der ersten Zeile wird der Namensraum **System** importiert, damit die dort angesiedelte Klasse **Console** im Programm ohne Namensraum-Präfix angesprochen werden kann (vgl. Abschnitt 1.2.5). 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 Bezeichnern zu beachten, dass C# 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.:

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

- 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.

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 8).

- 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 einem Methodennamen muss auf jeden Fall eine Parameterliste angegeben werden, gegebenenfalls (wie im Hallo-Beispiel) eben eine leere. Wir werden uns später ausführlich mit Methoden-Parametern 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. Bei einem Methodenaufruf handelt sich um eine Anweisung, die folglich mit einem Semikolon abzuschließen ist.

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

Speichern 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.

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\v2.0.50727\csc Hallo.cs
```

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.

Auf Dauer ist ein derart umständliches Kommando nicht sinnvoll. Wer längerfristig mit Notepad programmieren möchte, wird den Framework-Ordner in die Definition der Umgebungsvariablen PATH aufnehmen.

Beim Einsatz einer integrierten Entwicklungsumgebung (siehe Abschnitt 2.3) wird derselbe Compiler verwendet und dabei automatisch im Hintergrund 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:bak.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 1.2.2)

Wir werden uns im Kurs voraussichtlich auf die Ausgabetypen **exe** und **winexe** beschränken.

Über die Befehlszeilenoption **reference** werden dem Compiler die Assemblies bekannt gemacht, welche die im eigenen Quellcode verwendeten (referenzierten) Klassen implementieren, z.B.:

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

Das zentrale Bibliotheks-Assembly

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

wird aber auf jeden Fall 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 2.0 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.Data.dll
/r:System.Data.OracleClient.dll
/r:System.Data.SqlXml.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.Formatters.Soap.dll
/r:System.Security.dll
/r:System.ServiceProcess.dll
/r:System.Transactions.dll
/r:System.Web.dll
/r:System.Web.Mobile.dll
/r:System.Web.RegularExpressions.dll
/r:System.Web.Services.dll
/r:System.Windows.Forms.dll
/r:System.Xml.dll
```

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

Die integrierten Entwicklungsumgebungen (siehe Abschnitt 2.3) rufen den Compiler mit der Befehlszeilenoption

```
/noconfig
```

und verhindern damit die Auswertung der voreingestellten Antwortdatei. Folglich wird nur noch das zentrale Bibliotheks-Assembly **mscorlib.dll** automatisch durchsucht.

Über weitere Befehlszeilenoptionen informiert der Compiler beim folgenden Aufruf

```
csc /?
```

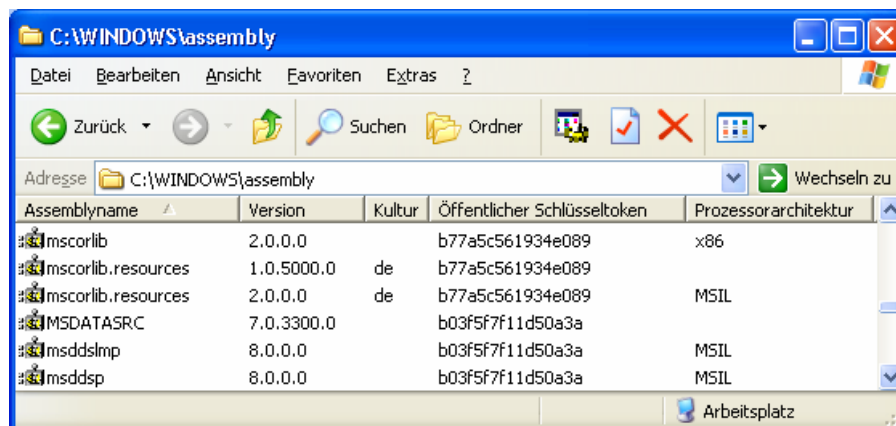
2.1.3 Ausführen

.NET – Programme können auf jedem Windows-Rechner mit passendem Framework auf übliche Weise gestartet werden, z.B. per Doppelklick. Das Hallo-Programm startet man am besten im Konsolenfenster durch Abschieken 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.

Benötigt ein Programm DLL-Assemblies, dann müssen diese entweder im Ordner des Programms oder im **Global Assembly Cache (GAC)** vorhanden sein, der sich unterhalb des Windows-Systemordners befindet, z.B.:



Eine nähere Beschäftigung mit dem GAC ist im ersten C# - Lernjahr nicht unbedingt erforderlich.

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, gerät z.B. in eine Endlosschleife oder stürzt ab.

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 Syntax-Fehlers 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:

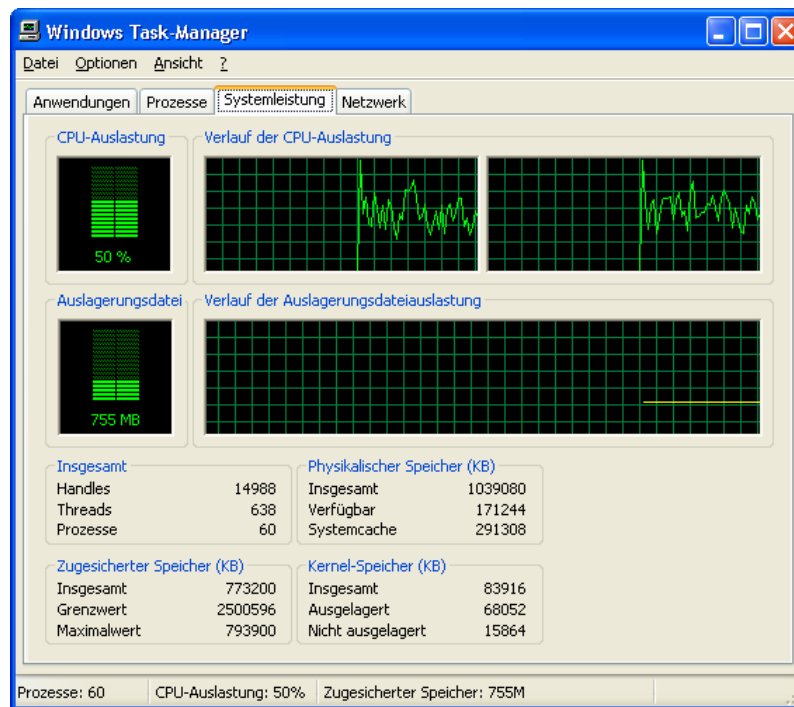
```
Hallo.cs(4,9): error CS0103: Der Name console ist im aktuellen Kontext nicht vorhanden.
```

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 Bruchrechnungsbeispiel 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 das Ungleich-Operatorzeichen `!=` 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 Hyper-Threading-CPU) belegt:



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

The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The user has run a program named 'BruchAddition'. The output shows the program's state before it crashed:

```
U:\Eigene Dateien\C#\Kurs\Bruch\Konsole>BruchAddition
1. Bruch
Zaehler: 1
Nenner : 0
^C
U:\Eigene Dateien\C#\Kurs\Bruch\Konsole>
```

The '^C' indicates that the user pressed Ctrl+C to interrupt the program.

2.2 .NET – SDK

Für Entwickler bietet Microsoft das

Microsoft .NET Framework SDK (Software Development Kit) 2.0 (x86)

kostenlos an, das

- eine ausführliche Dokumentation der Standardklassenbibliothek (FCL),
- viele Hilfsprogramme (z.B. **ildasm.exe** zur Analyse von Assemblies) sowie
- Beispiele

enthält. Hier finden Sie zahlreiche Informationen zu .NET – Details, die in der vorliegenden C# - Einführung nicht zur Sprache kommen.

Anforderungen der x86-Version:

- Windows 2000 (mit SP 3), Windows XP (mit SP 2), Windows Server 2003
- .NET Framework 2.0 (z.B. per Redistributable installiert)
Die Sprachversion des .NET Framework 2.0 Language Packs muss der installierten SDK-Sprachversion entsprechen.
- ca. 1,2 GB Festplattenspeicher (1200 MB!)

Bei der Installation von Microsofts Visual C# 2005 Express Edition (siehe Abschnitt 2.3.1) wird das SDK als Option angeboten, so dass eine separate Installation überflüssig und aufgrund des großen Platzbedarfs nicht sinnvoll ist.

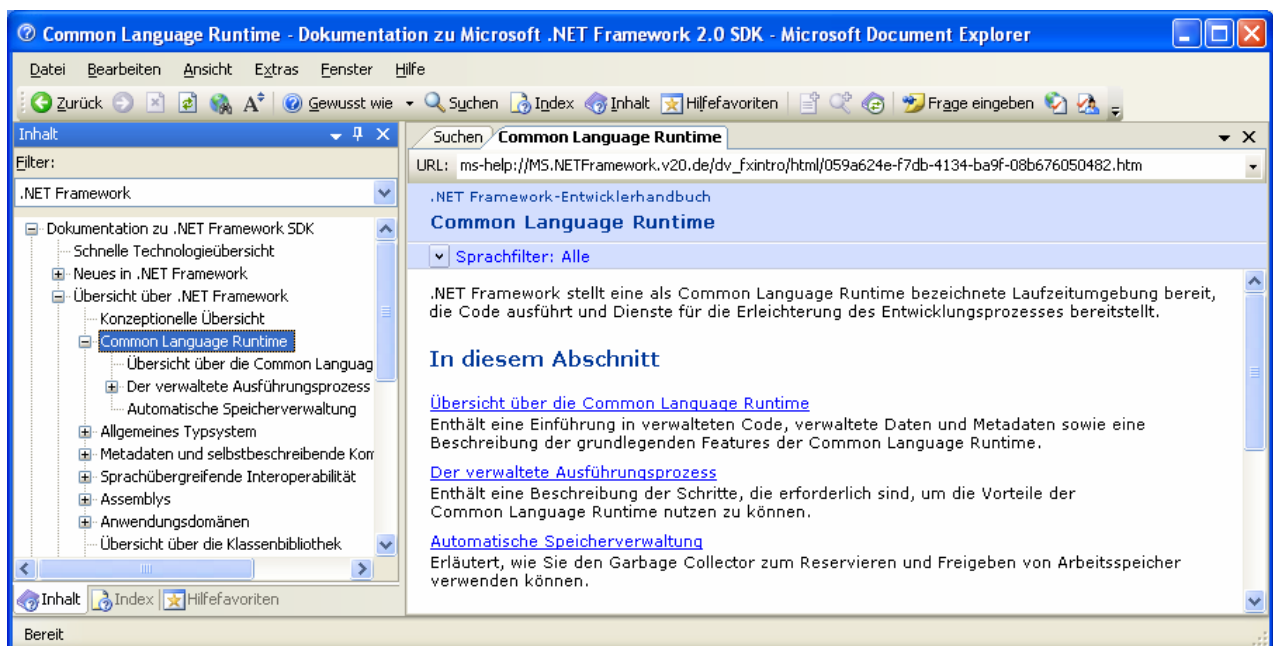
Über die Verknüpfung

Dokumentation

in der Programmgruppe

Microsoft .NET Framework SDK v2.0

startet man den **Document Explorer**, der unzählige Einführungstexte, Beispiele und eine detaillierte Referenz zur Standardbibliothek bietet:



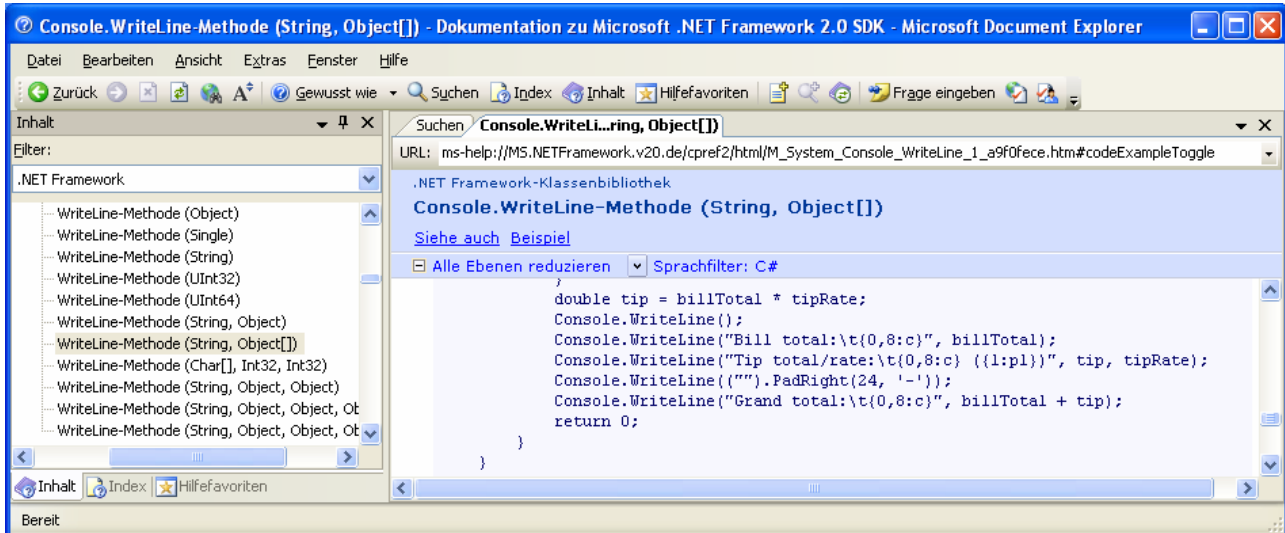
Auf dem **Inhalts**-Registerblatt der Navigationszone im linken Teil des Fensters wählt man die nahezu unverzichtbaren Referenzinformationen zur .NET – Standardbibliothek über:

Dokumentation zu .NET Framework SDK > Klassenbibliothek

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 *Klassenbibliothek* weiter mit den Stationen:

System > Console-Klasse > Console Methoden > WriteLine-Methode

Mit etwas Wissen über das (bislang noch nicht behandelte) Common Type System (CTS) findet man zu der passenden Spezialisierung (Überladung) detaillierte Informationen samt Beispiel:



Auf der Suche nach speziellen Informationen wird man oft den Einstieg per Index- oder Freitextsuche wählen.

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

2.3 Integrierte Entwicklungsumgebungen

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.

2.3.1 Microsoft Visual Studio 2005 Professional

2.3.1.1 Terminal-Server – Umgebung

Das Visual Studio 2005 Professional steht an der Universität Trier auf einem Terminalserver-Cluster zur Verfügung, wobei die Verbindung auf den Pool-PCs folgendermaßen hergestellt wird:

**Start > Alle Programme > Programmentwicklung >
Development Server > Anmelden**

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

Im voreingestellten Vollbildmodus der Terminalserver-Sitzung 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 Terminal-Server startet das Visual Studio automatisch. Beim Beenden der Entwicklungsumgebung wird die Terminalserver-Sitzung automatisch geschlossen.

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

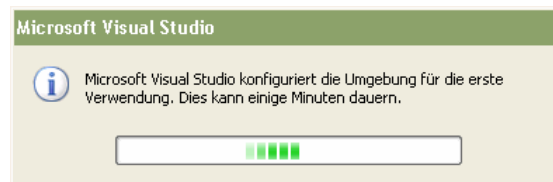
- Am oberen Bildschirmrand befindet sich ein gelbes Bedienelement. Eventuell müssen Sie den Bildschirmrand mit der Maus berühren, damit das Bedienelement erscheint.
- Hier finden sich die gewohnten Symbole, um den Vollbildmodus zu verlassen oder die Terminal-Sitzung die Taskleiste (der lokalen Windows-Sitzung)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

2.3.1.2 Konfiguration beim ersten Start

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



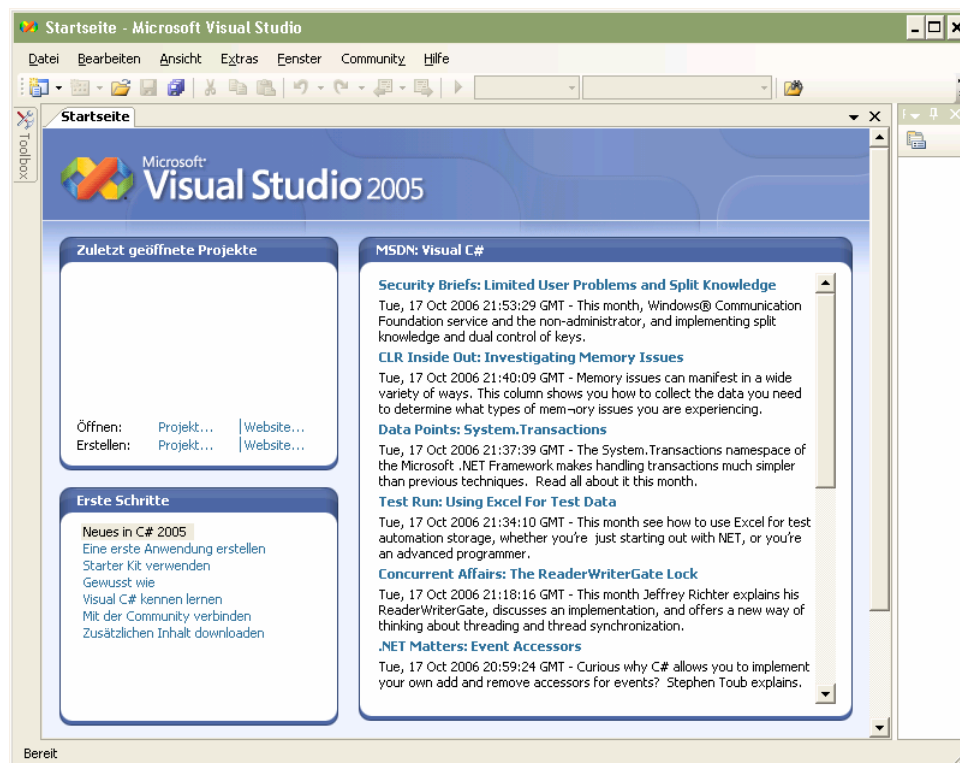
Nach einem Mausklick auf **Visual Studio starten** legt das Visual Studio etliche Ordner und Registry-Keys an:¹



Beim ersten Auftritt unserer Entwicklungsumgebung dominiert die Startseite:

¹ Es entstehen u.a.

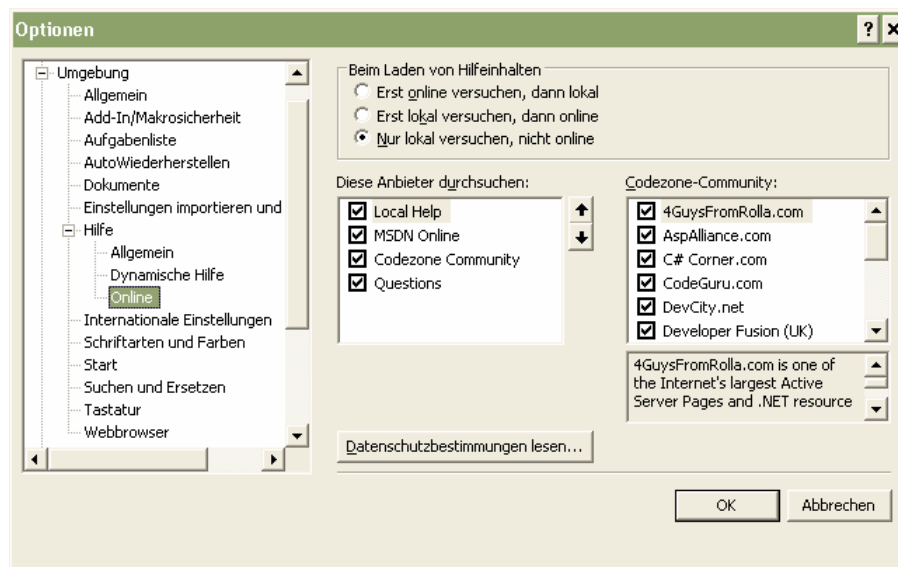
- die Ordner
 - o U:\NTProfil.fix\Anwendungsdaten\Microsoft\VisualStudio
 - o U:\NTProfil.fix\Anwendungsdaten\Microsoft\MSDN
- die Registry-Keys
 - o HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\8.0
 - o HKEY_CURRENT_USER\Software\Microsoft\MSDN\8.0



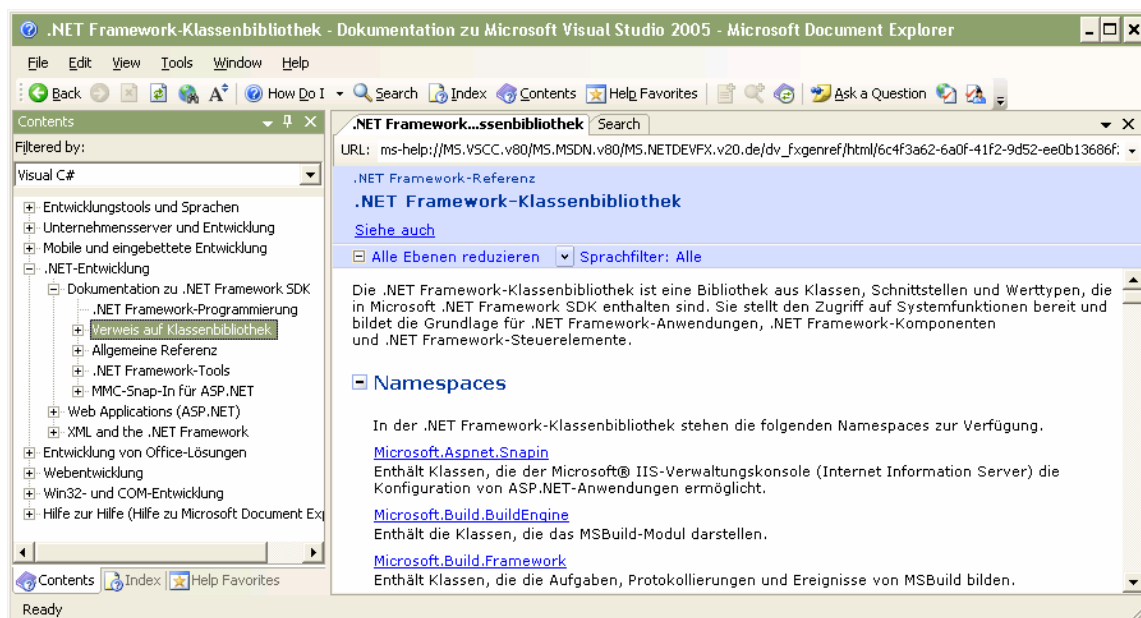
Wenn Sie auf eines der Angebote im Rahmen **Erste Schritte** klicken, kommt es zum Erstkontakt mit dem Hilfesystem und Sie dürfen sich für eine bevorzugte Informationsquelle entscheiden. Hier wird zugunsten der Reaktionsgeschwindigkeit auf Internet-Aktualität verzichtet:



Sie können Ihre Entscheidung später über
Extras > Optionen > Hilfe > Online
 korrigieren:



Das Anzeigen der Information übernimmt der Microsoft Document Explorer, den wir schon als SDK-Bestandteil kennen gelernt haben (siehe Abschnitt 2.2):



Im Visual Studio 2005 Professional bietet er aber mehr Inhalte an.

2.3.1.3 Eine erste GUI-Anwendung

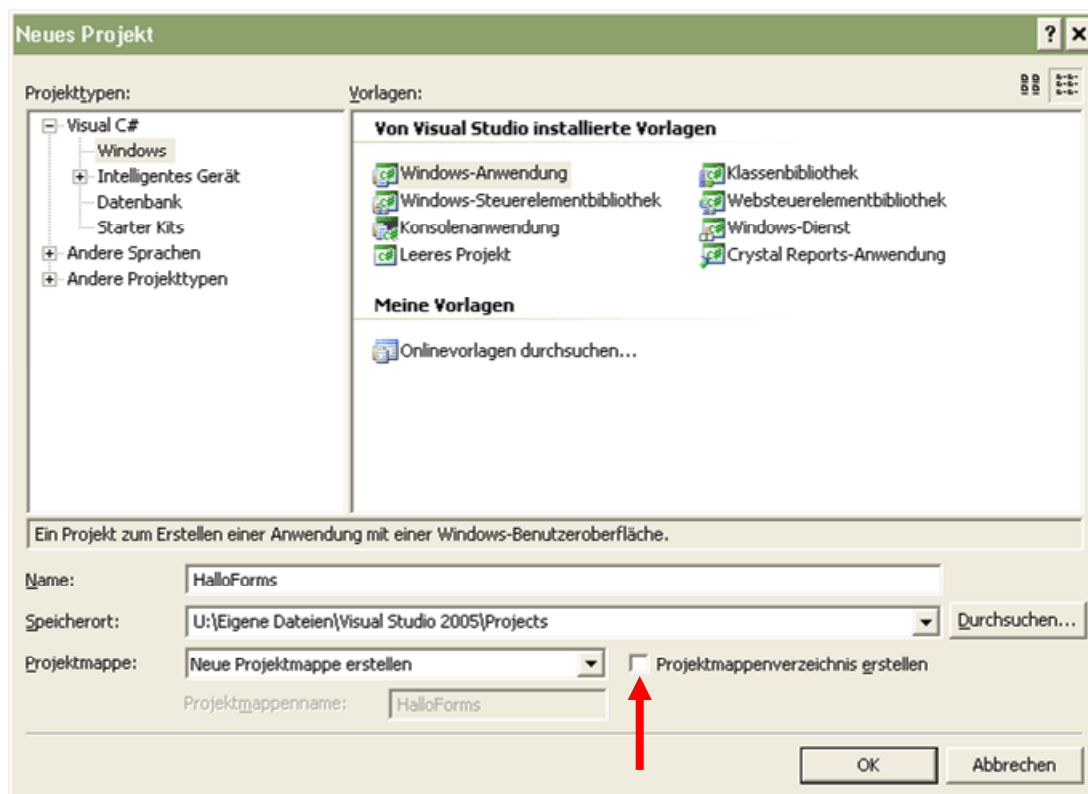
Wir folgen den Empfehlungen der Hilfeseite zum Thema

Erste Schritte > Eine erste Anwendung erstellen

und fordern über

Datei > Neu > Projekt

den folgenden Dialog an:

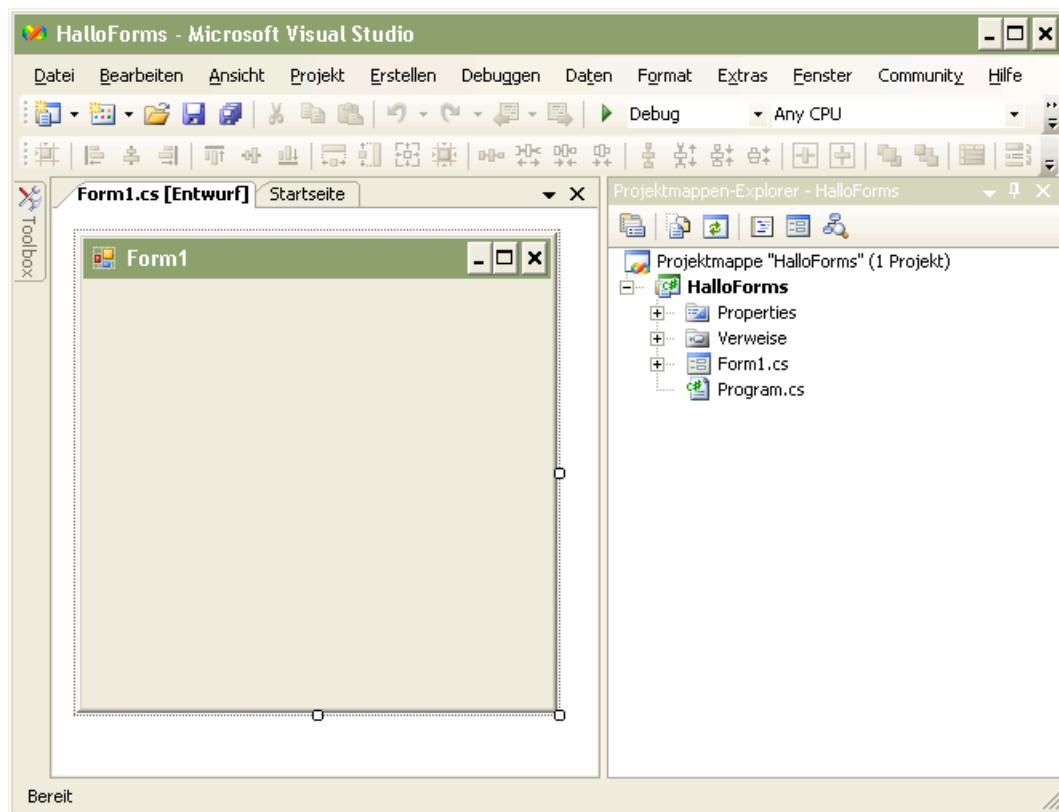


Hier akzeptieren wir die voreingestellte Projektvorlage **Windows-Anwendung** sowie den vorgeschlagenen Speicherort für das Projekt, das im Beispiel den Namen **HalloForms** erhält.¹

Allerdings wird aus dem Miniaturprojekt kaum eine zusammengehörige 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 Projektdatenverwaltung, die für praktisch alle im Kurs entstehenden Projekte angemessen ist.

Nach einem Mausklick auf **OK** präsentiert das Visual Studio den **Projektmappen-Explorer** und den Formulardesigner:

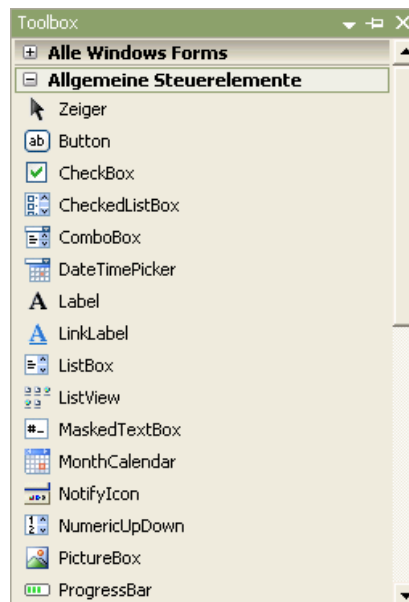
¹ Das Motiv für den Projektnamen wird Ihnen bald nach der Bekanntschaft mit den Klassen im Namensraum **System.Windows.Forms** plausibel erscheinen.



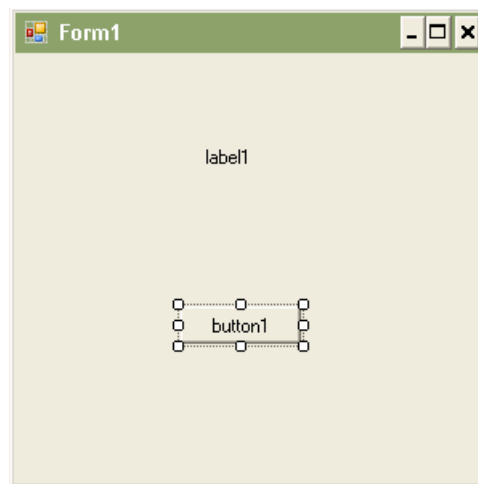
Öffnen Sie das **Toolbox**-Fenster mit

Ansicht > Toolbox

und erweitern Sie die **Allgemeinen Steuerelemente**:

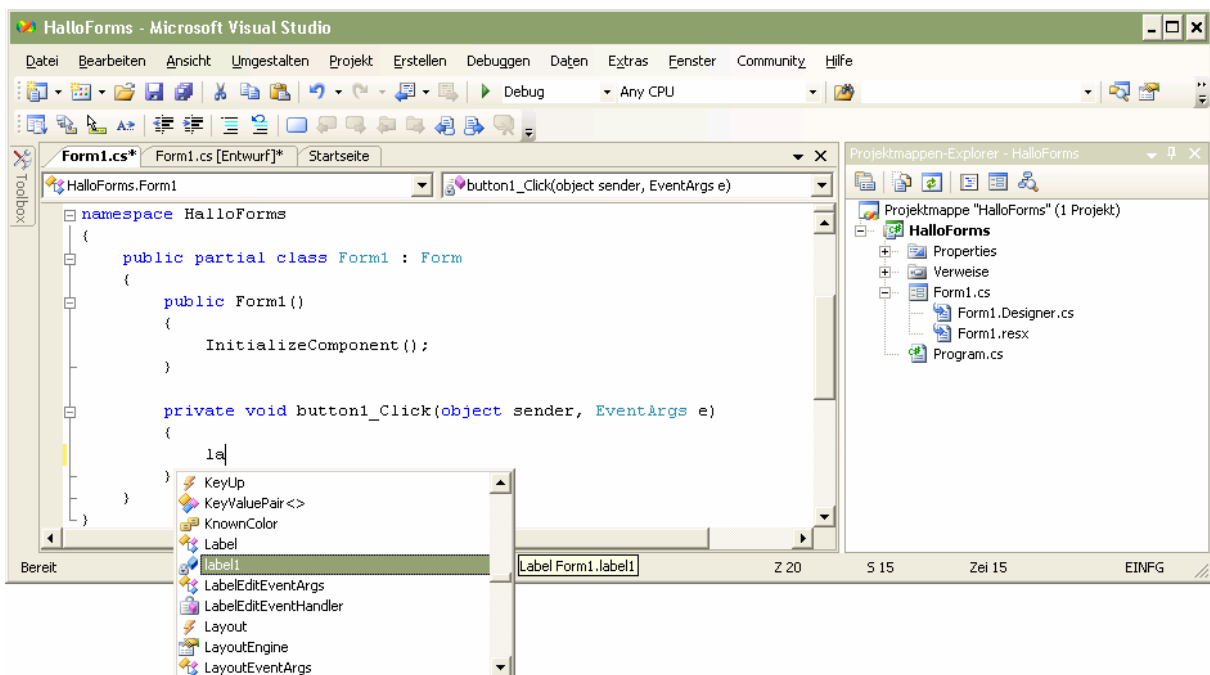


Ziehen Sie ein **Label**-Objekt (der Ausdruck ist durchaus im Sinn von Abschnitt 1.1 gemeint) und ein **Button**-Objekt auf das Formular, z.B.:



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.

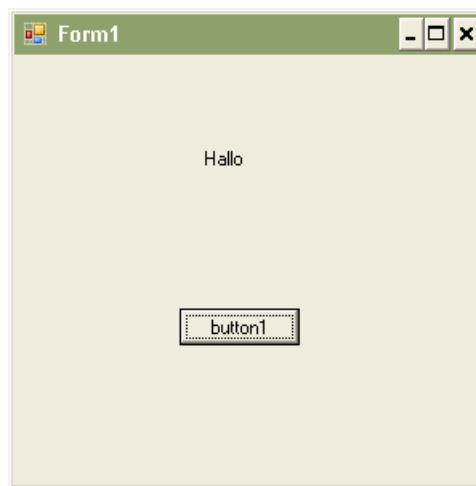
Sobald Sie damit beginnen, im Methodenrumpf eine Nachricht an das **Label**-Objekt mit der Bitte um **Text**-Änderung zu verfassen, errahnt das Visual Studio Ihre Absicht und bietet mögliche Fortsetzungen Ihrer Anweisung an:



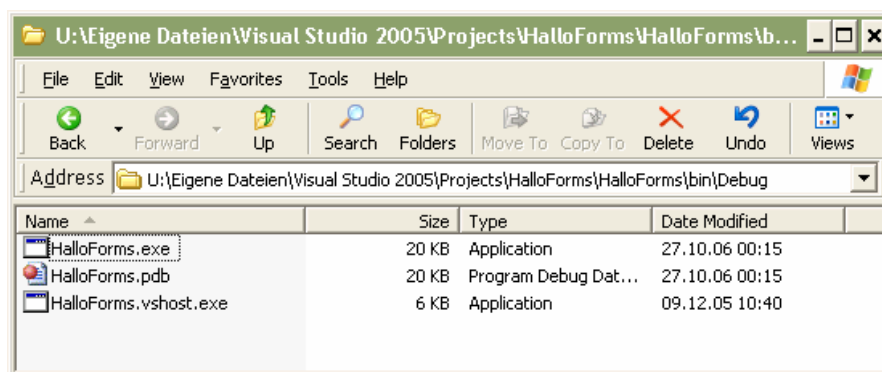
Akzeptieren Sie den Vorschlag per Tabulatortaste, und setzen Sie einen Punkt hinter den Objekt-namen. Nun erscheint eine Liste mit den Methoden und Eigenschaften des Objekts. Das **Label**-Objekt soll aufgefordert werden, seine **Text**-Eigenschaft auf den Wert *Hallo* zu setzen. Wählen Sie die Eigenschaft aus der Liste, und vervollständigen Sie die Anweisung:

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

Mit der Funktionstaste **F5** veranlassen Sie das Übersetzen und die Ausführung der Anwendung:



Das Assembly **HalloForms.exe** findet sich im folgenden Ordner:



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# 2005 Express Edition erstellt, deren Installation Thema des nächsten Abschnitts ist:

2.3.2 Microsoft Visual C# 2005 Express Edition

Die Microsoft Visual C# 2005 Express Edition ist ein kostenloser Ableger des Visual Studio 2005. Bei der Bedienung und bei der Projektverwaltung sind beide Versionen weitgehend kompatibel.

2.3.2.1 Installation

Voraussetzungen:

- Windows 2000 (mit SP 4), Windows XP (mit SP 2), Windows Server 2003
- CPU mit 600 MHz, 1 GHz empfohlen
- 192 MB RAM, 256 MB empfohlen
- Platzbedarf auf der Festplatte:

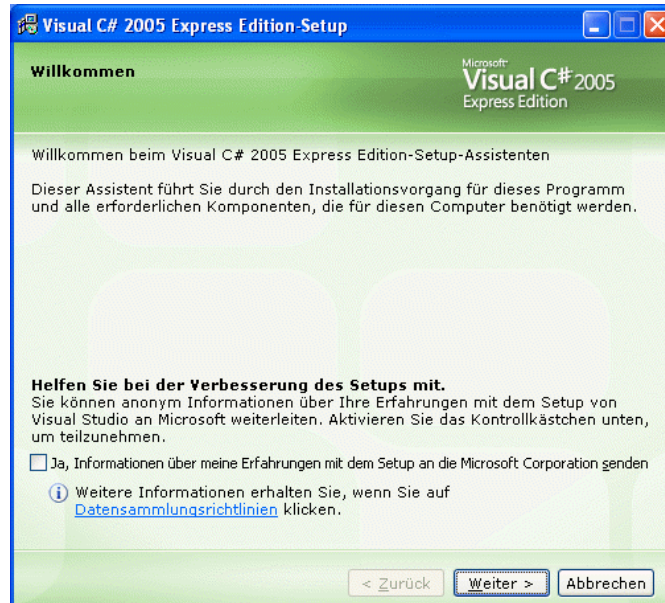
Produkt	Platzbedarf
Visual Studio 2005 Express Edition (ohne MSDN und SQL-Server)	ca. 210 MB
MSDN 2005 Express Edition	ca. 560 MB
SQL Server 2005 Express Edition x86	ca. 270 MB

Verfügbar auf der Webseite:

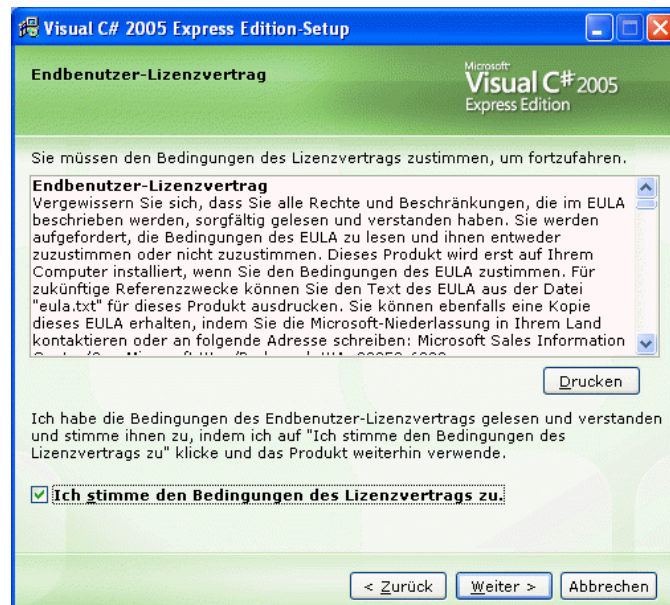
<http://www.microsoft.com/germany/msdn/vstudio/products/express/download.msp>

Die Installation der Microsoft Visual C# 2005 Express Edition verläuft einfach und zuverlässig:

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



- Stimmen Sie der Lizenzvereinbarung zu:



- Wählen Sie die gewünschten Zusatzoptionen:



Die **Microsoft MSDN 2005 Express Edition** enthält außerordentlich viele und nützliche Dokumentationen. Es handelt sich wohl im Wesentlichen um dieselbe Information und um dasselbe Anzeigeprogramm wie bei der SDK-Dokumentation (siehe Abschnitt 2.2). Wenn genügend Platz (siehe obige Tabelle) auf der Festplatte verfügbar ist, sollte die MSDN-Bibliothek installiert werden. Ist eine flotte Internetverbindung vorhanden, kann man aber bei geringer Komforteinschränkung auf die lokale MSDN-Installation verzichten und die Online-Variante nutzen (<http://msdn2.microsoft.com/en-us/library/default.aspx>).

Die **Microsoft SQL-Server 2005 Express Edition** benötigen wir bei der Entwicklung von Datenbank-basierter Software in Abschnitt 25.

Beide Zusatzkomponenten können später problemlos nachinstalliert werden.

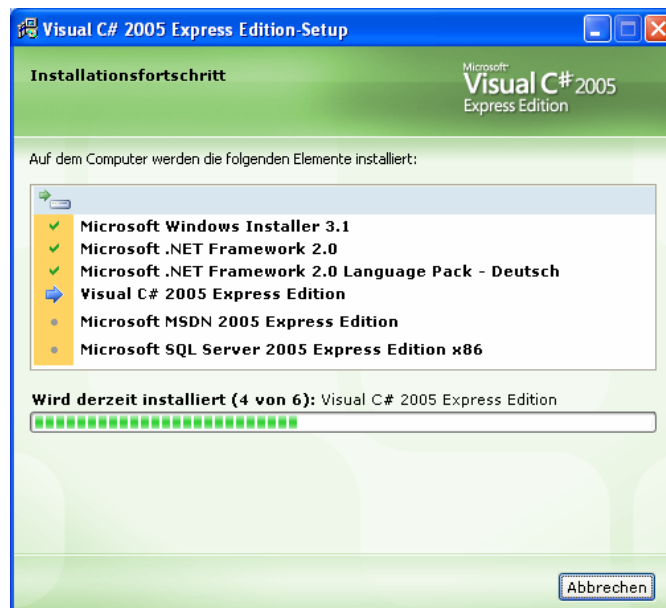
Wenn Sie die Microsoft Visual C# 2005 Express Edition samt MSDN-Bibliothek installieren, landet auch das .NET 2.0 SDK mit seinen Dokumentationen und Hilfsprogrammen auf der Platte.

Das .NET 2.0 Framework sowie das deutsche Sprachpaket werden bei Bedarf automatisch mit installiert.

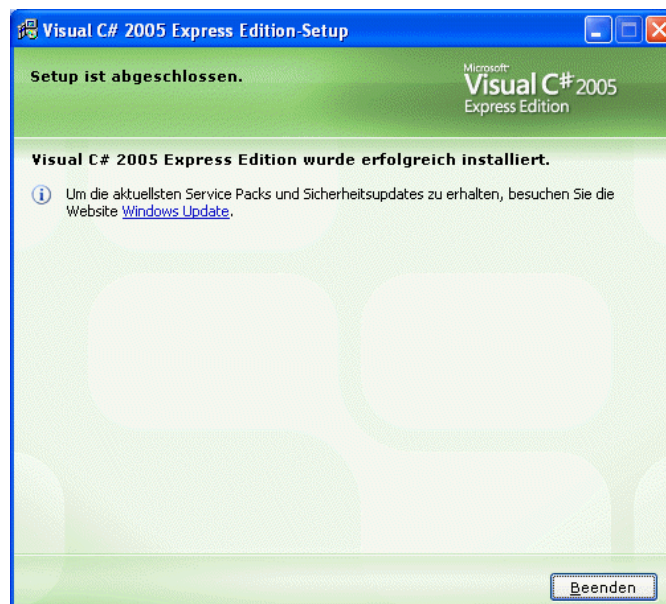
- Eventuell wollen Sie den Zielordner ändern:



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



- Fertig:



- Installieren Sie jetzt noch das **Service Pack 1** zur Microsoft Visual C# 2005 Express Edition, verfügbar über: <http://www.microsoft.com/downloads/details.aspx?FamilyID=7B0B0339-613A-46E6-AB4D-080D4D4A8C4E&displaylang=de>

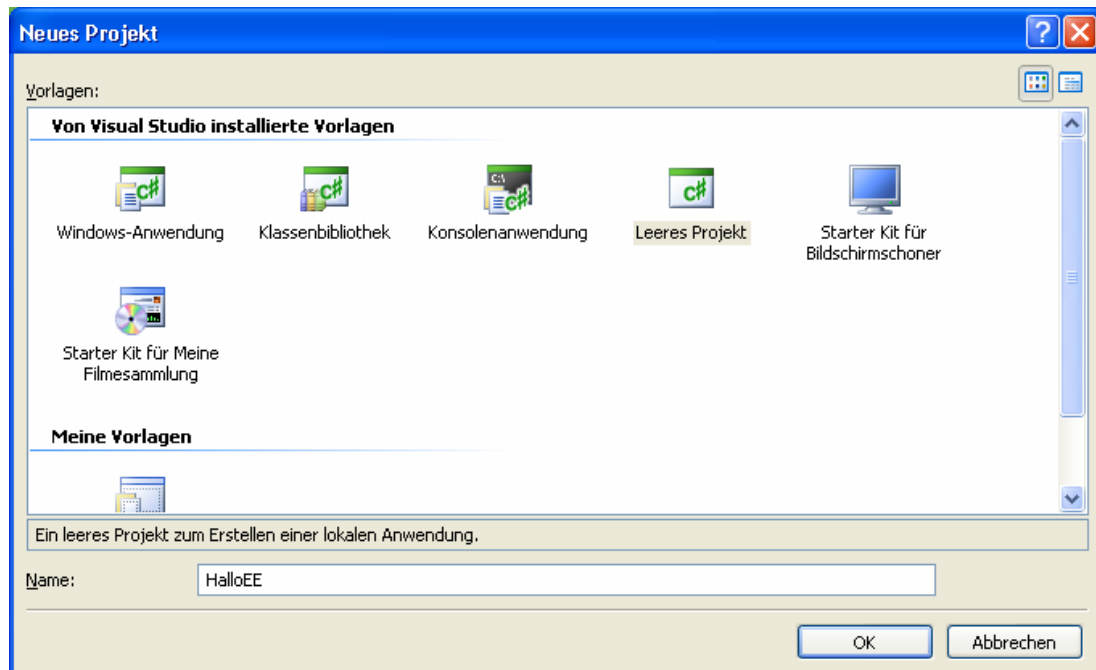
Bei den für uns relevanten Aufgaben gibt es keine wesentlichen Bedienungsunterschiede zwischen der Visual C# 2005 Express Edition und dem Visual Studio 2005 Professional (siehe Abschnitt 2.3.1).

2.3.2.2 Ein erstes Konsolen-Projekt

Starten Sie mit dem Schalter  oder dem Menübefehl

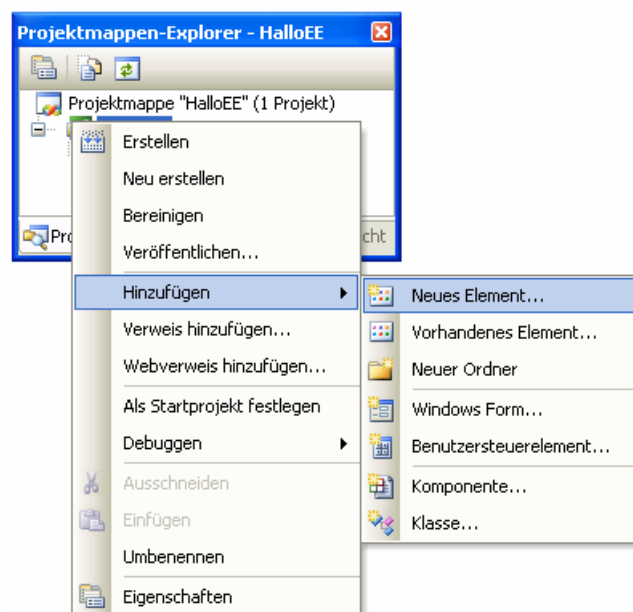
Datei > Neu > Projekt

den Dialog für neue Projekte:

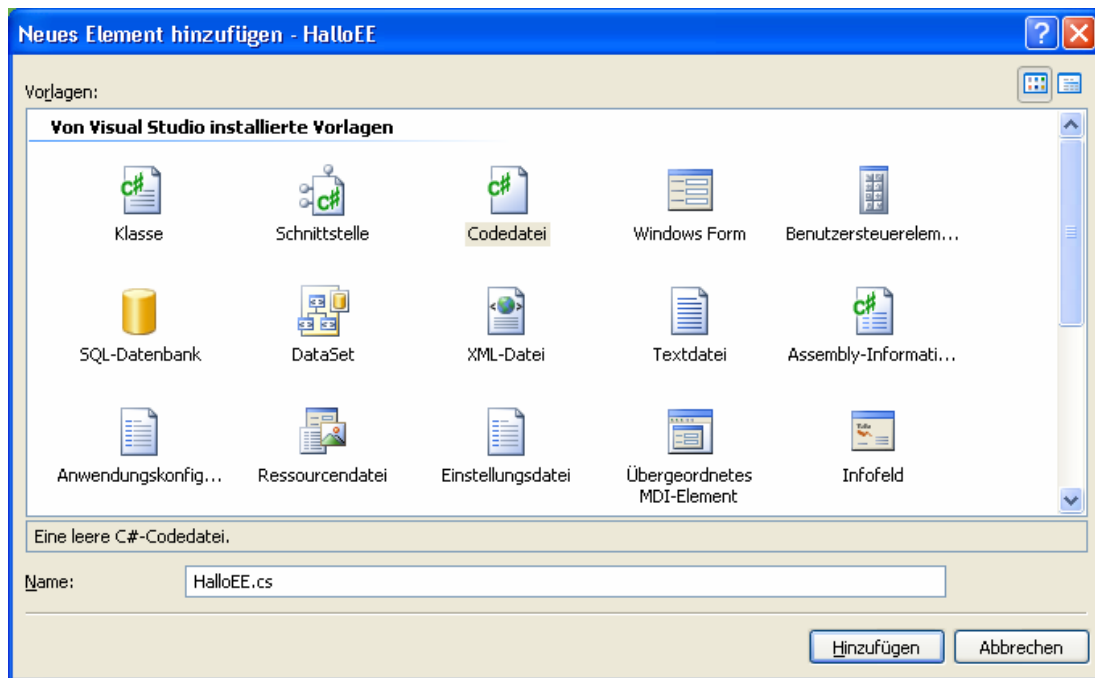


Wählen Sie die Vorlage **Leeres Projekt** und einen Projektnamen.

Öffnen Sie im Projektmappen-Explorer das Kontextmenü zum Projekt, und fügen Sie ein **neues Element** hinzu:

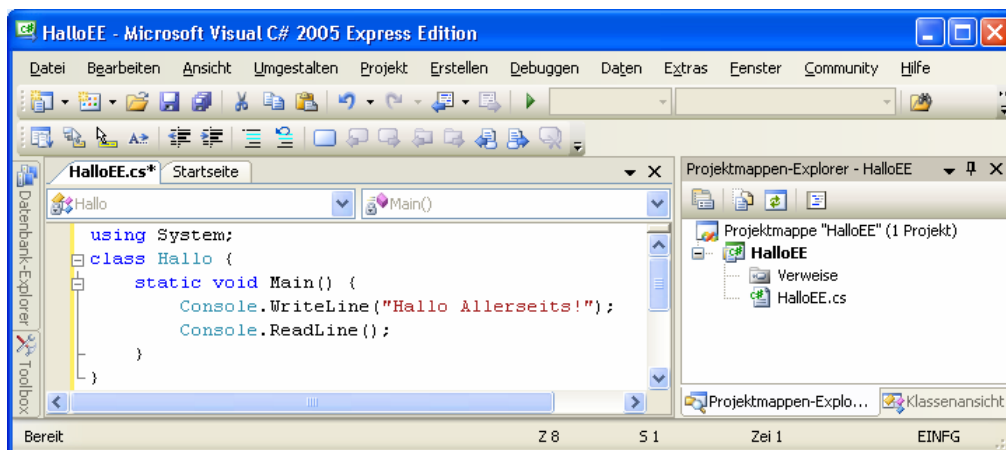


Entscheiden Sie sich für eine **Codedatei** mit geeignetem Namen:



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

Am Ende der **Main()**-Definition ergänzen wir diesmal einen **ReadLine()**-Methodenaufruf:

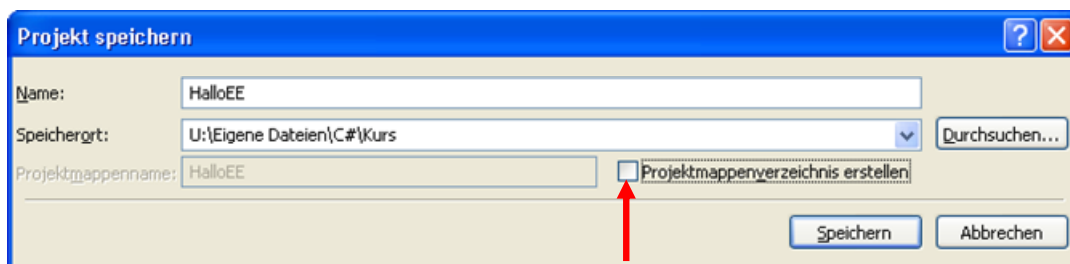


Er wartet auf die Enter-Taste und verhindert so, dass die im Rahmen der Entwicklungsumgebung ausgeführt mit **F5** Anwendung nach ihrer Bildschirmausgabe sofort verschwindet.

Fordern Sie über den Schalter  oder den Menübefehl

Datei > Alle Speichern

den folgenden Dialog zum Speichern des Projekts an:



Sicher wird aus dem Miniaturprojekt nie eine Familie von Projekten entstehen, die in einem gemeinsamen Ordner versammelt werden sollten. Entfernen Sie daher nötigenfalls die Markierung

beim Kontrollkästchen **Projektmappenverzeichnis erstellen**. Wir wählen damit eine flachere Projektdateiverwaltung, die für praktisch alle im Kurs entstehenden Projekte angemessen ist.

2.3.2.3 Referenzen und Ausgabtyp setzen

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

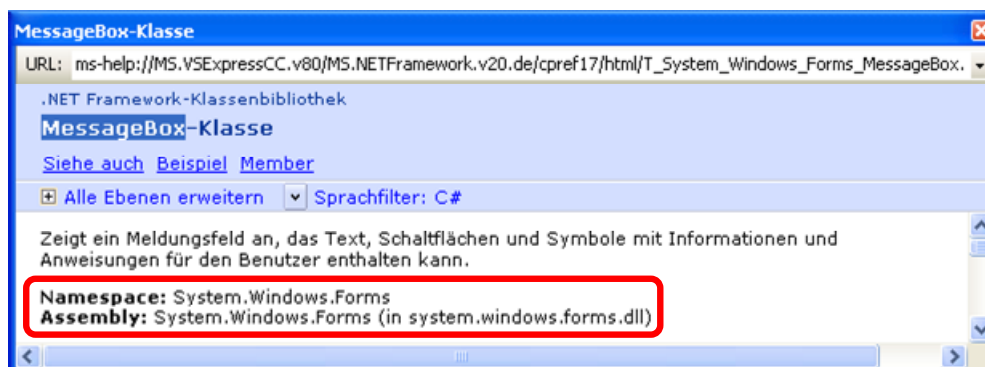
- Ausgabtyp des resultierenden Assemblies
- Referenzen auf Assemblies, die der Compiler nach Klassen durchsuchen soll

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 über zu können, erstellen wir nun ein „Hallo-Projekt“ gemäß Abschnitt 2.3.2.2, 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 schlägt das Übersetzen indes fehl, weil etliche Bezeichner unbekannt sind, insbesondere der Klassenname **MessageBox**. Mit der SDK-Dokumentation, die auch in der Hilfe der Visual C# 2005 Express Edition enthalten ist, ermittelt man leicht, welcher Namensraum anzugeben ist (als Präfix zum Klassennamen oder in einer **using**-Direktive):



Der funktionstüchtige Hallo-Quellcode mit MessageBox kann z.B. so aussehen:

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

Weil das die Klasse **MessageBox** implementierende Assembly **System.Windows.Forms.dll** in der voreingestellten Antwortdatei **csc.rsp**

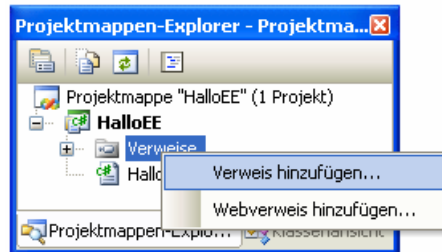
%SystemRoot%\Microsoft.NET\Framework\Version\csc.rsp

(vgl. Abschnitt 2.1.2) enthalten ist, wäre bei einem Compiler-Aufruf im Konsolenfenster keine **reference** – Option erforderlich. Die integrierten Entwicklungsumgebungen rufen den Compiler jedoch mit der Option

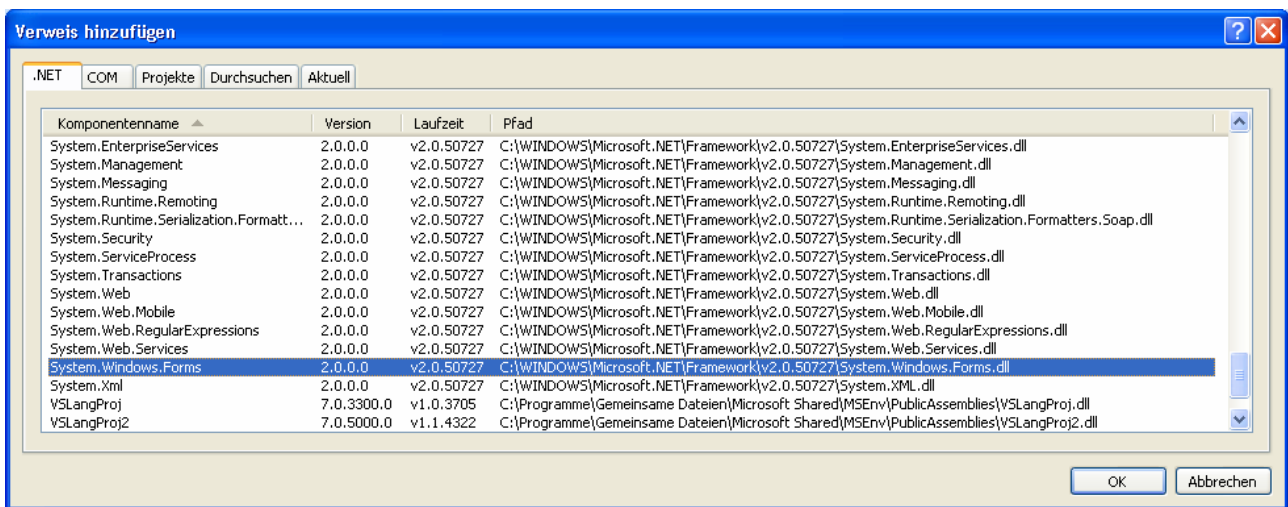
/noconfig

auf, welche die Auswertung der voreingestellten Antwortdatei verhindert, so dass nur noch das zentrale Bibliotheks-Assembly **mscorlib.dll** automatisch durchsucht wird.

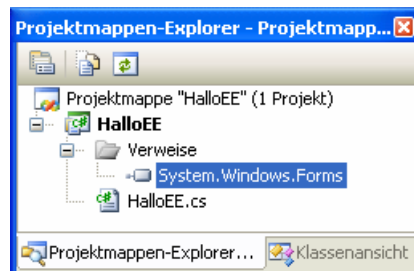
Folglich müssen wir eine Referenz auf das Assembly **System.Windows.Forms.dll** hinzufügen¹. Wählen Sie dazu im **Projektmappen-Explorer** aus dem Kontextmenü zum Projektnamen die Option **Verweis hinzufügen** (Visual C# 2005) bzw. **Referenz hinzufügen** (SharpDevelop 2.1):



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:



Beim Ausführen des erweiterten Programms erscheint zusätzlich zum gewohnten Konsolenfenster die MessageBox:

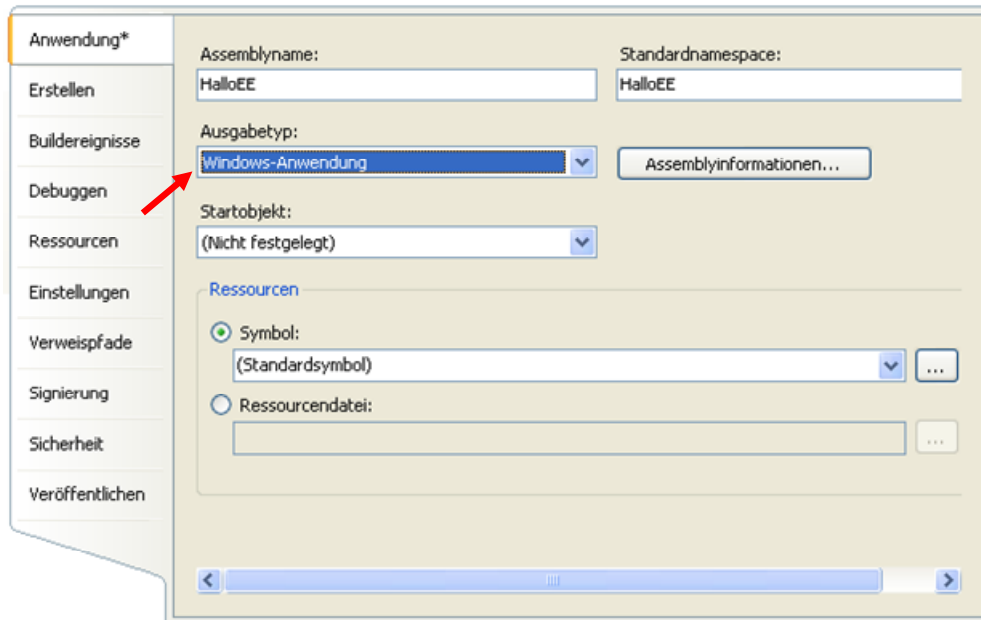


¹ Dies geschieht bei vielen Projektvorlagen der Entwicklungsumgebungen automatisch, nicht jedoch bei der Vorlage **Leeres Projekt**, die wir aus Gründen der Einfachheit und Transparenz derzeit bevorzugen.

Um den Auftritt eines überflüssigen Konsolenfensters zu verhindern, ersetzt man per Compiler-Option den voreingestellten Ausgabety **exe** durch die Alternative **winexe**. Bei einer integrierten Entwicklungsumgebung ändert man dazu eine Projektoption, z.B. bei der Visual C# Express Edition nach:

Projekt > Einstellungen > Anwendung

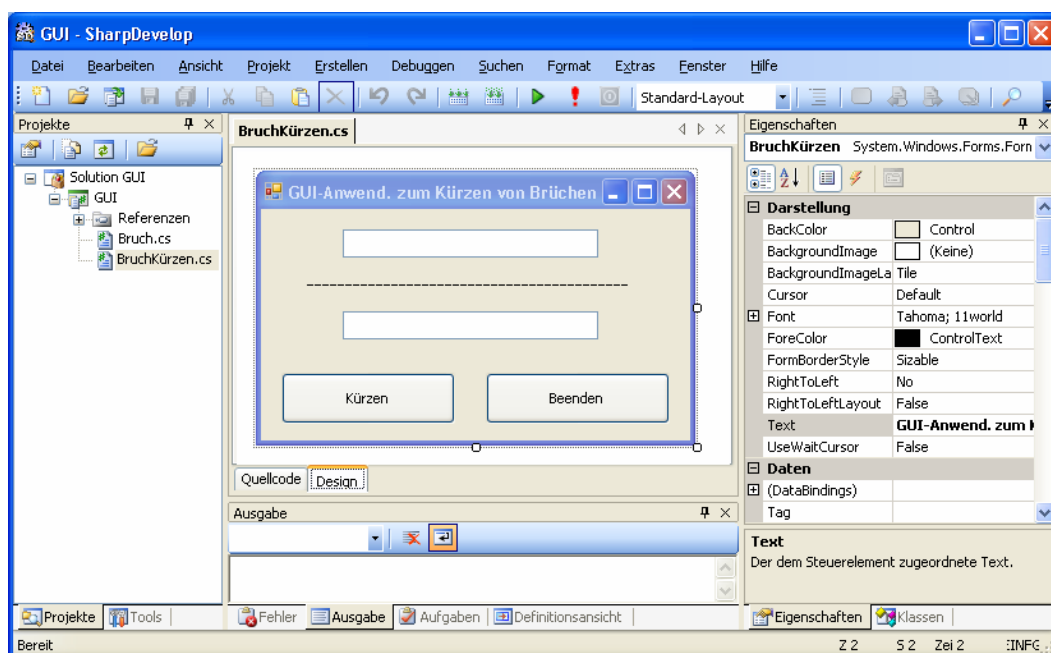
in folgendem Dialogfenster:



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

2.4 SharpDevelop 2.1

Das schlanke, unproblematisch zu bedienende und recht flink agierende Programm ist selbst in C# entwickelt worden:



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.

SharpDevelop setzt das .NET Framework voraus und nutzt im Hintergrund dessen Compiler. Das Programm läuft unter allen 32-Bit-MS-Betriebssystemen ab Windows 98.

Gehen Sie bei der Installation folgendermaßen vor:

- NET 2.0 Framework (Redistributable, siehe Abschnitt 1.2.1)
- Deutsches Sprachpaket für das .NET Framework 2.0 (siehe Abschnitt 1.2.1)
- NET 2.0 SDK (siehe Abschnitt 2.2)
Das SDK ist nicht obligatorisch für den Einsatz von SharpDevelop 2.1, allerdings sehr nützlich, weil es eine komplette FCL-Dokumentation, diverse Hilfsprogramme und viele Beispiele enthält. Mit einer flotten Internetverbindung kann man bei geringer Komforteinschränkung aber auch die Online-Variante der MSDN-Bibliothek nutzen (<http://msdn2.microsoft.com/en-us/library/default.aspx>).
- SharpDevelop 2.1

Weil sich die Projektverwaltung von SharpDevelop 2.1 stark am Visual Studio 2005 bzw. an der Visual C# 2005 Express Edition orientiert, sollte ein Wechsel zwischen den drei Entwicklungsumgebungen keine Schwierigkeiten bereiten.

2.5 Übungsaufgaben zu Kapitel 2

1) Installieren Sie nach Möglichkeit auf Ihrem privaten PC entweder die Visual C# 2005 Express Edition (gemäß Abschnitt 2.3.2.1) oder SharpDevelop 2.1 (gemäß Abschnitt 2.4).

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 1.2.5).

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 Bruchrechnungs-Beispiel in Abschnitt 1.1 besteht aus den beiden Klassen `Bruch` und `BruchAddition`.
- In einer Klassendefinition werden Felder deklariert 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 Dateien werden gemeinsam übersetzt, z.B.:

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

Der Compiler erstellt daraus ein **exe**-Assembly mit IL-Code und Metadaten.

- Von den Klassen eines Programms muss mindestens 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 Bruchrechnungs-Beispiel in Abschnitt 1.1 ist die Klasse `BruchAddition` startfähig.

- Eine C# - Quellcodedatei beginnt in der Regel mit dem Import von Namensräumen per **using**-Direktive, um später auf deren Klassen bequem (ohne Namensraum-Präfix vor den Klassennamen) zugreifen zu können.
Ohne die Klassen im Namensraum **System** ist kaum ein sinnvolles Programm möglich, so dass dieser Namensraum fast immer importiert wird.

- Sowohl die Klassen- als auch die Methodendefinitionen bestehen aus:

- **Kopf**

In Abschnitt 2.1.1 finden sich im Zusammenhang mit dem `Hallo`-Beispiel nähere Erläuterungen.

o Rumpf

Dieser besteht aus einem Block mit beliebig vielen Anweisungen, mit denen z.B. Variablen deklariert oder verändert werden. Ein Klassen- oder Methodenrumpf wird durch geschweifte Klammern begrenzt.

- 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:

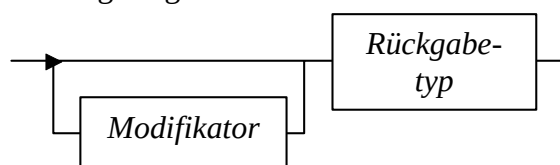
```
using System;
class Prog {
    static void Main() {
        // Platz zum Üben elementarer Sprachelemente
    }
}
```

Später werden Programme mit dieser Einfachstruktur gelegentlich als *Main()-Programme* bezeichnet, wobei Kenner älterer Programmiersprachen an die dort üblichen *Hauptprogramme* denken dürfen. Damit die wenig objektorientierten Beispiele Ihrem Programmierstil nicht prägen, wurde an den Beginn des Kurses ein Beispiel gestellt (Bruchrechnung), das 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.
- Bei Abzweigungen kann man sich für eine Richtung entscheiden, wenn nicht durch Pfeilspitzen eine Bewegungsrichtung vorgeschrieben ist z.B.:



- Für **konstante** Sprachbestandteile, die aus einer Box exakt in der angegebenen Form in konkreten Quellcode zu übernehmen sind, wird **fette** Schrift verwendet.
- *Platzhalter* sind durch *kursive* Schrift gekennzeichnet.

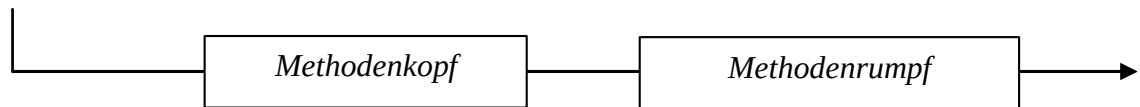
Als Beispiele betrachten wir die Syntaxdiagramme zur Definition von Klassen, Methoden- und Eigenschaften. Aus didaktischen Gründen zeigen die Diagramme nur solche Sprachbestandteile, die bisher in einem Beispiel verwendet oder im Text beschrieben wurden, so dass sie langfristig keinesfalls als Referenz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

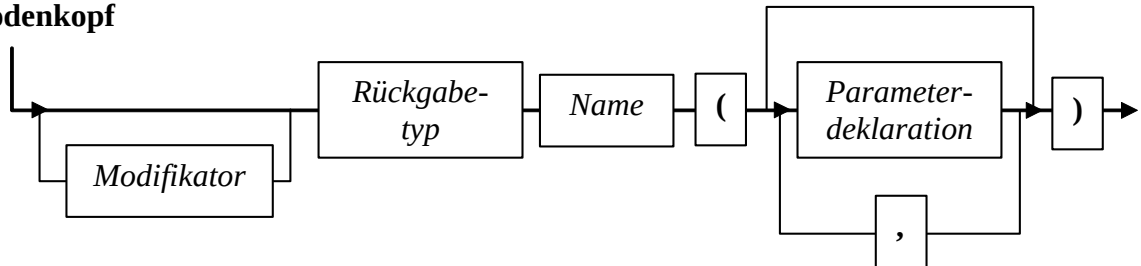
3.1.2.2 Methodendefinition

Weil ein Syntaxdiagramm für die komplette **Methodendefinition** recht unübersichtlich ist, werden die Begriffe *Methodenkopf* und *Methodenrumpf* separat beschrieben:

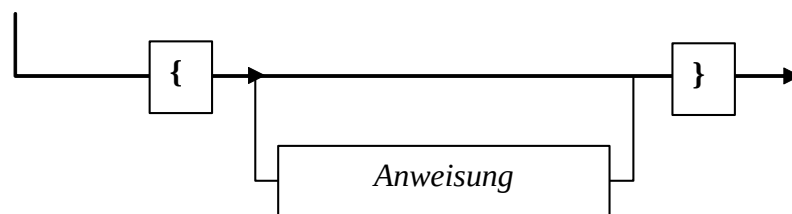
Methodendefinition



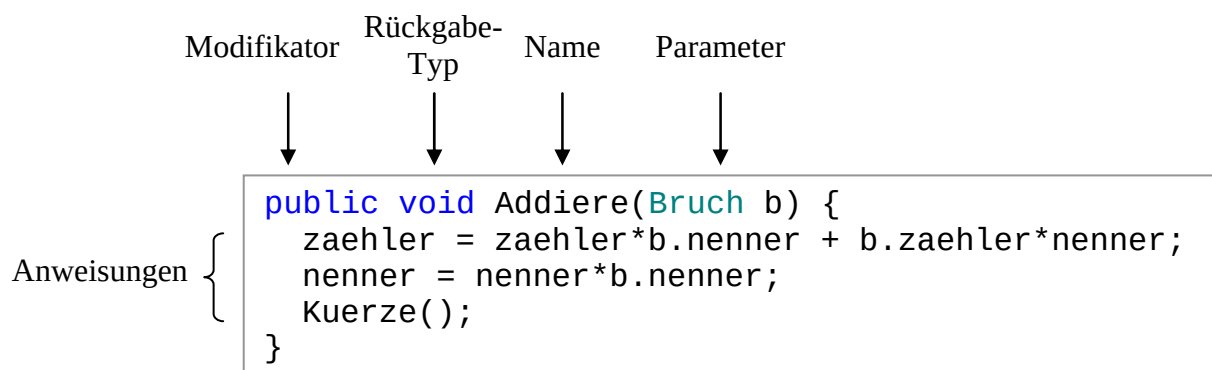
Methodenkopf



Methodenrumpf



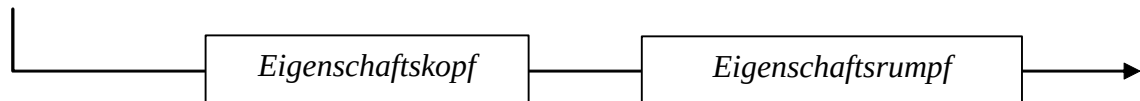
Als Beispiel betrachten wir die Definition der Bruch-Methode `Addiere()`:



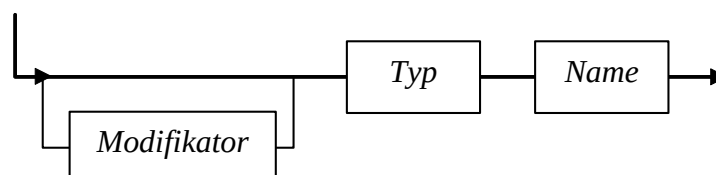
3.1.2.3 Eigenschaftsdefinition

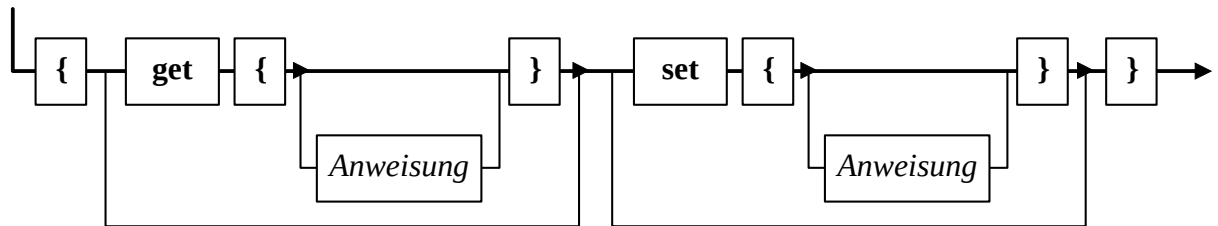
Auch beim Syntaxdiagramm für den Eigenschaftsbegriff gehen wir schrittweise vor:

Eigenschaftsdefinition

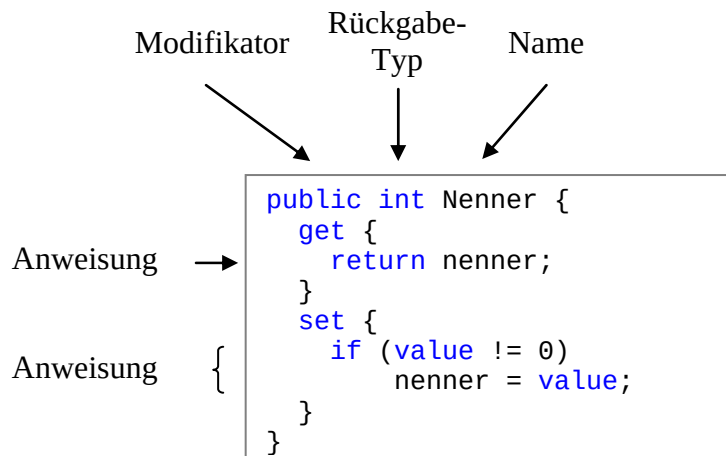


Eigenschaftskopf



Eigenschaftsrumpf

Als Beispiel betrachten wir die Bruch-Eigenschaft Nenner:

**3.1.3 Hinweise zur Gestaltung des Quellcodes**

Der Compiler hat nur wenige Anforderungen an die Formatierung des Quellcodes:

- 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).

Weil auch der Zeilenumbruch als Trennzeichen gilt, kann man den Quellcode übersichtlich gestalten. Dem Compiler ist aber z.B. auch die folgende Quelle recht:

```

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

Wer dieses Manuskript am Bildschirm liest oder an einen Farbdrucker geschickt hat, profitiert hoffentlich von der farblichen Gestaltung vieler Beispiele. Es handelt sich um die Syntaxhervorhebungen von Visual C# 2005 Express Edition, die via Zwischenablage in den Text übernommen wurden. Natürlich sind diese optischen Arbeitshilfen ausschließlich für den Programmierer gedacht, und der Compiler ist komplett farbenblind.

Ob man bei einer Klassen- bzw. Methodendefinition die öffnende geschweifte Klammer an das Ende der Kopfzeile setzt oder an den Anfang der nächsten Zeile, ist Geschmacksache:

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

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

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:

- 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 am Zeilenende kommentiert.

- 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.

- 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 eine HTML-Dokumentation erstellt werden kann (siehe z.B. Drayton et al. 2003, S. 93 ff).

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 0
/// gesetzt wird und hat daher stets einen sinnvollen Wert.
/// </summary>
public int Nenner {
    . . .
}
```

3.1.5 Namen

Für Klassen, Methoden, Variablen, Parameter und sonstige Elemente eines C# - Programms benötigen wir Namen, für die folgende Regeln gelten:

- Die Länge eines Namens ist nicht begrenzt.
- Das erste Zeichen muss ein Buchstabe oder ein Unterstrich sein, danach dürfen außerdem auch Ziffern auftreten.
- Die Groß-/Kleinschreibung ist *signifikant*.
Für den C# - Compiler sind also z.B.

Anz anz

zwei grundverschiedene Namen.

- C# - Programme werden intern im **Unicode**-Zeichensatz dargestellt. Daher erlaubt C# im Unterschied zu den meisten anderen Programmiersprachen in Bezeichnern auch Umlaute oder sonstige nationale Sonderzeichen, die als Buchstaben gelten.
- 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	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 sehr empfehlenswert:

- Die Namen von *lokalen* (methodeninternen) Variablen, Parametern und *privaten* (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 beginnen mit großen Anfangsbuchstaben, z.B.:

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

- Bei zusammengesetzten Namen beginnt jedes Wort mit einem Großbuchstaben, z.B.:

writeLine()

Eine Ausnahme stellen die nach obiger Empfehlung mit einem Kleinbuchstaben zu beginnenden Namen dar, z.B.:

numberOfObjects;

Den Unterstrich zur Verbesserung der Lesbarkeit zusammengesetzter Namen zu verwenden (z.B. number_of_objects), ist in C# *nicht* üblich.

- Die Namen von Methoden ohne Rückgabewert beginnen mit einem Verb, z.B.:

Kuerze()

Bei anderen Namen kommen auch Substantive oder Adjektive in Frage, z.B.:

Bruch	(Name einer Klasse)
Visible	(Eigenschaft der Klasse System.Windows.Forms.Control)

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 mailLink 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 unformatierte Ausgabe einer Zeichenfolge, zumal diese vor Ort aus Einzelstücken zusammengesetzt werden kann.

3.2.1 Unformatierte 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 verketteten, z.B.:

```
int i = 4711;
Console.WriteLine("i hat den Wert: " + i);
```

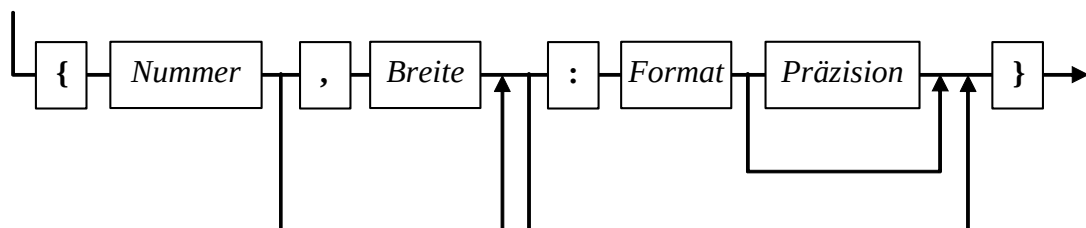
Der Wert der ganzzahligen Variablen `i` wird in eine Zeichenfolge gewandelt, die anschließend an die Zeichenfolge `"i hat den wert: "` angehängt und dann mit ihr zusammen ausgegeben wird.

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. Noch mehr Gestaltungsmöglichkeiten bietet die formatierte Ausgabe:

3.2.2 Formatierte Ausgabe

Bei der formatierten Ausgabe wird als erster Parameter eine Zeichenfolge übergeben, die Platzhalter mit optionalen Formatierungsangaben für die restlichen Parameter enthält. Für einen Platzhalter ist folgende Syntax vorgeschrieben:

Platzhalter für die formatierte Ausgabe



Darin bedeuten:

<i>Nummer</i>	Fortlaufende Nummer des auszugebenden Argumentes, bei 0 beginnend
<i>Breite</i>	Ausgabebreite für das zugehörige Argument Positive Werte bewirken eine <i>rechtsbündige</i> , negative Werte eine <i>linksbündige</i> Ausgabe.
<i>Format</i>	Formatspezifikation gemäß anschließender Tabelle
<i>Präzision</i>	Anzahl der Nachkommastellen oder sonstige Präzisionsangabe (abhängig vom Format), muss der Formatangabe unmittelbar folgen (ohne trennende Leerstellen)

Es werden u.a. folgende Formate unterstützt:

Format	Beschreibung	Beispiele	
		WriteLine-Parameterliste	Ausgabe
d, D	Dezimal (für ganze Zahlen geeignet)	("{0,7:d}", 4711) ("{0,-7:d}", 4711)	4711 4711
f, F	Fixkomma Präzision: Anzahl der Nachkommastellen	("{0,5:f2}", 4.711)	4,71
e, E	Gleitkomma Präzision: Anzahl Stellen in der Mantisse	("{0:e}", 47.11) ("{0:e2}", 47.11) ("{0:E}", 47.11)	4,711000e+001 4,71e+001 4,711000E+001

Mit den folgenden **Escape-Sequenzen** (vgl. Abschnitt 3.3.7.4), die wie gewöhnliche Zeichen in die Formatierungs- oder Ausgabezeichenfolge geschrieben werden, lässt sich die formatierte Ausgabe noch besser gestalten:

<code>\t</code>	Horizontaler Tabulator
<code>\n</code>	Zeilenwechsel (<i>new line</i>)

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 passendem Typ folgen. Damit kann die **WriteLine()**-Variante mit Formatierungszeichenfolge eine variable Anzahl von Argumenten verarbeiten.

Die beschriebenen Formatierungs-Techniken 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 3.5.9 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 =   47
```

Linksbündig:

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

3.3 Variablen und elementare 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 Wertes (z.B. einer Zahl) wird eine so genannte **Variable** verwendet, worunter Sie sich einen **benannten Speicherplatz von bestimmtem Datentyp** 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
    }
}
```

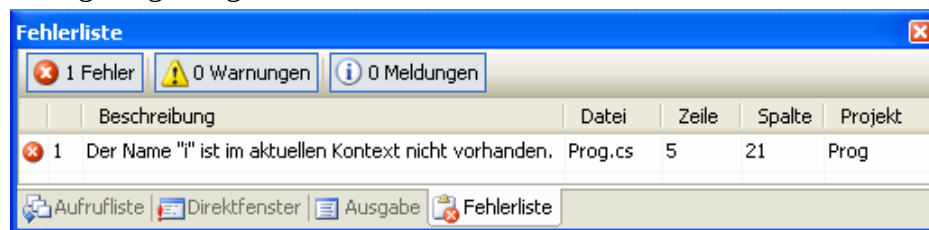
Um die Details bei der Verwaltung der Variablen im Arbeitsspeicher müssen wir uns nicht kümmern. Allerdings verlangt C# beim Umgang mit Variablen im Vergleich zu anderen Programmier- oder Skriptsprachen einige Sorgfalt, letztlich mit dem Ziel, Fehler zu vermeiden:

- Variablen müssen **explizit deklariert** werden.

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

Prog.cs(5,21): error CS0103: Der Name `i` ist im aktuellen Kontext nicht vorhanden.

Wurde der Compiler nicht per Konsole, sondern aus dem Visual C# 2005 heraus aufgerufen (z.B. mit der Funktionstaste **F5**), dann erscheint dieselbe Fehlermeldung in der **Fehlerliste** der Entwicklungsumgebung:



Durch den Deklarationszwang werden z.B. Programmfehler wegen falsch geschriebener Variablenamen verhindert.

- C# ist **streng typisiert**
Jede Variable besitzt einen **Datentyp**, und dieser legt fest, welche Informationen (z.B. ganze Zahlen, reelle Zahlen, Zeichen) gespeichert werden können.

In obigem Beispiel wird die Variable `ivar` vom Typ **int** deklariert, der ganze Zahlen im Bereich von -2147483648 bis 2147483647 als Werte aufnehmen kann.

Die Variable `ivar` 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 3.3.4).

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, zulässige Operationen, Speicherplatzbedarf
- **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 Speicherre-

gionen unterscheiden (*Stack* und *Heap*), weil Performanzgründe die gezielte Auswahl einer bestimmten Region nahe legen können.

3.3.1 Klassifikation von Datentypen und Variablen

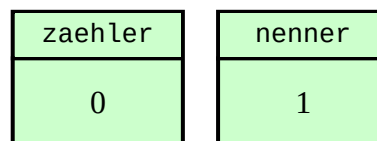
Bei den von C# unterstützten Datentypen ist die Unterscheidung zwischen *Wert-* und *Referenztypen* außerordentlich wichtig:

- **Werttypen**

Besitzt eine Variable einen Werttyp, dann enthält ihr Speicherplatz unmittelbar den Wert, der über den Variablennamen angesprochen wird.

Beispiel (aus der *Bruch*-Klassendefinition):

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



Bei den anschließend in Abschnitt 3.3.2 darzustellenden *elementaren* Datentypen handelt es sich stets um Werttypen.

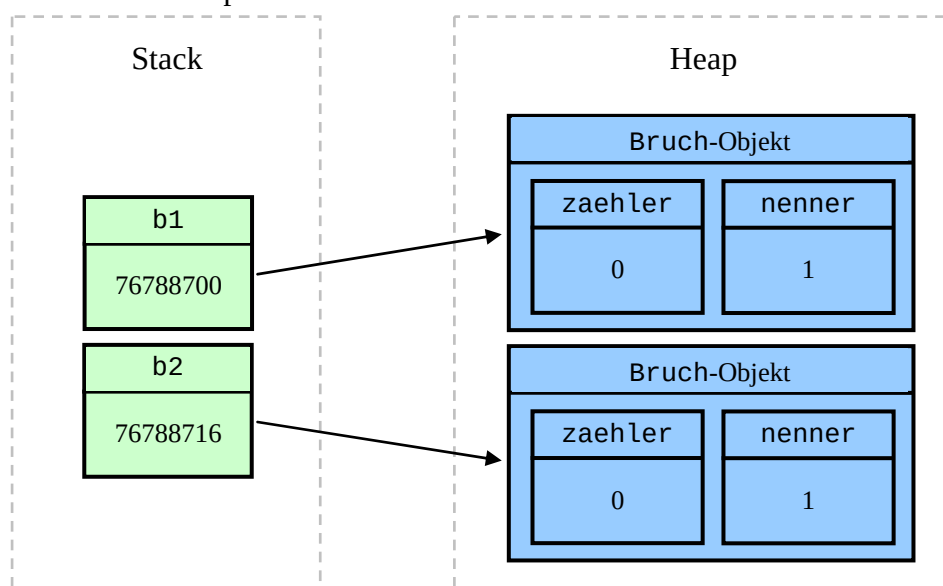
- **Referenztypen**

Besitzt eine Variable einen Referenztyp, dann kann ihr Speicherplatz die Adresse eines Objektes 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.

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. Das von *b1* referenzierte *Bruch*-Objekt wurde bei einem konkreten Programmlauf von der CLR an der Speicheradresse 76788700 (Hexadezimal: 0x0493b3dc) untergebracht. Wir werden uns nicht mit solchen Adressen plagen, sondern die dort abgelegten Objekte über Referenzvariablen ansprechen.



Jedes *Bruch*-Objekt enthält die Felder (Instanzvariablen) *zaehler* und *nenner* vom Werttyp **int**.

In C# unterscheiden sich Variablen nicht nur hinsichtlich des Datentyps, sondern auch hinsichtlich der Zuordnung zu einer Methode, zu einem Objekt oder zu einer Klasse:

- **Lokale Variablen**
Sie werden innerhalb einer Methode deklariert. Ihre Gültigkeit beschränkt sich auf die Methode bzw. auf einen Block innerhalb der Methode (siehe Abschnitt 3.3.5).
- **Instanzvariablen (nicht-statische Felder)**
Jedes Objekt (synonym: jede Instanz) einer Klasse verfügt über einen vollständigen Satz der Instanzvariablen (nicht-statische Felder) der Klasse. So besitzt z.B. jedes Objekt der Klasse `Bruch` einen `zaehler` und einen `nenner`.
- **Klassenvariablen (statische Felder)**
Diese Variablen beziehen sich auf eine gesamte Klasse, nicht auf einzelne Instanzen. Z.B. hält man oft in einer Klassenvariablen fest, wie viele Objekte einer Klasse bereits erzeugt worden sind.

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.

In diesem Abschnitt wurden bislang zwei wichtige Einteilungskriterien für C# - Variablen vorgestellt, die gemeinsam folgendes Klassifikations-Schema ergeben:

		Einteilung nach Zuordnung		
		Lokale Variable	Instanzvariable	Klassenvariable
Einteilung nach Datentyp (Inhalt)	Wertvariable			
	Referenzvariable			

Schließlich kann man bei den Datentypen noch unterscheiden

- **vordefinierte Typen** (als Bestandteile der .NET – Standardbibliothek (FCL))
- **benutzerdefinierte Typen** (Klassen, Strukturen, Aufzählungen)

Die Erläuterungen zur Einteilung der C# - Variablen sind etwas kompliziert geraten. Bis jetzt müssen Sie nur die Ausführungen verstanden haben, die für *lokale* Variablen von *elementarem* Datentyp relevant sind.

3.3.2 Elementare Datentypen

Als *elementare Datentypen* sollen die in C# vordefinierten Werttypen zur Aufnahme von einzelnen Zahlen, Zeichen oder Wahrheitswerten bezeichnet werden. Von der folgenden Tabelle muss man sich beim ersten Lesen nur merken, dass man zum Speichern konkreter Daten die Wahl zwischen vielen elementaren Datentypen hat, die sich hinsichtlich Wertebereich und Platzbedarf im Speicher unterscheiden:

Typ	Beschreibung	Werte	Bits
sbyte	Diese Variablentypen speichern ganze Zahlen <i>mit</i> Vorzeichen. Beispiel: <code>int zaehler = -7</code>	-128 ... 127	8
short		-32768 ... 32767	16
int		-2147483648 ... 2147483647	32
long		-9223372036854775808 ... 9223372036854775807	64
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 reelle Zahlen nach der Norm IEEE 754 (32 Bit) mit einer Genauigkeit von mind. 7 signifikanten Stellen. 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 reelle Zahlen nach der Norm IEEE 754 (64 Bit) mit einer Genauigkeit von mind. 15 signifikanten Stellen. 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	16

Typ	Beschreibung	Werte	Bits
bool	Variablen vom Typ bool speichern Wahrheitswerte. Beispiel: <code>bool cond = false;</code>	true, false	1

3.3.3 Vertiefung: Darstellung reeller Zahlen im Speicher

3.3.3.1 Binäre Gleitkommadarstellung

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

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { double d130 = 1.3f; double d125 = 1.25f; Console.WriteLine("{0,9:f7}", d130); Console.WriteLine("{0,10:f8}", d130); Console.WriteLine("{0,20:f18}", d125); } }</pre>	<pre>1,3000000 1,29999995 1,25000000000000000000</pre>

Bei der **float**-Zahl 1,3 zeigt sich bei einer Ausgabe mit mehr als 7 Nachkommastellen, dass der Wert nicht exakt abgespeichert worden ist. Demgegenüber treten bei der **float**-Zahl 1,25 *keine* Darstellungsfehler auf.

Diese Ergebnisse sind durch das zugrunde liegende **IEEE-754** – Format für die Darstellung von reellen Zahlen zu erklären. Es handelt sich um ein binäres Gleitkommaformat, wobei jede Zahl als Produkt aus 3 getrennt zu speichernden Faktoren dargestellt wird:

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

Als Basis dient beim IEEE-Standard 754 die 2.

Ferner werden alle Zahlen *normalisiert*, d.h. auf eine Mantisse im Intervall [1; 2) gebracht, z.B.:

$$25,28 = 1,58 \cdot 2^4$$

Im ersten Bit einer binären Gleitkommavariablen (Typ **float** oder **double**) wird das Vorzeichen gespeichert (0: positiv, 1: negativ).

Für den Exponenten stehen 8 (**float**) bzw. 11 (**double**) Bit zur Verfügung. Allerdings sind im Exponenten die Werte 0 und 255 (**float**) bzw. 0 und 2047 (**double**) für Spezialfälle (z.B. +/-Unendlich) reserviert (siehe Abschnitt 3.6.2). Um Beträge kleiner 1 darstellen zu können, wird der Exponent mit einer Verschiebung um den Wert 127 (**float**) bzw. 1023 (**double**) abgespeichert, so dass z.B. beim Exponenten 0 einer **float**-Variablen im Speicher der Wert

$$01111111_2 = 127$$

landet.

Die führende 1 der normalisierten Mantisse wird *nicht* abgespeichert, und für den Rest stehen 23 (**float**) bzw. 52 (**double**) Bit zur Verfügung.

Eine **float**- bzw. **double**-Variable mit dem Vorzeichen v (0 oder 1), dem Exponenten e und der Restmantisse m (jeweils dezimal interpretiert) speichert 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 die **float**-Darstellungen einiger Zahlen:

Wert	float-Darstellung		
	Vorz.	Exponent	Mantisse
1,0	0	01111111	000000000000000000000000
1,25	0	01111111	010000000000000000000000
2,0	0	10000000	000000000000000000000000
2,75	0	10000000	011000000000000000000000
-3,5	1	10000000	110000000000000000000000

Nun kommen wir endlich zur Erklärung der eingangs dargestellten Genauigkeitsunterschiede beim Speichern der Zahlen 1,25 und 1,3. Das i -te Bit (bei einer Nummerierung ab 1) in der Restmantisse einer **float**- oder **double**-Variablen hat die Wertigkeit 2^{-i} , so dass sich ihr dezimaler Restmantissenwert folgendermaßen ergibt:

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

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 3.3.2 z.B.

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

als betragsmäßig kleinsten **float**-Wert nennt, obwohl der minimale Exponent nach obigen Überlegungen -126 beträgt, was zum 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 0 zu ermöglichen.

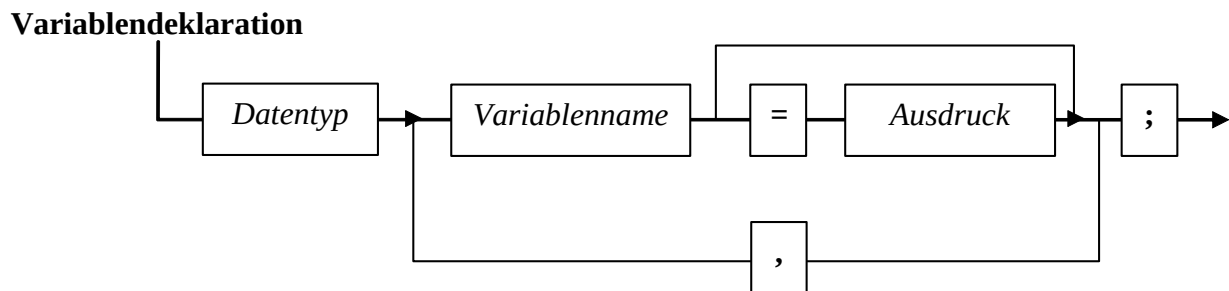
3.3.3.2 Dezimale Gleitkommadarstellung

Neben den eben beschriebenen *binären* Gleitkommatypen (mit der Basis 2) bietet C# auch einen *dezimalen* Gleitkommatyp, dessen Speicherorganisation die Basis 10 verwendet. Beim Datentyp **decimal** werden 102 Bit folgendermaßen verwendet¹:

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

3.3.4 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 **Variablendeklarationsanweisung** zeigt:



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 Methoden-Quellcode an beliebiger Stelle *vor* der ersten Verwendung erscheinen.

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.

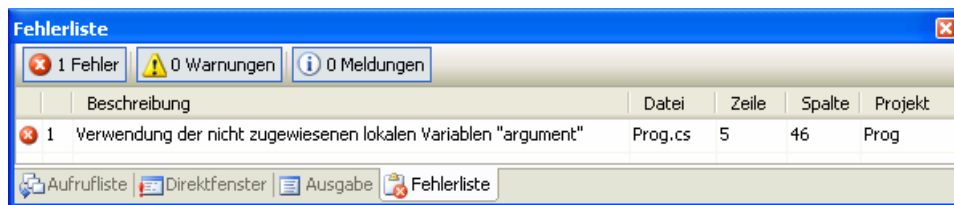
Mit der Objektkreation und auch mit der Konstruktion von gültigen *Ausdrücken*, die einen Wert von passendem Datentyp liefern müssen, werden wir uns noch ausführlich beschäftigen.

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);
    }
}
```

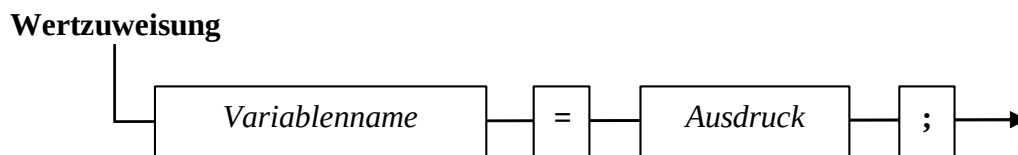
Der Compiler meint dazu:

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.

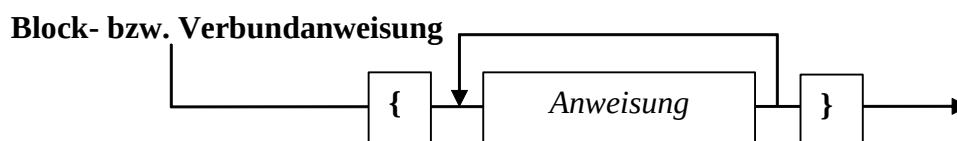
Um den Wert einer Variablen im weiteren Programmablauf zu verändern, verwendet man eine **Wertzuweisung**, die zu den einfachsten und am häufigsten benötigten Anweisungen gehört:



Beispiel: `az = az - an;`

3.3.5 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:



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. Ein realistisches Beispiel war schon in der Methode `Kuerze()` der Klasse `Bruch` zu sehen:

```

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

```

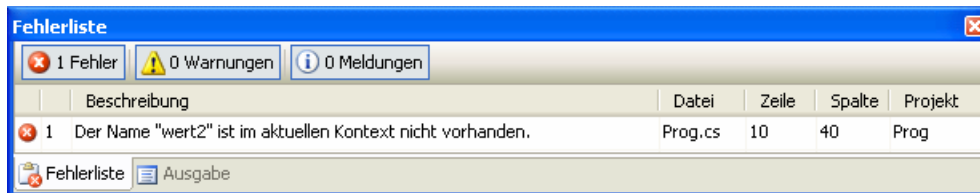
Anweisungsblöcke haben einen wichtigen Effekt auf die Gültigkeit der darin deklarierten Variablen: Eine lokale Variable existiert von der deklarierenden Zeile bis zur schließenden Klammer des innersten Blockes. Nur in diesem **Deklarationsbereich** kann sie über ihren Namen angesprochen werden, so dass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

```

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

```

mit einer Fehlermeldung ablehnt:



Bei hierarchisch geschachtelten Blöcken ist es in C# *nicht* erlaubt, auf mehreren Stufen Variablen mit identischem Namen zu deklarieren. Mit dem Verzicht auf diese kaum sinnvolle Option haben die C# - Designer eine weitere Möglichkeit beseitigt, schwer aufzuklärende Programmierfehler zu begehen.

Zur übersichtlichen Gestaltung von C# - Programmen ist das Einrücken von Anweisungsblöcken sehr zu empfehlen, wobei die Position der einleitenden Blockklammer und die Einrücktiefe zu wählen sind, z.B.:

<pre> if (wert1 == 1) { int wert2 = 2; Console.WriteLine("Wert2 = "+wert2); } </pre>	<pre> if (wert1 == 1) { int wert2 = 2; Console.WriteLine("Wert2 = "+wert2); } </pre>

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.

Spezialisierte Editoren für Programmierer bieten noch mehr Komfort, indem sie z.B. beim Markieren einer Blockklammer das Gegenstück hervorheben.

3.3.6 Konstanten

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

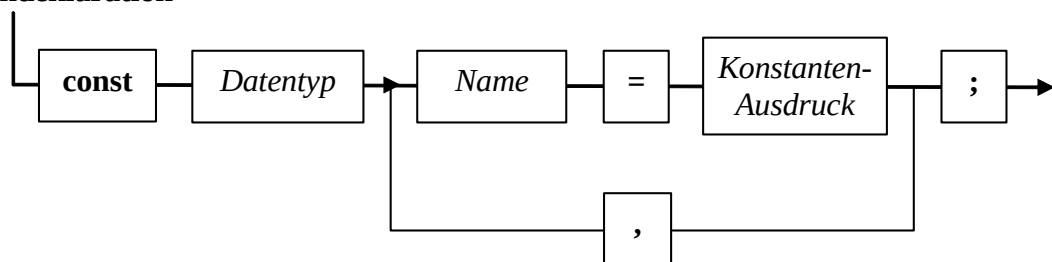
Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { const double MWST = 1.16; double netto = 28.10, brutto; brutto = netto * MWST; Console.WriteLine("Brutto: {0:f2}", brutto); } }</pre>	Brutto: 32,60

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

- Ihre Deklaration beginnt mit dem Modifikator **const** und *muss* eine Initialisierung enthalten. Dabei ist ein Ausdruck zu verwenden, den der *Compiler* berechnen kann.
- Ihr Wert kann im Programm nicht geändert werden.
- Der Name einer Konstanten wird in der Regel komplett groß geschrieben, um ihn leicht von Variablennamen unterscheiden zu können.

Das Syntaxdiagramm zur Deklaration einer Konstanten:

Konstantendeklaration



3.3.7 Literale

Die im Programmcode auftauchenden expliziten Werte bezeichnet man als *Literale*. Wie Sie aus dem Abschnitt 3.3.6 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.

3.3.7.1 Ganzzahlliterale

Ganzzahlliterale können im dezimalen oder im hexadezimalen Zahlensystem 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
```

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** hat den ersten Typ aus folgender Serie, der seinen Wert aufnehmen kann:

uint, ulong

Beispiele:

Literale	Typ
214U	uint
9223372036854775808u	ulong

- Ein Ganzzahlliteral mit Suffix **l** oder **L** (kleiner bzw. großer Anfangsbuchstabe von *Lima*!) hat den ersten Typ aus folgender Serie, der seinen Wert aufnehmen kann:

long, ulong

Beispiele:

Literale	Typ
214L	long
9223372036854775808L	ulong

Der Kleinbuchstabe **l** ist leicht mit der Ziffer **1** zu verwechseln und daher als Suffix weniger geeignet.

- Ein Ganzzahlliteral mit Suffix **ul**, **lu**, **UL**, **LU**, **uL**, **Lu**, **Ul**, oder **IU** hat den Typ **ulong**.
- Kann ein Wert von keinem Datentyp aufgenommen werden, produziert der Compiler eine Fehlermeldung, z.B.

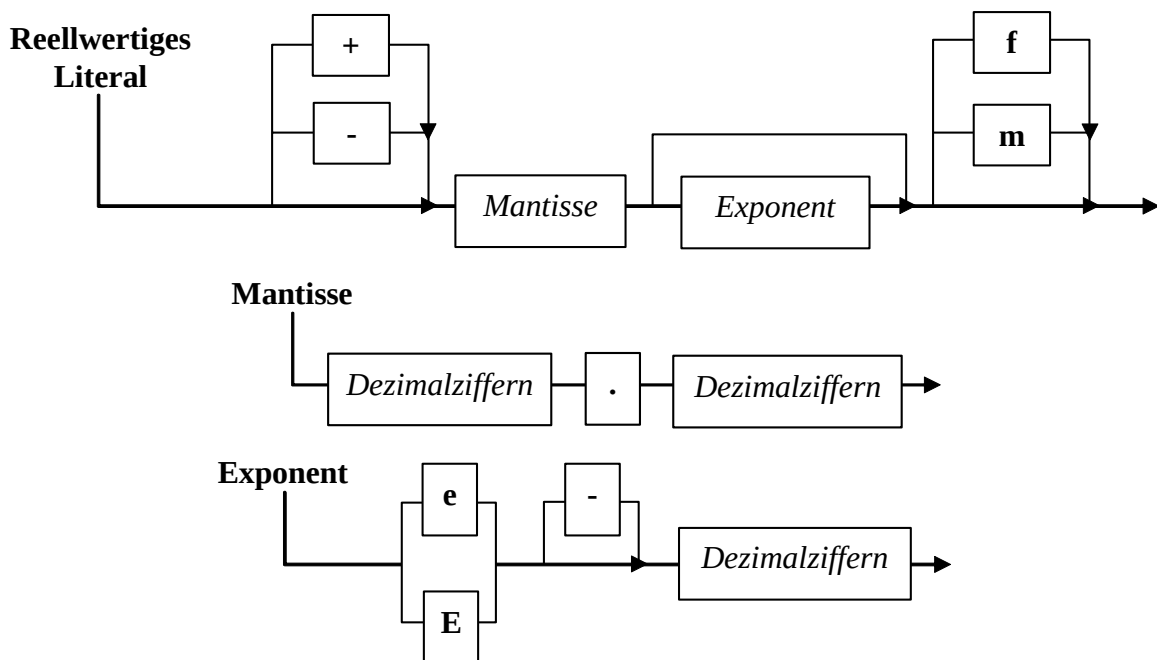
Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { Console.WriteLine(18446744073709551616); } }</pre>	Prog.cs(4,21): error CS1021: Die integrale Konstante ist zu groß

3.3.7.2 Literale für reelle Zahlen

Zahlen mit Dezimalpunkt oder Exponent sind in C# vom Typ **double**, wenn nicht per Suffix ein alternativer Typ erzwungen wird:

- Durch den Suffix **F** oder **f** wird der Datentyp **float** erzwungen, z.B.:
9.78f
- Durch den Suffix **M** oder **m** wird der Datentyp **decimal** erzwungen, z.B.:
19.78m

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



Beispiele:

```

9.78f
2279800223423485.45m
0.45875e-20
  
```

Der Compiler achtet bei Wertzuweisungen streng auf die Typkompatibilität. Z.B. führt die folgende Deklarations-Anweisung:

```
const float P = 4.252602;
```

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.
  
```

3.3.7.3 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 in der Konsolenausgabe anders erscheinen, z.B.:

Quellcode	Ausgabe
<pre> using System; class Prog { static void Main() { bool b = false; Console.WriteLine(b); } } </pre>	

3.3.7.4 *char-Literale*

char-Literale werden in C# mit einfachen Hochkommata eingerahmt. Dabei sind erlaubt:

- **Einfache Zeichen**

Beispiel:

```
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** mit einer hexadezimalen Unicode-Zeichennummer nach der Einleitung durch \u oder \x

So lassen sich Zeichen nutzen, die per Tastatur nicht einzugeben sind.

Beispiel:

```
char alpha = '\u03b1';
```

Im Konsolenfenster werden die Unicode-Zeichen oberhalb von \u00ff in der Regel als Fragezeichen dargestellt. Auf der graphischen Benutzeroberfläche erscheinen alle Unicode-Zeichen in voller Pracht (siehe nächsten Abschnitt).

3.3.7.5 *Zeichenkettenliterale*

Zeichenkettenliterale werden (im Unterschied zu **char**-Literalen) durch *doppelte* Hochkommata begrenzt. Hinsichtlich der erlaubten Zeichen und der Escape-Sequenzen gelten die Regeln für **char**-Literale analog, wobei das einfache und das doppelte Hochkomma ihre Rollen tauschen, z.B.:

```
string name = "Otto's Welt";
```

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

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

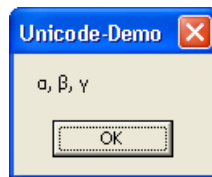
Während ein **char**-Literal stets *genau ein* Zeichen enthält, kann ein Zeichenkettenliteral aus beliebig vielen Zeichen bestehen oder auch leer sein, z.B.:

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

Das folgende Programm enthält einen Aufruf der statischen **Show()**-Methode der Klasse **MessageBox** aus dem Namensraum **System.Windows.Forms** (siehe Abschnitt 2.3.2.3) zur Anzeige eines Zeichenkettenliterals, das mehrere Unicode-Escape-Sequenzen enthält:

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

Es resultiert 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");
```

3.3.8 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:

```
class Prog {
    static void Main() {
        float pi = 3,141593;
        double radius = 2,0;
        System.Console.WriteLine('Der Flaecheninhalte betraegt: {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 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 diesen Wert zurück, und er landet in der **int**-Eigenschaft `Zaehler`.

Im Unterschied zu früheren Beispielen liegt hier eine Verschachtelung bzw. Hintereinanderschaltung zweier Methodenaufrufe vor, die bei Programmierern der kompakten Schreibweise wegen recht 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 einer unbehandelten Ausnahme, 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>	<pre>Ihre Lieblingszahl? drei Unbehandelte Ausnahme: System.FormatException: Die Eingabezeichenfolge hat das falsche Format.</pre>

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

Quellcode	Ein- und Ausgabe
<pre> using System; class Prog { static void Main() { int zahl; try { Console.WriteLine("Ihre Lieblingszahl? "); zahl = Convert.ToInt32(Console.ReadLine()); Console.WriteLine("Verstanden: " + zahl); } catch { Console.WriteLine("Falsche Eingabe!"); } } } </pre>	Ihre Lieblingszahl? drei Falsche Eingabe!

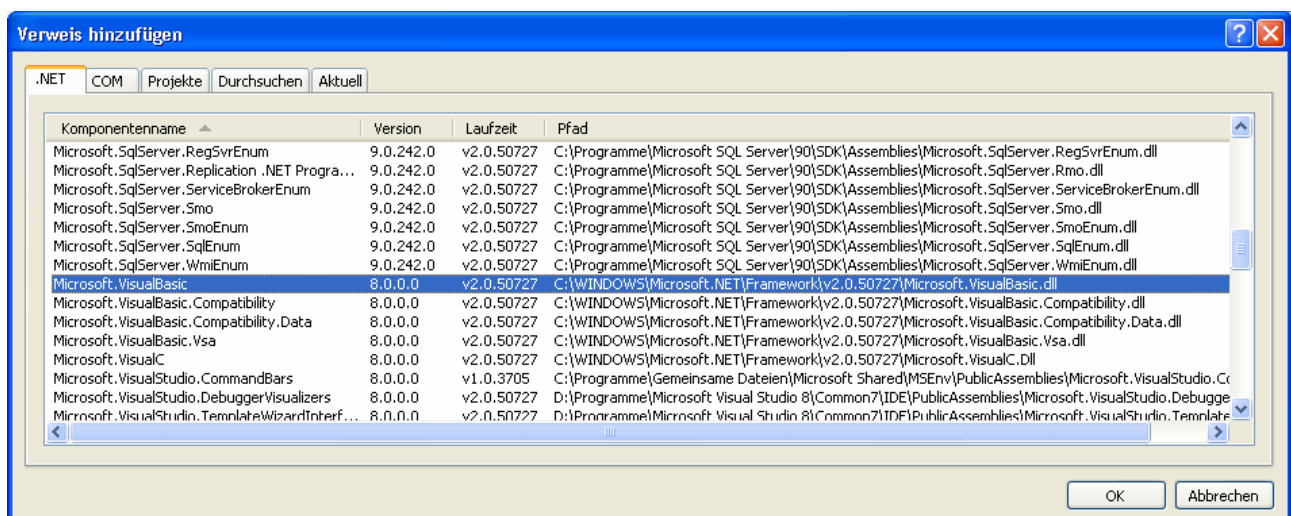
In den Übungs- bzw. Demoprogrammen verwenden wir der Einfachheit halber ungesicherte **ToInt32** – Aufrufe bzw. analoge Varianten für andere Datentypen.

3.4.2 Via InputBox

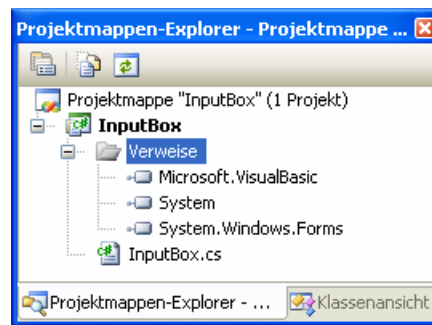
Wer schon jetzt Anwendungen mit grafikorientierter Benutzerinteraktion erstellen möchte, kann statt **ReadLine()** z.B. die statische Methode **InputBox()** der Klasse **Interaction** aus dem Namensraum **Microsoft.VisualBasic** benutzen. Zwar gehört diese Klasse nicht zur .NET – Standardbibliothek, doch befindet sie sich in einer Assembly-Datei (**microsoft.visualbasic.dll**), die mit dem .NET – Redistributable im **GAC** (Global Assembly Cache) installiert wird. Somit kann diese als Migrationshilfe für Visual Basic - Programmierer gedachte Klasse auch in C# - Programmen benutzt werden.

Dazu muss dem Compiler das Assembly per **/reference** – Option bekannt gemacht werden, z.B.:
 csc /reference:microsoft.visualbasic.dll Prog.cs

Wie Sie aus Abschnitt 2.3.2.3 bereits wissen, kann man unseren Entwicklungsumgebungen sehr bequem erklären, welche Assembly-Referenzen beim Übersetzen eines Projektes erforderlich sind. Man wählt im **Projektmappen-Explorer** aus dem Kontextmenü zum Projektnamen die Option **Verweis hinzufügen** (Visual C# 2005) bzw. **Referenz hinzufügen** (SharpDevelop 2.1) und kann dann in folgender Dialogbox das gesuchte Assembly lokalisieren und im markierten Zustand mit **OK** in die Verweisliste aufnehmen:



Anschließend erscheint die gewählte Referenz in der Projektansicht:



Sollte die Klasse im Quellcode häufiger auftreten, lohnt sich das Importieren des Namensraums:

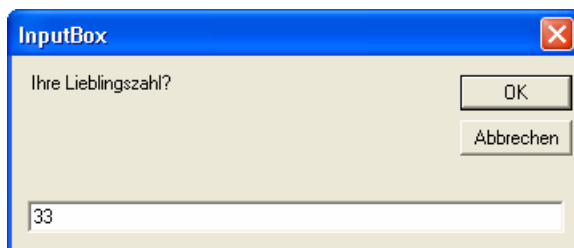
```
using Microsoft.VisualBasic;
```

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:

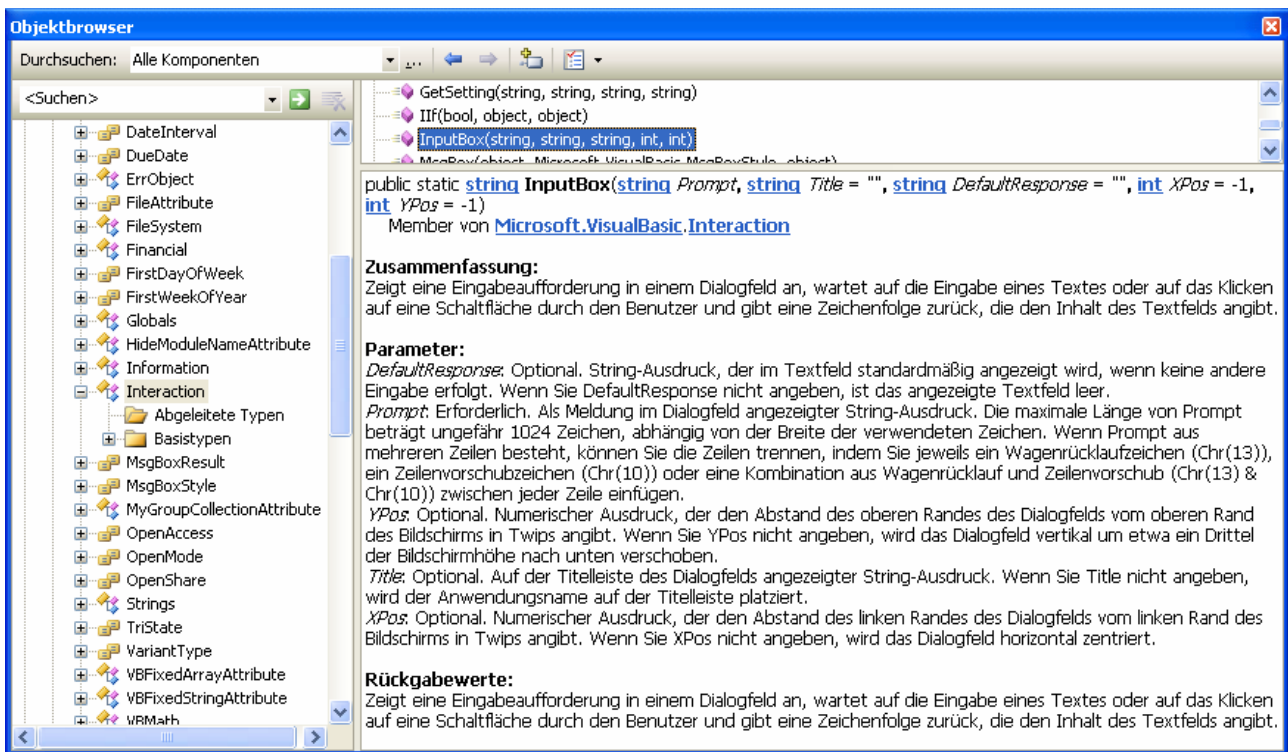


Wie man ein überflüssiges Konsolenfenster unterdrückt, wurde in Abschnitt 2.5 im Rahmen einer Übungsaufgabe gezeigt.

Über die Parameter (Argumente) des **Inputbox()**-Methodenaufrufs kann man sich in der Visual C# 2005 Express Edition über

Ansicht > Objektbrowser

informieren:



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.

Obwohl die Klasse **Interaction** nicht zur .NET – Standardbibliothek gehört, ist gegen ihre Verwendung wenig einzuwenden. Sollte sich das benötigte Assembly **microsoft.visualbasic.dll** irgendwann nicht mehr im GAC befinden, müsste es eben mit einer eigenen Anwendung mitgeliefert werden.

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.

Die **Convert**-Methode **ToInt32** reagiert natürlich auch im optisch aufgewerteten Programm auf ungeschickte Benutzereingaben mit einer Ausnahme. In einigen Wochen 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; 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. Am Ende des Abschnitts kann immerhin schon das Programmieren eines Währungskonverters als Übungsaufgabe gestellt werden. Allzu große Begeisterung wird wohl trotzdem nicht aufkommen, doch im Sinne einer zügigen und fehlerfreien Softwareentwicklung kommen wir an den handwerklichen Grundlagen nicht vorbei.

Schon bei einem Literal, einer Variablen oder einem Methodenaufruf handelt es sich um einen Ausdruck.

Abgesehen von Methodenaufrufen mit dem Pseudo-Rückgabebetyp **void** (siehe unten) hat jeder Ausdruck einen **Datentyp** und einen **Wert**.

Beispiel: 1.5

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

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

Beispiel: 2 * 1.5

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

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: ++ivar

Der Präinkrementoperator erhöht den Wert der Variablen `ivar` um 1 (Nebeneffekt) und liefert den neuen `ivar`-Inhalt als Ergebnis des Ausdrucks.

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 vielfach eine automatische (implizite) Typanpassung „nach oben“ statt (vgl. Abschnitt 3.5.6). Bevor z.B. ein ganzzahliges Argument an der Gleitkommaarithmetik teilnimmt, wird es automatisch in einen passenden Gleitkommatyp gewandelt.

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
-Var	Vorzeichenumkehr	<pre>int i = 2; Console.WriteLine(-i);</pre>	-2
Num1 + Num2	Addition	<pre>Console.WriteLine(2 + 3);</pre>	5
Num1 – Num2	Subtraktion	<pre>Console.WriteLine(2.6 - 1.1);</pre>	1,5
Num1 * Num2	Multiplikation	<pre>Console.WriteLine(4 * 5);</pre>	20
Num1 / Num2	Division	<pre>Console.WriteLine(8.0 / 5); Console.WriteLine(8 / 5);</pre>	1,6 1
Num1 % Num2	Modulo (Restwert) Ergebnis von <i>Num1</i> % <i>Num2</i> ist der Rest der Division von <i>Num1</i> durch <i>Num2</i> .	<pre>Console.WriteLine(9 % 5); Console.WriteLine(9 % 5.4);</pre>	4 3,6
++Var --Var	Präinkrement bzw. -dekrement Als Argument ist nur eine Variable erlaubt. ++Var liefert Var + 1 erhöht Var um 1 --Var liefert Var - 1 reduziert Var um 1	<pre>int i = 4; double a = 1.2; Console.WriteLine(++i + "\n" + --a);</pre>	5 0,2
Var++ Var--	Postinkrement bzw. -dekrement Als Argument ist nur eine Variable erlaubt. Var++ liefert Var erhöht Var um 1 Var-- liefert Var reduziert Var um 1	<pre>int i = 4; Console.WriteLine(i++ + "\n" + i);</pre>	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):

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { int i = 12; i++; Console.WriteLine(i); } }</pre>	13

Es handelt sich letztlich nur um eine vereinfachte Schreibweise: So ist z.B.

```
Console.WriteLine(++i);
```

äquivalent zu

```
i = i + 1;
Console.WriteLine(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 3.3.2) 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-Rückgabetyt **void** anzugeben.
- Abgesehen von ihrem Rückgabetyt hat eine Methode oft weitere Effekte, z.B. auf die Merkmalsausprägungen des handelnden Objekts oder auf die Konsolenausgabe.

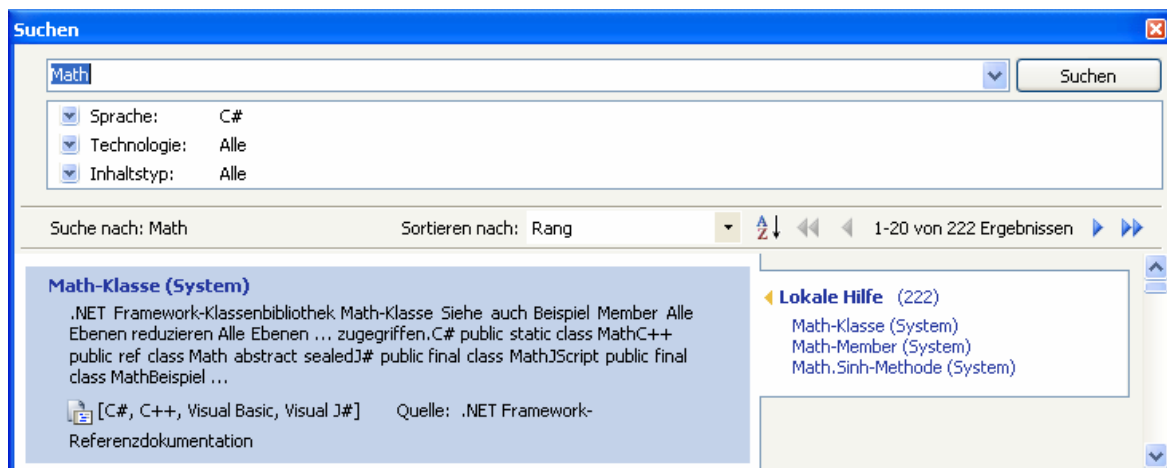
Ein Methodenaufruf stellt einen Ausdruck dar, wobei seine Rückgabe den Datentyp und den Wert des Ausdrucks bestimmt. Bei einer Methode ohne Rückgabewert resultiert ein Ausdruck vom Typ **void**, der nicht als Argument für Operatoren oder Methoden taugt. Als *Anweisung* (vgl. Abschnitt 3.7) ist ein solcher Methodenaufruf wegen der weiteren Effekte aber sehr gut zu gebrauchen, wie Sie aus den zahlreichen Einsätzen der Methode **Console.WriteLine()** bereits wissen.

Als Programmieranfänger muss man sich nicht mit der etwas abstrakten Vorstellung plagen, dass beim Methodenaufruf der einstellige Operator „()“ im Spiel ist (siehe Operatorentabelle in Abschnitt 3.5.9).

Stattdessen sollen an dieser Stelle die Methoden der Klasse **Math** im Namensraum **System** der FCL vorgestellt werden, die zahlreiche mathematische Standardfunktionen (z.B. Logarithmus, Wurzel, trigonometrische und hyperbolische Funktionen) realisieren. In folgendem Programm wird die Methode **Pow()** zur Berechnung der allgemeinen Potenz (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. Eine ausführliche Dokumentation der Klasse **Math** findet man in der SDK-Dokumentation z.B. über die Suchfunktion:



Im obigen Beispielprogramm liefert die Methode **Math.Pow()** einen Rückgabewert vom Typ **double**, der gleich als Argument der Methode **Console.WriteLine()** Verwendung findet:

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

Solche Verschachtelungen sind bei Programmierern wegen ihrer Kompaktheit ähnlich beliebt wie die Inkrement- bzw. Dekrementoperatoren. Ein etwas umständliches, aber für Anfänger leichter verständliches Äquivalent zur obigen Anweisung könnte z.B. lauten:

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

Mit Hilfe der **Math**-Methoden lassen im weiteren Verlauf des Manuskripts viele Beispiele interessanter gestalten. Aus diesem Grund wurden die Methodenaufrufe als spezielle Ausdrücke etwas früher als üblich behandelt.

3.5.3 Vergleichsoperatoren

Durch Anwendung eines *Vergleichsoperators* auf zwei komparable Argumentausdrücke entsteht ein **Vergleich**. Dies ist ein einfacher **logischer Ausdruck** (vgl. Abschnitt 3.5.4), 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 beliebige, miteinander vergleichbare 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. Einer der häufigsten 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. Im ersten **WriteLine()**-Aufruf des folgenden Programms wird das Ergebnis eines Vergleichs auf die Konsole geschrieben:

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 eine *Wertzuweisungsausdruck* (siehe Abschnitt 3.5.7) 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

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 Programms recht unangenehm werden kann.

In vielen Fällen warnt der C# - Compiler allerdings, wenn anstelle des Identitätsoperators eine Zuweisung auftritt.

Im Zusammenhang mit den Gleitkommatypen, insbesondere bei **float** und **double**, gilt es beim Identitätsoperator (auch bei korrekter Schreibweise) eine weitere Problematik zu beachten: Aufgrund der inhärenten Genauigkeitsprobleme bei den genannten Datentypen (siehe Abschnitt 3.3.3) muss man in der Regel an Stelle des Identitätsvergleichs eine Technik wählen, die technisch bedingte Abweichungen von der reinen Mathematik toleriert, z.B.:

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { const double DRF = 1.0e-14; double d1 = 10.0 - 9.9; double d2 = 0.1; Console.WriteLine(d1 == d2); Console.WriteLine(Math.Abs((d1 - d2)/d1) < DRF); } }</pre>	False True

Nach diesem Vorschlag werden zwei **double**-Werte d_1 und d_2 dann als „numerisch“ identisch betrachtet, wenn der relative Abweichungsbetrag kleiner ist als $1,0 \cdot 10^{-14}$, wobei die Wahl der Bezugsgröße (d_1 oder d_2) beliebig ist:

$$\left| \frac{d_1 - d_2}{d_1} \right| < 1,0 \cdot 10^{-14}$$

Wer an den mathematischen Details (derzeit) nicht interessiert ist, kann sich auf die Verwendung der vorgeschlagenen Unterschiedsschwelle beschränken und deren Begründung ignorieren: Seien

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

zwei **double**-Zahlen mit demselben wahren Wert t und Rundungsfehlern e_i . Dann gilt für den technisch bedingten relativen Abweichungsbetrag folgende Abschätzung:

$$\left| \frac{d_1 - d_2}{d_1} \right| = \left| \frac{e_1 - e_2}{t + e_1} \right| < \frac{|e_1| + |e_2|}{|t + e_1|} < \frac{2 \cdot 5 \cdot 10^{-15}}{|t + e_1|} = \frac{10^{-14}}{|t + e_1|} < 10^{-14} \quad (|t + e_1| \geq 1)$$

In der normalisierten **double**-Mantisse (siehe Abschnitt 3.3.3) mit Werten im Intervall $[1; 2)$

$$1, z_1 z_2 \dots z_{14} z_{15} \dots$$



potentieller Rundungsfehler

sind mindestens 15 Dezimal- und damit 14 Nachkommastellen signifikant, so dass man zu einem maximalen Rundungsfehler von $5 \cdot 10^{-15}$ kommt.

Ist bei zwei **double**-Werten d_1 und d_2 der relative Abweichungsbetrag gleich $1,0 \cdot 10^{-14}$ oder größer, können sie nicht mehr als numerisch identisch betrachtet werden.

Für die Betrachtung der *relativen* Abweichung spricht, dass Mantissen-Rundungsfehler in Abhängigkeit vom Exponenten zu sehr unterschiedlichen Gesamtfehlern führen.

Bei konkreten Algorithmen kann durchaus eine gröbere Indifferenzschwelle angemessen sein, wenn mit einer Fehlerkumulation zu rechnen ist.

In einer späteren Übungsaufgabe (siehe Abschnitt 3.7.4) sollen Sie sich erneut mit dem Problem der begrenzten Maschinengenauigkeit befassen.

3.5.4 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 <i>La1</i>	Negation <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>	Log. ODER <i>La1 La2</i> <i>La1 La2</i>	Exklusiv. 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.	<pre>bool erg = true; Console.WriteLine(!erg);</pre>	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.	<pre>int i = 3; bool erg = false && i++ > 3; Console.WriteLine(erg + "\n" + i); erg = true && i++ > 3; Console.WriteLine(erg + "\n" + i);</pre>	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.	<pre>int i = 3; bool erg = false & i++ > 3; Console.WriteLine(erg + "\n" + i);</pre>	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.	<pre>int i = 3; bool erg = true i++ == 3; Console.WriteLine(erg + "\n" + i); erg = false i++ == 3; Console.WriteLine(erg + "\n" + i);</pre>	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.	<pre>int i = 3; bool erg = true i++ > 3; Console.WriteLine(erg + "\n" + i);</pre>	True 4

Operator	Bedeutung	Beispiel	
		Programmfragment	Ausgabe
$La1 \wedge La2$	Exklusives logisches ODER $La1 \wedge La2$ 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, wie das Beispiel

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

zeigt. Der Postinkrementoperator erhöht beim Auswerten des rechten Ausdrucks den Wert der Variablen **i**. Eine solche Auswertung wird jedoch in der folgenden Variante des Beispiels unterlassen, wenn bereits nach Auswertung des linken Ausdrucks das Gesamtergebnis **false** fest steht:

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

Wie der Tabelle auf Seite 91 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 **|** im nächsten Abschnitt auch für bitorientierte Operatoren verwendet. 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.

3.5.5 Vertiefung: Bitorientierte Operatoren

Über unseren momentanen Bedarf hinaus gehend bietet C# einige Operatoren zur bitweisen Manipulation von Variableninhalten. Statt einer systematischen Darstellung der verschiedenen Operatoren (siehe z.B. SDK-Dokumentation) beschränken wir uns auf ein Beispielprogramm, das zudem generell 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 wird mit der **for**-Schleife eine Wiederholungsanweisung benutzt, die erst einige Abschnitte später offiziell vorgestellt wird:

Quellcode	Ausgabe
<pre>using System; class CBit { 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** `<<` in:

```
1 << i
```

verschiebt die Bits in der binären Repräsentation der Ganzzahl 1 um i Stellen nach links.

Von den 32 Bit, die ein **int**-Wert insgesamt belegt (siehe Abschnitt 3.3.2), interessieren im Augenblick nur die rechten 16. Bei der 1 erhalten wir:

```
0000000000000001
```

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

```
0000000001000000
```

Nach dem Links-Shift- kommt der **bitweise UND-Operator** zum Einsatz:

```
1 << i & cbit
```

Das Operatorzeichen **&** wird leider in doppelter Bedeutung verwendet: Wenn beide Argumente vom Typ **bool** sind, wird **&** als *logischer* Operator interpretiert (siehe Abschnitt 3.5.4). 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 1 enthält, wenn *beide* Argumentmuster an dieser Stelle eine 1 besitzen.

Bei `cbit = 'x'` ist das Unicode-Bitmuster

```
0000000001111000
```

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

```
0000000001000000
```

Das von `1 << i & cbit` erzeugte Bitmuster hat den Typ **int** und kann daher mit der 0 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 1 hat.

3.5.6 Typumwandlung (Casting)

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 der Konvertierung 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 = 9223372036854775807; Console.WriteLine("{0,20:f2}", i); Console.WriteLine("{0,20:f2}", i * 1.0); } }</pre>	<pre>9223372036854775807,00 9223372036854780000,00</pre>

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 3.5.5 bekannten) Bitmusters zum kleinen „x“ ausgegeben, damit Sie nachvollziehen können, warum das letzte Programm beim „Halbieren“ dieses Zeichens auf den Wert 60 kam:

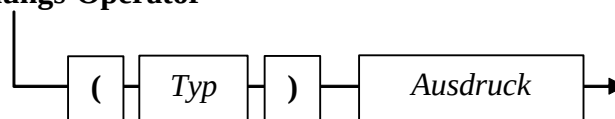
Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { Console.WriteLine((int)'x'); double a = 3.14159; Console.WriteLine((int)a); a = 1.45367e50; Console.WriteLine((int)a); } }</pre>	<pre>120 3 -2147483648</pre>

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** schwere Programmierfehler möglich, wenn die Wertebereiche der beteiligten Datentypen nicht beachtet werden.

Die C# - Syntax zur expliziten Typumwandlung:

Typumwandlungs-Operator



3.5.7 Zuweisungsoperatoren

Bei den ersten Erläuterungen zur Wertzuweisung (vgl. Abschnitt 3.3.4) blieb aus didaktischen Gründen unerwähnt, dass eine Wertzuweisung einen *Ausdruck* darstellt, dass wir es also mit einem 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 kompatibelem Typ stehen.
- Der zugewiesene Wert stellt auch den Ergebniswert des Ausdrucks dar.

Analog zum Verhalten des 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>	4711 4711

Beim Auswerten des Ausdrucks „ivar = 4711“ entsteht der an **WriteLine()** zu übergebende Wert, und die Variable ivar wird verändert.

Selbstverständlich kann eine Zuweisung auch als Operand in einen übergeordneten Ausdruck integriert werden, z.B.:

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

Weil bei mehrfachem Auftreten des Zuweisungsoperators eine Abarbeitung von **rechts nach links** erfolgt (siehe Abschnitt 3.5.9), passiert im letzten Beispiel folgendes:

- $j * i$ liefert das Ergebnis 8, das der Variablen j zugewiesen wird und gleichzeitig den Wert des Ausdrucks $j = j * i$ darstellt.
- In der zweiten Zuweisung (bei Betrachtung von rechts nach links) wird der Wert des Ausdrucks $j = j * i$ an die Variable i übergeben.

Ausdrücke der Art

$i = j = k;$

stammen übrigens *nicht* aus einem Kuriositätenkabinett, sondern sind in den C# - Programmen von schreibfaulen Autoren oft anzutreffen.

Wie wir spätestens seit Abschnitt 3.3.4 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 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 += 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

¹ Die Spalte zeigt jeweils den neuen Wert der Variablen *i*.

3.5.8 Konditionaloperator

Der **Konditionaloperator** erlaubt eine sehr kompakte Schreibweise, wenn beim neuen Wert einer Zielvariablen bedingungsabhängig zwischen zwei Ausdrücken zu entscheiden ist, z.B.

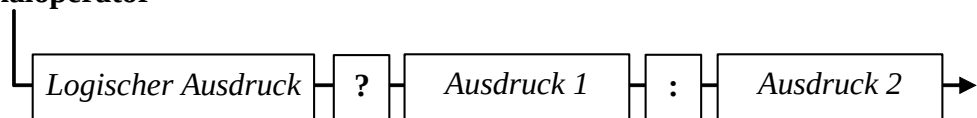
$$i = \begin{cases} i + j & \text{falls } k > 0 \\ i - j & \text{sonst} \end{cases}$$

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 : j-j; Console.WriteLine(i); } }</pre>	3

Eine Besonderheit des Konditionaloperators besteht darin, dass er 3 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*.

3.5.9 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 numerischen 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.:

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

Hier ist unbedingt zu beachten, dass in C# (wie in praktisch jeder anderen Programmiersprache) der Subtraktionsoperator von links nach rechts ausgewertet wird, so dass der Ausdruck

$$3 - 2 - 1$$

den Wert 0 besitzt.

3) Klammern

Wenn die aus obigen Regeln resultierende Auswertungsfolge nicht zum gewünschten Ergebnis führt, 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 mit Auswertungsrichtung aufgelistet. Gruppen von Operatoren mit gleicher Priorität sind durch fette horizontale Linien voneinander abgegrenzt. In der **Operanden**-Spalte werden die zulässigen Datentypen der Argument-Ausdrücke 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
<i>V_n</i>	Variable mit numerischem Datentyp

Operator	Bedeutung	Operanden
<i>Methode()</i>	Methodenaufruf	
++, --	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.

3.5.10 Übungsaufgaben zu Abschnitt 3.5

1) Welche Werte und Datentypen besitzen die folgenden Ausdrücke?

6/4*2.0

(int)6/4.0*3

$3*5+8/3\%4*5$

2) Welche Werte haben die Variable `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 recht 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 .NET - SDK-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. Liegt das Ergebnis eines Ausdrucks außerhalb des Zieltyp-Wertebereichs, ist ein korrektes Speichern nicht möglich. Bei der Auswertung eines komplexen Ausdrucks kann ein Wertebereichs-Problem auch schon „unterwegs“ auftreten. Im betroffenen Programm ist mit einem mehr oder weniger gravierenden Fehlverhalten zu rechnen, so dass ein solcher Über- oder Unterlauf unbedingt vermieden bzw. rechtzeitig diagnostiziert werden muss.

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 Zufallszahlgeneratoren 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:

2147483647 + 5 = 2147483652

Im letzten 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;
```

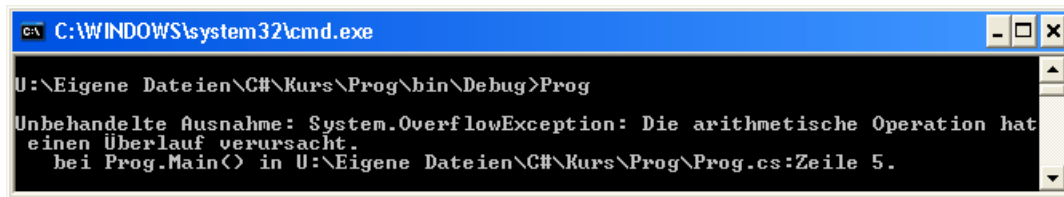
bleibt das Ergebnis falsch, denn ...

- Bei der Wertzuweisung
 $k = i + j$
wird zunächst der Ausdruck 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:

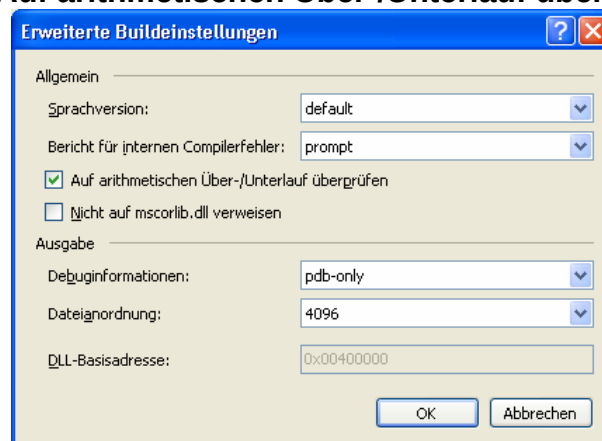
```
checked {
    . . .
    k = i + j;
    . . .
}
```

- **checked**-Compileroption
Man kann die Überwachung für einen kompletten Compiler-Lauf einschalten.
Beispiel:

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

Für ein Projekt der Visual C# 2005 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**



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 weder zu einem sinnlosen Zufalls- wert noch zu einem Ausnahmefehler, sondern zum speziellen Gleitkommawert +/- **Unendlich**, mit dem anschließend (im Rahmen des mathematisch vertretbaren) 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("13.0/0.0 =\t\t" + (13.0 / 0.0) + "");
    }
}

```

liefert die Ausgabe:

```

Double.MaxValue =      1,79769313486232E+308
Double.MaxValue * 10 = +unendlich
Unendl. + 10 =         +unendlich
13.0/0.0 =             +unendlich

```

Mit Hilfe der Unendlich-Werte „gelingt“ offenbar sogar die Division durch 0.

C# weigert sich aber zu Recht, „Unendlich - Unendlich“ zu berechnen. Hier resultiert wie bei der Division von 0 durch 0 ein **undefinierter Wert**, wie das folgende Programm zeigt:

```

using System;
class Prog {
    static void Main() {
        double bigd;
        bigd = Double.MaxValue * 10.0;
        Console.WriteLine("Unendl. - Unendl. =\t" + (bigd - bigd));
        Console.WriteLine("0.0 / 0.0 =\t\t" + (0.0 / 0.0));
    }
}

```

Es liefert die Ausgabe:

```

Unendl. - Unendl. =      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.:

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

Quellcode	Ausgabe
<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

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

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

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:

```
C:\WINDOWS\system32\cmd.exe

U:\Eigene Dateien\C#\Kurs\Prog\bin\Debug>Prog
Unbehandelte Ausnahme: System.OverflowException: Der Wert für eine Dezimalzahl war zu groß oder zu klein.
   bei System.Decimal.FCallMultiply<Decimal& result, Decimal d1, Decimal d2>
   bei System.Decimal.Multiply<Decimal d1, Decimal d2>
   bei System.Decimal.op_Multiply<Decimal d1, Decimal d2>
   bei Prog.Main() in U:\Eigene Dateien\C#\Kurs\Prog\Prog.cs:Zeile 5.
```

3.6.4 Unterlauf bei den Gleitkommatypen

Bei den Gleitkommatypen **float**, **double** und **decimal** ist auch ein **Unterlauf** möglich, wobei eine Zahl mit sehr kleinem Betrag (bei **double**: $< 4,94065645841247 \cdot 10^{-324}$) nicht mehr dargestellt werden kann. In diesem Fall rechnet ein C# - Programm mit dem Wert 0,0 weiter, was in der Regel akzeptabel ist, z.B.:

Quellcode	Ausgabe
<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.

3.7 Anweisungen

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.

Unsere Beispielprogramme bestanden im Wesentlichen aus einer **Sequenz** von **Anweisungen**, die wir nach bedarfsgerecht eingeführten Regeln gebildet haben. Nun werden wir uns systematisch mit dem Begriff einer C# - Anweisung und vor allem mit wichtigen Spezialfällen (bedingte Anweisung, Fallunterscheidung, Schleifen) befassen.

3.7.1 Überblick

Ausführbare Programmteile, die in C# stets als Methoden von Klassen zu realisieren sind, bestehen aus Anweisungen.

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

- **Variablendeklarationsanweisungen**

Die Variablendeklarationsanweisung wurde schon in Abschnitt 3.3.4 eingeführt.

Beispiel: `int i = 1, k;`

- **Ausdrucksanweisungen**

Folgende Ausdrücke werden zu Anweisungen, sobald man ein Semikolon dahinter setzt:

- o **Wertzuweisungen** (vgl. Abschnitte 3.3.4 und 3.5.7)

Beispiel: `k = i + j;`

- o **Prä- bzw. Postinkrement- oder -dekrementoperationen**

Beispiel: `i++;`

Hier ist nur der „Nebeneffekt“ des Ausdrucks `i++` von Bedeutung. Sein Wert bleibt ungenutzt.

- o **Methodenaufrufe**

Beispiel: `Console.WriteLine(cond);`

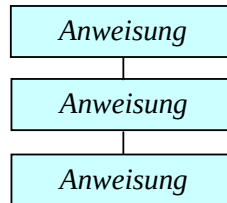
- **Leere Anweisung**

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.

- **Anweisungen zur Ablaufsteuerung**

Die Methoden der bisherigen Beispielpprogramme in Abschnitt 3 bestanden aus einer *Sequenz* von Anweisungen, die bei jedem Programmlauf komplett und linear durchlaufen wurde:



Oft möchte man jedoch z.B.

- die Ausführung einer Anweisung oder eines Anweisungsblocks von einer *Bedingung* abhängig machen
- oder 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**).

- **Blockanweisung**

Eine Folge von Anweisungen, die durch geschweifte Klammern zusammengefasst bzw. abgegrenzt werden, bildet eine **Verbund- bzw. Blockanweisung**. In Abschnitt 3.3.5 haben wir uns im Zusammenhang mit dem Deklarationsbereich von lokalen Variablen bereits mit Anweisungsblöcken beschäftigt.

Wie gleich näher erläutert wird, fasst man z.B. *dann* mehrere Anweisungen 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.

Als *zusammengesetzt* bezeichnet man Anweisungen, die andere Anweisungen enthalten. Dies betrifft neben den Blockanweisungen vor allem die (auch als *Kontrollstrukturen* bezeichneten) Anweisungen zur Ablaufsteuerung.

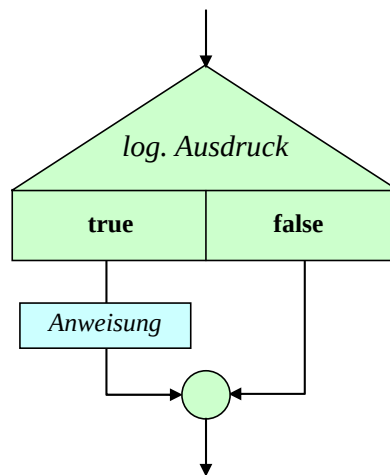
Anweisungen werden durch ein Semikolon abgeschlossen, sofern sie nicht mit einer schließenden Blockklammer enden.

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.

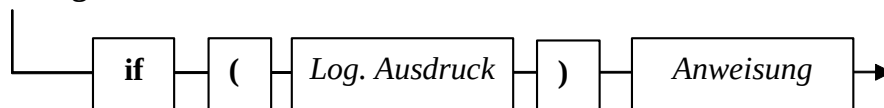
3.7.2.1 if-Anweisung

Soll eine (Block-)Anweisung nur dann ausgeführt werden, wenn ein logischer Ausdruck den Wert **true** annimmt,



verwendet man die **if**-Anweisung mit folgender Syntax:

Bedingte Anweisung



In diesem Beispiel wird eine Meldung ausgegeben, wenn die Variable `anz` den Wert 0 besitzt:

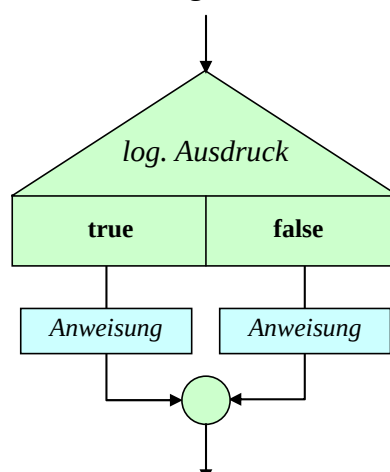
```
if (anz == 0)
    Console.WriteLine("Die Anzahl muss > 0 sein!");
```

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, z.B. in der Methode `Kuerze()` aus dem Bruchrechnungsbeispiel:

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

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:

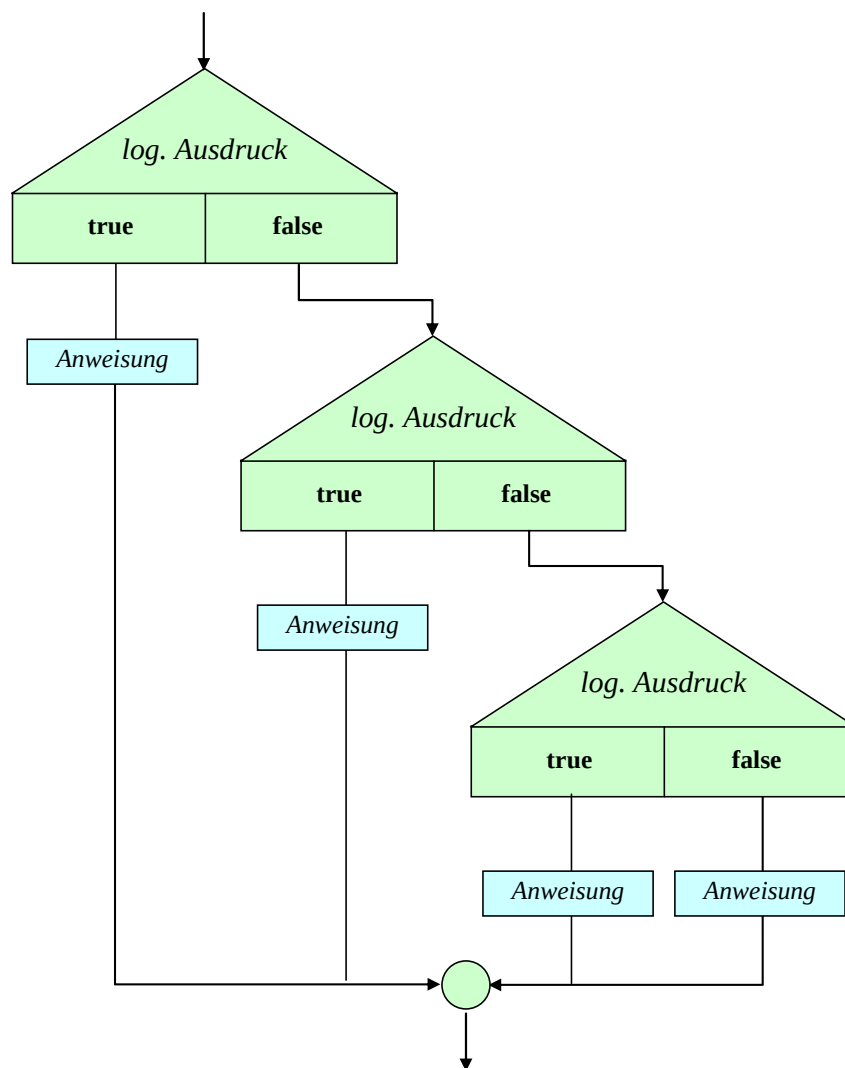
- Für **terminale Sprachbestandteile**, die exakt in der angegebenen Form in konkreten Quellcode zu übernehmen sind, wird **fette** Schrift verwendet.
- *Platzhalter* sind durch *kursive* Schrift gekennzeichnet.

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.

Im folgenden **if-else** - Beispiel wird der natürliche Logarithmus zu einer Zahl geliefert, falls diese positiv ist. Anderenfalls erscheint eine Fehlermeldung mit Alarmton (Escape-Sequenz \a). Das Argument wird vom Benutzer über eine **ToDouble-ReadLine** - Konstruktion erfragt (vgl. Abschnitt 3.4.1).

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { Console.Write("Argument: "); double arg = Convert.ToDouble(Console.ReadLine()); if (arg > 0) Console.WriteLine("ln(" + arg + ") = " + Math.Log(arg)); else Console.WriteLine("\aArgument <= 0!"); } }</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 3.7.2.2 vorzustellende **switch**-Anweisung, die bei Mehrfach-Fallunterscheidungen gegenüber einer verschachtelten **if-else** – Konstruktion zu bevorzugen ist, wenn die Fallzuordnung über die verschiedenen Werte *eines* Ausdrucks (z.B. vom Typ **int**) erfolgen kann.

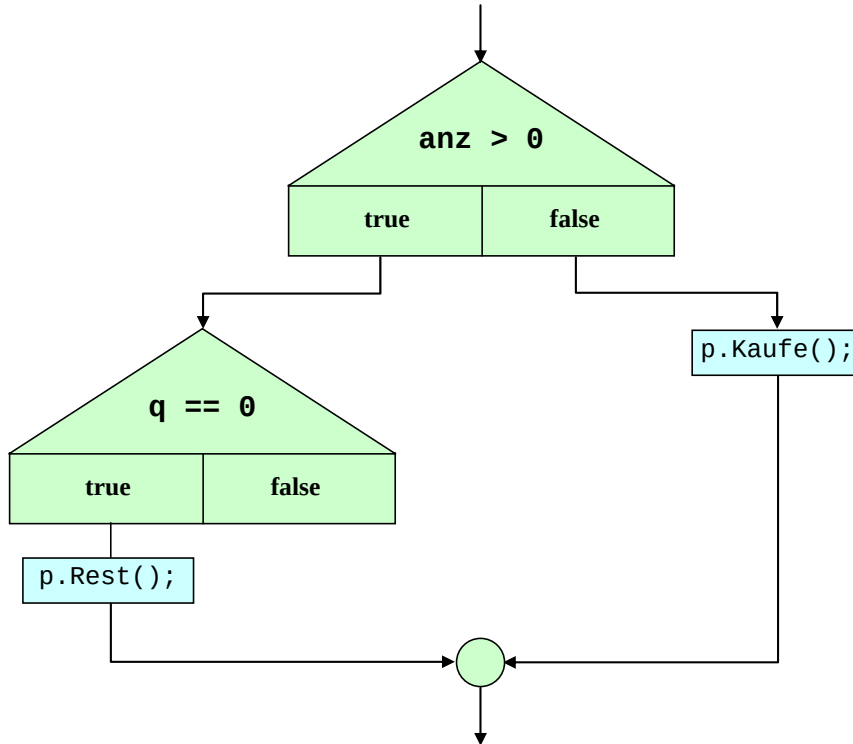
Beim Schachteln von bedingten Anweisungen kann es zum so genannten **dangling-else** - Problem kommen, wobei ein Missverständnis zwischen Compiler und Programmierer hinsichtlich der Zuordnung einer **else**-Klausel besteht. Im folgenden Codefragment

```

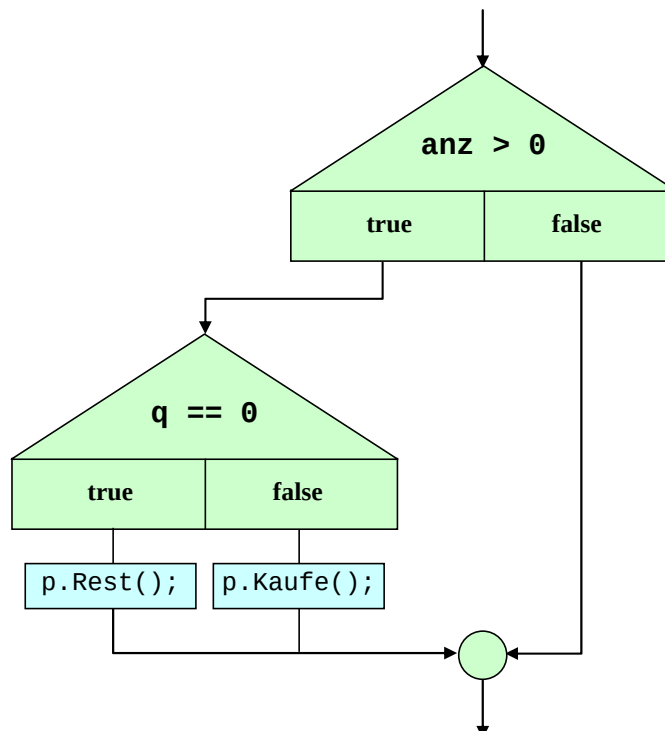
if (anz > 0)
  if (q == 0)
    p.Rest();
else
  p.Kaufe();

```

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 bezieht er die **else**-Klausel also auf die *zweite* **if**-Anweisung, so dass de facto folgender Programmablauf resultiert:



Mit Hilfe von Blockklammern (nötigenfalls auch für eine einzelne Anweisung) kann die gewünschte Zuordnung erzwungen werden:

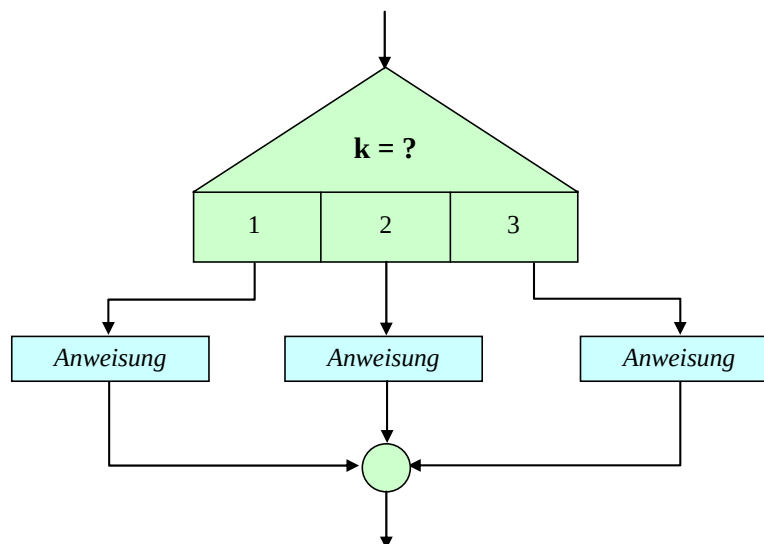
```
if (i > 0)
    {if (j > i) k = j;}
else
    k = i;
```

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 = i;
```

3.7.2.2 switch-Anweisung

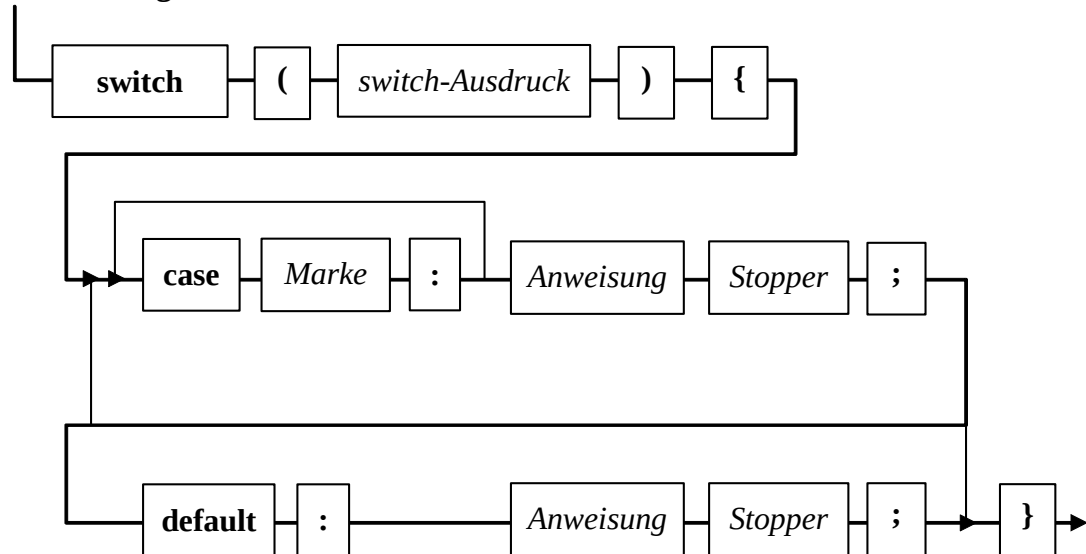
Wenn eine Fallunterscheidung mit mehr als zwei Alternativen in Abhängigkeit vom Wert *eines* Ausdrucks vorgenommen werden soll,



dann ist die Mehrfachauswahl per **switch**-Anweisung weitaus handlicher als eine verschachtelte **if-else** - Konstruktion. In Bezug auf den Datentyp des steuernden Ausdrucks ist C# im Vergleich zu anderen Sprachen sehr flexibel und erlaubt:

- numerische Datentypen
Dazu gehört auch der *integrale* Datentyp **char** (vgl. Abschnitt 3.3.2).
- 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 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:

```

using System;
class Prog {
    static void Main() {
        int zahl = 2;
        switch (zahl) {
            case 1: Console.WriteLine("Fall 1 (mit break-Stopper)");
                    break;
            case 2: Console.WriteLine("Fall 2 (mit Durchfall per goto)");
                    goto case 3;
            case 3: Console.WriteLine("Fall 3");
                    break;
            default: Console.WriteLine("Restkategorie");
                    break;
        }
    }
}
  
```

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

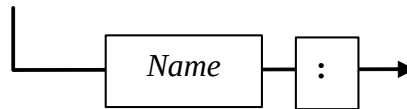
Stimmt beim Ablauf des Programms der Wert des **switch**-Ausdrucks mit einer **case**-Marke überein, dann wird die zugehörige Anweisung ausgeführt, ansonsten (falls vorhanden) die **default**-Anweisung. Es ist auch erlaubt, zu einer Anweisung *mehrere* **case**-Marken anzugeben.

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 dem Schlüsselwort **break**, wobei die Methode hinter der **switch**-Anweisung fortgesetzt wird. Mögliche Alternativen sind:

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

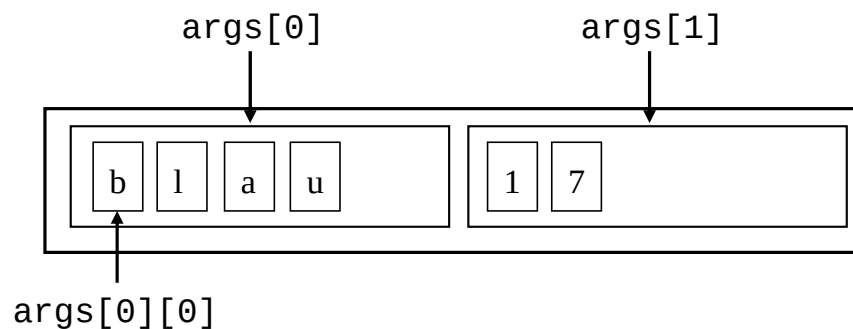
Wer z.B. die Farbe Blau und die Zahl 17 bevorzugt, muss das Programm also folgendermaßen starten:

```
PerST b 17
```

Im Quellcode wird jeweils nur *eine* Anweisung benötigt, um die (vom Benutzer durch Leerzeichen getrennten) Befehlszeilenargumente auszuwerten und das Ergebnis in eine 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 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. "Hallo").
- In der Parameterliste einer Methode kann die gewünschte Arbeitsweise näher spezifiziert werden.
- Besitzt die **Main()**-Methode einer Startklasse einen (ersten und einzigen) Parameter vom Datentyp **string[]** (Array mit **string**-Elementen), dann übergibt das .NET – Laufzeitsystem dieser Methode als **string**-Elemente die Spezifikationen, die der Anwender beim Start hinter den Programmnamen in die Kommandozeile geschrieben hat. Der Datentyp des Parameters ist fest vorgegeben, sein Name jedoch frei wählbar (im Beispiel: **args**). In der Methode **Main()** kann man auf **args** lesend genauso zugreifen wie auf lokale Variablen.
- Das erste Befehlszeilenargument landet im ersten Element des Zeichenketten-Arrays **args** und wird mit **args[0]** angesprochen, weil Array-Elemente mit 0 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 0 vom Array-Element Nummer 0) angesprochen und der **char**-Variablen **farbe** zugewiesen wird.
- Das zweite Element des Zeichenketten-Arrays **args** (mit der Nummer 1) 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:



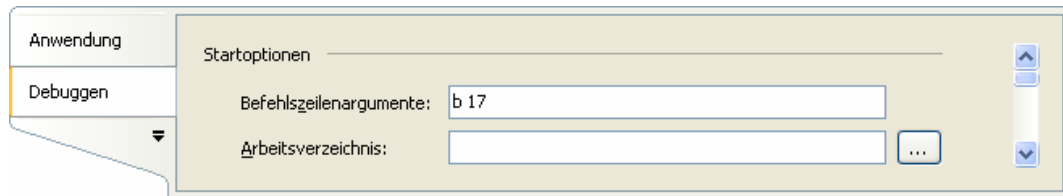
Ansonsten ist im Beispielprogramm noch die **return**-Anweisung von Interesse, welche die **Main()**-Methode und damit das Programm beendet:

```
return;
```

Sie ist uns oben schon als Stopper in der **switch**-Anweisung begegnet. Sie werden diese Anweisung im Zusammenhang mit der ausführlichen Behandlung der Methoden noch näher kennen lernen.

Soll das Programm in der Visual C# 2005 Express Edition gestartet werden, müssen die Befehlszeilenargumente folgendermaßen vereinbart werden:

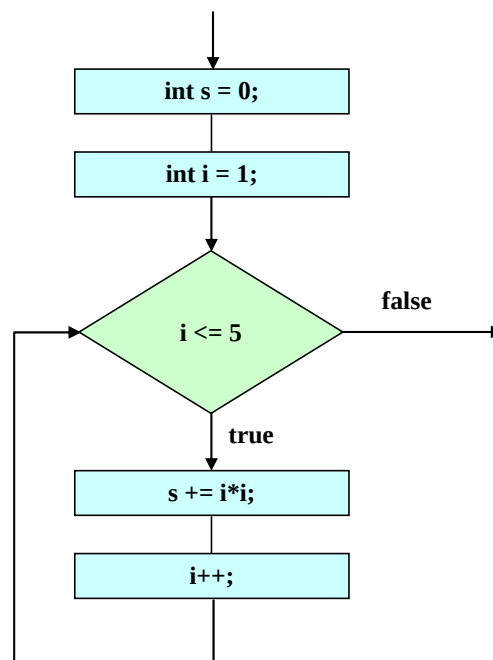
- Menübefehl **Projekt > Eigenschaften**
- Registerkarte **Debuggen**
- **Befehlszeilenargumente** eintragen, z.B.:



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.

In der folgenden Abbildung ist ein iterativer Algorithmus zu sehen, der die Summe der quadrierten natürlichen Zahlen von 1 bis 5 berechnet:



C# bietet verschiedene Wiederholungsanweisungen, die sich bei der Ablaufsteuerung unterscheiden und später im Detail vorgestellt werden:

- **Zählergesteuerte Schleife (for)**
Die Anzahl der Wiederholungen steht typischerweise schon vor Schleifenbeginn fest. Zur Ablaufsteuerung wird am Anfang eine Zählvariable initialisiert und dann bei jedem Durchlauf aktualisiert (z.B. inkrementiert). 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 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 ein weiteres mal ausführen
- o **false**: Schleife beenden

Bei der *kopf*gesteuerten **while**-Schleife wird die Bedingung *vor Beginn* eines Durchgangs geprüft, bei der *fuß*gesteuerten **do**-Schleife hingegen *am Ende*.

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 der von runden Klammern umgebene Schleifenkopf, wo die Vorbereitung der Indexvariablen, die Spezifikation der Fortsetzungsbedingung und die Aktualisierungsvorschrift untergebracht werden können. Am Ende steht die wiederholt auszuführende (Verbund-)Anweisung:

for (Vorbereitung; Bedingung; Aktualisierung)
Anweisung

Zu den drei Bestandteilen des Schleifenkopfes 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. Diese Variable wird im Schleifenkopf auf einen Startwert gesetzt und nötigenfalls dabei auch deklariert, z.B.:

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { double sq = 0.0; for (int i = 1; i <= 5; i++) sq += i*i; Console.WriteLine("Quadratsumme = " + sq); } }</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**

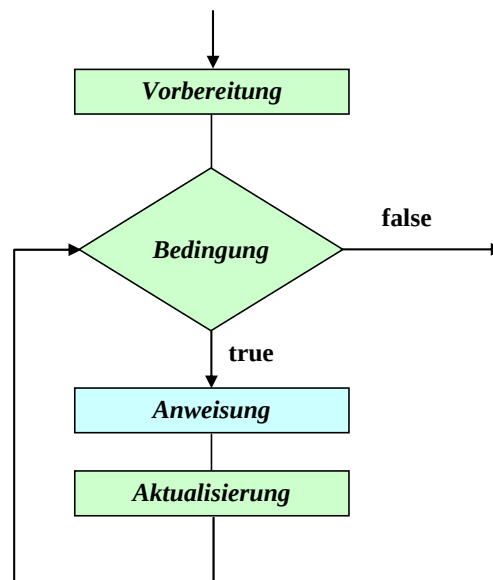
Üblicherweise wird eine Ober- oder Untergrenze für die Indexvariable gesetzt, doch erlaubt C# beliebige logische Ausdrücke.

Die Bedingung wird *zu Beginn jedes Schleifendurchgangs* 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.

In der folgenden Abbildung sind die semantischen Regeln zur **for**-Schleife dargestellt:



Im Unterschied zur obigen Algorithmus-Darstellung (Summe der quadrierten natürlichen Zahlen) sind an der grünen Farbe nun die Bestandteile des Schleifenkopfs zu erkennen.

Zu den (zumindest stilistisch) bedenklichen Konstruktionen, die der Compiler klaglos umsetzt, gehören **for**-Schleifenköpfe ohne Initialisierung 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 in der Schleifenanweisung eine Indexvariable modifiziert wird.

3.7.3.2 Iterieren über Kollektionen (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.
- Die Anweisung der **foreach**-Schleife wird nacheinander für jedes Element der Kollektion ausgeführt.
- Im Schleifenkopf wird eine Variable vom Datentyp der Kollektionselemente definiert, über die in der Schleifenanweisung das aktuelle Element angesprochen werden kann.

Die allgemeine Struktur der **foreach**-Schleife:

foreach (*ElementTyp Name in Kollektion*)
Anweisung

Das Beispielprogramm PerST in Abschnitt 3.7.2.2 hat demonstriert, wie man über einen Parameter der Methode **Main()** auf die Zeichenfolgen zugreifen kann, welche der Benutzer beim Start eines Konsolenprogramms hinter den Assembly-Namen geschrieben hat. In folgendem Programm wird durch zwei geschachtelte **foreach**-Schleifen für jedes Element im **string**-Array args mit den Befehlszeilenargumenten folgendes getan:

- Die Zeichenfolge wird ausgegeben.
- In einer inneren **foreach**-Schleife wird jedes Zeichen einzeln 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).

Man darf auf die Iterationsvariable nur *lesend* zugreifen, so dass in der inneren **foreach** - Schleife des letzten Beispiels die folgende Konstruktion *nicht* erlaubt 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.

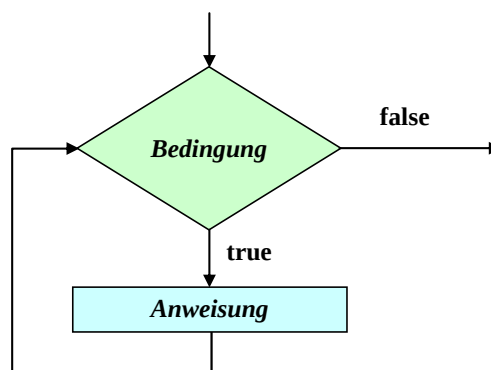
Unter der aktuellen Abschnittsüberschrift diskutiert man traditionsgemäß die **while**- und die **do**-Schleife.

3.7.3.3.1 while-Schleife

Die **while**-Anweisung kann als vereinfachte **for**-Anweisung beschreiben kann: Wer im Kopf einer **for**-Schleife auf Initialisierung 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 zu *Beginn* eines Schleifendurchgangs geprüft. Resultiert der Wert **true**, so wird der Anweisungsteil ausgeführt, anderenfalls wird die **while**-Schleife verlassen, eventuell noch vor dem ersten Durchgang:

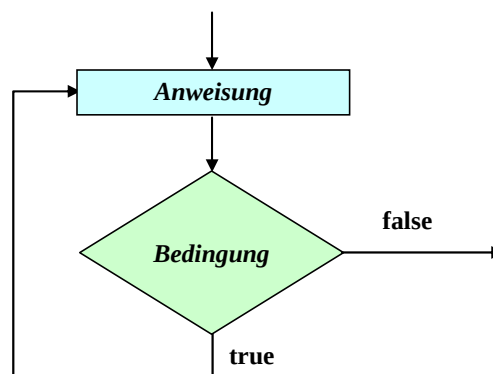


Im obigen Beispielprogramm zur Quadratsummenberechnung (vgl. Abschnitt 3.7.3.1) kann man die **for**-Schleife leicht durch eine **while**-Schleife ersetzen:

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { double sq = 0.0; int i = 1; while (i <= 5) { sq += i * i; i++; } Console.WriteLine("Quadratsumme = " + sq); } }</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 (leider) auch in der Syntax zur **do**-Schleife auf:

do <i>Anweisung</i> while (<i>Bedingung</i>);

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 geholt. Weil dieser Methodenaufruf ein Ausdruck vom Typ **string** ist, kann das erste Zeichen (mit der Nummer 0) per Indexzugriff (mit Hilfe der eckigen Klammern, siehe unten) 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>	Beenden? (j/n)? r Beenden? (j/n)? 4 Beenden? (j/n)? j

Bei einer **do**-Schleife mit Anweisungsblock sollte man die **while**-Klausel unmittelbar hinter die schließende Blockklammer setzen (in derselben 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 eingesetzt. 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 Schleifendurchgang wird per **continue** verlassen.

Mit der Eingabe 0 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, doch werden bei dem vorgeschlagenen Verfahren lästige Sonderfälle (unzulässige Werte, 0 als Terminierungssignal) auf besonders übersichtliche Weise abgehandelt, bevor der Kernalgorithmus startet.

Zum Kernalgorithmus der Primzahlendiagnose sollte vielleicht noch erläutert werden, warum die Suche nach einem Teiler des Primzahlkandidaten bei dessen Wurzel enden kann:

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

$$z = k \cdot d$$

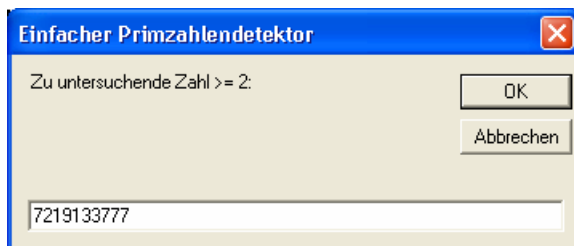
Dann ist auch k ein Teiler von z , und es gilt:

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

Anderenfalls wäre das Produkt $k \cdot d$ ja größer als z . Wir haben also folgendes Ergebnis: Wenn eine Zahl z keinen Teiler kleiner oder gleich $\sqrt{z}$ besitzt, dann ist es eine Primzahl.

Zur Berechnung der Wurzel verwendet das Beispielprogramm die Methode **Sqrt()** aus der Klasse **Math**, über die man sich bei Bedarf in der SDK-Dokumentation informieren kann.

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



Hier kommen die Methoden **MessageBox.Show()** (vgl. Abschnitt 2.5) und **Interaction.InputBox()** (vgl. Abschnitt 3.4.2) zum Einsatz. Sie finden das zugehörige Projekt 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) Erstellen Sie eine Variante des Primzahlen-Testprogramms aus Abschnitt 3.7.3.5, die ohne **break** bzw. **continue** auskommt.

3) Verbessern Sie das im Übungsaufgaben-Abschnitt 3.5.10 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.

4) Bei einem **double**-Wert sind maximal 16 signifikante Dezimalstellen garantiert (siehe Abschnitt 3.3.2). Folglich kann ein Rechner die **double**-Werte $1,0$ und $1,0 + 0,5^i$ ab einem bestimmten Exponenten i nicht mehr voneinander unterscheiden. Bestimmen Sie mit einem Testprogramm den größten ganzzahligen Index i , für den man noch erhält:

$$1,0 + 0,5^i > 1,0$$

5) 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: Für natürliche Zahlen u und v mit $u > v$ gilt:

$$\begin{array}{c} x \text{ ist gemeinsamer Teiler von } u \text{ und } v \\ \Leftrightarrow \\ x \text{ ist gemeinsamer Teiler von } u \text{ und } u \% v \end{array}$$

Der ggT-Algorithmus per Modulo-Operation läuft 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 $u \% v$

und startet das Verfahren neu.

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 (PC: Pentium 4 mit 2,26 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:	62,3844 Millisek.	Benöt. Zeit:	0 Millisek.

4 Klassen und Objekte

Eine 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 Bauplan-Charakter gefragt.

Für eine Klasse (bzw. 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 FCL entstammende) oder selbst erstellte Klassen zur Erledigung von Aufgaben verwendet. In der Regel werden auch **Objekte** aus Klassen mit Bauplan-Charakter erzeugt und mit Aufträgen versorgt. Mit dem „Beauftragen“ eines Objektes 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

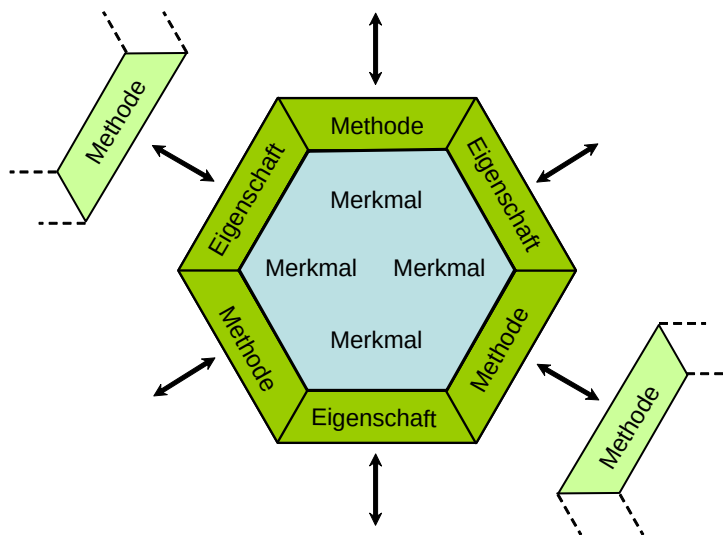
In diesem Abschnitt werden einige Prinzipien ausgespart, die mit den bisherigen Beispielen noch nicht plausibel vermittelt werden können (z.B. Polymorphie).

4.1.1.1 Datenkapselung

In der objektorientierten Programmierung (OOP) wird die traditionelle Trennung von Daten und Operationen aufgegeben. Hier besteht ein Programm aus **Klassen**, die durch Merkmale *und* Methoden (sowie weitere Bestandteile) definiert sind.

Eine Klasse wird in der Regel ihre Merkmale gegenüber anderen Klassen verbergen (**Datenkapselung, information hiding**) und so vor ungeschickten Zugriffen schützen. Von außen ansprechbar sind hingegen die Methoden und Eigenschaften¹ einer Klasse, welche ihre *Schnittstelle* bilden. Dies kommt in der folgenden Abbildung zum Ausdruck, die Sie im Wesentlichen schon aus Abschnitt 1 kennen:

¹ Wie Sie bereits aus dem Einleitungsbeispiel wissen, steht in C# eine Eigenschaft für ein Paar von Methoden zum Lesen bzw. Verändern eines Merkmals.



Aus der Datenkapselung ergeben sich gravierende Vorteile für die Softwareentwicklung:

- **Vermeidung von Programmierfehlern**

Direkte Zugriffe auf die Merkmale 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 0 gesetzt wird. Anwender unserer Klasse können einen Nenner einzig über die Eigenschaft `Nenner()` verändern, die aber den Wert 0 nicht akzeptiert. Treten in einem Programm trotz Datenkapselung Fehler wegen pathologischer Merkmalsausprägungen auf, sind diese relativ leicht zu lokalisieren, weil nur wenige Methoden verantwortlich sein können. Per Datenkapselung werden also die Kosten reduziert, die durch Programmierfehler und den Aufwand bei der Entfernung von Fehlern entstehen.

- **Komfort**

Wer als Programmierer eine Klasse *verwendet*, braucht sich um deren inneren Aufbau nicht zu kümmern, so dass neben dem Fehlriskiko auch der Einarbeitungsaufwand sinkt.

- **Flexibilität, Anpassungsfähigkeit**

Datenkapselung schafft günstige Voraussetzungen für die Wartung bzw. Verbesserung einer Klassendefinition. Solange die Methoden und Eigenschaften der Schnittstelle unverändert bzw. kompatibel bleiben, kann die interne Architektur einer Klasse ohne Nebenwirkungen auf andere Programmteile beliebig geändert werden.

- **Produktive 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 und Wiederverwendung von Software 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. Eine Klasse kann direkt genutzt werden (durch Erzeugen von Objekten) oder als Basisklasse dienen.

Im .NET – Framework wird das Vererbungsprinzip sogar auf die Spitze getrieben: Alle Klassen (und auch die Strukturen, s.u.) stammen von der Urahnklasse **Object** ab, die an der Spitze des hie-

rarchischen .NET - Klassensystems steht, das man als seiner Universalität wegen als **Common Type System** (CTS) bezeichnet.

Weil sich z.B. die Methode **ToString()** schon im Handlungsrepertoire der Urahnklasse befindet, liefert jedes .NET - Objekt und jede Struktur-Instanz (s.u.) nach dem Aufruf dieser Methode eine Zeichenfolge mit einer Selbstbeschreibung (z.B. Wertangabe) zurück. Das tut im folgenden Beispiel eine **double**-Variable:

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { double d = 1.5; Console.WriteLine(d.ToString()); Console.WriteLine(d); } }</pre>	<pre>1,5 1,5</pre>

Weil die Methode **Console.WriteLine()** für die übergebenen Parameter den **ToString()** – Aufruf automatisch vornimmt, können wir uns diesen Aufwand sparen (siehe zweiten Aufruf).

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 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 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 Unterprogrammammlung klassischer Art muss der Programmierer passende Daten bereitstellen, auf die dann vorgefertigte Funktionen 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 den *Strukturen* der Programmiersprache C oder den *Records* 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 Daten eines Falles lassen sich dann in *einer* Variablen vom selbst definierten Typ speichern. Dies vereinfacht z.B. das Lesen, Kopieren oder Schreiben solcher Daten.

Die Strukturen bzw. Records der älteren Programmiersprachen werden in C# durch *Klassen* ersetzt, wobei diese Datentypen nicht nur durch eine Anzahl von *Merkmale* (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 deutlich einfacher zu verwenden als klassische Unterprogramm-bibliotheken.

4.1.3 Auf-Bruch zu echter Klasse

In den Beispielprogrammen von Abschnitt 3 wurde mit den Klassendefinitionen lediglich eine in C# unausweichliche formale Anforderung an ausführbare Programmeinheiten erfüllt. Das in Abschnitt 1.1 vorgestellte Bruchrechnungsprogramm realisiert hingegen wichtige Prinzipien der objektorientierten Programmierung. Es wird nun wieder aufgegriffen und in verschiedenen Varianten bzw. Ausbaustufen als Beispiel verwendet.

Das Programm soll 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, Lernepisode, Testepisode, Fehler, Schüler) 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. 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*) für das Lesen und das (kontrollierte) Verändern des Feldes zur Verfügung.
- Wir erlauben uns einen 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 Rückgabotyp von **void** zu **bool**. Mit der **return**-Anweisung wird am Ende einer erfolgreichen Ausführung der Wert **true** oder nach dem Auftreten einer Ausnahme der Wert **false** zurückgemeldet.

Bei den unveränderten Methoden bzw. Eigenschaften ist der Quellcode verkürzt 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() {
        . . .
    }

    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 der Einleitung dieses Manuskriptes wird die Bruch-Klassen-
definition anschließend gründlich erläutert. Dabei machen die in Abschnitt 4.2 behandelten Instanz-

variablen 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.

Schon an dieser Stelle sollen einige Konventionen angesprochen bzw. in Erinnerung gerufen werden, die zu einem gut lesbaren und übersichtlich organisierten Quellcode führen:

- Der Klassenname **Bruch** beginnt einer allgemein akzeptierten C# - Konvention folgend mit einem Großbuchstaben (siehe Abschnitt 3.1.5).
- Hinsichtlich der Dateiverwaltung wird vorgeschlagen:
 - o Die **Bruch**-Klassendefinition sollte in einer eigenen Datei gespeichert werden.
 - o Den Namen dieser Datei sollte man aus dem Klassennamen durch Anhängen der Extension „.cs“ bilden.

4.2 Instanzvariablen (Felder)

Die Instanzvariablen (Felder) einer Klasse besitzen viele Gemeinsamkeiten mit den *lokalen* Variablen, die wir im Abschnitt über elementare Sprachelemente ausschließlich verwendet haben, doch gibt es auch wichtige Unterschiede, die im Mittelpunkt des aktuellen Abschnitts stehen.

Unsere Demoklasse **Bruch** besitzt nach der Erweiterung um ein beschreibendes Etikett folgende Instanzvariablen (Felder):

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

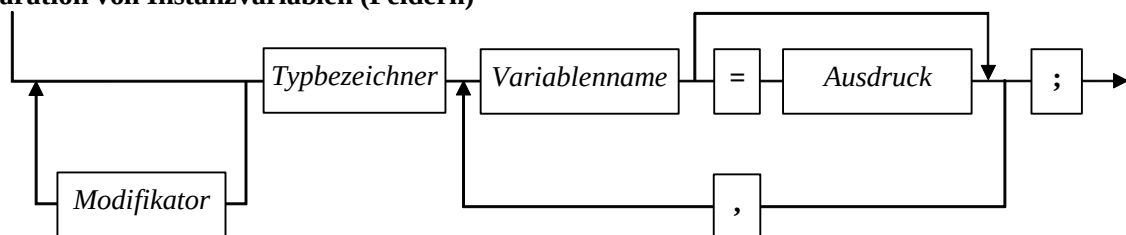
Jedes nach dem **Bruch**-Bauplan geschaffene Objekt erhält seine *eigene, komplette* Ausstattung mit diesen Variablen.

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.

Bei der Deklaration von Instanzvariablen werden zur Spezifikation der **Schutzstufe** (siehe unten) 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)



Bei den zulässigen Datentypen gibt es keine Unterschiede zwischen lokalen Variablen und Instanzvariablen.

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 Klassen können in die Eigenschaften `Zaehler` und `Nenner` neben der Null-Überwachung für den Nenner noch weitere Wert-Restriktionen einbauen (z.B. auf positive Zahlen).
- 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` ist der Spielraum für Schnittstellenkompatible Typanpassungen aber gering.

Trotz ihrer überzeugenden Vorteile soll die Datenkapselung nicht zum Dogma erhoben werden. Sie ist überflüssig, wenn bei einem Merkmal 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.

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** deklariert man *konstante* Felder von *elementarem* Datentyp, wobei der initialisierende Ausdruck *zur Übersetzungszeit* berechenbar sein muss, z.B.:

```
const int MAXZ = 1000;
```

Soll eine Instanzvariable (von *beliebigem* Typ) *zur Laufzeit* initialisiert und dann fixiert werden, verwendet man den Modifikator **readonly**, z.B.:

Quellcode	Ausgabe
<pre>using System; class Prog { readonly public int readomo; public Prog(int i) { readomo = i; } static void Main(string[] args) { Prog p = new Prog(Convert.ToInt32(args[0])); Console.WriteLine(p.readomo); //verboten: p.readomo = 13; } }</pre>	4711

Im Beispiel wird die Initialisierung durch einen so genannten *Konstruktor* (siehe unten) aufgrund eines Befehlszeilenarguments vorgenommen. Bei den Konstruktoren handelt es sich um spezielle Methoden, die den Namen ihrer Klasse tragen und keinen Rückgabotyp besitzen (auch nicht **void**).

4.2.3 Deklarationsbereich und Ablage im Hauptspeicher

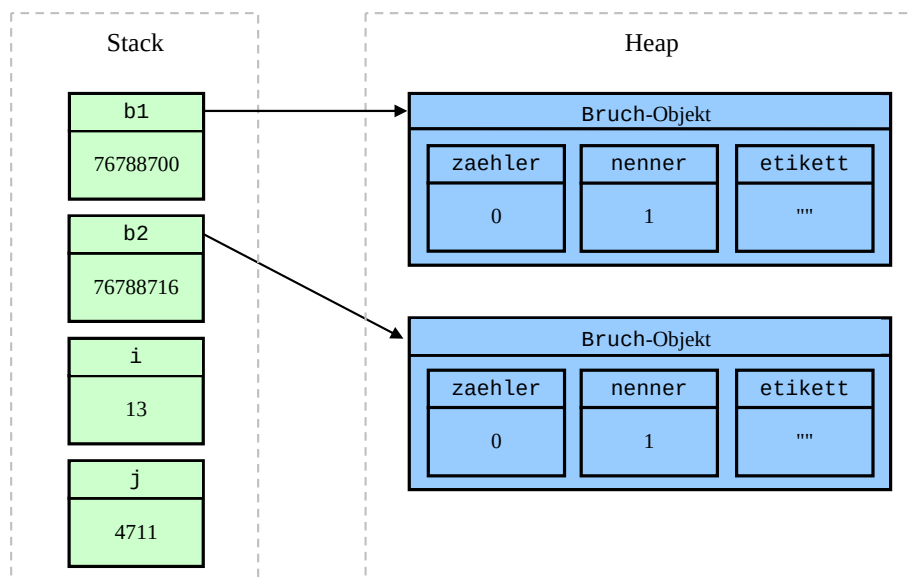
Von den lokalen Variablen einer Methode unterscheiden sich die Instanzvariablen einer Klasse u.a. in Bezug auf den Deklarationsbereich (Gültigkeitsbereich, engl.: *declaration space*) und die Ablage im Hauptspeicher:

- Eine lokale Variable existiert, während die Methode ausgeführt wird, genauer: bis zum Verlassen des lokalsten Blockes.
Sie wird auf dem so genannten **Stack** (deutsch: Stapel) abgelegt. Innerhalb des für ein Programm verfügbaren Speichers dient dieser Bereich zur Verwaltung von Methodenaufrufen.
- Für jedes neue Objekt wird ein Satz mit allen Instanzvariablen seiner Klasse erzeugt. Beim Ableben des Objektes wird der belegte Speicherplatz wieder frei gegeben.
Die Instanzvariablen eines neuen Objektes landen in einem Bereich des Programmspeichers, der als **Heap** (deutsch: Haufen) bezeichnet wird.

Während die folgende **Main()**-Methode

```
class BruchProg {
    static void Main() {
        Bruch b1 = new Bruch(), b2 = new Bruch();
        int i = 13, j = 4711;
        . . .
    }
}
```

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



4.2.4 Initialisierung

Während bei lokalen Variablen *keine* automatische Initialisierung stattfindet, erhalten die Instanzvariablen eines neuen Objektes automatisch folgende Startwerte, falls der Programmierer nicht eingreift:

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.

Datentyp	Initialisierung
sbyte, byte, short, ushort, int, uint, long, ulong	0
float, double, decimal	0,0
char	'\u0000'
bool	false
Referenztyp	null

Im Bruch-Beispiel wird nur die automatische `zaehler`-Initialisierung unverändert übernommen:

- Beim `nenner` eines Bruches wäre die Initialisierung auf 0 bedenklich, weshalb eine explizite Initialisierung auf den Wert 1 vorgenommen wird.
- Wie noch näher zu erläutern sein wird, ist **string** in C# *kein* 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 dies im Beispielprogramm zu Problemen führt, wird ein **string**-Objekt mit einer leeren Zeichenfolge erstellt. Das Erzeugen des **string**-Objekts erfolgt *implizit*, indem der **string**-Variablen `etikett` ein Zeichenfolgen-Literal zugewiesen wird.

4.2.5 Zugriff auf Instanzvariablen

In den Methoden 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** eine Referenz auf das handelnde Objekt voranzustellen (siehe Abschnitt 4.4.8).

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()` ersetzt das handelnde Objekt seinen `nenner`-Wert durch das Produkt aus diesem Wert und dem korrespondierende Wert eines anderen `Bruch`-Objektes, das über die Referenzvariable `b` angesprochen wird:

```
nenner = nenner*b.nenner;
```

Direkte Zugriffe auf die Instanzvariablen eines Objektes 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 syntaktische und eine semantische Regel kombiniert (und damit leicht unklar) 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 Methoden

In einer Bauplan-Klassendefinition werden Objekte entworfen, die eine Anzahl von Verhaltenskompetenzen (Methoden) besitzen, die von anderen Programmbestandteilen (Klassen oder Objekten) über festgelegte Kommunikationsregeln 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 dabei niemand etwas an (*information hiding*). Seine Instanzvariablen sind bei konsequenter Datenkapselung für Objekte (bzw. Methoden) fremder Klassen nicht direkt zugänglich. 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. Wie bei Instanzvariablen ist auch bei Methoden in C# die Schutzstufe **private** voreingestellt, die ausschließlich klasseneigenen Methoden (und Eigenschaften) eine Nutzung (d.h. einen Aufruf) erlaubt. Während man bei Instanzvariablen diese Voreinstellung meist belässt, ist sie bei allen Methoden zu ändern, die zur Schnittstelle einer Klasse gehören sollen.

Da es vom Verlauf einer Bildschirmausgabe nichts zu berichten gibt, liefert `Zeige()` keinen Rückgabewert. Folglich ist im Kopf der Methodendefinition der Typ **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.

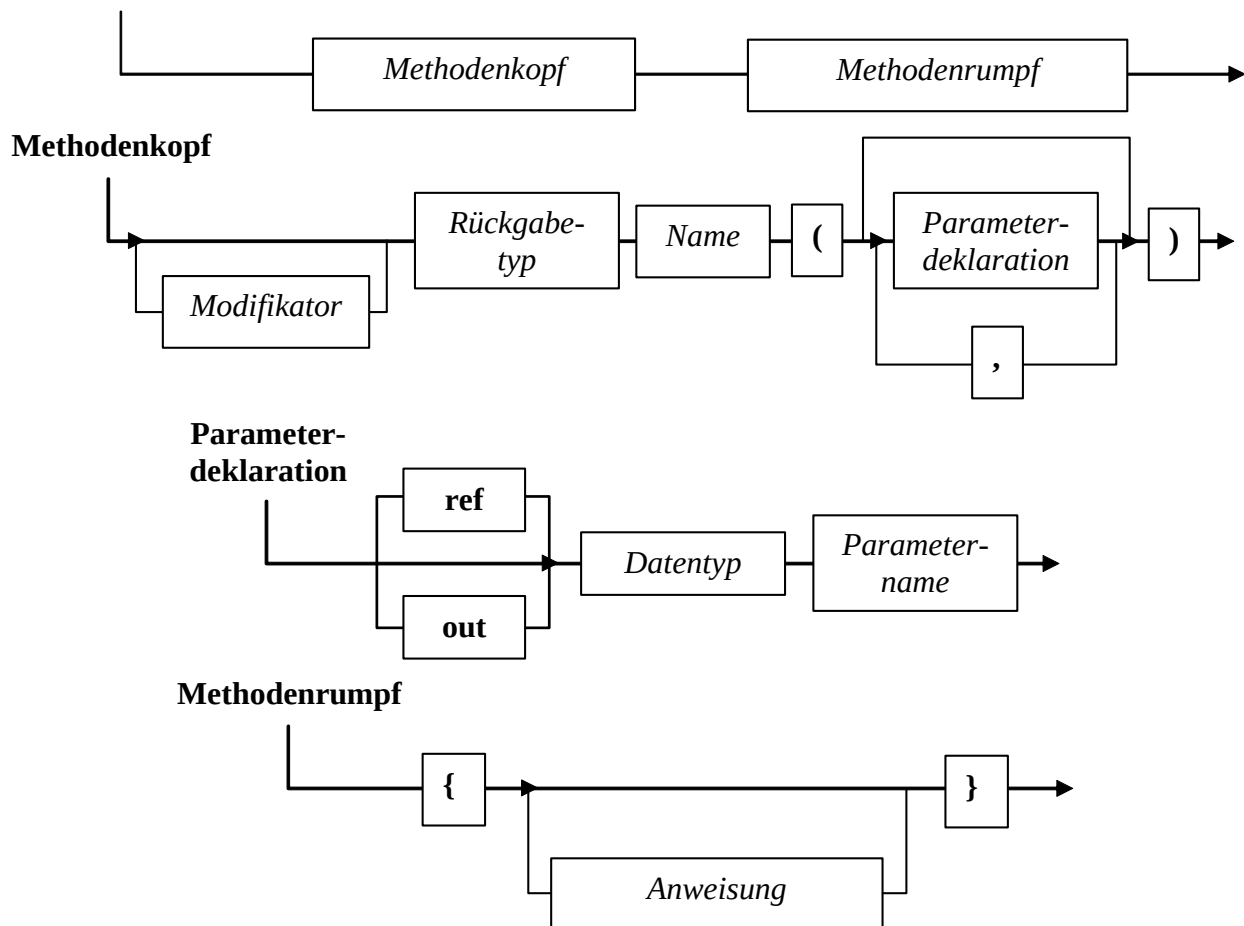
In der `Zeige()` - Implementierung wird die Länge des Etiketts zu einem Bruch über die **string**-Eigenschaft **Length** ermittelt. Damit gelingt eine brauchbar formatierte Konsolenausgabe von etikettierten Brüchen, z.B.:

Erster Bruch: $\frac{13}{221}$

Während jedes Objekt einer Klasse seine eigenen Instanzvariablen auf dem Heap besitzt, sind die Methoden einer geladenen Klasse nur *einmal* im Speicher vorhanden.

Die schon im Abschnitt 3.1.2.2 begonnene Serie von Syntaxdiagrammen zur Methodendefinition soll nun durch eine Explikation des Begriffs *Parameterdeklaration* vervollständigt werden:

Methodendefinition



Anschließend werden die (mehr oder weniger) neuen Bestandteile dieser Syntaxdiagramme erläutert.

4.3.1.1 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, wobei je nach Parameterart Variablen oder Ausdrücke in Frage kommen.

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 linken Nachbarn übereinstimmt.
- **Name**
Nach den Empfehlungen aus Abschnitt 3.1.5 ist bei Parameternamen ein kleiner Anfangsbuchstabe zu verwenden. Damit sich die Parameternamen von anderen Bezeichnungen (z.B. Namen von Instanzvariablen) unterscheiden, hängen manche Programmierer ein Suffix an, z.B. *par* oder einen Unterstrich.
- **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.

4.3.1.1.1 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 void Addiere(int zpar, int npar, bool autokurz) {
    zaehler = zaehler*npar + zpar*nenner;
    nenner = nenner*npar;
    if (autokurz)
        Kuerze();
}
```

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 wird.

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 { static void WertParDemo(int ipar) { Console.WriteLine(++ipar); } static void Main() { int iM = 4711; WertParDemo(iM); Console.WriteLine(iM); } }</pre>	4712 4711

Damit die Prog-Methode `WertParDemo()` in der statischen Prog-Methode **Main()** aufgerufen werden kann, muss sie ebenfalls statisch sein. Mit statischen Methoden werden wir uns bald (in Abschnitt 4.6.2) systematisch beschäftigen.

Als Wertaktualparameter sind nicht nur (initialisierte!) Variablen erlaubt, sondern beliebige Ausdrücke mit kompatibelem Typ.

4.3.1.1.2 Referenzparameter

Ein Referenzparameter ermöglicht es der aufgerufenen Methode, eine Variable der aufrufenden Programmeinheit zu *verändern*. Die Methode erhält beim Aufruf keine *Kopie* der betroffenen Variablen, sondern die Speicheradresse des Originals. Alle methodenintern über den Formalparameternamen vorgenommenen Modifikationen wirken sich direkt auf das Original aus.

Als Referenz-Aktualparameter sind nur *Variablen* erlaubt, die vom selben Typ wie der Formalparameter und außerdem initialisiert sein müssen. Wegen der letzten Forderung kann die aufgerufene Methode die übergebenen Aktualparameterwerte (vor einer möglichen Veränderung) auch als Informationen nutzen. Im Unterschied zu den Wert- bzw. Eingabeparametern und den bald vorzustellenden Ausgabeparametern ermöglichen Referenzparameter also einen Informationsfluss in *beide* Richtungen, so dass man sie auch als *Transferparameter* bezeichnen könnte.

In der Methodendefinition *und* beim Methodenaufruf sind Referenzparameter durch das Schlüsselwort **ref** zu kennzeichnen.

Zur Demonstration betrachten wir wie in Abschnitt 4.3.1.1.1 eine Klasse mit zwei statischen Methoden:

Quellcode	Ausgabe
<pre>using System; class Prog { static void Tausche(ref int a, ref int b) { int temp = a; a = b; b = temp; } static void Main() { int x = 1, y = 2; Console.WriteLine("Vorher: x = {0}, y = {1}", x, y); Tausche(ref x, ref y); Console.WriteLine("Nachher: x = {0}, y = {1}", x, y); } }</pre>	Vorher: x = 1, y = 2 Nachher: x = 2, y = 1

Abschließend noch ein Satz für Begriffsakrobaten: Wird eine *Referenzvariable*, die auf ein Objekt zeigt, als Referenz-Aktualparameter übergeben, kann man methodenintern nicht nur auf das Objekt

zugreifen, sondern auch den Inhalt der Referenzvariablen ändern, so dass sie z.B. anschließend auf ein anderes Objekt zeigt.

Es ist nur selten sinnvoll, bei einer Methodendefinition Referenzparameter mit Referenztyp (vom Typ einer Klasse) zu verwenden. Erlaubt ist diese für Anfänger recht verwirrende Doppelreferenz jedoch (vgl. Abschnitt 4.4.6).

4.3.1.1.3 Ausgabeparameter

Auch über Ausgabeparameter kann man einer Methode die Veränderung von Variablen der rufenden Programmeinheit ermöglichen. Der Compiler interessiert sich aber *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 { static 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() { int x, y; Lies(out x, out y); Console.WriteLine("\nx % y = " + (x % y)); } }</pre>	<pre>x = 29 y = 5 x % y = 4</pre>

4.3.1.1.4 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 { static void PriSum(params double[] args) { double summe = 0.0; foreach (double arg in args) summe += arg; Console.WriteLine("Die Summe ist = " + summe); } static void Main() { PriSum(1.2, 1.0); PriSum(1.2, 1.0, 3.6); } }</pre>	<pre>Die Summe ist = 2,2 Die Summe ist = 5,8</pre>

4.3.1.2 Modifikatoren

Bei Methoden gilt wie bei Instanzvariablen für den **Zugriffsschutz**:

- 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.

Im Bruch-Beispiel sind alle Methoden für die Verwendung durch fremde Klassen frei gegeben.

Nicht alle Modifikatoren im Kopf einer Methodendefinition dienen dem Zugriffsschutz. In den letzten Beispielen wurden mehrere Methoden mit dem Schlüsselwort **static** als Klassenmethoden (siehe Abschnitt 4.6) gekennzeichnet. Diese werden von der Klasse selbst ausgeführt, setzen also kein Objekt der Klasse als Ansprechpartner voraus. Wie Sie bereits wissen, muss die **Main()**-Methode einer startfähigen Klasse grundsätzlich als **static** deklariert werden.

4.3.1.3 Rückgabewerte und return-Anweisung

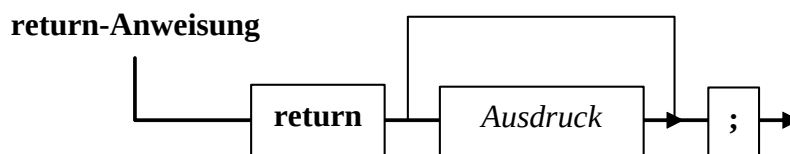
Für den Informationstransfer von einer Methode an ihren Aufrufer kann neben Referenz- und Ausgabeparametern auch ein Rückgabewert genutzt werden. Hier ist man auf einen einzigen Wert (von beliebigem Typ) beschränkt, doch lässt sich die Übergabe sehr elegant in den Programmablauf integrieren.

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 brauchbare 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 Rückgabetyt 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.



Bei Methoden ohne Rückgabewert ist die **return**-Anweisung nicht unbedingt erforderlich, kann jedoch (in der Variante ohne *Ausdruck*) dazu verwendet werden, die Methode vorzeitig zu beenden (z.B. im Rahmen einer bedingten Anweisung). Wir haben in früheren Beispielen diesen Abgang gelegentlich Gebrauch benutzt.

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**.

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.

Die in Abschnitt 4.3.1.3 als Beispiel betrachtete Bruch-Methode `Frage()` versorgt zwei Instanzvariablen via Konsoleneingabe mit Werten. Im Allgemeinen wird eine Methode etwas mehr Aktivität entfalten und sich dabei auch nicht unbedingt auf die Veränderung von lokalen Variablen und objekt-eigenen Instanzvariablen beschränken. Über eine als Parameter übergebene Referenzvariable (siehe Abschnitt 4.4.6) kann z.B. ein Objekt aus einer beliebigen Klasse per Methodenaufruf angesprochen werden.

Es kommt 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.:

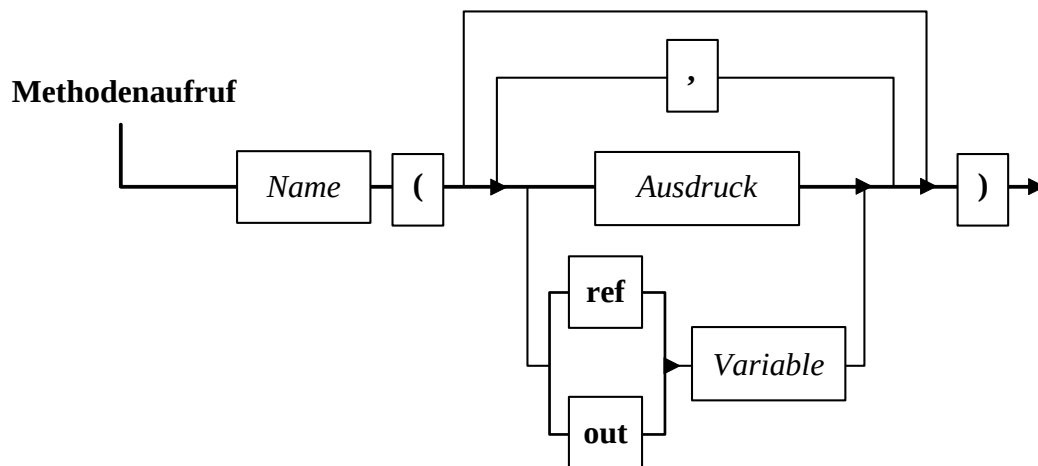
```
b1.Zeige();
```

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.:

```
do
    Console.WriteLine("Welchen Bruch möchten Sie kürzen?");
while (!b1.Frage());
```

Durch ein angehängtes Semikolon wird jeder Methodenaufruf zur vollständigen Anweisung (siehe erstes Beispiel), wobei ein Rückgabewert ggf. ignoriert wird, z.B.:

```
b1.Frage();
```

Soll in einer Methodenimplementierung vom aktuell handelnden Objekt eine andere Methode derselben Klasse ausgeführt werden, so muss beim Aufruf *keine* Objekt- oder Klassenbezeichnung 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 4.3.1.1.1 paramtergesteuert) gleich kürzen. Zum Kürzen kommt natürlich die entsprechende Bruch-Methode zum Einsatz. Weil sie vom gerade agierenden Objekt auszuführen ist, muss keine Objektreferenz angegeben werden.

Wer auch solche Methodenaufrufe nach dem Schema

Empfänger.Botschaft

realisieren möchte, kann mit dem Schlüsselwort **this** das aktuelle Objekt ansprechen, z.B.:

```
if (autokurz)
    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 *Überladung*. Von einer Methode unterschiedlich parametrisierte Varianten in eine Klassendefinition aufzunehmen, lohnt sich z.B. in folgenden Situationen:

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

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

```
public static 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.

Der Compiler versucht bei jedem Methodenaufruf, die angeforderte Überladung eindeutig zu bestimmen, und meldet bei Mehrdeutigkeit einen Fehler.

4.4 Objekte

4.4.1 (Wieder)verwendung von Klassen bzw. Objekten

Ein zentrales Ziel der OOP ist die Produktion von Software mit hohem Recycling-Potential. Daher wurde im Bruchrechnungs-Demoprojekt die recht universell verwendbare Bruch-Klasse konzipiert, die schon in Programmen mit stark unterschiedlicher Benutzerschnittstelle (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. Dies geschieht meist in den Methoden von anderen Klassen.

Jede .NET – Anwendung benötigt eine Startklasse mit statischer **Main()**-Methode, die von der CLR (Common Language Runtime) beim Programmstart aufgerufen wird. Für ein Konsolenprogramm zum Kürzen von Brüchen eignet sich die folgende Klasse BruchRechnung, deren **Main()**-Methode ein Objekt aus der Klasse Bruch erzeugt und mit Aufträgen versorgt:

```
using System;
class BruchRechnung {
    static void Main() {
        Bruch b = new Bruch();
        Console.WriteLine("Kürze Deinen Bruch:\n");
        b.Frage();
        b.Kuerze();
        b.Etikett = "Der gekürzte Bruch:";
        b.Zeige();
    }
}
```

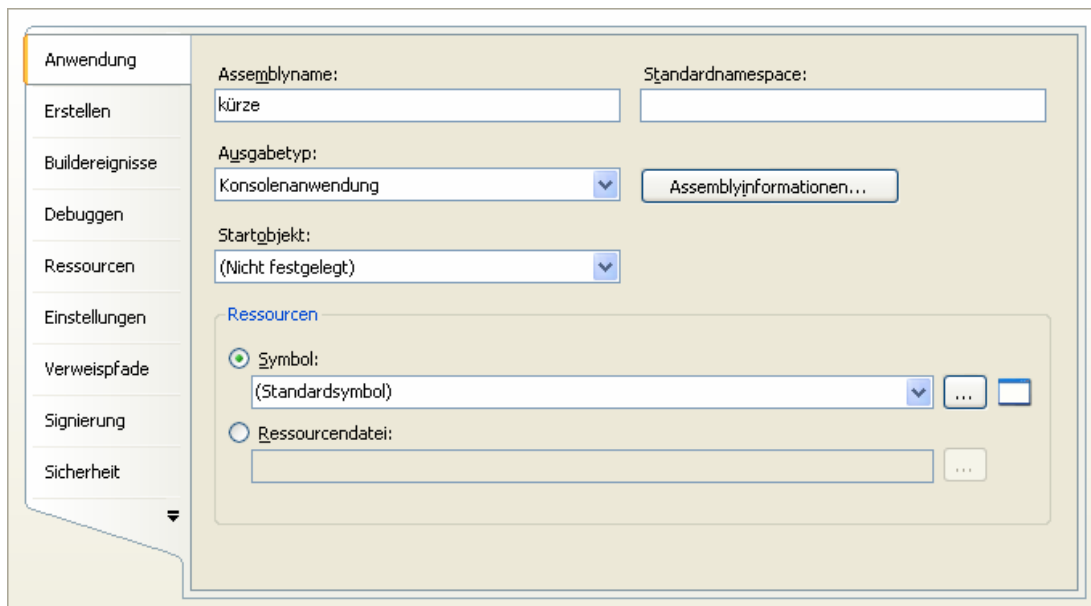
Hinsichtlich der Dateiverwaltung wird vorgeschlagen:

- Auch die BruchRechnung-Klassendefinition sollte in einer eigenen Datei gespeichert werden.
- Den Namen dieser Datei sollte man aus dem Klassennamen durch Anhängen der Extension „.cs“ bilden.

Beim gemeinsamen Übersetzen der Quellcode-Dateien **Bruch.cs** und **BruchRechnung.cs** entsteht *ein* Exe-Assembly, dessen Namen wir bei Verwendung der Entwicklungsumgebung Visual C# 2005 Express Edition nach dem Menübefehl

Projekt > Eigenschaften

festlegen können, z.B.:



Im Beispiel entsteht die Assembly-Datei **kürze.exe**.

Nach diesem spontanen Exkurs zur Projektverwaltung geht es nun um wichtige syntaktische und semantische Details bei der Verwendung von Objekten.

4.4.2 Referenzvariablen deklarieren

Um die Wirkungsweise der Variablendeklaration und -initialisierung

```
Bruch b = new Bruch();
```

zu erklären, beginnen wir mit einer einfacheren Variante:

```
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 Referenz-Literal steht für einen leeren Verweis. Eine Referenzvariable mit diesem Wert ist nicht undefiniert, sondern zeigt explizit auf nichts.

Wir halten fest, dass Klassen als Datentypen verwendet werden können und haben damit bisher die folgende Datentypen „offiziell“ kennen gelernt (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.

4.4.3 Objekte erzeugen

Steht die Deklarationsanweisung

```
Bruch b;
```

in einem Methodenrumpf, handelt es sich bei **b** um eine *lokale* Variable ohne Initialisierungswert. Bevor der Variablen ein Verweis auf ein **Bruch**-Objekt als Wert zugewiesen werden kann, muss ein solches Objekt erst erzeugt werden, was der **new**-Operator übernimmt, z.B. in folgendem Ausdruck:

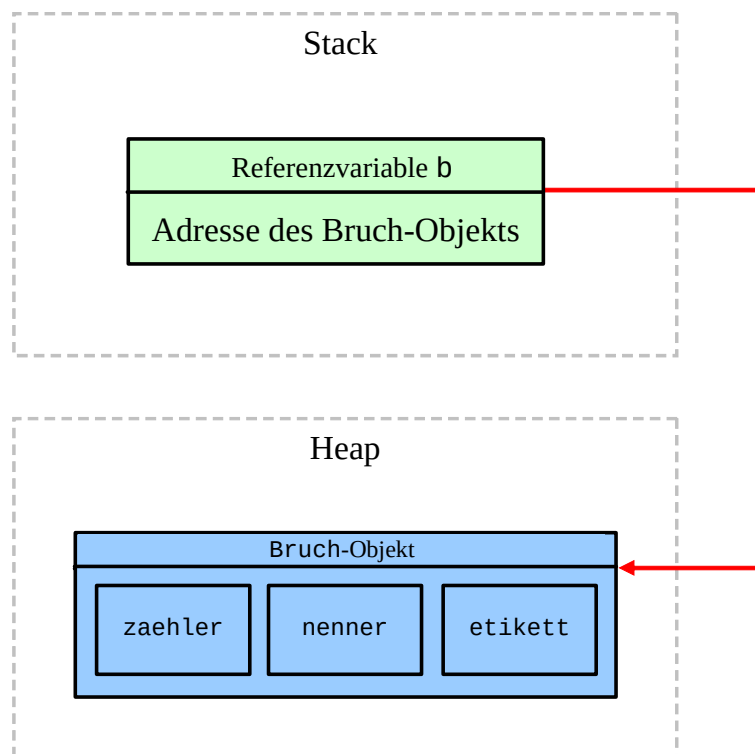
```
new Bruch()
```

Als Operanden erwartet **new** einen Klassennamen, dem eine Parameterliste zu folgen hat, weil er hier als Name eines *Konstruktors* (siehe Abschnitt 4.4.4) aufzufassen ist. Als Wert des Ausdrucks resultiert eine Referenz, die einen Zugriff auf das neue Objekt (seine Methoden, Eigenschaften, etc.) erlaubt.

In der **Main()**-Methode der Klasse **BruchRechnung** schreiben wir die vom **new**-Operator gelieferte Adresse in die Referenzvariable **b**:

```
Bruch b = new Bruch();
```

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.

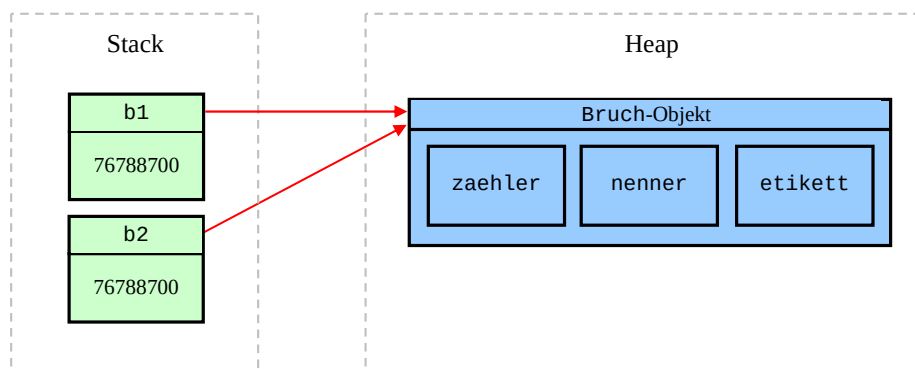
In einem Programm können mehrere Referenzvariablen auf dasselbe Objekt zeigen, z.B.:

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 Planer-Objekts, dem sie besondere Ereignisse wie Pannen melden.

4.4.4 Objekte initialisieren über Konstruktoren

In diesem Abschnitt werden spezielle Methoden behandelt, die beim Erzeugen von neuen Objekten automatisch aufgerufen werden, um diese 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 parameterloser **Standardkonstruktor** zum Einsatz. Wie der MSIL-Code zum Standardkonstruktor unserer Bruch-Klasse zeigt, fügt der Compiler automatisch Code für die im Quelltext enthaltenen Initialisierungen von Instanzvariablen ein:

```

Bruch::.ctor : void()
Suchen Weitsuchen
.method public hidebysig specialname rtspecialname
    instance void .ctor() cil managed
{
    // Code size          25 (0x19)
    .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: ret
} // end of method Bruch::.ctor

```

Für eine automatische Null-Initialisierung (vgl. Abschnitt 4.2.4) ist hingegen kein MSIL-Code erforderlich.

Am Ende (!) des MSIL-Codes zum Standardkonstruktor wird der parameterlose Konstruktor der Basisklasse aufgerufen, wobei unsere Klasse *Bruch* direkt von der Urahnklasse **Object** abstammt.

Weil der Standardkonstruktor (von einer momentan noch irrelevanten Ausnahme abgesehen) die Schutzstufe **public** besitzt, konnten wir Anweisungen nach folgendem Muster in Methoden beliebiger Klassen verwenden:

```
Bruch b = new Bruch();
```

In aller Regel ist es beim Klassendesign sinnvoll, *explizit* einen Konstruktor (oder sogar mehrere Konstruktoren) 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.
- Es darf eine Parameterliste definiert werden, was zum Zweck der Initialisierung ja auch unumgänglich ist.
- 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.
- 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 enthaltenen Feldinitialisierungen ein (siehe oben), z.B. auch bei einem Konstruktor mit leerem Anweisungsteil.
- Es sind beliebig viele Konstruktoren möglich, die alle denselben Namen und jeweils eine individuelle Parameterliste haben müssen. Das Überladen von Methoden (vgl. Abschnitt 4.3.3) ist also auch bei Konstruktoren erlaubt.
- Explizite Konstruktoren haben wie gewöhnliche Methoden die voreingestellte Schutzstufe **private**. Wenn sie für beliebige Klassen zur Objektkreation verfügbar sein sollen, ist in der Definition der Modifikator **public** anzugeben.

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() {}
```

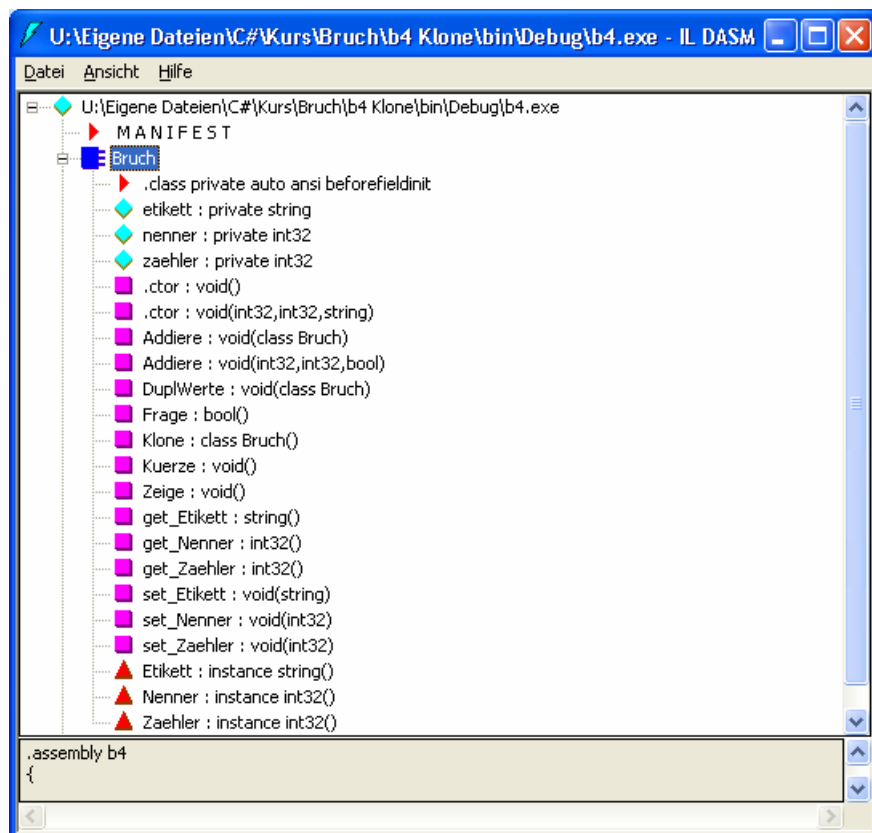
Im folgenden Programm werden beide Konstrukturen eingesetzt:

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 Objekt-Konstrukturen nur über den **new**-Operator aufgerufen 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 .NET – SDK - Hilfsprogramm **ILDasm** zeigt, erscheinen die Konstrukturen in der Methoden-Metadatatabelle mit dem Namen **.ctor** (Abkürzung für Konstruktor):



4.4.5 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 der Methode ungültig. Man kann eine Referenzvariable aber auch aktiv von einem Objekt „entkoppeln“, kann man ihr den Wert **null** (Zeiger 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 Objekt-Massen, die ein typisches Programm (z.B. ein Graphikeditor) benötigt, ist einiger Aufwand erforderlich, um eine störende und vielleicht sogar schädliche Verschwendung von Speicher zu verhindern.

Mit seinem vollautomatischen Müllsammler vermeidet C# lästigen Aufwand und 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).

4.4.6 Objekt-Referenzen 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 vom *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 zeigen.¹

Von den beiden `Addiere()` – Methoden der Klasse `Bruch` verfügt die ältere Variante über einen Wertparameter mit Referenztyp:

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

Mit dem Aufruf dieser Methode wird ein Objekt beauftragt, den via Parameter spezifizierten `Bruch` zu seinem eigenen Wert zu addieren und das Resultat gleich zu kürzen.

¹ Wertparameter vom Referenztyp arbeiten also analog zu den Referenzparametern (vgl. Abschnitt 4.3.1.1.2), 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.

Zähler und Nenner des fremden Bruch-Objektes können per Parameter und Punktoperator trotz Schutzstufe **private** direkt angesprochen werden, weil der Zugriff in einer Bruch-Methode stattfindet.

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 Objektes 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;
}
```

Hier liegt *kein* Verstoß gegen das Prinzip der Datenkapselung vor, weil der Zugriff durch eine klasseigene Methode erfolgt, die vom Klassendesigner gut konzipiert sein sollte.

In folgendem Programm wird das Bruch-Objekt `b1` beauftragt, die `DuplWerte()`-Methode auszuführen, wobei als Parameter eine Referenz auf das Objekt `b2` übergeben wird:

Quellcode	Ausgabe
<pre>using System; class BruchRechnung { static void Main() { Bruch b1 = new Bruch(1, 2, "b1 = "); Bruch b2 = new Bruch(5, 6, "b2 = "); b1.Zeige(); b2.Zeige(); b1.DuplWerte(b2); Console.WriteLine("Nach DuplWerte():\n"); b2.Zeige(); } }</pre>	<pre> 1 b1 = ---- 2 5 b2 = ---- 6 Nach DuplWerte(): 1 b2 = ---- 2</pre>

4.4.7 Rückgabewerte vom Referenztyp

Bisher haben die innerhalb von Methoden 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 kreierte Objekt nach Ende der Methodenausführung weiterhin zur Verfügung stehen, so muss eine Referenz nach Außen geschafft werden, z.B. über einen Rückgabewert mit Referenztyp.

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 Referenz an den Aufrufer zu übergeben:¹

```
public Bruch Klone() {
    return new Bruch(zaehler, nenner, etikett);
}
```

¹ Im Zusammenhang mit den Schnittstellen (Interfaces) werden wir noch eine alternative Methode zum Klonen erstellen.

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>

4.4.8 this als Referenz auf das aktuelle Objekt

Gelegentlich ist es erforderlich, dass ein Objekt sich selbst ansprechen bzw. seine Adresse mitteilen kann. Dies ist mit dem Schlüsselwort **this** möglich, das innerhalb einer Methode wie eine Referenzvariable benutzt werden kann. In folgendem Beispiel ermöglicht die **this**-Referenz die Verwendung von Formalparameternamen, die mit den Namen von Instanzvariablen übereinstimmen:

```
public void Addiere(int zaehler, int nenner, bool autokurz) {
    this.zaehler = this.zaehler * nenner + zaehler * this.nenner;
    this.nenner = this.nenner * nenner;
    if (autokurz)
        this.Kuerze();
}
```

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 relevanteren **this**-Verwendungsmöglichkeiten begegnen.

4.5 Eigenschaften

Sollen fremde Klassen Lese- und/oder Schreibzugriff auf ein gekapseltes (privates) Feld erhalten, sind entsprechende Zugriffsmethoden zu definieren. Im Bruch-Beispiel könnte man z.B. für das `nenner`-Feld die folgenden Methoden definieren:

```
public int GibNenner() {
    return nenner;
}

public void SetzeNenner(int value) {
    if (value != 0)
        nenner = value;
}
```

Entsprechend sähe der klassenfremde Zugriff auf einen Nenner z.B. so aus:

```
b1.SetzeNenner(2);
```

Mit den *Eigenschaften* (engl.: *properties*), die wegen ihrer großen Bedeutung wegen schon in der ersten Variante des Bruch-Beispiels genutzt wurden, bietet C# die Möglichkeit, den syntaktischen Aufwand beim Verwenden einer Klasse zu reduzieren. Aus den obigen Methodendefinitionen wird 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 Formalparameter-Definition über das Schlüsselwort **value** angesprochen wird.

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

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

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 die in Abschnitt 4.4.4 enthaltene Ausgabe des .NET – SDK - Hilfsprogramms **ILDasm** zeigt, übersetzt der Compiler die Eigenschaften unserer Klasse **Bruch** tatsächlich jeweils in ein Paar von Zugriffsmethoden.

Diese besitzen neben ihrer Genese eine weitere Besonderheit gegenüber den bisher besprochenen Methoden: Der Compiler fügt den (meist sehr kleinen) Zwischencode an jeder Aufrufstelle ein, um bei Eigenschaftszugriffen den Aufwand eines gewöhnlichen Methodenaufrufs zu vermeiden.¹

Um eine **Read**- bzw. **Write-Only** Eigenschaft zu definieren, verzichtet man einfach auf die **set**- bzw. die **get**-Implementation.

4.6 Klassenbezogene Member

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 daher oft von *statischen* Feldern, Eigenschaften etc.

Abgesehen von den statischen Konstruktoren (siehe unten) 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**).

¹ Diese auch von anderen Compilern eingesetzte Technik zur Optimierung von Funktions- bzw. Methodenaufrufen bezeichnet man als *inlining*.

4.6.1 Felder und Eigenschaften

In unserem Beispiel kann eine klassenbezogene Variable 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++;
    }

    . . .
}
```

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 unten) vorgenommen werden.

In der folgenden Tabelle werden wichtige Unterschiede zwischen Klassen- und Instanzvariablen erläutert:

	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. Sie werden beim Entladen der Klasse ungültig.

Ein statisches Feld kann in klasseneigenen Methoden (objektbezogen oder statisch) direkt angesprochen werden. In unserem Beispiel wird die (automatisch auf 0 initialisierte) Klassenvariable `anzahl` in den beiden Objekt-Konstrukturen inkrementiert.

Sofern Methoden *fremder* Klassen überhaupt (durch den Modifikator **public**) der direkte Zugriff auf eine Klassenvariable gewährt wird, müssen diese dem Variablennamen einen 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.

Damit fremde Klassen dennoch die Anzahl der bisher erzeugten Bruch-Objekte in Erfahrung bringen können, wird im Beispiel 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 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 Beispielprogramm zum Einsatz kam, z.B.:

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

Der Methodenname wird mit einem Vorspann aus dem Klassennamen und dem Punktoperator qualifiziert.

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();
b.Frage();
b.Kuerze();
b.Etikett = "Benutzerdefiniert: ";
```

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.

Ein Anwender der Bruch-Klasse kann die obige Sequenz durch eine einzelne Anweisung ersetzen:

Quellcode	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 2 Benutzerdefiniert: ----- 3</pre>

In Abschnitt 4.7.2 wird eine statische Bruch-Methode vorgestellt, die einen neuen Bruch als Summe von zwei vorhandenen Brüchen liefert.

Eine Klassenmethode kann natürlich auch von anderen Methoden der eigenen Klasse (objekt- oder klassenbezogen) aufgerufen werden, wobei der Klassenname *nicht* angegeben werden muss.

In früheren Abschnitten waren mit *Methoden* stets *objektbezogene* Methoden (*Instanzmethoden*) gemeint. Dies soll auch weiterhin so gelten.

4.6.3 Konstruktoren

Analog zum Einsatz eines Konstruktors für die Initialisierung von Instanzvariablen kann ein **statischer Konstruktor** für die Initialisierung von Klassenvariablen verwendet werden. In einer etwas gekünstelten Erweiterung des Bruch-Beispiels soll der parameterfreie Konstruktor zufallsabhängige, aber pro Programmlauf identische Werte zur Initialisierung der Felder `zaehler` und `nenner` verwenden. Dazu werden in der Klassendefinition statische Felder ergänzt, die vom statischen Konstruktor mit Zufallswerten initialisiert werden sollen:

```
static readonly int zaehlerVoreinst;
static readonly int nennerVoreinst;
```

Der Modifikator **readonly** (vgl. Abschnitt 4.2.1) sorgt dafür, dass die Felder nach der Initialisierung nicht mehr geändert werden können.

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 sein Wirken, 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>

Beim Einsatz von statischen Konstruktoren ist zu beachten:

- Pro Klasse ist nur *ein* statischer Konstruktor erlaubt.
- Es dürfen keine Zugriffsmodifikatoren angegeben werden. Diese werden auch nicht benötigt, weil ein statischer Konstruktor ohnehin nur vom Laufzeitsystem aufgerufen wird (beim Laden der Klasse).
- Bei einem Programm mit mehreren Klassen ist im .NET – Framework keine Reihenfolge für die Ausführung der eventuell vorhandenen statischen Konstruktoren definiert.

4.6.4 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.

4.6.5 Wiederholung zur Kategorisierung von Variablen

Mittlerweile haben wir verschiedene Variablentypen 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,
existieren, bis der innerste Block endet.
- **Instanzvariablen ...**
werden außerhalb jeder Methode deklariert,
landen (als Bestandteile von Objekten) auf dem Heap,
werden automatisch initialisiert,
existieren (sind verfügbar), solange das Objekt referenziert ist.
- **Klassenvariablen ...**
werden außerhalb jeder Methode mit dem Modifikator **static** deklariert,
werden automatisch initialisiert,
existieren, solange die Klasse geladen ist.
- **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 und Initialisierung festgelegt (siehe oben).

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) ersetzen sollten. Hier ist die effiziente Variante noch einmal zu sehen, wobei der GGT-Algorithmus vom restlichen Kürzungsverfahren getrennt und in eine eigene (private) Methode ausgelagert wurde:

```
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 = GGTr(Math.Abs(zaehler), Math.Abs(nenner));
        zaehler /= teiler;
        nenner /= teiler;
    }
}
```

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);
}
```

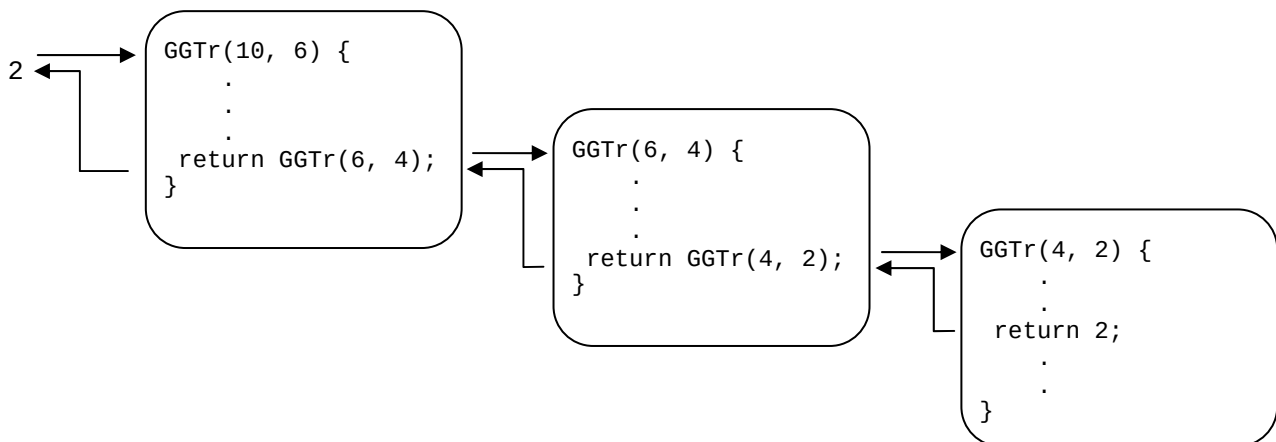
Statt eine Schleife zu benutzen, arbeitet die rekursive Methode nach folgender Logik:

- 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

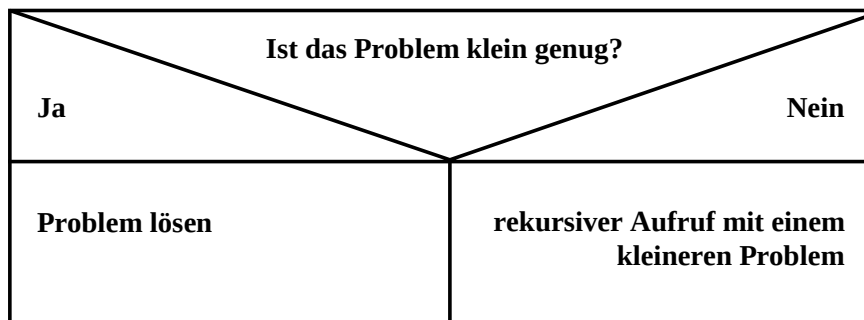
selbst mit neuen Aktualparametern auf. Dies geschieht recht elegant im Ausdruck der **return**-Anweisung:

```
return GGTr(b, rest);
```

Wird die Methode GGTr() z.B. mit den Argumenten 10 und 6 aufgerufen, kommt es zu folgender Aufrufverschachtelung:



Generell läuft ein rekursiver Algorithmus nach folgender Logik ab:



Wird bei einem fehlerhaften Algorithmus der linke Zweig in diesem *Struktogramm* nie erreicht, erschöpfen die geschachtelten Methodenaufrufe bald 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 übrigens in der Klasse **string** mit dem „+“ - Operator geschehen ist, so dass wir Zeichenfolgen bequem verketteten können.

Generell geht es beim Überladen von Operatoren um syntaktisch elegante Lösungen für Aufgaben, die ansonsten einen expliziten Methodenaufruf erfordern würden. Statt die Argumente in einer Aktualparameterliste anzugeben, können sie bei reduziertem Syntaxaufwand um ein Operatorzeichen gruppiert werden, das bisher nur für andere Aufgaben zuständig war.

Mit den aktuell in der Bruch – Klasse 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 weit eleganter, denselben Zweck mit folgender Formulierung zu erreichen:

```
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:

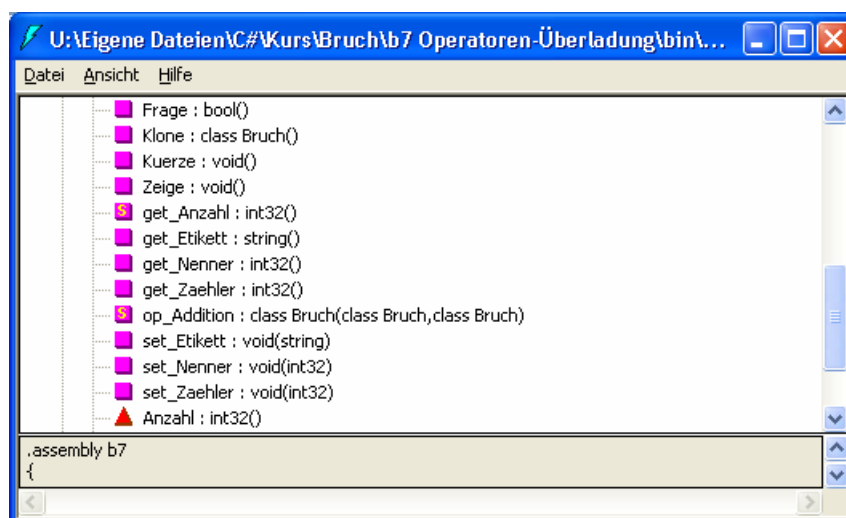
- Es ist grundsätzlich eine *statische* Definition erforderlich.
- Als Namen verwendet man das Schlüsselwort **operator** mit dem jeweiligen Operationszeichen als Suffix.

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>

Das .NET – SDK - Hilfsprogramm **ILDasm** zeigt die per Operatorzeichen aufrufbare Methode unter dem Namen **op_Addition**:



4.8 Aggregieren und Schachteln von 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. Wir erweitern das Bruchrechnungsprogramm um eine Klasse namens *Aufgabe*, die Trainingssitzungen unterstützen soll. In der *Aufgabe*-Klassendefinition tauchen mehrere Instanzvariablen vom Typ *Bruch* auf:

```
using System;

public class Aufgabe {
    Bruch b1, b2, lsg, antwort;
    char op;
    double dlsg;

    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;
        }
        dlsg = (double) lsg.Zaehler / lsg.Nenner;
    }

    public bool Korrekt {
        get {
            double dwert = (double) antwort.Zaehler/antwort.Nenner;
            return Math.Abs((dwert-dlsg)/dwert) < 1.0e-14 ? true : 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 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 eines Arbeitsauftrags 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.Frage();
        Console.WriteLine();
        if (auf.Korrekt)
            Console.WriteLine(" Gut!");
        else
            auf.Zeige(2);
        auf.NeueWerte('*', 3, 4, 2, 3);
        auf.Frage();
        Console.WriteLine();
        if (auf.Korrekt)
            Console.WriteLine(" Gut!");
        else
            auf.Zeige(2);
    }
}

```

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

Berechne bitte:

$$\begin{array}{r} 1 \\ \hline 2 \end{array} + \begin{array}{r} 2 \\ \hline 5 \end{array}$$

Welchen Zaehler hat dein Ergebnis: 3

Welchen Nenner hat dein Ergebnis: 7

Das korrekte Ergebnis: $\frac{9}{10}$

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 lokalen Klassen `Tochter` und `Sohn` definiert:

```
using System;

class Familie {
    string name;
    Tochter t;
    Sohn s;

    public Familie(string name_, int ato, int aso) {
        name = name_;
        t = new Tochter(this, ato);
        s = new Sohn(this, aso);
    }

    public void Info() {
        Console.WriteLine("Die Kinder von Familie {0}:\n", name);
        t.Info();
        s.Info();
    }

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

class Tochter {
    Familie f;
    int alter;

    public Tochter(Familie f_, int altpar) {
        f = f_;
        alter = altpar;
    }

    public void Info() {
        Console.WriteLine("    Ich bin die {0}-jährige Tochter von Familie {1}",
                           alter, f.name);
    }
}
```

```

class Sohn {
    Familie f;
    int alter;

    public Sohn(Familie f_, int altpar) {
        f = f_;
        alter = altpar;
    }

    public void Info() {
        Console.WriteLine("  Ich bin der {0}-jährige Sohn von Familie {1}",
                           alter, 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 von Familie Müller
Ich bin der 4-jährige Sohn von Familie Müller

```

Für das Schachteln von Klassendefinitionen gelten u.a. folgende Regeln:

- Die Methoden der inneren Klasse dürfen auf die privaten Member der umgebenden Klasse zugreifen, was z.B. in der folgenden Anweisung des Beispiels geschieht (name ist ein privates Feld in der Klasse Familie):

```
Console.WriteLine("  Ich bin die {0}-jährige Tochter von Familie {1}",
                  alter, f.name);
```
- Geschachtelte Klassen besitzen wie die sonstigen Klassen-Member (Felder, Methoden, Eigenschaften etc.) die voreingestellte Schutzstufe **private** (siehe Überblick in Abschnitt 4.9).
- Die *privaten* Member einer inneren Klasse sind in der umgebenden Klasse *nicht* verfügbar, weshalb im Beispiel die Konstruktoren und die Info()-Methoden der inneren Klassen als **public** definiert werden.
- In C# benötigt ein inneres Objekt beim Zugriff auf Member des umgebenden Objektes eine Referenz (z.B. in f.name), während in Java wie auf eigene Member zugegriffen wird.

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, wenn dabei auch noch einige kleine Vorgriffe 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.

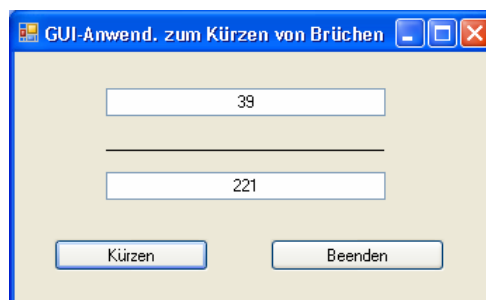
Für Klassen-Member (Felder, Methoden, Eigenschaften, geschachtelte Klassen etc.) unterstützt C# folgende Schutzstufen (siehe ECMA 2006, S. 31):

Modifikator(en)	Der Zugriff ist erlaubt für ...			
	die eigene Klasse	Klassen im Assembly	abgeleitete Klassen	sonstige Klassen
<i>ohne</i> oder private	ja	nein	nein	nein
internal	ja	ja	nein	nein
protected	ja	nein	ja	nein
protected internal	ja	ja	ja	nein
public	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.

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 viel Spaß machen kann. Wir erstellen nun das schon in Abschnitt 1.1.5 präsentierte Bruchkürzungsprogramm mit graphischer Benutzerschnittstelle:

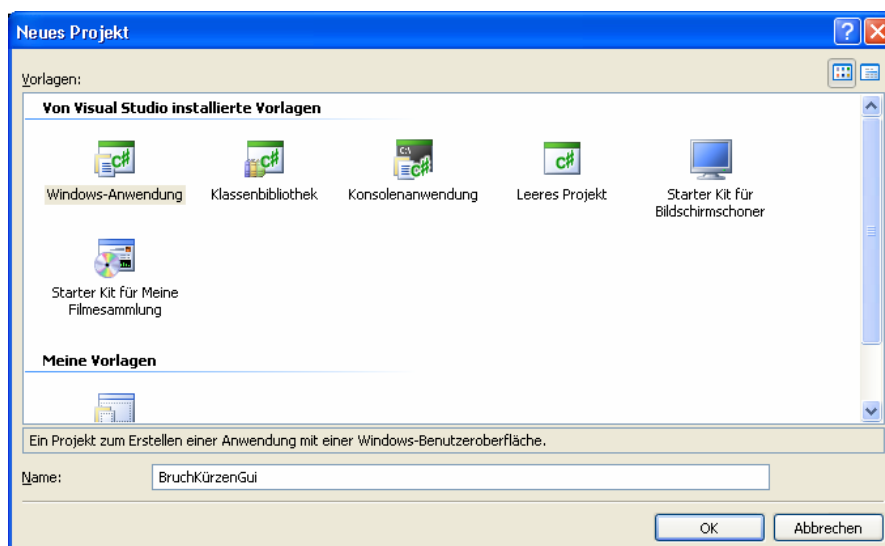


4.10.1 Projekt mit Vorlage Windows-Anwendung anlegen

In der folgenden Beschreibung wird die Visual C# 2005 Express Edition verwendet, doch bestehen keine wesentlichen Bedienungsunterschiede zum Visual Studio 2005 (vgl. Abschnitt 2.3.1.3). Verwenden Sie nach

Datei > Neu > Projekt

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

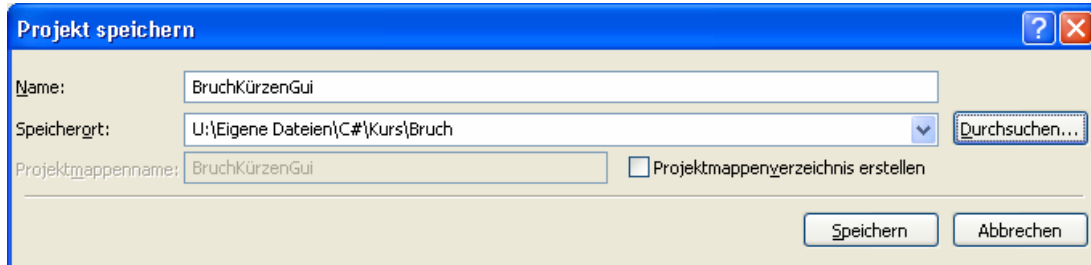


Nach einem Mausklick auf **OK** präsentiert die Entwicklungsumgebung den **Projektmappen-Explorer** und den Formulardesigner.

Speichern Sie das entstehende Projekt gleich über

Datei > Alle speichern

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



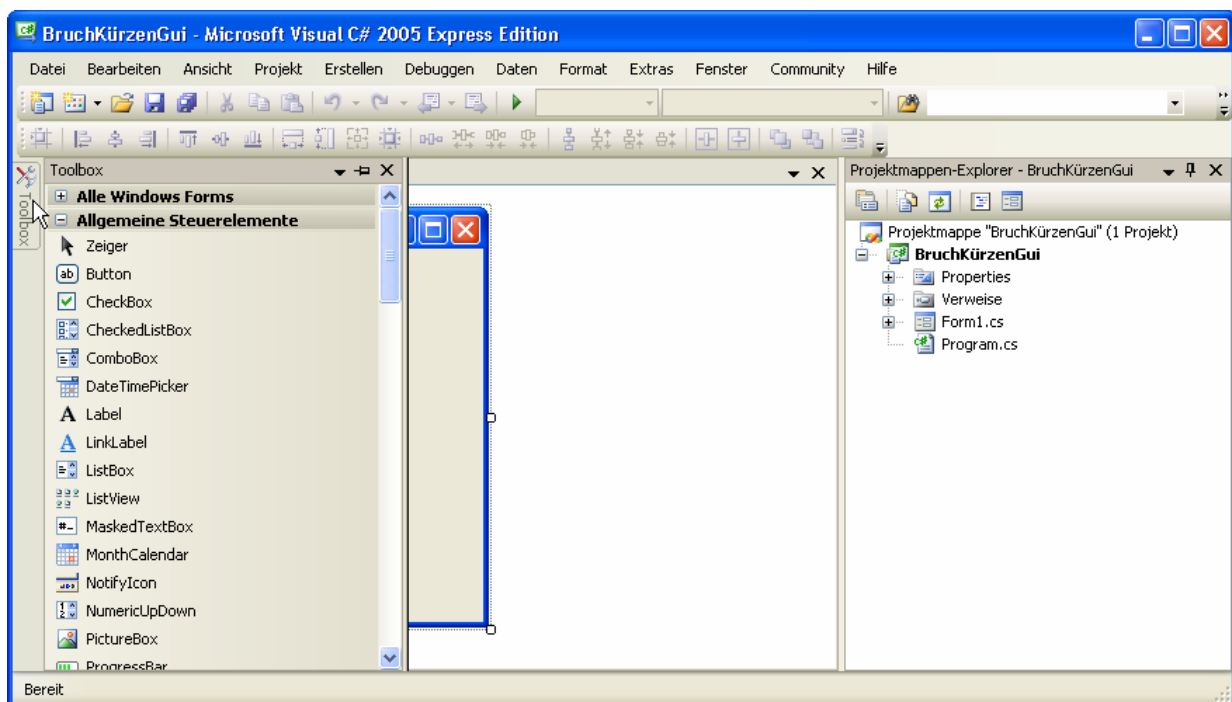
Durch den Verzicht auf ein **Projektmappenverzeichnis** wählen wir eine „flache“ Projektdatenverwaltung ohne Zusammenfassung von mehreren Projekten zu einer Mappe.

4.10.2 Steuerelemente aus der Toolbox übernehmen

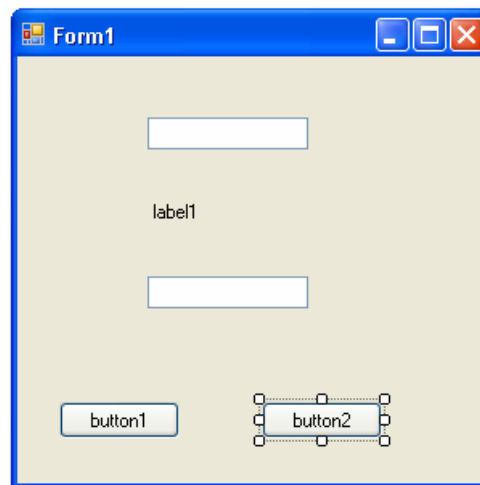
Öffnen Sie das **Toolbox**-Fenster mit dem Menübefehl

Ansicht > Toolbox

oder per Mauszeiger durch kurz Verharren auf der Toolbox-Beschriftung am linken Fensterrand:

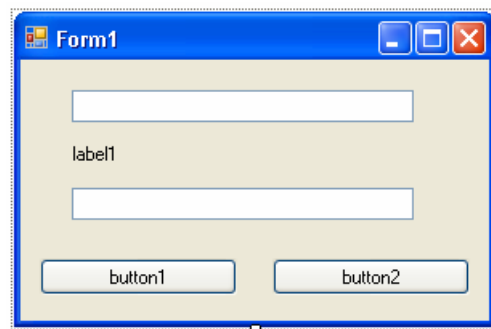


Ziehen Sie per Maus zwei **TextBox**-Objekte (für Zähler und Nenner), ein **Label** sowie zwei **Button**-Objekte auf das Formular, z.B.:

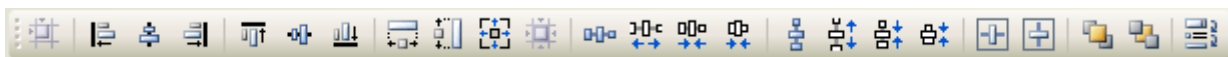


4.10.3 Steuerelemente gestalten

Nun können Sie wie in einem Graphik-Programm Positionierungen und Größenänderungen vornehmen, um das gewünschte Layout zu erzielen:



In der Symbolleiste **Layout** finden sich zahlreiche professionelle Gestaltungswerkzeuge zum Ausrichten, Angleichen etc.:



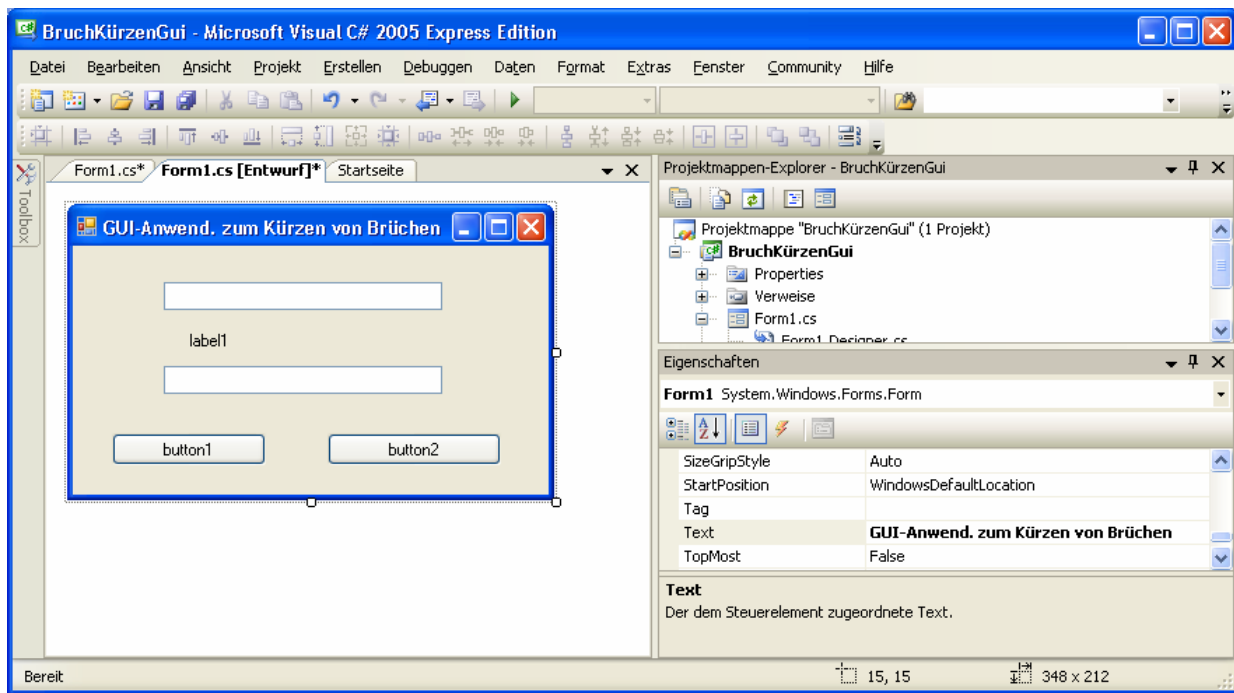
Bei gedrückter **Strg**-Taste kann man Steuerelemente wie in einem Grafikprogramm kopieren, wobei die Eigenschaften des Originals übernommen werden.

4.10.4 Eigenschaften der Steuerelemente ändern

Öffnen Sie das Eigenschaftsfenster mit dem Menübefehl

Ansicht > Eigenschaftsfenster

Nun 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
- die **Text** - Eigenschaft des Labels, um einen behelfsmäßigen Bruchstrich einzuzeichnen
- für beide Textboxen die Eigenschaft **TextAlign** auf den Wert **Center**
- der **AcceptButton**-Eigenschaft des Formulars auf den Namen des Befehlsschalters zum Kürzen, so dass er per **Enter**-Taste angesprochen werden kann

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

4.10.5 Der Quellcode-Generator

Aufgrund unserer 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** ausnahmsweise 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.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 Quellcodedatei **Form1.Designer.cs** ausnahmsweise per 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;

```

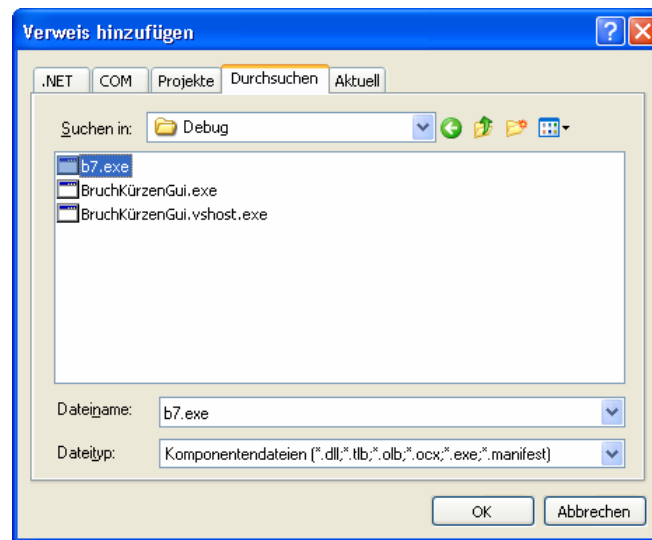
Aufgrund unserer Tätigkeit im Designer enthält die Formularklasse im Sinne der in Abschnitt 4.8.1 behandelten Aggregation mehrere Objekte anderer Klassen, die Steuerelemente der graphischen Benutzeroberfläche repräsentieren.

Eine weiter gehende Analyse des automatisch erstellten Quellcodes sparen wir uns an dieser Stelle.

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 Bruch-Klasse, 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\Bruch\BruchKürzenGui\bin\Debug
In der Praxis werden sich allgemein benötigte Assembly-Dateien an einem sinnvollen Ort befinden, so dass sie nicht unbedingt zur Nutzung in einem konkreten Projekt kopiert werden müssen.
- Nehmen Sie per Projektmappen-Explorer die Bruch-Assembly-Datei in die Verweisliste des Projekts auf, z.B.:



Die Bruch-Klasse befindet sich im globalen Namensraum, weil wir auf eine **namespace**-Definition verzichtet haben. Folglich muss beim Zugriff (im Unterschied zu den FCL-Klassen) kein Namensraum angegeben bzw. per **using**-Direktive importiert werden. Wegen der Gefahr von Namenskollisionen ist dieses Verfahren nicht empfehlenswert bei Klassen(bibliotheken), die in vielen Projekten verwendet werden sollen. Unsere Entwicklungsumgebungen 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. Wechseln Sie per Doppelklick auf den Befehlsschalter `button1` (mit dem Wert *Kürzen* für die Eigenschaft **Text**) zu der bei dieser Gelegenheit automatisch in der Datei **Form1.cs** erstellten Methode `button1_Click()`:

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

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 zwei Werte der Enumeration (siehe unten) **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()** aus der Klasse **Environment** ein, 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 Projekt-Unterordner **...\bin\Debug**. Beachten Sie beim Kopieren/Verschieben Ihres Programms, dass sich eine Assembly-Datei mit der verwendeten Bruch-Klasse im selben Ordner befinden muss.

4.11 Übungsaufgaben zu Kapitel 4

1) Welche von den folgenden Aussagen sind richtig?

- Alle Instanzvariablen einer Klasse müssen von elementarem Typ sein.
- In einer Klasse können mehrere Methoden mit demselben Namen existieren.
- Bei der Definition eines Konstruktors ist stets der Rückgabotyp **void** anzugeben.
- Mit der Datenkapselung wird verhindert, dass ein Objekt auf die Instanzvariablen anderer Objekte derselben Klasse zugreifen kann.
- Als Wertaktualparameter sind nicht nur Variablen erlaubt, sondern beliebige Ausdrücke mit kompatibel (erweiternd konvertierbarem Typ).

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 aber nicht bemerkt.

4) Die folgende Aufgabe eignet sich nur für Leser(innen) mit Grundkenntnissen in linearer Algebra: Erstellen Sie eine Klasse für zweidimensionale Vektoren, 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 1 **normieren**

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

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

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

- 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:

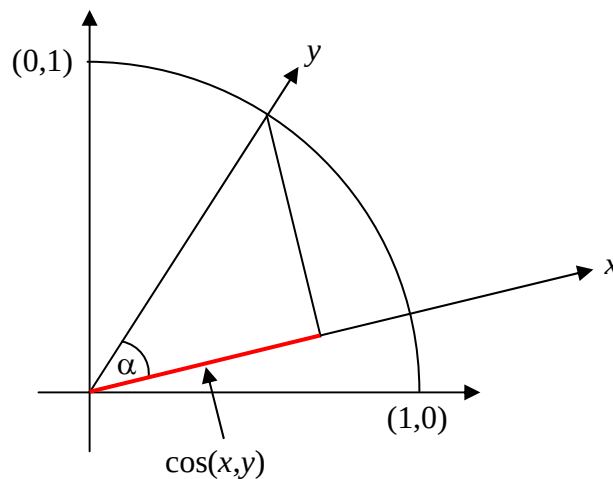
$$x \cdot y := x_1 y_1 + x_2 y_2$$

- **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

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

¹ Dies folgt aus dem Additionstheorem für den Kosinus.



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

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

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

- **Rotation** eines Vektors um einen bestimmten Winkelgrad
Mit Hilfe der Rotationsmatrix

$$D := \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}$$

kann der Vektor x um den Winkel α (im Bogenmaß!) gedreht werden:

$$x' = D x = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \cos(\alpha) x_1 - \sin(\alpha) x_2 \\ \sin(\alpha) x_1 + \cos(\alpha) x_2 \end{pmatrix}$$

Zum Berechnen 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.

In folgendem Beispiel wurde für eine formatierte Ausgabe der **double**-Werte gesorgt:

```
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.
- 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.

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.

Eine Struktur eignet sich vor allem bei folgender Konstellation:

- Der Datentyp ist relativ einfach aufgebaut.
- Es werden viele Instanzen benötigt.
- 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.

Beispiele:

- Komplexe Zahlen¹
- Punkte in einem zweidimensionalen Zahlenraum

5.1.1 Referenz- versus Wertsemantik

In diesem Abschnitt vergleichen wir eine Struktur und eine Klasse zur Repräsentation von Punkten der reellen Zahlenebene ($\mathbb{R}^2$). Für die beiden Koordinaten eines Punktes sind Felder mit dem elementaren Datentyp **double** vorhanden. Weil auch fremden Klassen der lesende und schreibende Zugriff erlaubt werden soll, verzichten wir auf eine Datenkapselung.

Eine Strukturdefinition unterscheidet sich von der gewohnten Klassendefinition auf den ersten Blick nur durch das neue Schlüsselwort **struct**:

¹ Dieser mathematische Begriff meint Paare aus reellen Zahlen, für die spezielle Rechenregeln gelten.

```

using System;

public struct Punkt {
    public double X, Y;

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

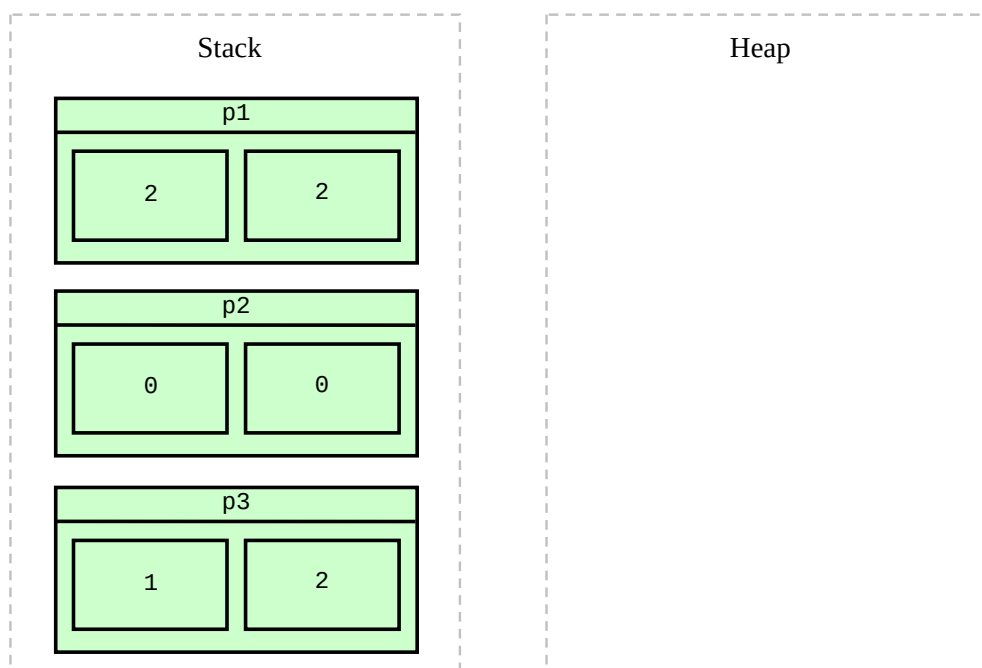
    public string Inhalt {
        get {
            return "("+X+";"+Y+")";
        }
    }
}

```

Beim folgenden Einsatz der Punkt-Struktur wird die Variable p1 vom parametrisierten Konstruktor mit dem Wert (1, 2) initialisiert. Der Punkt p2 erhält über die Standardinitialisierung auf den Wert (0, 0). Beim Punkt p3 wird *kein* initialisierender Konstruktor verwendet; er erhält stattdessen 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+")";
        }
    }
}

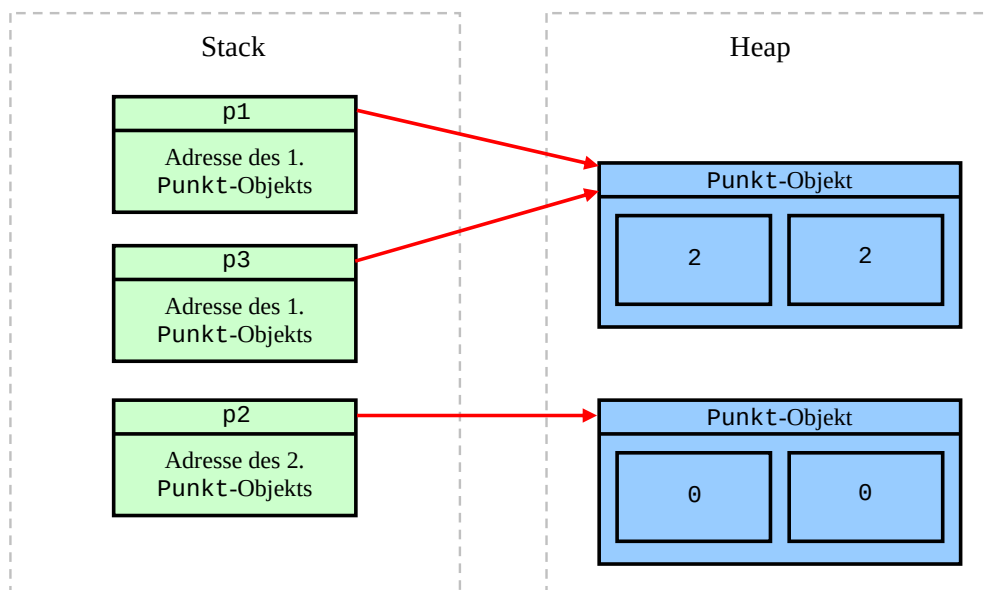
```

Weil bei Klassen (im Unterschied zu Strukturen) der Standardkonstruktor verloren geht, sobald ein expliziter Konstruktor vorhanden ist (vgl. Abschnitt 4.4.4), 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.

5.1.2 Unterschiede zwischen Klassen und Strukturen

Worauf man bei Strukturen im Vergleich zu Klassen verzichten muss, ist die *Vererbung*, 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 5.1.4). 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 Syntaxdiagramme zur Strukturdefinition sparen können:

- Das einleitende Schlüsselwort lautet **struct**.
- Bei der Deklaration von Instanzfeldern ist keine explizite Initialisierung erlaubt. Verboten ist also z.B.:

```
double delta = 1.0;
```

Felder von Struktur-Instanzen werden jedoch wie die Felder von Klassen-Instanzen per Voreinstellung mit der typspezifische Null initialisiert.

- Es sind beliebige viele Konstruktoren erlaubt, wobei der (parameterfreie) Standardkonstruktor *nicht* verloren geht und auch *nicht* ersetzt werden darf.
- Aus dem Vererbungsverbot ergeben sich einige zusätzliche Regeln (siehe ECMA 2006, S. 333).

5.1.3 Zur Eignung von Strukturen im Bruchrechnungs-Projekt

Grundsätzlich lässt sich auch unsere Bruch-Klassendefinition in eine analoge Strukturdefinition übersetzen:

```
using System;

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>

Weil bei der Felddeklaration keine Initialisierung erlaubt ist, und außerdem der parameterfreie Konstruktor nicht verändert werden darf, stellen sich gravierende Probleme bei parameterfrei konstruierten `Bruch`-Strukturinstanzen ein. Vor allem der Wert 0 im Nenner (siehe Ausgabe) missfällt sehr. Für Brüche ist eine Struktur als Datentyp offenbar weniger geeignet. Generell sollte eine Struktur so entworfen werden, dass die Null-Initialisierung durch den Standardkonstruktor zu einer regulären Instanz führt.

5.1.4 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 Namensraum **System**:

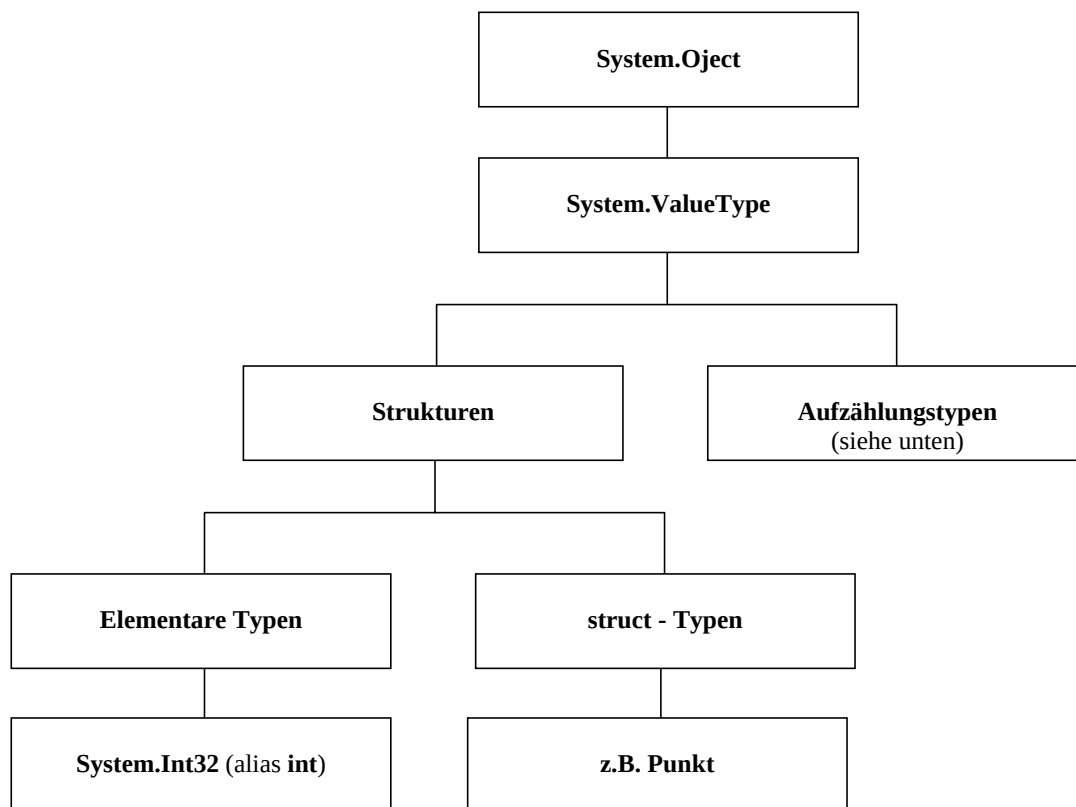
Aliasname	Struktur
sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

Nun wird z.B. klar, warum die **Convert**-Methode zur Konvertierung von Zeichenfolgen in **int**-Werte den Namen **ToInt32()** trägt.

Im Vergleich zu sonstigen Strukturtypen werden bei den elementaren Datentypen zusätzliche Operationen unterstützt (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 Urahnklasse **Object** einfügen:



Im folgenden Beispiel wird beim Ganzzahliliteral 2 (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(2.GetType()); } } </pre>	System.Int32

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?

Die beschriebene Zuweisungskonstellation ist keinesfalls exotisch. Wir haben uns vielmehr 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 Werte bei Bedarf automatisch in Objekte einer passenden Hilfsklasse verpackt werden, so dass letztlich ein Wert von beliebigem Typ als **Object**-Instanz behandelt werden kann.

Zur Erläuterung der Boxing-Technik bedienen wir uns in Anlehnung an ECMA (2004, S. 114f) einer nicht ganz korrekten, aber sehr 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 Wertes *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 weiß sich als Objekt zu benehmen.

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 allerdings beim *Unboxing*, also beim Auspacken eines Wertes, eine explizite Casting-Operation mit Angabe des Zieltyps:

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

5.3 Arrays

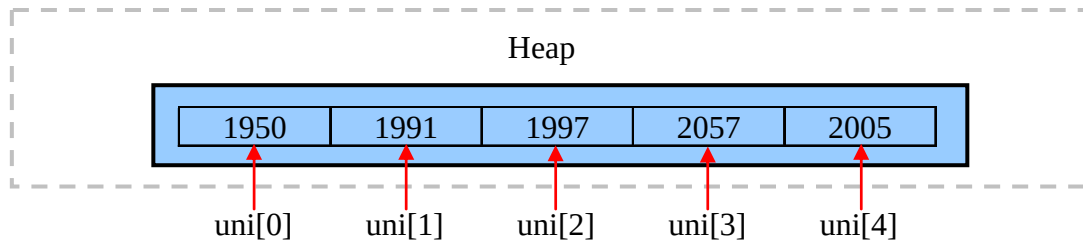
Ein Array ist ein Objekt, das als Instanzvariablen eine feste Anzahl von Elementen desselben Datentyps enthält. Die Elemente können gemeinsam verarbeitet (z.B. als Aktualparameter übergeben), aber auch einzeln über einen Index angesprochen werden.

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 Basis-klasse **Array** im Namensraum **System** abstammen.

Nun wird es Zeit für ein Beispiel. Hier ist ein eindimensionaler Array namens `uni` mit 5 Elementen vom Typ **int** zu sehen:



Neben den Array-Elementen enthält das Objekt noch Verwaltungsdaten, die wir aber nicht kennen müssen.

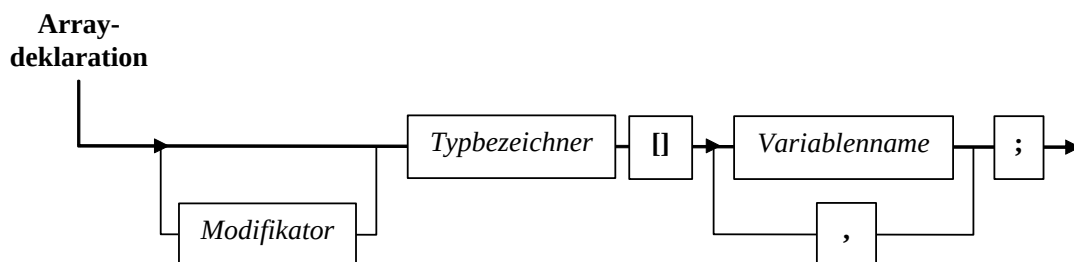
Beim Zugriff auf ein *einzelnes* Element gibt man nach dem Arraynamen den durch eckige Klammern begrenzten Index an, wobei die Nummerierung bei 0 beginnt.

Weil der Index auch über eine *Variable* geliefert werden kann, bietet ein Array z.B. im Zusammenhang mit einer Wiederholungsanweisung erhebliche Vorteile gegenüber einer Anzahl von selbständigen Variablen.

Wir befassen uns zunächst mit *eindimensionalen* Arrays, behandeln später aber auch den mehrdimensionalen Fall.

5.3.1 Array-Referenzvariablen deklarieren

Eine Array-Variable wird folgendermaßen deklariert:



Im Vergleich zu der bisher bekannten Variablendeklaration (ohne Initialisierung) ist hinter dem Typbezeichner zusätzlich ein Paar eckiger Klammern anzugeben.

In unserem 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.

5.3.2 Array-Objekte erzeugen

Mit Hilfe des **new**-Operators erzeugt man ein Array-Objekt mit einem bestimmten Typ 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];
```

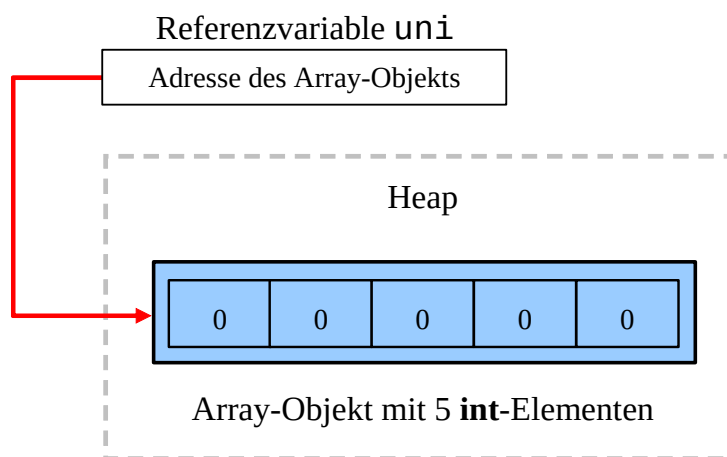

Im **new**-Operanden *muss* hinter dem Datentyp zwischen eckigen Klammern die Anzahl der Elemente festgelegt werden, was über einen beliebigen Ausdruck mit ganzzahligem Wert geschehen kann.

Man kann die Länge eines Arrays zur *Laufzeit* festlegen, z.B. in Abhängigkeit von einer Benutzereingabe. Existiert ein Array-Objekt erst einmal, kann die Anzahl seiner Elemente allerdings nicht mehr geändert werden. Wer noch dynamischere Datenstrukturen benötigt, kann z.B. die .NET – Klasse **ArrayList** benutzen, die Arrays variabler Länge unterstützt (siehe Abschnitt 5.3.8).

Die Deklaration einer Array-Referenzvariablen und die Erstellung des Array-Objektes 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. 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. Gemäß ihrer Deklaration kann eine Array-Referenzvariable auf einen beliebigen Array mit Elementen vom vereinbarten Typ zeigen.

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

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 folgender Schleife wird pro Durchgang ein zufällig (siehe unten) gewähltes `uni`-Element inkrementiert:

```
for (i = 1; i <= DRL; i++)
    uni[zzg.Next(5)]++;
```

Den Indexwert liefert hier ein Methodenaufruf mit Rückgabotyp **int**.

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]`. Hier können sich leicht Fehler einschleichen, die das Laufzeitsystem veranlassen, eine Ausnahme vom Typ **IndexOutOfRangeException** zu melden, z.B.:

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

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

Auch beim Entwurf von *Methoden* mit Array-Parametern ist es von Vorteil, dass die Länge eines übergebenen Arrays ohne entsprechenden Extra-Parameter 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.

Nach der folgenden Anweisung zeigt die Referenzvariable `zzg` auf ein Objekt der Klasse **Random** aus dem Namensraum **System**, das als Zufallszahlengenerator 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 zzg = new Random();
        for (i = 1; i <= DRL; i++)
            uni[zzg.Next(5)]++;
        Console.WriteLine("Absolute Häufigkeiten:");
        for (i = 0; i < 5; i++)
            Console.Write(uni[i] + " ");
        Console.WriteLine("\n\nRelative Häufigkeiten:");
        for (i = 0; i < 5; i++)
            Console.Write((double)uni[i]/DRL + " ");
    }
}
```

Die **Random**-Methode **Next()** liefert beim Aufruf mit dem Aktualparameterwert 5 als Rückgabewert 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

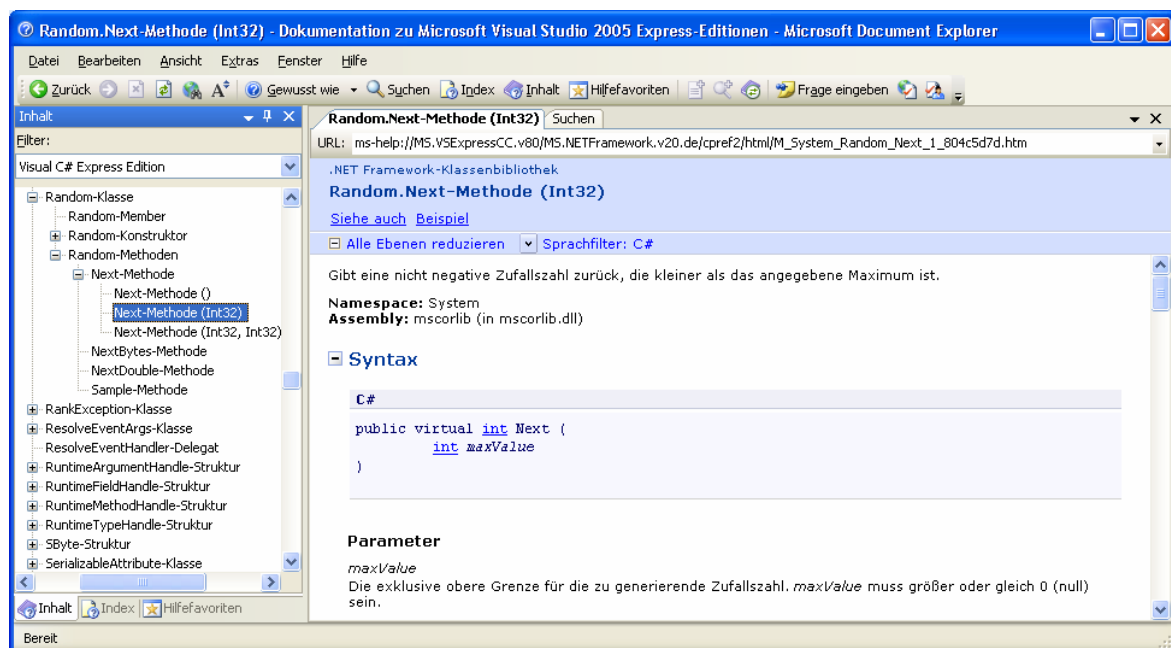
Relative Häufigkeiten:
0,1986 0,1983 0,1995 0,1995 0,2041

Über die im Beispielprogramm verwendete Klasse **Random** und deren **Next()**-Methode können Sie sich mit Hilfe der SDK-Dokumentation näher informieren, bei Verwendung der der Visual C# 2005 Express Edition z.B. so:

- Starten Sie den Document-Explorer über die Option **Inhalt** im **Hilfe**-Menü.
- Wählen Sie im Navigationsbereich den Ast:

MSDN Library für Visual Studio 2005 Express Editionen > Dokumentation zu .NET Framework SDK > Klassenbibliothek > System > Random-Klasse

- Markieren Sie die interessierende Überladung der **Next()**-Methode:



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;
```

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[] bruvek = {b1, b2}; bruvek[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 Deklaration bzw. Verwendung eines mehrdimensionalen rechteckigen Arrays werden die in eckigen Klammern eingeschlossenen Dimensionsangaben bzw. Indexwerte durch Komma getrennt.

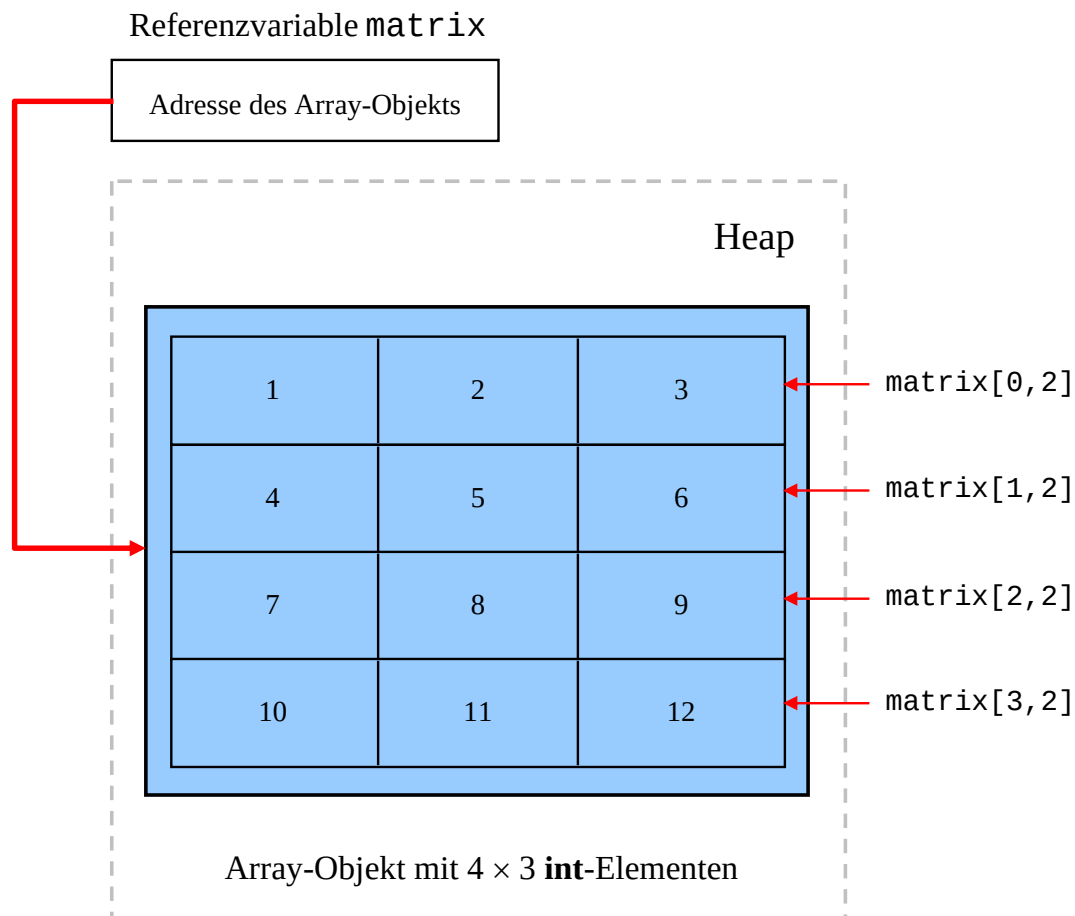
Auch im mehrdimensionalen Fall können Initialisierungslisten eingesetzt werden, z.B.:

```
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 SDK-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 geschlossen hintereinander, was einen schnellen Indexzugriff ermöglicht:

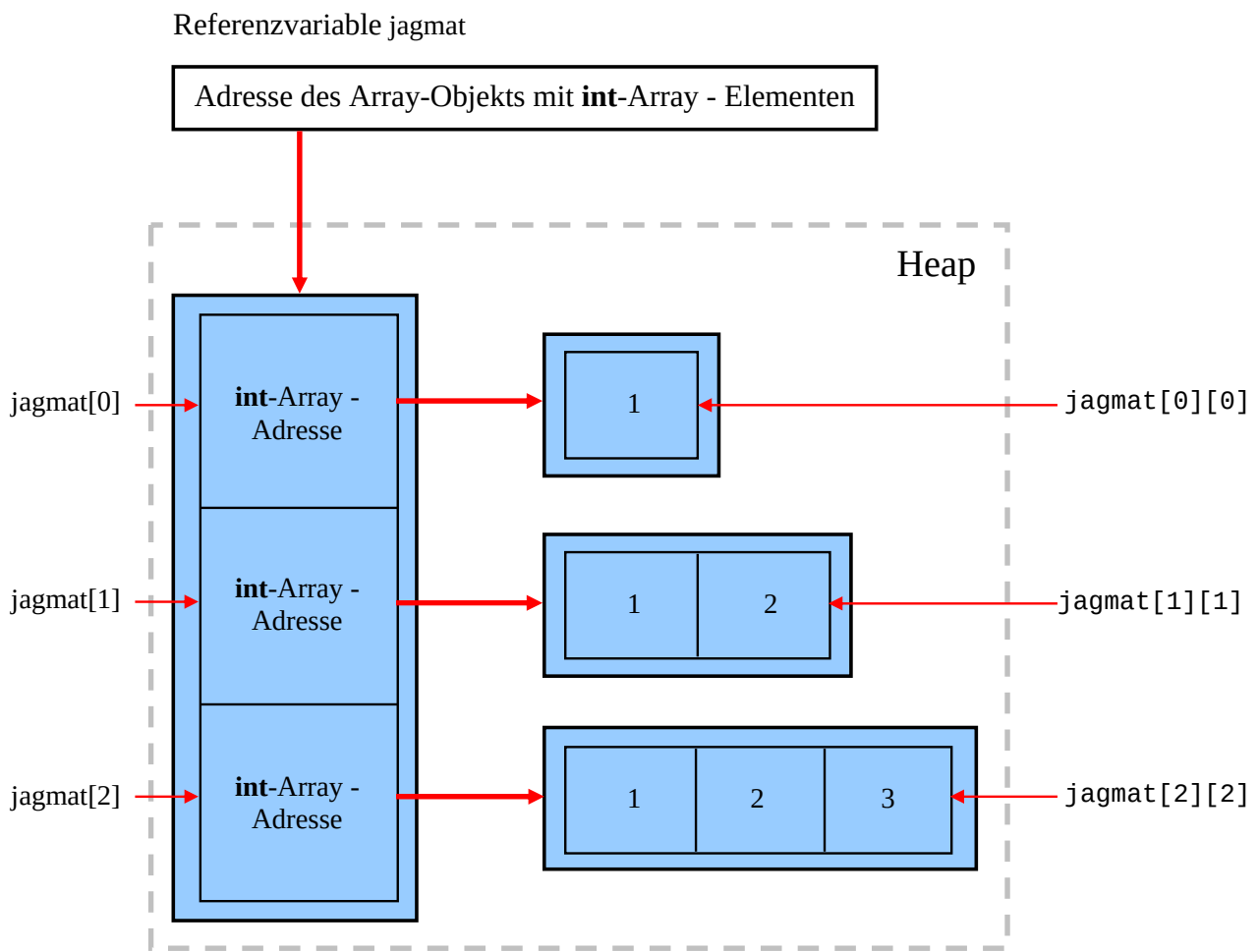


5.3.7.2 Mehrdimensionale Arrays mit unterschiedlich langen Elementen

Neben den rechteckigen Arrays unterstützt C# auch „ausgesägte“ Exemplare mit unterschiedlich langen 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[][]` - Objektes `jagmat` um Referenzen vom Typ `int[]`, die wiederum auf entsprechende Heap-Objekte 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.:¹

```
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 auch 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:

¹ Die für Initialisierungslisten eigentlich nahe liegende Schreibweise

```
jagmat[2] = {1, 2, 3};
```

akzeptiert der Compiler **nicht**.

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>	<pre> Was fällt Ihnen zu C#? : Tolle Sache : Nicht ganz trivial : Macht Spaß : Ihre Anmerkungen: Tolle Sache Nicht ganz trivial Macht Spaß </pre>

Weil die Klasse **ArrayList** einen *Indexer* bietet (siehe Abschnitt 5.6), kann man per Index-Syntax auf die Elemente zugreifen, wobei für die Indexwerte gilt:

- Das erste Element hat den Indexwert 0.
- Den Indexwert des letzten Elements kann man über die **Count**-Eigenschaft ermitteln, wobei aufgrund der nullbasierten Indexierung (**Count** – 1) der größte gültige Indexwert ist.

Als Datentyp für die Elemente dient **object** (alias **System.Object**), also die Klasse an der Wurzel 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(); string s = "Wort"; double d = 3.14; int i = 13; al.Add(s); al.Add(d); al.Add(i); foreach (object o in al) Console.WriteLine(o); } } </pre>	<pre> Wort 3,14 13 </pre>

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

5.4 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 „Write Once“ - 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 beliebig verändert werden können.

5.4.1 Die Klasse String für konstante Zeichenkette

Weil Objekte der Klasse **String** aus dem Namensraum **System** in C# - Programmen sehr oft benötigt werden, hat man diesem Datentyp das reservierte Wort **string** als Aliasnamen spendiert. Es unterscheidet sich vom Klassennamen nur durch den klein geschriebenen Anfangsbuchstaben.

In der folgenden Deklarations- und Initialisierungsanweisung

```
string s1 = "abcde";
```

wird:

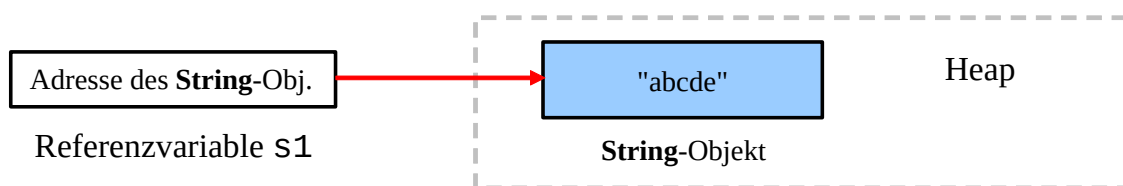
- eine **String**-Referenzvariable namens `s1` angelegt,
- ein neues **String**-Objekt mit dem Inhalt „abcd“ auf dem Heap erzeugt,
- die Adresse des neuen Heap-Objektes in der Referenzvariablen abgelegt.

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:



Erfolgt eine **String**-Initialisierung über einen *konstanten* Ausdruck, kommt der so genannte **interne String-Pool** ins Spiel: Existiert dort bereits ein Objekt mit demselben Inhalt, wird dessen Adresse verwendet. Anderenfalls wird ein neues Objekt im internen **String**-Pool angelegt.

5.4.1.1 String als Write Once - Klasse

Nachdem ein **String**-Objekt auf dem Heap erzeugt worden ist, kann es nicht mehr geändert werden. Eventuell werden Sie diese Behauptung 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 Speicher benötigt, wird es vom Garbage Collector eliminiert.

5.4.1.2 Elementare Operationen mit Strings

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 SDK-Dokumentation.

5.4.1.2.1 Verketteten von Strings

Zum Verketteten von Strings kann der „+“- Operator verwendet werden, wobei Operanden beliebige Datentypen bei Bedarf automatisch in Strings konvertiert werden. Wie Sie aus Abschnitt 5.4.1.1 wissen, entsteht dabei 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>	<pre>4 + 3 = 7</pre>

Wir haben schon in Abschnitt 3.2 das Verketteten von Zeichenketten kennen gelernt. Hinter diesem „Trick“ steckt eine in der Klasse **String** definierte **Überladung** des „+“ – Operators (siehe Abschnitt 4.7.2).

5.4.1.2.2 Vergleichen von Strings

Angewandt auf **string**-Variablen vergleichen die Operatoren `==` und `!=` *nicht* (wie z.B. in Java) die Adressen, sondern die Inhalte der referenzierten Zeichenfolge-Objekte, z.B.:

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { string s1 = "abcde"; string s2 = "abcde"; Console.WriteLine(s1 == s2); } }</pre>	<pre>True</pre>

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, Anja"; 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

5.4.1.2.3 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

5.4.1.2.4 Zeichen extrahieren, suchen oder ersetzen

Auf einzelne Zeichen eines Strings kann man per Index-Syntax 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>

Mit der Methode

string Substring(int start, int len)

lassen sich *len* Zeichen ab der Indexposition *start* extrahieren.

Eben wurde eine neue Technik zur Sprachbeschreibung im Manuskript erstmals benutzt, die auch in der SDK-Dokumentation üblich ist. Dabei wird die Benutzung einer Methode erläutert durch Angabe von:

- Rückgabetyp
- Methodenname
- Parameterliste (mit Angabe der Parametertypen)

Mit der Methode

int IndexOf(string gesucht)

kann man einen **String** nach einer anderen Zeichenkette durchsuchen. Als Rückgabewert erhält man ...

- nach erfolgreicher Suche: die Startposition der ersten Trefferstelle
- nach vergeblicher Suche: -1

Mit der Methode

bool StartsWith(string start)

lässt sich feststellen, ob ein String mit einer bestimmten Zeichenfolge beginnt.

Mit den Überladungen der Methode

string Replace(char alt, char neu)

string Replace(string alt, string neu)

erhält man als Rückgabewert die Adresse eines neuen Strings, der aus dem angesprochenen Original durch Ersetzen eines alten Zeichens (einer alten Zeichenfolge) durch ein neues Zeichen (eine neue Zeichenfolge) hervorgeht.

5.4.1.2.5 Groß-/Kleinschreibung normieren

Die Methoden **ToUpper()** bzw. **ToLower()** liefern einen neuen String mit komplett groß bzw. klein geschriebenen Zeichen, was für Vergleiche oft nützlich 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 unmittelbar zum Aufruf der Methode **IndexOf("T")** genutzt wird.

5.4.2 Die Klasse **StringBuilder** für veränderliche Zeichenketten

Für häufig zu ändernde Zeichenketten sollte man statt der Klasse **String** unbedingt die Klasse **StringBuilder** verwenden, weil hier beim Ändern einer Zeichenkette die (zeitaufwändige) Erzeugung eines neuen Objektes entfällt.

Ein **StringBuilder**-Objekt kann *nicht* implizit erzeugt werden, jedoch stehen bequeme Konstruktoren zur Verfügung, z.B.:

- **StringBuilder()**
Beispiel: `StringBuilder sb = new StringBuilder();`
- **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() {
        String s = "";
        long vorher = DateTime.Now.Ticks;
        for (int i = 0; i < 10000; 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 < 10000; i++)
            t.Append("");
        diff = DateTime.Now.Ticks - vorher;
        Console.WriteLine("Zeit für StringBuilder-Manipulation:\t"+diff/1.0e4);
    }
}
```

Die (in Millisekunden gemessenen) Laufzeiten unterscheiden sich erheblich:¹

```
Zeit für String-Manipulation:      93,75
Zeit für StringBuilder-Manipulation: 0
```

Ein **StringBuilder**-Objekt kennt u.a. die folgenden Methoden und Eigenschaften:

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>

¹ Gemessen auf einem Rechner mit einer Taktfrequenz von 3-GHz.

StringBuilder-Member	Erläuterung
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 wollten in einer Adressdatenbank auch den *Charakter* der erfassten Personen notieren und sich dabei an den vier Temperament-Typen 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.
- 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, z.B.:

```
if (otto.Temp == 1) ...
```

C# bietet mit den **Enumerationen (Aufzählungstypen)** eine Lösung, die beide Nachteile vermeidet, also bei geringem Speicherbedarf einen gut lesbaren Quellcode ermöglicht.

Bei der Definition eines Aufzählungstyps folgt auf das Schlüsselwort **enum** und den Typbezeichner eine geschweift eingeklammerte Liste mit Werten, die letztlich nur Namen für die Zahlen 0, 1, 2, ... sind:

```
enum Typbezeichner {Wert1, Wert2, Wert3, ...};
```

Per Voreinstellung verwenden Enumerationen den Basistyp **int**, belegen also 4 Byte Speicherplatz, doch kann in der Definition auch ein alternativer Basistyp angegeben werden (siehe SDK-Dokumentation).

Die Enumerationen sind Werttypen; ihre genaue Einordnung in das CTS (Common Type System) der .NET – Plattform ist einer Abbildung in Abschnitt 5.1.4 zu entnehmen.

Weil Enumerations-Werte 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!");
    // folgende Zuweisung wird nicht(!) verhindert:
    otto.Temp = (Temperament)13;
}

```

Das Zuweisen von Werten außerhalb des Definitionsbereichs wird *nicht* automatisch verhindert, so dass Enumerations-Instanzvariablen (abweichend von dem obigen schlechten Beispiel) in der Regel gekapselt werden sollten.

5.6 Vertiefung: Die Indexer der Klassen und Strukturen

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*.¹

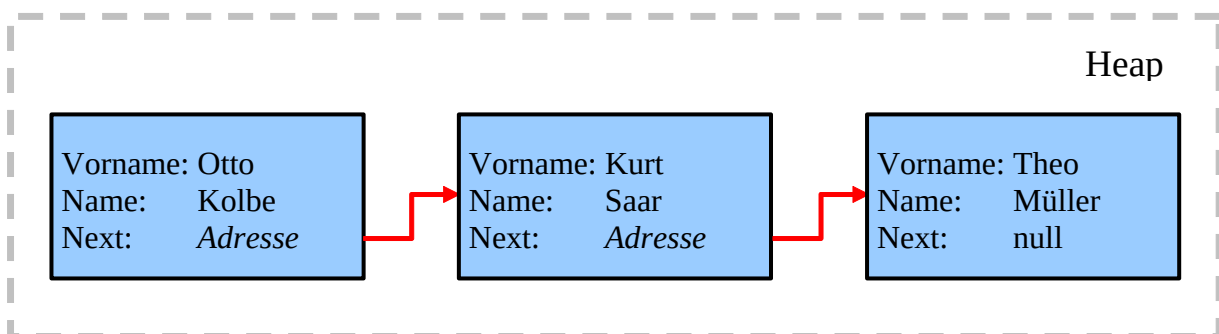
Als Beispiel betrachten wir eine „Datenbank“, die mit einer so genannten *verketteten Liste* von Objekten der Klasse **Person** arbeitet:

```

class Person {
    public string Vorname;
    public string Name;
    public Person Next;
    public Person(string vorname, string name) {
        Vorname = vorname;
        Name = name;
    }
}

```

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 3 **Person**-Objekten zu sehen:



Ein Objekt der Klasse **PersonDB** verwaltet beliebig viele Personen über eine verkettete Liste von **Person**-Objekten:

¹ Während es von den Eigenschaften auch *statische* Varianten gibt, fehlen diese bei den Indexern.

```

class PersonDB {
    int n;
    Person first, last;

    public int N {
        get {
            return n;
        }
    }

    public PersonDB(string vorname, string name) {
        first = last = new Person(vorname, name);
        n = 1;
    }

    public void Add(string vorname, string name) {
        Person neu = new Person(vorname, name);
        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) {
                Person sel = first;
                for (int j = 0; j < i - 1; j++)
                    sel = sel.Next; // Vorgänger
                value.Next = sel.Next.Next;
                sel.Next = value;
            }
        }
    }
}

```

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 Indexer-Typ 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 der Anwender dieser Klasse nichts:


```

using System;
class PersonDbDemo {
    static void Main() {
        PersonDB adb = new PersonDB("Otto", "Kolbe");
        adb.Add("Kurt", "Saar");
        adb.Add("Theo", "Müller");
        for (int i = 0; i < adb.N; i++)
            Console.WriteLine("Nummer {0}: {1} {2}",
                               i, adb[i].Vorname, adb[i].Name);
        adb[1] = new Person("Ilse", "Golter");
        Console.WriteLine();
        for (int i = 0; i < adb.N; i++)
            Console.WriteLine("Nummer {0}: {1} {2}",
                               i, adb[i].Vorname, adb[i].Name);
    }
}

```

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 Kollektions-Klasse **ArrayList** (siehe Abschnitt 5.3.8) realisieren können. 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 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 Primzahl-Kandidaten, die initial alle natürlichen Zahlen bis zu einer Obergrenze *K* enthält, also {1, 2, 3, ..., *K*}.

- Im ersten Schritt werden alle echten Vielfachen der Basiszahl 2 (also 4, 6, ...) aus der Kandidatenmenge gestrichen.
- Dann geschieht iterativ folgendes:
 - Als neue Basis b wird die kleinste Zahl gewählt, welche folgende Bedingungen erfüllt:
 - b ist größer als die vorherige Basiszahl.
 - b ist im bisherigen Verlauf nicht gestrichen worden.
 - Die echten Vielfachen der neuen Basis (also $2 \cdot b$, $3 \cdot b$, ...) werden aus der Kandidatenmenge gestrichen.
- Das Verfahren kann enden, wenn für eine neue Basis b gilt:

$$b > \sqrt{K}$$

Es kann keine Streichkandidaten mehr geben, denn: Ist eine natürliche Zahl $n \leq K$ keine Primzahl, dann gibt es natürliche Zahlen $x, y \in \{2, 3, \dots\}$ mit

$$n = x \cdot y, \text{ mit } x \leq \sqrt{K}$$

Wegen $\sqrt{K} < b$ sind aber zu x bereits alle Vielfachen kleiner oder gleich K gestrichen worden, also insbesondere auch n . 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. Die Primzahlen kleiner oder gleich 18 lauten 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 2 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.

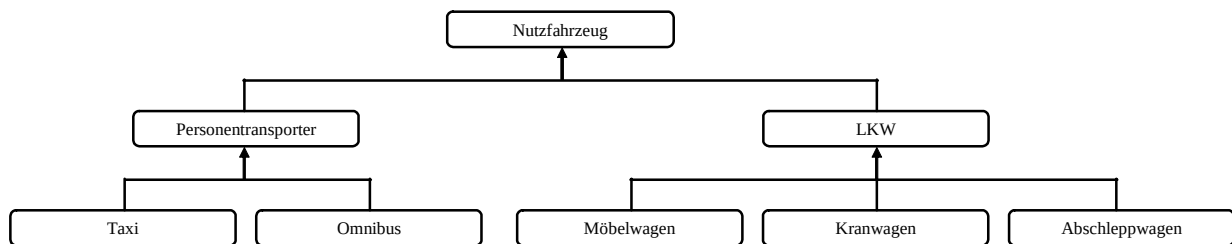
6 Vererbung und Polymorphie

Im Manuskript war schon mehrfach davon die Rede, dass sich die .NET – Klassen nicht auf *einer* Ebene befinden, sondern 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 die Nutzfahrzeuge ihres Fuhrparks folgendermaßen klassifizieren:

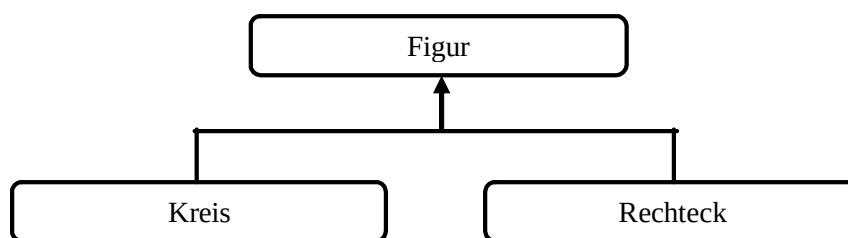


Einige Merkmale sind für alle Nutzfahrzeuge relevant (z.B. Anschaffungspreis, momentane Position, maximale Geschwindigkeit), andere betreffen nur spezielle Klassen (z.B. Anzahl der Fahrgäste, maximale Anhängelast, Hebekraft des Krans). Ebenso sind einige Handlungsmöglichkeiten bei allen Nutzfahrzeugen 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 Demonstrationsbeispielen bewegen wir uns in einem bescheideneren Rahmen und betrachten eine einfache Hierarchie mit Klassen für geometrische Figuren:

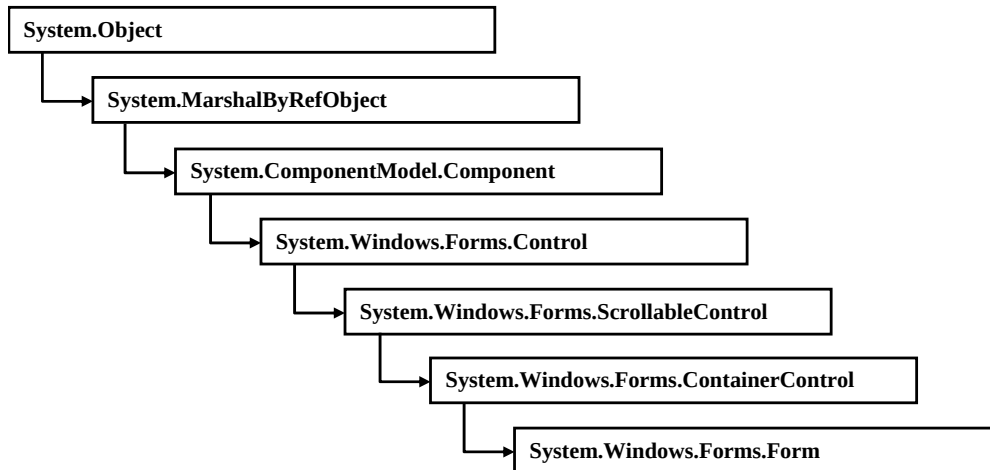


Die Vererbungstechnik der OOP

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 Windows-Anwendungen:



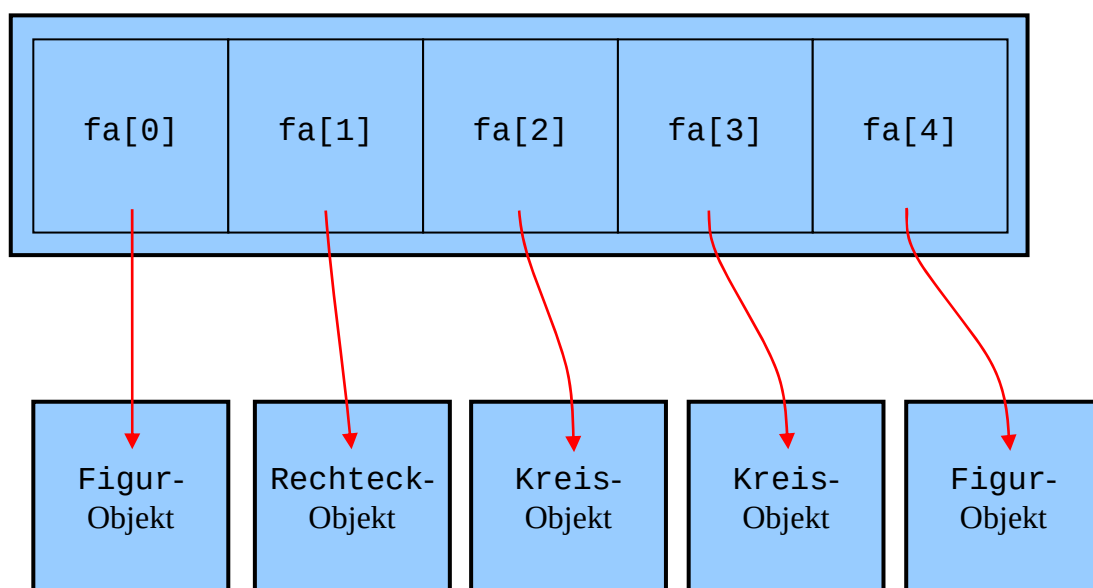
Wir werden die zentralen Klassen unserer GUI-Programme aus FCL-Klassen (z.B. **Forms**) ableiten, so dass die entstehenden Objekte eine reichhaltige Handlungskompetenz erben.

Polymorphie

Die Vererbungstechnologie der OOP erlaubt nicht nur die bequeme Definition von Klassenhierarchien, sondern bietet auch eine spezielle Flexibilität: Über Variablen vom Typ einer gemeinsamen Basisklasse lassen sich Objekte aus unterschiedlichen Klassen verwalten. Zwar können dabei nur Methoden aufgerufen werden, die schon in der Basisklasse bekannt sind, doch kommen ggf. die überschreibenden Realisationen der *abgeleiteten* Klasse zum Einsatz. Es ist also nicht ungewöhnlich, dass z.B. ein Array vom Basisklassentyp Objekte aus verschiedenen (abgeleiteten) Klassen aufnimmt, die sich beim Aufruf „derselben“ Methode unterschiedlich verhalten.

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**



Befragt man ein Objekt z.B. über die Methode **wo()** nach seiner Position, dann äußert es sich je nach Gestalt auf unterschiedliche Weise.

Wenn Objekte aus verschiedenen Klassen, die über Referenzvariablen eines gemeinsamen Basistyps verwaltet werden, auf dieselbe Botschaft unterschiedlich (jeweils unterklassentypisch) reagieren, dann spricht man von **Polymorphie**.

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 noch weit verbreitete Praxis, vorhanden Code per *Cut & Paste* in neuen Projekten zu verwenden, hat gegenüber einer sorgfältig geplanten Klassenhierarchie offensichtliche Nachteile.

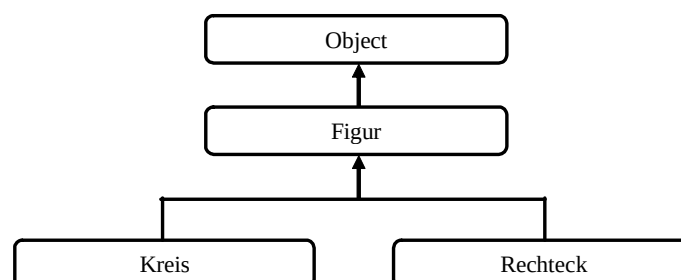
Natürlich kann auch C# nicht garantieren, dass jede Klassenhierarchie exzellent entworfen ist und bis in alle Ewigkeit von einer stetig wachsenden Programmierer-Gemeinde eingesetzt wird.

Auch die *Polymorphie* verbessert die Chancen für produktives Software-Recycling. Dank dieser OOP-Technik kann eine Software-Komponente auch mit Objekten aus solchen Klassen kooperieren, die zum Zeitpunkt ihrer Übersetzung noch gar nicht existierten.

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

Im .NET – Framework stammen *alle* Klassen und Strukturen (sowohl die in der FCL enthaltenen als auch die durch Programmierer definierten) von der Klasse **Object** aus dem Namensraum **System** ab. Wird (wie bei unseren bisherigen Beispielen) in der Definition einer Klasse keine Basis-klasse angegeben, dann stammt sie auf direktem Wege von **Object** ab.

Die oben dargestellte Klassenhierarchie zum Figuren-Übungsbeispiel muss also folgendermaßen vervollständigt werden:

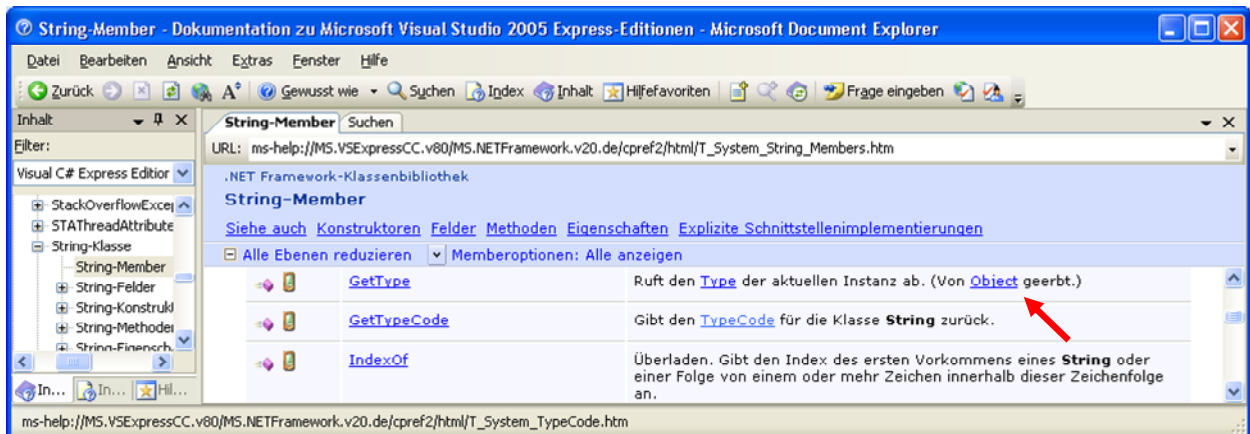


Auch die Strukturen sind in die globale Hierarchie eingeordnet: Sie stammen implizit von der Klasse **ValueType** im Namensraum **System** ab, die wiederum direkt von der Urahn-Klasse **Object** erbt (vgl. Abschnitt 5.2). Aus einer Struktur kann aber weder eine andere Struktur noch eine Klasse abgeleitet werden.

Jede Klasse bzw. Struktur erbt alle Merkmale und Handlungskompetenzen aus der eigenen Abstammungslinie von der Urahnklasse **Object** beginnend. Folglich kann z.B. jedes Objekt und jede Struktur-Instanz die in der Urahnklasse definierte Methode **GetType()** ausführen, die einen String mit der Typbezeichnung liefert:

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 SDK-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, Vererbungshierarchien aktiv zu gestalten.

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 sowie zwei Konstruktoren besitzt:

```
using System;
public class Figur {
    int xpos = 100, ypos = 100;
    public Figur(int xpos_, int ypos_) {
        xpos = xpos_;
        ypos = ypos_;
        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 {
    int radius = 75;
    public Kreis(int xpos_, int ypos_, int rad_) : base(xpos_, ypos_) {
        radius = rad_;
        Console.WriteLine("Kreis-Konstruktor");
    }
    public Kreis() {}
}
```

Die **Kreis**-Klasse erbt die beiden Positionsvariablen sowie die Methode **Wo()** und ergänzt eine zusätzliche Instanzvariable für den Radius.

Es wird ein expliziter **Kreis**-Konstruktor definiert, der über das Schlüsselwort **base** den Konstruktor der übergeordneten Klasse aufruft. Weil die **Figur**-Instanzvariablen noch als **private** deklariert sind, wäre dem **Kreis**-Konstruktor ohnehin kein direkter Zugriff erlaubt. In Abschnitt 6.4 werden wir uns mit einigen Regeln für **base**-Konstruktoren beschäftigen.

Konstruktoren werden generell *nicht* vererbt, so dass in der `Kreis`-Klasse ein parameterfreier Konstruktor neu definiert wird, obwohl auch die Basisklasse einen solchen Konstruktor besitzt (natürlich mit anderem Namen!). Wir werden allerdings sehen, dass bei der Entstehung eines Objekts der abgeleiteten Klasse stets ein Konstruktor aus *jeder* Basisklasse entlang der Ahnenreihe beteiligt ist.

Das folgende Programm erzeugt ein Objekt aus der Basisklasse und ein Objekt aus der abgeleiteten Klasse:

Quellcode	Ausgabe
<pre>using System; class Prog { static void Main() { Figur fig = new Figur(50, 50); Kreis krs = new Kreis(10, 10, 5); } }</pre>	Figur-Konstruktor Figur-Konstruktor Kreis-Konstruktor

Im .NET - Framework (so auch in C#) ist *keine Mehrfachvererbung* möglich: Man kann also in einer Klassendefinition nur *eine* Basisklasse angeben. Obwohl im Sinne einer realitätsnahen Modellierung eine Mehrfachvererbung gelegentlich wünschenswert wäre (z.B. für Hochschulangehörige, die sowohl zu den Studierenden als auch zu den Dozenten gehören), hat man diese Möglichkeit wegen einiger Risiken bewusst *nicht* aus C++ übernommen.

6.3 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 int xpos = 100, ypos = 100;
    public Figur(int xpos_, int ypos_) {
        xpos = xpos_;
        ypos = ypos_;
        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` als **protected** deklariert, können sie in der `Kreis`-Methode `SetzePos()` verändert werden:

```
using System;
public class Kreis : Figur {
    int radius = 75;
    public Kreis(int xpos_, int ypos_, int rad_) : base(xpos_, ypos_) {
        radius = rad_;
        Console.WriteLine("Kreis-Konstruktor");
    }
    public void SetzePos(int xpos_, int ypos_) {
        xpos = xpos_;
        ypos = ypos_;
    }
    public Kreis() {}
}
```

Es ist zu beachten, dass hier *geerbte Instanzvariablen* von `Kreis`-Objekten verändert werden. Auf die `xpos`-Eigenschaft von `Figur`-Objekten haben auch die Methoden der `Kreis`-Klasse *keinen* Zugriff.

Methoden fremder Klassen haben auf **protected**-deklarierte Member *keinen* Zugriff, z.B.:

```
using System;
class Prog {
    static void Main() {
        Kreis krs = new Kreis(10, 10, 5);
        // krs.xpos = 77;           verboten
        krs.SetzePos(77, 99); // erlaubt
    }
}
```

Auch bei instanz- oder klassenbezogenen *Methoden* ist der Modifikator **protected** erlaubt, z.B.:

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

6.4 base-Konstruktoren und Initialisierungs-Sequenzen

Zwar werden Konstruktoren nicht vererbt, doch ist bei der Entstehung eines Objekts der abgeleiteten Klasse stets ein Konstruktor der Basisklasse implizit oder explizit beteiligt.

Den *expliziten* Aufruf eines Basisklassen-Konstruktors erledigt man im Kopfbereich eines Unterklassen-Konstruktors über das Schlüsselwort **base**, z.B.:

```
public Kreis(int xpos_, int ypos_, int rad_) : base(xpos_, ypos_) {
    radius = rad_;
    Console.WriteLine("Kreis-Konstruktor");
}
```

Dadurch ist es auch möglich, geerbte Instanzvariablen zu initialisieren, die in der Basisklasse als **private** deklariert wurden.

In einem Unterklassen-Konstruktor *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 Basisklassen-Konstruktor wird übrigens auch vom Standard-Konstruktor der abgeleiteten Klasse aufgerufen.

Beim Erzeugen eines Unterklassen-Objektes laufen folgende Initialisierungs-Maßnahmen ab:

- Alle Instanzvariablen (auch die geerbten) werden angelegt und mit den typspezifischen Nullwerten initialisiert.
- Der Unterklassen-Konstruktor führt nacheinander folgende Aktionen aus:

- o Die Instanzvariablen der Klasse erhalten ggf. den in ihrer Deklaration angegebenen Initialisierungswert. Den zugehörigen MSIL-Code erzeugt der Compiler automatisch.
- o Es folgt der Aufruf eines Basisklassen-Konstruktors.
- o Nach Beendigung des Basisklassen-Konstruktors wird der Rumpf des Unterklassen-Konstruktors ausgeführt.
- Im aufgerufenen Basisklassen-Konstruktor läuft dieselbe Sequenz ab (Instanzvariablen der Klasse initialisieren, Aufruf des Basisklassen-Konstruktors, 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, 200, 50)
```

geschieht:

- Alle Instanzvariablen (auch die geerbten) werden angelegt und mit den typspezifischen Nullwerten initialisiert.
- Der `Kreis`-Konstruktor führt die Initialisierung `radius = 75` aus.
- Der `Figur`-Konstruktor (mit Positionsparametern) startet und führt die Initialisierungen `xpos = 100` sowie `ypos = 100` 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, 200)` wird ausgeführt, wobei `xpos` und `ypos` die Werte 150 bzw. 200 erhalten.
- Der Rumpf des Konstruktoraufrufs `Kreis(150, 200, 50)` wird ausgeführt, wobei `radius` den Wert 50 erhält.

6.5 Geerbte Methoden durch spezialisierte Varianten verdecken

Eine geerbte Basisklassen-Methode 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 Objektes ausgibt. In der `Kreis`-Klasse kann aber eine bessere Ortsangaben-Methode realisiert werden, weil hier auch die rechte untere Ecke definiert ist:¹

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

```

using System;
public class Kreis : Figur {
    int radius = 75;
    public Kreis(int xpos_, int ypos_, int rad_) : base(xpos_, ypos_) {
        radius = rad_;
    }
    new public void Wo() {
        base.Wo();
        Console.WriteLine("Unten Rechts:\t(" + (xpos + 2 * radius) +
            ", " + (ypos + 2 * radius) + ")");
    }
}

```

Im Definitionskopf der verdeckenden Methode sollte man den Modifikator **new**¹ angeben, um die folgende Warnung des Compilers zu vermeiden:

```

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

```

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 artgerechtes Verhalten:

Quellcode	Ausgabe
<pre> using System; class Prog { static void Main() { Figur f = new Figur(10, 20); f.Wo(); Kreis k = new Kreis(50, 50, 10); k.Wo(); } } </pre>	<pre> Oben Links: (10, 20) Oben Links: (50, 50) Unten Rechts: (70, 70) </pre>

Auch bei den vom Urahntyp **Object** geerbten Methoden kommt ein Verdecken in Frage, z.B. bei der Methode **ToString()**, deren **Object**-Implementierung lediglich den Klassennamen ausgibt.

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*.

Trotz identischer Signatur bleibt eine Verdeckung aus, wenn die Methode der abgeleiteten Klasse aufgrund ihrer Schutzstufe nicht sichtbar ist, z.B.:

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

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

<pre>class Prog { static void Main() { Basisklasse b = new Basisklasse(); b.M(); Abgeleitet a = new Abgeleitet(); a.M(); } }</pre>	

Neben objektbezogenen können auch *statische* Methoden verdeckt werden, z.B.:

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

Schließlich ist das Verdecken nicht nur bei Methoden erlaubt, sondern auch bei *Eigenschaften* und *Indexern*.

6.6 Geerbte Felder verdecken

Auch geerbte Instanz- bzw. 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 Sohn für eine Instanzvariable einen Namen, der bereits eine Variable der beerbten Klasse Vater bezeichnet, dann wird die Basisvariable verdeckt. Sie ist jedoch weiterhin vorhanden und kommt in folgenden Situationen zum Einsatz:

- Vom Vater geerbte Methoden greifen weiterhin auf die Vater-Variable zu, während die zusätzlichen Methoden der Sohn-Klasse auf die Sohn-Variable zugreifen.
- Wie bei Methoden steht die verdeckte Variante über das Schlüsselwort **base** weiterhin zur Verfügung.

Im folgenden Beispielprogramm führt ein Sohn-Objekt eine Vater- und eine Sohn-Methode aus, um die beschriebenen Zugriffsvarianten zu demonstrieren:

Quellcode	Ausgabe
<pre> using System; class Vater { protected string x = "Vast"; public void VM() { Console.WriteLine("x in Vater-Methode:\t"+x); } } class Sohn : Vater { new int x = 333; public void SM() { Console.WriteLine("x in Sohn-Methode:\t"+x); Console.WriteLine("base-x in Sohn-Methode:\t"+ base.x); } } class Prog { static void Main() { Sohn so = new Sohn(); so.VM(); so.SM(); } } </pre>	<pre> x in Vater-Methode: Vast x in Sohn-Methode: 333 base-x in Sohn-Methode: Vast </pre>

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.

```

6.7 Überschreiben von Methoden (Polymorphie)

Statt eine geerbte Methode zu *verdecken* (siehe Abschnitt 6.5), kann man sie auch *überschreiben*, um bei gleicher Signatur ein spezialisiertes Unterklassenverhalten zu realisieren.

Um den wesentlichen Unterschied zwischen den beiden Ersetzungsarten erklären zu können, muss zunächst eine wichtige Option der objektorientierten Programmierung behandelt werden: Eine Basisklassen-Referenzvariable 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 Basisklasse sowie in Bezug auf alle indirekten Basisklassen in der kompletten 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.

Objekte aus verschiedenen Klassen über Referenzvariable von einem gemeinsamen Basistyp anzusprechen, ermöglicht in vielen Situationen eine flexible und zukunftssichere Softwareentwicklung. Im folgenden Beispiel wird zwar das Prinzip vorgeführt, sein potentieller Nutzen jedoch kaum belegt²:

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

² Obwohl das Beispiel auf den ersten Blick bestens in den Vererbungs-Abschnitt zu passen scheint, liegt seine Alltagssemantik doch quer zur Spezialisierungsrelation zwischen einer Basis- und einer Unterklasse: Jedes Objekt der Klasse *Sohn* steht in der „ist-ein“-Relation zur Klasse *Vater*, während im Alltagsleben durchaus nicht jeder Sohn auch ein Vater ist.

```

class Vater {
}
class Sohn : Vater {
}
class Prog {
    static void Main() {
        Vater va = new Vater();
        Vater so = new Sohn();
    }
}

```

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*.

Wenn Objekte aus verschiedenen Klassen, die über Referenzvariablen eines gemeinsamen Basistyps verwaltet werden, auf dieselbe Botschaft unterschiedlich (jeweils unterklassentypisch) reagieren, dann spricht man von *Polymorphie*.

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

Ersetzungsart	Unterstützung der Polymorphie	Syntax
Überschreiben	Ja	Im Kopf der Basisklassenmethode muss der Modifikator virtual <i>und</i> 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.

Zur Demonstration der Polymorphie erweitern wir die Klasse *Vater* aus dem letzten Beispiel um eine virtuelle, d.h. mit dem Modifikator **virtual** als überschreibbar deklarierte, Methode *Hallo()*:

```

using System;
class Vater {
    public virtual void Hallo() {
        Console.WriteLine("Hallo-Methode des Vaters");
    }
}

```

In der abgeleiteten Klasse *Sohn* wird die väterliche *Hallo()*-Methode überschrieben und eine zusätzliche Methode *NurSo()* ergänzt:

```

using System;
class Sohn : Vater {
    override public void Hallo() {
        Console.WriteLine("Hallo-Methode des Sohnes");
    }
    public void NurSo() {
        Console.WriteLine("NurSo-Methode des Sohnes");
    }
}

```

Ein Array vom Typ Vater kann nach obigen Erläuterungen Referenzen auf Väter und Söhne aufnehmen, z.B.:

```
using System;
class Prog {
    static void Main() {
        Vater[] varray = new Vater[3];
        varray[0] = new Vater();
        varray[1] = new Sohn();
        varray[0].Hallo();
        varray[1].Hallo();

        Console.WriteLine("\nWollen Sie zum Abschluss noch einen"+
            " Vater oder einen Sohn erleben?" +
            "\nWählen Sie durch Abschicken von \"v\" oder \"s\": ");
        if (Console.ReadLine().ToUpper()[0] == 'V')
            varray[2] = new Vater();
        else
            varray[2] = new Sohn();
        Console.WriteLine();
        varray[2].Hallo();
    }
}
```

Beim Ausführen der Hallo()-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*):

Hallo-Methode des Vaters
Hallo-Methode des Sohnes

Wollen Sie zum Abschluss noch einen Vater oder einen Sohn erleben?
Wählen Sie durch Abschicken von "v" oder "s": s

Hallo-Methode des Sohnes

Zum „Beweis“, dass tatsächlich eine späte Bindung erfolgt, darf im Beispielprogramm die Klasse des Array-Elementes varray[2] vom Benutzer festgelegt werden.

Über eine Vater-Referenzvariable, die auf ein Sohn-Objekt zeigt, sind *Erweiterungen* der Sohnklasse *nicht* unmittelbar zugänglich. Wenn (auf eigene Verantwortung des Programmierers) eine Basisklassenreferenz als Unterklassenreferenz behandelt werden soll, um eine unterklassenspezifische Methode oder Variable anzusprechen, dann muss der Typumwandlungs-Operator verwendet werden, z.B.:

```
((Sohn)varray[1]).NurSo();
```

Geschieht dies zu Unrecht, tritt ein Ausnahmefehler auf, z.B.:

Unbehandelte Ausnahme: System.InvalidCastException: Das Objekt des Typs Vater kann nicht in Typ Sohn umgewandelt werden.
bei Prog.Main()

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

```
if (varray[1] is Sohn)
    ((Sohn)varray[1]).NurSo();
```

6.8 Abstrakte Methoden und Klassen

Um die eben beschriebene gemeinsame Verwaltung von Objekten aus diversen Unterklassen über einen Array vom Basisklassentyp realisieren und dabei Polymorphie nutzen zu können, müssen die betroffenen Methoden in der Basisklasse vorhanden sein. Wenn es für die Basisklasse zu einer Me-

thode keine sinnvolle Implementierung gibt, kann man die Methode dort als **abstract** deklarieren und auf eine Implementierung verzichten, z.B.:

```
abstract class Vater {
    public abstract void Hallo();
}
```

Abstrakte Methoden sind grundsätzlich virtuell (überschreibbar, vgl. Abschnitt 6.7), wobei das Schlüsselwort **virtual** überflüssig und verboten ist.

Enthält eine Klasse mindestens eine abstrakte Methode, dann handelt es sich um eine **abstrakte Klasse**, und die Klassendefinition muss mit dem Modifikator **abstract** eingeleitet 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.

Im aktuellen Beispiel werden aus dem abstrakten Vater zwei konkrete Klassen abgeleitet:

```
using System;
class Sohn : Vater {
    public override void Hallo() {
        Console.WriteLine("Hallo-Methode des Sohnes");
    }
}
class Tochter : Vater {
    public override void Hallo() {
        Console.WriteLine("Hallo-Methode der Tochter");
    }
}
```

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() { Vater[] varray = new Vater[2]; varray[0] = new Tochter(); varray[1] = new Sohn(); varray[0].Hallo(); varray[1].Hallo(); } }</pre>	<pre>Hallo-Methode der Tochter Hallo-Methode des Sohnes</pre>

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 `Ac1` zum Abfragen eines Passwortes, will ihr Programmierer eventuell verhindern, dass `Passwd()` in einer von `Ac1` abstammenden Klasse `Bc1` überschrieben wird. Damit kann dem Nutzer der Klasse `Bc1` die ursprüngliche Funktionalität von `Passwd()` garantiert werden.

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

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 wieder analog für Eigenschaften und Indexer.

Die eben für das Versiegeln von Methoden genannten Sicherheitsüberlegungen können auch zum Entschluss führen, eine komplette **Klasse** mit dem Schlüsselwort **sealed** zu fixieren, so dass sie zwar verwendet, aber nicht beerbt werden kann.

6.10 Übungsaufgaben zu Kapitel 6

1) Warum kann der folgende Quellcode nicht übersetzt werden?

```
using System;
class Vater {
    int iva = 3;
    public Vater(int iva_) {iva = iva_;}
    public virtual void Hallo() {
        Console.WriteLine("Hallo-Methode des Vaters");
    }
}
class Sohn : Vater {
    override public void Hallo() {
        Console.WriteLine("Hallo-Methode des Sohnes");
    }
}
class Prog {
    static void Main() {
        Sohn s = new Sohn();
        s.Hallo();
    }
}
```

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

```
public class Figur {
    int xpos, ypos;
}

class Kreis : Figur {
    int radius;
    public void SetzePos(int xpos_, int ypos_) {
        xpos = xpos_;
        ypos = ypos_;
    }
}
```

Trotzdem erlaubt der Compiler den Kreis-Objekten keinen direkten Zugriff auf ihre geerbten Instanzvariablen xpos und ypos (in der Methode SetzePos()). Wie ist das Problem zu erklären und zu lösen?

3) Erweitern Sie das Beispiel mit den geometrischen Figuren (siehe Abschnitt 6.2) um eine aus `Figur` abgeleitete Klasse `Rechteck` mit zusätzlichen Instanzvariablen für Breite und Höhe und einer `wo()`-Methode, welche die geerbte `Figur`-Version überschreibt. Erstellen Sie ein Testprogramm, das polymorphe Methodenaufrufe demonstriert.

4) 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 Interfaces

Zu vielen Klassen oder Strukturen der FCL führt die SDK-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 gibt die SDK – Dokumentation auch den Stammbaum wieder, der im Beispiel **String** die direkte Abstammung von der Urahnkasse **Object** zeigt:

```
System.Object  
    System.String
```

Am Anfangsbuchstaben *I* sind in der SDK-Dokumentation zuverlässig die von einer Klasse oder Struktur implementierten *Interfaces* zu erkennen. Diese Verpflichtungserklärungen enthalten wichtige Informationen über die Handlungskompetenzen der Instanzen eines Typs.

Im konkreten Fall wird u.a. über die Klasse **String** ausgesagt:

- **IComparable**

Die Klasse implementiert das Interface **IComparable** (aus dem Namensraum **System**), besitzt also eine Methode namens **CompareTo()**. Infolgedessen können die Objekte in einem **String**-Array mit der (statischen) Methode **Array.Sort()** bequem sortiert werden, z.B.:

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>

- **IClonable**

Die Klasse **String** implementiert auch das Interface **IClonable** (aus dem Namensraum **System**) und besitzt folglich eine Methode **Clone()**, welche eine Kopie des angesprochenen Objektes erzeugt.

Die mit .NET 2.0 eingeführten *generischen* Interfaces, zu erkennen am spitz eingeklammerten Typparameter, ignorieren wir vorläufig noch.

Ein Interface definiert Verhaltenskompetenzen abstrakt durch Methoden, Eigenschaften, Indexer und Ereignisse (siehe unten) ohne Anweisungsteil. Im Vergleich zu einer Klasse oder Struktur ist also lediglich eine Schnittstellendefinition vorhanden, so dass man Interfaces zu Recht auch als **Schnittstellen** bezeichnet. Man könnte sie in erster Näherung auch als Klassen mit ausschließlich abstrakten Methoden beschreiben.

Interfaces können auch als **Referenzdatentypen** verwendet werden. Zwar sind keine *Objekte* von diesem Datentyp denkbar (so wenig wie bei abstrakten Klassen), aber *Referenzvariablen* sind erlaubt und als Abstraktionsmittel sehr nützlich. Sie dürfen auf Objekte aus *allen* Klassen zeigen, welche die Schnittstelle implementieren. Auch implementierende Strukturen werden unterstützt, wobei aber Boxing-Operationen erforderlich sind. Somit können Objekte und Struktur-Instanzen unabhängig von den Vererbungsbeziehungen ihrer Typen gemeinsam verwaltet werden, wobei Methodenaufrufe grundsätzlich polymorph erfolgen (mit später bzw. dynamischer Bindung).

Implementiert eine Klasse oder Struktur ein Interface, dann ...

- muss sie alle damit verbundenen Verpflichtungen erfüllen, also z.B. die im Interface enthaltenen abstrakten Methoden implementieren.
- können bestimmte Methoden mit Instanzen des Typs ausgeführt werden.
So kann z.B. ein eindimensionales Array-Objekt mit der Methode

Array.Sort(Array ar)

sortiert werden, wenn sein Elementdatentyp das **Comparable**-Interface implementiert.

- werden Variablen vom Typ dieser Klasse vom Compiler akzeptiert, wo der Interface-Datentyp vorgeschrieben ist.

7.1 Interfaces definieren

Wenngleich das *Implementieren* einer Schnittstelle im Programmieralltag eher relevant wird als das *Definieren*, fangen wir mit Letzterem an, weil dabei Inhalt und Funktion einer Schnittstelle gut zu erkennen sind. Betrachten wir als Beispiel die Definition der Schnittstelle **Comparable** aus der FCL:

```
public interface Comparable {
    int CompareTo(object obj);
}
```

Im Schnittstellenkopf ist neben dem Schlüsselwort **interface** und dem Namen noch der Modifikator **public** mit der üblichen Bedeutung erlaubt. Im Schnittstellenrumpf werden abstrakte Methoden, Eigenschaften, Indexer und Ereignisse (siehe unten) aufgeführt, deren Rumpf jeweils durch ein Semikolon ersetzt ist. In einer Interface-Definition wird also festgeschrieben, dass die Instanzen einer implementierenden Klasse bzw. Struktur bestimmte Methodenaufrufe etc. beherrschen müssen, wobei die Implementierung offen bleibt.

Einige allgemeine Regeln für Schnittstellendefinitionen:

Modifikator public	Wird public <i>nicht</i> angegeben, ist die Schnittstelle nur innerhalb ihres Assemblies verwendbar (Schutzstufe internal).
Modifikator abstract	Weil Schnittstellen grundsätzlich abstract sind, wird der Modifikator <i>nicht</i> angegeben.
Schlüsselwort interface	Das obligatorische Schlüsselwort dient zur Unterscheidung von Klassen- und Strukturdefinitionen.
Schnittstellenname	Per Konvention beginnt der Interfacename mit einem großen <i>I</i> .
Vererbung	Ein Interface kann andere Interfaces beerben (bzw. erweitern), wobei dieselbe Syntax wie bei Klassen zu verwenden ist. Es gilt: • Während bei <i>Klassen</i> die (z.B. von C++ bekannte) Mehrfachvererbung <i>nicht</i> unterstützt wird, ist sie bei <i>Schnittstellen</i> möglich (und oft auch sinnvoll). • Die Hierarchie der Interfaces ist unabhängig von der Klassenhierarchie.

Methodendefinition	In einer Schnittstelle sind alle Methoden grundsätzlich public und abstract (folglich auch virtual). Die drei Schlüsselwörter sind überflüssig und zudem verboten. Die grundsätzlich virtuellen und damit zu überschreibenden Schnittstellenmethoden dürfen nicht als static definiert werden, weil Klassenmethoden nicht überschreibbar sind.
erlaubte Interface-Member	Als Interface-Member sind <i>nur Methoden, Eigenschaften, Indexer und Ereignisse</i> erlaubt. Insbesondere sind Felder verboten.

7.2 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 **Comparable**, damit `Figur`-Arrays bequem sortiert werden können:

```
using System;
public class Figur : Comparable {
    public string name = "unbenannt";
    public int xpos, ypos;
    public Figur(string name_, int xpos_, int ypos_) {
        name = name_; xpos = xpos_; ypos = ypos_;
    }
    public Figur() {}
    public int CompareTo(object obj) {
        return (xpos - ((Figur)obj).xpos);
    }
}
```

Alle Methoden, Eigenschaften, Indexer und Ereignisse (siehe unten) einer im Kopf angemeldeten Schnittstelle müssen implementiert werden. Nach der in Abschnitt 7.1 wiedergegebenen **Comparable**-Definition ist im aktuellen Beispiel aber 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 eine negative Zahl zurück gemeldet.
- Haben beide Figuren in der linken oberen Ecke dieselbe X-Koordinate, lautet die Antwort 0.
- Ansonsten wird eine positive Zahl gemeldet.

Damit wird eine *Anordnung* der `Figur`-Objekte definiert, und einem erfolgreichen Sortieren (z.B. per **Array.Sort()**) steht nichts mehr im Wege.

Um über den (per **Comparable**-Definition vorgeschriebenen) Parameter vom Typ **Object** die speziellen Member von `Figur`-Objekten ansprechen zu können, ist eine explizite Typanpassung erforderlich:

```
((Figur)obj).xpos)
```

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 7.3).

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.

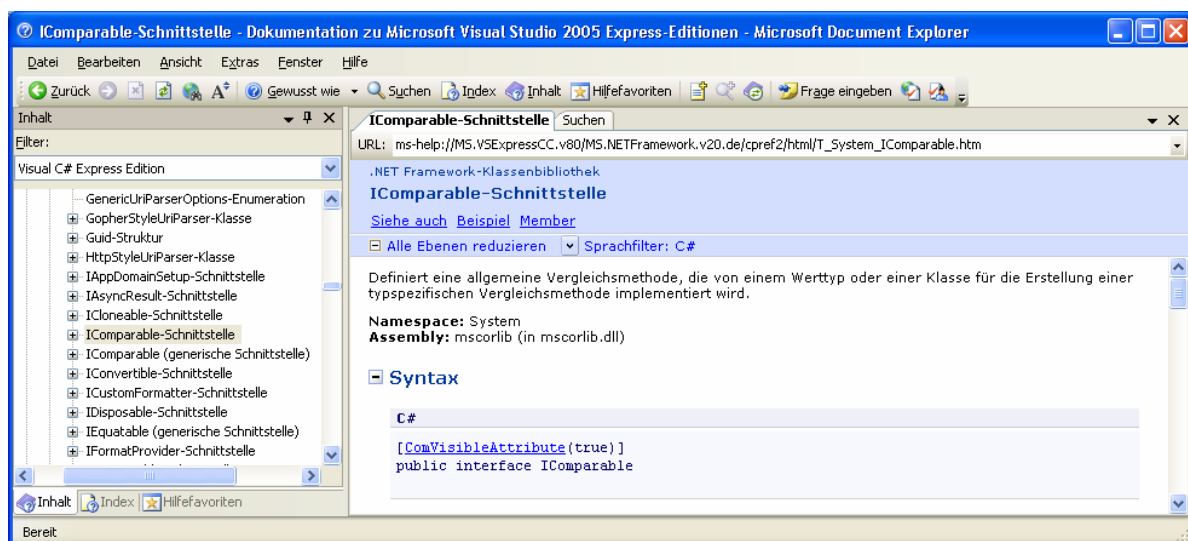
Auch Schnittstellen ändern nichts daran, dass für C# - Klassen eine Mehrfachvererbung ausgeschlossen ist. Diese Möglichkeit wurde wegen einiger Risiken bewusst *nicht* aus C++ übernommen. Allerdings erlauben Schnittstellen in vielen Fällen eine Ersatzlösung, denn:

- Eine Klasse darf beliebig viele Schnittstellen implementieren, so dass ihre Objekte entsprechend viele Datentypen erfüllen.
Wie wir inzwischen wissen, wird der Klasse 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 auf Felder mit *Kreis*-Elementen angewandt werden.

Weil die Schnittstellenhierarchie von der Klassenhierarchie unabhängig ist, kann ein Interface von beliebigen Klassen und Strukturen implementiert werden.

In der FCL sind die Interfaces zusammen mit den anderen Typen (z.B. Klassen und Strukturen) in Namensräumen organisiert, z.B.:



7.3 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 Objekte einer Klasse zeigen, welche die Schnittstelle implementiert, z.B.:

Quellcode	Ausgabe
<pre>using System; interface IDemo { int SagWas(); } class Kid1 : IDemo { public int SagWas() {return 1;} } class Kid2 : IDemo { public int SagWas() {return 2;} } class Prog { static void Main() { IDemo o1 = new Kid1(), o2 = new Kid2(); Console.WriteLine("Kid1-Objekt: "+o1.SagWas()); Console.WriteLine("Kid2-Objekt: "+o2.SagWas()); } }</pre>	<pre>Kid1-Objekt: 1 Kid2-Objekt: 2</pre>

Damit wird es z.B. möglich, Instanzen von beliebigen Klassen und Strukturen, die dasselbe Interface implementieren, in einem Array gemeinsam zu verwalten.

Wird eine Struktur-Instanz einer Interface-Variablen zugewiesen, ist allerdings ein Boxing erforderlich, weil es sich um eine Referenzvariable handelt, die nur Adressen von Heap-Objekten aufnehmen kann.

Weil die Methoden einer Schnittstelle grundsätzlich virtuell sind, erfolgt ihr Aufruf mit dynamischer Bindung (polymorph).

Leider führt die Verwendung von Interface-Referenzvariablen gelegentlich zu einer mangelhaften Typsicherheit. Als Beispiel betrachten wir die in Abschnitt 7.2 vorgestellte Klasse **Figur**, die das Interface **IComparable** implementiert. Das folgende Programm wird vom Compiler nicht kritisiert, doch löst die **Sort()**-Methode eine **InvalidCastException** aus:

```
using System;
class Prog {
    static void Main() {
        IComparable[] ic = new IComparable[2];
        ic[0] = new Figur("F", 100, 100);
        ic[1] = 13;
        Array.Sort(ic);
    }
}
```

Weil auch die Struktur **System.Int32** das Interface **IComparable** implementiert,

```
public struct Int32 : IComparable, IFormattable, IConvertible,
    IComparable<int>, IEquatable<int>
```

gelingt die gemeinsame Verwaltung von **Figur**- und **Int32**-Instanzen mit **IComparable**-Referenzvariablen, jedoch kann der **int**-Wert 13 in der **Figur**-Implementation von **CompareTo()** nicht in ein **Figur**-Objekt konvertiert werden.

7.4 Übungsaufgaben zu Kapitel 7

- 1) Erstellen Sie eine Variante unserer Bruch-Klasse (vgl. z.B. Abschnitt 4.1.3), welche die Interfaces **Comparable** und **Cloneable** implementiert.
- 2) Wie unterscheiden sich Interfaces von abstrakten Klassen?
- 3) In welchem Sinn kann die .NET-Schnittstellentechnik partiell die Mehrfachvererbung von C++ ersetzen?

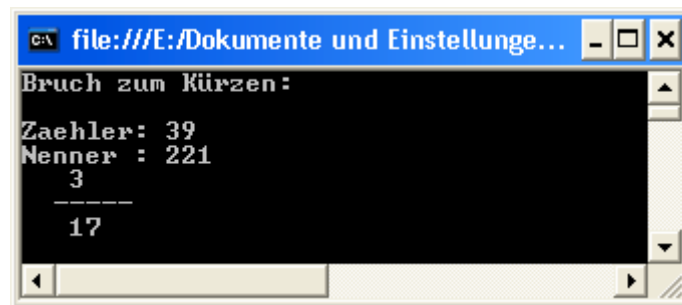
8 Einstieg in die WinForms-Programmierung

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 *controls*, *Steuerelemente* oder *widgets* (Wortkombination aus *window* und *gadgets*). Weil die Steuerelemente intuitiv und in verschiedenen Programmen weitgehend konsistent zu bedienen sind, erleichtern Sie den Umgang mit moderner Software erheblich. Zur bequemen Bedienung trägt auch die Unterstützung der **Maus** als Eingabegerät bei.

Im Vergleich zu Konsolenprogrammen geht es nicht nur intuitiver, sondern vor allem auch „ereignisreicher“ zu:¹ Eine Konsolenanwendung entscheidet in der Regel selbst darüber, welche Programmeinheit als nächstes ausgeführt wird, wenngleich das Programm auf Umgebungsbedingungen, Benutzerwünsche oder eingelesene Daten flexibel reagieren kann. 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 Paradigma** wesentlich, wobei das Laufzeitsystem als Quelle oder Vermittler 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 und Tastatur seine Wünsche artikuliert. Eine GUI-Anwendung wartet also die meiste Zeit darauf, dass eine Ihrer Methoden durch ein Ereignis aufgerufen wird.

Betrachten wir zur Illustration eine Konsolen- und eine GUI-Anwendung zum Kürzen von Brüchen. Bei der Konsolenanwendung (vgl. Abschnitt 4.1.3)

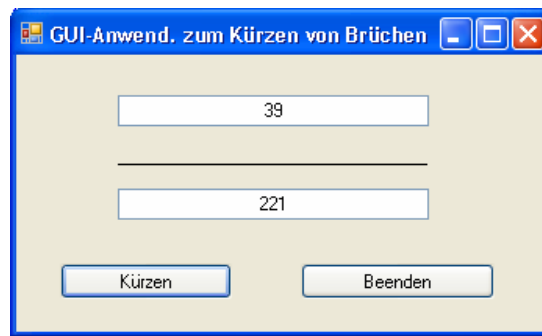


wird der gesamte Ablauf vom Programm diktiert:

- Es fragt nach dem Zähler.
- Es fragt nach dem Nenner.
- Es schreibt das Ergebnis auf die Konsole.

Im Unterschied zu diesem **programmgesteuerten Ablauf** wird bei der GUI-Variante

¹ Momentan wird bewusst ein starker Kontrast zwischen den bisher vertrauten Konsolenanwendungen und den nun vorzustellenden GUI-Anwendungen hinsichtlich der ereignisorientierten Programmierung herausgearbeitet. Allerdings kann grundsätzlich auch eine Konsolenanwendung mit Ereignissen umgehen. Wir werden z.B. in Abschnitt 11.3.3 ein Konsolenprogramm erstellen, das auf Ereignisse im Dateisystem (z.B. auf das Erstellen, Umbenennen oder Löschen von Dateien) reagiert.



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 .NET – Standardbibliothek (speziell im Namensraum **System.Windows.Forms**) außerordentlich leistungsfähige Klassen zur GUI-Programmierung, deren Verwendung durch Hilfsmittel der Entwicklungsumgebungen (z.B. Formular designer) zusätzlich erleichtert wird. Man spricht auch vom **Windows Forms - Framework** (kurz: **WinForms-Framework**) und bezeichnet die damit erstellten Fenster als **Formulare**.

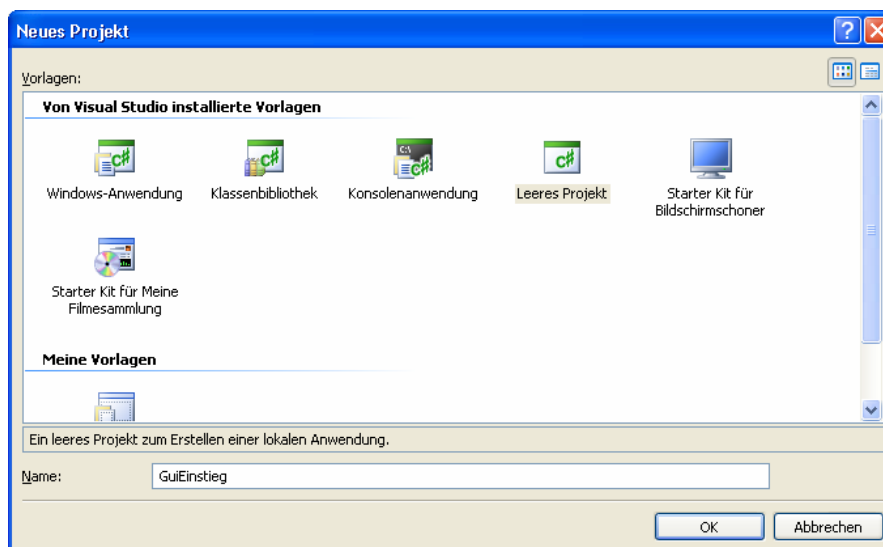
In diesem Abschnitt soll ein grundlegendes Verständnis von Aufbau und Funktionsweise einer WinForms-Anwendung vermittelt werden. Dabei verzichten wir vorübergehend auf die Assistenten der Entwicklungsumgebung, die einen relativ komplexen Quellcode produzieren, der den Blick auf das Wesentliche erschwert¹. Sobald wir die Assistentenproduktionen verstehen, steht ihrer Verwendung zur Vereinfachung von Routineaufgaben nichts im Wege.

8.1 Ein Projektrahmen zum Üben

Es wurde schon in Abschnitt 2.3.2.2 beschrieben, wie man mit der Visual C# Express Edition ein leeres Projekt anlegt. Starten Sie mit dem Schalter  oder dem Menübefehl

Datei > Neu > Projekt

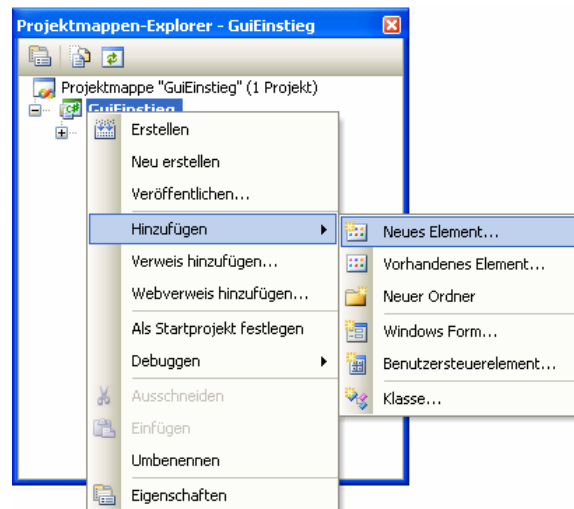
den Dialog für neue Projekte:



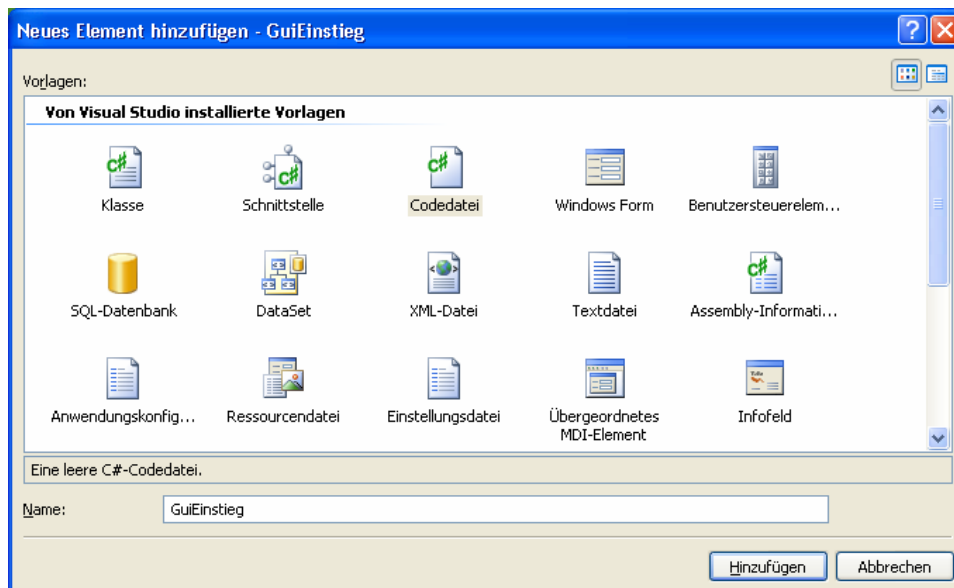
¹ Diese Vorgehensweise wird übrigens von Charles Petzold, einem exzellenten und sehr erfahrenen Autor von Büchern zur Windows-Programmierung, generell empfohlen. In seinem Buch zur Windows-Programmierung mit C# (Petzold 2002) schreibt er auf S. 45: „In diesem Buch werden Sie und ich den Code selbst schreiben“.

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

Öffnen Sie im Projektmappen-Explorer das Kontextmenü zum Projekt, und fügen Sie ein **neues Element** hinzu:



Entscheiden Sie sich für eine **Codedatei** mit geeignetem **Namen**, z.B.:



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

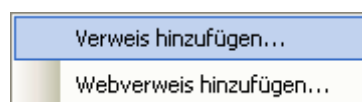
Weil wir eine WinForms-Anwendung planen (ohne eine entsprechende Projektvorlage verwendet zu haben), müssen wir folgende DLL-Assemblies in die Verweisliste des Projekts aufnehmen:

- **System.dll**
- **System.Windows.Forms.dll**

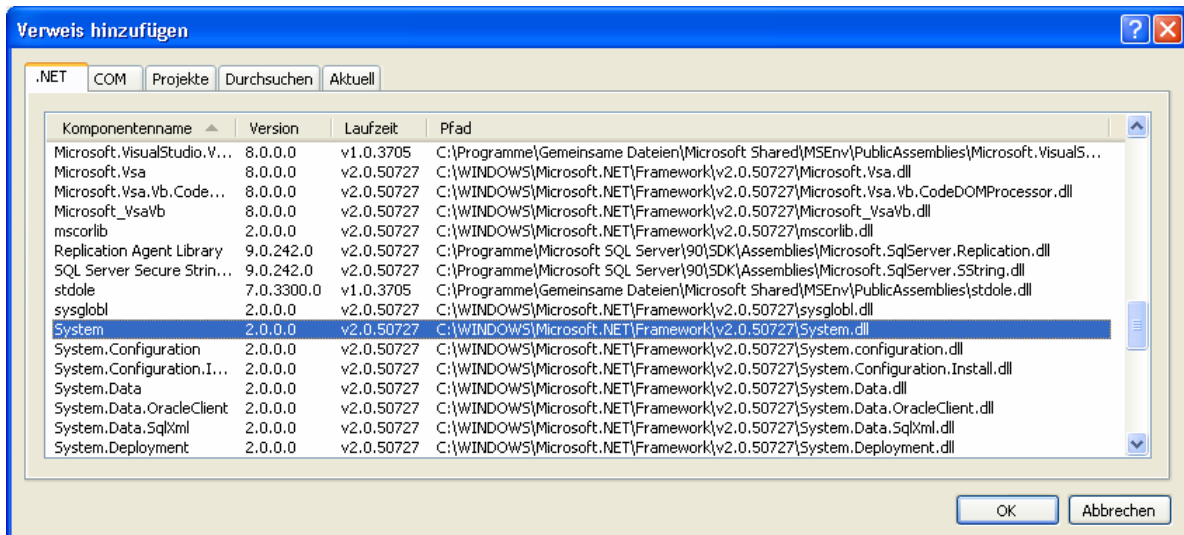
Spätere Beispielprogramme benötigen meist auch eine Referenz auf

- **System.Drawing.dll**

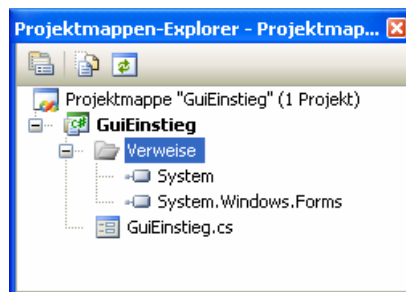
Wählen Sie im Projektmappen-Explorer aus dem Kontextmenü zum Eintrag **Verweise** die Option **Verweise hinzufügen**:



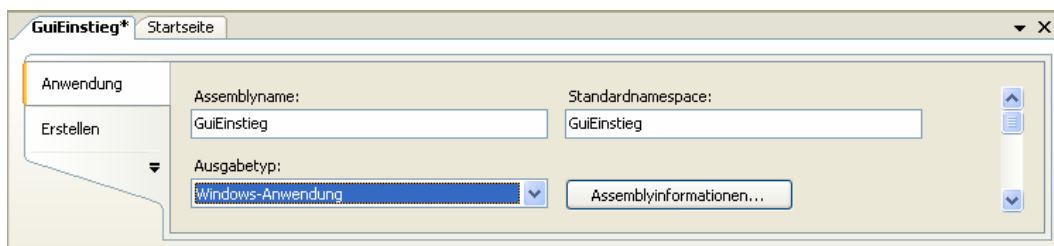
In der folgenden Dialogbox lassen sich die benötigten Verweise leicht auswählen:



Das Ergebnis im Projektmappen-Explorer:



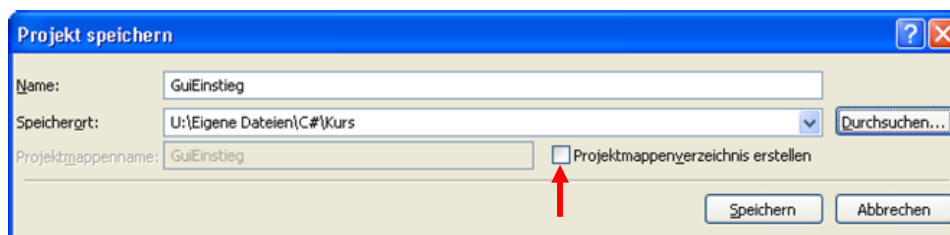
Um das Erscheinen eines Konsolenfensters beim Programmstart zu vermeiden, sollten Sie nach **Projekt > Projekteigenschaften > Anwendung** den **Ausgabotyp** auf **Windows-Anwendung** ändern:



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 Projektdatenverwaltung, die für praktisch alle im Kurs entstehenden Projekte angemessen ist.

Nun kümmern wir uns um den erforderlichen Quellcode für ein (sehr) einfaches WinForms-Programm:

8.2 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 {
    static void Main() {
        GuiEinstieg hf = new GuiEinstieg();
        hf.Text = "WinForms-Einstieg";
        Application.Run(hf);
    }
}
```

Allzu viel sollte man von einem solchen Achtzeiler 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.

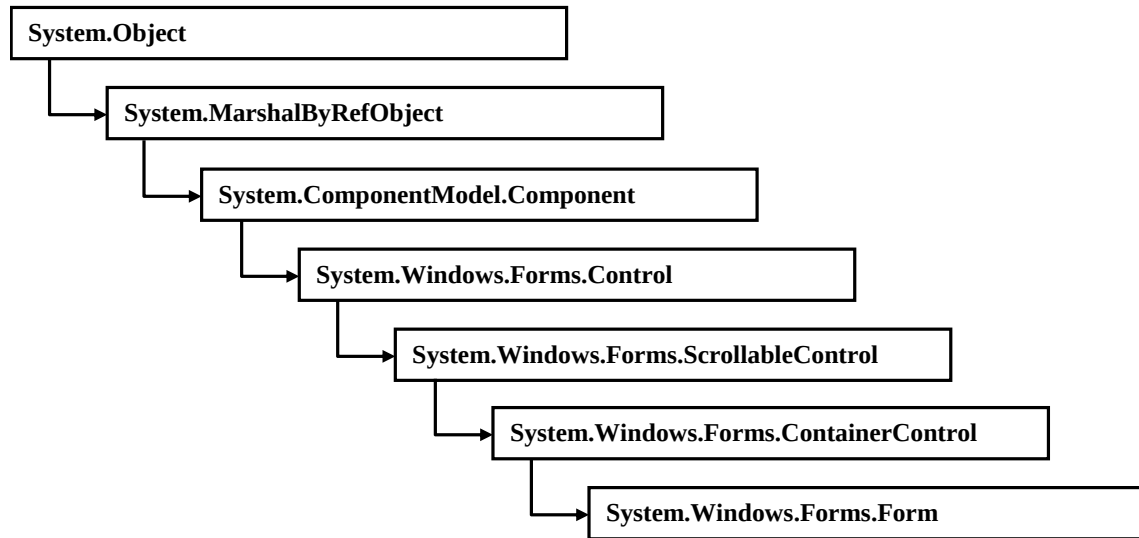
Aus dem sehr umfangreichen Namensraum **System.Windows.Forms** (mit mehreren hundert Klassen und sonstigen Typen), den man bei einer GUI-Anwendung am besten per **using**-Direktive importiert, werden Sie in Abschnitt 8 folgende Klassen kennen lernen:

- **Form**
Aus dieser Klasse stammen alle Formulare (z.B. für Anwendungs- oder Dialogfenster) ab. Im .NET – Umfeld werden die Begriffe *Fenster* und *Formular* synonym verwendet.
- **Application**
Diese Klasse bietet u.a. essentielle statische Methoden zum Betreiben eines Windows-Programms (z.B. **Run()**, **Exit()**).
- Klassen für Delegaten (z.B. **EventHandler**)
- Klassen für Steuerelemente (z.B. **Label**, **TextBox**, **Button**)

Durch die **using**-Direktive in der ersten Zeile des Beispielprogramms wird dafür gesorgt, dass der Compiler jeden Klassennamen, der nicht im projekteigenen Quellcode definiert ist, den Präfix **System.Windows.Forms** voranstellt und dann die referenzierten Assemblies nach dem vervollständigten Namen durchsucht. Im Beispielprogramm lohnt sich die **using**-Direktive eigentlich nicht, weil nur *ein* Typbezeichner aus dem importieren Namensraum auftaucht. Bei fast allen späteren WinForms-Programmen ist die Direktive jedoch empfehlenswert, so dass sie der Einheitlichkeit und Übung halber auch im Einstiegsbeispiel verwendet wird.

8.2.1 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.
- 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** abstammenden 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 Beispielprogramm mit der Anweisung

```
hf.Text = "WinForms-Einstieg";
```


die Titelzeilenbeschriftung des Fensters über die **Form**-Eigenschaft **Text** gewählt wird, sind zahlreiche weitere Eigenschaften veränderbar, z.B.:

- Die Startgröße eines Formulars kann über die Eigenschaften **Height** und **Width** geändert werden, z.B.:

```
hf.Height = 150; hf.Width = 300;
```
- Über die booleschen Eigenschaften **ControlBox**, **HelpBox**, **MaximizeBox** und **MinimizeBox** wählt man die Ausstattung der Titelzeile mit Standardschaltflächen, z.B.

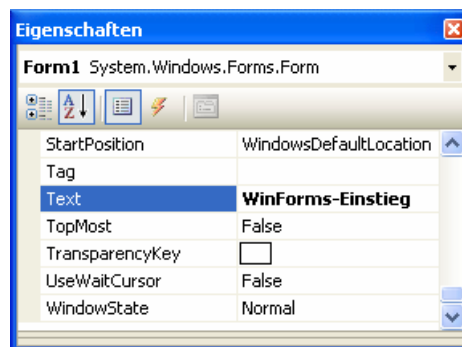
```
hf.MaximizeBox = false; hf.MinimizeBox = false;
```
- Indem man der Eigenschaft **FormBorderStyle** einen Wert der Enumeration **FormBorderStyle** aus dem Namensraum **System.Windows.Forms** zuordnet, kann man das Design und/oder die Funktionalität eines Formulars beeinflussen. Z.B. führt der Wert **FixedSingle** zu einem Fenster mit fixierter Größe:

```
hf.FormBorderStyle = FormBorderStyle.FixedSingle;
```
- Mit der Eigenschaft **BackColor** setzt man die Hintergrundfarbe des *Klientenbereichs* (Fensterfläche abzüglich Rahmen, Titelzeile, Menüleiste, Rollbalken und ggf. Symbolleiste), z.B.:

```
hf.BackColor = System.Drawing.Color.FromArgb(255, 255, 0);
```

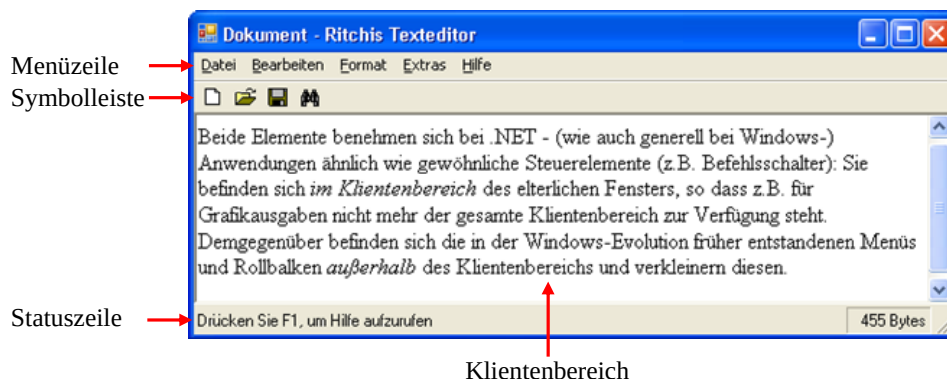
Den erforderlichen Wert vom Strukturtyp **Color** aus dem Namensraum **System.Drawing** liefert im Beispiel die statische **Color**-Methode **FromArgb()**, wobei die Parameter für die Farbkomponenten Rot, Grün und Blau stehen.

Beim Einsatz eines Formulargenerators 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.

In Kürze werden wir die Fenster unserer Programme mit Bedienelementen wie Menüleiste, Symbolleiste und Statuszeile ausstatten, z.B.:



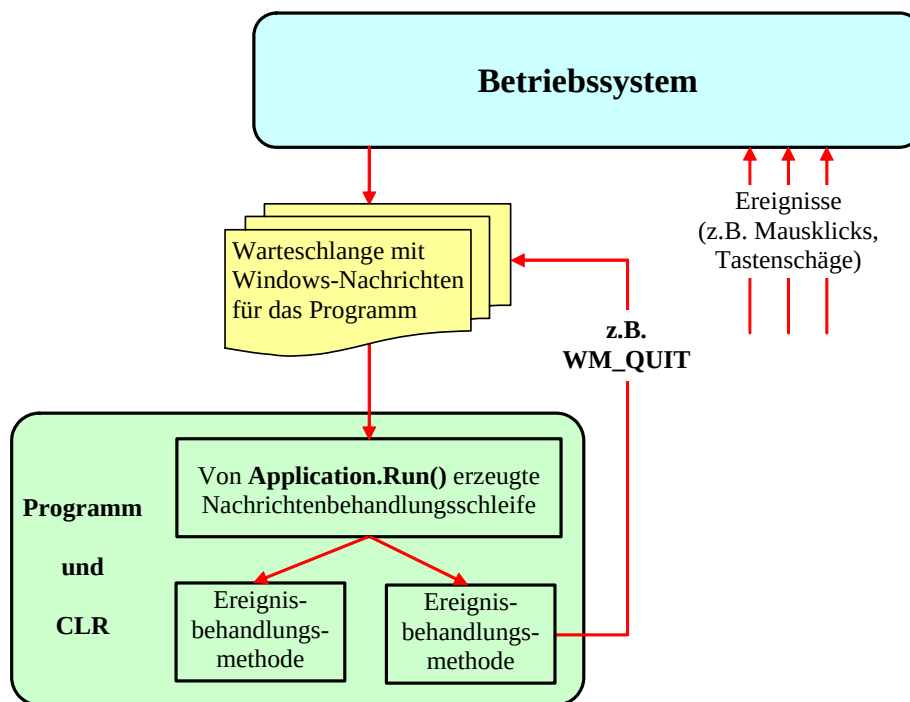
8.2.2 Windows-Nachrichten und die Klasse Application

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 .NET – GUI-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.

Zu manchen Nachrichten werden von Windows oder von der CLR ohne Zutun des Programmierers Behandlungsroutinen bereitgestellt. So kann unser Beispielprogramm z.B. auf die Standardschaltflächen in der Titelzeile (z.B. Maximieren, Minimieren, 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:



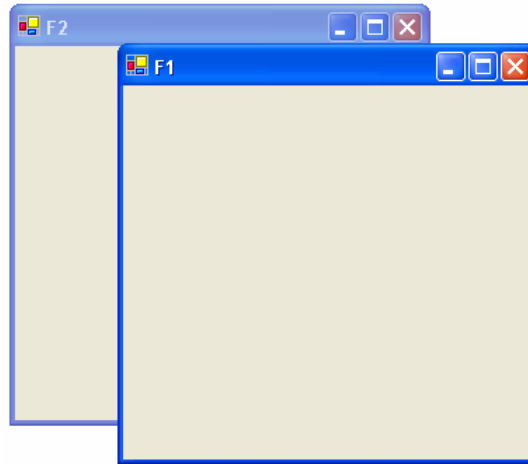
Wird die Nachricht **WM_QUIT** aus der Warteschlange gefischt, endet die Nachrichtenbehandlungsschleife, und der **Run()**-Aufruf kehrt zurück. Oft wird die Nachricht **WM_QUIT** von einer (automatisch eingerichteten) Ereignisroutine zum Hauptfenster abgeschickt, das so auf den Befehl des Benutzers zum 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 Beispielprogramms

```

using System.Windows.Forms;
class Prog : Form {
    static void Main() {
        Prog p1 = new Prog();
        Prog p2 = new Prog();
        p1.Text = "F1";
        p2.Text = "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.

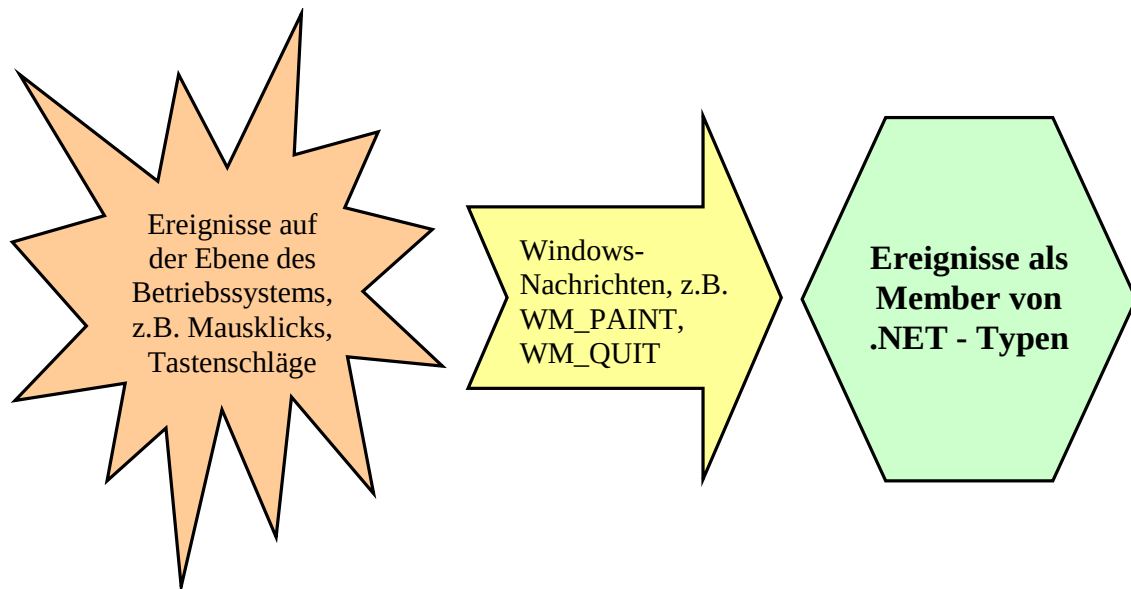


Fassen wir leicht vereinfachend zusammen, welche Bedeutung der Methodenaufruf **Application.Run()** in einer WinForms-Anwendung hat:

- Das **Hauptfenster** wird geöffnet.
- Das Programm wird um eine **Nachrichtenschleife**¹ 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).

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 wollen. Eine registrierte Methode wird dann 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):

¹ Genau genommen ist für jeden Thread (siehe unten), der ein Fenster präsentiert, eine Nachrichtenwarteschlange und dementsprechend eine Nachrichtenschleife erforderlich.



8.3 Delegaten und Ereignisse

8.3.1 Delegaten

In diesem Abschnitt lernen Sie die so genannten *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 Rückgabebetyp und Formalparameterliste) zeigen. Dazu enthalten Delegatenobjekte eine (ein- oder mehrelementige) Aufrufliste mit kompatiblen Methoden. Diese Liste kann enthalten:

- statische Methoden
- von konkreten Objekten auszuführende Instanzmethoden

Über ein Delegatenobjekt lassen sich mit *einer* Aktualparameterliste alle Methoden seiner Aufrufliste nacheinander starten, wobei die zuletzt ausgeführte Methode ggf. den Rückgabewert des Aufrufs liefert.¹

Vielleicht helfen die folgenden Begriffserläuterungen, Missverständnisse zu vermeiden:

Ein **Delegatentyp** besitzt:

- einen Namen, z.B. `DelTyp`
- eine Signatur, z.B. `delegate void DelTyp(double d, int i);`

Ein **Delegatenobjekt** besitzt:

- einen Typ, z.B. `DelTyp`
- eine aktuelle Aufrufliste mit kompatiblen Klassen- und Instanzmethoden

¹ Somit bietet C# mit den Delegaten eine objektorientierte Variante der Funktions-Pointer aus anderen Programmiersprachen (z.B. C++, Pascal, Modula).

Ein **Delegatenvariable** besitzt:

- einen Typ, z.B. `DelTyp`
- einen Wert, z.B. `null` oder die Adresse eines `DelTyp`-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.

8.3.1.1 Delegatentypen definieren

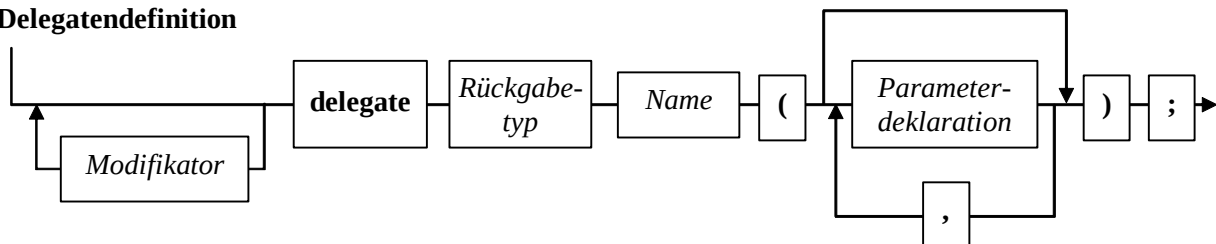
Mit der folgenden Anweisung wird über das Schlüsselwort **delegate** der Delegatentyp `HalloGate` definiert:

```
delegate void HalloGate(int w);
```

Über Objekte dieses Typs können Instanz- oder Klassenmethoden mit dem Rückgabebetyp **void** und einem einzigen Parameter vom Typ **int** aufgerufen werden.

Das Syntaxdiagramm zur Definition eines Delegatentyps:

Delegatendefinition



8.3.1.2 Delegatenobjekte erzeugen und aufrufen

In der folgenden Klasse `Sprecher` ist eine Instanzvariable namens `HalloVar` vom eben definierten Delegatentyp `HalloGate` vorhanden:

```
class Sprecher {
    int n;
    public HalloGate HalloVar;
    public Sprecher(int w) {
        n = w;
    }
    public void Sprich() {
        if (HalloVar != null)
            HalloVar(n);
    }
}
```

Weil ein `Sprecher`-Objekt in seiner `Sprich()`-Methode ein `HalloGate`-Delegatenobjekt aufruft, lässt sich sein Verhalten folgendermaßen beeinflussen:

- Man definiert eine Methode mit HalloGate-Signatur.
- Man erzeugt ein HalloGate-Delegatenobjekt, das auf diese Methode zeigt.
- Man stellt dem Sprecher-Objekt das Delegatenobjekt zur Verfügung (über seine HalloGate-Referenzvariable).

Wie in der Sprecher-Methode `Sprich()` demonstriert wird, sollte man vor einem Methodenaufruf per Delegatenreferenzvariable vorsichtshalber prüfen, ob tatsächlich ein Delegatenobjekt vorhanden ist.

In der folgenden Klasse `Prog` werden zwei statische Methoden definiert, welche die Signatur des Delegatentyps `HalloGate` erfüllen:

```
class Prog {
    static void Muster1(int w){
        Console.Write("H");
        for (int i = 1; i <= w; i++)
            Console.Write("a");
        Console.WriteLine("llo!");
    }
    static void Muster2(int w){
        Console.Write("Hallo");
        for (int i = 1; i <= w; i++)
            Console.Write("!");
        Console.WriteLine();
    }
    static void Main(){
        Sprecher h = new Sprecher(3);
        h.HalloVar = new HalloGate(Muster1);
        h.Sprich();
    }
}
```

In der **Main()**-Methode erzeugt der folgende Ausdruck ein `HalloGate`-Objekt, das auf die Methode `Prog.Muster1()` zeigt:

```
new HalloGate(Muster1);
```

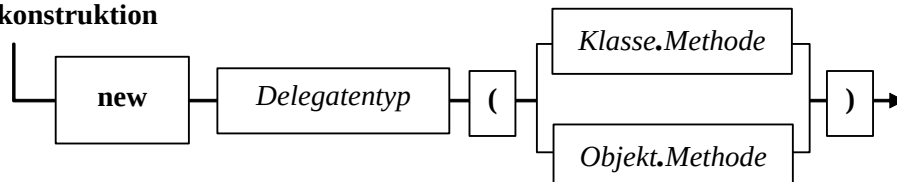
Die entstehende Objektreferenz landet in der `HalloVar`-Instanzvariablen eines `Sprecher`-Objektes, das beim anschließenden Aufruf seiner `Sprich()`-Methode folgendermaßen agiert:

Haaaallo!

Delegatenobjekte sind keinesfalls auf statische Methoden beschränkt. Ihre Aufruflisten können auch Instanzmethoden aufnehmen, wobei neben der Methode auch das ausführende Objekt gespeichert wird.

Dem implizit definierten Konstruktor einer Delegatenklasse übergibt man als einzigen Parameter den Namen einer statischen Methode (nötigenfalls mit vorangestelltem Klassennamen) oder den Namen einer Instanzmethode mit vorangestellter Objektreferenz:

Delegatenkonstruktion



Dabei wird den Methodennamen *keine* Parameterliste angehängt.

8.3.1.3 Delegatenobjekte kombinieren

Wir betrachten die folgende Variante der **Main()**-Methode aus dem letzten Beispielprogramm:

```
static void Main() {
    Sprecher h = new Sprecher(3);
    h.HalloVar = new HalloGate(Muster1);
    h.Sprich();
    // Hier entsteht ein neues(!) Delegatenobjekt mit verlängerter Aufrufliste:
    h.HalloVar += new HalloGate(Muster2);
    h.Sprich();
}
```

Hinter der Kommentarzeile wird per **new**-Operator ein weiteres Delegatenobjekt vom Typ `HalloGate` erstellt, dessen Aufrufliste die statische Methode `Prog.Muster2()` enthält:

```
h.HalloVar += new HalloGate(Muster2);
```

Dieses Objekt wird anschließend per „+“ - Operator mit dem von `h.HalloVar` referenzierten Objekt kombiniert, wobei ein weiteres `HalloGate`-Delegatenobjekt mit einer *zweielementigen* Aufrufliste entsteht, dessen Adresse schließlich in der Referenzvariablen `h.HalloVar` landet. Die beiden Delegatenobjekte mit einelementiger Aufrufliste werden zu obsoletem Müll (vgl. ECMA 2006, S. 365).

Beim Aufruf des neuen Delegatenobjektes werden nacheinander *zwei* Methoden ausgeführt:

```
Haaallo!
Hallo!!!
```

Über den „-“ - Operator kann man ein Delegatenobjekt mit *verkürzter* Aufrufliste erzeugen.

Neben den Aktualisierungsoperatoren `+=` und `-=` kann man auch den Additions- oder den Subtraktionsoperator verwenden, z.B.:

```
h.HalloVar = h.HalloVar + new HalloGate(Muster2);
```

8.3.2 Ereignisse

Möchte eine .NET – Klasse anderen Programmeinheiten die Möglichkeit geben, zu besonderen Gelegenheiten eine Botschaft zu erhalten, d.h. eine bestimmte Methode ausführen zu lassen, dann bietet sie ein *Ereignis* an. Darunter ist im .NET - Framework eine *Delegatenvariable* zu verstehen, 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 Instanzereignis der Klasse **System.Windows.Forms.Button** (siehe unten) heißt **Click**.

Innerhalb der veröffentlichenden Klasse stellt ein Ereignis eine private Referenzvariable mit Delegatentyp dar, die bei passender Gelegenheit aufgerufen wird, sofern sie auf ein Delegatenobjekt zeigt (siehe Abschnitt 8.3.2.2). Obwohl Ereignisse in der Regel als **public** deklariert werden, liegt stets eine *private* Delegatenreferenzvariable vor. Der Compiler ergänzt jedoch *öffentliche* Zugriffsmethoden zum Erweitern und Verkürzen der Aufrufliste durch fremde Klassen und überschreibt damit die bequemen Aktualisierungsoperatoren:

- `+=` nimmt eine Behandlungsmethode in die Aufrufliste des zum Ereignis gehörigen Delegatenobjekts auf
- `-=` entfernt eine Behandlungsmethode aus der Aufrufliste des zum Ereignis gehörigen Delegatenobjekts

Die in Abschnitt 8.2.2 vorgestellte Klasse **Application** bietet z.B. mit dem Ereignis (bzw. mit der Delegatenvariablen) **ApplicationExit** anderen Klassen die Möglichkeit, sich über das bevorstehende Programmende informieren zu lassen. In der SDK-Dokumentation sind die (allesamt statischen) Ereignisse der Klasse **Application** beschrieben:



8.3.2.1 Behandlungsmethoden registrieren

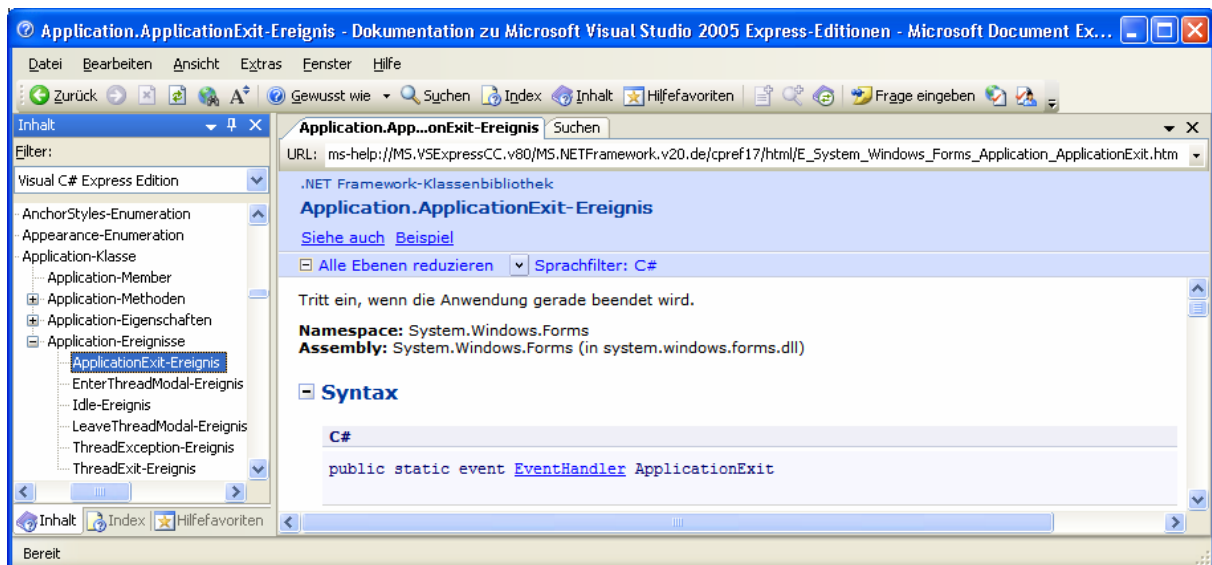
Um auf ein Ereignis einer .NET - Klasse reagieren zu können, ...

- implementiert man eine kompatible Methode
Die Signatur der Methode muss zum Delegatentyp des Ereignisses kompatibel sein.
- erzeugt per **new**-Operator ein Delegatenobjekt, das auf diese Methode zeigt
- weist der Ereignisvariablen dieses Delegatenobjekt per „+=“ - Operator zu.
In Abschnitt 8.3.1.3 wurde im Detail beschrieben, wie bei einer solchen Zuweisung die Aufrufliste des zum Ereignis gehörigen Delegatenobjekts erweitert wird.

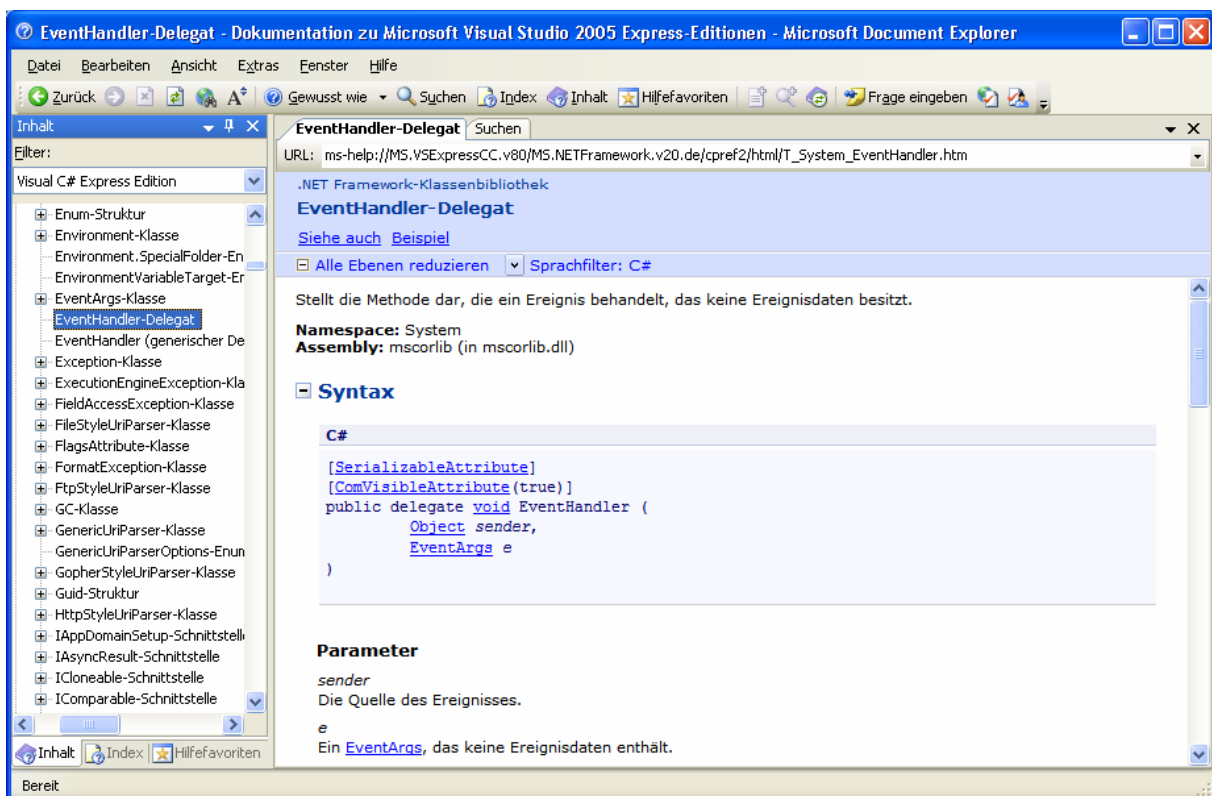
In der folgenden Variante des GUI-Einstiegsbeispiels aus Abschnitt 8.2 wird die Methode `ApplicationOnExit()` implementiert und beim Ereignis **ApplicationExit** der Klasse **Application** registriert, um auf das bevorstehende Programmende reagieren zu können:

```
using System;
using System.Windows.Forms;
class ApplicationExitDemo : Form {
    void ApplicationOnExit(object sender, EventArgs e) {
        MessageBox.Show("Vielen Dank für den Einsatz dieser Software!",
            "WinForms-Einstieg");
    }
    static void Main() {
        ApplicationExitDemo hf = new ApplicationExitDemo();
        hf.Text = "ApplicationExit-Ereignisbehandlung";
        Application.ApplicationExit += new EventHandler(hf.ApplicationOnExit);
        Application.Run(hf);
    }
}
```

Zu welchem Delegatentyp eine Ereignisbehandlungsmethode kompatibel sein muss, erfährt man z.B. in der SDK-Dokumentation. Beim Ereignis **ApplicationExit** handelt es sich um den Typ **EventHandler** aus dem Namensraum **System**:



Nach einem Mausklick auf den Deleгатentyp **EventHandler** ist bekannt, welchen Rückgabotyp und welche Parameter eine kompatible Methode benötigt, z.B.:



Die **EventHandler**-Signatur verlangt von kompatiblen Behandlungsmethoden zwei Parameter, die allerdings beim Ereignis **Application.ApplicationExit** irrelevant sind:

- 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 Instanzvariablen 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-Programmierer haben auch beim Ereignis **Application.Exit**, wo weder eine Absenderangabe noch eine Ereignisbeschreibung erforderlich ist, den Delegatentyp **EventHandler** gewählt, der einen Parameter namens **sender** vom Typ **Object** sowie einen Parameter namens **e** vom Typ **EventArgs** besitzt. 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:

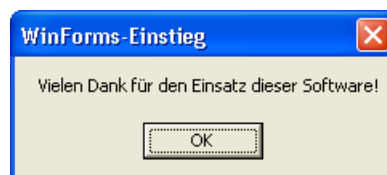
```
Application.ApplicationExit += new EventHandler(hf.ApplicationOnExit);
```

Weil es sich um eine Instanzmethode handelt, ist auch anzugeben, welches Objekt die Methode ausführen soll (im Beispiel: `hf`).

Das neue **EventHandler**-Delegatenobjekt wird per „+=“ – Operator beim Ereignis registriert, wobei ggf. eine vorhandene Aufrufliste verlängert wird (vgl. Abschnitt 8.3.1.3).

Über den „-=“ – Operator lässt sich eine Behandlungsmethode aus der Aufrufliste zu einem Ereignis entfernen.

Damit eine öffentlich zugängliche Delegatenvariable zu Recht als *Ereignis* bezeichnet wird, muss sie von der veröffentlichenden Klasse natürlich auch ausgeführt werden. Im Beispiel kommt die Methode `ApplicationOnExit()` z.B. dann zum Einsatz, wenn der Benutzer auf das Schließkreuz in der Titelzeile des Anwendungsfensters klickt:



8.3.2.2 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 10 ganze Zufallszahlen aus dem Intervall von 0 bis 9. Bei jedem Auftreten der Zahl 7 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 8.4 über Befehlsschalter und sonstige Steuerelemente in Kauf genommen:

```

public class EventProd : Button {
    public event SimpleDelegate Seven;
    Random zzg = new Random();
    override protected void OnClick(EventArgs 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);
            }
        }
    }
}

```

Warum die überschreibende Methode **OnClick()** bei einem Mausklick auf den Befehlsschalter aufgerufen wird, erfahren Sie später.

Im Unterschied zu einer gewöhnlichen Variablendeklaration ist bei Ereignissen das Schlüsselwort **event** anzugeben:

```
public event SimpleDelegate Seven;
```

Als Datentyp kommt nur ein Delegat in Frage, der im Beispiel folgendermaßen definiert wird:

```
public delegate void SimpleDelegate(object sender, MyEventArgs e);
```

Zur Ereignisbeschreibung dient hier die Klasse **MyEventArgs**, die von **EventArgs** abgeleitet wird:

```

public class MyEventArgs : EventArgs {
    public int Pos;
}

```

Man darf ein Ereignis nur dann auslösen, wenn tatsächlich ein Delegatenobjekt vorhanden ist:

```

if (zzg.Next(10) == 7 && Seven != null) {
    Seven(null, mea);
}

```

Bei einer Multithreaded-Anwendung (vgl. Abschnitt 12) sind weitere Absicherungen gegen eine **NullReferenceException** erforderlich (siehe z.B. Richter 2006, S. 251).

Die folgende WinForms-Anwendung enthält einen Schalter aus der **EventProd**-Klasse auf Ihrem Formular:

```

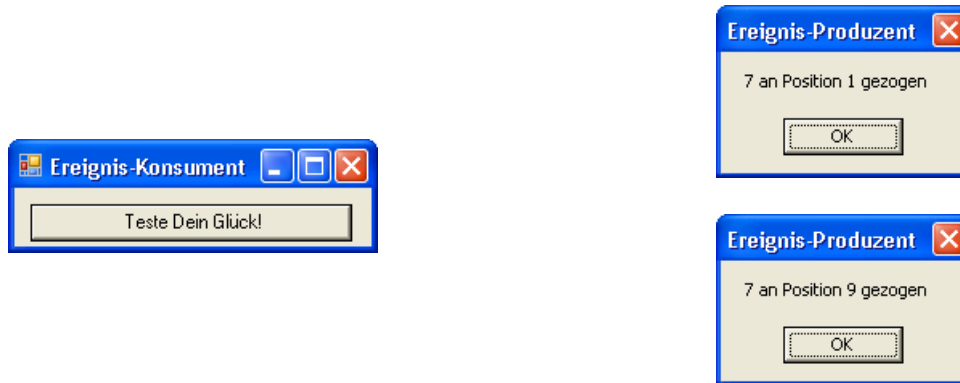
class EventConsumer : Form {
    public EventConsumer() {
        Height = 70; Width = 230;
        Text = "Ereignis-Konsument";
        EventProd ep = new EventProd();
        ep.Text = "Teste Dein Glück!";
        ep.Top = 10;
        ep.Left = 10;
        ep.Width = 200;
        Controls.Add(ep);
        ep.Seven += new SimpleDelegate(this.EventProdOnSeven);
    }
    protected void EventProdOnSeven(object sender, MyEventArgs e) {
        MessageBox.Show("7 an Position " + e.Pos + " gezogen",
            "Ereignis-Produzent");
    }
    static void Main() {
        Application.Run(new EventConsumer());
    }
}

```

Um auf das Ereignis Seven reagieren zu können, implementiert der die Klasse EventConsumer eine kompatible Methode und registriert sie 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 EventProdOnSieben() beim Aufruf übergeben wird, z.B.:



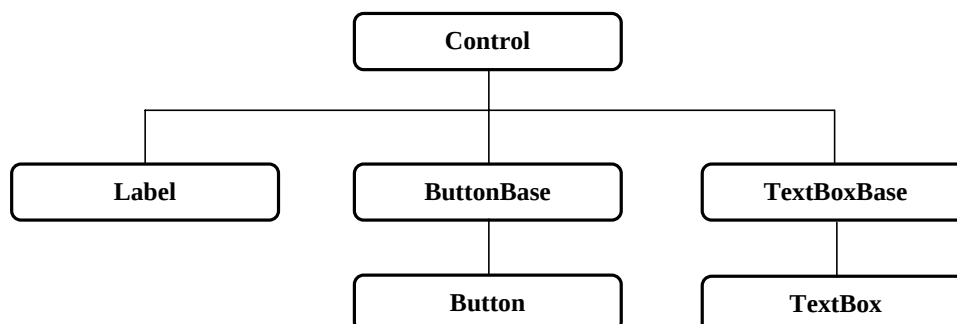
8.4 Steuerelemente

Die FCL bietet zahlreiche Klassen zur Realisation von Steuerelementen (Schaltflächen, Textfeldern, Kontrollkästchen, Listen etc.), 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:

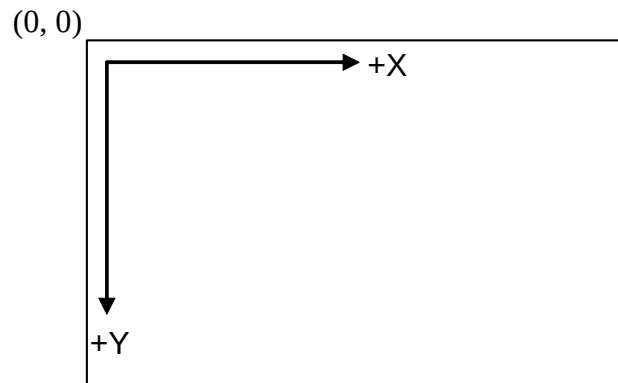
- Ihre Objekte können als (Kind-)fenster auf dem Bildschirm auftreten und dabei selbständig mit dem Benutzer interagieren. Wenn wir z.B. ein Kontrollkästchen in ein Formular einbauen, erscheint bzw. verschwindet bei einem Mausklick des Benutzers die Markierung des Kontrollkästchens, ohne dass wir uns um diese Anpassung der Optik kümmern müssten.
- Die Steuerelemente kommunizieren über Ereignisse (im Sinn von Abschnitt 8.3.2) mit anderen Klassen. Will man z.B. über den Markierungswechsel bei einem Kontrollkästchen informiert werden, registriert man eine Behandlungsmethode bei seinem **CheckedChanged**-Ereignis.

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.:

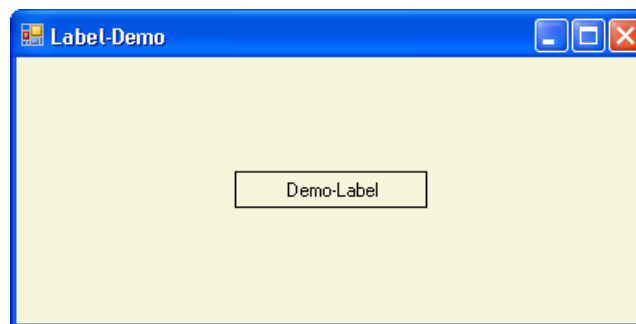


Bei den häufig erforderlichen Positions- und Größenangaben in Bildschirmpunkten (Pixeln) liegt das folgende Koordinatensystem mit dem Ursprung (Nullpunkt) in der linken oberen Ecke des Fenster-Klientenbereichs zugrunde:



8.4.1 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 Steuerelemente erzeugt und in ein Formular aufnimmt.



Dazu wird die aus **Form** abgeleitete Klasse `LabelDemo`

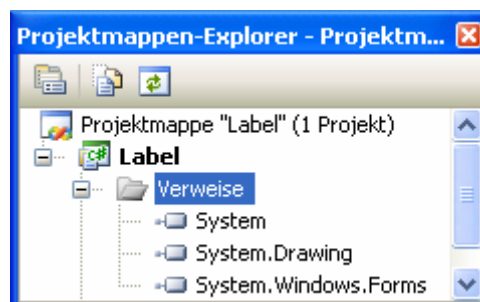
```
class LabelDemo : Form {  
    . . .  
}
```

sukzessiv vorgestellt.

Wie bei den meisten WinForms-Anwendungen empfiehlt sich der Import der folgenden Namensräume:

```
using System;  
using System.Windows.Forms;  
using System.Drawing;
```

Ferner benötigen wir Verweise auf die folgenden Assemblies:



8.4.1.1 Steuerelemente erzeugen und konfigurieren

Im Konstruktor der Formalkasse

```
public LabelDemo() {
    Height = 200; Width = 400;
    BackColor = Color.LightGray;
    Text = "Label-Demo";

    Label info = new Label();
    info.Text = "Demo-Label";
    info.Width = 120;
    info.Height = Font.Height + 15;
    info.Left = (ClientSize.Width-info.Width)/2;
    info.Top = (ClientSize.Height-info.Height)/2;
    info.TextAlign = ContentAlignment.MiddleCenter;
    info.BorderStyle = BorderStyle.FixedSingle;
    . . .
}
```

wird ein **Label**-Objekt erzeugt

```
Label info = new Label();
```

und durch Modifikation von **Label**-Eigenschaften gestaltet:

- 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 **Label**-Beschriftung resultiert aus der aktuellen Größe des Steuerelementes. Man kann sie per **Size**-Eigenschaft (nimmt Werte vom Strukturtyp **System.Drawing.Size** an) oder (wie im Beispiel) über die Eigenschaften **Width** und **Height** verändern. Alle genannten Eigenschaften sind schon in der Basisklasse **Control** definiert.
- Bei der Höhe eines Labels bietet sich oft die Orientierung an der Schrifthöhe an, wobei die Anweisung

```
info.Height = Font.Height + 15;
```

einige erläuterungsbedürftige Bezeichnungen enthält:

- Auf der linken Seite der Zuweisung wird mit **Height** eine Eigenschaft des **Label**-Steuerelementes angesprochen.
- Auf der rechten Seite handelt es sich um die **Height**-Eigenschaft einer Schriftart, d.h. um den vertikalen Abstand zwischen den Basislinien von zwei aufeinander folgenden Textzeilen.
- Bei dieser Schriftart handelt es sich um dasjenige Objekt der Klasse **Font** (im Namensraum **System.Drawing**), auf das die Eigenschaft **Font** des Formular-Objekts zeigt.
- Der Compiler identifiziert den Bezeichner hinter dem Zuweisungsoperator als Eigenschaftsnamen, weil die Interpretation als Klassenname sofort zu einem Fehler führt: In der Klasse **Font** gibt es keine statische Eigenschaft namens **Height**.

In der folgenden Schreibweise wird die **this**-Referenz auf das handelnde Formular-Objekt verwendet, um die Lesbarkeit zu verbessern:

```
info.Height = this.Font.Height + 15;
```

Mit der **Label**-Eigenschaft **AutoSize** (Voreinstellung: **false**) kann man für eine automatische Größenanpassung an den Inhalt sorgen, wobei aber eventuell andere Steuerelemente überdeckt werden:

```
info.AutoSize = true;
```

- Steuerelemente übernehmen per Voreinstellung die Schriftart des umgebenden Containers (siehe unten). 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.
- Mit den **Control**-Eigenschaften **Left** bzw. **Top** spricht man die Position der linken bzw. oberen Grenze eines Steuerelements relativ zum umgebenden Container an. Im Beispiel werden diese Eigenschaften so aus der Größe des Formular-Klientenbereichs abgeleitet, dass ein horizontal und vertikal zentriertes Label resultiert.
- 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 spezieller Rahmen gewünscht, steht die **Label**-Eigenschaft **BorderStyle** bereit, wobei die Werte des Aufzählungstyps **System.Windows.Forms.BorderStyle** zu verwenden sind.

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 {
    public LabelDemo() {
        Label info = new Label();
        . . .
    }
    . . .
}
```

In der alternativen Situation wäre z.B. die folgende Konstruktion angemessen:

```
class LabelDemo : Form {
    Label info;
    public LabelDemo() {
        info = new Label();
        . . .
    }
    . . .
}
```

Eine überflüssige Instanzvariable erhöht das Risiko eines Programmierfehlers (z.B. durch eine ungeplante Steuerelement-Modifikation per Tippfehler).

8.4.1.2 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.:


```
public 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 auf der elterlichen Fensterfläche, ihre Positionseigenschaften beziehen sich auf diese Fläche, und darüber hinaus ragende Steuerelementteile werden abgeschnitten.

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 8 vorgestellten GUI-Anwendungen um das außerordentlich wichtige und häufige **Paint**-Ereignis (definiert in der Klasse **Control**) nicht selbst kümmern.

Bei komplexen Fenstern kommen oft hierarchisch geschachtelte Container zum Einsatz. Aus Container zur Aufnahme einer Gruppe von Bedienelementen recht beliebt ist die Klasse **Panel**.

8.4.1.3 Ereignisbehandlungsmethoden

Nach Bedarf werden in der Formalklasse Behandlungsmethoden definiert und bei Ereignissen des Formulars oder der Steuerelemente registriert. Im Beispiel wird die Methode `InfoOnClick()` definiert und beim **Click**-Ereignis des **Label**-Objekts registriert:

```
public 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 erzeugt
- dem Ereignis (also einer Delegetenvariablen) die Adresse des neuen Delegetenobjekts zugewiesen

Ist bereits ein Delegetenobjekt für das Ereignis vorhanden, entsteht ein neues Objekt mit entsprechend verlängerter Aufrufliste (vgl. Abschnitt 8.3).

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 {
    public LabelDemo() {
        Height = 200; Width = 400;
        BackColor = Color.LightGray;
        Text = "Label-Demo";
        Label info = new Label();
        info.Text = "Demo-Label";
        info.Width = 120;
        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;
    }

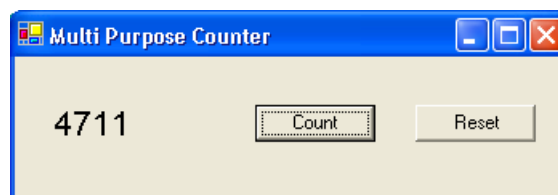
    static void Main() {
        Application.Run(new LabelDemo());
    }
}
```

8.4.2 Befehlsschalter

Befehlsschalter werden im .NET-Framework durch die Klasse **Button** realisiert.

8.4.2.1 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;

    public ButtonDemo() {
        Height = 120; Width = 350;
        Text = "Multi Purpose Counter";
        FormBorderStyle = FormBorderStyle.FixedSingle;

        stand = new Label();
        stand.Text = anzahl.ToString();
        stand.Font = new Font("Arial", 16);
        stand.Left = 20; stand.Top = 30;
        stand.Width = 130;
        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());
    }
}

```

Im Beispiel wird die **Height**-Eigenschaft der Steuerelemente 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

Die für das **Click**-Ereignis *beider* Schalter registrierte Ereignisbehandlungsmethode `ButtonOnClick()` wertet den Parameter **sender** aus und orientiert ihr Verhalten an der Ereignisquelle.

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;
```

8.4.2.2 Eingabefokus und Standardschaltfläche

Obwohl beide Schalter des Multi-Counters im Fensterkonstruktor analog konfiguriert werden, sehen sie nach dem Programmstart etwas unterschiedlich aus:



Die zuerst auf das Formular gesetzte Schaltfläche (ansprechbar über die Instanzvariable `count`) besitzt in dieser Situation den **Eingabefokus** und ist an einem dekorierten Rahmen zu erkennen:

- durchgezogene schwarze Umrisslinie
- innen liegende gepunktete Linie
Bei Windows XP erscheint die gepunktete Linie allerdings eventuell erst nach einem Fokuswechsel per Tabulatortaste.¹

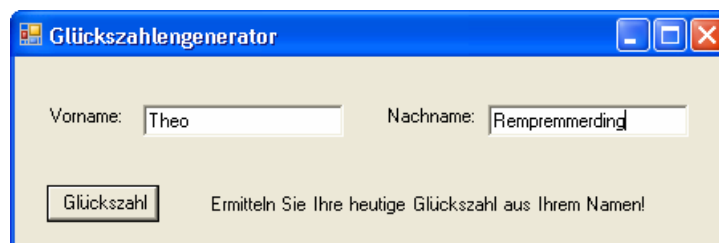
Folglich werden Tastaturereignisse (abgesehen von Eingaben zur Fokussteuerung, z.B. per Tabulatortaste) an den `count`-Schalter geleitet. Auf die Enter-Taste reagiert er wie auf einen linken Mausklick, was die Bedienung des Programms erleichtert.

Dank der Fokusverwaltung durch das Formular (vgl. Abschnitt 8) können Benutzer per Tabulator- oder Pfeiltaste den Eingabefokus zwischen eingabe-tauglichen 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 eine **Standardschaltfläche** festgelegt werden. Sofern kein anderer Schalter den Eingabefokus besitzt, wird per Enter-Taste das **Click**-Ereignis der Standardschaltfläche ausgelöst. In dieser Situation ist der privilegierte Schalter an einer durchgezogenen schwarzen Umrisslinie zu erkennen. Im Beispielprogramm zum **TextBox**-Steuerelement (siehe Abschnitt 8.4.3) wird über die Formular-Eigenschaft **AcceptButton** eine Standardschaltfläche ernannt:

```
AcceptButton = cbGluecksZahl;
```

Nun kann der Schalter zum Anfordern der persönlichen Glückszahl auch dann per Enter-Taste ausgelöst werden, wenn eines der Textfelder den Eingabefokus besitzt:



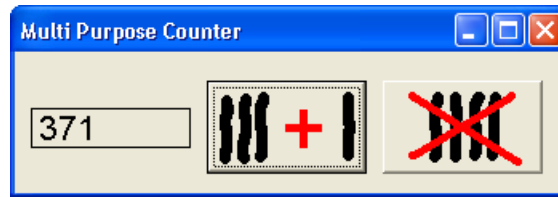
Im Beispiel bietet es sich sogar an, den Schalter über den Wert **false** seiner **TabStop**-Eigenschaft aus der Liste der per Tastatur fokussierbaren Steuerelemente zu entfernen, damit die Tabulatortaste nur noch zwischen den beiden Textfeldern wechselt:

```
cbGluecksZahl.TabStop = false;
```

¹ Deaktiviert man im Windows XP - Systemsteuerungsapplet **Anzeige** auf der Registerkarte **Darstellung** nach einem Klick auf den Schalter **Effekte** das Kontrollkästchen **Unterstrichene Buchstaben für Tastaturnavigation ausblenden**, dann erscheint die gepunktete Fokuslinie auch in Windows XP auf jeden Fall.

8.4.2.3 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.:



Nach der kreativen Arbeit in einem Grafikprogramm (die Windows-Zugabe MS-Paint genügt) sind im Formalkonstruktor 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;
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\bin\Debug**). Später werden Sie eine Möglichkeit kennen lernen, die Bitmap-Datei als so genannte *Ressource* in das Assembly aufzunehmen.

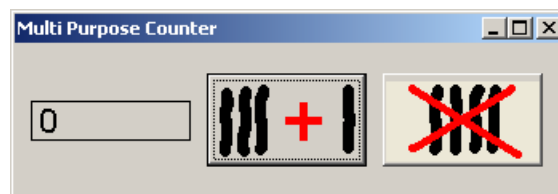
Wählen Sie die Größe eines zu bemalenden Schalters so, dass um das Bild herum mindestens 8 Pixel Rand verbleiben, damit dort ggf. der gepunktete Fokus-Rahmen angezeigt werden kann, z.B.:

```
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 (z.B. klassisch vs. XP) optisch mit den restlichen Fensterbestandteilen harmonieren, z.B.:

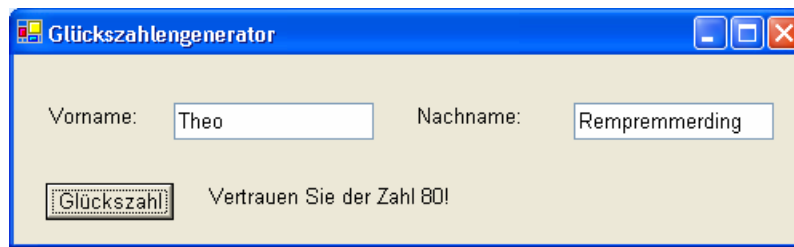
```
Bitmap bmp = new Bitmap("count.bmp");
bmp.MakeTransparent bmp.GetPixel(1, 1));
count.Image = bmp;
```

Wenn im Beispielprogramm ausschließlich der *linke* Schalter eine transparente Hintergrundfarbe erhält, und anschließend das klassische Windowsdesign eingestellt wird, resultiert folgendes Erscheinungsbild:



8.4.3 Textfelder

Texteingaben der Benutzer erfasst man im .NET-Framework mit Steuerelementen der Klasse **TextBox**. Auf dem bereits in Abschnitt 8.4.2.2 präsentierten Formular eines Programms zur Berechnung der persönlichen Glückszahl in Abhängigkeit vom Vor- und Nachnamen sind zwei Exemplare davon zu sehen:



Für Steuerelemente des Programms, die nicht nur im Formalkonstruktor, sondern auch in 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 Formalklasse:

```
public TextBoxDemo() {
    Height = 150; Width = 450;
    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 Eigenschaft **Text** kann man auf den Inhalt eines Texteingabefeldes zugreifen:

```

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) {
        for (int i = 0; i < vn.Length; i++)
            seed += (int) vn[i];
        for (int i = 0; i < nn.Length; i++)
            seed += (int) nn[i];
        Random zzg = new Random(DateTime.Today.Day + seed);
        lbInfo.Text="Vertrauen Sie heute der Zahl "+
            (zzg.Next(100)+1).ToString()+"!";
        bValToRemove = true;
    }
}

```

Über das bereits in der Klasse **Control** definierte 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;
    }
}

```

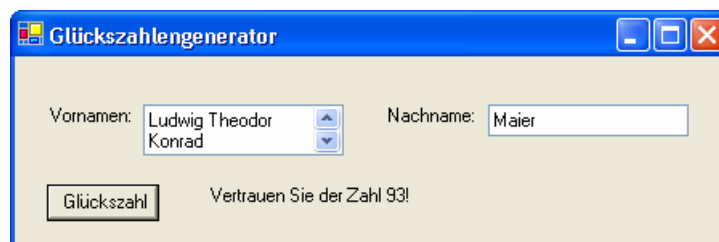
TextBox-Steuerelemente besitzen etliche Kompetenzen, die den Benutzer erfreuen und dabei den Programmierer nicht belasten, z.B.:

- 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**

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

...\BspUeb\WinForms\TextBox

Um ein *mehrzeiliges Textfeld* zu erhalten, setzt man die Eigenschaft **Multiline** auf den Wert **true**, sorgt für genügend Platz und schaltet eventuell noch vertikale Bildlaufleisten ein, z.B.:



Für diese Variante des Glückszahlenprogramms sind nur drei zusätzliche Zeilen erforderlich (im Fensterkonstruktor):

```

tbVorname.Multiline = true;
tbVorname.Height = 32;
tbVorname.ScrollBars = ScrollBars.Vertical;

```

Wir werden später mit Hilfe des leistungsfähigeren Steuerelements **RichTextBox** einen kompletten Texteditor erstellen.

8.4.4 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.

8.4.4.1 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 *labRed*, *labGreen* und *labBlue* mit den Hintergrundfarben Rot, Grün und Blau nacheinander in ein Formular aufgenommen:

```
using System;
using System.Drawing;
using System.Windows.Forms;

public class Stapelung : Form {
    public Stapelung() {
        Text = "Stapelung";
        ClientSize = new Size(300,150);

        Label labRed = new Label();
        labRed.Size = new Size(250,75);
        labRed.Left = 0;
        labRed.BackColor = Color.Red;
        labRed.Parent = this;
        labRed.Click += new EventHandler(PanelOnClick);

        Label labGreen = new Label();
        labGreen.Size = new Size(200,100);
        labGreen.Left = 100;
        labGreen.BackColor = Color.Green;
        labGreen.Parent = this;
        labGreen.Click += new EventHandler(PanelOnClick);

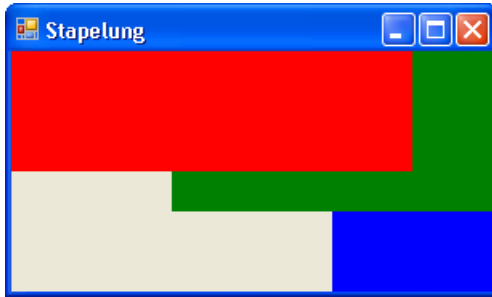
        Label labBlue = new Label();
        labBlue.Size = new Size(100,150);
        labBlue.Left = 200;
        labBlue.BackColor = Color.Blue;
        labBlue.Parent = this;
        labBlue.Click += new EventHandler(PanelOnClick);
    }

    void PanelOnClick(object sender, EventArgs e) {
        (sender as Control).BringToFront();
    }

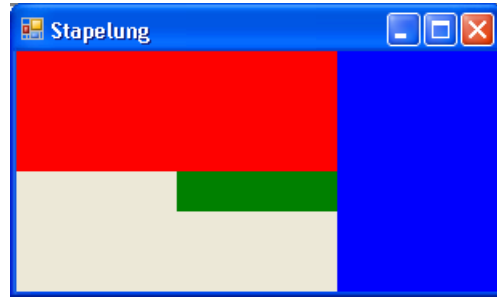
    public static void Main() {
        Application.Run(new Stapelung());
    }
}
```

Im gemeinsamen **Click**-Ereignishandler der **Panel**-Steuerelemente gelangt das getroffene Objekt durch Aufruf der Methode **BringToFront()** in den Vordergrund und erhält den neuen **Controls**-Index 0:

Nach dem Start



Nach einem Mausklick auf das blaue Panel



8.4.4.2 Verankern

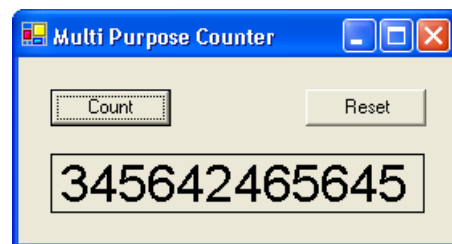
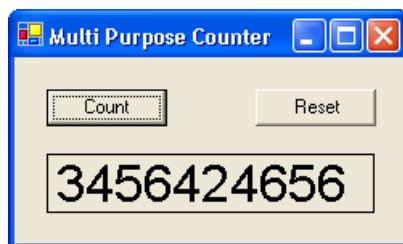
Ist ein Steuerelement an einer Formularseite *verankert*, behält es bei einer Veränderung der Fenstergröße nach Möglichkeit seinen Abstand zu dieser Seite konstant. Gesteuert wird dieses Verhalten durch die **Anchor**-Eigenschaft, der eine beliebige Oder-Verknüpfung von Werten der **AnchorStyles**-Enumeration zugewiesen werden kann:

Name	Wert
None	0
Top	1
Bottom	2
Left	4
Right	8

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`

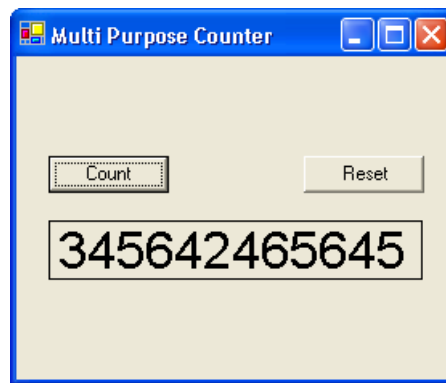
besitzt. Bei der folgenden Variante des Mehrzweck-Zählprogramms aus Abschnitt 8.4.2 mit neu angeordneten Steuerelementen ist das Label mit dem Zählerstand links und rechts verankert, so dass es mit zunehmender Formularbreite mehr Platz bietet:



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 Steuerelemente weder oben noch unten verankert sind, bleiben sie bei einer Änderung der Fensterhöhe vertikal zentriert:



Nach den beschriebenen Modifikationen sieht der Fensterkonstruktor so aus:

```
public ButtonDemo() {
    Height = 150; Width = 250;
    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 = Width - 46;
    stand.Height = stand.Font.Height;
    stand.BorderStyle = BorderStyle.FixedSingle;
    stand.Anchor = AnchorStyles.Left | AnchorStyles.Right;
    Controls.Add(stand);

    count = new Button();
    count.Text = "Count";
    count.Left = 20; count.Top = 20;
    count.Anchor = AnchorStyles.Left;
    Controls.Add(count);

    reset = new Button();
    reset.Text = "Reset";
    reset.Left = 150; reset.Top = 20;
    reset.Anchor = AnchorStyles.Right;
    Controls.Add(reset);

    EventHandler eh = new EventHandler(ButtonOnClick);
    count.Click += eh;
    reset.Click += eh;
}
```

Mit der folgenden Anweisung wird verhindert, 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:

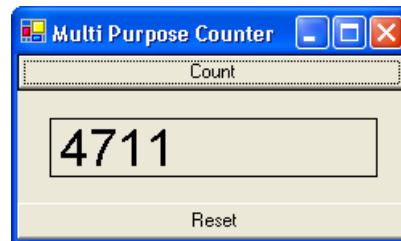
```
MinimumSize = new Size(Width, Height);
```

8.4.4.3 Andocken

Oft ist es erwünscht, dass ein Steuerelement nahtlos an einer Formulareseite andockt und sich bei jeder Fenstergröße über die komplette Seitenlänge erstreckt. Dieses Verhalten wäre durch Verankern an drei Seiten zu erzwingen (siehe Abschnitt 8.4.4.2), 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	Freie Fläche des Containers belegen

Bei folgender Variante des Zählprogramms sind die beiden Schalter oben bzw. unten andockt:



Die erforderlichen Zeilen im Formelarkonstruktor:

```
count.Dock = DockStyle.Top;
reset.Dock = DockStyle.Bottom;
```

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 erste und damit bescheidenste Steuerelement diesen **DockStyle** erhalten.

Im folgenden Fensterkonstruktor werden die drei **Label**-Steuerelemente labRed, labGreen und labBlue nacheinander links andockt, wobei labRed über die **Dock**-Eigenschaftsausprägung **DockStyle.Fill** auch an den drei restlichen Seiten andockt:

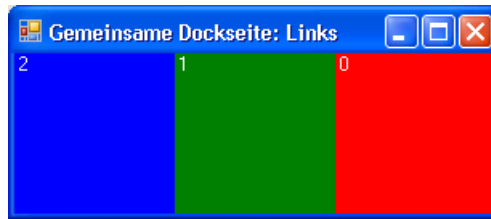
```
public GemeinsameDockseite() {
    Text = "Gemeinsame Dockseite: Links";
    ClientSize = new Size(300, 100);
    ResizeRedraw = true;
    ForeColor = Color.White;

    Label labRed = new Label();
    labRed.Size = new Size(100, 100);
    labRed.BackColor = Color.Red;
    labRed.Text = "0";
    labRed.Parent = this;
    labRed.Dock = DockStyle.Fill;

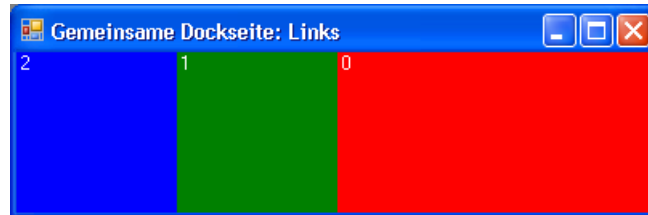
    Label labGreen = new Label();
    labGreen.Size = new Size(100, 100);
    labGreen.BackColor = Color.Green;
    labGreen.Text = "1";
    labGreen.Parent = this;
    labGreen.Dock = DockStyle.Left;

    Label labBlue = new Label();
    labBlue.Size = new Size(100, 100);
    labBlue.BackColor = Color.Blue;
    labBlue.Parent = this;
    labBlue.Text = "2";
    labBlue.Dock = DockStyle.Left;
}
```

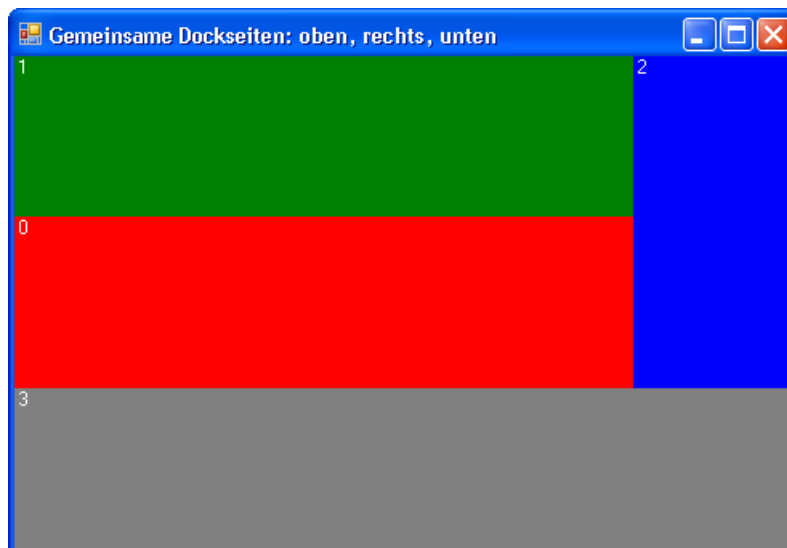
Es zeigt sich, dass die älteren Steuerelemente für Neulinge Platz machen. Das zuerst aufgenommene Steuerelement `labRed` (siehe **Controls**-Indexnummern links oben, vergeben in der Aufnahme-reihenfolge) ist am weitesten vom linken Fensterrand entfernt:



Bei einer Verbreiterung des Fensters passt sich `labRed` wegen seiner **Dock**-Eigenschaftsausprägung **DockStyle.Fill** an, während die anderen **Label**-Objekte nicht reagieren:



Im folgenden Beispiel überlässt das dienstälteste und auf allseitiges Andocken eingestellte **Label**-Steuerelement (Indexnummer 0, **DockStyle.Fill**) gleich an drei Seiten (**Top**, **Bottom**, **Right**) jüngeren **Dock**-Konkurrenten den Fensterrand:



8.4.4.4 *TableLayoutPanel*

Seit der Version 2.0 bietet das .NET - Framework mit dem **TableLayoutPanel** einen Container mit Matrix-Struktur, der in jeder Zelle ein Steuerelement aufnehmen kann. An Stelle gewöhnlicher Steuerelemente können prinzipiell auch weitere **TableLayoutPanel**-Objekte in die Zellen gesteckt werden, wobei Microsoft jedoch aus Performanzgründen von dieser Praxis abrät.

Für die Zeilen und Spalten sind absolute oder prozentuale Größenangaben erlaubt, so dass bei einer Größenänderung des äußeren Containers der verfügbare Platz möglichst sinnvoll neu aufgeteilt wird.

Der Bequemlichkeit halber verwenden wir zur Demonstration erneut das in Abschnitt 8.4.2 vorgestellten Mehrzweck-Zählprogramm, obwohl für sein einfaches Formular eigentlich kein **TableLayoutPanel** erforderlich ist. In den folgenden Zeilen 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 die 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 % für den Zählerstand und jeweils 25 % für die beiden Schalter.
- Das **TableLayoutPanel**-Steuerelement wird auf das Formular gesetzt und darf den gesamten Raum einnehmen (**DockStyle.Fill**).

Das **Label** und die beiden **Button**-Objekte dürfen sich in den zugeordneten Zellen frei entfalten (**AnchorStyles.None**):

```

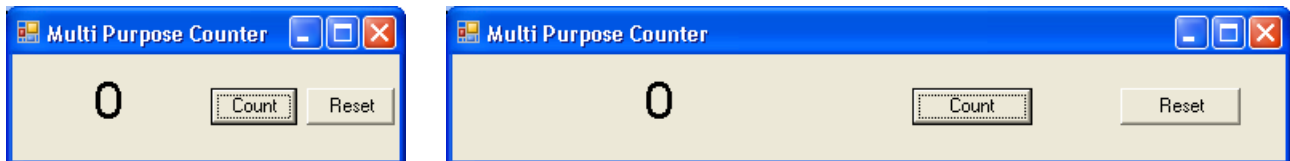
table.Controls.Add(stand, 0, 0);
stand.Anchor = AnchorStyles.None;

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 Beispielsprogramms finden Sie im Ordner

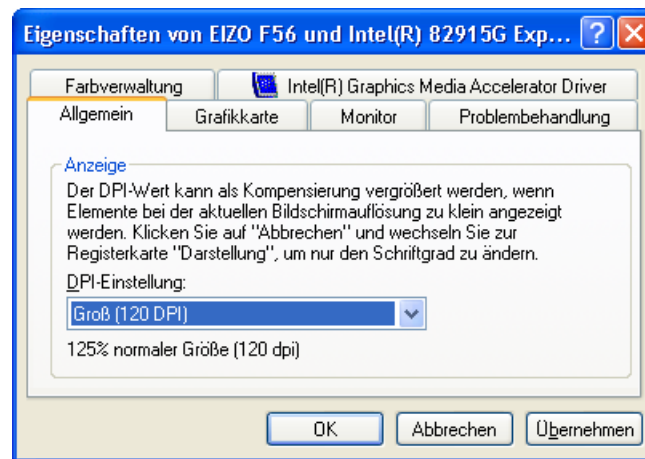
...\\BspUeb\\WinForms\\Spezielle Layout-Techniken\\TableLayoutPanel

Beim Chat-Klienten, den Sie in einer Übungsaufgabe zum Abschnitt 13 (Netzwerkprogrammierung) entwickeln sollen, profitiert das Formulardesign erheblich von der Flexibilität des **TableLayoutPanel**-Steuerelements (siehe Seite 329).

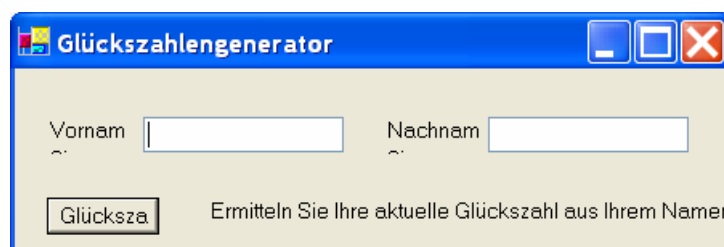
Eine weitere Option zur Gestaltung größenflexibler Formulare ist das (ebenfalls seit .NET 2.0 in der FCL vorhandenen) Steuerelement **FlowLayoutPanel**.

8.4.4.5 Automatische Positions- und Größenanpassungen an die Schriftart des Formulars

Vergrößert ein Benutzer unter Windows XP via **Systemsteuerung > Anzeige**



die Windows-Standardschriftart, dann machen viele Anwendungen eine weniger glückliche Figur, z.B.



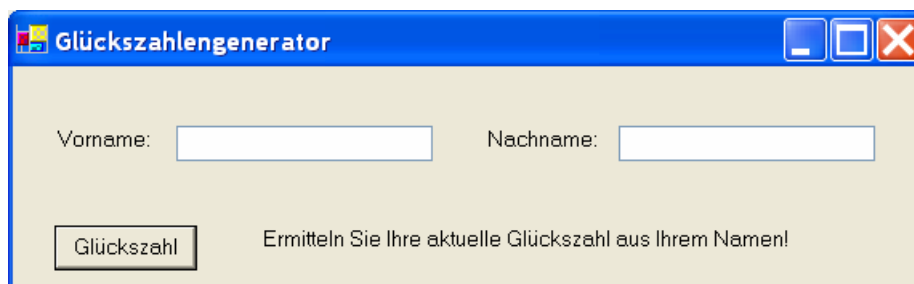
Dieses Formular verwendet die vom Benutzer zwecks verbesserter Lesbarkeit gewünschte Schriftgröße, und diese wird per Voreinstellung auch von den Steuerelementen übernommen. Leider reicht nun die Breite der Steuerelemente nicht mehr aus zur vollständigen Anzeige der Beschriftungen.

Zum Glück bietet das .NET – Framework eine bequeme Option zur automatischen Anpassung aller Positions- und Größenangaben in einem Formular an seine Schriftart, wobei auch die enthaltenen Steuerelemente angepasst werden. Dazu muss lediglich der Formular-Eigenschaft

AutoScaleBaseSize im Konstruktor ein passender Wert explizit zugewiesen werden, z.B.:

```
AutoScaleBaseSize = new Size(5, 13);
```

Hier wird vereinbart, dass sich alle Positions- und Größenangaben im Programm auf eine 5 Pixel breite und 13 Pixel hohe Schrift beziehen. Sorgt der Benutzer für eine größere Schrift (z.B. 6 Pixel breit und 15 Pixel hoch), dann werden alle Höhen mit dem Faktor $6/5$ und alle Breiten mit dem Faktor $15/13$ angepasst. Es resultiert ein größeres Formular mit proportional vergrößerten Steuerelementen, die genügend Platz zur Anzeige der Beschriftungen bieten:

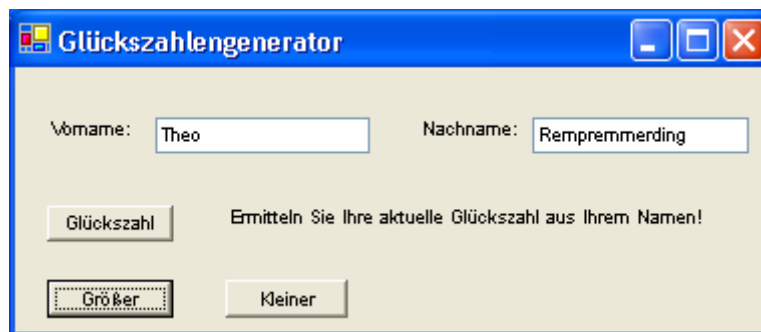
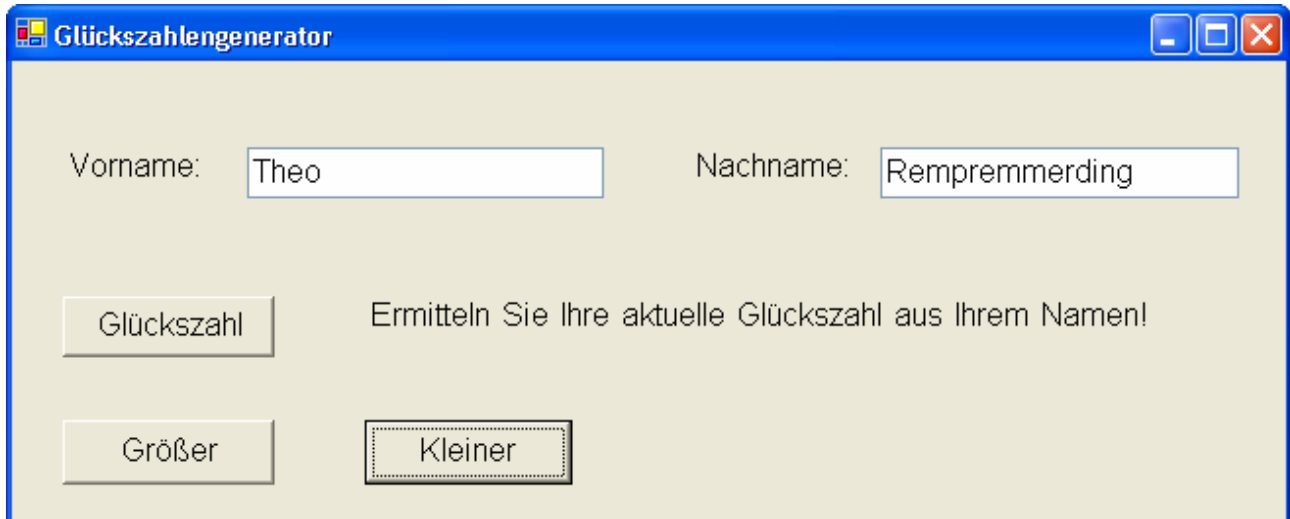


Bei der *nach Beendigung des Formulkonstruktors* einmalig ausgeführten automatischen Skalierung wird sowohl ein vom Benutzer über die Anzeigeeigenschaften des Betriebssystems gewählter Vergrößerungsfaktor als auch ein im Programm über die **Form**-Eigenschaft **Font** festgelegter Schriftgrad berücksichtigt. Man kann also wie im folgenden Beispiel

```
public AutoScaleDemo() {
    Font = new Font("Arial", 12);
    AutoScaleBaseSize = new Size(5, 13);
    . . .
}
```

die Formulschrift ohne Risiko für die sorgfältig ausgetüftelten Anordnungen und Größen von Steuerelementen ändern.

Über die **Control**-Methode **Scale()** lässt sich *jederzeit* eine Skalierung erzwingen, z.B. zur Anpassung an einen während der Laufzeit neu festgelegten Schriftgrad. Bei der folgenden Glückszahlen-Variante kann der Benutzer die gesamte Anwendung mit allen Bedienelementen beliebig vergrößern oder verkleinern:



Die **Click**-Ereignisse der Zoom-Schalter werden in folgender Methode bearbeitet:

```
void ZoomOnClick(object sender, EventArgs e) {
    SizeF vorher = GetAutoScaleSize(Font);
    int fontSize = (int) Font.SizeInPoints;
    if (sender == cbBigger) {
        if (fontSize < 16)
            fontSize++;
    } else {
        if (fontSize > 4)
            fontSize--;
    }
    Font = new Font("Arial", fontSize);
    SizeF nachher = GetAutoScaleSize(Font);
    Scale(nachher.Width/vorher.Width, nachher.Height/vorher.Height);
}
```

8.5 Zusammenfassung

Charakteristische Merkmale einer GUI-Anwendung sind:

- Steuerelemente zur Benutzerinteraktion (Schalter, Optionsfelder, Menüs etc.)
- Maus als Eingabegerät
- Ereignisse, benutzergesteuerter Ablauf
Eine GUI-Anwendung wartet die meiste Zeit darauf, dass eine Ihrer Methoden durch ein Ereignis aufgerufen wird.

Die .NET 2.0 - Plattform bietet im Namensraum **System.Windows.Forms** leistungsfähige Klassen zur GUI-Programmierung.

Als elementare Klassen im Windows Forms - Framework haben Sie kennen gelernt:

- **Form**
Alle Formulare (Fenster) 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.
 - Beim Schließen des Hauptfensters endet das Programm.
- **Application**
Diese Klasse bietet u.a. essentielle statische Methoden zum Betreiben einer WinForms-Anwendung. Durch den Methodenaufruf **Application.Run()** wird ...
 - das **Hauptfenster** der Anwendung geöffnet,
 - das Programm um eine **Nachrichtenschleife** erweitert.

Delegaten

Die Objekte eines Delegates zeigen auf (eine Liste von) Methoden mit einer bestimmten Signatur.

Beim Erstellen eines Delegateobjekts wird dem Konstruktor ein Methodenname als Parameter übergeben.

Beim Aufruf eines Delegateobjekts werden alle Methoden seiner Aufrufliste nacheinander ausgeführt.

Ereignisse

Ein Windows-Programm kann als Ansammlung von Ereignisbehandlungsmethoden aufgefasst werden.

Technisch gesehen sind .NET - Ereignisse Delegatevariablen, die vom Ereignisanbieter bei bestimmten Gelegenheiten aufgerufen werden (z.B. bei einem Klick auf einen Schalter).

Wer auf ein Ereignis reagieren möchte, registriert dort eine Behandlungsmethode:

- Man definiert eine Methode mit passender Signatur,
- und befördert sie per Aktualisierungsoperator „+=“ in die Aufrufliste zum Ereignis

Steuerelemente

Sie besitzen im Vergleich zu gewöhnlichen Objekten 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**

Techniken zur dynamischen Anpassung der Position und Größe von Steuerelementen:

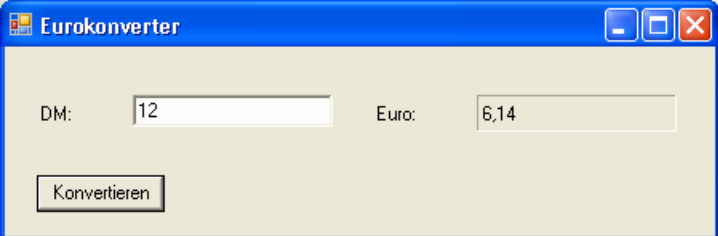
- Verankern
- Andocken
- **TableLayoutPanel**
- **AutoScaleBaseSize**

8.6 Übungsaufgaben zu Kapitel 8

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 Formular.

2) Erstellen Sie (ausnahmsweise 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.

9 Ausnahmebehandlung

Durch Designfehler (z.B. versuchter Feldzugriff mit ungültigem Indexwert) oder durch besondere Umstände (z.B. unterbrochene Netzverbindungen) kann die reguläre Ausführung einer Methode scheitern. C# bietet ein modernes Verfahren zur Meldung und Behandlung solcher Störungen: An der Unfallstelle wird ein Ausnahmeobjekt aus der Klasse **System.Exception** oder aus einer problemspezifischen Unterklasse erzeugt und dem Aufrufer „geworfen“.

Entlang der Aufrufersequenz haben die beteiligten Methoden jeweils zwei Reaktionsmöglichkeiten:

- Ausnahmeobjekt abfangen und das Problem behandeln, wobei oft der im Ausnahmeobjekt enthaltenen „Unfallbericht“ von Nutzen ist
- Ausnahmeobjekt dem Vorgänger in der Aufrufersequenz zuwerfen

Die Initiative beim Auslösen einer Ausnahme kann ausgehen

- von der **CLR**
Entdeckt das Laufzeitsystem einen Fehler, der nicht zu schwerwiegend ist und vom Benutzerprogramm behoben werden kann (z.B. versuchter Feldzugriff mit ungültigem Indexwert), löst es eine **SystemException** aus.
- vom **Benutzerprogramm**, wozu auch die verwendeten Bibliotheksklassen gehören
In jeder Methode kann mit der **throw**-Anweisung (siehe Abschnitt 9.4) eine Ausnahme erzeugt werden. Dies stellt eine effiziente Technik dar, den Aufrufer über ein Problem zu informieren und mit relevanten Informationen zu versorgen.

Wir werden uns anhand von verschiedenen Versionen eines Beispielprogramms damit beschäftigen,

- was bei unbehandelten Ausnahmen geschieht,
- wie man Ausnahmen abfängt,
- wie man selbst Ausnahmen wirft.

9.1 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 int Kon2Int(string instr) {
        int arg = Convert.ToInt32(instr);
        if (arg < 0 || arg > 170) {
            Console.WriteLine("Unzulässiges Argument: "+arg);
            Environment.Exit(1);
            return 0;
        }
        else
            return arg;
    }
}
```

```

static void Main(string[] args) {
    int argument = -1;
    if (args.Length == 0) {
        Console.WriteLine("Kein Argument angegeben");
        Environment.Exit(1);
    }
    argument = Kon2Int(args[0]);
    double fakul = 1.0;
    for (int i = 1; i <= argument; i++)
        fakul = fakul * i;
    Console.WriteLine("Fakultät: " + fakul);
}
}

```

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.
- `Kon2Int()` überprüft, ob die ermittelte **int**-Zahl im zulässigen Wertebereich liegt. Damit werden unsinnige Ergebnisse verhindert (z.B. Fakultät(-4) = 1).

Beide Methoden reagieren auf ein Problem mit dem sofortigen Abbruch des Programms. Dazu rufen sie die Methode **Environment.Exit()** auf, der als Aktualparameter ein Exitcode übergeben wird. Dieser landet beim Betriebssystem und steht in der Umgebungsvariablen `ERRORLEVEL` zur Verfügung, z.B.:

```

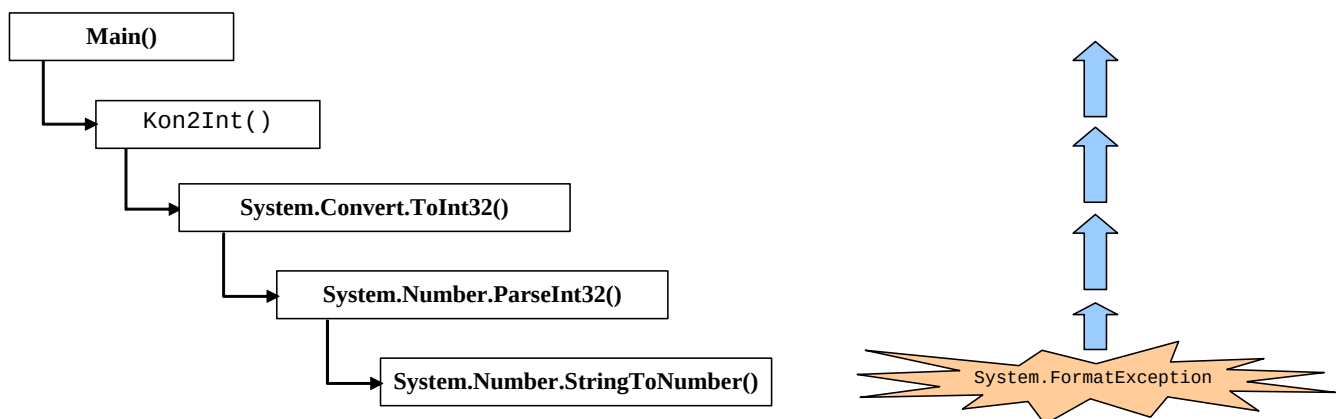
>fakul
Kein Argument angegeben

>echo %ERRORLEVEL%
1

```

Die **Main()**-Reaktion auf ein fehlendes Kommandozeilenargument kann als akzeptable Fehlerbehandlung gelten. Bei der Methode `Kon2Int()`, die eventuell auch im Rahmen anderer Programme zum Einsatz kommt, wirkt der „Notalausstieg“ jedoch etwas plump, und wir werden im weiteren Verlauf von Abschnitt 9 eine sinnvolle Alternative kennen lernen, wobei das Wertebereichsproblem nicht zum Beenden des Programms führt, sondern zur Benachrichtigung des Aufrufers per **Exception**-Objekt.

Trotz seiner präventiven Bemühungen ist das Programm leicht aus dem Tritt zu bringen, indem man es mit einer nicht konvertierbaren Zeichnfolge füttert (z.B. „Vier“). Die zunächst betroffene, FCL-intern aufgerufene, Methode **Number.StringToNumber()** wirft daraufhin eine **FormatException**. Diese wird vom Laufzeitsystem entlang der Aufrufsequenz an alle beteiligten Methoden bis hinauf zu **Main()** gemeldet:



Weil kein Aufrufer eine geeignete Behandlungsroutine bereithält, endet das Programm mit einer Fehlermeldung:

Unbehandelte Ausnahme: System.FormatException: Die Eingabezeichenfolge hat das falsche Format.

```
bei System.Number.StringToNumber(String str, ..., Boolean parseDecimal)
bei System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
bei System.Convert.ToInt32(String value)
bei Fakul.Kon2Int(String instr) in U:\Eigene Dateien\ ... \Fakul.cs:Zeile 5.
bei Fakul.Main(String[] args) in U:\Eigene Dateien\ ... \Fakul.cs:Zeile 21.
```

9.2 Ausnahmen abfangen

9.2.1 Die try-catch - Anweisung

In C# wird die Behandlung von Ausnahmen über die **try**-Anweisung unterstützt:

```
try {
    Überwachter Block mit Anweisungen für den normalen Programmablauf
}
catch (Ausnahme-Klasse Parametername) {
    Anweisungen für die Behandlung von Ausnahmen der ersten Exception-Klasse
}
// Optional können weitere Ausnahmen abgefangen werden:
catch (Ausnahme-Klasse Parametername) {
    Anweisungen für die Behandlung von Ausnahmen der zweiten Exception-Klasse
}

. . .

// Optionaler Block mit Abschlussarbeiten.
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 geschützten bzw. überwachten Blocks *keine* Fehler auf, wird das Programm hinter der **try**-Anweisung fortgesetzt, wobei ggf. vorher noch der **finally**-Block ausgeführt wird.

Tritt im **try**-Block eine Ausnahme auf, wird seine Ausführung abgebrochen. Anschließend sucht das Laufzeitsystem nach einer **catch**-Klausel, welche eine Ausnahme der betroffenen Klasse behandeln kann.

Grundsätzlich besitzt eine **catch**-Klausel, die man oft auch als *Exception-Handler* bezeichnet, analog zu einer Methode im Kopfbereich eine einelementige Parameterliste. Das Laufzeitsystem sucht nach einer **catch**-Klausel, deren Formalparameter vom Typ der zu behandelnden Ausnahme ist, und führt dann den zugehörigen Anweisungsblock aus. Weil die Liste der **catch**-Klauseln von oben nach unten durchsucht wird, müssen *breitere* Ausnahmeklassen stets unter spezielleren stehen.

Folgende Regeln dienen zur Vereinfachung der Schreibweise:

- Man kann auf die Angabe eines Formalparameternamens verzichten, hat dann aber im **catch**-Block kein Ausnahmeobjekt zur Verfügung:

```
catch (Ausnahme-Klasse) {
    Anweisungen für die Behandlung der Ausnahme
}
```

- Fehlt bei einer **catch**-Klausel auch die Typangabe, ist die (maximal breite) Ausnahme-Basisklasse **Exception** eingestellt, was offensichtlich nur bei der letzten **catch**-Klausel sinnvoll ist:

```
catch {
    Anweisungen für die Behandlung der Ausnahme
}
```

In der folgenden Variante der Methode `Kon2Int()` werden die von **Convert.ToInt32()** zu erwartenden Ausnahmen abgefangen:

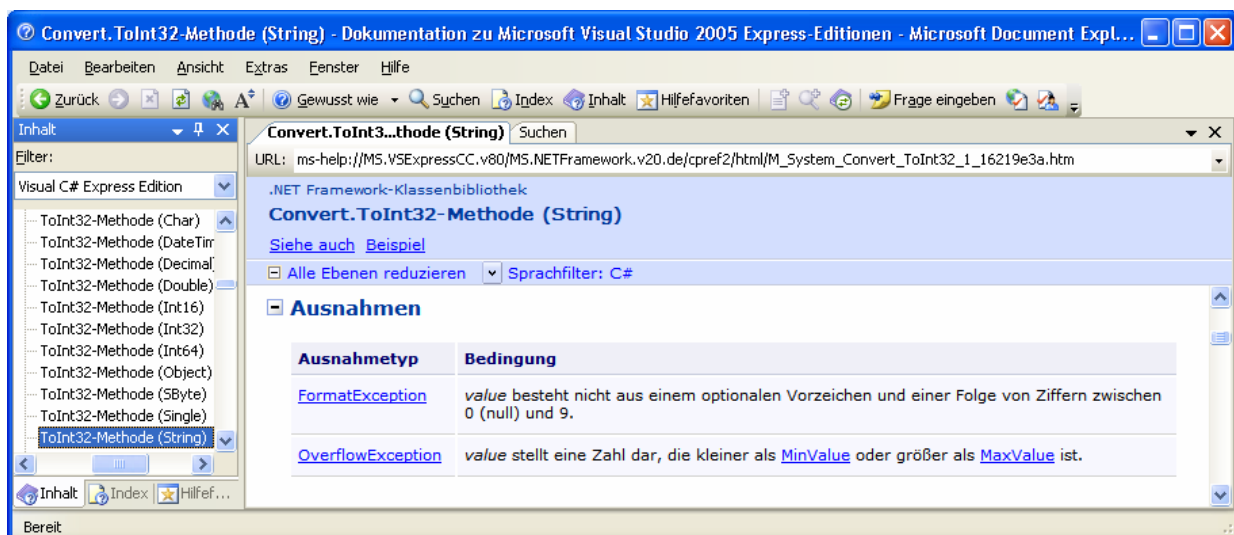
```
static int Kon2Int(string instr) {
    int arg = -1;
    try {
        arg = Convert.ToInt32(instr);
    } catch (FormatException) {
        Console.WriteLine("Ihre Eingabe ist keine Zahl.");
        Environment.Exit(1);
    } catch (OverflowException e){
        Console.WriteLine(e.Message);
        Environment.Exit(1);
    }
    if (arg < 0 || arg > 170) {
        Console.WriteLine("Zahl ausserhalb von [0, 170]");
        Environment.Exit(1);
        return 0;
    }
    else
        return arg;
}
```

Beim Auftreten einer **FormatException** (z.B. wegen des Kommandozeilenarguments „Vier“) endet das verbesserte Programm mit folgender Meldung:

Ihre Eingabe ist keine Zahl.

Weil stets dieselbe Meldung erscheinen soll, wird im Kopf der zugehörigen **catch**-Klausel *kein* Parametername angegeben, das von **Convert.ToInt32()** geworfene Ausnahmeobjekt wird also ignoriert.

Wie die SDK-Dokumentation zu **Convert.ToInt32()** zeigt,



ist außer **System.FormatException** noch eine Ausnahme aus der Klasse **System.OverflowException** möglich, die im Beispiel ebenfalls abgefangen wird.

Im Kopf der zugehörigen **catch**-Klausel ist ein Parametername angegeben, so dass im Anweisungsblock das übergebene Ausnahmeobjekt angesprochen werden kann. Im Beispiel wird seiner **Message**-Eigenschaft (siehe unten) eine Fehlerbeschreibung entnommen.

Ein **finally**-Block wird in jedem Fall ausgeführt, also ...

- nach der ungestörten Ausführung des **try**-Blocks,
- nach einer Ausnahmebehandlung in einem **catch**-Block,
- nach dem Auftreten einer unbehandelten Ausnahme.

In Abschnitt 9.2.2 folgen ausführlichere Erläuterungen zum Programmablauf bei verschiedenen Ausnahme-Konstellationen.

Auch die aktuelle `Kon2Int()`-Variante reagiert auf Probleme mit einem plumpen Notausstieg, doch immerhin erscheinen nun informative Fehlermeldungen des Programms an Stelle des „Absturzprotokolls“ der CLR.

9.2.2 Programmablauf bei der Ausnahmebehandlung

Findet das Laufzeitsystem für eine Ausnahme in der aufrufenden Methode keinen **catch**-Block, dann sucht es entlang der Aufrufsequenz weiter. Dies macht es leicht, die Behandlung einer Ausnahme der bestgerüsteten Methode zu überlassen. In folgendem Beispiel dürfen Sie allerdings eine optimierte Einsatzplanung *nicht* erwarten. Es soll demonstrieren, welche Programmabläufe sich bei Ausnahmen ergeben können, die auf verschiedenen Stufen einer Aufrufhierarchie behandelt werden. 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. `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, hingegen nicht die **ArithmeticException**, die das Laufzeitsystem beim Kommandozeilenargument „0“ werfen wird.

Wir betrachten einige Konstellationen mit ihren Konsequenzen für den Programmablauf:

- a) Normaler Ablauf
- b) Exception in `Calc()`, die dort auch behandelt wird
- c) Exception in `Calc()`, die in **Main()** behandelt wird
- d) Exception in **Main()**, die nirgends behandelt wird

a) Normaler Ablauf

Beim Programmablauf *ohne* Ausnahmen (hier mit Kommandozeilen-Argument „8“) werden die **try**- und die **finally**-Blöcke von **Main()** und `Calc()` ausgeführt:

```
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 „acht“ von **Convert.ToInt32()** geworfen), dann kommt der zugehörige **catch**-Block zum Einsatz. Dann folgen:

- **finally**-Block in `Calc()`
- restliche Anweisungen in `Calc()`

An **Main()** wird keine Ausnahme gemeldet, also werden nacheinander ausgeführt:

- **try**-Block
- **finally**-Block
- restliche Anweisungen

Insgesamt erhält man die Ausgabe:

```
try-Block von Main()
try-Block von Calc()
FormatException-Handler in Calc()
finally-Block von Calc()
Nach try-Anweisung in Calc()
10 % acht = 0
finally-Block von Main()
Nach try-Anweisung in Main()
```

Zu der wenig überzeugenden Ausgabe

```
10 % acht = 0
```

kommt es, weil die **FormatException** in `Calc()` nicht *sinnvoll* behandelt wird. Wir werden bald elegante Möglichkeiten kennen lernen, derartige Pannen zu vermeiden. 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()
```

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.

9.2.3 Vergleich mit der traditionellen Ausnahmebehandlung

Eine konventionelle Ausnahmebehandlung stützt sich wesentlich auf die *Rückgabewerte*, mit denen Funktionen den Aufrufer über das Ergebnis ihrer Bemühungen informieren. Meist wird ein ganzzahliger *Returncode* verwendet, wobei die 0 einen erfolgreichen Ablauf meldet, während andere Zahlen für einen bestimmten Fehlertyp stehen. Nun ist im Prinzip nach jedem Funktionsaufruf der Returncode auszuwerten, so dass eine harmlose Sequenz von drei Funktionsaufrufen:

```
f1();
f2();
f3();
```

nach Ergänzen der Fehlerbehandlung zu einer länglichen und recht unübersichtlichen Konstruktion wird (nach Mössenböck 2001, S. 236):

```

returncode = f1();
if (returncode == 0) {
    returncode = f2();
    if (returncode == 0) {
        returncode = f3();
        if (returncode == 0) {
            . . .
        }
        else {
            Ausnahmebehandlung für diverse f3()-Fehler}
    }
    else {
        Ausnahmebehandlung für diverse f2()-Fehler}
}
else {
    Ausnahmebehandlung für diverse f1()-Fehler}

```

Ein gut gesetzter Rückgabewert nutzt natürlich nichts, wenn sich der Aufrufer nicht darum kümmert. Ebenso unbeachtet kann eine *Statusvariable* bleiben, die oft alternativ zum Returncode als Kommunikationsmittel genutzt wird.

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 kann/soll.

Gegenüber der konventionellen Ausnahmebehandlung hat die in C# realisierte Technik 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 Ausnahme-Behandlungen, 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, meist ganzzahligen, 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.

9.2.4 Diagnostische Ausgaben

Statt im **catch**-Block eine eigene Fehlermeldung zu formulieren, kann man auch die **ToString()**-Methode des übergebenen Ausnahmeobjekts aufrufen, was hier implizit im Rahmen eines **WriteLine()**-Aufrufs geschieht:

```
Console.WriteLine(e);
```

Man erhält:

- den Namen der Ausnahmeklasse
- die beim Erzeugen der Ausnahme formulierte Fehlermeldung
- die Aufrufreihenfolge

Beispiel:¹

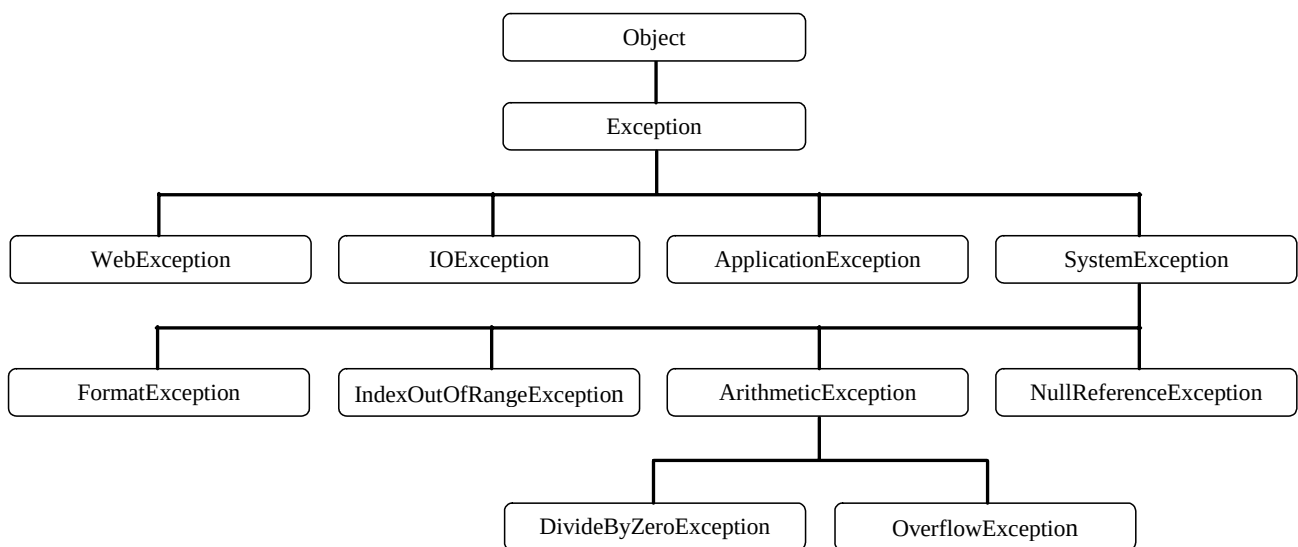
```
System.FormatException: Die Eingabezeichenfolge hat das falsche Format.
bei System.Number.StringToNumber(String str, . . .)
bei System.Number.ParseInt32(String s, . . .)
bei System.Convert.ToInt32(String value)
bei Sequenzen.Calc(String instr) in U:\Eigene Dateien\ ... \Sequenzen.cs:Zeile 8.
```

Soll nur die Fehlermeldung erscheinen, gibt man die **Message**-Eigenschaft des Ausnahmeobjekts aus, z.B.:

```
Console.WriteLine(e.Message);
```

9.3 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.

Sind mehrere **catch**-Blöcke vorhanden, dann werden diese beim Auftreten einer Ausnahme sequenziell von oben nach unten auf Zuständigkeit untersucht. Folglich ist darauf zu achten, dass speziellere Ausnahmeklassen *vor* allgemeineren stehen. Nach Ausführung des ersten passenden **catch**-Blocks wird das Programm hinter der **try**-Anweisung fortgesetzt, wobei ggf. vorher noch der **finally**-Block ausgeführt wird.

Eigene Ausnahmeklassen (siehe Abschnitt 9.5) sollten aus **ApplicationException** abgeleitet werden.

¹ Dateinamen und Zeilennummern enthält die Aufrufreihenfolge nur dann, wenn mit aktiver Debug-Option übersetzt worden ist, z.B.:

```
csc /debug Sequenzen.cs
```

9.4 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 9 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",
            "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.

Die meisten Ausnahmeklassen besitzen u.a. folgende Konstruktoren:

- einen parameterfreien Konstruktor
- einen Konstruktor mit einem **String**-Parameter für eine Fehlermeldung (nähere Beschreibung der Ausnahme)

Im Beispiel wird ein Konstruktor mit zwei **String**-Parametern verwendet, wobei der Name 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
```

In einer **catch**-Klausel darf das Schlüsselwort **throw** auch *ohne* Ausnahmeobjekt-Referenz stehen. In diesem Fall wird die gerade behandelte Ausnahme erneut geworfen.

In der aktuellen `Kon2Int()`-Variante findet sich kein Notausstieg mehr. Es fehlt aber auch eine Behandlung der von **Convert.ToInt32** potentiell zu erwartenden Ausnahmen. 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 vielen Situationen eine praktische Möglichkeit, von verschiedenen potentiellen

Schadstellen aus zur selben Ausnahmebehandlung zu verzweigen. 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

9.5 Ausnahmen definieren

Mit Hilfe von Ausnahmeobjekten kann eine Methode beim Auftreten von Fehlern den Aufrufer ausführlich und präziser 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 class BadFakulArgException : ApplicationException {
    protected int fehlerTyp, wert;
    protected string eingabeString;
    public BadFakulArgException(string fehlerMeldung, string eingabeString_,
                                int fehlerTyp_, int wert_)
        : base(fehlerMeldung) {
        eingabeString = eingabeString_;
        fehlerTyp = fehlerTyp_;
        wert = wert_;
    }
    public string Eingabe {get {return eingabeString;}}
    public int FehlerTyp {get {return fehlerTyp;}}
    public int Wert {get {return wert;}}
}
```

Hier wird die Klasse `BadFakulArgException` aus der Ausnahmeklasse **`ApplicationException`** abgeleitet.

Wir verwenden die handgestrickte Ausnahmeklasse `BadFakulArgException` in einer weiteren Variante unserer Methode `Kon2Int()`, um beim Auftreten von irregulären Argumenten den Aufrufer ausführlich informieren zu können:

- in der Eigenschaft **Message** (geerbt von **`ApplicationException`**) steht wie üblich eine Fehlermeldung
- in der Eigenschaft `Eingabe` steht die zu konvertierende Zeichenfolge
- in der Eigenschaft `FehlerTyp` wird ein numerischer Indikator für die Fehlerart angeboten, z.B.:
 - o 1: Zeichenfolge kann nicht konvertiert werden
 - o 2: Ganzzahlüberlauf
 - o 3: Wert außerhalb [0, 170]

- in der Eigenschaft `wert` 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(e.Message, instr, 2, -1);
    }
    catch (FormatException) {
        throw new BadFakulArgException("Fehler beim Konvertieren", instr, 1, -1);
    }
}
```

In der **Main()**-Methode 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(e.Message);
        Console.WriteLine("Fehlertyp: " + e.FehlerTyp);
        if (e.FehlerTyp == 3)
            Console.WriteLine("Wert: " + e.Wert);
        else
            Console.WriteLine("String: " + e.Eingabe);
        Environment.Exit(1);
    }
    double fakul = 1.0;
    for (int i = 1; i <= argument; i++)
        fakul = fakul * i;
    Console.WriteLine("Fakultät: " + fakul);
}
```

9.6 Übungsaufgaben zu Kapitel 9

1) Erstellen Sie ausnahmsweise ein Programm, das eine **NullReferenceException** auslöst. Hinweis: Man kann eine Referenzvariable „unbrauchbar“ machen, indem man ihr den Wert **null** zuweist.

2) Beim Rechnen mit Gleitkommazahlen produziert C# in kritischen Situationen üblicherweise keine Ausnahmen, sondern operiert mit speziellen Werten wie **Double.POSITIVE_INFINITY** oder **Double.NaN**. Dieses Verhalten ist sicher oft nützlich, kann aber eventuell 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 **Math.Log()** zurückgreift und daher bei ungeeigneten Argumenten (≤ 0) als Rückgabewert **Double.NaN** liefert.

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(8); double b = Log2(-1); Console.WriteLine(a*b); } }</pre>	n. def.

Erstellen Sie eine Version, die bei ungeeigneten Argumenten eine **ArgumentOutOfRangeException** wirft.

¹ Für eine positive Zahl a ist ihr Logarithmus zur Basis b (> 0) definiert durch:

$$\log_b(a) := \frac{\ln(a)}{\ln(b)}$$

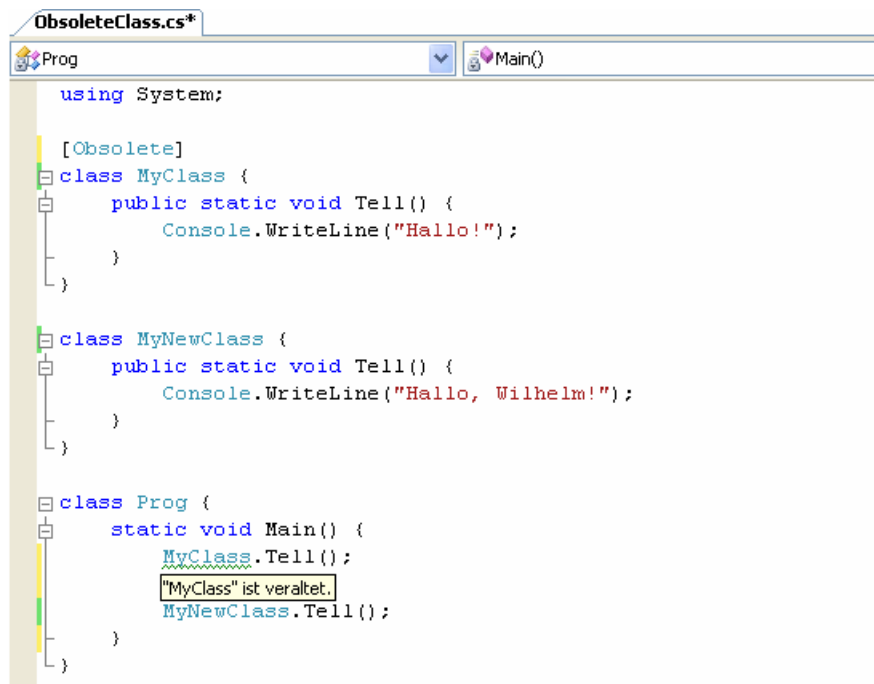
Dabei steht $\ln()$ für den natürlichen Logarithmus zur Basis e (Eulersche Zahl).

10 Attribute

An Typen (Klassen, Schnittstellen, Strukturen, usw.), Member (Methoden, Eigenschaften, usw.) sowie Assemblies und Module kann man Attribute anheften, um zusätzliche Metainformationen bereit zu stellen, die beim Übersetzen und zur Laufzeit berücksichtigt werden können.

10.1 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# 2005 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 diese Compiler-Meldung zu dürftig findet, kann das Attribut **Obsolete** mit Parametern anreichern, z.B.:

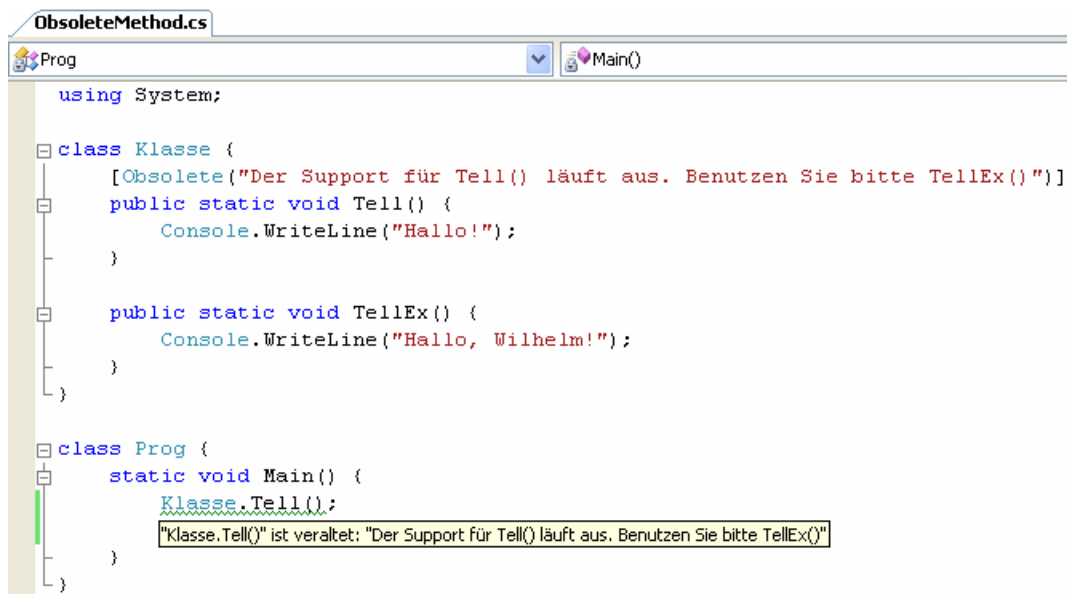
```
[Obsolete("Benutzen Sie bitte MyNewClass")]
```

Dann meldet der Compiler:

```
ObsoleteClass.cs(19,9): warning CS0618: MyClass ist veraltet: Benutzen Sie bitte MyNewClass.
```

In der zweiten Variante kommt ein alternativer Konstruktor der Klasse **ObsoleteAttribute** (aus dem Namensraum **System**) zum Einsatz. Sie haben richtig gelesen: bei den Attributen handelt es sich um spezielle Klassen (abstammend von **System.Attribute**). Bei der Vergabe eines Attributes entsteht ein Objekt der entsprechenden Klasse und ergänzt die Metadaten der Zielklasse. Als syntaktische Besonderheit darf bei der Vergabe eines Attributes (also beim Erstellen eines Attributobjekts) der Namensteil *Attribute* weglassen werden (siehe Beispiel).

Neben Klassen können auch andere Programmbestandteile mit Attributen versehen werden, z.B. Methoden:



Beim Übersetzen meldet der Compiler:

ObsoleteMethod.cs(16,3): warning CS0618: Klasse.Tell() ist veraltet: Der Support für Tell() läuft aus. Benutzen Sie bitte TellEx().

In den vom Visual Studio oder einer vergleichbaren Entwicklungsumgebung unter Verwendung der Vorlage **Windows-Anwendung** erstellten Projekten ist die Startmethode **Main()** (zu finden in der Quellcodedatei **Program.cs**) mit 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 Common Object Model (COM), der noch sehr weit verbreiteten Windows-Komponententechnologie, das Threading-Modell STA (Single-Threaded Apartment) zum Einsatz kommen soll. Wir werden uns mit Threads und der COM-Interoperabilität später noch beschäftigen. Bei unseren derzeitigen Programmen spielt das Attribut **STAThread** keine Rolle, so dass wir es bei selbst erstelltem Quellcode nicht setzen.

10.2 Attribute auswerten

Mit der Methode **GetCustomAttributes()** der Klasse **System.Type** lassen sich zur Laufzeit Informationen über Attribute von Programmbestandteilen gewinnen, z.B.:

```

using System;
[Obsolete("Benutzen Sie bitte MyNewClass")]
class MyClass {
    public static void Tell() {
        Console.WriteLine("Hallo!");
    }
}
class Prog {
    static void Main() {
        Type type = typeof(MyClass);
        foreach (Attribute at in type.GetCustomAttributes(false))
            if (at is ObsoleteAttribute) {
                Console.WriteLine("MyClass ist obsolet");
                Console.WriteLine((at as ObsoleteAttribute).Message);
            }
    }
}

```

In der **Main()**-Methode wird zur Klasse `MyClass` ein Beschreibungsobjekt aus der Klasse **Type** erzeugt. Dieses wird per **GetCustomAttributes()** beauftragt, einen Array mit allen der Klasse angehefteten benutzerdefinierten Attributen abzuliefern, wobei geerbte Attribute nicht interessieren (Wert **false** für den Parameter **inherit**). Anschließend wird untersucht, ob ein **ObsoleteAttribute**-Objekt vorhanden ist.

Mit dem bisher noch nicht vorgestellten **as**-Operator lassen sich in der Anweisung

```
Console.WriteLine(((ObsoleteAttribute)at).Message);
```

im Vergleich zum Typwandlungsoperator zwei Klammern einsparen.

Das Beispielprogramm liefert folgende Ausgabe:

```
SomeClass ist obsolete
Benutzen Sie bitte MyNewClass
```

Auf einige *Assembly*-Attribute (z.B. mit Versionsangaben) kann man sehr bequem über statische Eigenschaften der Klasse **Application** zugreifen (siehe Abschnitt 10.4).

10.3 Attribute definieren

Eigene Attribute zu definieren, ist eher selten erforderlich, macht aber auch keinerlei Probleme, z.B.:

```
public class MyAttribute : Attribute {
    private int level;
    public MyAttribute(int level_) {
        level = level_;
    }
    public int Level {
        get {
            return level;
        }
    }
}

[MyAttribute(13)]
[Obsolete("Useless class")]
class SomeClass {
    public string Dummy {get {return "Dummy";}}
}
```

10.4 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 die Visual C# 2005 Express Edition beim Erstellen eines neuen Projekts automatisch bei Verwendung der Vorlage **Windows-Anwendung**, wobei die Entwicklungsumgebung jeweils eine eigene 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 2007")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
[assembly: ComVisible(false)]
[assembly: Guid("80bc8315-7573-45d7-aaae-ff44698c81e7")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]

```

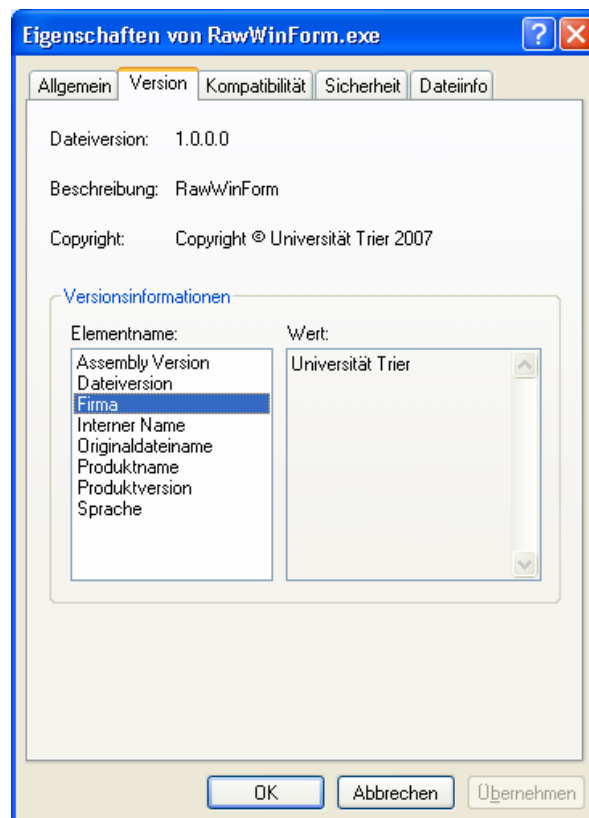
Auf einige Assembly-Attribute kann man im Programm über statische Eigenschaften der Klasse **Application** zugreifen, z.B.:

```

private void button1_Click(object sender, EventArgs e) {
    MessageBox.Show("Version "+Application.ProductVersion,
        Application.ProductName, MessageBoxButtons.OK,
        MessageBoxIcon.None, MessageBoxDefaultButton.Button1);
}

```

Außerdem erscheinen die Assembly-Attribute im Eigenschaftsdialog einer Assembly-Datei, z.B.:



10.5 Übungsaufgaben zu Kapitel 10

1) Erweitern Sie das Beispiel in Abschnitt 10.3 um eine „**Main()**-Klasse“, die zur Klasse SomeClass alle benutzerdefinierten Attribute und insbesondere den Level-Wert des MyAttribut-Objekts ausgibt.

11 Ein-/Ausgabe über Datenströme

Viele Programme benötigen die Fähigkeit, Daten einzulesen und/oder Verarbeitungsergebnisse abzuspeichern, wobei als Quellen bzw. Senken z.B. Dateien oder Netzwerkverbindungen in Frage kommen. Wir beschränken uns in diesem Abschnitt auf einfache Verfahren, um Werte elementarer Typen (z.B. **int**, **double**), Textdaten oder ganze 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:

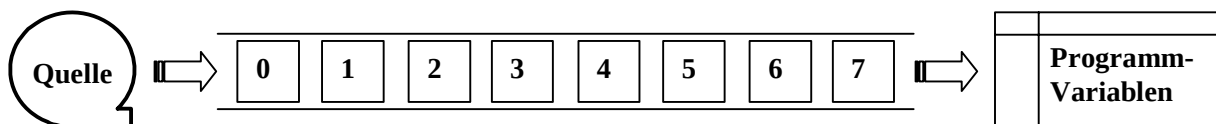
```
using System.IO;
```

11.1 Datenströme aus Bytes

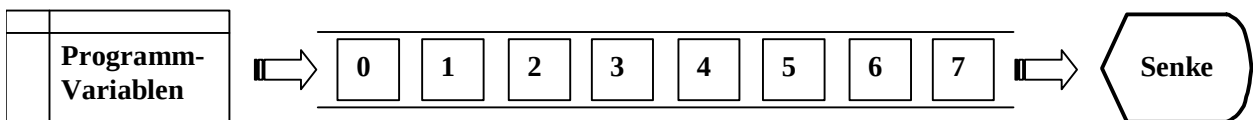
11.1.1 Das Grundprinzip

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):



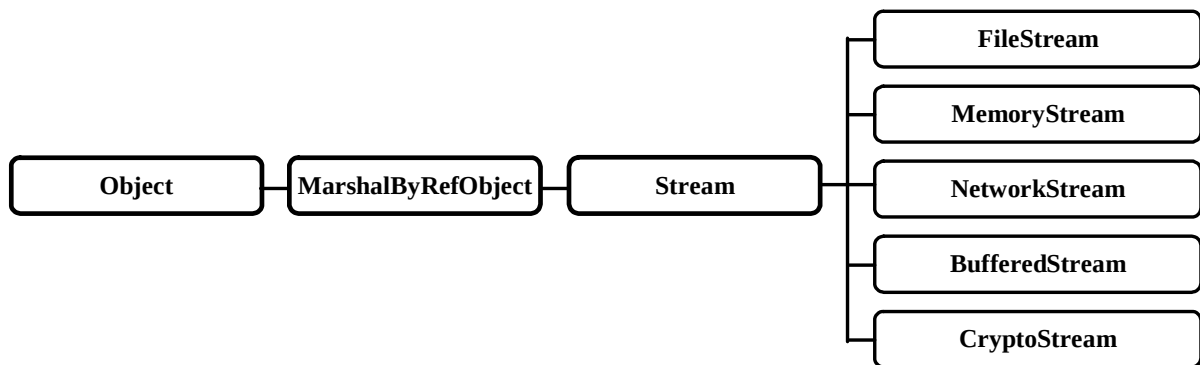
Ein- bzw. Ausgabeströme werden in .NET - Programmen durch Objekte aus geeigneten Klassen des Namensraums **System.IO** repräsentiert, wobei die Auswahl u.a. von der angeschlossenen Datenquelle bzw. -senke sowie vom Typ der zu transportierenden Daten abhängt.

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 Sie Erläuterungen in den nächsten Abschnitten.

11.1.2 Wichtige Methoden und Eigenschaften der Basisklasse Stream

Alle Ableitungen der abstrakten Basisklasse **Stream** verfügen u.a. über die folgenden Methoden und Eigenschaften:

- **Eigenschaft Position**
Mit dieser Eigenschaft wird die aktuelle Position im Strom angesprochen. Bei einer Datei kann die Lese-/Schreibposition ermittelt oder gesetzt werden, z.B.:
`fs.Position = 0;`
- **Methode ReadByte()**
Diese Methode liest ein Byte aus dem Strom und liefert es als Rückgabewert. Die aktuelle Position im Strom wird entsprechend um ein Byte erhöht. Ist das Ende des Stroms erreicht, wird keine Ausnahme ausgelöst, sondern der Rückgabewert -1 geliefert.
- **Methode Read(byte[] buffer, int offset, int count)**
Dieser Methode liest eine Serie von *count* Bytes aus dem Strom und legt sie im **byte**-Array *buffer* ab Position *offset* ab. Die aktuelle Position des Stroms wird um *count* Bytes erhöht.

- Methode **WriteByte()**
Diese Methode schreibt ein Byte in den Strom und erhöht die aktuelle Position des Stroms entsprechend um ein Byte. Ist das Ende des Stroms erreicht, wird keine Ausnahme ausgelöst, sondern der Rückgabewert -1 geliefert.
- Methoden **Write(byte[] buffer, int offset, int count)**
Diese Methode schreibt eine Serie von *count* Bytes, die im **byte**-Array *buffer* ab Position *offset* liegen, in den Strom und erhöht die Position des Stroms entsprechend.
- Methode **Flush()**
Ausgabeströme verwenden in der Regel einen Puffer, um die Anzahl der Zugriffe auf die angeschlossene Senke möglichst gering zu halten. Mit **Flush()** wird dieser Puffer ausgegeben, so dass der aktuelle Inhalt für Abnehmer zur Verfügung steht. Bei der Klasse **FileStream** (siehe unten) wird eine voreingestellte Puffergröße von 4096 Bytes benutzt; Netzwerkströme verzichten auf einen Puffer.
- Methode **Close()**
Mit dem Schließen eines Datenstroms per **Close()**-Aufruf oder durch eine alternative Technik beschäftigen wir uns gleich in Abschnitt 11.1.3.
- Eigenschaft **Length**
Mit dieser Eigenschaft wird die Länge des Stroms in Bytes angesprochen. Bei einer Datei enthält **Length** die Anzahl der darin befindlichen Bytes.
- Methode **Seek(long offset, SeekOrigin origin)**
Mit dieser Methode kann man den Strom relativ zu einem Bezugspunkt neu positionieren, der über einen Wert der Enumeration **SeekOrigin** zu wählen ist (Werte: **Begin**, **Current**, **End**).
Bei einer Datei kann die Lese-/Schreibposition gesetzt werden, z.B.:
`fs.Seek(-4, SeekOrigin.Current);`
- Eigenschaften **CanRead**, **CanSeek**, **CanWrite**
Über diese Eigenschaften lässt sich feststellen, ob ein Strom das Lesen, Schreiben und Positionieren erlaubt, z.B.:

Quellcode	Ausgabe
<code>Console.WriteLine("CanRead " + fs.CanRead);</code>	CanRead True
<code>Console.WriteLine("CanSeek " + fs.CanSeek);</code>	CanSeek True
<code>Console.WriteLine("CanWrite " + fs.CanWrite);</code>	CanWrite True

11.1.3 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 Lese- bzw. Schreibversuche zu Ausnahmefehlern. Rufen Sie **Close()** also *nicht* auf, wenn solche Ausnahmefehler möglich sind, weil im Programm noch weitere Referenzen auf das **Stream**-Objekt existieren (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 Objektes per Garbage Collector alle verwende-

ten 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 be-reithalten, weil sie das Interface **IDisposable** (man sagt auch: *das Beseitigungsmuster*) implemen-tiert.

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 11.2.2). Es kommt zu Datenverlusten wenn der Garbage Collector bei seinen Aufräumarbeiten (ohne garantierte Reihen-folge!) das **Stream**-Objekt *vor* dem **StreamWriter**-Objekt beseitigt.

Außerdem kann es sinnvoll sein, Ströme durch ein explizites **Close()** so früh wie möglich zu schlie-ßen, um (exklusive) Zugriffe durch andere Programme zu ermöglichen.

Im Beispiel FSDemo ist das Schließen der Datei erforderlich, um sie anschließend löschen zu kön-nen.

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 **us-ing**-Blocks gültig (referenziert) sind. Beim Verlassen des Blocks werden automatisch per **Dispo-se()**-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 zwar das Schließen eines Stroms recht elegant gelöst, doch fehlt noch eine Behandlung der bei Ein-/Ausgabe – Operationen zu erwartenden Ausnahmen (siehe Abschnitt 11.1.4).

11.1.4 Ausnahmefehler abfangen

Weil Methodenaufrufe bei Datenstromobjekten diverse Ausnahmen produzieren können (z.B. **IOException**), sind sie am besten im geschützten Block einer **try-catch** – Anweisung untergebracht. Wenn ein **Close()**-Aufruf angemessen ist (vgl. die Warnungen und Empfehlungen in Abschnitt 11.1.3), gehört er in den **Finally**-Block einer **try-catch** – Anweisung, damit er auf jeden Fall ausge-führt wird. Mit Ausnahmebehandlung wird das Beispielprogramm FSDemo etwas komplizierter:

```

using System;
using System.IO;

class FSDemo {
    static void Main() {
        FileStream fs = null;
        String name = "demo.bin";
        byte[] arr = {0,1,2,3,4,5,6,7};
        try {
            fs = new FileStream(name, FileMode.Create);
            fs.Write(arr, 0, arr.Length);
            fs.Position = 0;
            fs.Read(arr, 0, arr.Length);
            foreach (byte b in arr)
                Console.WriteLine(b);
        } catch (Exception e) {
            Console.WriteLine(e.Message);
        } finally {
            if (fs != null) {
                fs.Close();
                try {
                    File.Delete(name);
                } catch (Exception e) {
                    Console.WriteLine("Fehler beim Löschen: \n"+e.Message);
                }
            }
        }
    }
}

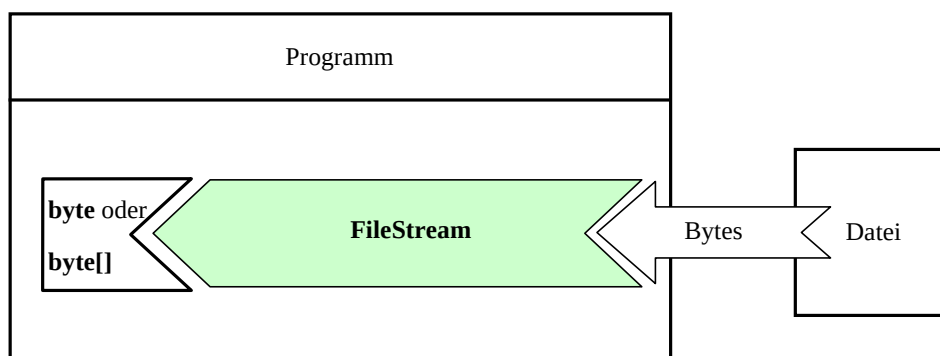
```

Hier sind nun aber auch sehr unwahrscheinliche Fehlermöglichkeiten berücksichtigt. Z.B. findet das Löschen der temporären Datei in einer eigenen **try-catch** – Anweisung statt, um folgende Konstellation zu berücksichtigen: Nach dem Schließen der Datei (Voraussetzung für das Löschen) wird diese von einem anderen Prozess zum Schreiben geöffnet und kann daher nicht gelöscht werden.

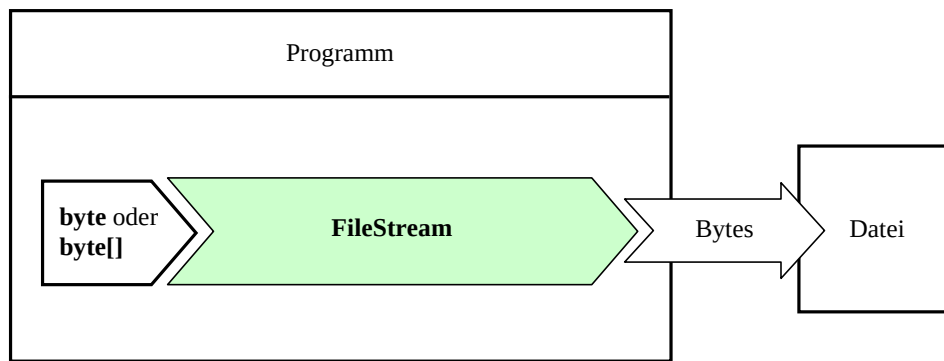
Bei den Beispielen im weiteren Verlauf von Abschnitt 11 werden wir der Einfachheit halber meist auf eine Ausnahmebehandlung verzichten.

11.1.5 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.

11.1.5.1 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.

11.1.5.2 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);
```

11.1.5.3 Optionen für gemeinsamen Zugriff

Durch den Eröffnungsmodus wird auch geregelt, ob die Datei von anderen Prozessen zum gemeinsamen Zugriff erneut geöffnet werden darf. Die Eröffnungsmodi **FileMode.Append** und **FileMode.Create** erlauben z.B. 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);
```

11.2 Verarbeitung von Daten mit höherem Typ

Bisher haben wir uns auf das Schreiben und Lesen von *Bytes* beschränkt. Im Abschnitt 11.2 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.

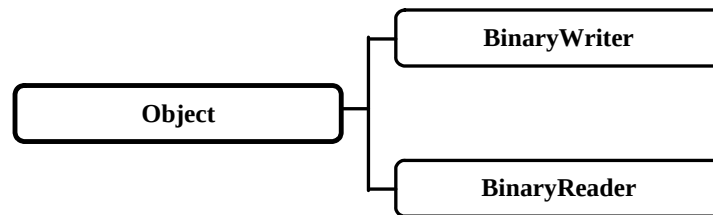
Bei den Dateien auf einem Rechner kann man unterscheiden:

- **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 11.2.1 werden die zum Schreiben bzw. Lesen binärer Daten optimierten Klassen **BinaryWriter** bzw. **BinaryReader** vorgestellt.
- **Textdateien**
Diese Dateien können von Menschen mit einem simplen 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 11.2.2 werden die zum Schreiben bzw. Lesen von Zeichenfolgen optimierten Klassen **StreamWriter** bzw. **StreamReader** vorgestellt.

In Abschnitt 11.2.3 beschäftigen wir uns mit dem Schreiben und Lesen von kompletten Objekten (oder auch Struktur-Instanzen), wobei eine Binärdatei oder eine XML-formatierte Textdatei zum Einsatz kommen kann.

11.2.1 Schreiben und Lesen im Binärformat

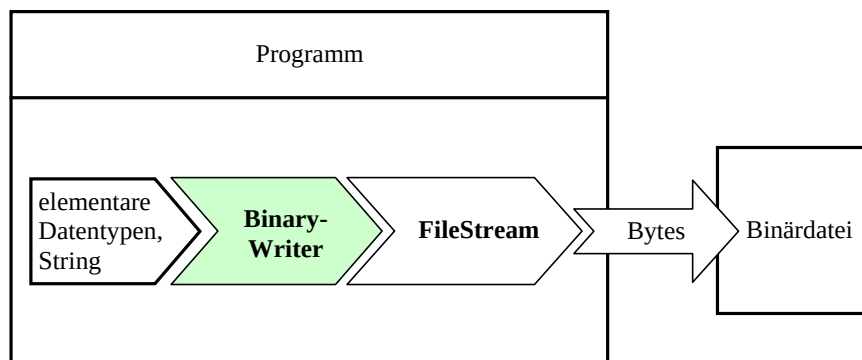
Um Werte von einem beliebigen elementaren Datentyp (z.B. **int**, **double**) sowie Zeichenfolgen in einen binär organisierten Strom (z.B. in eine Binärdatei) zu schreiben bzw. aus einem Binärstrom mit bekanntem Aufbau zu lesen, verwendet man ein Objekt der Klasse **BinaryWriter** bzw. **BinaryReader**.



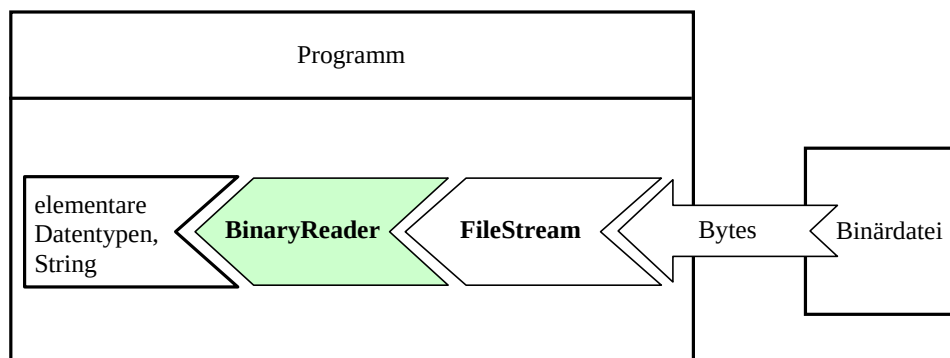
Die beiden Klassen stammen *nicht* von **Stream** ab, verwenden aber für die Verbindung mit einer Datenquelle oder –senke ein **Stream**-Objekt, das im Konstruktor anzugeben ist, z.B.:

BinaryWriter(**Stream** *ausgabestrom*)

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() {
        const 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 0 zu setzen (siehe Beispiel in Abschnitt 11.1.1).

Die von beiden **FileStream**-Objekten verwendete Datei wird zunächst mit dem **FileMode.Create** geöffnet (Schreiben und Lesen erlaubt). Nach den Schreibzugriffen per **Write()**-Methode wird die Datei per **Close()**-Aufruf geschlossen, damit das anschließende Öffnen mit dem **FileMode.Open** gelingt.

Durch den **Close()**-Aufruf wird der Schreibpuffer des **FileStream**-Objekts *fso* geleert, so dass die geschriebenen Daten komplett in der Datei ankommen. Ein **BinaryWriter** verwaltet übrigens *keinen* eigenen Puffer, sondern reicht Schreibaufträge stets direkt an das angeschlossene **Stream**-Objekt weiter, wie der folgende Ausschnitt aus dem **BinaryWriter**-Quellcode zeigt:

```

public virtual void Write(int value) {
    _buffer[0] = (byte)value;
    _buffer[1] = (byte)(value >> 8);
    _buffer[2] = (byte)(value >> 16);
    _buffer[3] = (byte)(value >> 24);
    OutStream.Write(_buffer, 0, 4);
}

```

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 einfach durchgereicht.

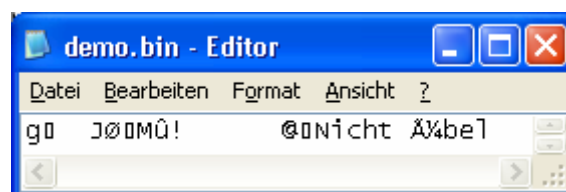
Die Ausgabe des Programms ist nicht übel:

```

4711
3,1415926
Nicht übel

```

Es macht aber wenig Sinn, die vom Beispielprogramm erzeugte Binärdatei mit einem Texteditor zu öffnen:



Um elementare Datentypen im Textformat zu schreiben, verwendet man ein **TextWriter**-Objekt (siehe Abschnitt 11.2.2).

Bei der Ausgabe von **char**- und **String**-Variablen per **BinaryWriter** ist die Kodierung relevant. Wer das voreingestellte **UTF8Encoding** ersetzen möchte, hat einen **BinaryWriter**-Konstruktor mit entsprechendem Parameter zur Verfügung:

BinaryWriter(Stream ausgabestrom, Encoding kodierung)

Im Unterschied zu den „bidirektionalen“ **Stream**-Klassen sind für das Schreiben bzw. Lesen von elementaren Datenwerten „gerichtete“ Klassen zuständig (**Writer** bzw. **Reader**). Während die Klasse **BinaryWriter** für alle unterstützten Datentypen eine Überladung der Methode **Write()** be-

sitzt, sind in der Klasse **BinaryReader** typspezifisch benannte Lesemethoden vorhanden (z.B. **ReadInt32()**, **ReadDouble()**).

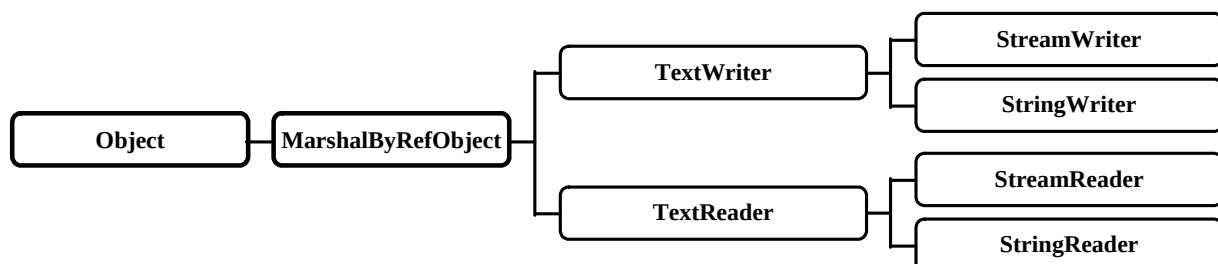
Es besteht übrigens kein Risiko, wenn ein Gespann aus einem **BinaryWriter**-Objekt und einem **FileStream**-Objekt dem Garbage Collector anheim fallen, obwohl beim automatischen Finalisieren (ohne garantierte Reihenfolge!) zuerst das **FileStream**-Objekt beseitigt werden könnte:

- 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.

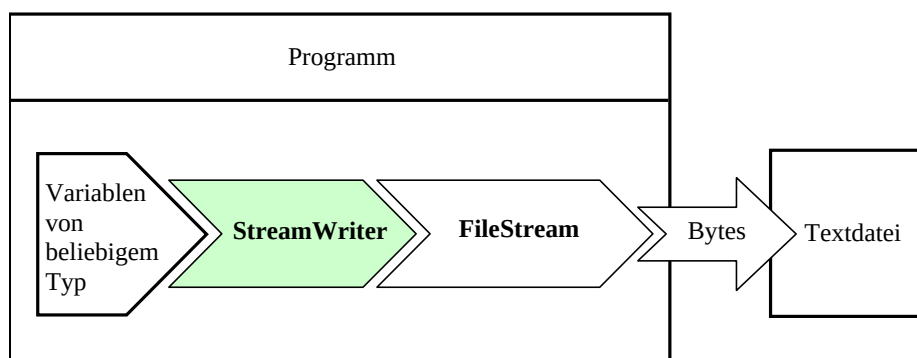
Objekte der anschließend behandelten Klasse **StreamWriter** müssen aufgrund ihres lokalen Puffers jedoch unbedingt *vor* dem zugrunde liegenden **Stream**-Objekt geschlossen werden, was nur durch einen expliziten **Close()**-Aufruf (oder einen äquivalenten **Dispose()**-Aufruf) sicher gestellt ist. Es wäre also verständlich, wenn Sie sich der Einfachheit halber dafür entscheiden, grundsätzlich die „höheren“ Ausgabeobjekte explizit zu schließen, wobei der Basisstrom automatisch ebenfalls geschlossen wird.

11.2.2 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). Beide Klassen sind abstrakt, doch bietet die FCL auch konkrete Ableitungen für **Stream**- bzw. **String**-Objekte als Senken bzw. Quellen:

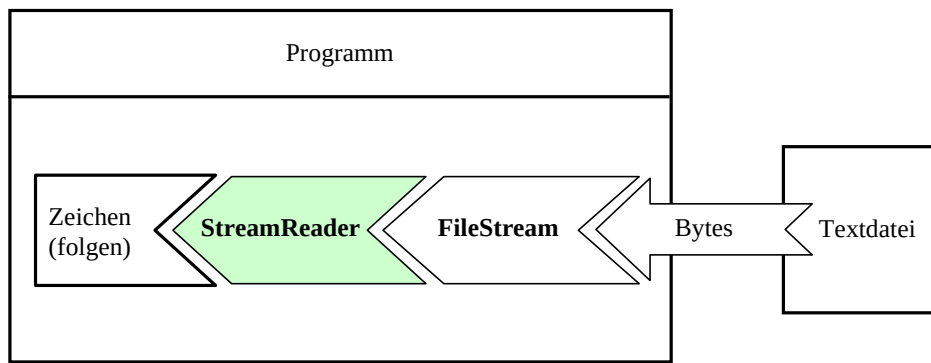


Um in eine *Textdatei* zu schreiben bzw. von dort zu lesen, verwendet man Objekte der Klassen **StreamWriter** bzw. **StreamReader**, die jeweils über eine Instanzvariable auf ein **FileStream**-Objekt aufsetzen (wie bei den Klassen **BinaryWriter** und **BinaryReader**).¹ Beim Schreiben haben wir also folgende Situation:



Beim Lesen aus einer Textdatei reisen die Daten in umgekehrter Richtung:

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.



Der Basisstrom für einen **StreamWriter** oder **-Reader** muss nicht explizit erzeugt werden, 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.

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() {
        const 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 " +
            ((FileStream)sr.BaseStream).Name + "\n");
        for (int i = 0; sr.Peek() >= 0; i++)
            Console.WriteLine(i + ":\t" + 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
```

und 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 11.2.1) besitzt ein **StreamWriter** einen lokalen Puffer (Datentyp: **char[]**, voreingestellte Größe: 1024). Daher muss ein **StreamWriter** unbedingt nach Gebrauch explizit 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 wird. 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 pufferndes **StreamWriter**-Objekt ...

- wird zunächst der interne Puffer ausgegeben
- und dann ein **Flush()**-Aufruf an das **Stream**-Objekt gerichtet.

Hinweise zu einigen **TextReader**-Methoden:

- **ReadLine()** liest eine Zeile, liefert das Ergebnis als **String**-Objekt ab und verschiebt die Position des Eingabestroms entsprechend.
- **Read()** liefert als **int**-Wert 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.
- **Peek()** liefert ebenfalls 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. Daher lässt sich per **Peek()**-Aufruf das Ende eines Stroms diagnostizieren, ohne den Zeiger des Stroms zu verändern, z.B.:

```
for (int i = 0; sr.Peek() >= 0; i++ )
    Console.WriteLine(i + ":\t" + sr.ReadLine());
```

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:

¹ Die folgende Implementierung der **Write()**-Methode mit einfachem **char**-Parameter aus der Klasse **StreamWriter** zeigt, wie die **AutoFlush**-Eigenschaft über die private **bool**-Variable **autoFlush** umgesetzt wird:

```
public override void Write(char value) {
    if (charPos == charLen) Flush(false, false);
    charBuffer[charPos++] = value;
    if (autoFlush) Flush(true, false);
}
```

Unicode-Zeichen		Anzahl Bytes
von	bis	
\u0000	\u0000	2
\u0001	\u007F	1
\u0080	\u07FF	2
\u0800	\uFFFF	3

Bei einigen Überladungen des **StreamWriter**-Konstruktors lassen sich auch alternative Kodierungen einstellen, z.B.:

```
FileStream fs = new FileStream("unicode.txt", FileMode.Create);
StreamWriter swUnicode = new StreamWriter(fs, Encoding.Unicode);
```

Die statische Eigenschaft **Unicode** der Klasse **Encoding** im Namensraum **System.Text** zeigt auf ein Objekt der Klasse **UnicodeEncoding**, das die Kodierung übernimmt. Es verzichtet auf Platzsparmaßnahmen und verwendet für *jedes* Zeichen 16 Bit.

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 {
    const 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());
    }
}
```

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 folgendem Ergebnis 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.

11.2.3 Serialisieren von Objekten und Instanzen

Wer objektorientiert programmiert, möchte natürlich auch objektorientiert speichern und laden. Erfreulicherweise können in C# Objekte von Klassen (und auch Instanzen von Strukturen) tatsächlich genau so einfach wie elementare Datentypen in einen Datenstrom geschrieben bzw. von dort gelesen werden. Die Übersetzung eines Objektes mit all seinen Instanzvariablen und den enthalte-

nen (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.

Das Serialisieren der Objekte einer Klasse muss durch das Attribut **Serializable** explizit erlaubt werden. Bei Bedarf können dabei 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.

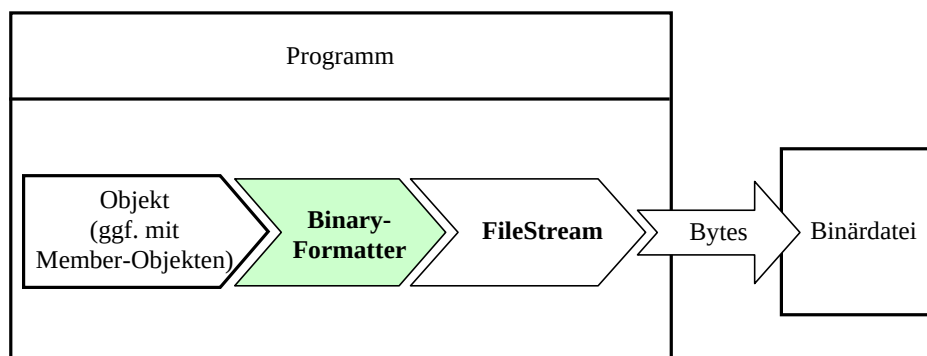
Die in folgender Quelle definierte Klasse Kunde ist als serialisierbar definiert, wobei jedoch für das Feld `stimmung` eine Ausnahme gemacht wird:

```
using System;
using System.Runtime.Serialization;

[Serializable]
public class Kunde {
    int nr;
    string vorname;
    string name;
    [NonSerialized]
    int stimmung;
    int nkaeufo;
    double aussen;
    public Kunde(int nr_, string vorname_, string name_, int stimmung_,
                 int nkaeufo_, double aussen_) {
        nr = nr_;
        vorname = vorname_;
        name = name_;
        stimmung = stimmung_;
        nkaeufo = nkaeufo_;
        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" + nkaeufo);
        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 {
    const 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();

        foreach (Kunde k in kunden)
            bifo.Serialize(fs, k);

        fs.Position=0;
        Console.WriteLine("\nRekonstruiert:\n");
        for (int i = 0; i < kunden.Length; i++) {
            Kunde unbekannt = (Kunde)bifo.Deserialize(fs);
            unbekannt.prot();
        }
    }
}
```

Beim Schreiben eines Objekts wird auch seine Klasse dokumentiert. Beim Lesen eines Objekts 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. Dabei wird kein Konstruktor aufgerufen.

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
```

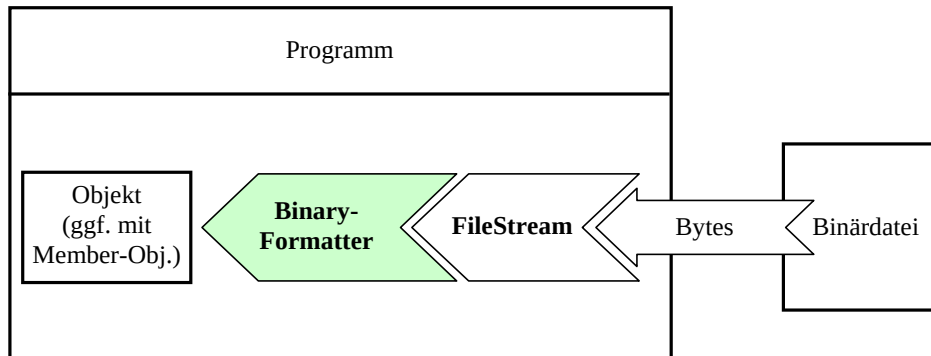
Rekonstruiert:

```
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 Objekte besitzen den Initialwert 0, während die übrigen Elementvariablen bei der (De)serialisierung ihre Werte behalten.

In der folgenden Abbildung wird die Rekonstruktion der Objekte skizziert:

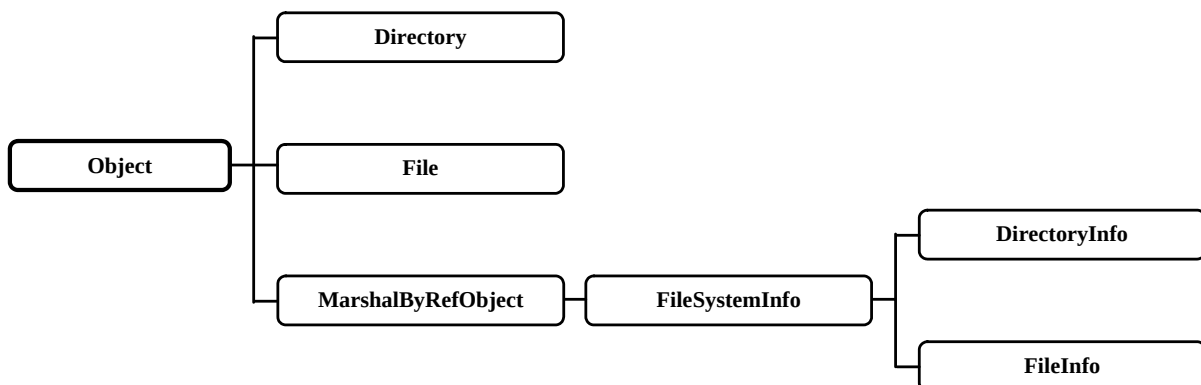


Abschließend sollen noch zwei Serialisierungs-Optionen erwähnt werden:

- An Stelle des **BinaryFormatters**, der ein kompaktes Binärformat verwendet, kann ein **SoapFormatter** eingesetzt werden, der ein XML-Format benutzt.
- Durch Implementieren der Schnittstelle **ISerializable** kann eine Klasse bei ihrer Serialisierung die Kontrolle übernehmen, muss dann aber auch einen hohen Eigenbeitrag leisten.

11.3 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.

11.3.1 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();
    }
}
```

Zunächst ein Nachtrag zur Syntax für Zeichenkettenlitterale: Mit dem Präfix „@“ vor einer Zeichenkette wird die Auswertung von Escape-Sequenzen abgeschaltet, so dass die in Windows-Pfadangaben üblichen Rückwärtsschrägstrich nicht mehr durch Verdoppeln von ihrer Sonderfunktion (vgl. Abschnitt 3.3.7.4) befreit beraubt werden müssen.

Wichtige Methoden der Klasse **File** (allesamt statisch):

- **Create()**
Die **File**-Methode **Create()** erzeugt eine Datei und ein zugehöriges **FileStream**-Objekt. Der Methodenaufruf
`File.Create(PFAD1)`
ist äquivalent mit
`new FileStream(PFAD1, FileMode.Create)`
- **CreateText()**
Die **File**-Methode **CreateText()** erzeugt eine Datei (mit **FileMode.Create**) und ein **StreamWriter**-Objekt (mit UTF8-Kodierung), wobei natürlich auch das vermittelnde **FileStream**-Objekt entsteht. Der Methodenaufruf
`File.CreateText(PFAD1)`
ist äquivalent mit
`new StreamWriter(PFAD1)`

- **Exists()**
Mit **File.Exists()** prüft man die Existenz einer Datei.
- **Delete()**
Zum Löschen einer Datei dient die **File**-Methode **Delete()**.
- **Copy()**
Bei der im Beispiel verwendeten **Copy()**-Überladung erlaubt der Wert **true** des dritten Parameters das Überschreiben einer vorhandenen Zieldatei.
- **Move()**
Zum Umbenennen oder Verschieben einer Datei verwendet man die Methode **File.Move()**.
- **SetLastWriteTime()**
Von den zahlreichen Methoden zum Modifizieren von Dateiattributen wird im 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
```

11.3.2 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**:

- **GetCurrentDirectory(), SetCurrentDirectory()**
Mit **GetCurrentDirectory()** bzw. **SetCurrentDirectory()** kann man das aktuelle Verzeichnis zum laufenden Programm ermitteln bzw. setzen.
- **Exists()**
Mit **Exists()** prüft man die Existenz eines Ordners.
- **CreateDirectory(), Delete()**
Zum Erzeugen bzw. Löschen eines Ordners stehen die Methoden **CreateDirectory()** bzw. **Delete()** bereit. Per Voreinstellung kann mit **delete()** nur ein *leerer* Ordner gelöscht werden. Über einen Parameter vom Typ **bool** lässt sich diese Bedingung aber aufheben.

Wichtige Instanzmethoden der Klasse **DirectoryInfo**:

- **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.
- **GetDirectories()**
Analog liefert die Instanzmethode **GetDirectories()** einen Array mit **DirectoryInfo**-Objekten zu den Unterordnern.

11.3.3 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' + Enter\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 Tastendrücke reagieren und dabei die Ausgabe von Zeichen auf dem Bildschirm verhindern (Wert **true** für den ersten und einzigen Parameter). Man erhält eine Instanz der Struktur **ConsoleKeyInfo**, die u.a. das zu einer Taste gehörige Unicode-Zeichen kennt.

11.4 Übungsaufgaben zu Kapitel 11

1) Erstellen Sie ein Statistikprogramm zur Berechnung des Mittelwerts, das als Eingabe eine Textdatei mit Daten akzeptiert, wobei das Semikolon als Trennzeichen dient. In folgender Beispieldatei liegen 3 Variablen (Spalten) für 5 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.");
    }
}
```

12 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 Multithread-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) 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 12.2).

Während jeder *Prozess* einen eigenen Adressraum besitzt, laufen die *Threads* eines Programms im selben Adressraum ab und haben einen gemeinsamen Heap. Allerdings benötigt jeder Thread als selbständiger Kontrollfluss bzw. Ausführungsfaden einen eigenen Stack.

Bei C# ist die Multithread-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.

Aus Zeitgründen können folgende Themen der Thread-Programmierung in diesem Manuskript *kaum* behandelt werden:

- Threadpools
Statt für eine Aufgabe bestimmten Typs (z.B. Bedienung eines Webzugriffs) jeweils einen neuen Thread zeitaufwändig zu erzeugen und anschließend wieder zu zerstören, erstellt man hier eine feste Anzahl von Arbeits-Threads. Eingehende Aufträge werden einem freien Thread zugeteilt oder in eine Warteschlange gestellt.
- Hintergrund-Threads

12.1 Threads erzeugen

Ein Thread wird in C# über ein Objekt der gleichnamigen Klasse aus dem Namensraum **System.Threading** realisiert. Jede parameterfreie Instanz- oder Klassenmethode mit dem Rückgabotyp **void** kann in einem eigenen Thread gestartet werden, indem ein zugehöriges Delegatenobjekt vom Typ **ThreadStart** erzeugt und dem **Thread**-Konstruktor übergeben wird, z.B.:

```
Thread pt = new Thread(new ThreadStart(pro.Run));
```

Diese Anweisung 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. Lieferant und Kunde 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 Transaktionen überschritten ist:

```
using System;
using System.Threading;

public class Lager {
    int bilanz;
    int anz;
    const int MANZ = 20;
    const int STARTKAP = 100;

    public bool Offen() {
        if (anz < MANZ)
            return true;
        else {
            Console.WriteLine("\nLieber " + Thread.CurrentThread.Name +
                ", es ist Feierabend!");
            return false;
        }
    }

    public void Ergaenze(int add) {
        if (!Offen()) return;
        bilanz += add;
        anz++;
        rumoren();
        Console.WriteLine("Nr. " + anz + ":\t" + Thread.CurrentThread.Name +
            " ergänzt\t" + add + "\tum " + DateTime.Now.TimeOfDay + " Uhr. Stand: " + bilanz);
    }

    public void Liefere(int sub) {
        if (!Offen()) return;
        bilanz -= sub;
        anz++;
        rumoren();
        Console.WriteLine("Nr. " + anz + ":\t" + Thread.CurrentThread.Name +
            " entnimmt\t" + sub + "\tum " + DateTime.Now.TimeOfDay + " Uhr. Stand: " + bilanz);
    }

    void rumoren() {
        double d;
        for (int i = 0; i < 40000; i++)
            d = i * i;
    }

    public Lager(int start) {
        bilanz = start;
    }
}
```

```

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(new ThreadStart(pro.Run));
    Thread kt = new Thread(new ThreadStart(kon.Run));
    pt.Name = "Produzent";
    kt.Name = "Konsument";
    pt.Start();
    kt.Start();
}
}

```

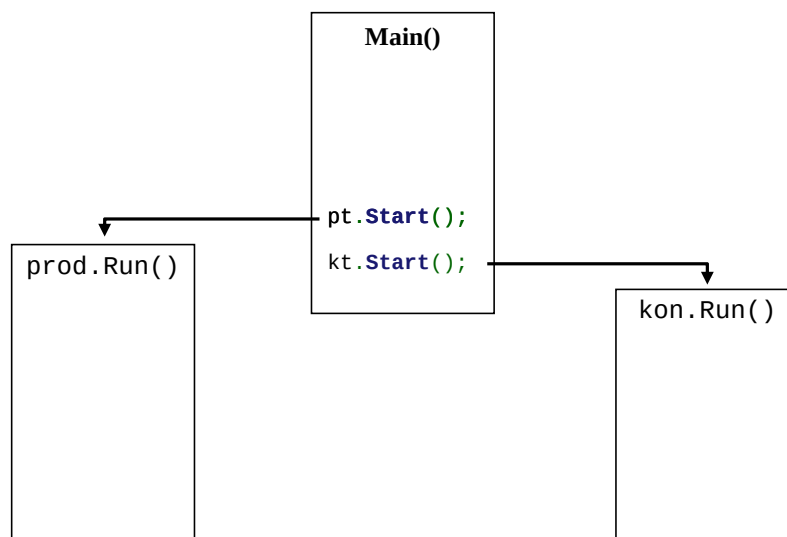
Die **Main()**-Methode der Klasse Lager erzeugt als Startmethode des Programms die beteiligten Objekte:¹

- 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).



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 wei-

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).

ter bis zum Ende ihrer jeweiligen Startmethode, und das Programm endet mit seinem letzten Vordergrund-Thread.¹

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 Offen()`
Hier ist zu erfahren, ob das Lager noch geöffnet hat.
- `public void Ergaenze(int add)`
Diese Methode wird vom Produzenten genutzt, um Ware virtuell einzuliefern.
- `public void Liefere(int sub)`
Diese Methode wird vom Konsumenten genutzt, um Ware virtuell zu beziehen.
- `void rumoren()`
Diese private Methode dient dazu, Aufwand beim Ausführen der Aufträge zu simulieren.

Produzent und Konsument kommen mit einer recht simplen Klassendefinition aus:

```
using System;
public class Produzent {
    Lager lager;
    Random rand;

    public Produzent(Lager lager_) {
        lager = lager_;
        rand = new Random((int) DateTime.Now.Ticks + 1);
    }
    public void Run() {
        while (lager.Offen()) {
            lager.Ergaenze(5 + rand.Next(100));
            System.Threading.Thread.Sleep(1000 + rand.Next(3000));
        }
    }
}

using System;
public class Konsument {
    Lager lager;
    Random rand;

    public Konsument(Lager lager_) {
        lager = lager_;
        rand = new Random((int) DateTime.Now.Ticks);
    }

    public void Run() {
        while (lager.Offen()) {
            lager.Liefere((5 + rand.Next(100)));
            System.Threading.Thread.Sleep(1000 + rand.Next(3000));
        }
    }
}
```

Neben dem Konstruktor ist jeweils nur eine Methode namens `Run()` vorhanden, die sich auf eine **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**-Metho-

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.

de **Sleep()** aufrufen. Diese befördert den Thread vom Zustand **Running** in den Zustand **WaitSleepJoin** (siehe unten)

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 zu Beginn eines **while** - Schleifendurchgangs ein geschlossenes Lager festgestellt, wenn also der Methodenaufruf **lager.Offen()** 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.

In den beiden Ausführungsfäden **pt** bzw. **kt** wird die Startmethode **Run()** von einem Objekt aus der Klasse **Produzent** bzw. **Konsument** ausgeführt. Alle von einer Startmethode via Methodenaufruf 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** im Produzenten- und im Konsumenten-Thread zum Einsatz:

- Die Methode **Offen()** entscheidet, ob weitere Veränderungen des Lagers möglich sind.
- 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 (wie der gespaltene Besen in Göthes *Zauberlehrling*), 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	39	um 02:33:12.8437500	Uhr.	Stand: 52
Nr. 2:	Konsument	entnimmt	87	um 02:33:12.8437500	Uhr.	Stand: 52
Nr. 3:	Konsument	entnimmt	94	um 02:33:14.7500000	Uhr.	Stand: -42
Nr. 4:	Produzent	ergänzt	64	um 02:33:15.6250000	Uhr.	Stand: 22
Nr. 5:	Produzent	ergänzt	69	um 02:33:16.9062500	Uhr.	Stand: 91
Nr. 6:	Konsument	entnimmt	23	um 02:33:18.3906250	Uhr.	Stand: 68
Nr. 7:	Produzent	ergänzt	5	um 02:33:20.2968750	Uhr.	Stand: 73
Nr. 8:	Produzent	ergänzt	39	um 02:33:22.1406250	Uhr.	Stand: 112
Nr. 9:	Konsument	entnimmt	61	um 02:33:22.1562500	Uhr.	Stand: 51
Nr. 10:	Produzent	ergänzt	30	um 02:33:25.1250000	Uhr.	Stand: 81
Nr. 11:	Konsument	entnimmt	27	um 02:33:25.5000000	Uhr.	Stand: 54
Nr. 12:	Konsument	entnimmt	57	um 02:33:27.3750000	Uhr.	Stand: -3
Nr. 13:	Produzent	ergänzt	53	um 02:33:27.7656250	Uhr.	Stand: 50
Nr. 14:	Produzent	ergänzt	26	um 02:33:29.4372350	Uhr.	Stand: 76
Nr. 15:	Konsument	entnimmt	84	um 02:33:30.6718750	Uhr.	Stand: -8
Nr. 16:	Produzent	ergänzt	74	um 02:33:30.7343750	Uhr.	Stand: 66

Nr. 17: Produzent ergänzt	7	um 02:33:32.9531250 Uhr. Stand: 73
Nr. 18: Konsument entnimmt	56	um 02:33:33.3593750 Uhr. Stand: 17
Nr. 19: Produzent ergänzt	92	um 02:33:34.4062500 Uhr. Stand: 109
Nr. 20: Konsument entnimmt	32	um 02:33:36.2031250 Uhr. Stand: 77

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 39 Einheiten zu einem Bestand von 52 Einheiten geführt habe, und auch die Auftragsnummer 2 ist falsch.
- Im zweiten Eintrag wird behauptet, dass die Entnahme von 87 Einheiten ohne Effekt auf den Lagerbestand geblieben sei.
- Der Bestand wird mehrfach negativ, was in einem realen Lager nicht passieren kann.

Je nach Prozessorleistung Ihres Rechners müssen Sie den virtuellen Aufwand in der Lager - Methode `rumoren()` eventuell noch etwas steigern, um die Effekte beobachten zu können.

12.2 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.

12.2.1 Die lock-Anweisung

Am Anfang des oben wiedergegebenen Ablaufprotokolls stehen zwei „wirre“ Einträge, die folgendermaßen zu erklären sind (*Race Condition*):

- Der etwas früher gestartete Produzenten-Thread „kommt als erster beim Lager an“, ruft die Lager-Methode `Ergaenze()` mit dem Parameterwert 39 auf und bringt mit den Anweisungen


```
bilanz += add;
anz++;
```

 die `bilanz` auf den Wert 139 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 87 auf und bringt mit den Anweisungen


```
bilanz -= sub;
anz++;
```

 die Lagerbilanz auf 52 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 `bilanz` und `anz` verwendet.
- Dann schreibt auch der Konsumenten-Thread seine Protokollzeile. Allerdings ist der Stand von 52 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 void Ergaenze(int add) {
    if (!Offen()) return;
    lock(this) {
        bilanz += add;
        anz++;
        Console.WriteLine("Nr. "+anz+":\t"+Thread.CurrentThread.Name+
            " ergänzt\t"+add+"\tum "+DateTime.Now.TimeOfDay+" Uhr. Stand: "+bilanz);
    }
}

```

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 Sperrobject zum Blockieren der Anwendung missbraucht werden könnte. 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.

In unserem Beispielprogramm unterbleiben nun die wirren Protokolleinträge, doch die Ausflüge in negative Lagerzustände finden nach wie vor statt:

Der Laden ist offen (Bestand: 100)

Nr. 1:	Produzent	ergänzt	32	um 00:48:35.2031250	Uhr. Stand: 132
Nr. 2:	Konsument	entnimmt	92	um 00:48:35.2031250	Uhr. Stand: 40
Nr. 3:	Konsument	entnimmt	75	um 00:48:36.5312500	Uhr. Stand: -35
Nr. 4:	Konsument	entnimmt	20	um 00:48:37.7968750	Uhr. Stand: -55
Nr. 5:	Produzent	ergänzt	27	um 00:48:38.7187500	Uhr. Stand: -28
Nr. 6:	Konsument	entnimmt	75	um 00:48:40.2812500	Uhr. Stand: -103
Nr. 7:	Produzent	ergänzt	27	um 00:48:42.3906250	Uhr. Stand: -76
Nr. 8:	Konsument	entnimmt	85	um 00:48:43.9062500	Uhr. Stand: -161
Nr. 9:	Produzent	ergänzt	104	um 00:48:45.7343750	Uhr. Stand: -57
Nr. 10:	Konsument	entnimmt	7	um 00:48:46.2968750	Uhr. Stand: -64
Nr. 11:	Produzent	ergänzt	21	um 00:48:48.0000000	Uhr. Stand: -43
Nr. 12:	Konsument	entnimmt	26	um 00:48:48.5312500	Uhr. Stand: -69
Nr. 13:	Produzent	ergänzt	73	um 00:48:49.4531250	Uhr. Stand: 4
Nr. 14:	Konsument	entnimmt	59	um 00:48:50.7031250	Uhr. Stand: -55
Nr. 15:	Produzent	ergänzt	104	um 00:48:50.9218750	Uhr. Stand: 49
Nr. 16:	Konsument	entnimmt	22	um 00:48:52.1562500	Uhr. Stand: 27
Nr. 17:	Produzent	ergänzt	91	um 00:48:54.3281250	Uhr. Stand: 118
Nr. 18:	Konsument	entnimmt	25	um 00:48:55.5937500	Uhr. Stand: 93
Nr. 19:	Produzent	ergänzt	86	um 00:48:56.8437500	Uhr. Stand: 179
Nr. 20:	Konsument	entnimmt	82	um 00:48:57.8906250	Uhr. Stand: 97

Lieber Produzent, es ist Feierabend!

Lieber Konsument, es ist Feierabend!

12.2.2 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 Multithread-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()**
Sollte innerhalb eines kritischen Blocks eine Ausnahme auftreten, wird die Sperre auf jeden Fall aufgehoben.

Durch den **finally**-Block wird 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.

12.2.3 Koordination per Wait() und Pulse()

Mit Hilfe der statischen **Monitor**-Methoden **Wait()** und **Pulse()** können in unserem 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 void Liefere(int sub) {
    if (!Offen()) return;
    lock(this) {
        while (bilanz < sub) {
            Console.WriteLine(Thread.CurrentThread.Name+
                " muss warten: Keine "+sub+" Einheiten vorhanden.");
            Monitor.Wait(this);
        }
        bilanz -= sub;
        anz++;
        Console.WriteLine("Nr. "+anz+":\t"+Thread.CurrentThread.Name+
            " entnimmt\t"+sub+"\tum "+DateTime.Now.TimeOfDay+" Uhr. Stand: "+bilanz);
    }
}
```

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 sorgt. 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 void Ergaenze(int add) {
    if (!Offen()) return;
    lock(this) {
        bilanz += add;
        anz++;
        Console.WriteLine("Nr. "+anz+":\t"+Thread.CurrentThread.Name+
            " ergänzt\t"+add+"\tum "+DateTime.Now.TimeOfDay+" Uhr. Stand: "+bilanz);
        Monitor.Pulse(this);
    }
}
```

Der reaktivierte Konsumenten-Thread bewirbt sich wieder um Prozessorzeit und Lagerzugangsbe-
rechtigung. Als Parameter wird im **Pulse()**-Aufruf das synchronisierende Objekt angegeben, das die
Warteschlange verwaltet.

Nun produziert das Beispielprogramm nur noch realistische Lagerprotokolle, z.B.:

Der Laden ist offen (Bestand: 100)

Nr. 1:	Produzent ergänzt	61	um 01:12:04.6718750	Uhr.	Stand: 161
Nr. 2:	Konsument entnimmt	92	um 01:12:04.6718750	Uhr.	Stand: 69
Konsument muss warten: Keine 75 Einheiten vorhanden.					
Nr. 3:	Produzent ergänzt	18	um 01:12:06.5312500	Uhr.	Stand: 87
Nr. 4:	Konsument entnimmt	75	um 01:12:06.5312500	Uhr.	Stand: 12
Nr. 5:	Produzent ergänzt	42	um 01:12:07.7812500	Uhr.	Stand: 54
Nr. 6:	Konsument entnimmt	20	um 01:12:07.7968750	Uhr.	Stand: 34
Konsument muss warten: Keine 75 Einheiten vorhanden.					
Nr. 7:	Produzent ergänzt	7	um 01:12:10.7187500	Uhr.	Stand: 41
Konsument muss warten: Keine 75 Einheiten vorhanden.					
Nr. 8:	Produzent ergänzt	7	um 01:12:11.8437500	Uhr.	Stand: 48
Konsument muss warten: Keine 75 Einheiten vorhanden.					
Nr. 9:	Produzent ergänzt	57	um 01:12:14.0156250	Uhr.	Stand: 105
Nr. 10:	Konsument entnimmt	75	um 01:12:14.0156250	Uhr.	Stand: 30
Konsument muss warten: Keine 85 Einheiten vorhanden.					
Nr. 11:	Produzent ergänzt	100	um 01:12:17.9062500	Uhr.	Stand: 130
Nr. 12:	Konsument entnimmt	85	um 01:12:17.9062500	Uhr.	Stand: 45
Nr. 13:	Konsument entnimmt	7	um 01:12:20.2812500	Uhr.	Stand: 38
Nr. 14:	Produzent ergänzt	55	um 01:12:20.3437500	Uhr.	Stand: 93
Nr. 15:	Produzent ergänzt	94	um 01:12:22.3750000	Uhr.	Stand: 187
Nr. 16:	Konsument entnimmt	26	um 01:12:22.5312500	Uhr.	Stand: 161
Nr. 17:	Produzent ergänzt	65	um 01:12:23.4062500	Uhr.	Stand: 226
Nr. 18:	Konsument entnimmt	59	um 01:12:24.7031250	Uhr.	Stand: 167
Nr. 19:	Produzent ergänzt	75	um 01:12:25.5625000	Uhr.	Stand: 242
Nr. 20:	Konsument entnimmt	22	um 01:12:26.1406250	Uhr.	Stand: 220

Lieber Produzent, es ist Feierabend!

Lieber Konsument, es ist Feierabend!

An Stelle der Methode **Pulse()**, die den Thread mit dem ältesten **Wait()**-Aufruf anspricht, ist oft die
Methode **PulseAll()** sinnvoller, die alle wartenden Threads weckt.

Die Monitor-Methoden **Wait()**, **Pulse()** und **PulseAll()** dürfen nur in einem synchronisierten Block
aufgerufen werden.

12.3 Threads abbrechen

Ein Thread endet „auf natürlichem Weise“ mit der zugrunde liegenden Startmethode. Er kann aber
auch schon vorher durch die **Thread-Methode Abort()** gestoppt werden. Diese löst eine
ThreadAbortException aus, so dass der betroffene Thread per Ausnahmebehandlung für einen
sinnvollen Abgang sorgen kann.

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 void Liefere(int sub) {
    if (!Offen()) return;
    lock(this) {
        anz++;
        if (bilanz < sub) {
            Console.WriteLine("Nr. "+anz+":\t"+Thread.CurrentThread.Name+
                " fordert\t"+sub+"\tum "+DateTime.Now.TimeOfDay+
                " Uhr.\n          und wird abgewickelt.");
            Thread.CurrentThread.Abort();
        }
        bilanz -= sub;
        Console.WriteLine("Nr. "+anz+":\t"+Thread.CurrentThread.Name+" entnimmt\t"+
            sub+"\tum "+DateTime.Now.TimeOfDay+" Uhr. Stand: "+bilanz);
    }
}
```

Der Konsument nutzt die Ausnahmereaktion für eine Beschwerde:

```
public void Run() {
    Random rand = new Random(4711);
    try {
        while (pl.Offen()) {
            pl.Liefere((5 + rand.Next(100)));
            Thread.Sleep(1000 + rand.Next(3000));
        }
    } catch (ThreadAbortException) {
        Console.WriteLine("Als Kunde muss ich mir so etwas "+
            "nicht gefallen lassen!");
    }
}
```

In der folgenden Sequenz muss der Konsument frühzeitig aussteigen:

Der Laden ist offen (Bestand: 100)

Nr. 1:	Produzent ergänzt	71	um 01:26:05.8906250 Uhr.	Stand: 171
Nr. 2:	Konsument entnimmt	92	um 01:26:05.8906250 Uhr.	Stand: 79
Nr. 3:	Konsument entnimmt	75	um 01:26:07.2343750 Uhr.	Stand: 4
Nr. 4:	Konsument fordert	20	um 01:26:08.5000000 Uhr.	
	und wird abgewickelt.			

Als Kunde muss ich mir so etwas nicht gefallen lassen!

Nr. 5:	Produzent ergänzt	15	um 01:26:08.7031250 Uhr.	Stand: 19
Nr. 6:	Produzent ergänzt	80	um 01:26:10.6875000 Uhr.	Stand: 99
Nr. 7:	Produzent ergänzt	88	um 01:26:11.9531250 Uhr.	Stand: 187
Nr. 8:	Produzent ergänzt	91	um 01:26:14.4375000 Uhr.	Stand: 278
Nr. 9:	Produzent ergänzt	29	um 01:26:17.8906250 Uhr.	Stand: 307
Nr. 10:	Produzent ergänzt	89	um 01:26:21.2500000 Uhr.	Stand: 396
Nr. 11:	Produzent ergänzt	19	um 01:26:22.5468750 Uhr.	Stand: 415
Nr. 12:	Produzent ergänzt	97	um 01:26:24.1093750 Uhr.	Stand: 512
Nr. 13:	Produzent ergänzt	77	um 01:26:25.7812500 Uhr.	Stand: 589
Nr. 14:	Produzent ergänzt	97	um 01:26:28.2500000 Uhr.	Stand: 686
Nr. 15:	Produzent ergänzt	61	um 01:26:31.1718750 Uhr.	Stand: 747
Nr. 16:	Produzent ergänzt	91	um 01:26:34.6718750 Uhr.	Stand: 838
Nr. 17:	Produzent ergänzt	72	um 01:26:38.1562500 Uhr.	Stand: 910
Nr. 18:	Produzent ergänzt	84	um 01:26:40.0625000 Uhr.	Stand: 994
Nr. 19:	Produzent ergänzt	100	um 01:26:41.9218750 Uhr.	Stand: 1094
Nr. 20:	Produzent ergänzt	55	um 01:26:45.2031250 Uhr.	Stand: 1149

Lieber Produzent, es ist Feierabend!

Statt es bei einem mürrischen Protest zu belassen, kann der Konsument in seiner **ThreadAbortException**-Ausnahmereaktion über die **Thread**-Methode **ResetAbort()** das Beenden seines Threads durchaus verhindern, z.B.:

```

public void Run() {
    Random rand = new Random(4711);
    while (pl.Offen()) {
        try {
            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));
    }
}

```

12.4 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.

12.4.1 Scheduling und Prioritäten

Den Bestandteil der CLR, der die verfügbare Rechenzeit auf den verfügbaren Prozessoren auf die arbeitswilligen Threads (im Zustand **Started**) verteilt, bezeichnet man als **Scheduler**.

Er orientiert sich u.a. an den **Prioritäten** der Threads, die per **Priority**-Eigenschaft festgelegt wird. Erlaubt sind die folgenden Werte des Enumerationstyps **ThreadPriority**:

- **Highest**
- **AboveNormal**
- **Normal**
- **BelowNormal**
- **Lowest**

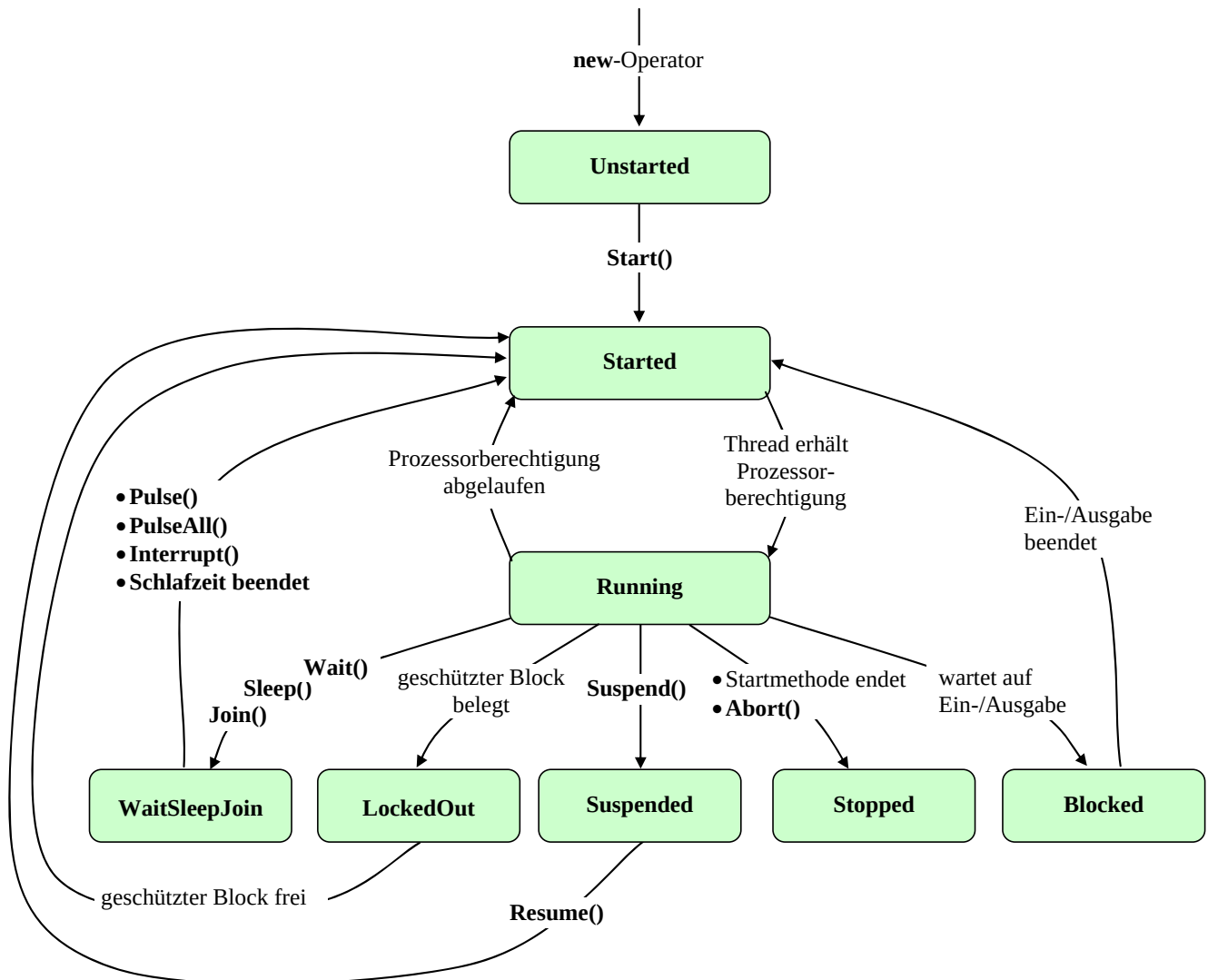
Per Voreinstellung haben Threads die Priorität **Normal**. Über die **Thread** - Eigenschaft **Priority** lässt sich die Priorität ermitteln und verändern.

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 (in den Zustand **Running** gelangen), wenn *kein* Thread mit höherer Priorität arbeitswillig (im Zustand **Started**) ist. Ein *Verhungern* (engl. *starvation*) von Threads mit niedriger Priorität, die permanent den Kürzere ziehen, wird also *nicht* verhindert.
- Die Threads gleicher Priorität werden reihum (*Round-Robin*) jeweils für eine festgelegte Zeitspanne ausgeführt.
- Ist die Zeitscheibe eines Threads verbraucht, wird er vom Scheduler in den Zustand **Started** versetzt, und der Nachfolger erhält Zugang zum Prozessor.

12.4.2 Zustände von Threads

In der folgenden Abbildung werden wichtige Thread-Zustände (bezeichnet durch Werte der Enumeration **ThreadState**) und Anlässe für Zustandsübergänge dargestellt:



Einige in der Abbildung enthaltene Thread-Instanzmethoden wurden bisher noch nicht angesprochen:

- **Join()**
Der aufrufende Thread wartet bis der angesprochene Thread beendet ist.
- **Interrupt()**
Der angesprochene Thread wird per **ThreadInterruptedException** geweckt, wenn er sich (jetzt oder später) im **WaitSleepJoin**-Zustand befindet. Begibt sich der Thread nie in diesen Zustand, bleibt der **Interrupt()**-Aufruf ohne Folgen.
- **Suspend(), Resume()**
Mit diesem als *veraltet* (engl.: *deprecated*) eingestuft Methoden kann man einen Thread anhalten bzw. wieder starten. Die Methoden sind in Misskredit geraten, weil ihr Einsatz zu Deadlock-Situationen (siehe Abschnitt 12.5) führen kann.

12.5 Deadlocks

Wer sich beim Einsatz von exklusiven Blöcken zur Thread-Synchronisation ungeschickt anstellt, kann so genannte *Deadlocks* 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;

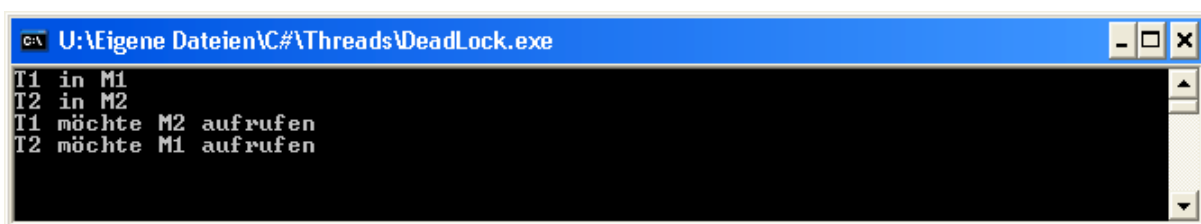
class DeadLock {
    static object lock1 = new object();
    static object lock2 = new object();

    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 Schläpfchen 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:



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.
- Thread 2 möchte Block A betreten, wartet also darauf, dass Thread 1 diesen Block verlässt.

12.6 Timer

Mit Hilfe der Klasse **Timer** im Namensraum **System.Threading** kann man die CLR beauftragen, eine Methode regelmäßig in einem eigenen Thread auszuführen. Im diesem Beispiel

```
tim = new System.Threading.Timer(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)
- **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.
- **int** *dueTime*
Die Zeit bis zum ersten Aufruf in Millisekunden
- **int** *period*
Die Zeit zwischen zwei Aufrufen in Millisekunden

Die CLR erzeugt nicht bei jedem Aufruf der Methode einen neuen Thread, sondern arbeitet Ressourcen schonend mit einem **ThreadPool**. Ein fälliger Methodenaufruf wird von einem freien Thread aus dem Pool übernommen, bei großem Andrang nach einer Wartezeit. Nach Beendigung des Aufrufs macht der beteiligte Thread wieder Bereitschaftsdienst.

Wir stellen uns die Aufgabe, in einer GUI-Anwendung eine umfangreiche Hintergrundaktivität regelmäßig per ThreadPool im Hintergrund erledigen zu lassen, wobei die Benutzeroberfläche verzögerungsfrei bedienbar bleiben soll. Im folgenden Programm berechnet die Methode Hintergrund-Aktion 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 zzg;

    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);
        zzg = new Random();

        tim = new System.Threading.Timer(HintergrundAktion, null, 0, 1000);

        TextBox eingabe = new TextBox();
        eingabe.Left = 20; eingabe.Top = 50; eingabe.Width = 125;
        Controls.Add(eingabe);
    }
}
```

```

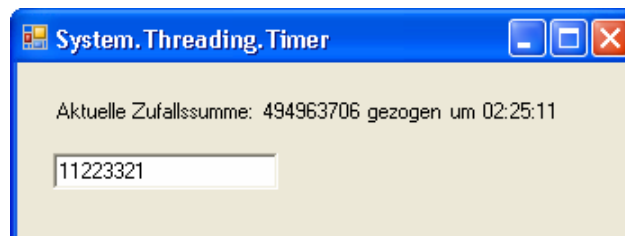
void HintergrundAktion(object info) {
    long l = 0;
    for (int i = 0; i < 10000000; i++)
        l += zsg.Next(100);
    BeginInvoke(new LabelUpdateDelegate(LabelUpdater), l.ToString());
}

void LabelUpdater(string s) {
    anzeige.Text = "Aktuelle Zufallssumme: "+s+" berechnet um "+
        DateTime.Now.ToLongTimeString();
}

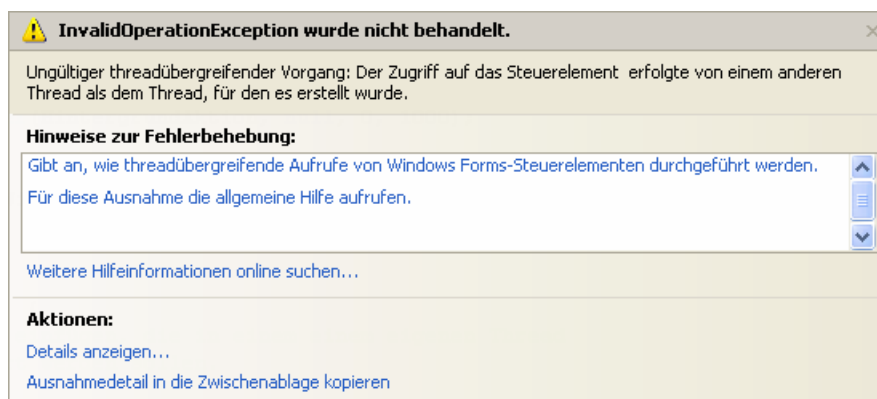
static void Main() {
    Application.Run(new ThreadingTimer());
}
}

```

Trotz des erheblichen CPU-Zeitverbrauchs durch die Intergrundaktivität bleibt reagiert die Benutzeroberfläche des Programms verzögerungsfrei:



Die von einem Hintergrund-Thread ausgeführte Methode muss ein Resultat in die **Text**-Eigenschaft eines **Label**-Steuerelements übertragen. Dies stellt ein Problem dar, weil Zugriffe auf Steuerelemente aus anderen Threads per Voreinstellung vom Compiler verhindert werden, z.B.:



Im .NET-Framework sind die meisten Eigenschaften und Methoden der Steuerelemente bewusst *nicht* Thread - sicher, um Leistungsverluste durch die Thread-Synchronisation zu vermeiden. Folglich sollten Zugriffe auf Steuerelemente dem Thread vorbehalten bleiben, der sie erzeugt hat. 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 Delegationstyp (im Beispiel: `LabelUpdaterDelegate`),
- erzeugt ein Delegatenobjekt (`new LabelUpdateDelegate(LabelUpdater)`)
- und veranlasst den GUI-Thread über die **BeginInvoke()**-Methode des Formulars, das Delegatenobjekt auszuführen.

Im Namensraum **System.Windows.Forms** ist ebenfalls eine Klasse namens **Timer** vorhanden; diese arbeitet jedoch ereignisorientiert und ist sehr bequem in eine GUI-Anwendung einzubinden ist, 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();
    }

    static void Main() {
        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. Offenbar eignet sich die Klasse **Timer** im Namensraum **System.Windows.Forms** nur für Methoden mit geringem Rechenzeitbedarf.

12.7 Übungsaufgaben zu Kapitel 12

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

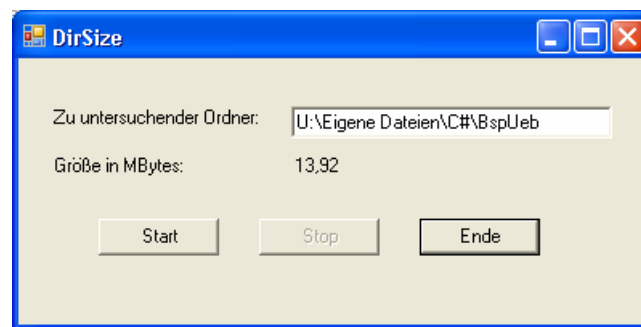
```

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.:



13 Netzwerkprogrammierung

Die FCL (Framework Class Library) 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 weiterreichenden Kontrolle wählen kann.

Die serverseitige Webprogrammierung mit dem ASP.NET – Framework 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 ASP.NET – Programm, wobei der Zustand einer Interaktion z.B. in der Webseite gespeichert wird.

- **Webdienste**

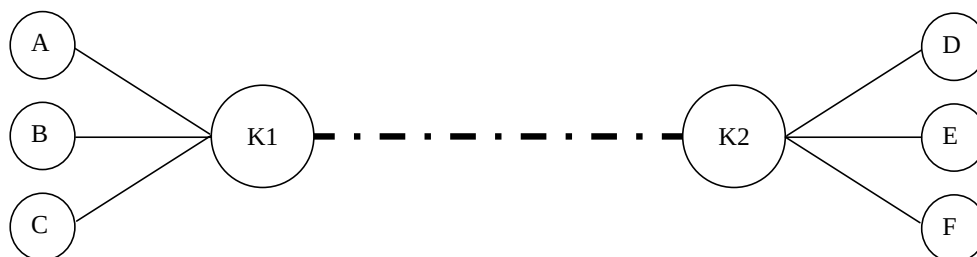
Mit dieser Technik wird der Aufruf von Prozeduren auf entfernten Rechnern (*Remote Procedure Call*) via Internet unterstützt¹. Dabei kommt das XML-basierte **SOAP**-Protokoll (*Simple Object Access Protocol*) zum Einsatz. So kann z.B. eine lokale Anwendung vom Server einer Fluglinie Daten über Verbindungen, freie Plätze und Preise beschaffen.

13.1 Wichtige Konzepte der Netzwerktechnologie

Als **Netzwerk** bezeichnet man eine Anzahl von Systemen (z.B. Rechnern), die über ein **gemeinsames Medium** (z.B. Ethernetkabel, WLAN, Infrarotkanal) verbunden sind und über ein **gemeinsames Protokoll** 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. Versand einer E-Mail über einen SMTP-Server) bis zur physikalischen Ebene (z.B. elektromagnetische Wellen auf einem Ethernetkabel) sind zahlreiche Übersetzungen vorzunehmen bzw. Aufgaben zu lösen, jeweils unter Beachtung der zugehörigen Regeln.

¹ Server und Klienten für Webdienste müssen allerdings nicht als .NET – Anwendungen realisiert werden.

- **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 besteht eine Adresse aus 32 Bits, üblicherweise durch 4 per Punkt getrennte Dezimalzahlen (aus dem Bereich von 0 bis 255) dargestellt, z.B.:

136.199.8.62

4. Transportschicht (Übertragung von Datenströmen, 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 (Kommunikation zwischen Anwendungen, z.B. via TCP)

Auf dieser Ebene sind Regeln angesiedelt, die den Datenaustausch zwischen zwei Anwendungen (meist auf verschiedenen Rechnern) ermöglichen. Auch solche Aufgaben werden vom TCP abgedeckt, wobei die so genannten **Ports** eine zentrale Rolle spielen. Hierbei handelt es sich um Zahlen zwischen 0 und 65535, die auf einem beteiligten Rechner eine Server- oder Klienten-Anwendung identifizieren. So wird es z.B. möglich, auf einem Server 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 in der Regel 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-Nummer des Serverrechners und die Portnummer des Dienstes
- Die IP-Nummer 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 Kooperation 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).

Schließlich ist noch das **Internet Control Message Protocol (ICMP)** zu erwähnen, 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.

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

Hier wird für verschiedene Dienste festgelegt, wie Anforderungen zu formulieren sind. Einem SMTP-Server, der an Port 25 lauert, kann man z.B. folgendermaßen eine Mail übergeben:

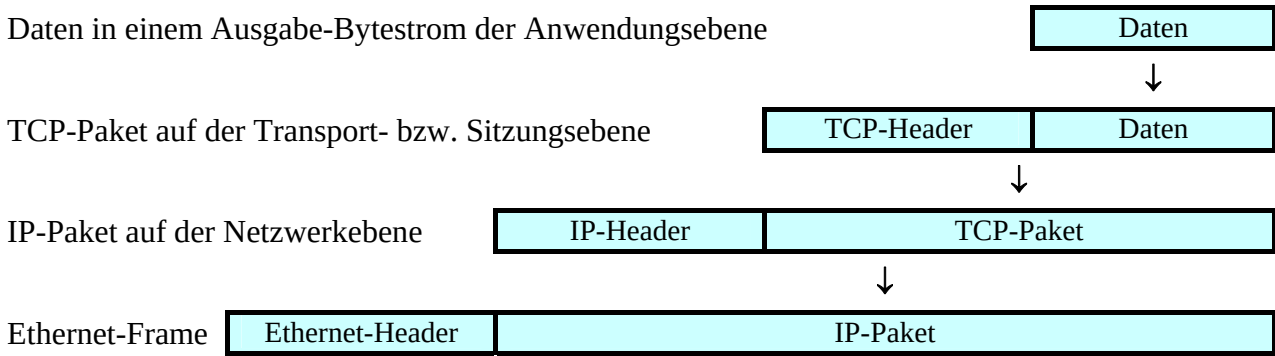
Klient	Serverantwort
telnet srv.srv-dom.de 25 HELO mainpc.client-dom.de MAIL FROM: otto@client-dom.de RCPT TO: empf@srv-dom.de DATA From: egal@weg.te To: ziel@soso.te Subject: Thema Dies ist der Inhalt! . QUIT	220 srv.srv-dom.de ESMTP Postfix 250 srv.srv-dom.de 250 Ok 250 Ok 354 End data with <CR><LF>.<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!

13.1.2 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, Telnet) 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:



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 0 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.

13.1.3 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 13 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.

13.2 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:	<i>Proto-</i>	<i>://</i>	<i>User:Pass@</i>	<i>Rechner</i>	<i>:Port</i>	<i>Pfad</i>	<i>?Abfrage</i>
	<i>koll</i>		(optional)		(optional)		(optional)
Beispiel:	http	://		www.egal.de	:81	/cgi-bin/beispiel/cgi.pl	?vorname=Kurt

13.2.1 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* oder *https* (sicheres HTTP), dann liefert **Create()** ein Objekt der Klasse **HttpWebRequest**.

Ist eine protokollspezifische Konfigurationen der Anforderung erforderlich, kann man nach einer expliziten Typumwandlung die entsprechenden Eigenschaften der zugehörigen **WebRequest** – Unterklasse ansprechen. Aufgrund der folgenden **UserAgent** – Manipulation stellt sich ein C# – Programm beim Webserver als Firefox – Browser vor:

```
((HttpWebRequest)request).UserAgent =  
    "Mozilla/5.0 (Windows; U; Windows NT 5.1; de; rv:1.8.1.1) "+  
    "Gecko/20061204 Firefox/2.0.0.7";
```

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);
```

Den eigentlichen Inhalt erhält man vom **WebResponse** – Objekt über die Methode **GetResponseStream()** als **Stream**-Objekt, 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;  
}
```

Mit der **StreamReader** – Methode **ReadLine()** kommt man an die Zeilen der Serverantwort heran:

```
String s;
while ((s = reader.ReadLine()) != null) {
    Console.WriteLine(s);
    Console.ReadLine();
}
```

Nach dem Lesen der Serverantwort kann 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() {
        WebRequest request = WebRequest.Create("http://www.uni-trier.de/");
        ((HttpWebRequest)request).UserAgent =
            "Mozilla/5.0 (Windows; U; Windows NT 5.1; de; rv:1.8.1.1) "+
            "Gecko/20061204 Firefox/2.0.0.1";
        WebResponse response = request.GetResponse();
        Console.WriteLine("Letzte Änderung:\t"+
            ((HttpWebResponse)response).LastModified);
        Console.WriteLine("Zeichensatz: \t" +
            ((HttpWebResponse)response).CharacterSet);
        Stream content = response.GetResponseStream();
        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;
        }
        String s;
        while ((s = reader.ReadLine()) != null) {
            Console.WriteLine(s);
            Console.ReadLine();
        }
    }
}
```

Ein Programmlauf (am 8.2.2007) bringt folgendes Ergebnis:

```
Letzte Änderung:      30.01.2007 14:45:08
Zeichensatz:         ISO-8859-1

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta content="text/html; charset=ISO-8859-1" http-equiv="Content-Type">
  <meta content="Offizielle Homepage der Universit&auml;t Trier" name="description">
  . . .
  . . .
```

Seit .NET 2.0 findet sich im Namensraum **System.Windows.Forms** die Steuerelementklasse **Web-Browser** mit der Fähigkeit, HTML zu rendern. Sie sollen als Übungsaufgabe einen kleinen Web-browser erstellen, der HTML-Seiten attraktiver anzeigen kann, z.B.:



13.2.2 Datei-Download per HTTP - Protokoll

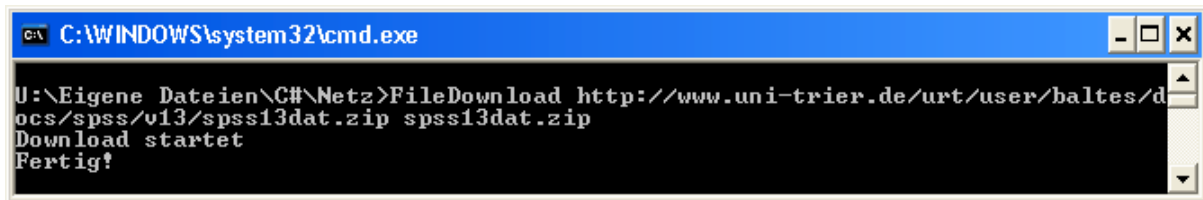
Per Request/Response – Technologie und HTTP - Protokoll kann man nicht nur HTML-Seiten von einem Server beziehen, sondern auch Binärdateien herunterladen. Wir nutzen zu diesem Zweck die Klasse **WebClient** kennen, welche durch Verpacken der ohnehin schon recht bequemen Klassen **WebRequest** und **WebResponse** 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. Mit Hilfe der **HttpWebRequest** - Eigenschaft **IfModifiedSince** kann man z.B. den Zugriff auf eine Web-Ressource vom Datum der letzten Änderung abhängig machen. Das folgende Download-Programm wertet drei Kommandozeilenargumente aus:

- Webadresse der Quelldatei
- lokaler Dateiname
- akzeptables Datum der letzten Änderung

Liegt das Änderungsdatum einer Datei weiter zurück, wirft die **HttpWebRequest** - Methode **GetResponse()** eine **WebException**¹:

```
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>");
            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;
        }
        WebRequest request = WebRequest.Create(args[0]);
        ((HttpWebRequest)request).IfModifiedSince = date;
        try {
            response = request.GetResponse();
        } catch (WebException) {
            Console.WriteLine("Keine aktuelle Version vorhanden");
            return;
        }
    }
}
```

¹ Es ist eher ungewöhnlich und nicht zur Nachahmung empfohlen, eine gewöhnliche Programmablaufsteuerung per Ausnahmebehandlung vorzunehmen.

```

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);
}
}
}

```

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.

13.2.3 Dynamische erstellte Webseiten per GET oder POST anfordern

13.2.3.1 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. PHP, Perl, Java oder auch ASP.NET). 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. Allerdings gewinnen allmählich Lösungen an Popularität, die stärker mit dem Webserver verzahnt sind, permanent im Speicher verbleiben und so eine bessere Leistung bieten (z.B. PHP als Apache - Modul, ASP.NET als IIS - Erweiterung, Java-Container wie Tomcat). 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).

Die Palette der möglichen CGI - Anwendungen ist außerordentlich vielseitig (z.B. Datenbankabfragen, Suchmaschinen), so dass die Nutzung solcher Ressourcen durch klientenseitige C# - Programme attraktiv ist. Wie zu Beginn von Abschnitt 13 verabredet, werden wir uns mit der *Produktion* von CGI – Lösungen in ASP.NET nicht beschäftigen.

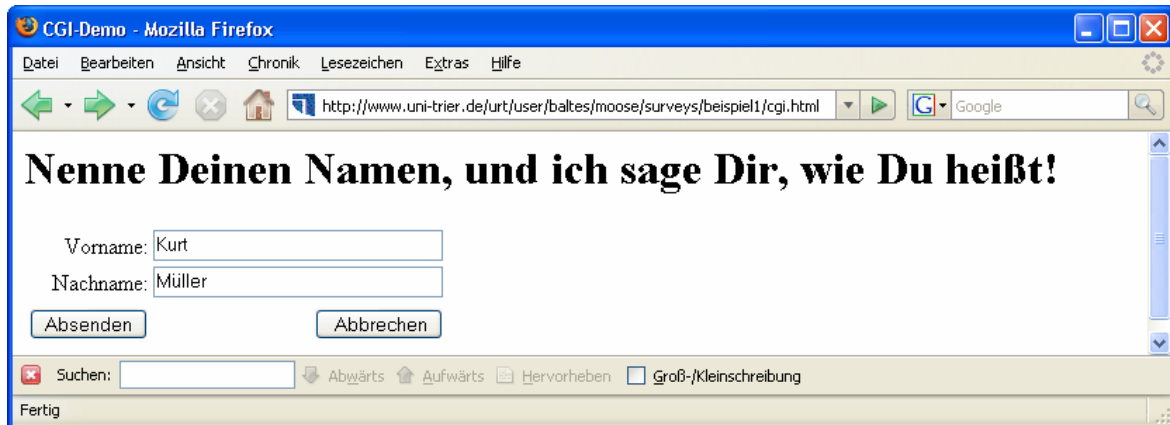
Es besteht eine Verwandtschaft zu den oben erwähnten *Webdiensten*, die allerdings keine HTML-Seiten für Browser produzieren, sondern XML - Dateien und die zur Kommunikation das **SOAP**-Protokoll benutzen. Von dieser interessanten, aber noch nicht sehr verbreiteten, Technik kann in diesem Kurs weder die server- noch die klientenseitige Programmierung behandelt werden.

13.2.3.2 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. Hinter diesem Fenster



steckt der folgende HTML-Code mit Formular:

```
<html>
<head>
<title>CGI-Demo</title>
</head>
<h1>Nenne Deinen Namen, und ich sage Dir, wie Du heißt!</h1>
<form method="get" action="/cgi-bin/moose/surveys/beispiel1/cgi.pl">
<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>
<td align="right"><input type="submit" value=" Absenden "> </td>
<td align="right"><input type="reset" value=" Abbrechen"> </td>
</tr>
</table>
</form>
</html>
```

Klickt der Benutzer auf den Schalter **Absenden**, werden die Formularfelder mit der Syntax

Name=Wert

als Parameter für die CGI - Software zum WWW-Server übertragen. Zwei Felder werden jeweils durch ein &-Zeichen getrennt, so dass im obigen Beispiel mit den Feldern vorname und nachname (siehe HTML-Quelltext) folgende Sendung resultiert:

vorname=Kurt&nachname=M%FCller

Den Umlaut „ü“ kodiert der Browser automatisch durch ein einleitendes Prozentzeichen und seinen Hexadezimalwert im Zeichensatz, um die URL-Syntaxregeln einzuhalten. 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.

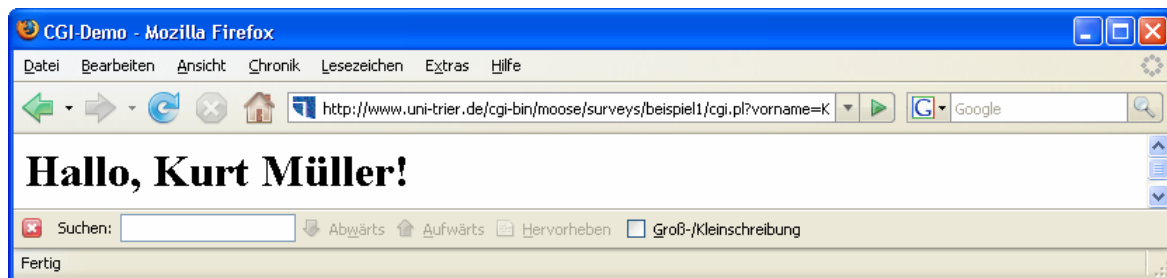
Auf gleich noch näher zu erläuternde Weise übergibt der WWW-Server die Formulardaten an das im **action**-Attribut der Formulardefinition angegebene externe Programm oder Skript. Im Beispiel handelt es sich um folgendes Perl-Skript, das wenig kreativ aus den übergebenen Namen einen Gruß formuliert:

```
#!/perl
use CGI;
my $query = CGI->new();
print "Content-type: text/html\n\n";
print "<html>\n<head><title>CGI-Demo</title></head>\n<body>\n<h1>";
print 'Hallo, ' . $query->param("vorname") . ' ' . $query->param("nachname") . '!';
print "</h1>\n</body>\n</html>";
```

Es schreibt Header-Felder und den eigentlichen Inhalt per **print**-Kommando an die Standardausgabe, in unserem Fall:

- Es wird nur das minimal erforderliche Header-Feld mit der Typspezifikation geschrieben.
- Der Inhaltsbereich besteht aus einer HTML-Seite.

Vom WWW-Server wird die Perl-Produktion über das HTTP-Protokoll an den Browser geschickt, z.B.:



13.2.3.3 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, werden die Parameter hinter einem trennenden Fragezeichen an den URI des CGI-Aufrufs angehängt und erscheinen nach Eintreffen der Antwortseite in der Adresszeile eines Browsers (siehe obiges Beispiel). Der WWW-Server schreibt die CGI-Parameter in eine Umgebungsvariable namens **QUERY_STRING** und stellt auf analoge Weise der anschließend gestarteten 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 Perl-Skript ist der Zugriff auf die Umgebungsvariable **QUERY_STRING** nicht so ganz offensichtlich, weil das Perl-Modul **CGI.pm** verwendet (siehe z.B. Patwardhan et al. 2002, S.

376ff) und ein CGI-Objekt durch Aufruf seiner **param()**-Methode mit dem Lesen der Umgebungsvariablen beauftragt wird, z.B.:

```
$query->param("vorname")
```

Dasselbe Perl-Objekt beherrscht auch den Empfang vom **POST**-Parametern (siehe unten), so dass zur Demonstration der beiden Parameter-Transfermethoden nur *ein* Perl-Skript benötigt wird.

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 GetDemo {
    static void Main() {
        const String CGIURI=
            "http://www.uni-trier.de/cgi-bin/moose/surveys/beispiel1/cgi.pl";

        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 = client.DownloadData(CGIURI+"?vorname="+
            vorName+"&nachname="+nachName);

        String s = HttpUtility.UrlDecode(antwort, Encoding.Default);
        Console.WriteLine("\nAntwort : \n{0}", s);
    }
}
```

Mit der statischen **HttpUtility**-Methode **UrlEncode()** wird für die korrekte URL-Kodierung der Umlaute etc. gesorgt (siehe Abschnitt 13.2.3.2)¹. 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 13.2.1).

Weil das Programm die vom CGI-Skript gelieferte HTML-Seite nur als Text darstellt, ist sein Auftritt nicht berauschend:

¹ Wegen der Klasse **HttpUtility** aus dem Namensraum **System.Web** benötigt der Compiler einen Verweis auf das GAC-Assembly **System.Web**.

```
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>
```

13.2.3.4 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 übertragen¹. Was genau gemäß HTTP-Protokoll zu tun ist, braucht C# - Programmierer(innen) kaum zu interessieren, weil die Klasse **WebClient** in ihrer Methode **UploadValues()** die Arbeit komplett erledigt:

```
using System;
using System.Collections.Specialized;
using System.Net;
using System.Text;
using System.Web;

class CgiPost {
    static void Main() {
        const string CGIURI =
            "http://www.uni-trier.de/cgi-bin/moose/surveys/beispiel1/cgi.pl";
        NameValueCollection formData = new NameValueCollection();

        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);

        formData.Add("vorname", vorName);
        formData.Add("nachname", nachName);

        Console.WriteLine("\nAnforderung wird übertragen ...");
        WebClient client = new WebClient();

        byte[] antwort = client.UploadValues(CGIURI, formData);

        String s = HttpUtility.UrlDecode(antwort, Encoding.Default);
        Console.WriteLine("\nAntwort : \n{0}", s);
    }
}
```

Weil dasselbe (für *beide* Parameterübergabetechniken geeignete) Perl-Skript angesprochen wird wie im **GET**-Beispiel, erhält man natürlich dieselbe Ausgabe.

¹ In obigem Perl-Skript ist das Lesen der Parameter aus STDIN durch die Kapselung in der **param()**-Methode eines CGI-Objekts nicht mehr unmittelbar zu erkennen.

13.3 Internet-Adressen 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 **DNS (Domain Name System)** geleistet wird.

Im .NET – Framework liefert die statische Methode **GetHostEntry()** der Klasse **Dns** die IP-Adresse(n) zu einem Host-Namen bzw. den Namen zu einer Adresse. Man erhält ein **IPHostEntry**-Objekt, das in seinen Eigenschaften **AddressList** bzw. **HostName** die gewünschten Daten bereit hält. Im folgenden Programm werden beide Konvertierungsrichtungen verwendet:



Aufgrund der graphischen Oberfläche ist der Quelltext deutlich länger als bei den bisherigen Beispielprogrammen in Abschnitt 13. Daher werden nur die Ereignisbehandlungsmethoden zu den beiden Schaltflächen wiedergegeben:

```
void CbToNumberClick(object sender, System.EventArgs e) {
    try {
        IPHostEntry 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 {
        IPHostEntry host = Dns.GetHostEntry(tbNumber.Text);
        tbName.Text = host.HostName;
    } catch (Exception ex) {
        MessageBox.Show(ex.ToString(), "Fehler");
    }
}
```

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 Programm zeigt nur die erste IP-Adresse an:

```
tbNumber.Text = host.AddressList[0].ToString();
```

Den vollständigen Quellcode finden Sie im Ordner

...\BspUeb\Netzwerk\DNS

13.4 Socket-Programmierung

Unsere bisherigen Beispielprogramme im Abschnitt 13 haben hauptsächlich WWW-Inhalte von Servern bezogen und dazu FCL-Klassen benutzt, die ein fest „verdrahtetes“ Anwendungsprotokoll (meist HTTP) realisieren. Im aktuellen Abschnitt bleiben wir zwar auf der Anwendungsebene, ge-

winnen aber 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 13.1.1) angesiedelte TCP-Protokoll schafft zwischen zwei (durch *Portnummern* identifizierten) Anwendungen, die sich meist auf verschiedenen (durch *IP-Nummern* 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.

13.4.1 TCP-Server

Wir erstellen einen Server, der am TCP-Port 13 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;
const int SVRPORT = 13000;
IPAddress ip = Dns.GetHostEntry("Bernhard").AddressList[0];

server = new TcpListener(ip, SVRPORT);
```

Da viele Rechner über mehrere IP-Adressen verfügen, hält der Compiler nichts vom Konstruktoraufruf *ohne* IP-Adresse:

```
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 er 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, ansprechbar 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 und das **Socket**-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:

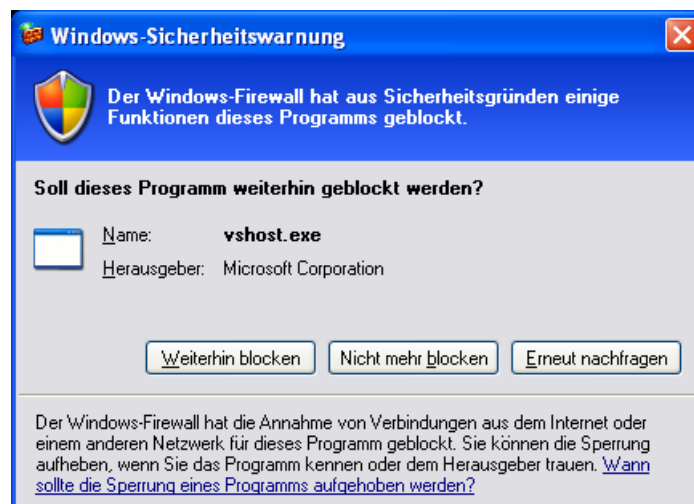
```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class TcpListenerDemo {
    static void Main() {
        TcpListener server = null;
        Socket socke = null;
        const int SVRPORT = 13000;
        IPAddress ip = Dns.GetHostEntry("bernhard").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) {
                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();
        }
    }
}
```

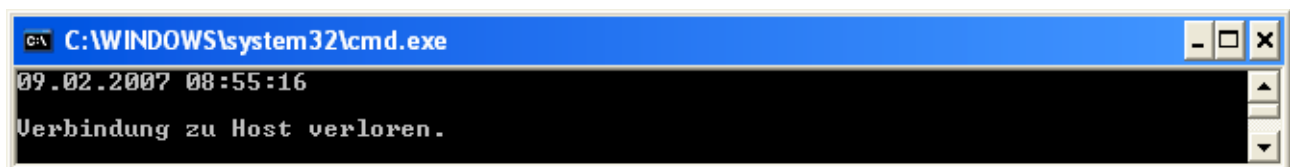
Der Kürze halber verzichten wir auf ein Verfahren zum regulären Beenden des Programms, so dass es rabiät unterbrochen werden muss.

Sobald nicht die interne IP-Adresse 127.0.0.1 (verbunden mit dem Host-Namen *localhost*) verwendet wird, ist unter Windows XP mit SP 2 für den Betrieb des Zeitservers eine Firewall-Ausnahme erforderlich:

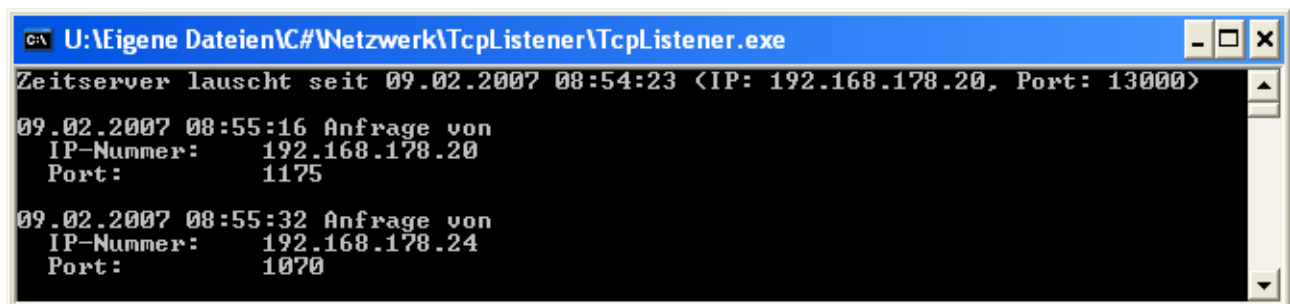


Der Zeitserver kann auch per Telnet-Klient angesprochen werden, z.B. mit dem folgenden Aufruf:
telnet bernhard 13000

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
```

13.4.2 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.:

```
const string SVR = "localhost";
const 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 geschützten **try**-Block.

Der Klient bietet die empfangenen Daten über ein Objekt der Klasse **NetworkStream** an:

```

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:

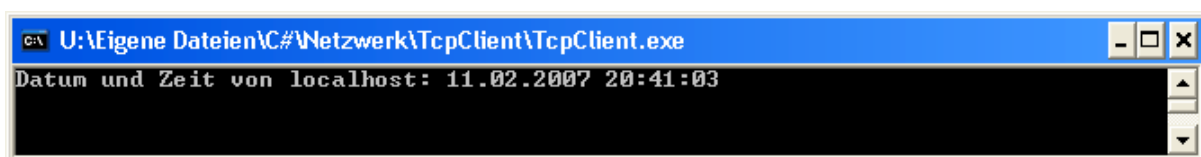
```

using System;
using System.Net.Sockets;
using System.Text;

class TcpClientDemo {
    static void Main() {
        const string SVR = "localhost";
        const 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.:



13.4.3 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:

```
public void Run() {
    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 ab

```
stream.ReadTimeout = 10000;
```

und der Kontakt wird beendet:

```
void Close() {
    // Socket in TcpClient auffordern, vor dem Schließen
    // alle Daten zu empfangen bzw. zu senden
    tcpClient.Client.Shutdown(SocketShutdown.Both);
    // Ressourcen frei geben
    tcpClient.Close();
    server.CheckOut(clientID);
}
```

Das Dienstleistungszentrum wird durch ein Objekt der Klasse `MultiThreadEchoServer` realisiert. Sobald der Lauscher vom Typ **TcpListener** eine Klientenanfrage feststellt, wird ein `EchoHandler` - Objekt erzeugt, das im Konstruktoraufbau folgende Informationen erhält:

- eine Referenz auf das Dienstleistungszentrum (das `MultiThreadEchoServer` - Objekt)
So kann der `EchoHandler` die `MultiThreadEchoServer` - 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 in einem neuen Thread gestartet:

```
EchoHandler echoHandler = new EchoHandler(this, tcpClient, nr);
Thread echoThread = new Thread(new ThreadStart(echoHandler.Run));
echoThread.IsBackground = true;
echoThread.Start();
```

Damit ein solcher Thread ein geplantes Programmende nicht verhindern kann, wird er in den Hintergrund verbannt.

In der folgenden Situation bedient der `MultiThreadEchoServer` drei Klienten:

The screenshot shows three overlapping command prompt windows. The top window, titled 'U:\Eigene Dateien\C#\Netzwerk\MultiClientEchoServer\MultiClientEchoServer.exe', displays the server's startup and client connections:


```

12.02.2007 23:50:14 MultiClientEchoServer gestartet
12.02.2007 23:50:31 Klient 1 akzeptiert. Aktiv: 1
IP-Nummer: 127.0.0.1
Port: 2393
12.02.2007 23:50:33 Klient 2 akzeptiert. Aktiv: 2
IP-Nummer: 127.0.0.1
Port: 2394
12.02.2007 23:50:33 Klient 3 akzeptiert. Aktiv: 3
IP-Nummer: 127.0.0.1
Port: 2395
  
```

 The middle window, titled 'U:\Eigene Dateien\C#\Netzwerk\MultiClientEchoServer\DauerSender.exe', shows a list of 83 data transmissions:


```

Sendung 61: 7651313285
Sendung 62: 7435076540
Sendung 63: 0533511834
Sendung 64: 0031192715
Sendung 65: 6916373582
Sendung 66: 1294903604
Sendung 67: 2439121871
Sendung 68: 5379201667
Sendung 69: 7628025083
Sendung 70: 6276726158
Sendung 71: 5586785028
Sendung 72: 9792451498
Sendung 73: 6184435119
Sendung 74: 4906247512
Sendung 75: 7571382511
Sendung 76: 0126337414
Sendung 77: 3393396300
Sendung 78: 7636014606
Sendung 79: 0009936439
Sendung 80: 7337845474
Sendung 81: 1941448561
Sendung 82: 9045333930
Sendung 83: 2291242586
Sendung 84: 6086405854
Sendung 85: 4171079735
Sendung 86: 0507199389
Sendung 87: 2279591813
Sendung 88: 8449514705
  
```

 The bottom window, also titled 'U:\Eigene Dateien\C#\Netzwerk\MultiClientEchoServer\DauerSender.exe', shows a continuation of the data transmissions, starting from 'Sendung 60' and ending with 'Sendung 83'.

Den vollständigen Quellcode finden Sie im Ordner

...BspUeb\Netzwerk\MultiClientEchoServer\MultiThreadEchoServer

Statt für jeden Klienten einen neuen Thread zeitaufwändig zu erzeugen und anschließend wieder zu zerstören, kann man den vom .NET - Laufzeitsystem für jede Anwendung verwalteten Threadpool verwenden. Hier befinden sich etliche Threads in Bereitschaft und übernehmen Aufträge, die in einer Warteschlange eintreffen.

In unserer Anwendung wird der Quellcode sogar vereinfacht, wenn wir auf eigene Threads verzichten und stattdessen Arbeitsaufträge in die Warteschlange des Threadpools einreihen:

```
ThreadPool.QueueUserWorkItem(new WaitCallback(echoHandler.Run));
```

Im Aufruf der statischen Methode **QueueUserWorkItem()** aus der Klasse **ThreadPool** ist als Parameter ein Objekt vom Delegationstyp **WaitCallback** anzugeben, das auf die im Thread auszuführende Methode zeigt.

13.5 Übungsaufgaben zu Abschnitt 13

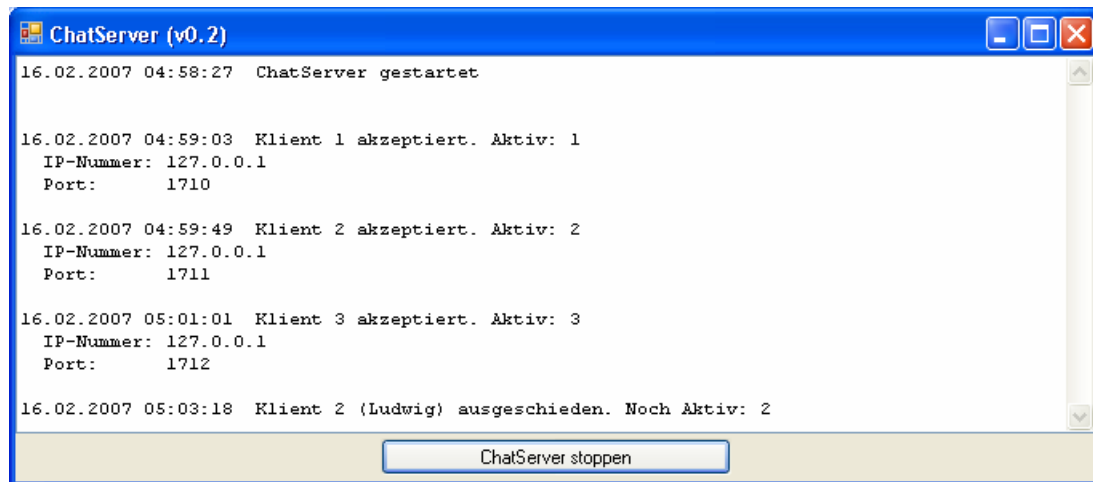
1) Erstellen Sie mit Hilfe der Steuerelementklasse **WebBrowser** eine WinForms-Anwendung, die eine **TextBox** zur Eingabe einer Webadresse bietet, HTML anzeigt, sinnvoll auf Änderungen der Formulargröße reagiert und elementare Navigationsmöglichkeiten bietet (siehe Bildschirmphoto in Abschnitt 13.2.1).

2) Erstellen Sie einen TCP-Klienten, der den in Abschnitt 13.4.3 vorgestellten Multithread-Echoserver durch das permanente Absenden von jeweils zehn zufällig gewählten Ziffern beschäftigt.

3) Erstellen Sie eine ChatRoom - Anwendung mit Server- und Klientenprogramm. Die Serveranwendung sollte ...

- an einem TCP-Port auf Verbindungswünsche warten
- mehrere Klienten simultan bedienen, z.B. 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 (eine Schaltfläche) anbieten zum Beenden des Programms, wobei die aktiven Klienten über das Dienstende zu benachrichtigen sind

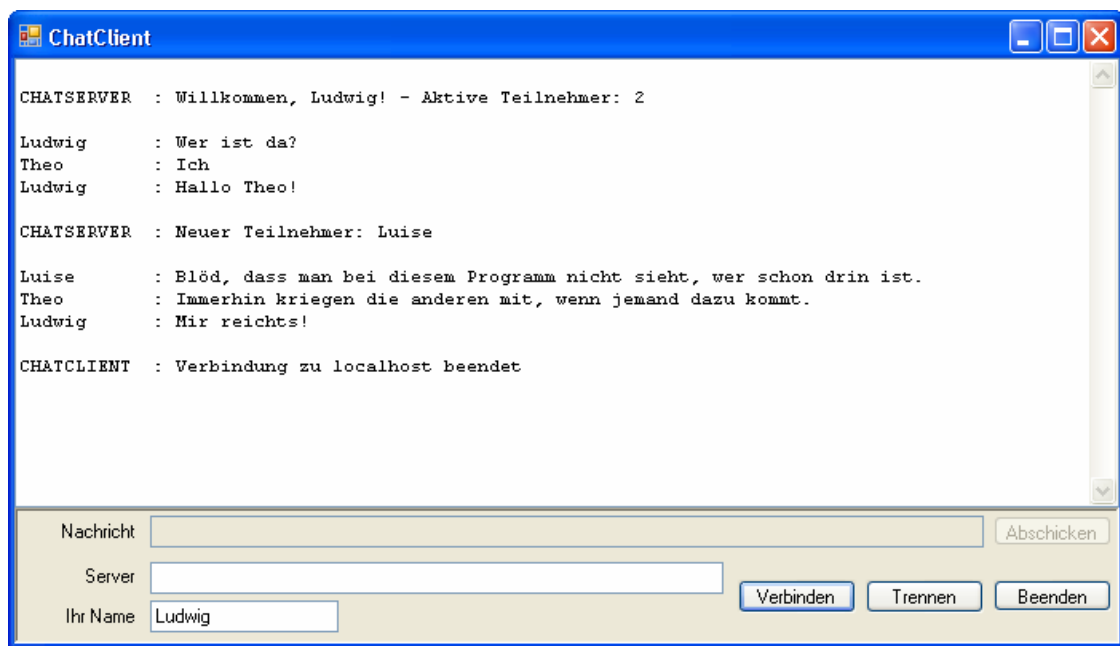
Ein typisches Server-Protokoll könnte so aussehen:



Die Klientenanwendung sollte ...

- **TextBox**-Steuerelemente anbieten für:
 - den Servernamen
 - den Wunschnamen des Chat-Teilnehmers
 - die Nachrichten
- **Button**-Steuerelemente bieten ...
 - zum An- und Abmelden beim Server
 - zum Abschicken der Nachrichten
 - 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 seinen 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 **Focus()** 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.

14 Graphikausgabe 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. Mehr Gestaltungsfreiheit benötigt z.B. auch ein Entwickler, der eigene Steuerelemente erstellen möchte. Windows stellt den Anwendungsprogrammen seit jeher mit dem **Graphics Device Interface** (GDI) eine umfangreiche Sammlung von graphischen Ausgaberroutinen zur Verfügung. Mit dieser Softwareschicht wird die 2D-Grafik - Programmierung von den spezifischen Eigenschaften bestimmter Ausgabegeräte (z.B. Bildschirm, 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.

14.1 Die Klasse Graphics

Bei der Graphikausgabe 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 sich in der Regel auf einem von **Windows.Forms.Control** abgeleiteten Fenster-Objekt befindet,
- beherrscht diverse Methoden zur Ausgabe von Grafiken (z.B. **DrawEllipse()**, **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:

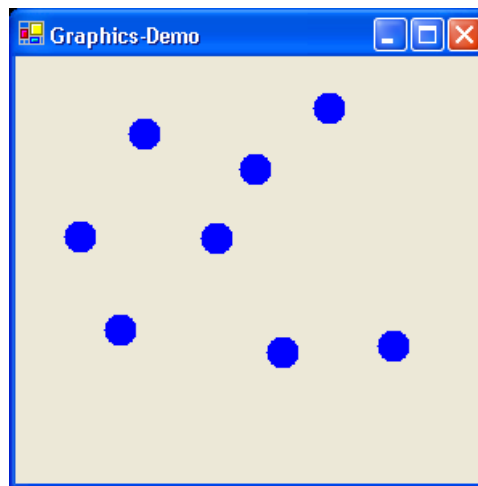
```
using System;
using System.Windows.Forms;
using System.Drawing;

class GraphicsDemo : Form {
    public 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();
    }

    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()**) werden regelmäßig Hilfsobjekte zur Beschreibung der gewünschten Ausführung übergeben (z.B. Pinsel und Stifte, siehe unten).

14.2 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 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 von obsolet gewordenen (nicht mehr referenzierten) **Graphics**-Objekten rechtzeitig frei gegeben werden.

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**)
- Zeichensätze (**Font**)

Bei den über statische Eigenschaften der Klassen **Brushes** und **Pens** verfügbaren Objekten (z.B. `Brushes.Blue`) ist ein **Dispose()**-Aufruf hingegen überflüssig und nachteilig (vgl. Bayer 2006, S. 874).

Im weiteren Verlauf von Abschnitt 14 wird bei trivialen Beispielprogrammen der Einfachheit halber auf die konsequente Schonung von GDI-Ressourcen verzichtet.

14.3 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.

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 in obigem Beispiel 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:

- Eine Methode schreiben und als **PaintEventHandler** für das **Paint**-Ereignis registrieren
Wir haben uns in Abschnitt 8.3 ausführlich mit dem Erstellen und Registrieren von Ereignisbehandlungsmethoden beschäftigt.
- **OnPaint()** überschreiben
Das Überschreiben der geerbten Methode **OnPaint()** kann natürlich nur in einer eigenen Klassendefinition stattfinden. Da in unseren GUI-Beispielprogrammen 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. Wird ein Fenster allerdings über den Bildschirmrand hinaus geschoben, geht sein Inhalt teilweise verloren und muss ggf. per **WM_PAINT** - Ereignisbehandlungsmethode wiederhergestellt werden.

14.3.1 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;

class PEHDemo : Form {
    ArrayList punkte = new ArrayList();

    public 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));
}

static void Main() {
    Application.Run(new PEHDemo());
}

```

Im Vergleich zu der Punktverlustversion besitzt das neue Programm ein **ArrayList**-Objekt, 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 14.2).

14.3.2 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;

class OnPaintDemo : Form {
    ArrayList punkte = new ArrayList();

    public OnPaintDemo() {
        Text = "OnPaint-Demo";
        MouseDown += new MouseEventHandler(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));
    }

    static void Main() {
        Application.Run(new OnPaintDemo());
    }
}

```

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 14.3.4).

14.3.3 Fensteraktualisierung per Programm anfordern

Bei allen bisherigen Varianten des Pünktchen-Programms wurde (auch) in einem **MouseEvent-Handler**, 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 per **Invalidate()**-Aufruf die sofortige 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. Alternative **Invalidate()**-Überladungen erlauben es, über **Rectangle**-Instanzen (siehe Abschnitt 14.4.3) bzw. **Region**-Objekte (siehe Abschnitt 14.10.3) rechteckige oder anders geformte *Teile* eines Fensters zu verwerfen. Wenn die entsprechend gestaltete Renovierungsmethode (**PaintEventHandler** oder **OnPaint()**-Überschreibung) auf die beschädigte Stelle des Fensters beschränkt, lässt sich oft ein Flackern vermeiden.

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();
}
```

14.3.4 Ereignisse und On-Methoden

Neben **Paint** verfügen auch alle weiteren im Namensraum **System.Windows.Forms** definierten Ereignisse über zugehörige Behandlungsmethoden, wobei die Bezeichnung aus dem Ereignisnamen und dem Präfix **On** besteht, z.B.

- Methode **OnMouseDown()** der Klasse **Control**
- Methode **OnClick()** der Klasse **Button**

Bei früheren Beschreibungen der ereignisorientierten Kommunikation (siehe z.B. Abschnitt 8.4.1.3) haben wir diese On-Methoden der Übersichtlichkeit halber weggelassen.

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 Klasse **Form** regelmäßig beerben. In der folgenden Variante des Beispielprogramms aus Abschnitt 14.3 wird auch das **MouseDown**-Ereignis des Formulars durch eine überschriebene **On**-Methode behandelt:

```
using System;
using System.Windows.Forms;
using System.Drawing;
using System.Collections;

class OnMethodenDemo : Form {
    ArrayList punkte = new ArrayList();

    public OnMethodenDemo() {
        Text = "OnMethoden-Demo";
    }

    protected override void OnPaint(PaintEventArgs e) {
        foreach (Point p in punkte)
            e.Graphics.FillEllipse(Brushes.Blue, p.X, p.Y, 20, 20);
    }

    protected override void OnMouseDown(MouseEventArgs e) {
        Graphics g = this.CreateGraphics();
        g.FillEllipse(Brushes.Blue, e.X - 10, e.Y - 10, 20, 20);
        punkte.Add(new Point(e.X - 10, e.Y - 10));
        g.Dispose();
    }

    static void Main() {
        Application.Run(new OnMethodenDemo());
    }
}
```

In der SDK-Dokumentation findet sich zu Recht die Mahnung, in einer überschreibenden **On**-Methode die zugehörige Basisklassenmethode aufzurufen, z.B.:

```
protected override void OnPaint(PaintEventArgs e) {
    base.OnPaint(e);
    foreach (Point p in punkte)
        e.Graphics.FillEllipse(Brushes.Blue, p.X, p.Y, 20, 20);
}
```

Bei **OnPaint()** wird als Begründung die Rolle dieser Methoden beim Aufruf von registrierten Ereignisbehandlungsmethoden genannt, z.B.:

Hinweise zur Vererbung: Wenn Sie **OnPaint** in einer abgeleiteten Klasse überschreiben, müssen Sie unbedingt die **OnPaint**-Methode der Basisklasse aufrufen, so dass registrierte Delegaten das Ereignis empfangen.

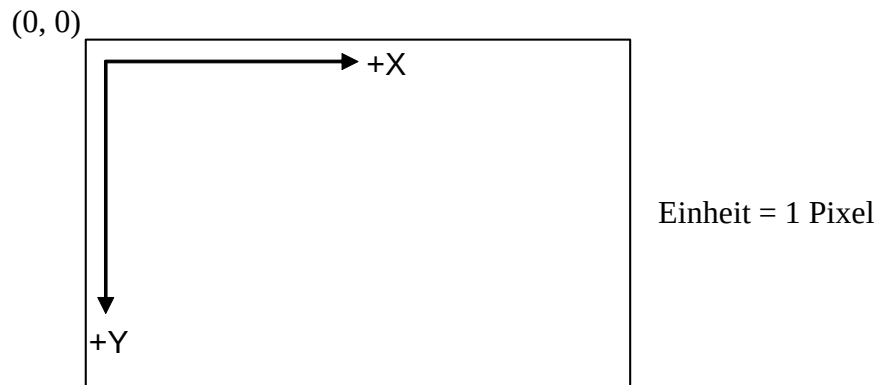
Da unsere Klasse **OnMethodenDemo** keine Behandlungsmethoden beim **Paint**- oder beim **MouseDown**-Ereignis registriert, ist in der **OnPaint()** - bzw. **OnMouseDown()** – Überschreibung *kein* Aufruf der jeweiligen Basisklassenmethode erforderlich. Man darf die SDK-Mahnung aber nur dann ignorieren, wenn ganz sicher kein Delegat beim betroffenen Ereignis registriert ist, weder aktuell noch bei einer möglichen späteren Verwendung der eigenen (abgeleiteten) Klasse.

14.4 Positionen und Größen

In diesem Abschnitt werden wichtige Typen zur einfachen Verwaltung von Positionen und Größen vorgestellt.

14.4.1 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 14.8 werden transformierte Koordinatensysteme behandelt, wobei man ...

- den Ursprung verschieben
- und/oder die Maßeinheiten für die Achsen verändern kann.

14.4.2 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(); // Initialisierung mit (0, 0)!
```

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. Ihre Instanzen verfügen über die Eigenschaften **Height** und **Width** vom Typ **Int32** (mit Lese- und Schreibzugriff).

Viele Klassen der FCL 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;

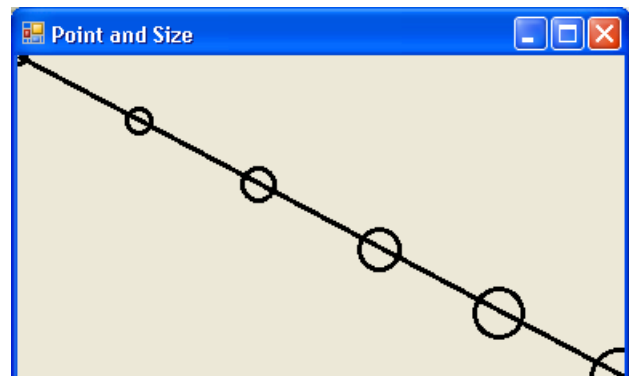
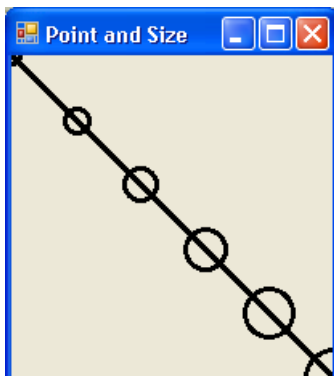
class PointAndSize : Form {
    Pen pen = new Pen(Color.Black, 3);

    public 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;
        int distX = ClientSize.Width / 5;
        int distY = ClientSize.Height / 5;
        g.DrawLine(pen, pkt1, pkt2);
        for (int i = 0; i < 6; i++) {
            int d = 10 + 5 * i;
            g.DrawEllipse(pen, pkt1.X-d/2, pkt1.Y-d/2, d, d);
            pkt1.Offset(distX, distY);
        }
    }

    static void Main() {
        Application.Run(new PointAndSize());
    }
}
```

Weil die **ResizeRedraw**-Eigenschaft des Formulars den Wert **true** erhalten hat, wird **OnPaint()** auch beim Verändern der Fenstergröße ausgeführt und der Punkt **pkt2** bleibt folglich in der unteren rechten Fensterecke:



Später werden wir auch mit 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.

14.4.3 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);

    public 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();
    }

    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;
```

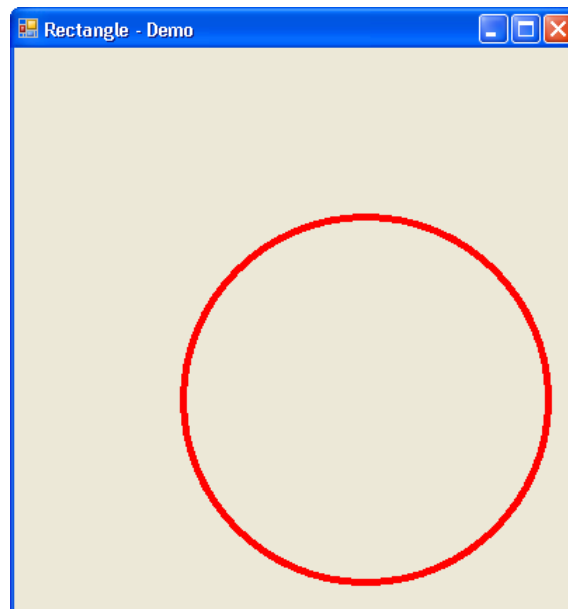
Mit der *rechten* Maustaste können Benutzer den Kreis durch Klick in die linke Fensterhälfte verkleinern bzw. durch Klick in die rechte Fensterhälfte vergrößern. Dazu wird das Begrenzungsrechteck per **Inflate()**-Methode aufgefordert, sich bei konstantem Mittelpunkt zu vergrößern bzw. zu verkleinern. Der Aufruf

```
ra.Inflate(20, 20)
```

wirkt wie:

```
ra.X -= 20;
ra.Y -= 20;
ra.Width += 20;
ra.Height += 20;
```

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**
Fenster-relative X-Koordinate des rechten Randes (= **Left** + **Width**)

Weitere Eigenschaften und Methoden der **Rectangle**-Struktur sind in der SDK-Dokumentation zu finden.

Die später benötigte Struktur **RectangleF** steht zu **Rectangle** in derselben Beziehung wie **Point** bzw. **Size** zu **PointF** bzw. **SizeF**.

14.5 Farben und Zeichenwerkzeuge

Um per **Graphics**-Objekt auf eine Fensterfläche zu zeichnen, benötigt man Farben und Werkzeuge (Stifte, Pinsel).

14.5.1 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.

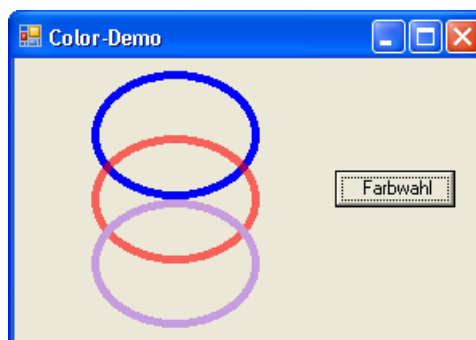
`Color.Red`

Die Liste der Farbnamen ist übrigens relativ kompatibel mit den von Webbrowsern beim Rendern von HTML-Seiten unterstützten Bezeichnungen.

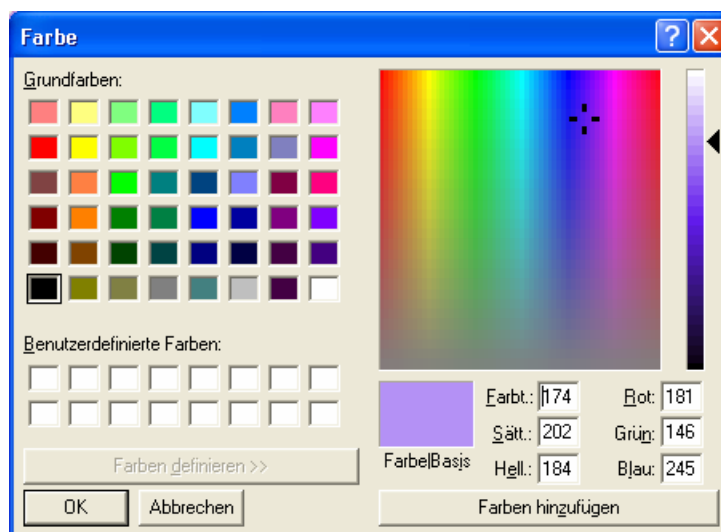
Eigene Farben mixt man mit der statischen **Color**-Methode **FromArgb()** zusammen. In der folgenden **OnPaint()**-Überschreibung wird über eine Ellipse in sattem Blau eine zweite in relativ transparentem Rot gemalt:

```
protected override void OnPaint(PaintEventArgs e) {
    e.Graphics.DrawEllipse(new Pen(Color.Blue, 5), 50, 10, 100, 75);
    Color rot150 = Color.FromArgb(150, 255, 0, 0);
    e.Graphics.DrawEllipse(new Pen(rot150, 5), 50, 50, 100, 75);
    if (puc)
        e.Graphics.DrawEllipse(new Pen(uc, 5), 50, 90, 100, 75);
}
```

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 ButtonOnClick(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).

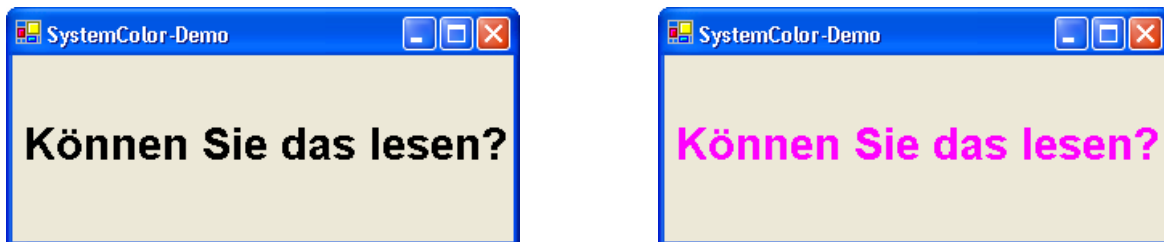
Die bisher im aktuellen Abschnitt vorgestellten Programmsegmente stammen aus dem Projekt:

...\BspUeb\GDI+\Color

Durch Verwendung der **Windows-Systemfarben**, die über statische Eigenschaften der Klasse **SystemColors** ansprechbar sind, kann man sicherstellen, dass ein Programm unabhängig vom Windows-Design, das ein Benutzer im Dialogfeld **Eigenschaften von Anzeige** gewählt hat, bedienbar bleibt. In der folgenden **OnPaint()**-Methode wird zum Schreiben die aktuelle Systemfarbe für Fenstertext verwendet:

```
protected override void OnPaint(PaintEventArgs e) {
    e.Graphics.DrawString("Können Sie das lesen?",
        new Font("Arial", 20, FontStyle.Bold),
        new SolidBrush(SystemColors.WindowText), 3, 40);
}
```

Das Ergebnis hängt von der betroffenen Anzeige-Einstellung des Benutzers ab, z.B.:



Wie gleich im Abschnitt 14.5.3 zu erfahren ist, kann der zum Malen (von Text etc.) in der Systemfarbe **WindowText** benötigte Pinsel noch einfacher gewählt werden:

```
e.Graphics.DrawString("Können Sie das lesen?",
    new Font("Arial", 20, FontStyle.Bold),
    SystemBrushes.WindowText, 3, 40);
```

14.5.2 Pen

Beim Zeichnen von Linien oder Kurven (z.B. mit der **Graphics**-Methode **DrawLine()**) ist ein Objekt der Klasse **Pen** (dt.: *Stift*) als Werkzeug anzugeben. Beim Herstellen dieses Werkzeugs kann man über diverse Linienattribute entscheiden. Im folgenden **Pen**-Konstruktor werden Farbe und Liniestärke festgelegt:

```
Pen stift = new Pen(Color.Black, 10);
```

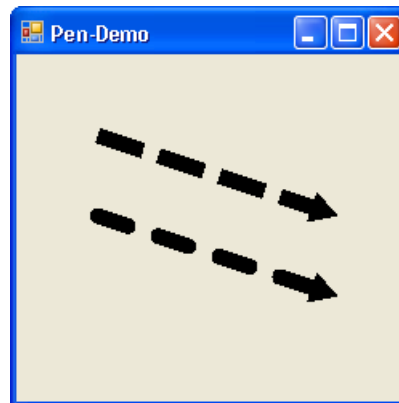
Die zahlreichen Eigenschaften der **Pen**-Klasse bieten weitere Gestaltungsmöglichkeiten, z.B.:

- **DashStyle** (Linienstil), **StartCap** (Stil für den Linienanfang), **EndCap** (Stil für das Linienende) und **DashCap** (Stil für die Segmentgrenzen in einer gestichelten Linie)

Die folgende **OnPaint()**-Überschreibung:

```
protected override void OnPaint(PaintEventArgs e) {
    Pen stift = new Pen(Color.Black, 10);
    stift.EndCap = LineCap.ArrowAnchor;
    stift.DashStyle = DashStyle.Dash;
    e.Graphics.DrawLine(stift, 50, 50, 200, 100);
    stift.DashCap = DashCap.Round;
    stift.StartCap = LineCap.Round;
    e.Graphics.DrawLine(stift, 50, 100, 200, 150);
}
```

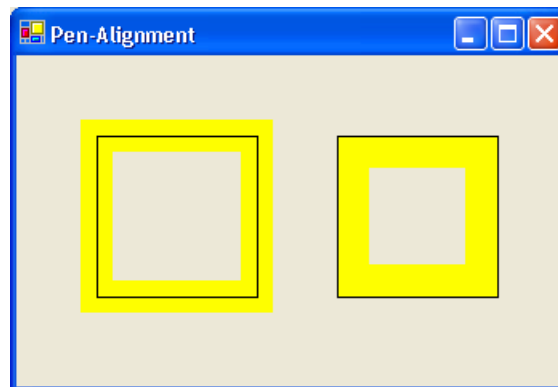
zeichnet einen gestrichelten Pfeil zunächst mit flachen und dann mit runden Begrenzungen der Segmente:



Die zur Wahl der Stile benutzten Enumerationen **LineCap**, **DashStyle** und **DashCap** befinden sich im Namensraum **System.Drawing.Drawing2D**.

- **Alignment**

Bei geschlossenen Kurven oder Vielecken legt diese Eigenschaft das Verhalten eines **Pen**-Objekts mit Breite größer 1 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 Fenster des folgenden Beispielsprogramms:



Bei der alternativen Einstellung **PenAlignment.Inset** landet die gesamte Linienbreite im Innenraum (siehe rechtes Quadrat).

Eine vollständige Liste mit den Eigenschaften und sonstigen Mitgliedern der **Pen**-Klasse findet sich in der .NET-SDK – Dokumentation.

Die Klasse **Pens** bietet über statische Eigenschaften für jede *Standardfarbe* einen zugehörigen Stift mit der Breite 1, z.B.:

```
e.Graphics.DrawLine(Pens.Red, 50, 50, 200, 100);
```

Die **SystemPens** bietet über statische Eigenschaften für jede *Windows-Systemfarbe* einen zugehörigen Stift mit der Breite 1, z.B.:

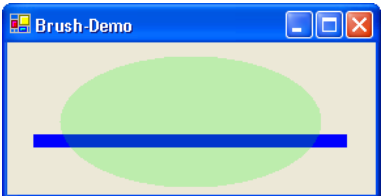
```
e.Graphics.DrawLine(SystemPens.ControlDark, 50, 50, 200, 100);
```

14.5.3 Brush

Um das *Innere* von Rechtecken, Ellipsen, Buchstaben etc. zu füllen, verwendet man ein **Brush**-Objekt (dt.: einen *Pinzel*). Bei **Brush** handelt es sich um die abstrakte Basisklasse von diversen Konkretisierungen für spezielle optische Effekte, z.B.:

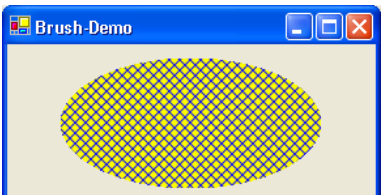
- **SolidBrush** (Namensraum **System.Drawing**)

Diese **Brush**-Ableitung ist für einfarbige Füllungen zuständig, z.B.:

Quellcodefragment	Ausgabe
<pre>gc.DrawLine(new Pen(Color.Blue, 10), 20, 75, 260, 75); gc.FillEllipse(new SolidBrush(Color.FromArgb(50, 0, 255, 0)), ra);</pre>	

- **HatchBrush** (Namensraum **System.Drawing.Drawing2D**)

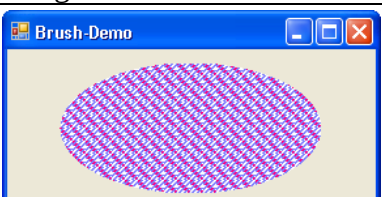
Diese **Brush**-Ableitung ist zuständig für schraffierte Füllungen, die durch ein vektoriell beschriebenes Muster, eine Vordergrundfarbe und eine Hintergrundfarbe definiert sind, z.B.:

Quellcodefragment	Ausgabe
<pre>gc.FillEllipse(new HatchBrush(HatchStyle.DiagonalCross, Color.Blue, Color.Yellow), ra);</pre>	

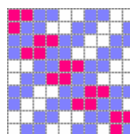
Über die **HatchStyle**-Eigenschaft lassen sich diverse Werte aus der gleichnamigen Enumeration als Muster einstellen (siehe Übungsaufgabe in Abschnitt 14.11).

- **TextureBrush** (Namensraum **System.Drawing**)

Diese **Brush**-Ableitung füllt das Innere einer Form mit einer Bitmap-Graphik, wobei das Bild nötigenfalls bis zur Füllung der Fläche wiederholt ausgegeben wird (auf mehreren Kacheln). Für das erste Beispiel:

Quellcodefragment	Ausgabe
<pre>gc.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-Graphik so groß, dass es *nicht* zu einer Mehrfachdarstellung kommt:

Quellcodefragment	Ausgabe
<pre>gc.FillEllipse(new TextureBrush(new Bitmap("land.bmp")), ra);</pre>	

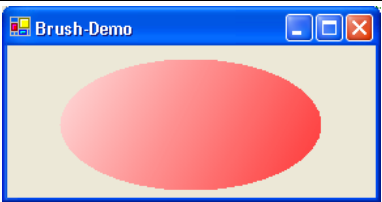
Eine verwendete Bitmap-Datei wird beim empfehlenswerten Verzicht auf eine Pfadangabe im Verzeichnis mit dem Assembly erwartet (z.B. in `...\Brush\bin\Debug`). Später werden Sie eine Möglichkeit kennen lernen, Bitmap-Dateien als so genannte *Ressourcen* in das Assembly aufzunehmen.

Mit Texturen kann man nicht nur Formen füllen, sondern auch Schriftzüge, z.B.:

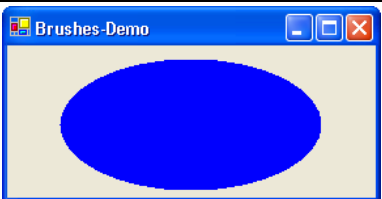
Quellcodefragment	Ausgabe
<pre>BackColor = Color.Red; gc.DrawString("Schreiben mit Textur", new Font("Arial", 20, FontStyle.Bold), new TextureBrush(new Bitmap("water.bmp")), 3, 40);</pre>	

- **LinearGradientBrush** (Namensraum **System.Drawing.Drawing2D**)

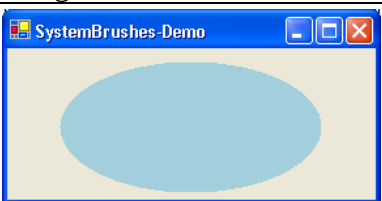
Diese **Brush**-Ableitung erzeugte Füllungen mit Farbverlauf, z.B.:

Quellcodefragment	Ausgabe
<pre>gc.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>gc.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>gc.FillEllipse(SystemBrushes.Desktop, ra);</pre>	

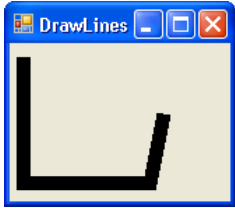
14.6 Linien, Kurven und Flächen

14.6.1 Einfache Linien und Figuren

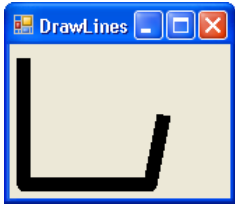
Graphics-Objekte beherrschen zahlreiche Methoden zum Zeichnen von einfachen Linien und Figuren:

- **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.:

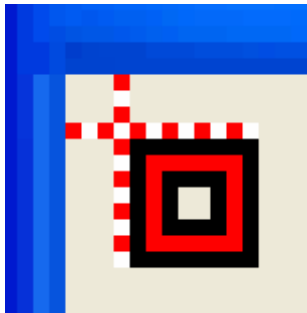
Quellcodefragment	Ausgabe
<pre>Pen stift = new Pen(Color.Black, 10); Point[] pts = {new Point(10, 10), new Point(10, 100), new Point(100, 100), new Point(110, 50)}; g.DrawLines(stift, pts);</pre>	

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:

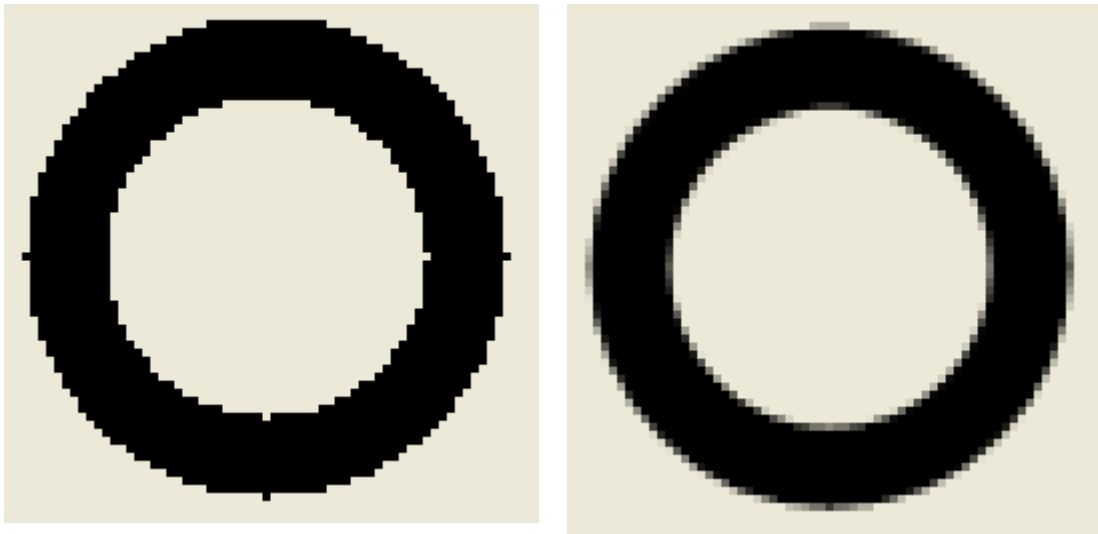
Quellcodefragment	Ausgabe
<pre>Pen stift = new Pen(Color.Black, 3); g.DrawRectangle(stift, 5, 5, 5, 5); Pen rotstift = new Pen(Color.Red); g.DrawRectangle(rotstift, 5, 5, 5, 5);</pre>	

Der stark vergrößerte Ausschnitt zeigt:

- o Bei einer Linienbreite von einem Pixel bildet wie erwartet der Pixel (5,5) die linke obere Ecke (siehe rotes Quadrat).
- o Die Breite des roten Quadrates beträgt **6** Pixel! Petzold (2002, S. 176) spricht hier vom *1-Pixel-Fehler*.
- o Die 3-Pixel breite Linie wird mittig gezeichnet (vgl. Abschnitt 14.5.2), so dass beim schwarzen Quadrat eine Gesamtbreite von 8 Pixeln erreicht wird.

- **DrawEllipse()**

Im Zusammenhang mit dem Zeichnen einer Ellipse per **DrawEllipse()** soll die Kantenglättung (das 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:

```
g.SmoothingMode = SmoothingMode.HighQuality;
Pen stift = new Pen(Color.Black, 10);
g.DrawEllipse(stift, 15, 15, 50, 50);
```

Ganz ohne Geschwindigkeitseinbuße sind derartige Verschönerungen natürlich nicht zu haben.

- **DrawArc()**

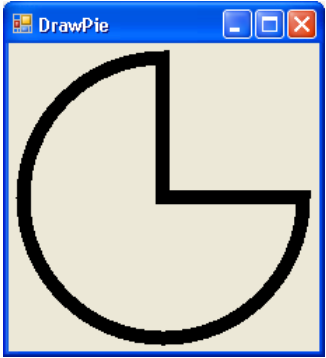
Im folgenden Beispiel wird einem Quadrat mit 200 Pixeln Kantenlänge ein Bogen von 0° bis 270° (jeweils ab X-Achse im Uhrzeigersinn gemessen) eingezeichnet:

Quellcodefragment	Ausgabe
<pre>Pen stift = new Pen(Color.Black, 5); stift.EndCap = LineCap.ArrowAnchor; g.DrawArc(stift, 10, 10, 200, 200, 0, 270);</pre>	

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*.

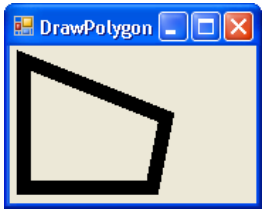
- **DrawPie()**

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); g.DrawPie(stift, 10, 10, 200, 200, 0, 270);</pre>	

- **DrawPolygon()**

Im Unterschied zur verwandten Methode **DrawLines()** liefert **DrawPolygon()** automatisch eine geschlossene Figur, z.B.:

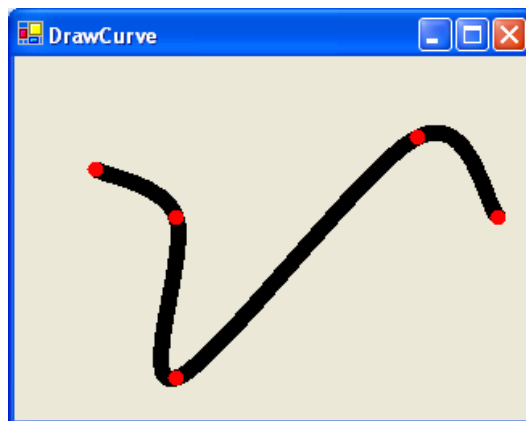
Quellcodefragment	Ausgabe
<pre>Pen stift = new Pen(Color.Black, 10); Point[] pts = {new Point(10, 10), new Point(10, 100), new Point(100, 100), new Point(110, 50)}; g.DrawPolygon(stift, pts);</pre>	

14.6.2 Splines

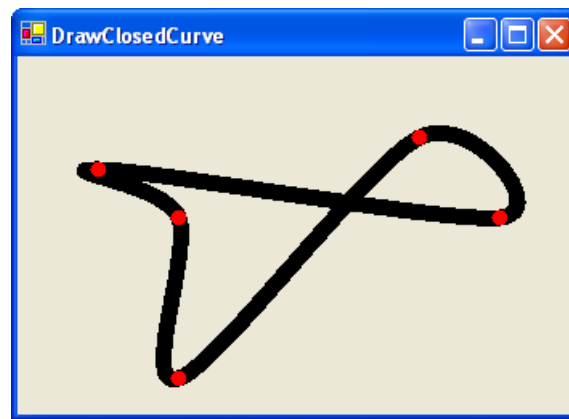
Bei den mit **DrawCurve()** erzeugten *kardinalen* (kanonischen, traditionellen) Splines wird durch vorgegebene Punkte eine geglättete Linie gelegt, z.B.:

```
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:



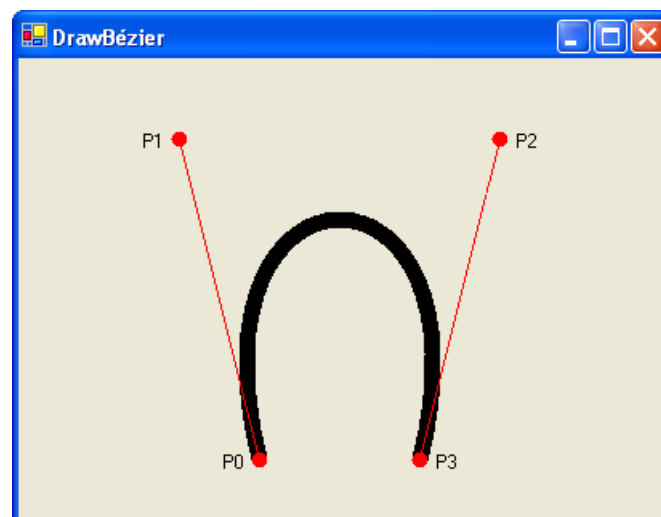
Mit **DrawClosedCurve()** erhält man einen *geschlossenen* kardinalen Spline, z.B.:



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:
 - P2 der „linken“ Kurve
 - der Grenzpunkt
 - P1 der „rechten“ Kurve

14.6.3 Flächen

Zum Zeichnen von Flächen (gefüllten Formen) bietet das Grafiksystem folgende Methoden, wobei ein Pinsel (**Brush**-Objekt, siehe Abschnitt 14.5.3) für Gestaltungsmöglichkeiten sorgt:

- **FillRectangle()**

Mit dieser Methode lassen sich auch *einzelne* Punkte in einer Pixelmatrix setzen, z.B.

```
g.FillRectangle(Brushes.Red, 5, 5, 1, 1);
```

Dabei wird tatsächlich genau *ein* Pixel eingefärbt (vgl. Abschnitt 14.6.1 zum 1-Pixel-Fehler bei **DrawRectangle()**).

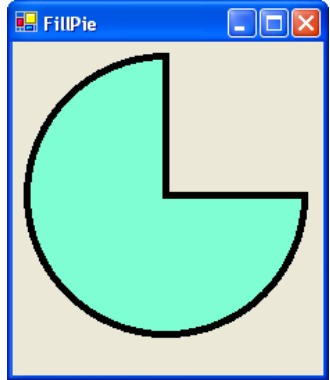
- **FillEllipse()**

Diese Methode haben wir schon im ersten Beispiel von Abschnitt 14 dazu verwendet, um gefüllte Kreise zu malen:

```
g.FillEllipse(Brushes.Blue, e.X - 10, e.Y - 10, 20, 20);
```

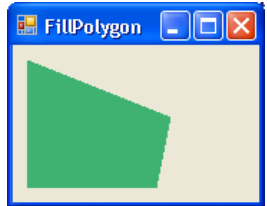
- **FillPie()**

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()**

Beispiel:

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 5-zackiger Stern geschickt über ein Polygon mit lediglich 5 Punkten gezeichnet:

```
using System;
using System.Windows.Forms;
using System.Drawing;
using System.Drawing.Drawing2D;

class Stern : Form {
    public Stern() {
        Text = "Stern";
        ClientSize = new Size(350, 350);
    }
}
```

```

protected override void OnPaint(PaintEventArgs e) {
    Point[] pts = new Point[5];
    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;
    }
    e.Graphics.FillPolygon(Brushes.MediumSeaGreen, pts, FillMode.Alternate);
}

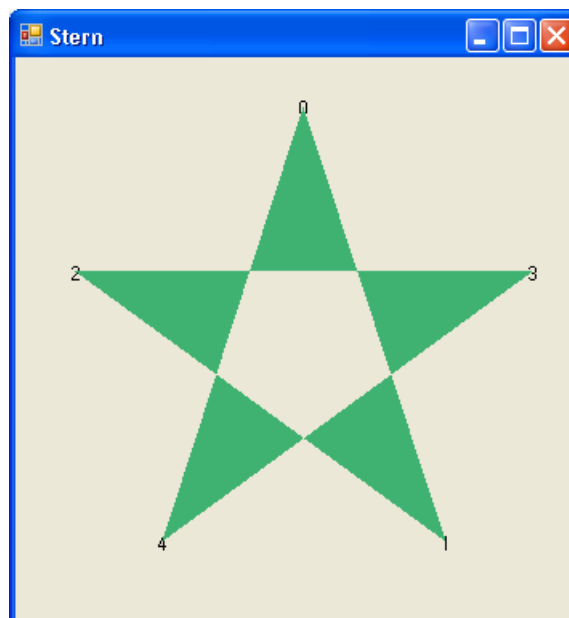
static void Main() {
    Application.Run(new Stern());
}

```

Zur Berechnung der Koordinaten ist ein wenig Trigonometrie erforderlich:

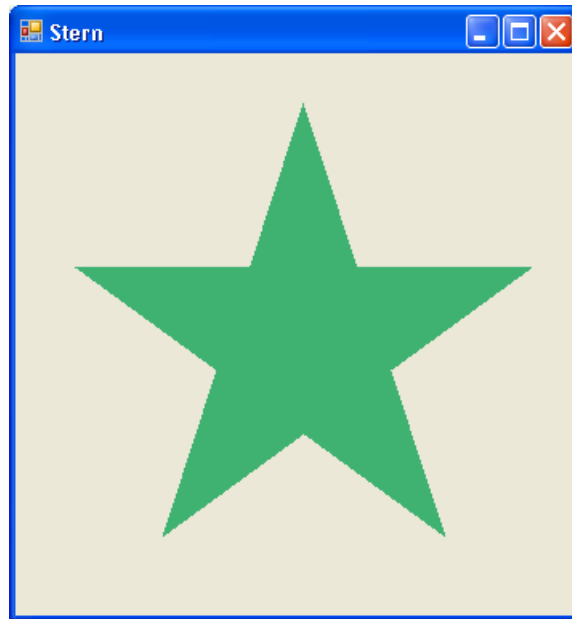
- Die Koordinaten Eckpunkte des Sterns werden über die Kosinus- bzw. Sinus-Funktion des aktuellen Winkels ermittelt, wobei zusätzlich eine Verschiebung und eine Streckung erfolgt.
- Weil die Y-Koordinate nach unten wächst, wird der Winkel initial auf $\frac{3}{2}\pi$ ($= 270^\circ$) gesetzt, damit der Stern „aufrecht“ steht.
- Beim 5-maligen Durchlaufen der **for**-Schleife werden zwei vollständige Kreisläufe absolviert. Durch die Winkelaktualisierung um $\frac{4}{5}\pi$ werden die Eckpunkte in der Reihenfolge 0, 2, 4, 1, 3 auf dem Kreisbogen angeordnet.

Beim voreingestellten Füllmodus **Alternate** (*abwechselnd*) bleibt ein 5-eckiger Bereich ungefärbt:



Zur Klärung der Konstruktion wurden hier zusätzlich die Nummern der Eckpunkte per **Draw-String()** ausgegeben.

Der Füllmodus **Winding** (*gewunden*) färbt in der Regel alle umgrenzten Flächen:



14.7 Text malen

Zur Ausgabe einer individuell gestalteten Beschriftung erzeugt man zunächst ein **Font**-Objekt, z.B.:

```
Font font = new Font("Arial", 12, FontStyle.Bold);
```

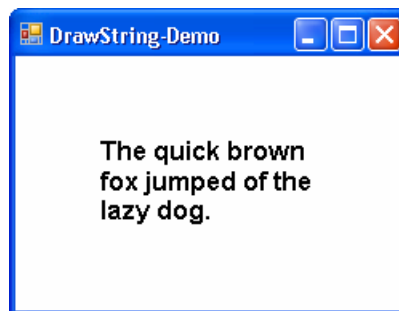
Position und Ausdehnung einer Beschriftung lassen sich z.B. per **Rectangle**-Instanz festlegen:

```
Rectangle rect = new Rectangle(50, 50, 150, 70);
```

Der folgende **DrawString()**-Aufruf

```
g.DrawString(text, font, SystemBrushes.WindowText, rect);
```

liefert die Ausgabe:



Als Werkzeug verwendet **DrawString()** keinen Stift, sondern einen *Pinsel* (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.

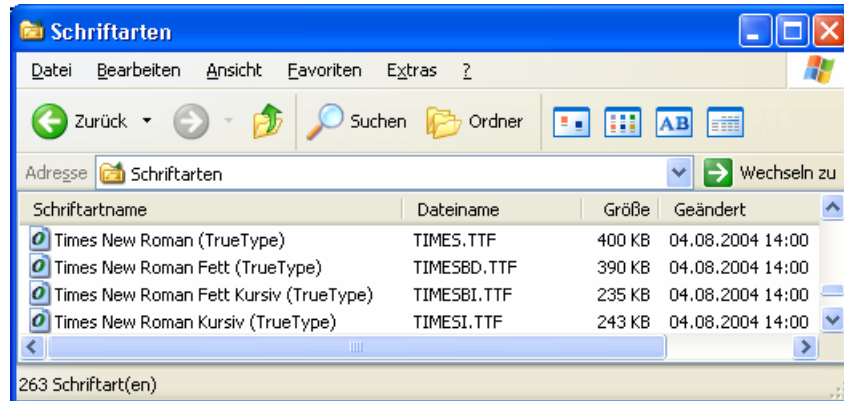
14.7.1 Schriftarten und -familien

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.

In 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 entwickelt von Apple und Microsoft für MacOS bzw. Windows)
- **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.:



Mit den folgenden 13 Schriftarten ist bei praktisch jeder Windows-Installation zu rechnen:

- 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 (Συμβολ)

Wie Sie bereits aus mehrfacher Erfahrung (z.B. mit der **Font**-Eigenschaft der Klasse **Control**) wissen, ist im .NET – Framework ein Objekt der Klasse **System.Drawing.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 **Geviertgröße** einer Schriftart wird meist 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 leicht 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 size und Schriftstil style (falls vorhanden) in den **Font**-Array fonts übernommen (siehe Formularkonstruktor):

```
using System.Windows.Forms;
using System.Drawing;
using System;

public class FontFamilyDemo : Form {
    Font[] fonts;
    int nFonts;

    public 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;
    }
}

static void Main() {
    Application.Run(new FontFamilyDemo());
}

```

Zweck des Programms ist die Anzeige einer rollbaren Liste mit allen zu einem bestimmten Schnitt verfügbaren Schriftarten, z.B. zu Arial:



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);
```

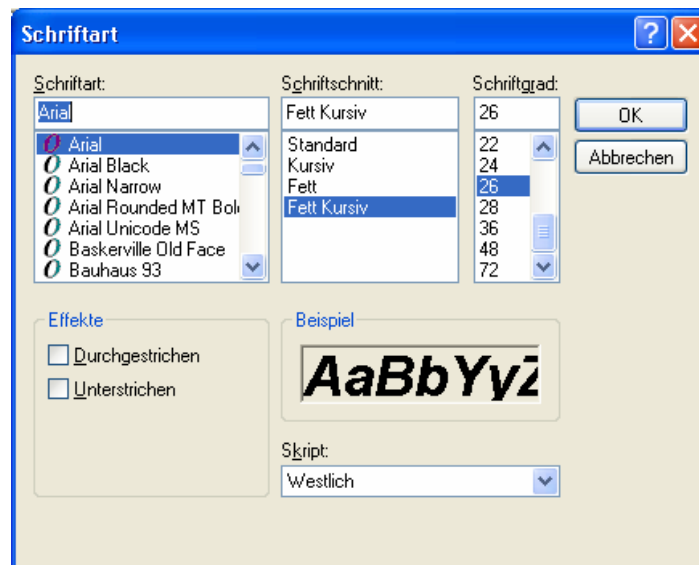
Weil mit einer längeren Liste zu rechnen ist, benötigt das Programm eine vertikale **Bildlaufleiste**, die auf folgende Weise mit geringem Aufwand realisiert wird:

- Auf dem Formular wird ein Steuerelement der Klasse **Panel** untergebracht, das als Zeichenfläche für die **DrawString()**-Aufrufe dient. Sein **OnPaint**-Ereignis wird in der Methode **PanelOnPaint** behandelt (siehe oben).
- Besitzt die **AutoScroll**-Eigenschaft eines Formulars den Wert **true**, dann ergänzt das Laufzeitsystem automatisch Bildlaufleisten, wenn die Steuerelemente des Formulars größer sind als sein Klientenbereich.
- 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.

14.7.2 Standarddialog zur Schriftauswahl

Analog zum Einsatz des Farbauswahl-Standarddialogs (siehe Abschnitt 14.5.1) 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 = fd.Font;
        text = font.Name;
        Refresh();
    }
}
```

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.

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

...\\BspUeb\\GDI+\\Textausgabe\\FontDialog

Weitere Informationen zur Klasse **FontDialog** folgen später im Kontext mit den übrigen Windows-Standarddialogen (zur Farb- und Dateiauswahl).

14.7.3 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 6 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 oben bereits ausreichend informiert. Als Pinsel wird man in der Regel ein **SolidBrush**-Objekt verwenden (vgl. Abschnitt 14.5.3), wobei die Wahl der Farbe eine kurze Überlegung wert ist. Sinnvoller als Schwarz kann die vom Benutzer zu beeinflussende Farbe **SystemColors.WindowText** sein, wobei sich dann als Hintergrundfarbe die Entsprechung **SystemColors.Window** eignet.

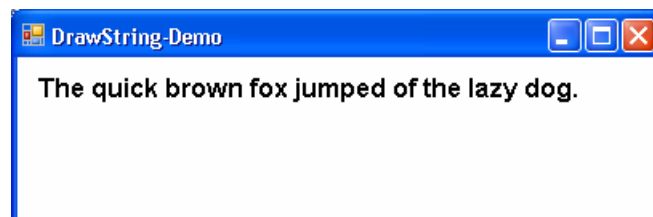
Verwendet man zur Textpositionierung ein **PointF**-Objekt oder ein Koordinatenpaar aus zwei **float**-Werten so resultiert eine *einzig*e Ausgabezeile, wenn die Zeichenfolge keinen Zeilenumbruch enthält. Mit dem Text

```
string text = "The quick brown fox jumped of the lazy dog.";
```

liefert der Aufruf

```
g.DrawString(text, font, SystemBrushes.WindowText, 10, 10);
```

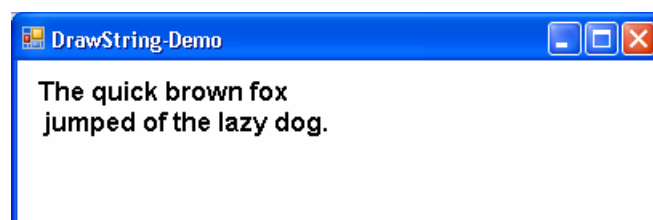
folgendes Ergebnis:



Der Text

```
string text = "The quick brown fox\n jumped of 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!«";
```

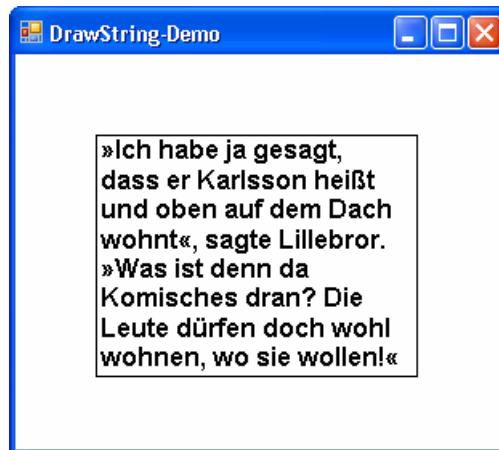
wird vom **DrawString()**-Aufruf

```
g.DrawString(text, font, SystemBrushes.WindowText, rect);
```

unter Verwendung des Rechtecks

```
Rectangle rect = new Rectangle(50, 50, 200, 150);
```

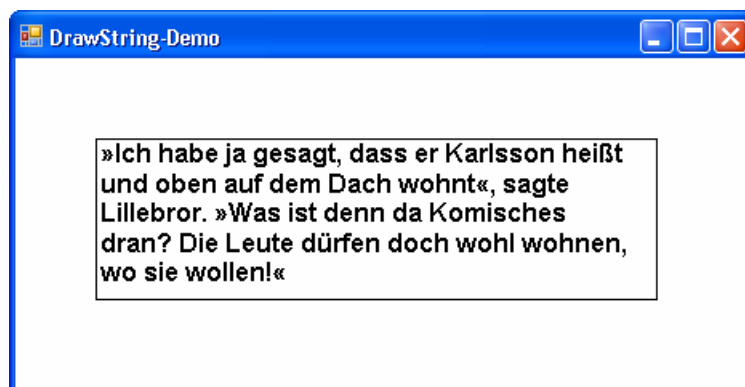
so umgebrochen:



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();
```

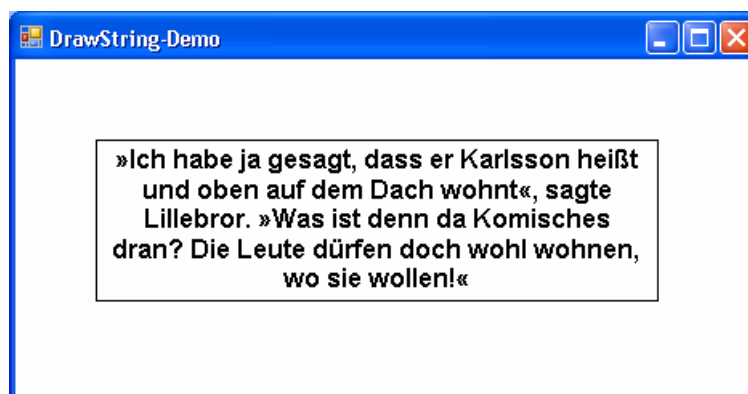
soll für eine horizontal und vertikal zentrierte Textausgabe sorgen:

```
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);
```

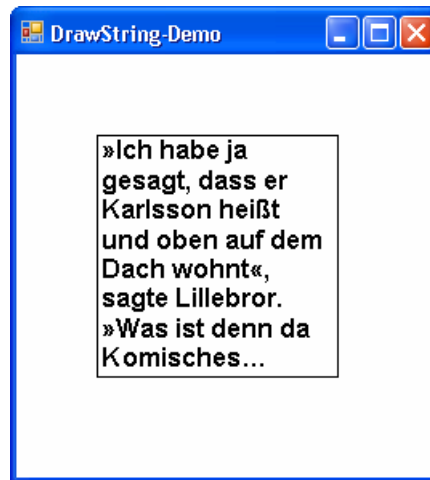
liefert mit dem letzten Textbeispiel folgende Ausgabe:



Über die **StringFormat**-Eigenschaft **Trimming** legt man fest, wie ein Text gekürzt werden soll, wenn er nicht vollständig in das Layout-Rechteck 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:



14.7.4 Maß halten

Bei der Textausgabe werden gelegentlich genaue Angaben zum horizontalen und vertikalen Platzbedarf einer bestimmten Zeichenfolge bei Verwendung einer bestimmten Schriftart benötigt.

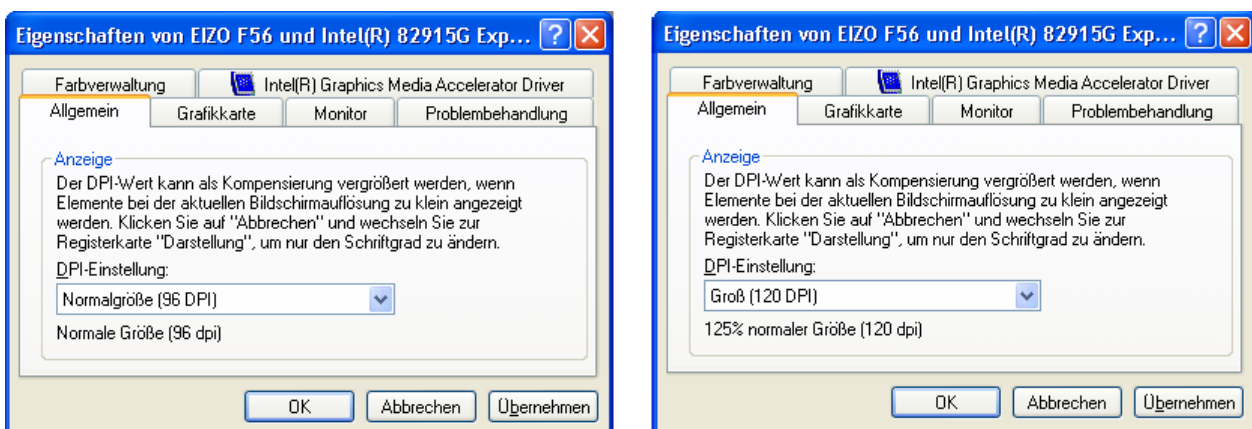
14.7.4.1 Schrifthöhe und Zeilenabstand

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 per

Systemsteuerung > Anzeige > Einstellungen > Erweitert

beeinflusst werden kann, z.B.:



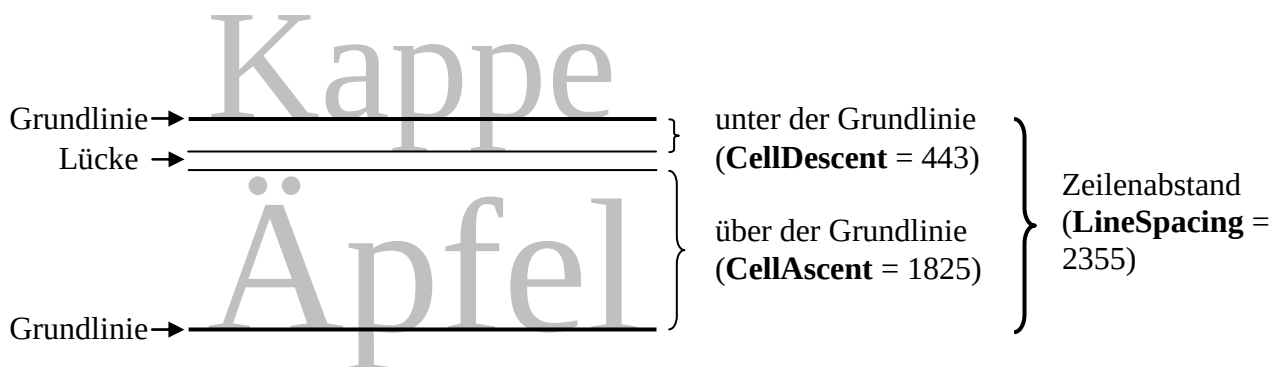
Die aktive Ein- (bzw. Unter-) stellung lässt sich über die **DpiY**-Eigenschaft eines zur Bildschirmausgabe befähigten **Graphics**-Objektes ermitteln (siehe unten). In Abschnitt 14.8.1 über metrische

Maßeinheiten (an Stelle von Pixeln) werden wir uns mit der potentiellen Ungenauigkeit dieser Unterstellung nochmals 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**:

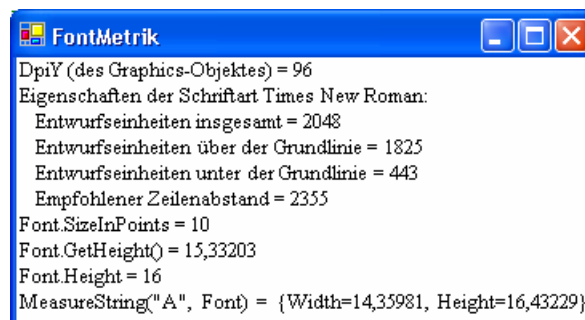


In der Abbildung ist die **EmHeight** des Schriftschnitts von 2048, die als Referenzgröße für die anderen Angaben fungiert, *nicht* darstellbar.

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 eine Angabe zur maximalen vertikalen Ausdehnung einer Schrift beigesteuert, die oft über dem empfohlenen Zeilenabstand liegt und nicht genutzt werden sollte.

Im Projektordner

... \BspUeb\GDI+\Textausgabe\FontMetrik

finden Sie ein Programm, das die eben diskutierten Größen per **DrawString()** ausgibt.

14.7.4.2 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 Textbreite und damit die Startposition der nächsten Textausgabe zu berechnen:

```
using System.Windows.Forms;
using System.Drawing;
using System;

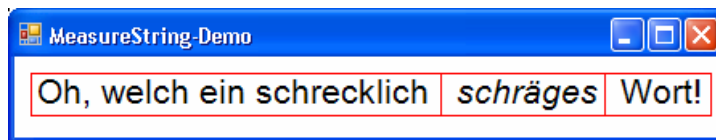
public class MeasureStringDemo : Form {
    Brush brush = SystemBrushes.WindowText;
    Font[] fonts;
    string[] texte;

    public 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;
        }
    }

    static void Main() {
        Application.Run(new MeasureStringDemo());
    }
}
```

Das als Rückgabewert gelieferte **SizeF**-Objekt ist allerdings nicht nur zu hoch (siehe oben), sondern auch zu breit, so dass zwischen horizontal aufeinander folgenden **DrawString()**-Ausgaben unschöne Lücken entstehen, wenn man sich bei der Positionierung an diesem **SizeF**-Objekt orientiert, z.B.:



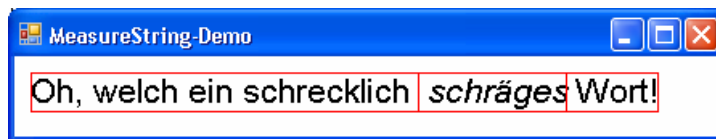
Das Ergebnis wird präziser, wenn bei **DrawString()** und bei **MeasureString()** ein spezielles **StringFormat**-Objekt assistiert, das von der statischen **StringFormat**-Eigenschaft **GenericTypographic** geliefert wird.

```
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.:



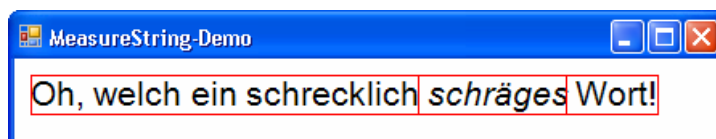
Eine weitere Verbesserung stellt sich ein, wenn man über die **Graphics**-Eigenschaft **TextRenderingHint** das **Antialiasing** einschaltet, z.B.

```
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.

14.8 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 wird die Beschäftigung mit alternativen Maßeinheiten, Skalierungsfaktoren und Koordinatentransformationen unvermeidlich.

14.8.1 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 (via **Systemsteuerung > Eigenschaften der Anzeige**) beeinflussbaren Einstellungen (vgl. Abschnitt 14.7.4.1):

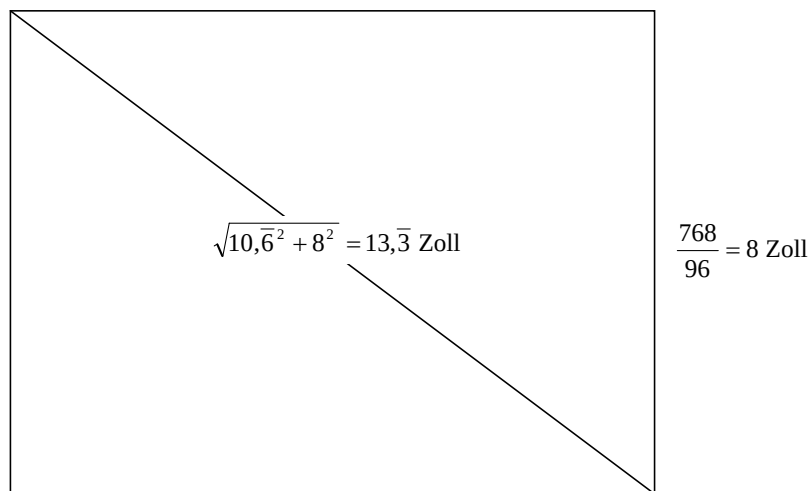
- Anzahl der Bildpunkte pro Zeile bzw. Spalte
- Bildschirmauflösung in der Einheit DPI (*Dots Per Inch*)

Daraus ist die Größe des Bildschirms leicht zu errechnen. Bei der Überprüfung eines konkreten 17-Zoll-Röhrenmonitors ergaben sich folgende Werte bzw. Einstellungen:

- Bildpunkte: 1024×768
- DPI-Einstellung: 96

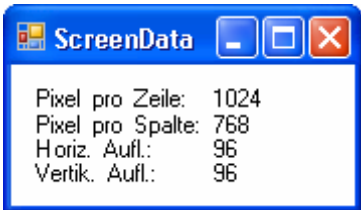
Das Betriebssystem geht von einer Bildschirmdiagonale mit 13,33 Zoll aus:

$$\frac{1024}{96} = 10,6 \text{ Zoll}$$



Die sichtbare Bildschirmdiagonale des Monitors betrug aber 15,75 Zoll, so dass z.B. eine Linie mit der geplanten Länge von 1 cm (ca. 38 Pixel) eine tatsächliche Länge von ca. 1,18 cm Länge hat. Dieser „automatische Zoom-Faktor“ stellt aber nur in seltenen Fällen ein echtes Problem dar.

Im folgenden Quellcodesegment wird demonstriert, wie man die Anzahl der horizontalen bzw. vertikalen Punkte auf dem primären Bildschirm ermitteln kann:

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>	

Von welchen Auflösungen ein **Graphics**-Objekt (mit einem Bildschirm oder auch mit einem Drucker verbunden) ausgeht, erfährt man über seine Eigenschaften **DpiX** und **DpiY**.

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 auf ca. 100 DPI eingestellt sind, genügt für einfache Zwecke die Orientierung an dieser Standardauflösung.

14.8.2 Metrische Maßeinheiten

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

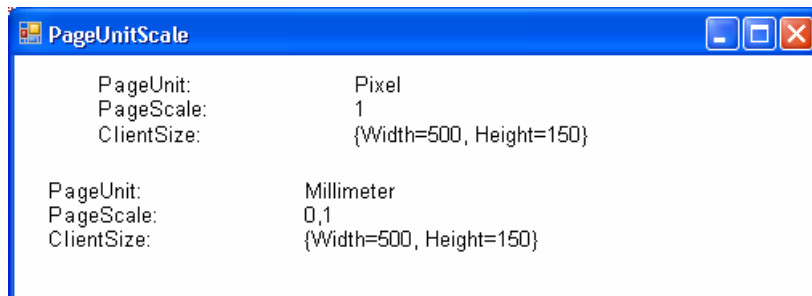
Über die **Graphics**-Eigenschaft **PageScale** lässt sich die Schrittweite verfeinern oder vergrößern. Nach den folgenden Anweisungen versteht das **Graphics**-Objekt alle Positionen und Größen in der Einheit 1/10 Millimeter:

```
g.PageUnit = GraphicsUnit.Millimeter;
g.PageScale = 0.1f;
```

Weil die per **PageUnit** und **PageScale** definierte **Seitentransformation** mit einem **Graphics**-Objekt vereinbart wird, hat sie z.B. keinen Einfluss auf die Ausgabe der 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);
}
```

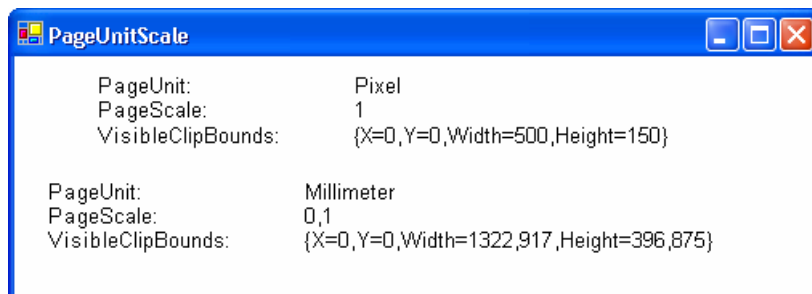
Die Seitentransformation verändert zwar die horizontale Position der **DrawString()**-Ausgabe, aber nicht den **ClientSize**-Wert:



Ist man an der Größe des Klientenbereichs in metrischen Koordinaten interessiert, hilft die **VisibleClipBounds**-Eigenschaft der **Graphics**-Objekte weiter, wie der folgende **Paint**-Handler

```
protected override void OnPaint(PaintEventArgs e) {
    Graphics g = e.Graphics;
    g.DrawString("PageUnit:\t\t" + g.PageUnit.ToString() +
        "\nPageScale:\t\t" + g.PageScale.ToString() +
        "\nVisibleClipBounds:\t" + g.VisibleClipBounds.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() +
        "\nVisibleClipBounds:\t" + g.VisibleClipBounds.ToString(),
        new Font("Arial", 10), SystemBrushes.WindowText, 50, 200);
}
```

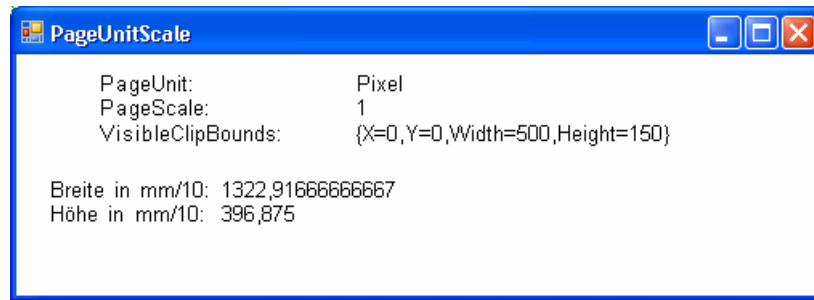
demonstriert:



Leider liefert die **VisibleClipBounds**-Eigenschaft auf meinem Rechner mit dem .NET - Framework 2.0 nicht mehr das gewünschte Resultat. Mit etwas einfacher Mathematik

```
g.DrawString("Breite in mm/10:\t" +
    (ClientSize.Width/g.DpiX*2.54*100).ToString()+
    "\nHöhe in mm/10:\t" +
    (ClientSize.Height/g.DpiX*2.54*100).ToString(),
    new Font("Arial", 10), SystemBrushes.WindowText, 50, 200);
```

gewinnt man aber doch die gewünschten Werte:



14.8.3 Welttransformationen

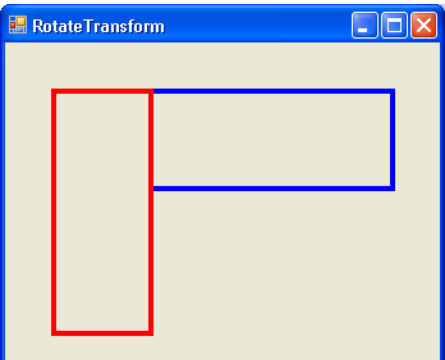
Mit der **Graphics**-Methode **TranslateTransform()** lässt sich der Ursprung des Koordinatensystems aus seiner linken oberen Ecke locken, z.B.:

Quellcodefragment	Ausgabe
<pre>protected override void OnPaint(PaintEventArgs e) { Graphics g = e.Graphics; Pen p = new Pen(Color.Black, 2); for (int i = 0; i < 6; i++) { g.DrawEllipse(p, 0, 0, 30, 30); g.TranslateTransform(40, 40); } p.Dispose(); }</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; Pen pen = Pens.Black; g.DrawEllipse(pen, 20, 20, 300, 300); g.ScaleTransform(1, 0.5f); g.DrawEllipse(pen, 20, 20, 300, 300); }</pre>	

Mit der **Graphics**-Methode **RotateTransform()** kann man eine Figur drehen, ohne Koordinatenberechnungen durchführen zu müssen, z.B.:

Quellcodefragment	Ausgabe
<pre>protected override void OnPaint(PaintEventArgs e) { Graphics g = e.Graphics; g.PageUnit = GraphicsUnit.Millimeter; g.TranslateTransform(30, 10); g.DrawRectangle(Pens.Blue, 0, 0, 50, 20); g.RotateTransform(90); g.DrawRectangle(Pens.Red, 0, 0, 50, 20); }</pre>	

Nach Ausgabe des blauen Rechtecks per **DrawRectangle()** wird das Koordinatensystem um 90 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()** in der Mitte des Klientenbereichs 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 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.

14.9 Rastergrafik

Mit der Klasse **Bitmap** unterstützt das .NET – Framework die Darstellung und Bearbeitung von Rastergrafiken, die z.B. aus einer Datei importiert werden können:

```
Bitmap image = new Bitmap("land.bmp");
```

Das folgende Fenster zeigt links das per **DrawImage()**

```
g.DrawImage(image, 10, 10);
```

gemalte Original und rechts die per

```
RectangleF neueForm = new RectangleF(280, 10,
                                     1.5f * image.Width, image.Height);
g.DrawImage(image, neueForm);
```

horizontal gedehnte Variante:



Zur Definition der gewünschte Position und Größe kommt bei der zweiten Ausgabe eine **RectangleF**-Instanz zum Einsatz, die im Vergleich zu einer **Rectangle**-Instanz zugunsten einer verbesserten Genauigkeit mit **float**-Eigenschaften ausgestattet ist.

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)

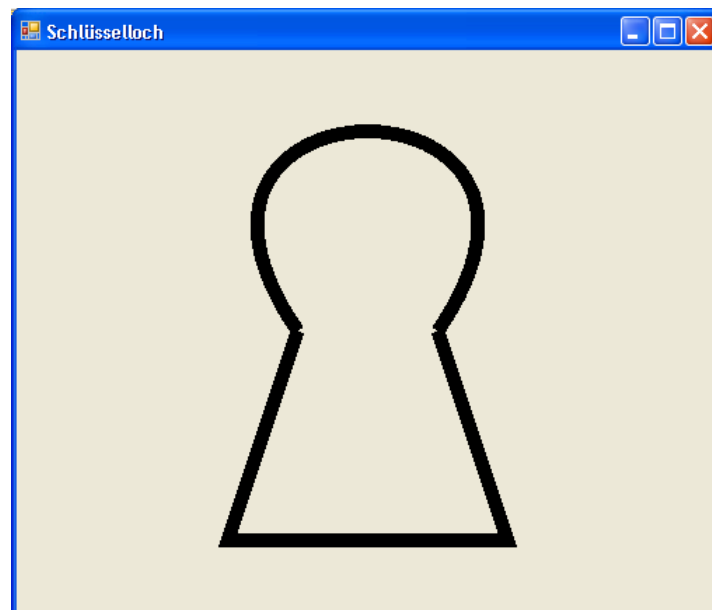
14.10 Pfade, Zeichen- und Anzeigebereiche

14.10.1 Pfade

Im folgenden **Paint**-Handler

```
protected override void OnPaint(PaintEventArgs e) {
    Graphics g = e.Graphics;
    Pen stift = new Pen(Color.Black, 10);
    Point ul = new Point(150, 350);
    Point ur = new Point(350, 350);
    Point p0 = new Point(200, 200);
    Point p1 = new Point(70, 10);
    Point p2 = new Point(430, 10);
    Point p3 = new Point(300, 200);
    Point[] lz = new Point[] { p3, ur, ul, p0 };
    g.DrawLines(stift, lz);
    g.DrawBezier(stift, p0, p1, p2, p3);
}
```

wird mit den **Graphics**-Methoden **DrawLines()** und **DrawBezier()** (siehe oben) 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**. Einem Objekt dieser Klasse kann man eine Sequenz von Linien, Kurven und Bögen übergeben und dann den Auftrag erteilen, den gesamten Pfad mit perfekten Übergängen zu zeichnen.

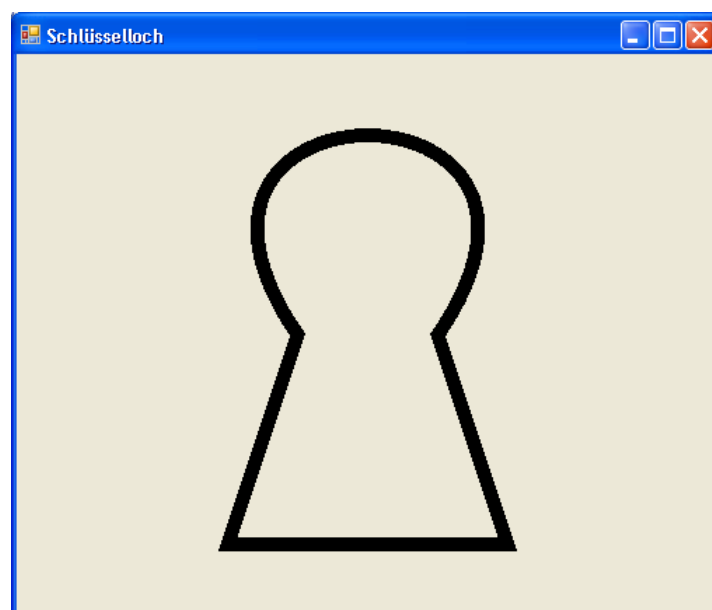
Werden in obigem **Paint**-Handler die **Graphics**-Methoden

```
g.DrawLines(stift, lz);  
g.DrawBezier(stift, p0, p1, p2, p3);
```

durch folgende Konstruktion

```
GraphicsPath gp = new GraphicsPath();  
gp.AddLines(lz);  
gp.AddBezier(p0, p1, p2, p3);  
gp.CloseFigure();  
g.DrawPath(stift, gp);
```

ersetzt, dann resultiert ein perfektes Schlüsseloch:



Anstelle der **Graphics**-Methoden **DrawLines()** und **DrawBezier()** werden die **GraphicsPath**-Methoden **AddLines()** und **AddBezier()** verwendet, die (abgesehen vom fehlenden **Pen**-Parameter)

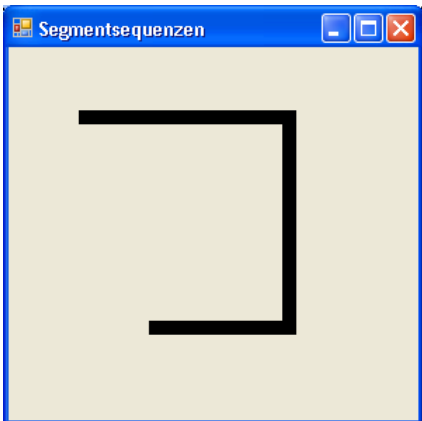
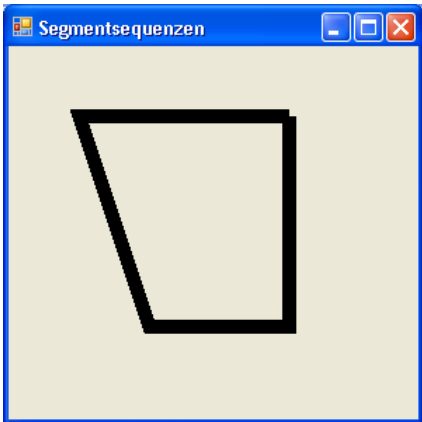
über dieselbe Signatur verfügen. Ohne den Aufruf der Methode **CloseFigure()** (siehe unten) bleibt der Erfolg unvollständig.

Weitere **GraphicsPath**-Methoden zum Bestücken eines Pfads sind:

- **AddLine()**
- **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.

Ein Pfad kann aus mehreren **Figuren** bestehen, die sich wiederum aus beliebig vielen Segmenten zusammensetzen, welche jeweils mehrere Punkte enthalten (je nach Typ). Wir haben eben eine Schlüsselloch-Figur aus 2 Segmenten aufgebaut (ein Linienzug mit 3 Punkten und ein Bézier-Spline) und dabei den Startpunkt jedes Segmentes identisch mit dem Endpunkt des Ergänzungsstücks gewählt. Generell werden zwei nacheinander in eine Figur eingefügte Segmente automatisch durch Linien zwischen dem Endpunkt des früheren und Startpunkt des späteren Segmentes 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. Schreibfaule könne sich am Ende einer Figur ein Liniensegment sparen, weil dieses von **CloseFigure()** ohnehin ergänzt wird.

Während **CloseFigure()** die aktuell im Aufbau befindliche Figur schließt, bearbeitet **CloseAllFigures()** alle offenen Figuren. Der folgende Pfad:

```
GraphicsPath path = new GraphicsPath();
path.AddBezier(100, 150, 150, 250, 400, 200, 250, 50);
path.CloseFigure();
path.AddEllipse(250, 100, 15, 15);
path.AddLine(200, 200, 180, 150);
```

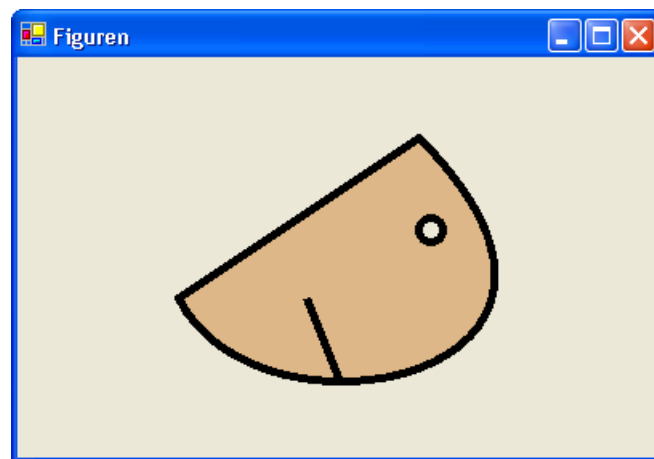
besteht aus drei Figuren:

- Die erste Figur startet mit einem Bézier-Spline und wird per **CloseFigure()** durch eine Linie komplettiert.
- Durch die Methode **AddEllipse()** wird eine neue 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 eine weitere Figur.
- Mit **StartFigure()** kann man eine neue Figur beginnen, ohne die aktuelle zu schließen.

Zur Darstellung eines Pfades bietet die Klasse **Graphics** die Methoden **DrawPath()** und **FillPath()**. Für den eben beschriebenen Pfad liefern die Aufrufe

```
g.FillPath(Brushes.BurlyWood, path);
g.DrawPath(pen, path);
```

folgendes Ergebnis:



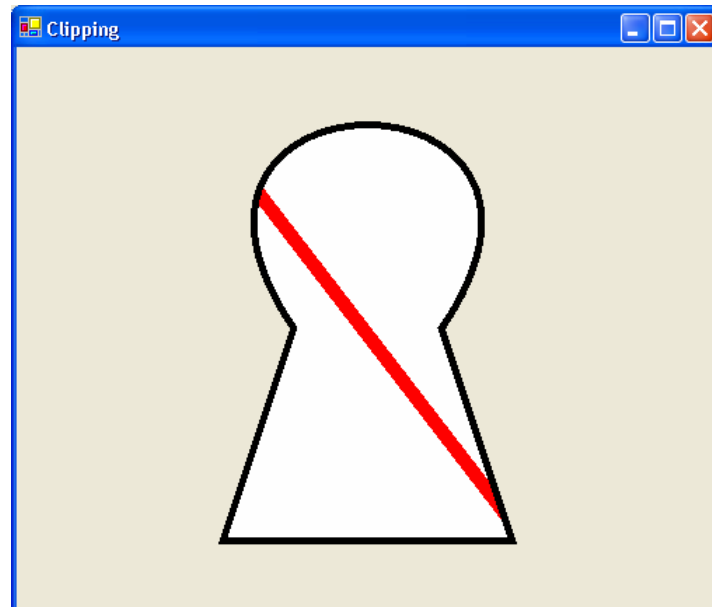
In Petzold (2003, S. 597ff) findet sich eine zahlreiche weitere Optionen zum Erzeugen und Transformieren von Pfaden.

14.10.2 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 14.10.1:

```
g.SetClip(gp);
g.Clear(Color.White);
Pen rotstift = new Pen(Color.Red, 10);
g.DrawLine(rotstift, 100, 10, 400, 400);
```

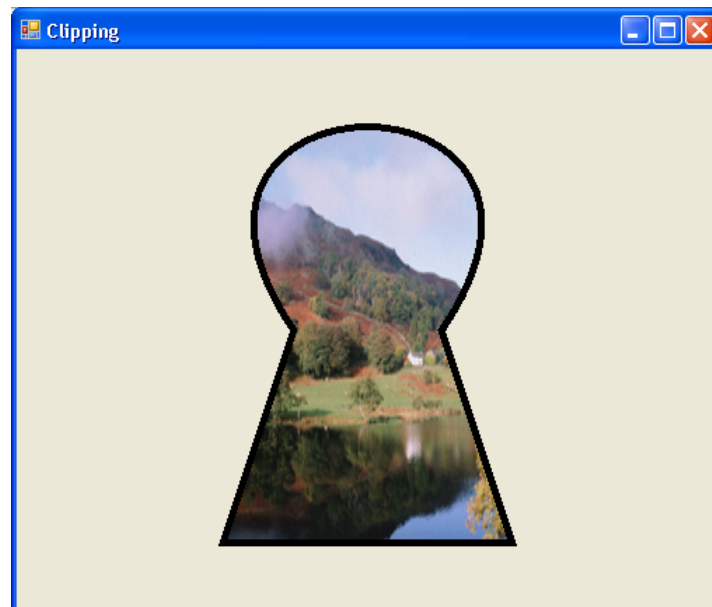
Die Ausgaben der **Graphics**-Methoden **Clear()** und **DrawLine()** sind nur im Schlüsselloch zu beobachten:



Mit der Clipping-Technik lassen sich beliebig geformte (per Pfad definierte) Bereiche von Bitmaps anzeigen, z.B.:

```
Bitmap image = new Bitmap("land.bmp");
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:



14.10.3 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, so dass die restliche Fensterfläche unsichtbar bleibt. 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:

Quellcodesegment	Ausgabe
<pre> public RegionDemo() { Text = "Region-Demo"; Size = new Size(500, 400); Point ul = new Point(150, 350); Point ur = new Point(350, 350); Point p0 = new Point(200, 200); Point p1 = new Point(70, 10); Point p2 = new Point(430, 10); Point p3 = new Point(300, 200); Point[] lz = new Point[] { p3, ur, ul, p0 }; GraphicsPath gp = new GraphicsPath(); gp.AddLines(lz); gp.AddBezier(p0, p1, p2, p3); gp.CloseFigure(); Region = new Region(gp); image = new Bitmap("land.bmp"); Button cbClose = new Button(); cbClose.Top = 280; cbClose.Left = 210; cbClose.Text = "Beenden"; cbClose.Parent = this; cbClose.Click += new EventHandler(ButtonOnClick); stift = new Pen(Color.CadetBlue, 10); shiftX = -((Width - ClientRectangle.Width)/2); shiftY = -(Height - ClientSize.Height + shiftX); } </pre>	

Weil das Formular durch die **Region**-Definition seine Titelzeile samt Standardschaltflächen verloren hat, muss über Ereignismethoden für Beweglichkeit und eine Terminierungsmöglichkeit gesorgt werden.

Über die Konstruktion

```

class RegionDemo : Form {
    . . .
    Pen stift;
    int shiftX, shiftY;

    public RegionDemo() {
        . . .
        stift = new Pen(Color.CadetBlue, 10);
        shiftX = -((Width - ClientSize.Width)/2);
        shiftY = -(Height - ClientSize.Height + shiftX);
    }

    protected override void OnPaint(PaintEventArgs e) {
        Graphics g = e.Graphics;
        g.TranslateTransform(shiftX, shiftY);
        g.DrawImage(image, new RectangleF(120, 10,
                                         image.Width, 1.5f * image.Height));
        g.DrawPath(stift, gp);
    }

    . . .
}

```

wird dafür gesorgt, dass der mit **DrawPath()** gezeichnete Rand des Schlüssellochs an der richtigen Stelle erscheint:

- Bei der **Region** - Definition werden die Koordinaten (im **GraphicsPath**-Objekt) relativ zur linken oberen Ecke *des Formulars* interpretiert. Im **DrawPath()**-Aufruf beziehen sich die

Koordinaten hingegen auf die linke obere Ecke des *Klientenbereichs*. Dies wird in **OnPaint()** durch eine Verschiebung des Ursprungs per **TranslateTransform()** kompensiert.

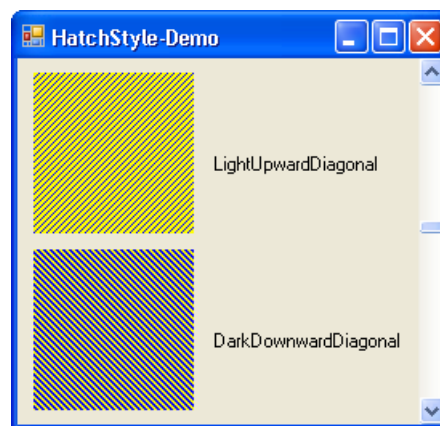
- Bei der Verschiebung muss man die Breite des Randes und der Titelzeile berücksichtigen. Weile diese von der Windows-Version abhängen könnten, werden sie im Programm durch Größenvergleich von Formular und Klientenbereich ermittelt.

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

...\\BspUeb\\GDI+\\Pfade, Zeichen- und Anzeigebereiche\\Region

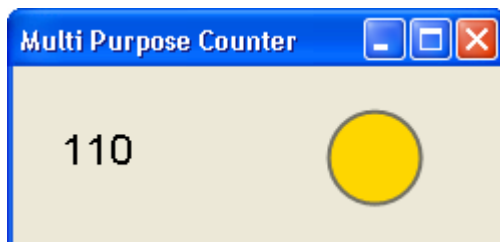
14.11 Übungsaufgaben zu Abschnitt 14

1) Erstellen Sie ein Programm, das die für **HatchBrush**-Objekte verfügbaren Muster demonstriert, z.B. mit folgender Bedienoberfläche:

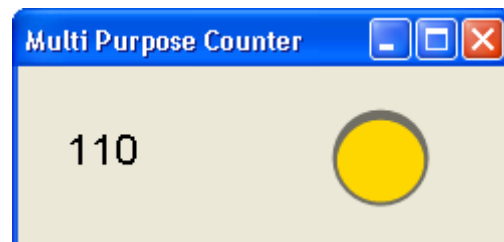


2) Realisieren Sie mit Hilfe der Klassen **GraphicsPath** und **Region** ein Formular mit einem *runden* Befehlsschalter. Eine mögliche Lösung besteht darin, ein **Button**-Objekt zeichnerisch zu überarbeiten, was z.B. in der folgenden Variante des Multi Purpose Counters (vgl. Abschnitt 8.4.2) geschieht:

Normalzustand



gedrückt



15 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 vom Schlage der Windows-Zugabe *WordPad* eignet. Wir führen das **RichTextBox**-Steuerelement in Abschnitt 15.6 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.

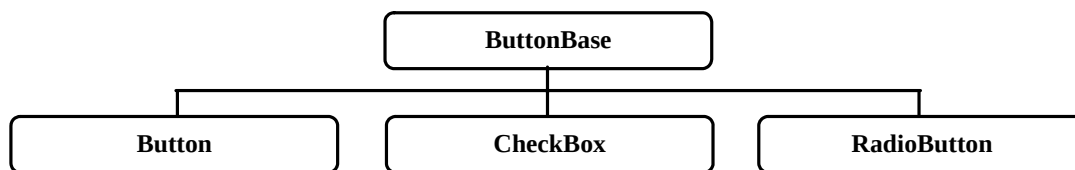
Wer sich beim Einsatz von Windows-Bedienelementen streng an die Vorgaben der Firma Microsoft halten möchte an, kann das kostenlos auf einem Webserver angebotenen Dokument [*Official Guidelines for User Interface Developers and Designers*](#) (Microsoft 2007) konsultieren.

15.1 Kontrollkästchen und Optionsfelder

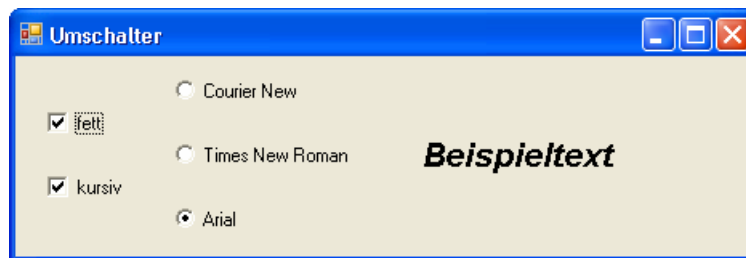
In diesem Abschnitt werden zwei Umschalter vorgestellt:

- 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 8.4.2 vorgestellten Klasse **Button** von der abstrakten Basisklasse **ButtonBase** ab:



In folgendem Programm kann für den Text einer **Label**-Komponente über zwei Kontrollkästchen der Schriftschnitt und über ein Optionsfeld die Schriftart gewählt werden:



Die beiden Kontrollkästchen (cbBold und cbItalic genannt) erlauben das separate Ein- bzw. Ausschalten der Schriftattribute **fett** und **kursiv**. Sie werden im Konstruktor zu obigem Formular folgendermaßen definiert:

```
public Umschalter() {  
    ClientSize = new Size(440, 125);  
    Text = "Umschalter";  
  
    lbTextbeispiel = new Label();  
    lbTextbeispiel.Text = "Beispieltext";  
    lbTextbeispiel.Font = new Font("Arial", 16);  
    lbTextbeispiel.Location = new Point(250, 50);  
    lbTextbeispiel.Width = 200;  
    Controls.Add(lbTextbeispiel);  
}
```

```

cbBold = new CheckBox();
cbBold.Text = "fett";
cbBold.Location = new Point(20, 30);
cbBold.Width = 60;
Controls.Add(cbBold);

cbItalic = new CheckBox();
cbItalic.Text = "kursiv";
cbItalic.Location = new Point(20, 70);
cbItalic.Width = 60;
Controls.Add(cbItalic);

EventHandler eh = new EventHandler(CheckBoxOnClick);
cbBold.Click += eh;
cbItalic.Click += eh;

. . .
}

```

Als **EventHandler** wird für beide Kontrollkästchen dasselbe Delegaten-Objekt verwendet, das auf der Methode `CheckBoxOnClick()` basiert:

```

protected void CheckBoxOnClick(object sender, EventArgs e) {
    if (cbBold.Checked)
        lbTextbeispiel.Font = new Font(lbTextbeispiel.Font,
            lbTextbeispiel.Font.Style | FontStyle.Bold);
    else
        lbTextbeispiel.Font = new Font(lbTextbeispiel.Font,
            lbTextbeispiel.Font.Style & ~FontStyle.Bold);
    if (cbItalic.Checked)
        lbTextbeispiel.Font = new Font(lbTextbeispiel.Font,
            lbTextbeispiel.Font.Style | FontStyle.Italic);
    else
        lbTextbeispiel.Font = new Font(lbTextbeispiel.Font,
            lbTextbeispiel.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 angeknüpft.
- Durch die bitweise Negation (~) und die anschließende bitweise AND-Operation werden die zum Attribut gehörigen Bits abgeschaltet.

Der Einfachheit halber werden (wenig Ressourcen schonend) bei jedem `CheckBoxOnClick()`-Aufruf 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 Markierungszustand und den Wert seiner **Checked**-Eigenschaft. Im Beispiel wird dieser Komfort genutzt, und die **Click**-Behandlungsmethoden greifen nur lesend auf die **Checked**-Eigenschaft zu.

RadioButton-Objekte treten stets in *Gruppen* auf, wobei nur *ein* Mitglied eingerastet sein kann (Wert **true** bei der Eigenschaft **Checked**). Die CLR gewährleistet diese Exklusivität und betrachtet dabei alle **RadioButton**-Objekte in einem Container als Gruppe. Im Umschalter-Beispielprogramm bildet das Hauptfenster *selbst* einen solchen gruppenbildenden Container:

```

public Umschalter() {
    . . .

    rbCourierNew = new RadioButton();
    rbCourierNew.Text = "Courier New";
    rbCourierNew.Location = new Point(100, 10);
    Controls.Add(rbCourierNew);

    rbTimesNewRoman = new RadioButton();
    rbTimesNewRoman.Text = "Times New Roman";
    rbTimesNewRoman.Location = new Point(100, 50);
    rbTimesNewRoman.Width = 190;
    Controls.Add(rbTimesNewRoman);

    rbArial = new RadioButton();
    rbArial.Text = "Arial";
    rbArial.Location = new Point(100, 90);
    rbArial.Checked = true;
    Controls.Add(rbArial);

    eh = new EventHandler(RadioButtonOnClick);
    rbCourierNew.Click += eh;
    rbTimesNewRoman.Click += eh;
    rbArial.Click += eh;
}

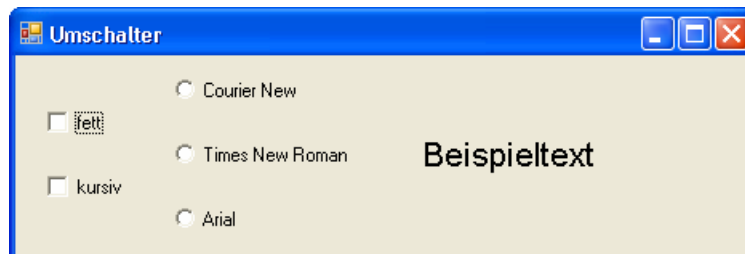
```

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 SDK – 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;
```

Bei Verzicht auf eine derartige Vorbesetzung, würde das Umschalter-Programm so starten:



Im Beispielprogramm wird für alle **RadioButton**-Objekte ein gemeinsamer **EventHandler** verwendet:

```

protected void RadioButtonOnClick(object sender, EventArgs e) {
    if (rbCourierNew.Checked)
        lbTextbeispiel.Font = new Font("Courier New",
            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("Arial",
                lbTextbeispiel.Font.Size, lbTextbeispiel.Font.Style);
}

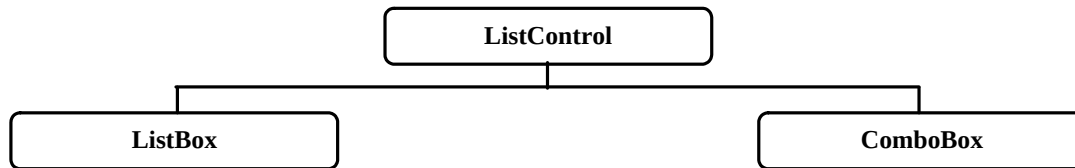
```

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

...\BspUeb\WinForms\Umschalter

15.2 Listen- und Kombinationsfelder

In diesem Abschnitt werden die Steuerelementklassen **ListBox** und **ComboBox** vorgestellt, die von der abstrakten Klasse **ListControl** abstammen:

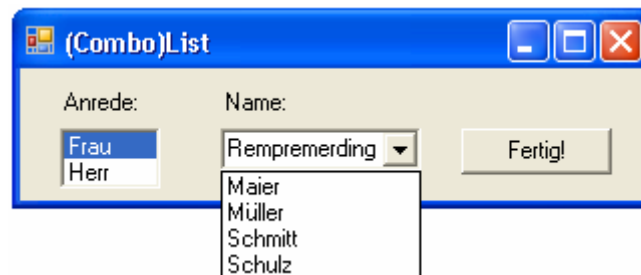


Ein **ListBox**-Steuerelement präsentiert eine Liste von Elementen, die per Mausklick oder geschickte Tastaturbedienung ausgewählt werden können.

Das **ComboBox**-Steuerelement bietet eine Kombination aus einem einzeiligen Textfeld und einer Liste. Um eine 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 Frage nach dem Namen durch eine Liste mit den häufigsten Exemplaren erleichtert:



In diesem einfachen Beispiel gibt es (abgesehen vom Klassennamen) keine syntaktischen Unterschiede zwischen dem **ListBox**- und dem **ComboBox**-Objekt:

```

using System;
using System.Windows.Forms;
using System.Drawing;
class ComboList : Form {
    ListBox listBoxAnrede;
    ComboBox listBoxNachname;

    public 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.:

- **Add(String element)**
Die Liste wird um *ein* Element erweitert. Hat die **Sorted**-Eigenschaft des Steuerelements den Wert voreingestellten **false**, wird das neue Element am Ende der Liste angehängt, anderenfalls gemäß seiner Sortierposition.
- **AddRange(String[] 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" });
```
- **Remove(String element)**
Das angegebene Element wird aus der Liste entfernt.
- **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, wobei aber oft der Zugriff die aktuell gewählte Zeichenfolge über die **ComboBox**-Eigenschaft **Text** sinnvoller ist, z.B.:

```
comboBoxNachname.Text
```

Über den **Sorted**-Eigenschaftswert **true** beauftragen wir ein **ListBox**- bzw. **ListBox**-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.

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

...\BspUeb\WinForms\Listen\ComboList

In einem weiteren Beispiel soll demonstriert werden:

- Mehrfachauswahl
- Dynamische Veränderung von Listen

Wir erstellen und befüllen wie eben ein **ListBox**-Objekt:

```
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 können.

Weil die Eigenschaft **SelectionMode** bei beiden Listen den Wert **MultiExtended** erhält, können mit Hilfe der Umschalt- und der Steuerungstasten bequem aufeinander folgende oder auch separierte Elemente gewählt werden., z.B.:

```
listeAnwaerter.SelectionMode = SelectionMode.MultiExtended;
```

In der Ereignisroutine zu den Schaltflächen

```
public void ButtonOnClick(object sender, EventArgs e) {
    if (sender == cbRein) {
        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 **SelectItems** große Ähnlichkeiten mit der bereits bekannten Eigenschaft **Items**, die auf eine Liste mit *allen* Elemente zeigt.

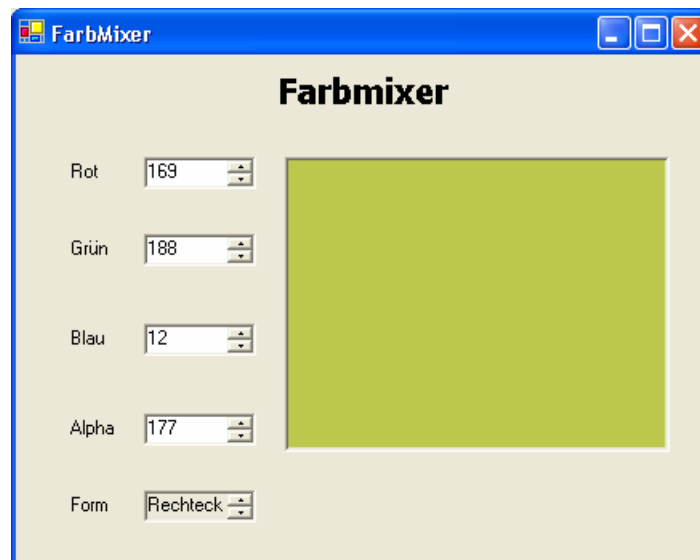
Beim Transport von links nach rechts achtet die Methode `ButtonOnClick()` darauf, dass in der Teilnehmerliste keine Dubletten entstehen.

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

...**BspUeb\WinForms\Listen\AddRemoveItems**

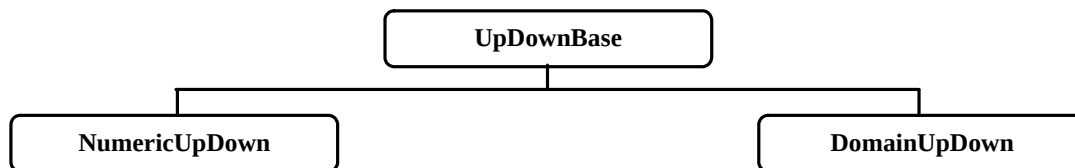
15.3 UpDown-Regler

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 findet sich eine für numerische Werte geeignete und eine für Zeichenfolgenwerte konzipierte **UpDown**-Variante, die von der gemeinsamen (abstrakten) Basisklasse **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 inkrementieren 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**-Ereignisroutine per **Refresh()**-Aufruf eine sofortige Renovierung des **Panel**-Steuerelements mit der Farbprobe, so dass die registrierte **Paint**-Behandlungsmethode eine neue Farbe gemäß Reglerstand mixt und aufträgt:

```

using System;
using System.Drawing;
using System.Windows.Forms;

public class UpDown : Form {
    Panel panel;
    NumericUpDown numUpDownRot, numUpDownBlau, numUpDownGruen, numUpDownAlpha;
    DomainUpDown domainUpDown;

    public 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(NumericUpDownOnSelectedItemChanged);
        . . .

        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.Width - 5, pan.Height - 5);
        } else
            g.FillRectangle(brush, 0, 0, pan.Width - 5, pan.Height - 5);
        brush.Dispose();
    }

    void NumericUpDownOnSelectedItemChanged(object sender, System.EventArgs e) {
        panel.Refresh();
    }

    void DomainUpDownOnSelectedItemChanged(object sender, System.EventArgs e) {
        panel.Refresh();
    }

    public static void Main() {
        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 Wertes gesorgt.

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\UpDown

15.4 Rollbalken

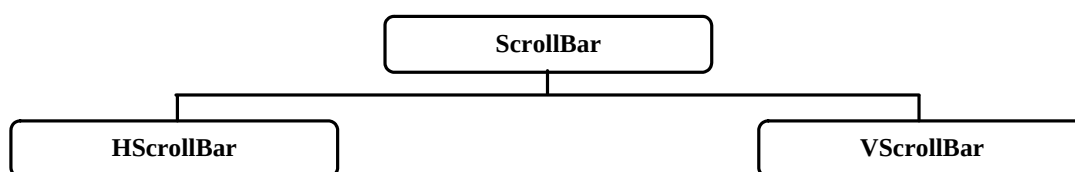
Viele Steuerelemente werden automatisch mit Rollbalken ausgestattet (z.B. **ListBox**-Objekte) oder können leicht über eine Eigenschaft (z.B. **ScrollBars** bei der Klasse **TextBox**) mit diesen Bedienelementen versorgt werden. Solche Lösungen erfüllen ihren Zweck und ersparen uns lästigen Aufwand.

Besitzt die **AutoScroll**-Eigenschaft einer von **ScrollableControl** im Namensraum **System.Windows.Forms** abgeleiteten Klasse (z.B. **Form** oder **Panel**) den Wert **true**, dann ergänzt das Laufzeitsystem automatisch Bildlaufleisten, wenn die untergeordneten Steuerelemente nicht vollständig angezeigt werden können. Bei einem Beispielprogramm zur **FontFamily**-Klasse haben wir uns vom **AutoScroll**-Angebot allerdings zu einer wenig performanten Lösung verleiten lassen (siehe Abschnitt 14.7.1). 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 Anzeigebereich befördert werden. Dass bei jeder Rollaktion das **Panel**-Element komplett neu zu erstellen ist, erschwert 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 Basisklasse **ScrollBar** ab:



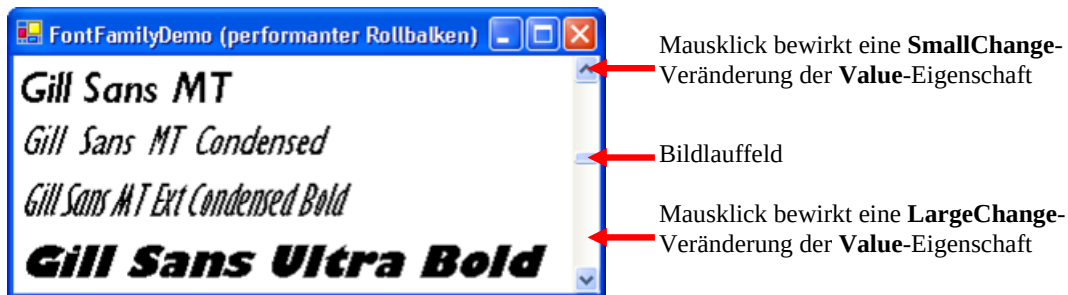
Zwar kann man die **ScrollBar**-Abkömmlinge frei positionieren, dimensionieren und färben, doch wird man sie in der Regel mit dem voreingestellten Erscheinungsbild per **DockStyle**-Wert am passenden Fensterrand anbringen. Im Fensterkonstruktor des performanzoptimierten **FontFamily**-Demonstrationsprogramms wird eine vertikale Bildlaufleiste am rechten Fensterrand verankert:

```

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 0 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, den Klientenbereich des Formulars zu verwerfen, so dass die **OnPaint**-Methode aktiv wird und die sichtbaren Schriftproben neu malt:

```

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 MouseEventHandler(FormOnMouseWheel);
```

Über die **MouseEventArgs**-Eigenschaft **Delta** erfahren wir, ob der Benutzer das Mausrad vor (**Delta** = 120) oder zurück gedreht hat (**Delta** = -120).

Den vollständigen Quellcode des Beispielprogramms finden Sie im Ordner

...\BspUeb\WinForms\ScrollBar

15.5 Timer-Komponente

Über die in Abschnitt 12.6 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 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.

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, wobei die minimale Taktlänge vom Systemzeitgeber abhängt und z.B. unter Windows NT/2000/XP 10 Millisekunden beträgt.

Die CLR verhindert, dass *mehrere* unerledigte **Tick**-Ereignisse in der Warteschlange auflaufen.

Im folgenden Beispielprogramm wird für eine per Standardkonstruktor erzeugte **Timer**-Komponente das Auslöseintervall auf 1000 Millisekunden gesetzt und eine Behandlungsmethode registriert, welche mit Hilfe eines Labels die aktuelle Zeit anzeigt:

```

using System;
using System.Windows.Forms;
using System.Drawing;
class TimerKomponente : Form {
    Label lbTime;

    public TimerKomponente() {
        Height = 100; Width = 225;
        Text = "Timer-Komponente";

        lbTime = new Label();
        lbTime.Left = 65; lbTime.Top = 20;
        lbTime.Parent = this;
        lbTime.Font = new Font("Arial", 15);
    }
}

```

```

    Timer tim = new Timer();
    tim.Interval = 1000;
    tim.Tick += new EventHandler(TimerOnTick);
    tim.Start();
}

void TimerOnTick(object sender, EventArgs e) {
    lbTime.Text = DateTime.Now.ToLongTimeString();
}

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**.

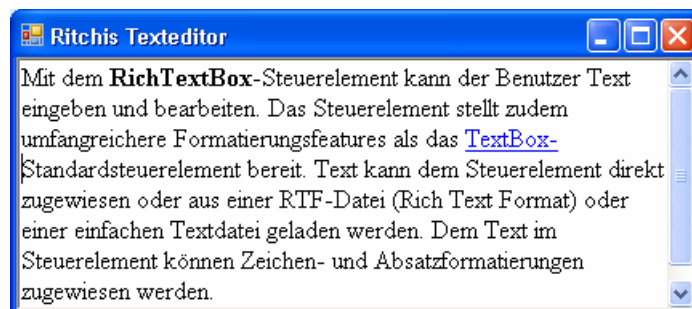
Die aktuelle Zeit liefert im Beispiel eine Instanz der Struktur **DateTime**, welche über die statische Eigenschaft **Now** dieser Struktur angesprochen wird. Es stehen auch Methoden zur benutzerfreundlichen Zeitanzeige zur Verfügung, von denen im Beispielprogramm **ToLongTimeString()** zum Einsatz kommt:



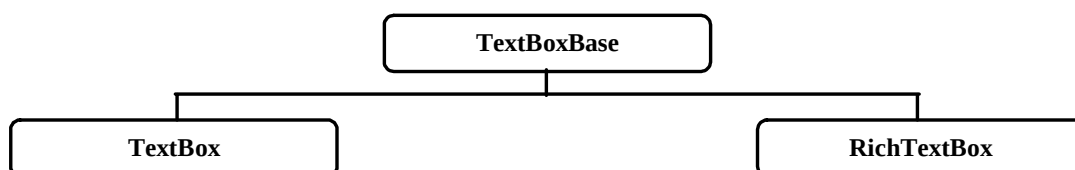
Wir haben die **Now**-Instanz der **DateTime**-Struktur schon mehrfach zur Zeitmessung verwendet und dabei per **Ticks**-Eigenschaft die Anzahl der 100-Nanosekunden-Intervalle seit dem 1. Januar 1, 00:00:00 Uhr, abgerufen (10^9 Nanosekunden = 1 Sekunde).

15.6 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 8.4.3 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 Undo-Funktion beherrscht, muss man erstaunlich wenig Aufwand betreiben:

```
using System;
using System.Windows.Forms;

class RichTextBoxEinfach : Form {
    public RichTextBoxEinfach() {
        Text = "Ritchis Texteditor (einfach)";
        RichTextBox editor = new RichTextBox();
        editor.Dock = DockStyle.Fill;
        Controls.Add(editor);
    }

    static void Main() {
        Application.Run(new RichTextBoxEinfach());
    }
}
```

Mit dem Wert **DockStyle.Fill** für die **Dock**-Eigenschaft dafür gesorgt wird, dass die **RichTextBox**-Komponente den gesamten Klientenbereich des Formulars belegt.

Das Einfüllen des obigen Beispieltextes gelingt mit dem Minimaleditor 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 Menüs

Wir erweitern nun das in Abschnitt 15.6 begonnene Editor-Projekt um ein Menü:



Die im .NET-Framework 2.0 hinzu gekommenen Klassen **MenuStrip** und **ContextMenuStrip** mit leicht erweiterter Funktionalität (insbesondere bei der Positionierung) können aus Zeitgründen leider nicht behandelt werden (siehe z.B. Kühnel 2006, Kap. 18).

16.1 Erbschaftsangelegenheiten und wichtige Begriffe

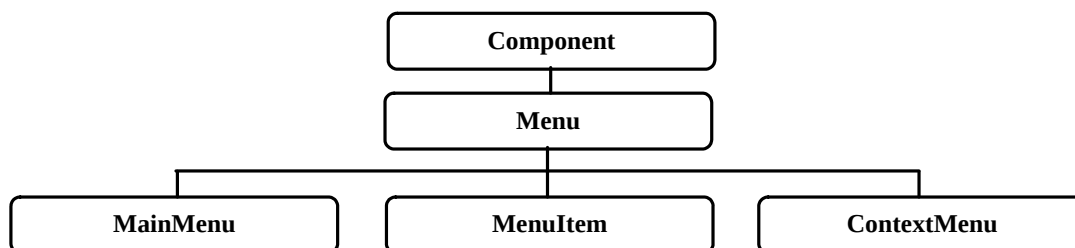
Das Hauptmenü (die Menüleiste) eines Formulars realisiert man durch ein Objekt der Klasse **MainMenu**, das der **Form**-Eigenschaft **Menu** zugewiesen wird:

```
Menu = new MainMenu();
```

Mit den folgenden Anweisungen werden die **MenuItem**-Komponenten mitDatei, mitExtras und mitHilfe erzeugt und in das Hauptmenü des Formulars eingefügt:

```
MenuItem mitDatei, mitExtras, mitHilfe;  
  
mitDatei = Menu.MenuItems.Add("&Datei");  
mitExtras = Menu.MenuItems.Add("&Extras");  
mitHilfe = Menu.MenuItems.Add("&Hilfe");
```

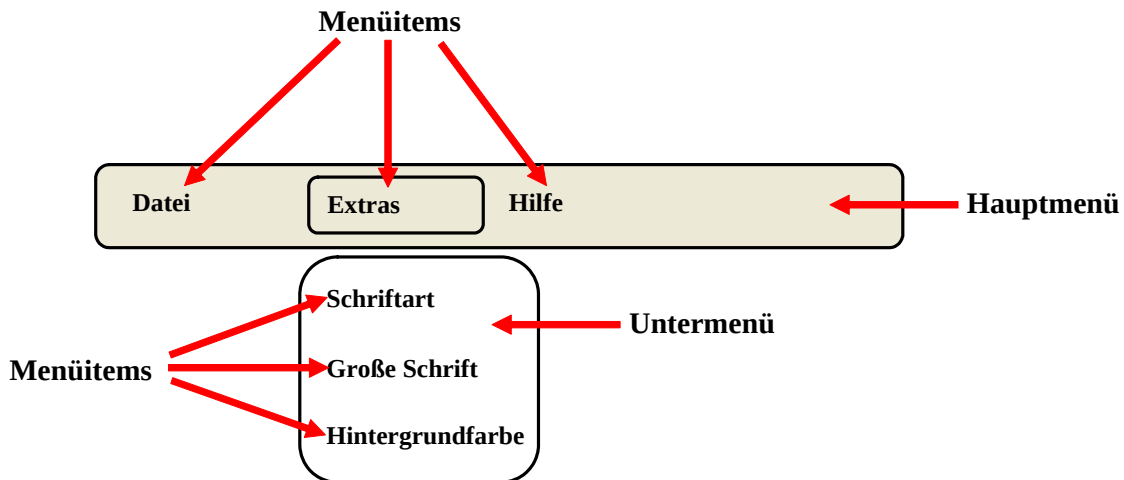
Vor der weiteren Beschäftigung mit dem Quellcode des Beispiels sollten wir uns um einige Erbschaftsangelegenheiten kümmern und wichtige Begriffe klären. Die Klassen **MainMenu** und **MenuItem** stammen von der gemeinsamen (abstrakten) Basisklasse **Menu** ab:



Bemerkenswert ist, dass **Menu** nicht von **Control** abstammt, 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** (bitte den Plural beachten), 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.

16.2 Menüitems erzeugen

Statt wie in obigen Beispielen **MenuItem**-Objekte implizit durch einen Aufruf der **Menu**-Methode **Add()** zu erzeugen, kann man sie (mit leicht erweiterter Gestaltungsmöglichkeit) auch über einen expliziten Konstruktor-Aufruf erstellen, z.B.:

```
MenuItem mitInfo;

mitInfo = new MenuItem("&Info",
                      new EventHandler(MitInfoOnClick),
                      Shortcut.F1);
mitHilfe.MenuItems.Add(mitInfo);
```

Hier wird neben dem Item-**Text** auch ein **EventHandler** für das **Click**-Ereignis und eine so genannte **Zugriffstaste** vereinbart, wobei für die letztgenannte Option die Enumeration **Shortcut** eine reichhaltige Auswahl an Bezeichnungen für Tasten(kombinationen) bietet.

Allmählich wird es Zeit, die Bedeutung des (optionalen) &-Zeichens in den **MenuItem**-Beschriftungen zu erklären. Es veranlasst, dass im laufenden Programm ...

- (spätestens nach Betätigen der **Alt**-Taste) der dem &-Zeichen folgende Buchstabe einer **MenuItem**-Beschriftung durch Unterstreichen hervorgehoben wird,
- die Taste zum markierten Buchstaben, nach der **Alt**-Vorschalttaste gedrückt, das **Click**-Ereignis zum Menüitem auslöst, wobei ggf. das zugehörige Untermenü erscheint.

In obigem 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 Zugriffstasten(kombination) zu einem Item erscheint im aufgeklappten Menü, sofern seine Eigenschaft **ShowShortcut** den voreingestellten Wert **true** besitzt, z.B.



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, mitAktFont;

mitFont = mitExtras.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;
mitAktFont = 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 `mitAktFont` wird der Quellcode zum Umschalten zwischen den Schriftarten vereinfacht (siehe unten).

Durch ein Menüitem mit Bindestrich als Wert der **Text**-Eigenschaft fügt man eine horizontale Trennlinie in ein Untermenü ein, z.B.:

```
mitExtras.MenuItems.Add("-");
```

16.3 Ereignisse und Eigenschaften von Menüitems

Soll die Wahl eines Menüitems, das kein Untermenü besitzt, einen Effekt haben, muss eine geeignete Behandlungsmethode bei seinem **Click**-Ereignis registriert werden. Hier wird das Menüitem **Datei > Beenden** erzeugt

```
mitDatei.MenuItems.Add(new MenuItem("&Beenden",
    new EventHandler(MitExitOnClick),
    Shortcut.AltF4));
```

und mit der folgenden Methode verknüpft, die das Formular und somit die Anwendung schließt:

```
protected void MitExitOnClick(object sender, EventArgs e) {
    Close();
}
```

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) {
    mitAktFont.Checked = false;
    mitAktFont = (MenuItem)sender;
    mitAktFont.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 Platz sparend 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 **mitAktFont** zeigt stets auf das aktuell markierte Schriftartenitem.
- In **MitFontOnClick()** wird zunächst die aktuelle (alte) Markierung aufgehoben. Dann erhält **mitAktFont** 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.:

```
protected void MitEditOnPopup(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;
    }
}
```

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 Zugriffstastenkombination „ohne“ zugehöriges Menüitem mit einer Behandlungsmethode zu verbinden, setzt man die **Visible**-Eigenschaft des Menüitems auf den Wert **false**.

Den vollständigen Quellcode zum aktuellen Entwicklungsstand des Beispielsprogramms finden Sie im Ordner

...\BspUeb\WinForms\Menüs\Hauptmenü und Menüitems

16.4 Kontextmenüs

Über die **Control**-Eigenschaft **ContextMenu** kann man jedem Steuerelement ein Kontextmenü verpassen, das sich vom Benutzer über einen rechten Mausklick hervorlocken lässt. Unser Editor erhält durch die folgende Erweiterung seines Fensterkonstruktors ein Kontextmenü zum Einstellen der Hintergrundfarbe:

```
EventHandler eh = new EventHandler(ContextMitOnClick);
MenuItemEx[] mie = {
    new MenuItemEx(Color.White, "Weiss", eh),
    new MenuItemEx(Color.AliceBlue, "Aliceblau", eh),
    new MenuItemEx(Color.AntiqueWhite, "Altweiss", eh),
    new MenuItemEx(Color.Azure, "Azur", eh),
    new MenuItemEx(Color.Beige, "Beige", eh),
    new MenuItemEx(Color.FloralWhite, "Blumenweiss", eh),
    new MenuItemEx(Color.LightGreen, "Blassgrün", eh),
    new MenuItemEx(Color.Silver, "Siber", eh),};
for (int i = 0; i < mie.Length; i++)
    mie[i].RadioCheck = true;
mitAktColor = mie[0];
mitAktColor.Checked = true;
editor.ContextMenu = new ContextMenu(mie);
```

Es kommen einige bereits bekannte Techniken zum Einsatz:

- Alle Menüitems werden vom selben **Click**-Handler versorgt.
- Weil es sich um „Optionsfeld-Items“ handelt, erhält ihre **RadioCheck**-Eigenschaft den Wert **true**. Zur Vereinfachung der Markierungsverwaltung wird die zusätzliche MenuItem-Referenzvariable **mitAktColor** eingesetzt.

Das Ergebnis:



Dem **ContextMenu**-Konstruktor wird ein Array mit Menüitems übergeben, wobei es sich um Objekte einer aus **MenuItem** abgeleiteten Klasse handelt:

```
class MenuItemEx : MenuItem {
    public object Value;
    public MenuItemEx(object val, string text, EventHandler eh)
        : base(text, eh) {
        Value = val;
    }
}
```

Es kommt eine öffentliche Instanzvariable namens `Value` vom Typ **object** hinzu, so dass sich ein `MenuItemEx`-Objekt seine Bedeutung merken kann (z.B. die mit diesem Item gewählte Farbe). Ein passender Konstruktor unterstützt die zusätzliche Instanzvariable.

Im universellen **Click**-Handler kann die neue Editor-Hintergrundfarbe direkt aus dem markierten Item abgelesen werden:

```
protected void ContextMitOnClick(object sender, EventArgs e) {
    mitAktColor.Checked = false;
    mitAktColor = (MenuItemEx)sender;
    mitAktColor.Checked = true;
    editor.BackColor = (Color)mitAktColor.Value;
}
```

In analogen Fällen wurden bisher umständliche Fallunterscheidungen verwendet (z.B. in der Methode `MitFontOnClick()`, siehe Abschnitt 16.3).

Selbstverständlich erfüllt die Klasse `MenuItemEx` denselben Zweck beim Aufbau eines normalen Untermenüs. Es ist allerdings *nicht* möglich, ein Menüitem (oder einen Menüitem-Array) *doppelt* zu verwenden, z.B. in einem Untermenü *und* in einem Kontextmenü.

Über die **ContextMenu**-Methode **Show()** besteht die interessante Möglichkeit, ein Kontextmenü programmgesteuert auftauchen zu lassen, das dazu mit keinem Steuerelement über dessen **ContextMenu**-Eigenschaft verbunden sein muss. Man kann etwa in einer Mausklick-Behandlungsmethode in Abhängigkeit von der Trefferstelle ein angepasstes Kontextmenü öffnen.

Den vollständigen Quellcode zum aktuellen Entwicklungsstand des Beispielsprogramms finden Sie im Ordner

...\\BspUeb\\WinForms\\Menüs\\Kontextmenü

16.5 Übungsaufgaben zu Abschnitt 16

1) Nehmen Sie an unserem Editor-Projekt ausgehend vom Entwicklungsstand in

...\BspUeb\WinForms\Menüs\Kontextmenü

folgende Änderungen vor:

- Ergänzen Sie im Hauptmenü das Item **Bearbeiten**.
- Statten Sie das **Bearbeiten**-Untermenü folgendermaßen aus:

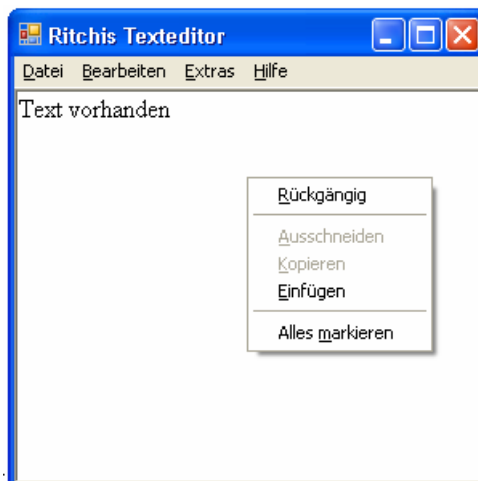


Die **Click**-Ereignishandler sind mit Hilfe der **RichTextBox**-Methoden **Undo()**, **Cut()**, **Copy()**, **Paste()** und **SelectAll()** leicht zu realisieren. Um den ausgewählten Text zu löschen, setzt man die **RichTextBox**-Eigenschaft **SelectedText** auf die leere Zeichenfolge, z.B.:

```
editor.SelectedText = "";
```

Setzen Sie die **Enabled**-Eigenschaft der Menüitems je nach Verfügbarkeit auf **true** oder **false** (siehe Bildschirmfoto).

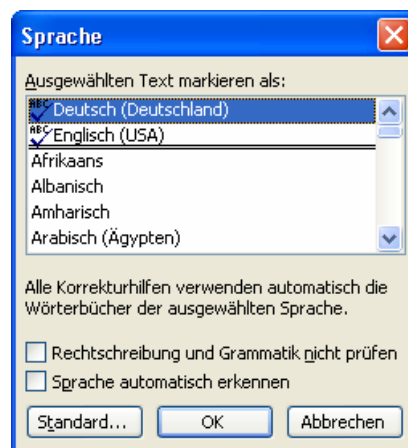
- Ersetzen Sie das in Abschnitt 16.4 erstellte Kontextmenü durch die folgende konventionelle Variante:



17 Dialogboxen

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 .NET-Standarddialoge beschränkt haben. Wir beschäftigen uns nun mit der Verwendung von selbst entworfenen Dialogboxen und erweitern anschließend unser Wissen über die Standarddialoge.

Dabei wird sich zeigen, dass *alle* Dialogboxen durch Objekte einer aus **System.Windows.Forms.Form** abgeleiteten Klasse realisiert werden, dass es also keine spezielle Klasse für Dialogboxen gibt. Der Begriff *Dialogbox* 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 Dialogboxen (zusammen mit den Nachrichtenfenstern und Werkzeugpaletten) den *sekundären Fenstern* zu (Microsoft 2007, S. 217). Typische Kennzeichen von Dialogboxen sind die unveränderliche Formulargröße und die sparsame Ausstattung der Titelzeile mit Standardschaltflächen, z.B. beim folgenden Dialog aus MS-Word 2003:



Die Initiative für den Auftritt einer Dialogbox für eine spezielle Kommunikationsaufgabe kann vom Benutzer ausgehen (z.B. durch Wahl einer Menüoption) oder vom Programm (z.B. Nachfrage vor dem Löschen einer Datei).

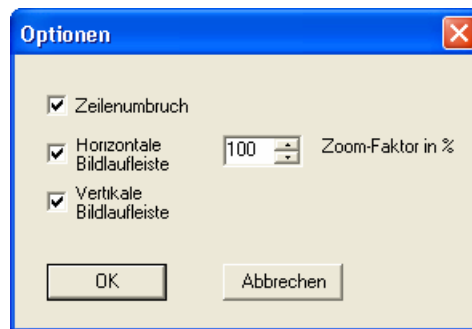
17.1 Modale Dialoge

Die meisten Dialogboxen 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. auch bei den Standarddialogen zur Font- 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 von Fenstern auffordern
- **Tick**-Ereignisse von **Timer**-Objekten

17.1.1 Konfiguration

In der Regel definiert man zu einer Dialogbox eine eigene Klasse, welche direkt oder indirekt von **Form** abstammt. Wir erstellen als Beispiel die Klasse `OptionsDialog`, die eine Optionen-Dialogbox für unseren seit Abschnitt 15.6 sukzessive aufgebauten Editor realisiert:



Folgende Einstellungen des Editors können 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**
Diese **UpDown**-Regler wirkt auf die **RichTextBox**-Eigenschaft **ZoomFactor**, die (bei einem erlaubten Wert zwischen 1/64 und 64,0) die angezeigte Größe der Schrift verändert.

Im Konstruktor einer Dialogbox 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 folgendem Wert für die Eigenschaft **FormBorderStyle** erreicht man eine fixierten Rahmen ohne Anwendungs-Symbol in der Titelzeile:

```
FormBorderStyle = FormBorderStyle.FixedDialog;
```

- Oft 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:

```
AcceptButton = btnOK;
CancelButton = btnCancel;
```

In den **Click**-Handlern dieser beiden Schalter wird in der Regel auf gleich näher zu beschreibende Weise die Dialogbox beendet.

Ansonsten bietet der **OptionsDialog**-Konstruktor nur die mittlerweile wohl vertrauten Steuerelement-Gestaltungsarbeiten:

```
public OptionsDialog() {
    Text = "Optionen";
    MaximizeBox = false;
    MinimizeBox = false;
    FormBorderStyle = FormBorderStyle.FixedDialog;
    ClientSize = new Size(310, 170);
    StartPosition = FormStartPosition.Manual;
    Location = ActiveForm.DesktopLocation + ActiveForm.Size
        - ActiveForm.ClientSize;

    Button btnOK = new Button();
    btnOK.Text = "OK";
    btnOK.Parent = this;
    btnOK.Location = new Point(20, 130);
    AcceptButton = btnOK;
    btnOK.DialogResult = DialogResult.OK;

    Button btnCancel = new Button();
    btnCancel.Text = "Abbrechen";
    btnCancel.Parent = this;
    btnCancel.Location = new Point(110, 130);
    CancelButton = btnCancel;
    btnCancel.DialogResult = DialogResult.Cancel;

    ckbxWordWrap = new CheckBox();
    ckbxWordWrap.Text = "Zeilenumbruch";
    ckbxWordWrap.Location = new Point(20, 20);
    ckbxWordWrap.Parent = this;

    ckbxHorBar = new CheckBox();
    ckbxHorBar.Text = "Horizontale Bildlaufleiste";
    ckbxHorBar.Location = new Point(20, 50);
    ckbxHorBar.Width = 100;
    ckbxHorBar.Parent = this;

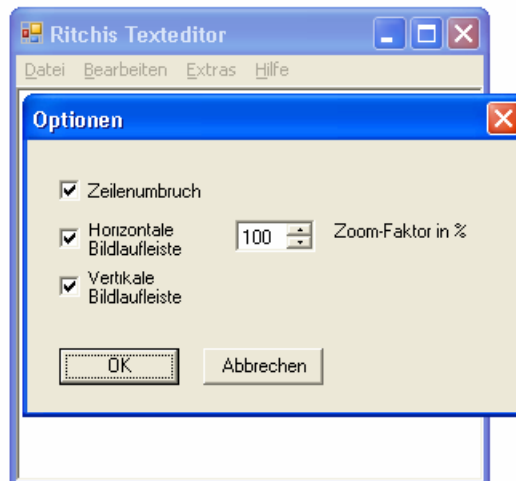
    ckbxVertBar = new CheckBox();
    ckbxVertBar.Text = "Vertikale Bildlaufleiste";
    ckbxVertBar.Location = new Point(20, 80);
    ckbxVertBar.Parent = this;

    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;
}
```

Der Optionsdialog soll ...

- 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 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, wobei über die statische **Form**-Eigenschaft **ActiveForm** ermittelt wird, welches Formular das Erstellen der Dialogbox veranlasst hat

```
Location = ActiveForm.DesktopLocation + ActiveForm.Size
          - ActiveForm.ClientSize;
```

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;
```

Ansonsten benötigt unsere OptionsDialog-Klasse für den Datenaustausch mit einer nutzenden Klasse noch öffentliche Eigenschaften mit Lese- und Schreibzugriff, z.B.:

```
public bool WordWrap {
    get {return ckbxWordWrap.Checked;}
    set {ckbxWordWrap.Checked = value;}
}
```

Zu Beginn des nächsten Abschnitts ist zu sehen, wie diese Eigenschaften vor dem Öffnen des Dialogs initialisiert und nach Beenden des Dialogs auf aktuelle Benutzerwünschte untersucht werden.

Um den Optionsdialog im Hauptfenstermenü anzubieten, wird das **Extras**-Untermenü um ein Item erweitert (vgl. Abschnitt 16):

```
mitExtras.MenuItems.Add(new MenuItem("&Optionen...",
    new EventHandler(MitOptionenOnClick)));
```

Mit drei Punkten am Ende einer Menüitem-Beschriftung kündigt man den Auftritt einer Dialogbox an. Die zugehörige **Click**-Behandlungsmethode mit dem Aufruf des Optionsdialogs wird im nächsten Abschnitt vorgestellt.

17.1.2 Auftritt und Abgang

Ein modaler Dialog wird über die **Form**-Methode **ShowDialog()** gestartet, 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. Folgende Werte können auftreten:

None	Retry
OK	Ignore
Cancel	Yes
Abort	No

In der Methode `MitOptionenOnClick()` werden nur nach Beendigung der Optionen-Dialogbox mit dem Rückgabewert **DialogResult.OK** die dortigen Eigenschaften ins Anwendungsfenster übernommen.

Über den **ShowDialog()**-Rückgabewert erfahren wir den Wert der **DialogResult**-Eigenschaft des Dialogbox-Objektes. Eine Änderung dieser Eigenschaft hat folgende Konsequenzen, wenn das zugehörige Formular als *modales* Dialogfeld angezeigt wird:

- Das Dialogfeld 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:

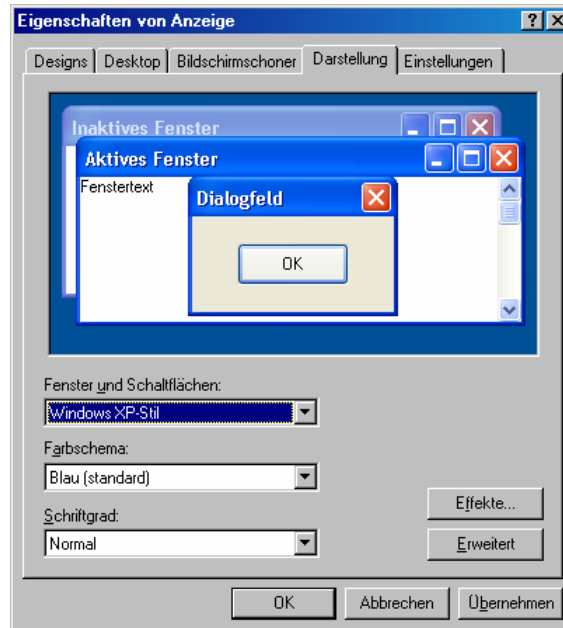
- Klickt der Benutzer auf die Schließen-Schaltfläche der Titelzeile, erhält **DialogResult** den Wert **Cancel**.
- Mit der **Button**-Eigenschaft **DialogResult** legt man einen Wert fest, den die **DialogResult**-Eigenschaft des elterlichen Formulars bei einem **Click** auf einen Schalter erhalten soll. Somit kann man sich den **Click**-Handler für eine Schaltfläche sparen, wenn lediglich ein Beenden des Dialogs mit bestimmtem **DialogResult**-Wert bezweckt ist. Im Beispiel setzt der Schalter `btnOK` den Wert **OK** und der Schalter `btnCancel` den Wert **Cancel**.

Den vollständigen Quellcode zum aktuellen Entwicklungsstand des Editors finden Sie im Ordner **...\BspUeb\WinForms\Dialogboxen\Modal\Optionendialog**

17.1.3 Übernehmen Sie keinen Aberglauben

Auch bei modalen Dialogen kann man den Benutzern den Luxus bieten, die Effekte von Einstellungsänderungen auf das Hauptfenster zu besichtigen, 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.

Streng genommen sollte es sich allerdings nicht um eine Direktaktivierung, sondern um eine *Vorschau* handeln, so dass der Dialog auf jeden Fall mit dem Schalter **Abbrechen** ohne permanente Effekte verlassen werden kann (siehe z.B. die .NET-SDK - Dokumentation zum Ereignis **FontDialog.Apply**). Microsoft hält sich nicht streng an diese Designvorgabe, wie man mit dem folgenden Dialog zur Wahl eines Stils für **Fenster und Schaltflächen** in Windows XP testen kann:



Vielleicht ist die Unsicherheit hinsichtlich der korrekten Verwendung der **Übernehmen**-Schaltfläche ein Grund dafür, dass sie in Microsoft-Programmen nur selten zum Einsatz kommt.

Bei Benutzern ist übrigens der Aberglauben verbreitet, dass ein vorhandener **Übernehmen**-Schalter auf jeden Fall vor dem Quittieren der Dialogboxen 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, Dialogbox-Effekte tatsächlich vom Schalter **Übernehmen** abhängig zu machen.

In vielen Situationen ist wohl eine mit **Vorschau** beschriftete Schaltfläche sinnvoll:

- Hier wird kein abergläubisches Verhalten angeregt.
- Den Benutzern und Entwicklern wäre klar, dass die Effekte des Schalters nur temporär sein können bzw. dürfen.

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 Dialogbox-Klasse ein öffentliches Ereignis, z.B. mit dem Namen `Apply`:

```
public event EventHandler Apply;
```
- Auf dem Dialogfeld-Formular 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.:

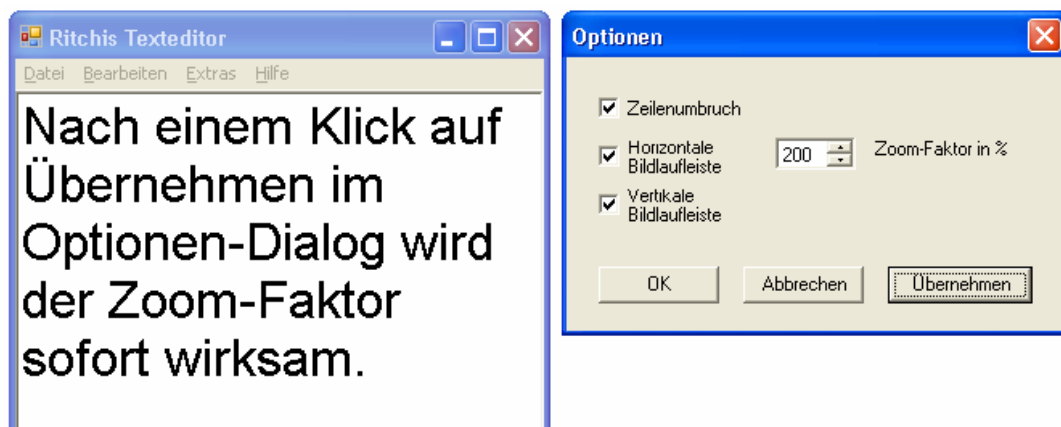
```
protected void ButtonApplyOnClick(object sender, EventArgs ea) {
    if (Apply != null)
        Apply(this, EventArgs.Empty);
}
```
- Man definiert eine Methode der Hauptfensterklasse, welche die `Apply`-Delegatentyp erfüllt und nach einem Klick auf **Übernehmen** ausgeführt werden soll, z.B.:

```
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 Dialogbox-Objektes 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` für ungültig erklärt werden, damit der **Paint**-Handler aktiv wird. Im Beispiel ist dies nicht erforderlich.

Unser Editor erlaubt nun z.B. eine bequeme Auswahl des Zoom-Faktors:



In der folgenden **Click**-Behandlungsmethode zum Optionen-Menüitem des Editors wird beim Verlassen des Dialogs mit **Abbrechen** der vorherige Zustand wiederhergestellt, sodass ein vorheriges **Übernehmen** keine permanenten Folgen hat:

```
protected void MitOptionenOnClick(object sender, EventArgs e) {
    if (optionsDialog == null) {
        optionsDialog = new OptionsDialog();
        optionsDialog.Apply += new EventHandler(OptionsDialogOnApply);
    }

    optionsDialog.ZoomFactor = editor.ZoomFactor;
    optionsDialog.ScrollBars = editor.ScrollBars;
    optionsDialog.WordWrap = editor.WordWrap;

    float OldZoomFactor = editor.ZoomFactor;
    RichTextBoxScrollBars OldScrollBars = editor.ScrollBars;
    bool OldWordWrap = editor.WordWrap;

    DialogResult result = optionsDialog.ShowDialog();
    if (result == DialogResult.OK) {
        editor.WordWrap = optionsDialog.WordWrap;
        editor.ScrollBars = optionsDialog.ScrollBars;
        editor.ZoomFactor = optionsDialog.ZoomFactor;
    } else if (result == DialogResult.Cancel) {
        editor.ZoomFactor = OldZoomFactor;
        editor.ScrollBars = OldScrollBars;
        editor.WordWrap = OldWordWrap;
    }
}
```

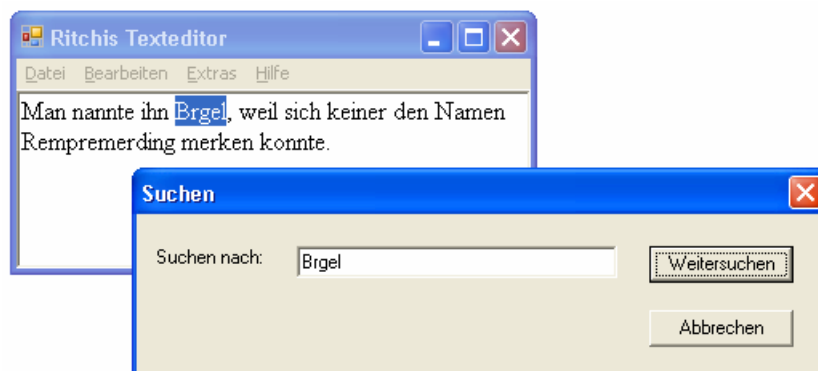
Den vollständigen Quellcode zum aktuellen Entwicklungsstand des Editors finden Sie im Ordner
...\BspUeb\WinForms\Dialogboxen\Modal\Übernehmen

17.2 Nicht-modale Dialoge

Viele Dialogboxen 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).

17.2.1 Konfiguration

Auch unser seit Abschnitt 15.6 im Aufbau befindlicher Editor soll nun mit einem einfachen Suchdialog ausgestattet werden:



Wir definieren dazu die von **Form** abstammende Klasse SuchDialog:

```
using System;
using System.Windows.Forms;
using System.Drawing;
using System.ComponentModel;
```

```

class SuchDialog : Form {
    TextBox tbSuchText;
    public event EventHandler Weitersuchen;

    public SuchDialog() {
        Text = "Suchen";
        MaximizeBox = false;
        MinimizeBox = false;
        ShowInTaskbar = false;
        FormBorderStyle = FormBorderStyle.FixedDialog;
        ClientSize = new Size(430, 100);
        StartPosition = FormStartPosition.Manual;
        Location = ActiveForm.DesktopLocation + ActiveForm.Size
            - ActiveForm.ClientSize;
        Label lbSuchen = new Label();
        lbSuchen.Text = "Suchen nach:";
        lbSuchen.Parent = this;
        lbSuchen.Location = new Point(10, 20);
        lbSuchen.Width = 80;
        tbSuchText = new TextBox();
        tbSuchText.Parent = this;
        tbSuchText.Location = new Point(100, 20);
        tbSuchText.Size = new Size(200, Font.Height);
        Button btnNext = new Button();
        btnNext.Text = "Weitersuchen";
        btnNext.Parent = this;
        btnNext.Location = new Point(320, 20);
        btnNext.Width = 90;
        AcceptButton = btnNext;
        btnNext.Click += new EventHandler(ButtonNextOnClick);
        Button btnCancel = new Button();
        btnCancel.Text = "Abbrechen";
        btnCancel.Parent = this;
        btnCancel.Location = new Point(320, 60);
        btnCancel.Width = 90;
        CancelButton = btnCancel;
        btnCancel.Click += new EventHandler(ButtonCancelOnClick);
    }

    protected void ButtonNextOnClick(object sender, EventArgs ea) {
        if (tbSuchText.Text.Length > 0 && Weitersuchen != null)
            Weitersuchen(this, EventArgs.Empty);
    }

    protected void ButtonCancelOnClick(object sender, EventArgs ea) {
        Verstecken();
    }

    protected override void OnClosing(CancelEventArgs cea) {
        cea.Cancel = true;
        Verstecken();
    }

    protected void Verstecken() {
        tbSuchText.Focus();
        Hide();
        Owner.Activate();
    }

    public string SuchText {
        get { return tbSuchText.Text; }
        set { tbSuchText.Text = value; }
    }
}

```

Weil in der Windows-Taskleiste nur Hauptfenstern erscheinen sollten, wird die Eigenschaft **ShowInTaskbar** auf den Wert **false** gesetzt:

```
ShowInTaskbar = false;
```

Wie der in Abschnitt 17.1 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 SuchDialog-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 17.2.2. Im Vergleich zum modalen Optionsdialog müssen wir beim nicht-modalen Suchdialog also mehr Aufwand investieren, damit während einer Editor-Sitzung nur ein einziges SuchDialog-Objekt verwendet wird.
- Um die Position beim ersten Auftritt festzulegen, wird im SuchDialog-Konstruktor dieselbe Technik gewählt wie im OptionsDialog-Konstruktor (siehe Abschnitt 17.1.1).

Um den Suchdialog im Hauptfenstermenü anzubieten, wird ein **Bearbeiten**-Untermenü mit dem Item **Suchen** ergänzt (vgl. Abschnitt 16):

```
MenuItem mitBearbeiten;

mitBearbeiten = Menu.MenuItems.Add("&Bearbeiten");
mitBearbeiten.MenuItems.Add(new MenuItem("&Suchen",
    new EventHandler(MitSuchenOnClick),
    Shortcut.CtrlF));
```

Im der zugehörigen **Click**-Behandlungsmethode wird nötigenfalls ein Objekt der Klasse SuchDialog erzeugt:

```
protected void MitSuchenOnClick(object sender, EventArgs e) {
    if (editor.Text.Length == 0)
        return;
    if (suchDialog == null) {
        suchDialog = new SuchDialog();
        suchDialog.Owner = this;
        suchDialog.Weitersuchen += new EventHandler(SuchDialogOnWeitersuchen);
    }
    suchDialog.SuchText = suchText;
    suchDialog.Show();
}
```

Hier enthaltene Details zum Auftritt des Suchdialogs (siehe Abschnitt 17.2.2) und zu seiner Kommunikation mit dem Hauptfenster (siehe Abschnitt 17.2.3) werden später erläutert.

Bei einer nicht-modalen Dialogbox ist es in der Regel erwünscht, dass sie sich stets *über* dem Anwendungsfenster befindet, also von diesem nicht abgedeckt werden kann. Außerdem sollte die Dialogbox beim Minimieren des Anwendungsfensters ebenfalls vom Desktop verschwinden. Um diese Verhaltensweisen anzufordern, setzt man die **Owner**-Eigenschaft des Dialogs auf das Hauptfenster, z.B. in der **Click**-Behandlungsmethode zum Editor-Menüitem **Suchen**:

```
suchDialog.Owner = this;
```

Im Konstruktor des Hauptfensters wird mit folgender Anweisung

```
editor.HideSelection = false;
```

dafür gesorgt, dass markierter Text auch dann hervorgehoben ist, wenn ein anderes Fenster (speziell: der Suchdialog) aktiv ist. Beim Durchsuchen des Textes soll nämlich eine Trefferstelle durch Markieren gekennzeichnet werden (siehe oben).

17.2.2 Auftritt und Abgang

Anstelle der zum Starten von modalen Dialogen erforderlichen **Form**-Methode **ShowDialog()**, welche die Kontrolle erst beim Beenden der Dialogbox zurückgibt, wird zum Öffnen einer nicht-modalen Dialogbox die Methode **Show()** verwendet, z.B. in der **Click**-Behandlungsmethode zum Editor-Menüitem **Suchen**:

```
suchDialog.Show();
```

Die zum Beenden von *modalen* Dialogen wichtige Eigenschaft **DialogResult** (siehe Abschnitt 17.1.2) spielt bei nicht-modalen Dialogen *keine* Rolle. Letztere werden häufig über die **Form**-Methode **Close()** beendet, z.B.:

```
protected void ButtonCancelOnClick(object sender, EventArgs ea) {
    Close();
}
```

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. Auch bei einem Mausklick auf die Schließen-Schaltfläche in der Titelzeile einer nicht-modalen Dialogbox wird per Voreinstellung die **Close()**-Methode ausgeführt.

In unserem Editor soll aber nur *ein* Objekt der Klasse `Suchdialog` (samt zugehörigem Fenster) verwendet werden. Neben einem kleinen (vermutlich nicht spürbaren) Spareffekt gewinnen wir den Vorteil, dass die Fensterposition der Dialogbox ohne Aufwand für das Hauptfenster zwischen zwei Auftritten erhalten bleibt. Daher *verstecken* wir das Suchfenster, wenn es vorübergehend nicht mehr benötigt wird, statt es zu schließen:

```
protected void ButtonCancelOnClick(object sender, EventArgs ea) {
    Verstecken();
}
protected void Verstecken() {
    tbSuchText.Focus();
    Hide();
    Owner.Activate();
}
```

Mit

```
Owner.Activate();
```

wird sichergestellt, dass nach dem Verstecken der Dialogbox deren Besitzer (also unser Editor) das aktive Programm ist.

Damit das Dialogfenster auch bei einem Mausklick auf das Schließkreuz ihrer Titelzeile nicht zu Schaden kommt, erweitern wir die Klassendefinition um folgende **OnClosing()**-Überschreibung:

```
protected override void OnClosing(CancelEventArgs cea) {
    cea.Cancel = true;
    Verstecken();
}
```

Mit dem Wert **true** für die **CancelEventArgs**-Eigenschaft **Cancel** verhindert man das Schließen des Fensters durch die CLR. Wie Sie bereits wissen, werden beim Schließen des Hauptfensters automatisch alle übrigen Fenster einer Anwendung geschlossen (vgl. Abschnitt 8.2).

17.2.3 Kommunikation mit dem Anwendungsfenster

Während eine modale Dialogbox nur bei Anwesenheit einer **Übernehmen**-Schaltfläche vor ihrem Ende Information zum Anwendungsfenster übertragen muss, ist bei einem nicht-modalen Dialog auf jeden Fall ein solcher Informationskanal erforderlich, der meist über öffentliche Ereignisse und Eigenschaften der Dialogbox-Klasse realisiert wird. In Abschnitt 17.2.1 war die **Click**-Behandlungsmethode zum Editor-Menüitem **Suchen** zu sehen. Dort wird nötigenfalls ein `SuchDialog`-

Objekt erzeugt und die Editor-Methode SuchDialogOnWeitersuchen() beim Weitersuchen-Ereignis des Suchdialogs registriert:

```
suchDialog.Weitersuchen += new EventHandler(SuchDialogOnWeitersuchen);
```

Mit dem Ereignis weitersuchen soll ein SuchDialog-Objekt alle Interessenten darüber informieren, dass der Benutzer eine Suchzeichenfolge mit dem Schalter **Weitersuchen** quittiert hat.

Folglich muss die Klasse SuchDialog das Ereignis weitersuchen definieren und in der **Click**-Behandlungsmethode der **Weitersuchen**-Schaltfläche auslösen:

```
public event EventHandler weitersuchen;

protected void ButtonNextOnClick(object sender, EventArgs ea) {
    if (tbSuchText.Text.Length > 0 && weitersuchen != null)
        weitersuchen(this, EventArgs.Empty);
}
```

Damit der Editor die Suchzeichenfolge in Erfahrung bringen und bei anderer Gelegenheit auch setzen kann, stellt die Klasse SuchDialog eine öffentliche Eigenschaft zur Verfügung:

```
public string SuchText {
    get { return tbSuchText.Text; }
    set { tbSuchText.Text = value; }
}
```

In seiner eben schon erwähnten Methode SuchDialogOnWeitersuchen() erfährt der Editor die Suchzeichenfolge und ruft seine Methode weitersuchen():

```
protected void SuchDialogOnWeitersuchen(object sender, EventArgs e) {
    suchText = ((SuchDialog)sender).SuchText;
    weitersuchen();
}
```

In der Methode weitersuchen():

```
protected void weitersuchen() {
    int start = editor.SelectionStart + editor.SelectionLength;
    int ende = editor.TextLength;
    bool anfangRelevant = (start > 0);
    int fundStelle;
    StarteSuche:
    if ((fundStelle =
        editor.Find(suchText, start, ende, RichTextBoxFinds.None)) >= 0) {
        editor.SelectionStart = fundStelle;
        editor.SelectionLength = suchText.Length;
        return;
    }
    if (!anfangRelevant) {
        MessageBox.Show("'" + suchText + "' kann nicht gefunden werden.",
            suchDialog.Text);
        return;
    } else {
        anfangRelevant = false;
        if (MessageBox.Show("Soll die Suche am Textanfang fortgesetzt werden?",
            suchDialog.Text, MessageBoxButtons.YesNo,
            MessageBoxIcon.Question,
            MessageBoxDefaultButton.Button1) == DialogResult.Yes) {
            ende = start + suchText.Length - 1;
            start = 0;
            goto StarteSuche;
        }
    }
}
```

wird die eigentliche Sucharbeit von der **RichTextBox**-Methode **Find** verrichtet:

```
editor.Find(suchText, start, ende, RichTextBoxFinds.None))
```

Mit dem Parameterwert **RichTextBoxFinds.None** wird die Differenzierung zwischen Groß- und Kleinbuchstaben abgeschaltet.

Dass die Suche hinter dem Ende der aktuellen Markierung startet (ab `editor.SelectionStart + editor.SelectionLength`), hat einen einfachen Grund. Eine Trefferstelle wird markiert,

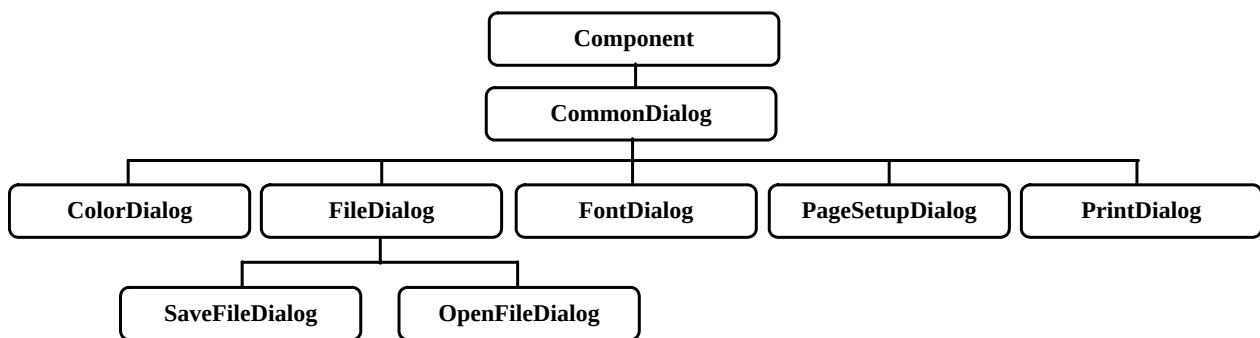
```
editor.SelectionStart = fundStelle;
editor.SelectionLength = suchText.Length;
```

und bei einer Suche ab *Markierungsanfang* würde der Algorithmus nicht über den ersten Treffer hinauskommen.

Den vollständigen Quellcode zum aktuellen Entwicklungsstand des Editors finden Sie im Ordner **...\BspUeb\WinForms\Dialogboxen\Nicht-modal**

17.3 Standarddialoge

Im Sinne einer möglichst konsistenten Bedienoberfläche für alle Programme bietet Windows seinen Benutzern (und Entwicklern) seit der Version 3.1 für Routineaufgaben wie das Öffnen einer Datei oder die Auswahl einer Farbe Standarddialoge an, die natürlich auch in der WinForms-Bibliothek verfügbar sind:



Wir haben die WinForms-Standarddialoge zur Font- 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 Sichern bzw. Öffnen von Dateien kennen. Mit den Klassen **PageSetupDialog** und **PrintDialog** beschäftigen wir uns im Abschnitt über das Drucken.

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 15.6 allmählich zu einem praxistauglichen Programm entwickeln.

17.3.1 FontDialog

Ist mit dem mehrfachen Einsatz eines Schriftauswahldialogs zu rechnen, sollte man ressourcenschonend ein *einziges* **FontDialog**-Objekt per Referenz-Instanzvariable verwalten, z.B. folgendermaßen im Konstruktor des Hauptfensters deklariert:

```
FontDialog fontDialog;
```

Vor dem ersten Einsatz des Schriftauswahldialogs werden mit den folgenden Anweisungen


```
fontDialog.Font = editor.Font;
fontDialog.ShowColor = true;
fontDialog.Color = editor.ForeColor;
fontDialog.ShowApply = true;
fontDialog.Apply += new EventHandler(FontDialogOnApply);
```

etliche **FontDialog**-Eigenschaften angepasst:

- Per **Font**-Eigenschaft sorgen wir dafür, dass der Dialog mit der aktuellen Schriftart der **RichTextBox**-Komponente startet.
- Über die **ShowColor**-Eigenschaft wird für eine Möglichkeit zur Auswahl der Schriftfarbe gesorgt (mit der aktuellen Editor-Schriftfarbe als **Color**-Startwert).
- Erhält die **ShowApply**-Eigenschaft den Wert **true**, dann erscheint im Standarddialog die Schaltfläche **Übernehmen** (vgl. Abschnitt 17.1.3).
- Damit sind wir verpflichtet, eine geeignete Behandlungsmethode beim **Apply**-Ereignis des Standarddialogs zu registrieren, weil sich die Benutzer ansonsten zurecht über die Nutzlosigkeit des **Übernehmen**-Schalters beklagen werden.

Weitere Gestaltungsmöglichkeiten:

- 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** per **ShowDialog()** zum Einsatz gebracht. Nötigenfalls wird es zuvor erzeugt und initialisiert:

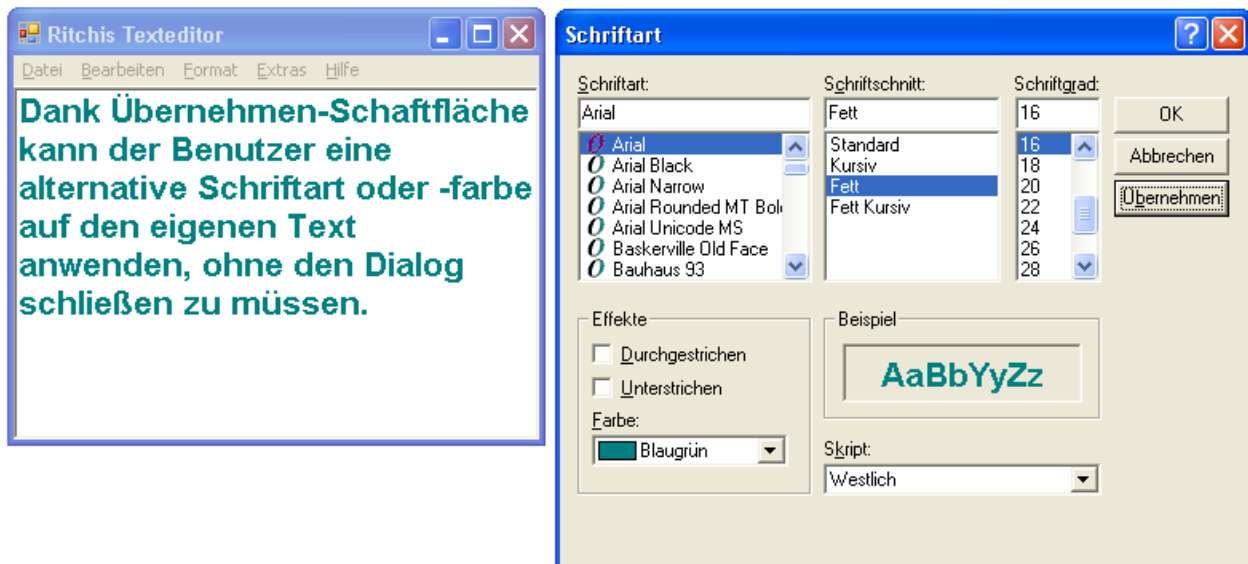
```
protected void MitFontOnClick(object sender, EventArgs e) {
    if (fontDialog == null) {
        fontDialog = new FontDialog();
        fontDialog.Font = editor.Font;
        fontDialog.ShowColor = true;
        fontDialog.Color = editor.ForeColor;
        fontDialog.ShowApply = true;
        fontDialog.Apply += new EventHandler(FontDialogOnApply);
    }

    Font oldFont = editor.Font;
    Color oldForeColor = editor.ForeColor;

    DialogResult result;
    result = fontDialog.ShowDialog();
    if (result == DialogResult.OK) {
        editor.Font = fontDialog.Font;
        editor.ForeColor = fontDialog.Color;
    } else if (result == DialogResult.Cancel) {
        editor.Font = oldFont;
        editor.ForeColor = oldForeColor;
    }
}
```

Nur beim Rückgabewert **DialogResult.OK** werden die Eigenschaften **Font** und **Color** vom Standarddialog in die passenden Hauptfenster-Eigenschaften übernommen. Beim Rückgabewert **DialogResult.Cancel** werden die (eventuell per **Übernehmen** veränderten) Hauptfenster-Eigenschaften auf ihre vorherigen Werte zurückgesetzt (vgl. Abschnitt 17.1.3).

Dank **Übernehmen**-Schaltfläche kann der Benutzer eine alternative Schriftart oder -farbe auf den eigenen Text anwenden, ohne den Dialog schließen zu müssen:



Der **Apply**-Ereignishandler übernimmt die Eigenschaftswerte vom Standarddialog:

```
protected void FontDialogOnApply(object sender, EventArgs e) {
    editor.Font = fontDialog.Font;
    editor.ForeColor = fontDialog.Color;
}
```

Wie Sie bereits wissen, werden die **Paint**-Handler des Hauptfensters auch dann aufgerufen, wenn andere Ereignisse durch den modalen Standarddialog blockiert sind.

Die Schriftart-Menüitems in früheren Editor-Versionen dienten hauptsächlich dazu, Menüoptionen zu demonstrieren. Für die nun integrierte Standardlösung wählen wir auch den üblichen Ort im Menü: **Format > Schriftart**:

```
mitFormat.MenuItems.Add(new MenuItem("&Schriftart...", MitFontOnClick));
```

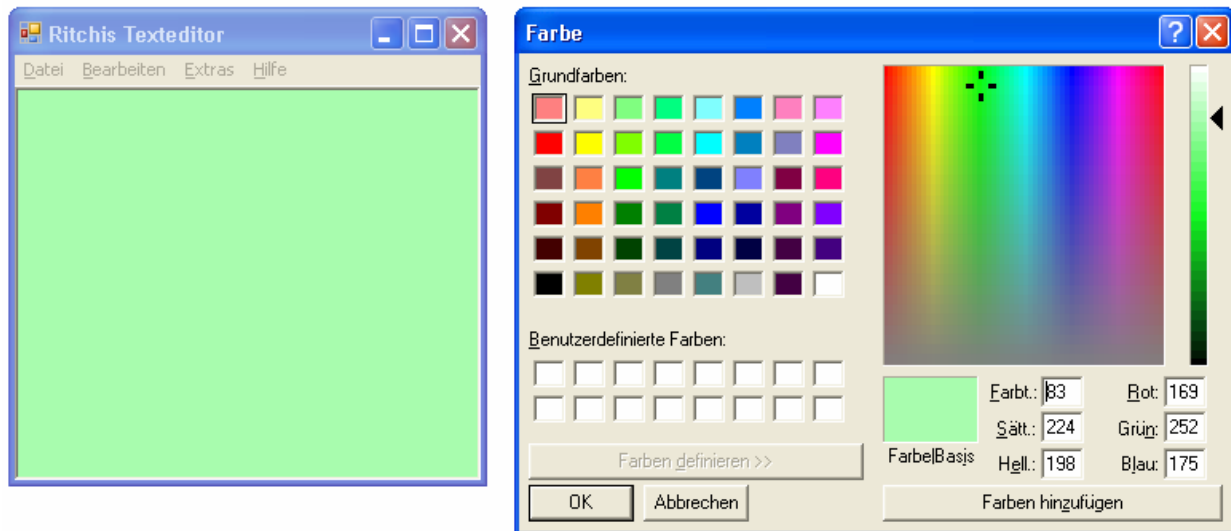
Beim Speichern im RTF-Format (siehe Abschnitt 17.3.3) wird die eingestellte Schriftart (**RichTextBox**-Eigenschaft **Text**) zum Bestandteil des Dokuments, so dass ein **Format**-Untermenü mit **Schriftart**-Item sinnvoll ist.

17.3.2 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 zum Einsatz kommt, z.B. folgendermaßen im Konstruktor des Hauptfensters deklariert:

```
ColorDialog colorDialog;
```

Weil die Option zum Mischen benutzerdefinierter Farben eingeräumt werden soll, spricht neben dem Ressourcen-Spareffekt ein noch wichtiger Grund für die Verwendung eines *einzigsten* **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 Dialogobjektes. Soll das Gedächtnis die aktuelle Sitzung überdauern, ist allerdings ein Speichern des Farb-Arrays (z.B. in der Windows-Registry) 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 registriert werden muss

Eine **Übernehmen**-Schaltfläche wird vom Farbwahl-Standarddialog *nicht* unterstützt.

In unserem Editor wird das **ColorDialog**-Objekt in der **Click**-Behandlungsmethode zum Menüitem **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;
    }
    Color oldColor = editor.Color;
    DialogResult result;
    result = colorDialog.ShowDialog();
    if (result == DialogResult.OK) {
        editor.Color = colorDialog.Color;
    } else if (result == DialogResult.Cancel) {
        editor.Color = oldColor;
    }
}
```

17.3.3 SaveFileDialog

Vor dem (ersten) Einsatz eines Objekts aus der Klasse **SaveFileDialog** zum Ermitteln einer Zielfeld für das Speichern wird man in der Regel über seine **Filter**-Eigenschaft festlegen, zwischen wel-

chen Dateitypen der Benutzer wählen kann. In unserem Editor-Beispiel sollen vorläufig Text- und RTF-Dateien unterstützt werden:

```
sfDialog.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:

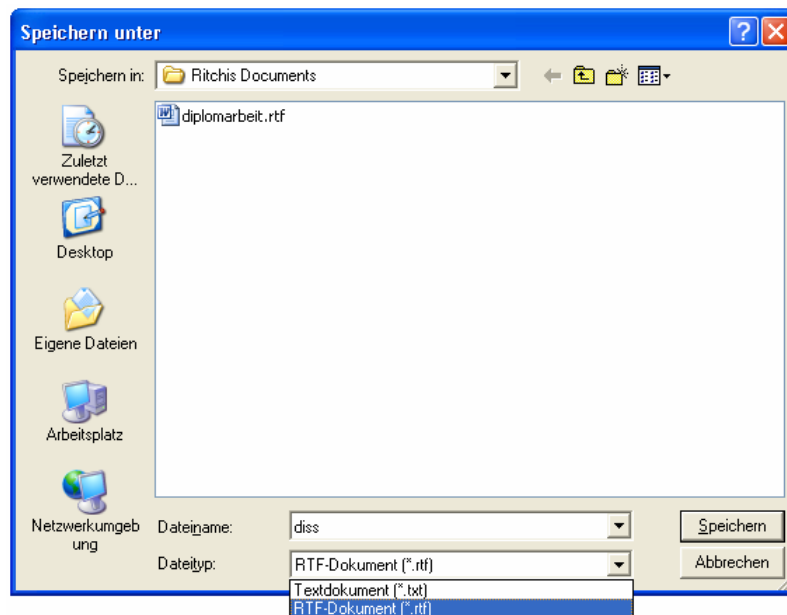
- Mit der Eigenschaft **AddExtension** (Voreinstellung **true**) legt man fest, ob an einen Dateinamen nötigenfalls die zum aktuell gewählten Dateityp gehörige Namenserweiterung automatisch angehängt werden soll.
- Per Voreinstellung ändert der Benutzer durch sein Navigationsverhalten im **SaveFileDialog** 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 Zielfile erstellt werden soll.
- Hat die Eigenschaft **OverwritePrompt** den voreingestellten Wert **true**, wird der Benutzer ggf. gefragt, ob eine bereits vorhandene Zielfile überschrieben werden soll.
- Per **ShowHelp** (Voreinstellung **false**) legt man fest, ob ein **Hilfe**-Schalter erscheinen soll, für den ggf. aber auch eine Behandlungsroutine 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 geänderter Text vorhanden ist, 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 && editor.Text.Length > 0)))
        return;
    if (fileName == null || sas) {
        if (sfDialog.ShowDialog() != DialogResult.OK)
            return;
        fileName = sfDialog.FileName;
        fileType = sfDialog.FilterIndex;
        Text = appName + " - " + Path.GetFileName(fileName);
    }
    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 {
        MessageBox.Show("Fehler beim Speichern", 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** 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 mit dem Attribut **STAThread** dekoriert werden (vgl. Abschnitt 10):

```
[STAThread]
static void Main() {
    Application.Run(new RitchisTexteditor());
}
```

Ein **RichTextBox**-Steuerelement unterstützt in seinen Methoden **SaveFile()** und **LoadFile()** (siehe unten) die folgende, durch Werte der Enumeration **RichTextBoxStreamType** identifizierten Dateiformate, wobei sich unser Editor vorläufig auf **PlainText** und **RichText** beschränkt:

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.

17.3.4 OpenFileDialog

Mit einem Standarddialog aus der Klasse **OpenFileDialog** erfragt man beim Benutzer den Namen einer vorhandenen Datei zum Öffnen eines Dokuments. Zur Vorbereitung eines **OpenFileDialog**-Aufrufs ist analog zum Vorgehen bei **SaveFileDialog** (siehe Abschnitt 17.3.3) in der Regel ein Filter-String mit den unterstützten Dateitypen zu erstellen, z.B.:

```
ofDialog.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 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 (!TextSicher("Wollen Sie trotzdem ein neues Dokument öffnen?"))
        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 {
            MessageBox.Show("Fehler beim Öffnen", Text);
        } finally {
            fs.Close();
        }
    }
}
```

Zunächst wird geprüft, ob im Editor ungesicherter Text beim Öffnen eines anderen Dokuments gefährdet ist. Diese Prüfung gelingt in der Methode **TextSicher()** mit Hilfe der **RichTextBox**-Eigenschaft **Modified** recht bequem:

```
protected bool TextSicher(string frage) {
    if (editor.Modified && editor.Text.Length > 0)
        if (MessageBox.Show("Der Text wurde geändert oder formatiert." + frage,
                             appName, MessageBoxButtons.YesNoCancel,
                             MessageBoxIcon.Exclamation,
                             MessageBoxDefaultButton.Button2) !=
            DialogResult.Yes)
            return false;
        else
            return true;
    return true;
}
```

Gegen Datenverlust beim Beenden unseres Editors hilft die folgende Überschreibung der **Form-Methode OnClosing**:

```
protected override void OnClosing(System.ComponentModel.CancelEventArgs cea){
    if (!TextSicher("Wollen Sie das Programm trotzdem beenden?"))
        cea.Cancel = true;
}
```

Damit die **OpenFileDialog-Methode ShowDialog()** im Debug-Modus ausgeführt werden kann, muss die **Main()-Methode** mit dem Attribut **STAThread** dekoriert werden (vgl. Abschnitt 10):

```
[STAThread]
static void Main() {
    Application.Run(new RitchisTexteditor());
}
```

Den vollständigen Quellcode zum aktuellen Entwicklungsstand des Editors finden Sie im Ordner **...\BspUeb\WinForms\Dialogboxen\Standardddialoge**

17.4 Übungsaufgaben zu Abschnitt 17

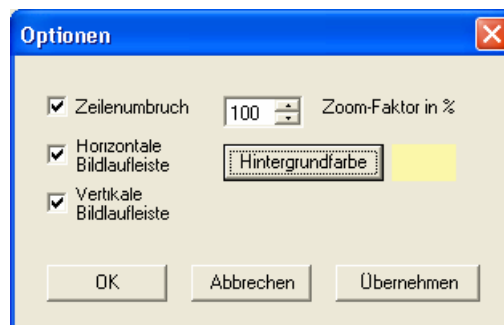
1) Erweitern Sie unseren Editor ausgehend vom Entwicklungsstand in

...\BspUeb\WinForms\Dialogboxen\Standardddialoge

um das **Bearbeiten-Menüitem Weitersuchen** (mit Zugriffstaste **F3**), wobei die letzte Suchzeichenfolge erneut verwendet werden soll.

Außerdem fehlt im **Datei-Menü** noch das Standarditem **Neu**.

2) Die Modifikation der Editor-Hintergrundfarbe sollte eigentlich über den Optionsdialog erfolgen, z.B.:



Nehmen Sie an Ihrer Lösung zu Aufgabe 1 die nötigen Änderungen vor.

18 Einstellungen in der Windows-Registry speichern

18.1 Die Windows-Registrierungsdatenbank

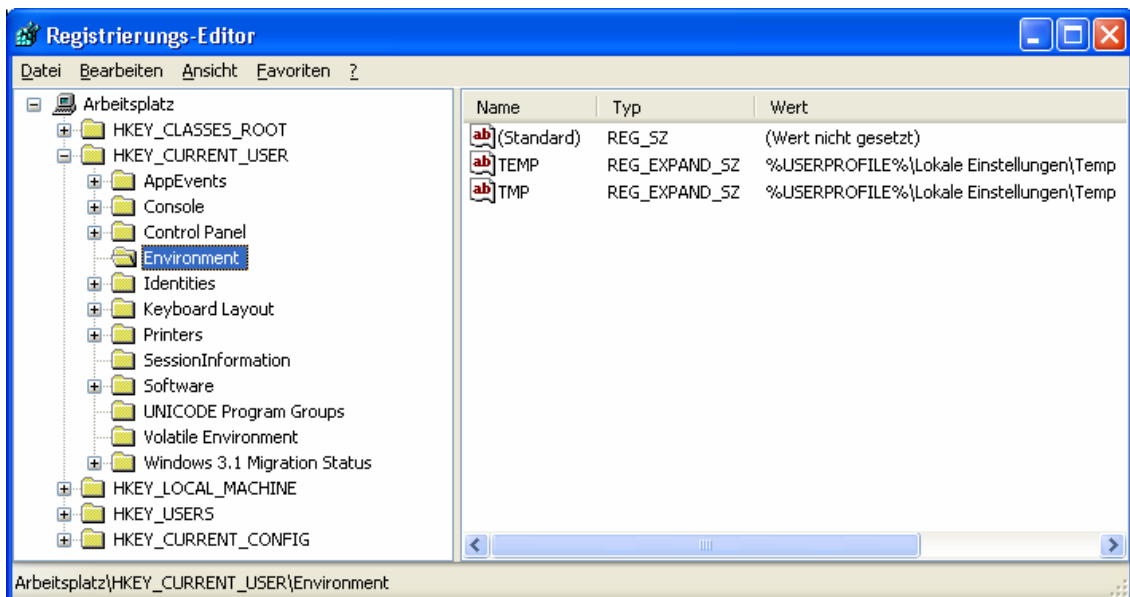
Unter Windows landen die vom Betriebssystem und von den Anwendungsprogramme verwalteten Einstellungen größtenteils in einer recht umfangreichen, hierarchisch organisierten Datenbank namens **Registry**. Sie enthält sowohl benutzer- als auch installationsbezogene Daten in so genannten *Keys* (dt. *Schlüsseln*), die aufgrund der baumartigen Registry-Struktur mit den Pfaden eines Dateisystems vergleichbar sind. Im folgenden Registry-Key wird z.B. die nächste Entwicklungsstufe unseres Editors die vom Benutzer per Optionen-Dialog vorgenommenen Einstellungen speichern, damit sie das Programmende überdauern und beim nächsten Einsatz wieder in Kraft gesetzt werden können:

HKEY_CURRENT_USER\Software\RitchisTexteditor

In den Registrierungs-Schlüsseln befinden sich so genannte *Values* (dt.: *Werte*) von unterschiedlichem Typ (z.B. Zeichenfolge, DWORD), welche die eigentlichen Daten enthalten. Im Editor-Beispiel werden wir folgende Werte einsetzen:

Name	Typ	mögliche Daten
WordWrap	DWORD	0 Zeilenumbruch ist aus 1 Zeilenumbruch ist an
ScrollBars	Zeichenfolge	None Keine Rollbalken Vertical Nur ein vertikaler Rollbalken Horizontal Nur ein horizontaler Rollbalken Both Beide Rollbalken
ZoomFactor	DWORD	64 - 640
BackColor	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 (bis zum System-Totalverlust).



Eine zum NTFS-Dateisystem analoge Rechteverwaltung verhindert bei den Windows-Systemen mit NT-Technik, dass die Registry durch unberechtigte Benutzer verändert wird.

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.

18.2 Die FCL-Klassen *RegistryKey* und *Registry*

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

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 Variablen **RegistryKey**-Objekte zu den Stammknoten der Registrierung zugänglich macht, z.B.:

Registry-Key	Registry-Klassenvariable
HKEY_LOCAL_MACHINE	LocalMaschine
HKEY_CURRENT_USER	CurrentUser

Zur Lesen und Verändern von Daten 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.

In unserem Editor-Projekt wird die neue Funktionalität durch Erweiterungen der Klasse *RitchisTexteditor* realisiert. Zunächst wird der Namensraum **Microsoft.Win32** importiert:

```
using Microsoft.Win32;
```

Weil dieser Namensraum im stets bekannten Bibliotheks-Assembly **mscorlib.dll** implementiert ist, benötigt das Projekt keinen zusätzlichen Verweis.

Die mehrfach benötigten Namen für den HKEY_CURRENT_USER-Unterschlüssel und die dort angesiedelten Werte verwalten wir in **String**-Variablen:

```
const string strUserKey = "Software\\RitchisTexteditor";
const string strWordWrap = "WordWrap";
const string strScrollBars = "ScrollBars";
const string strZoomFactor = "ZoomFactor";
const string strBackColor = "BackColor";
```


Zum Sichern von Einstellungen in die Registry eignet sich eine Überschreibung der **Form**-Methode **OnClosed()**, die beim Schließen des Formulars ausgeführt wird:

```
protected override void OnClosed(EventArgs ea) {
    base.OnClosed(ea);
    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 {
        userKey.Close();
    }
}
```

Es wird versucht, den Registry-Key unseres Editors zum Schreiben zu öffnen. Falls dies nicht gelingt, existiert der Key vermutlich nicht und wird daher erzeugt.

Erhält die verwendete **SetValue()**-Überladung neben dem Namen des betroffenen Wertes 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.

Am Ende der **OnClosed()**-Überschreibung wird die **OnClosed()** Methode der Basisklasse aufgerufen, welche das **Closed**-Ereignis auslöst. Dies ist empfehlenswert, weil aktuell oder bei einer zukünftigen Verwendung unserer Klasse **RitchisTexteditor** Delegaten beim **Closed**-Ereignis registriert sein könnten.

Zum Lesen der Einstellungen eignet sich eine Überschreibung der **Form**-Methode **OnLoad()**, die beim Laden des Formulars ausgeführt wird:

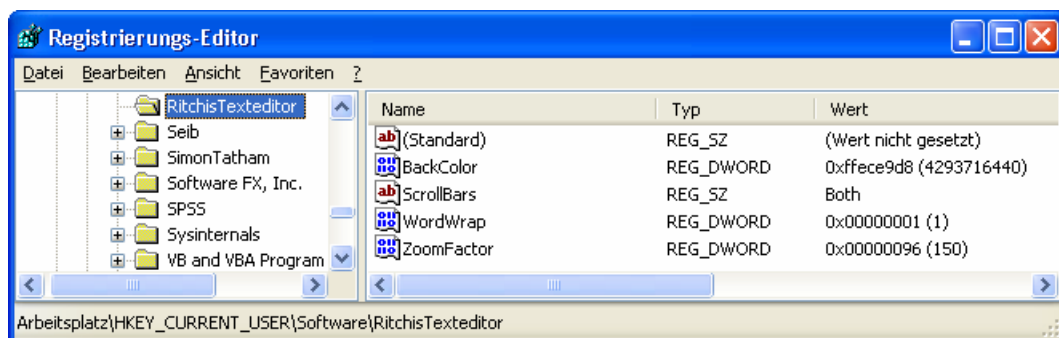
```

protected override void OnLoad(EventArgs ea) {
    base.OnLoad(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 Rückgabetyt **object** besitzt, ist eine explizite Typwandlung erforderlich.

So legt unser Programm seine Einstellungen in der Windows-Registry ab:



Den vollständigen Quellcode zum aktuellen Entwicklungsstand des Editors finden Sie im Ordner

...\BspUeb\Registry\Nur Optionen

18.3 Übungsaufgaben zu Abschnitt 18

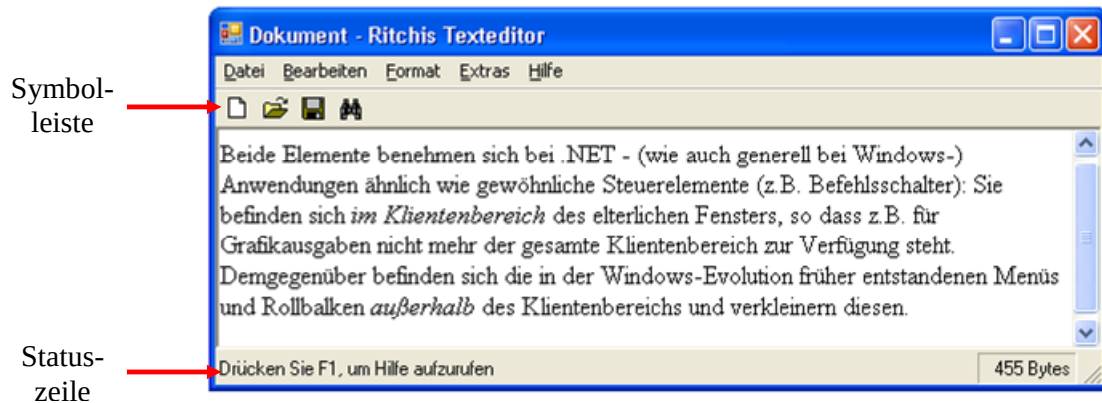
1) Erweitern Sie unseren Editor ausgehend vom Entwicklungsstand in

...\BspUeb\Registry\Nur Optionen

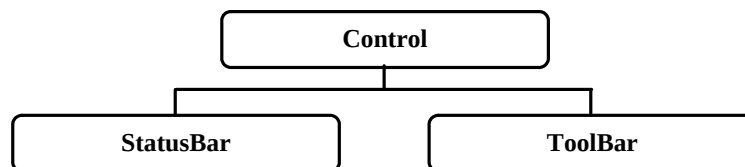
so, dass auch die Position und die Größe des Anwendungsfensters beim Beenden in der Registry gesichert und beim Starten von dort restauriert werden.

19 Statuszeile und Symbolleisten

Nachdem wir Menüs und Dialogboxen kennen gelernt haben, sind Statuszeilen und Symbolleisten auf dem Weg zur zeitgemäßen Bedienoberfläche eines Windows-Programms die nächsten Schritte:



Beide Elemente benehmen sich bei .NET - (wie auch generell bei Windows-) Anwendungen ähnlich wie gewöhnliche Steuerelemente (z.B. Befehlsschalter): Sie befinden sich *im Klientenbereich* des elterlichen Fensters, so dass z.B. für Grafikausgaben nicht mehr der gesamte Klientenbereich zur Verfügung steht. Demgegenüber befinden sich die in der Windows-Evolution früher entstandenen Menüs und Rollbalken *außerhalb* des Klientenbereichs und verkleinern diesen. Das Verhalten der Klassen **StatusBar** und **ToolBar**, welche im .NET – Framework Statuszeilen bzw. Symbolleisten realisieren, wird aus ihrer Abstammung von der Klasse **Control** verständlich:



Die im .NET-Framework 2.0 hinzu gekommenen Klassen **StatusStrip** und **ToolStrip** mit leicht erweiterter Funktionalität (insbesondere bei der Positionierung) können aus Zeitgründen leider nicht behandelt werden (siehe z.B. Kühnel 2006, Kap. 18).

19.1 Statuszeile

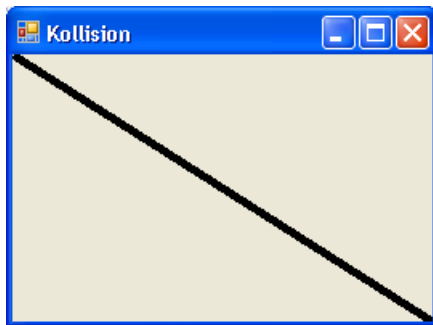
19.1.1 Jeder an seinen Platz

Im folgenden Programm wird per **StatusBar**-Objekt eine noch ziemlich nutzlose Statuszeile realisiert:

```
using System;
using System.Windows.Forms;
using System.Drawing;
class Kollision : Form {
    public Kollision() {
        Text = "Kollision";
        Size = new Size(270, 200);
        ResizeRedraw = true;
        StatusBar sb = new StatusBar();
        sb.Parent = this;
        sb.Text = "Eine Statuszeile belegt Platz im Klientenbereich!";
    }
    protected override void OnPaint(PaintEventArgs e) {
        Pen stift = new Pen(Color.Black, 5);
        e.Graphics.DrawLine(stift, 0, 0, ClientSize.Width, ClientSize.Height);
    }
    static void Main() {
        Application.Run(new Kollision());
    }
}
```

Hauptaufgabe des Programms ist es, den Platzverbrauch einer Statuszeile im Klientenbereich zu demonstrieren:

Ohne Statuszeile



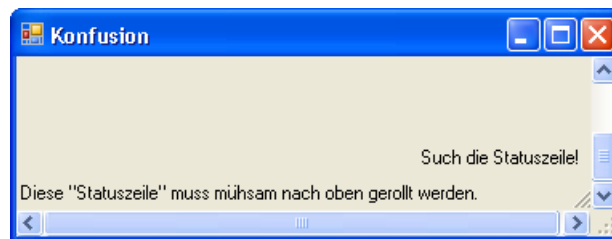
Mit Statuszeile



Weitere wichtige Merkmale der **StatusBar**-Objekte:

- Solange die **DockStyle**-Eigenschaft eines **StatusBar**-Objekts den voreingestellten Wert **DockStyle.Bottom** besitzt, haftet die Statuszeile am unteren Fensterrand und läuft bei jeder Fenstergröße vom linken bis zum rechten Rand.
- Als angenehmen Nebeneffekt einer Statuszeile erhält man eine relativ große Mausclicktref-ferzone zum Initiieren der Fenstergrößenänderung durch Verschieben der rechten unteren Ecke (siehe Zone mit „eingestanztem“ Muster).

Das Einschalten von Rollbalken über die **Form**-Eigenschaft **AutoScroll** ist bei Anwesenheit einer Statuszeile nicht ratsam, z.B.:



Mit folgendem Verfahren gelangt die Statuszeile an die richtige Position:

- Man erzeugt zuerst ein Container-Steuerelement, das über den Wert **DockStyle.Fill** für die **Dock**-Eigenschaft den gesamten Klientenbereich des Formulars belegt. Oft wird es sich da-bei um ein **Panel**-Objekt handeln. In unserem Editor-Projekt ist aber bereits ein **RichText-Box**-Objekt mit der erwünschten Eigenschaft vorhanden.
- Anschließend erhält das Formular eine Statuszeile, so dass sich der Container und das **StatusBar**-Objekt den Klientenbereich teilen.
- Bei Bedarf wird auch noch eine Symbolleiste in das Formular aufgenommen (siehe unten). Alle weiteren Steuerelemente gehören *nicht* direkt zum Formular, landen im Container. Dort erfolgen ggf. auch Grafikausgaben.
- Den Container kann man gefahrlos per **AutoScroll**-Eigenschaft mit Rollbalken ausstatten.

Im folgenden Programm wird das Verfahren mit einem **Panel**-Container demonstriert:

```
using System;
using System.Windows.Forms;
using System.Drawing;

class PanelTrick : Form {
    public PanelTrick() {
        Text = "PanelTrick";
        Size = new Size(330, 170);
        ResizeRedraw = true;

        Panel pan = new Panel();
```

```

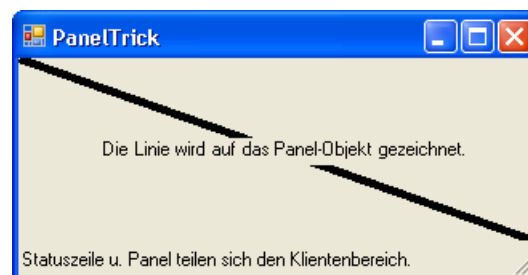
    pan.Parent = this;
    pan.Dock = DockStyle.Fill;
    pan.AutoScroll = true;
    pan.Paint += new PaintEventHandler(PanOnPaint);

    StatusBar sb = new StatusBar();
    sb.Parent = this;
    sb.Text = "Statuszeile u. Panel teilen sich den Klientenbereich.";

    Label lb = new Label();
    lb.Location = new Point(50, 50);
    lb.AutoSize = true;
    lb.Text = "Die Linie wird auf das Panel-Objekt gezeichnet.";
    lb.Parent = pan;
}
protected void PanOnPaint(object sender, PaintEventArgs e) {
    Pen stift = new Pen(Color.Black, 5);
    Panel pan = (Panel)sender;
    e.Graphics.DrawLine(stift, 0, 0, pan.ClientSize.Width,
        pan.ClientSize.Height);
}
static void Main() {
    Application.Run(new PanelTrick());
}
}

```

Weil der **DrawLine()**-Aufruf an der **Panel**-Fläche zeichnet, kommt die erwünschte Diagonale zu Stande:



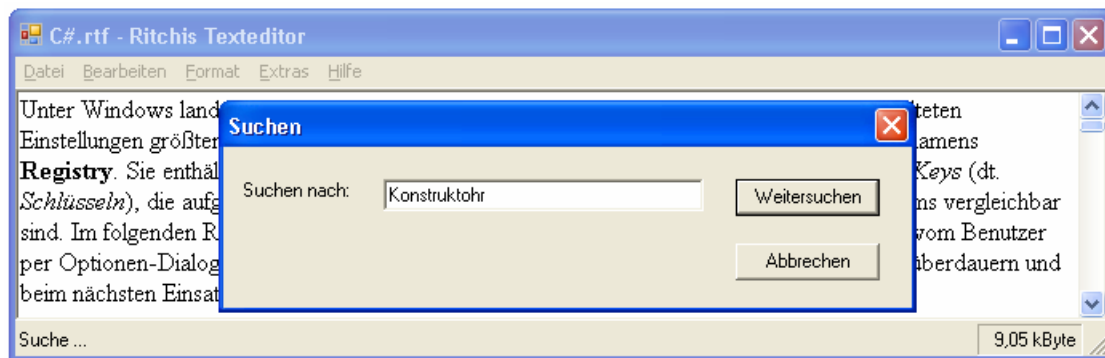
19.1.2 Statuszeilensegmente

Natürlich hat eine Statuszeile mehr zu bieten als einen statischen Text am unteren Fensterrand und einen bequemen Fenstervergrößerungsanfasser. Man kann dort eine Kollektion von **StatusBarPanel**-Objekten unterbringen, die jeweils eine Zeichenfolge und/oder ein Symbol anzeigen. Die **StatusBar**-Eigenschaft **Panels** zeigt auf ein Kollektions-Objekt zur Verwaltung der Statuszeilensegmente, das die Methoden **Add()**, **AddRange()**, **Insert()** und **Remove()** sowie den Indexzugriff beherrscht.

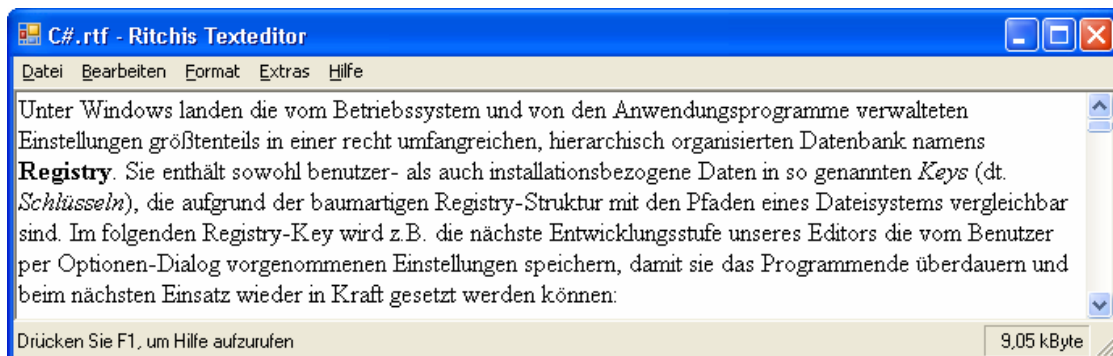
Unser Editor auf **RichTextBox**-Basis erhält nun eine Statuszeile mit folgenden Segmenten:

- 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:



Um die vorgestellte Statuszeile zu realisieren, verwenden wir ein **StatusBar**-Objekt und zwei **StatusBarPanel**-Objekte, wobei die letztgenannten über Instanzvariablen von beliebigen Methoden zwecks Aktualisierung ansprechbar sein sollten.

```
StatusBarPanel sbpInfo, sbpTextSize;
```

Im Fensterkonstruktor wird zunächst das **StatusBar**-Objekt erzeugt und konfiguriert:

```
StatusBar sb = new StatusBar();
sb.Parent = this;
sb.ShowPanels = true;
```

Weil das Formular, auf dem sich die **RichTextBox**-Komponente mit der **Dock**-Eigenschaft **DockStyle.Fill** bereits breit gemacht hat, die Elternschaft übernimmt, klappt die räumliche Koexistenz von Editor und Statuszeile (vgl. Abschnitt 19.1.1).

Um die **StatusBarPanel**-Inhalte erscheinen zu lassen, muss man die **StatusBar**-Eigenschaft **ShowPanels** auf den Wert **true** setzen. Dabei wird ggf. der Inhalt der Statuszeilen-**Text**-Eigenschaft überlagert.

Das **StatusBarPanel**-Objekt zur Informationsanzeige wird *zuerst* in die Statuszeile eingefügt, so dass es an deren linkem Rand erscheint:

```
sbpInfo = new StatusBarPanel();
sbpInfo.BorderStyle = StatusBarPanelBorderStyle.None;
sbpInfo.Text = "Drücken Sie F1, um Hilfe aufzurufen";
sbpInfo.AutoSize = StatusBarPanelAutoSize.Spring;
sb.Panels.Add(sbpInfo);
```

Für die Statuszeilensegmente ist als **BorderStyle**-Wert **StatusBarPanelBorderStyle.Sunken** vor eingestellt (siehe Textgrößenanzeige in obigen Screenshots). Bei einer Infoanzeige wird jedoch in der Regel auf einen Rahmen verzichtet (z.B. im Windows-Editor WordPad). Wir übernehmen dieses Muster und setzen die Eigenschaft **BorderStyle** auf den Wert **StatusBarPanelBorderStyle.None**. Außerdem räumen wir der Infoanzeige über den **AutoSize**-Wert **StatusBarPanelAutoSize.Spring** in horizontaler Richtung den gesamten unbelegten Platz der Statuszeile ein.

Für das Statuszeilensegment zur Anzeige der Textlänge verlangen wir eine am Inhalt orientierte automatische Größenberechnung und eine rechtsbündige Ausrichtung:

```
sbpTextSize = new StatusBarPanel();
sbpTextSize.Text = " 0 Bytes ";
sbpTextSize.AutoSize = StatusBarPanelAutoSize.Contents;
sbpTextSize.Alignment = HorizontalAlignment.Right;
sb.Panels.Add(sbpTextSize);
```

Für das Aktualisieren der Textlängenanzeige sorgt eine beim **RichTextBox**-Ereignis **TextChanged** registrierte Methode:

```
editor.TextChanged += new EventHandler(RichTextBoxOnTextChanged);
.
.
protected void RichTextBoxOnTextChanged(object sender, EventArgs ea) {
    double d = editor.TextLength;
    if (d < 1024)
        sbpTextSize.Text = String.Format("{0:f0}", d) + " Bytes ";
    else {
        d /= 1024;
        if (d < 1024)
            sbpTextSize.Text = String.Format("{0:f2}", d) + " kByte ";
        else {
            d /= 1024;
            if (d < 1024)
                sbpTextSize.Text = String.Format("{0:f2}", d) + " MByte ";
            else {
                d /= 1024;
                sbpTextSize.Text = String.Format("{0:f2}", d) + " GByte ";
            }
        }
    }
}
```

Über die **StatusBarPanel**-Eigenschaft **ToolTipText** lässt sich der Komfort unseres Editors noch steigern:

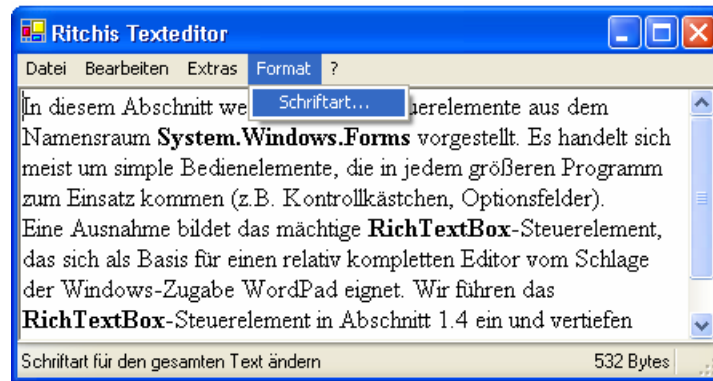


Mit der folgenden Anweisung

```
sbpTextSize.ToolTipText = "Textlänge (ohne RTF-Formatierungscodes)";
```

sorgen wir dafür, dass beim kurzen Verharren des Mauszeigers über dem Textlängen-Statuszeilensegment ein ToolTip-Text darüber informiert, dass eventuell vorhandene RTF-Codes (z.B. für Textauszeichnungen) *nicht* in die Berechnung eingehen.

Ein weiterer Standardservice für Windows-Anwender ist die Anzeige eines kurzen Hilfetextes zum aktuell gewählten Menüitem, z.B.:



Bei der Realisation ist zunächst dafür zu sorgen, dass nach dem Verlassen des Menüs der Statuszeilen-Infotext wieder denselben Inhalt hat wie beim Öffnen des Menüs. Dazu überschreiben wir die **Form**-Ereignisbehandlungsmethoden **OnMenuStart()** und **OnMenuComplete()**:

```
protected override void OnMenuStart(EventArgs ea) {
    infoTextBak = sbpInfo.Text;
}
protected override void OnMenuComplete(EventArgs ea) {
    sbpInfo.Text = infoTextBak;
}
```

Alternativ könnten wir passende **EventHandler** bei den **Form**-Ereignissen **MenuStart** und **MenuComplete** registrieren. Um auf das Hervorheben eines Menüitems reagieren zu können, registrieren wir die folgende Methode:

```
protected void MenuItemOnSelect(object sender, EventArgs ea) {
    if (sender == mitOpen)
        sbpInfo.Text = "Dokument öffnen";
    else if (sender == mitSave)
        sbpInfo.Text = "Dokument speichern";
    . . .
    else if (sender == mitFont)
        sbpInfo.Text = "Schriftart für den gesamten Text ändern";
    else if (sender == mitOptions)
        sbpInfo.Text = "Optionen wählen";
}
```

bei den **Select**-Ereignissen aller Menüitems:

```
EventHandler mitSelectHandler = new EventHandler(MenuItemOnSelect);
foreach (MenuItem mmit in Menu.MenuItems)
    foreach (MenuItem mit in mmit.MenuItems)
        mit.Select += mitSelectHandler;
```

Um die Menüitems im gemeinsamen **Select**-Handler identifizieren zu können, sind zugehörige Instanzvariablen erforderlich. In Petzold (2003, S. 845ff) findet sich der interessante Vorschlag, durch Definition einer eigenen **MenuItem**-Ableitung mit Hilfstext-Instanzvariable den **Select**-Ereignishandler sowie die gesamte Programmstruktur zu vereinfachen.

Dabei könnte gleich noch ein **Unselect**-Ereignis nachgerüstet werden, zur Not unter Verwendung der Windows-Nachricht WM_MOUSELEAVE. Wer unseren Editor sehr kritisch beäugt, wird feststellen, dass der Hilfstext zu einem nicht mehr selektieren Menüitem stehen bleibt, bis ein anderes Item gewählt wird oder das **Form**-Ereignis **MenuComplete** eintritt, z.B.:

korrekte Anzeige



unpassender Hilfstext



Mit dem .NET - Ereignis **MouseLeave** wäre das Problem zu lösen. Es ist in der Klasse **Control** vorhanden, fehlt aber in der **MenuItem**-Basisklasse **Component**.

Den vollständigen Quellcode des Editors im bisher vorgestellten Entwicklungszustand finden Sie im Ordner

...\BspUeb\WinForms\Statuszeile\RitchisTextEditor

19.2 Symbolleisten

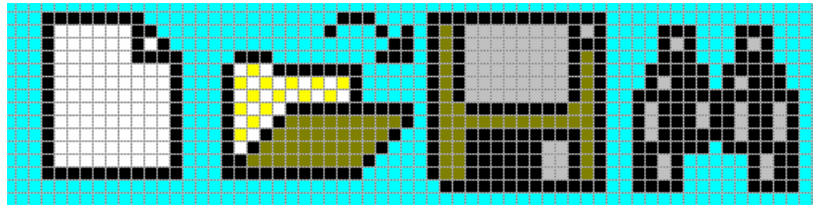
Besonders häufig benötigte Funktionen aus dem Menüsystem bieten Windows-Programme meist auch über Symbolleisten an. Zur Realisation einer Symbolleiste benötigen wir:

- ein **ToolBar**-Objekt
- ein **ImageList**-Objekt mit Bildern
Es genügen Bilder in einer Größe von 16×16 Pixeln.
- eine Serie von **ToolBarButton**-Objekten
Die **ToolBar**-Eigenschaft **Buttons** zeigt auf ein Kollektions-Objekt zur Verwaltung der einzelnen Symbolschaltflächen, das die Methoden **Add()**, **AddRange()**, **Insert()** und **Remove()** sowie den Indexzugriff beherrscht.

Im Fensterkonstruktor unseres Editors werden folgende Zeilen ergänzt:

```
ImageList il = new ImageList();
il.Images.AddStrip(Image.FromFile("Toolbar.bmp"));
il.TransparentColor = Color.Cyan;
ToolBar tb = new ToolBar();
tb.ImageList = il;
tb.Parent = this;
tb.AutoSize = false;
tb.Height = 24;
tb.Appearance = ToolBarAppearance.Flat;
tb.ButtonClick += new ToolBarButtonClickEventHandler(ToolBarOnButtonClick);
string[] tips = { "Neu", "Öffnen", "Speichern", "Suchen" };
MenuItem[] tags = { mitNew, mitOpen, mitSave, mitFind };
for (int i = 0; i < 4; i++) {
    ToolBarButton tbb = new ToolBarButton();
    tbb.ImageIndex = i;
    tbb.ToolTipText = tips[i];
    tbb.Tag = tags[i];
    tb.Buttons.Add(tbb);
}
```

Im Beispiel wird dem **ImageList**-Objekt über die **AddStrip()**-Methode seiner **Images**-Eigenschaft ein Streifen mit 4 Bildern zugeordnet, die zusammen eine Breite von 64×16 Pixeln erreichen:



Wer sich nicht selbst künstlerisch betätigen will, findet geeignete Bitmap-Dateien im Zubehör vieler Entwicklungsumgebungen oder im Internet. Anwender(innen) der Microsoft Visual Studio C# 2005 Express Edition haben nach der Registrierung (z.B. über das Item **Produkt registrieren** im Hilfemenü der Entwicklungsumgebung) Zugang zum *Visual Studio 2005 Registration Benefits Portal*. Von dort kann mit der *IconBuffet Studio Edition Icon Suite* eine Sammlung mit zahlreichen professionellen und lizenzfreien Symbolen bezogen werden.

Weil **Color.Cyan** als Transparentfarbe des **ImageList**-Objekts deklariert ist, erscheinen die entsprechend gefärbten Bitmap-Teile stets in der aktuellen Hintergrundfarbe. Das fertige deklarierte **ImageList**-Objekt wird schließlich einem **ToolBar**-Objekt als Wert seiner **ImageList**-Eigenschaft zugewiesen.

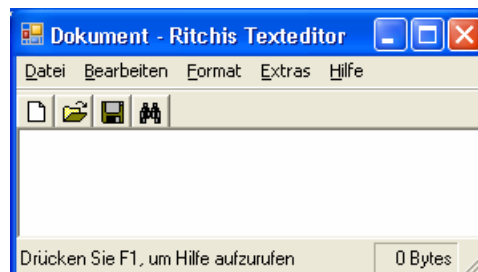
Das **ToolBar**-Objekt wird als Kind des Formulars deklariert. Weil seine **DockStyle**-Eigenschaft den voreingestellten Wert **DockStyle.Top** benutzt, erscheint die Symbolleiste (als Bestandteil des Klientenbereichs!) unterhalb des Hauptmenüs und nimmt stets die ganze Fensterbreite ein. Die beiden Anweisungen

```
tb.AutoSize = false;
tb.Height = 24;
```

dienen lediglich dazu, den meiner Meinung nach zu groß geratenen unteren Rand der Symbolleiste zu korrigieren:



Das heutzutage bevorzugte *flache* Erscheinungsbild der Symbolschaltflächen muss über der Wert **ToolBarAppearance.Flat** für die **ToolBar**-Eigenschaft **Appearance** angefordert werden. Mit dem voreingestellten Wert **ToolBarAppearance.Normal** erhält man folgendes Erscheinungsbild:



Schließlich werden im Beispiel vier **ToolBarButton**-Objekte erzeugt, mit einer Bildnummer versorgt und zur Symbolleiste hinzugefügt.

Der Klarheit halber wird man die Piktogramme in der Regel durch ToolTip-Texte ergänzen, z.B.:



Alternativ kann man aber auch Texte *auf* den Symbolleistenschaltflächen unterbringen, z.B.:



Um diese Darstellungsvariante zu erhalten, wurde die **ToolBar**-Eigenschaft **TextAlign** auf den Wert **ToolBarTextAlign.Right** gesetzt:

```
tb.TextAlign = ToolBarTextAlign.Right;
```

Außerdem erhielten die **ToolBarButton**-Objekte passende Werte für ihre **Text**-Eigenschaft.

Weil die Symbolleistenschaltflächen funktional äquivalent zu korrespondierenden Menüitems sein sollen, ist die Ereignisbehandlung relativ schnell erledigt. Man setzt die **Tag**-Eigenschaft jedes **ToolBarButton**-Objekts auf das zugehörige Menüitem (siehe obige Quelle). Das relevante Ereignis heißt **ButtonClick** und gehört zum **ToolBar**-Objekt:

```
tb.ButtonClick += new ToolBarButtonClickEventHandler(ToolBarOnButtonClick);
```

Was zu tun ist, erfährt unsere Ereignisbehandlungsmethode über einen Parameter vom Typ **ToolBarButtonClickEventArgs**. Dieses Objekt enthält in seiner **Button**-Eigenschaft den **Tag**-Wert des betroffenen Symbolleistenschalters, welcher nach unseren Vorbereitungen auf das korrespondierende Menüitem zeigt. Nach der vom Compiler benötigten expliziten Typwandlung können wir mit Hilfe der gewonnenen Referenz einen Klick auf das Menüitem simulieren:

```
protected void ToolBarOnButtonClick(object sender,
                                     ToolBarButtonClickEventArgs e) {
    ((MenuItem)e.Button.Tag).PerformClick();
}
```

ToolBarButton-Objekte besitzen wie **MenuItem**-Objekte eine **Enabled**-Eigenschaft, um ihre Dienstbereitschaft festzustellen oder festzulegen. Den passiven Zustand signalisieren sie sogar selbstständig durch ein angepasstes Erscheinungsbild von Symbol und/oder Beschriftung. Allerdings existiert kein Analogon zum **Popup**-Ereignis der Klasse **Menu**. Schließlich ist eine Symbolleiste in der Regel permanent sichtbar. In unserem Editor-Projekt sollte das **ToolBarButton**-Objekt 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)
        tbbFind.Enabled = true;
    else
        tbbFind.Enabled = false;
    . . .
}
```

Den vollständigen Quellcode des Editors im bisher vorgestellten Entwicklungszustand finden Sie im Ordner

...\BspUeb\WinForms\Symbolleisten\Ritschis Texteditor

ToolBarButton-Objekte funktionieren oft wie Schaltflächen, doch sind auch alternative Verhaltensweisen möglich (z.B. wie bei einem Kontrollkästchen oder einer Liste, siehe Petzold 2003, S. 859ff).

19.3 Übungsaufgaben zu Abschnitt 19

1) Erweitern Sie bei unserem Editor ausgehend vom Entwicklungsstand in

...\BspUeb\Symbolleisten\Einfach

die Symbolleiste um Schalter zu den **Bearbeiten**-Menüitems **Ausschneiden**, **Kopieren**, **Einfügen** und **Rückgängig**.

20 Drucken

Wir machen uns nun daran, unseren Editor mit einer Druckfunktion auszustatten, die u.a. folgende Leistungsmerkmale bieten soll:

- Das Drucken kann via Menüitem und Symbolleistenschalter angefordert werden, wobei der Windows-Standarddialog erscheint und die üblichen Optionen bietet (z.B. einen frei wählbaren Druckumfang).
- Per Menüitem sind die Standarddialoge zur Seiteneinrichtung bzw. zur Seitenansicht erreichbar.

20.1 Wichtige Klassen

Bei der Verwaltung von Druckaufträgen spielen Objekte der Klasse **PrintDocument** die zentrale Rolle. Zur Verwaltung von Drucker- und Seiteneinstellungen verwenden sie Objekte aus den Klassen **PrinterSettings** und **PageSettings**, die zuerst dargestellt werden sollen.

20.1.1 Die Klasse **PrinterSettings** zur Verwaltung von Druckereinstellungen

Ein Objekt der Klasse **PrinterSettings** aus dem Namensraum **System.Drawing.Printing** enthält die beim Drucken relevanten Geräteeinstellungen. Über etliche Eigenschaften kann man einen Drucker wählen sowie dessen Einstellungen ermitteln und/oder modifizieren, z.B.:

PrinterName	Über diese Eigenschaft vom Typ String 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	Zum Wert false kommt es z.B. dann, wenn PrinterName keinen Namen eines im aktuellen System verfügbaren Druckers enthält.
DefaultPageSettings	Beim Erzeugen eines PrinterSettings -Objekts wird stets ein PageSettings -Objekt (siehe Abschnitt 20.1.2) mit den voreingestellten Seiteneigenschaften des Druckers erstellt. Bei einer validen Änderung der PrinterName -Eigenschaft wird das PageSettings -Objekt aktualisiert.
SupportsColor	Gibt an, ob Farbdrucke möglich sind
CanDuplex	Gibt an, ob beidseitiges Drucken möglich ist
Duplex	Gibt an, ob derzeit beidseitiges Drucken eingestellt ist
LandscapeAngle	Bei verfügbarer Landscape-Orientierung lautet der Wert 90 oder 270, anderenfalls 0. Man erhält den Drehwinkel des Querformats gegenüber dem Hochformat.
PaperSizes	Zeigt auf eine Kollektion mit den unterstützten Papiergrößen, jeweils beschrieben durch ein PaperSize -Objekt
PaperSources	Zeigt auf eine Kollektion mit den verfügbaren Papierquellen, jeweils beschrieben durch ein PaperSource -Objekt
PaperResolutions	Zeigt auf eine Kollektion mit den unterstützten Auflösungen, jeweils beschrieben durch ein PaperResolution -Objekt
FromPage	Stellt oder legt die Nummer der ersten zu druckenden Seite fest
ToPage	Stellt oder legt die Nummer der letzten zu druckenden Seite fest
MinimumPage	Stellt oder legt die kleinste Seitennummer fest, die im Standarddialog <i>Drucken</i> (siehe Abschnitt 20.1.4) bei der Wahl eines Seitenbereichs als Druckumfang erlaubt ist
MaximumPage	Stellt oder legt die größte Seitennummer fest, die im Standarddialog <i>Drucken</i> (siehe Abschnitt 20.1.4) bei der Wahl eines Seitenbereichs als Druckumfang erlaubt ist

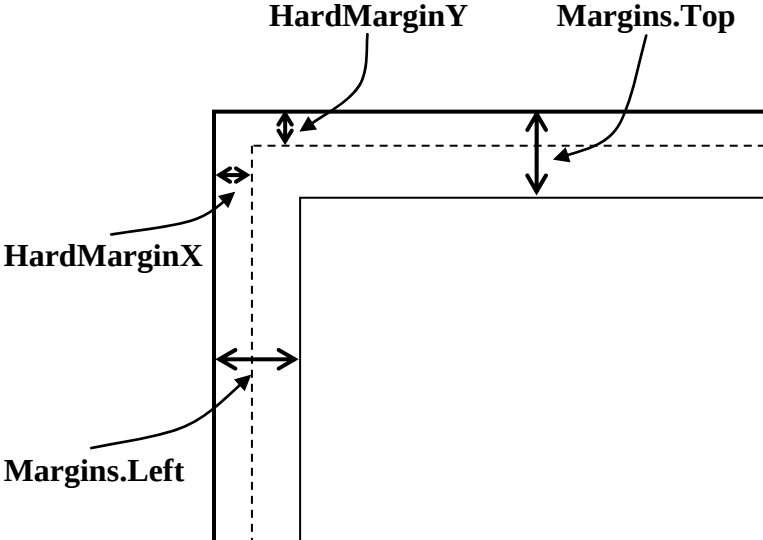
Über die statische Eigenschaft **InstalledPrinters** der Klasse **PrinterSettings** erhält man ein Objekt der internen Klasse **PrinterSettings.StringCollection** mit den Namen der verfügbaren Drucker.

Das Erstellen eines **PrinterSettings**-Objekts per Konstruktor findet eher selten statt. Meist verwendet man die in einem **PrintDocument**-Objekt (siehe Abschnitt 20.1.3) enthaltene **PrinterSettings**-Instanz. Diese ist zunächst mit dem Standarddrucker des angemeldeten Benutzers verbunden, kann aber beliebig konfiguriert werden.

20.1.2 Die Klasse **PageSettings** zur Verwaltung von Seiteneigenschaften

Einzelne Druckseiten werden durch Objekte der Klasse **PageSettings** beschrieben, die u.a. folgende Eigenschaften haben:

PrinterSettings	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	Stellt oder legt fest, ob die Seite im Querformat gedruckt werden soll
PaperSize	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	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	Stellt oder legt die Seitenränder in 1/100 Zoll fest. Voreinstellung: 1 Zoll Über den Standarddialog <i>Seiteneinrichtung</i> (siehe Abschnitt 20.3.2) kann man dem Benutzer die Wahl der Seitenränder ermöglichen.
PrintableArea	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 Rectangle -Objekt mit der bedruckbaren Seitengröße in 1/100 Zoll, wobei die per Landscape -Eigenschaft festgelegte Seitenausrichtung berücksichtigt wird.

HardMarginX, HardMarginY	Über die seit .NET 2.0 verfügbaren Eigenschaften HardMarginX bzw. HardMarginY ermittelt man die X- bzw. Y-Koordinate des druckerbedingten linken Randes in 1/100 Zoll. 
PaperSource	Stellt oder legt die Papierquelle fest. Über die verfügbaren Papierquellen informiert die PrinterSettings -Eigenschaft PaperSources (siehe oben).
PrinterResolution	Stellt oder legt die Druckauflösung fest. Über die verfügbaren Auflösungen informiert die PrinterSettings -Eigenschaft PrinterResolutions (siehe oben).
Color	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 20.1.3) enthaltene **PageSettings**-Instanz, um Seiteigenschaften festzulegen.

20.1.3 Die Klasse **PrintDocument** zur Verwaltung von Druckaufträgen

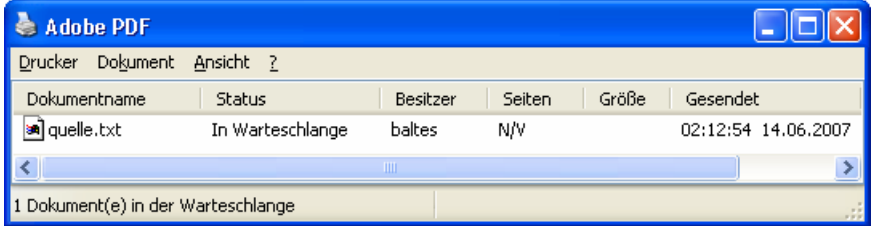
Beim Drucken fungiert ein Objekt der Klasse **PrintDocument** als zentrale Koordinierungsstelle, so dass seine Aufgaben zunächst einmal im Überblick dargestellt werden:

- 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 z.B. in der Behandlungsmethode zum **PrintDocument**-Ereignis **PrintPage**.

Obwohl es der Name **PrintDocument** suggeriert, dienen Objekte dieser Klasse *nicht* zur Aufnahme der auszugebenden Daten.

20.1.3.1 Eigenschaften und Methoden

Ein Objekt der Klasse **PrintDocument** besitzt u.a. die folgenden Eigenschaften:

PrinterSettings	Diese Eigenschaft zeigt auf ein Objekt aus der Klasse PrinterSettings , das vom PrintDocument -Konstruktor angelegt wird ¹ . Es ist zunächst mit dem Standarddrucker des angemeldeten Benutzers verbunden. Bei der Druckausgabe werden die Eigenschaften dieses PrinterSettings -Objekts verwendet.
DefaultPageSettings	Diese Eigenschaft zeigt auf ein PageSettings -Objekt, das ebenfalls vom PrintDocument -Konstruktor angelegt wird. Bei der Druckausgabe werden die Eigenschaften dieses PageSettings -Objekts verwendet.
DocumentName	Name des Druckauftrags Er wird von Windows zur Verwaltung des Druckauftrags verwendet und z.B. in der Warteschlange eines Druckes angezeigt: 
OriginAtMargins	Bei der via PrintDocument-Objekt veranlassten Druckausgabe kommt ein Graphics-Objekt ins Spiel, das wir gemäß Abschnitt 14 (<i>Graphikausgabe mit dem GDI+</i>) 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. • true Der Ursprung (0, 0) ist von der oberen linken Ecke des Papiers in horizontaler bzw. vertikaler Richtung um den jeweiligen Seitenrand entfernt. 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 ...

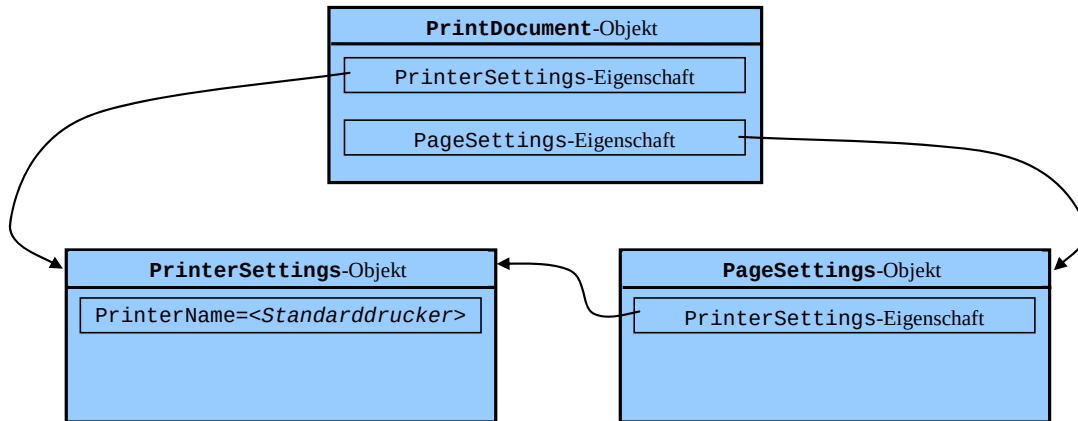
¹ Den **PrintDocument**-Konstruktor kann sich ungefähr so vorstellen:

```
public PrintDocument() {
    documentname = "document";
    printersettings = new PrinterSettings();
    defaultpagesettings = new PageSettings();
    printcontroller = new StandardPrintController();
}
```

Dieses Exemplar stammt aus dem **Mono**-Projekt, der Open Source - .NET-Portierung für Linux (Version 1.1.18, siehe <http://www.mono-project.com/Downloads>).

- 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;
```

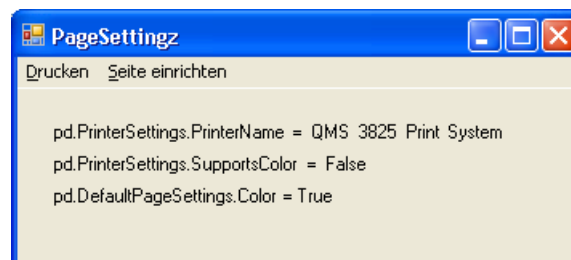
Demgegenüber bleibt die nächste Anweisung ohne Effekt, weil das falsche **PageSettings**-Objekt angesprochen wird:

```
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 angeschlossen 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.

20.1.3.2 Ereignisse

Nach dem Aufruf seiner **Print()**-Methode feuert ein **PrintDocument**-Objekt nacheinander die folgenden **Ereignisse** und ruft ggf. die registrierten Behandlungsmethoden auf:¹

- **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 und in seiner **PrintAction**-Eigenschaft über das Ausgabeziel informiert (Datei, Seitenansicht, Drucker). Weitere Details zum **BeginPrint**-Handler und ein ausführliches Beispiel folgen in Abschnitt 20.2.2.
- **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 Orientierung:
`pd.DefaultPageSettings.Landscape = true;`
Über die **Cancel**-Eigenschaft des Ereignisobjekts lässt sich der Druckauftrag abbrechen, und per **PrintAction**-Eigenschaft erfährt man das Ausgabeziel (Datei, Seitenansicht, Drucker).
- **PrintPage** (für jede einzelne Seite)
Dem Handler wird als Aktualparameter ein Objekt der Klasse **PrintPageEventArgs** übergeben, das u.a. ein **Graphics**-Objekt mitbringt, das nun ein Befüllen der Seite ermöglicht. 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 20.2.3.
- **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 Zeilen

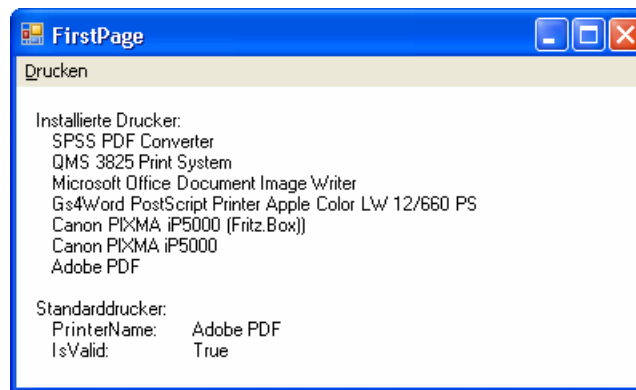
```
PrintDocument pd = new PrintDocument();
pd.Print();
```

zum Ausdruck einer leeren Seite durch den Standarddrucker des Benutzers.

20.1.3.3 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:

¹ 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 20.3.1 kommt eine **PrintController**-Ableitung zur Sprache, die beim Erstellen der Seitenansicht hilft.



Über das Hauptmenüitem **Drucken** können diese Informationen auch zu Papier gebracht werden.

Um die Klassen **PrinterSettings**, **PageSettings** und **PrintDocument** im Quelltext bequem ansprechen zu können, wird der Namensraum **System.Drawing.Printing** importiert:

```
using System.Drawing.Printing;
```

An die Liste der installierten Drucker gelangt das Beispielprogramm 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) {
    g.DrawString("Installierte Drucker:", Font, brush, 10, Font.Height);
    for (int i = 0; i < pc.Count; i++)
        g.DrawString(pc[i], Font, brush, 20, (i+2)*Font.Height);
    g.DrawString("Standarddrucker:", Font, brush,
        10, (pc.Count+3)*Font.Height);
    g.DrawString("PrinterName:\t"+pd.PrinterSettings.PrinterName, Font,
        brush, 20, (pc.Count+4)*Font.Height);
    g.DrawString("IsValid: \t"+pd.PrinterSettings.IsValid.ToString(), Font,
        brush, 20, (pc.Count + 5) * Font.Height);
}
```

Diese Methode wird sowohl in der **OnPaint()**-Überschreibung des Formulars zur Bildschirmausgabe

```
protected override void OnPaint(PaintEventArgs e) {
    DrawToDevice(e.Graphics);
}
```

als auch in der folgenden **PrintPage**-Ereignisbehandlungsmethode zur Druckausgabe genutzt:

```
void PrintDocumentOnPrintPage(object sender, PrintPageEventArgs e) {
    DrawToDevice(e.Graphics);
}
```

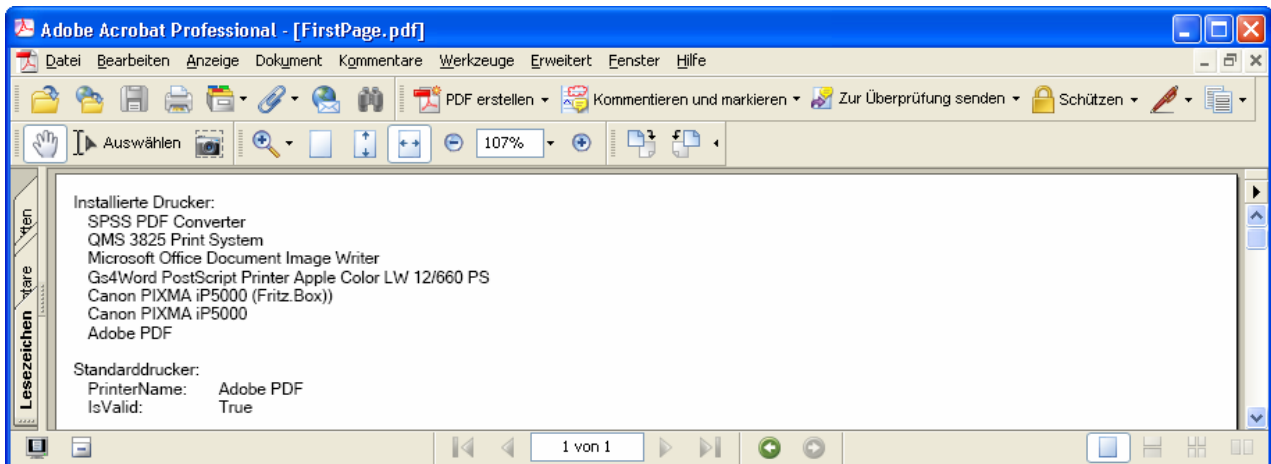
Der **PrintPage**-Handler ist beim **PrintDocument**-Objekt registriert:

```
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 14.8.2). Dass **DrawToDevice()** 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.

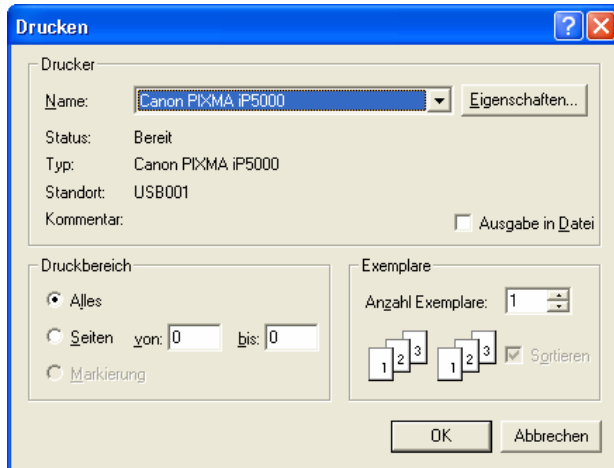
Den vollständigen Quellcode des Programms finden Sie im Ordner

...\BspUeb\Drucken\FirstPage

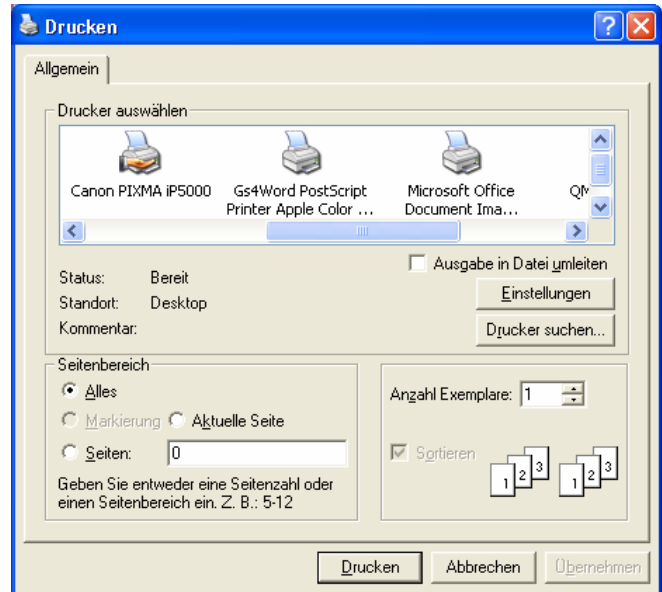
20.1.4 Der Standarddialog Drucken

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:

Standardversion



Erweiterte Version



Hier können Benutzer ihren Druckauftrag auf vielfältige Weise beeinflussen, z.B.:

- Auswahl des Druckers
- Seitenorientierung (via Schalter **Eigenschaften**)
- Auswahl eines Druckbereichs

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 MyApp : Form {
    PrintDocument myPrintDocument = new PrintDocument();
    PrintDialog myPrintDialog = 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.:
`myPrintDialog.Document = myPrintDocument;`
- **PrinterSettings**
Das im Druckdialog zu modifizierende **PrinterSettings**-Objekt wird direkt festgelegt.

Andere Eigenschaften zur Konfiguration des Dialogs:

- **AllowSelection** (Voreinstellung: **false**)
Soll der Benutzer als Druckumfang die aktuelle Auswahl angeben können?
- **AllowSomePages** (Voreinstellung: **false**)
Soll der Benutzer als Druckumfang einen Seitenbereich angeben können?
- **AllowCurrentPage** (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 *erweiterte* Druckdialog gewählt werden.
- **AllowPrintToFile** (Voreinstellung: **false**)
Soll der Benutzer die Druckausgabe in eine Datei leiten können?

- **PrintToFile** (Voreinstellung: **false**)
Hat der Benutzer das Kontrollkästchen zur Dateiausgabe markiert?
- **ShowHelp** (Voreinstellung: **false**)
Wird die Eigenschaft auf den Wert **true** gesetzt, muss eine **HelpRequest**-Ereignisbehandlungsmethode implementiert und registriert werden.
- **UseExDialog** (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 20.3.1), ohne eine Druckausgabe zu starten, wenn man den Druckdialog anschließend mit **Abbrechen** verlässt. Wer diese Bedienungsoption anbieten möchte, kann aber nicht gleichzeitig die Regel beachten, dass ein mit **Abbrechen** beendeter Dialog ohne Folgen bleibt (vgl. Abschnitt 17.1.3).

Ist dem Benutzer eine flexible Auswahl des Druckumfangs erlaubt, werden folgende Eigenschaften des eingebundenen **PrinterSettings**-Objekts relevant:

- **PrintRange**
Wir erfahren über folgende Werte der Enumeration **PrintRange**, welcher Druckbereich gewählt wurde:
 - o **Allpages**
 - o **CurrentPage**
 - o **Selection**
 - o **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.

20.2 Druckfunktionalität für den Editor auf **RichTextBox**-Basis

Wir verwenden nun das neu erworbene Wissen dazu, unseren Editor auf **RichTextBox**-Basis mit einer Druckfunktion auszustatten. Dabei beschränken wir uns auf den Ausdruck von unformatiertem Text, 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 und dabei sogar einen automatischen Zeilenumbruch vornimmt.

Vermutlich beherrscht das **RichTextBox**-Steuerelement im .NET - Framework irgendwann einmal Methoden zum Drucken. Auf der Webseite

<http://www.microsoft.com/downloads/details.aspx?FamilyID=a79f5f90-8867-457d-93b1-00da386f6221&displaylang=en>

bietet Microsoft schon jetzt eine **RichTextBox**-Ableitung an, die das formatierte Drucken (WYSIWYG!) über Routinen aus dem Win32-API realisiert. Aber im Abschnitt 20 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 auszugswise vorgestellt. Eine vollständige Version finden Sie im Ordner

...\BspUeb\Drucken\RitchisTextEditor

20.2.1 Beteiligte Objekte

Wir verwenden in unserem Editor *ein* **PrintDocument**-Objekt, das am Anfang der Klassendefinition erzeugt und der Instanzvariablen `druckAuftrag` zugewiesen wird:

```
class RitchisTexteditor : Form {
    . . .
    PrintDocument druckAuftrag = new PrintDocument();
    . . .
}
```

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 `druckDialog` zugewiesen wird:

```
class RitchisTexteditor : Form {
    . . .
    PrintDialog druckDialog = new PrintDialog();
    . . .
}
```

Schließlich deklarieren wir auf Klassenebene noch einige Hilfsvariablen:

```
string tempText;
int fromPage, toPage, actPage;
```

für folgende Zwecke:

- `tempText` dient zu Extraktion des zu druckenden Textes.
- `fromPage` erhält die Nummer der ersten auszugebenden Seite.
- `toPage` erhält die Nummer der letzten auszugebenden Seite.
- `actPage` enthält während des gesamten Druckvorgangs die Nummer der gerade in Bearbeitung befindlichen Seite.

Im Fensterkonstruktor legen wir das Menüitem **Datei > Drucken** an:

```
mitPrint = mitFile.MenuItems.Add("&Drucken",
    new EventHandler(MitPrintOnClick));
mitPrint.Shortcut = Shortcut.CtrlP;
```

Außerdem werden im Fensterkonstruktor die Objekte für den Druckauftrag und den Druckauswahl-Standarddialog auf ihren Einsatz vorbereitet:

```
druckDialog.Document = druckAuftrag;
druckDialog.AllowSomePages = true;
druckDialog.AllowCurrentPage = true;
druckDialog.UseEXDialog = true;
druckAuftrag.BeginPrint += new PrintEventHandler(DruckAuftragOnBeginPrint);
druckAuftrag.PrintPage += new PrintPageEventHandler(DruckAuftragOnPrintPage);
```

Insbesondere ...

- wird der Druckdialog über seine **Document**-Eigenschaft mit dem Druckauftrag verknüpft, um das im Druckdialog zu modifizierende **PrinterSettings**-Objekt festzulegen.
- wird der Druckdialog angewiesen, die Auswahl eines Seitenbereich sowie der aktuellen Seite als Druckumfang anzubieten und mit erweiterter Funktionalität aufzutreten.
- werden Behandlungsmethoden bei den Ereignissen **BeginPrint** und **PrintPage** des Druckauftrags registriert. Die beiden Methoden werden später in eigenen Abschnitten vorgestellt.

In der Behandlungsmethode zum **Click**-Ereignis des Menüitems **Datei > Drucken** setzen wir die Druckdialog-Eigenschaft **AllowSelection** auf **true**, wenn beim Aufruf ein Textbereich markiert ist.

```
protected void MitPrintOnClick(object sender, EventArgs e) {
    druckDialog.AllowSelection = (editor.SelectionLength > 0);
    if (druckDialog.ShowDialog() == DialogResult.OK)
        druckAuftrag.Print();
}
```

Wenn der Benutzer den Druckdialog mit **OK** quittiert, initiieren wir die Druckausgabe über die **Print()**-Methode des Druckauftrags.

20.2.2 BeginPrint - Behandlung

In der Behandlungsmethode zum **BeginPrint**-Ereignis des **PrintDocument**-Objekts findet eine Druckvorbereitung aufgrund der vom Benutzer via Druckdialog im **PrinterSettings**-Objekt vorgenommenen Einstellungen statt:

```
protected void DruckAuftragOnBeginPrint(object sender, PrintEventArgs e) {
    switch (druckDialog.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 = druckDialog.PrinterSettings.FromPage;
            toPage = druckDialog.PrinterSettings.ToPage;
            break;
        case PrintRange.CurrentPage:
            tempText = editor.Text;
            fromPage = 1;
            toPage = Int32.MaxValue;
            break;
    }
    actPage = 1;
    if (fileName != null)
        druckAuftrag.DocumentName = Path.GetFileName(fileName);
    else
        druckAuftrag.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.

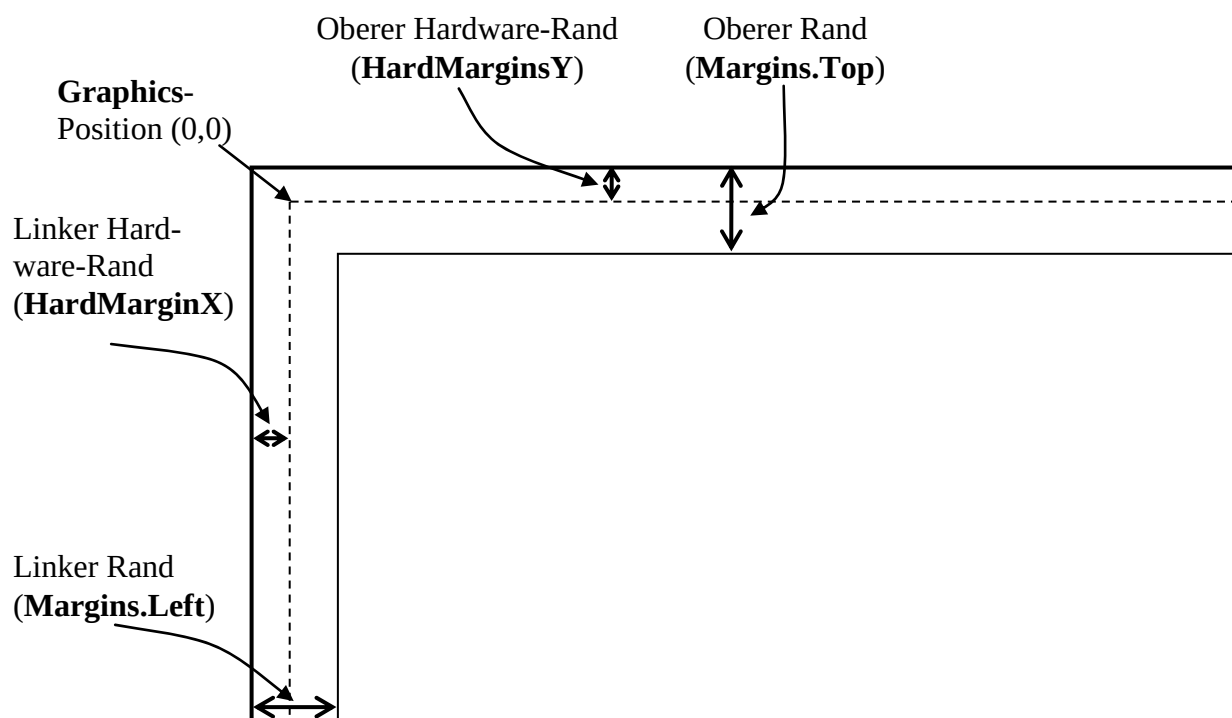
20.2.3 PrintPage - Behandlung

Kernstück bei der Druckunterstützung eines Programms ist die Behandlungsmethode zum **PrintPage**-Ereignis des Druckauftragsobjektes. Sie erhält wesentliche Informationen für ihre Tätigkeit über ein Aktualparameterobjekt vom Typ **PrintPageEventArgs**, das die folgenden **public**-Eigenschaften besitzt:

Graphics	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 14.8.2). Als alternative PageUnit -Eigenschaft kommt z.B. GraphicsUnit.Millimeter in Frage.
Cancel	Um einen Druckauftrag abzubrechen, setzt ein PrintPage -Handler die Cancel -Eigenschaft auf true .
HasMorePages	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	Über diese Eigenschaft sind die Einstellungen der aktuellen Seite zu ermitteln.
PageBounds	Bietet einen lesenden Zugriff auf eine Rectangle -Instanz mit der aktuellen Seitengröße unter Berücksichtigung der Seitenausrichtung (Maßeinheit: 1/100 Zoll)
MarginBounds	Bietet einen lesenden Zugriff auf eine Rectangle -Instanz, die aus dem PageBounds -Rechteck durch Berücksichtigung der (meist vom Benutzer eingestellten) Seitenränder entsteht, die in PageSettings.Margins hinterlegt sind (Maßeinheit: 1/100 Zoll)

20.2.3.1 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** könnte 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 20.1.3). Bei dieser Einstellung resultieren aber für Druckausgaben in der Randzone (z.B. Kopf- oder Fußzeilen) negative Koordinaten, so dass wir die Voreinstellung *nicht* ändern.

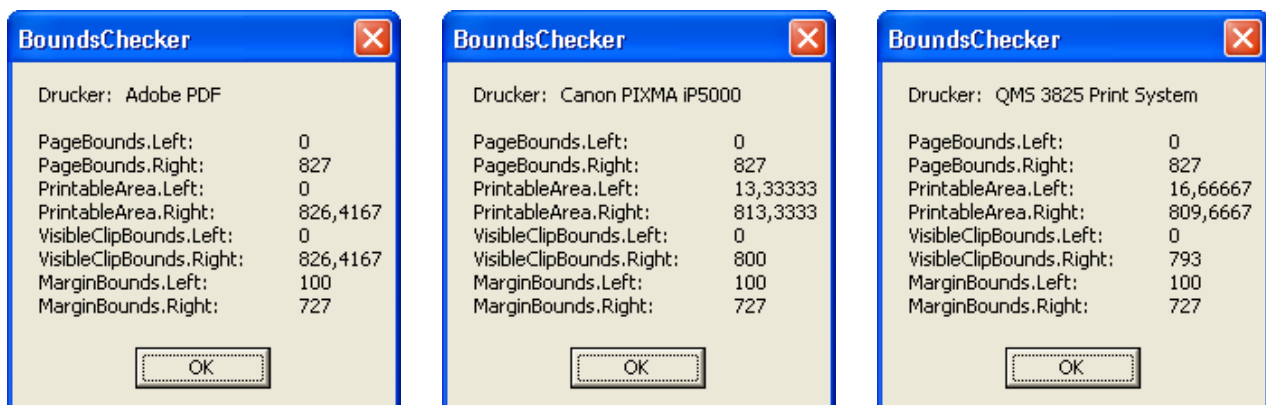
Wir kennen in der **PrintPage**-Behandlungsmethode:

- Papiergröße (unter Berücksichtigung der Seitenausrichtung)
Diese liefert die **PrintPageEventArgs**-Eigenschaft **PageBounds**, welche den **Bounds** – Wert des eingebundenen **PageSettings**-Objekts durchreicht.
- Seitenränder
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 auf eine **RectangleF**-Instanz, welche die bedruckbare Seite beschreibt. Drucker können das Papier in der Regel nicht bis zu den Rändern bedrucken, weil z.B. Randstreifen für den Papiertransport benötigt werden.
Über die **PrintableArea**-Eigenschaft des eingebundenen **PageSettings**-Objekts erhalten wir die bedruckbare Seite in „Papierkoordinaten“, die gegenüber den **Graphics**-Koordinaten um die druckerbedingten Ränder verschoben sind.

In einer Übungsaufgabe (siehe Abschnitt 20.4) sollen Sie für einen Druckauftrag ausgeben:

- erste und letzte horizontale Koordinate der Seite
- erste und letzte horizontale Koordinate der bedruckbaren Seite
- erste und letzte horizontale **Graphics**-Koordinate der bedruckbaren Seite
- erste und letzte horizontale Koordinate der Seite bei Berücksichtigung der Ränder

Wie die folgenden Beispielausgaben zeigen, hängt die Größe der bedruckbaren Seite vom Ausgabegerät ab:



Während der Pseudodruckertreiber **Adobe PDF** zum Erzeugen von PDF-Dateien praktisch 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 Ausgabeposition (0, 0) des **Graphics**-Objekts, das unsere Ausgabeanweisungen ausführt, ist vom linken Rand des Papiers in horizontaler Richtung unterschiedlich weit entfernt, beim **QMS 3825** z.B. ca. 4 mm. Um den (meist vom Benutzer gewünschten) linken Rand (**MarginBounds.Left**) einzuhalten, muss der druckerbedingte linke Rand subtrahiert werden.

Mit Hilfe der verfügbaren Informationen erzeugen wir ein **RectangleF**-Objekt, das zum Begrenzen der Textausgabe 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 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, dass am unteren Rand keine solchen unvollständigen Zeilen

18
19
20

Seite 2

erscheinen:

```
zeilenabstand = editor.Font.GetHeight(g);
ganzeZeilen = (int)(rectPage.Height / zeilenabstand);
rectPage.Height = ganzeZeilen * zeilenabstand;
```

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.

20.2.3.2 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
- Berechnung der Zeilenabstände

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);
```

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;
}
```

Die Schriftgröße wird übrigens unabhängig von den **Graphics**-Eigenschaften **PageUnit** und **PageScale** grundsätzlich in Punkten zu je 1/72 Zoll angegeben.

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:¹

```
while (actPage < firstPage && tempText.Length > 0) {
    g.MeasureString(tempText, editor.Font, rectPage.Size, sfText,
                    out anzZeichen, out anzZeilen);
    if (!editor.WordWrap)
        anzZeichen = NumberOfCharsInBlock(tempText, anzZeilen);
    tempText = tempText.Substring(anzZeichen);
    actPage++;
}
```

Wir erfahren über einen Ausgabeparameter, wie viele Zeichen des aktuellen tempText-Zeichenfolge im Ausgaberechteck Platz finden und streichen über die **String**-Methode **Substring()** entsprechend viele Zeichen am Anfang des Hilfstextes. 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 (2003, S. 906) übernommene Methode:

```
int NumberOfCharsInBlock(string text, int anzZeilen) {
    int last = 0;
    for (int i = 0; i < anzZeilen; 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;
}
```

¹ Die in Abschnitt 14.7.4.2 diskutierten **MeasureString()**-Probleme bei der Breitenberechnung für kurze, aufeinander folgende und individuell formatierte Zeichenfolgen kommen hier *nicht* zum Tragen.

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 > zeilenabstand) {
    StringFormat sfKF = new StringFormat();
    sfKF.Alignment = StringAlignment.Center;
    RectangleF rectHead = rectPage;
    rectHead.Height = zeilenabstand;
    rectHead.Y = Math.Max((e.PageSettings.Margins.Top - zeilenabstand) / 2
        - e.PageSettings.HardMarginY, 0);
    g.DrawString(druckAuftrag.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:

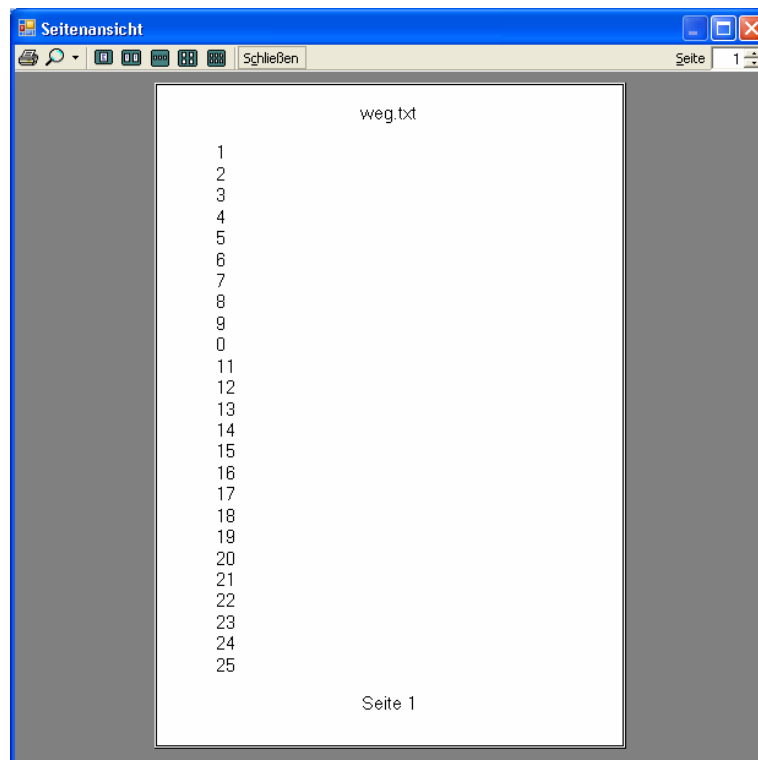
```
g.MeasureString(tempText, editor.Font, rectPage.Size, sfText,
    out anzZeichen, out anzZeilen);
if (!editor.WordWrap)
    anzZeichen = NumberOfCharsInBlock(tempText, anzZeilen);
tempText = tempText.Substring(anzZeichen);
actPage++;
if (tempText.Length > 0 && actPage <= toPage)
    e.HasMorePages = 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.

20.3 Weitere Standarddialoge zur Unterstützung der Druckausgabe

20.3.1 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 angeboten werden, was natürlich auch in unserem Editor geschehen 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 zum 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 des Editors her:

```
previewDialog.Document = druckAuftrag;
```

Per Menüitem oder Symbolleistenhalter kann die folgende Ereignisbehandlungsmethode gestartet werden:

```
protected void MitPreviewOnClick(object sender, EventArgs e) {
    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. Daher wird in der **BeginPrint**-Behandlungsmethode des **PrintDocument**-Objekts (siehe Abschnitt 20.2.2) folgende Fallunterscheidung ergänzt, welche die **PrintAction**-Eigenschaft des **PrintEventArgs**-Objekts nutzt:

```
if (e.PrintAction == PrintAction.PrintToPreview) {
    tempText = editor.Text;
    fromPage = 1;
    toPage = Int32.MaxValue;
} else
    . . .
```

Für die eigentliche Seitenvorschau ist ein Steuerelement aus der Klasse **PrintPreviewControl** verantwortlich. Sie können 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.

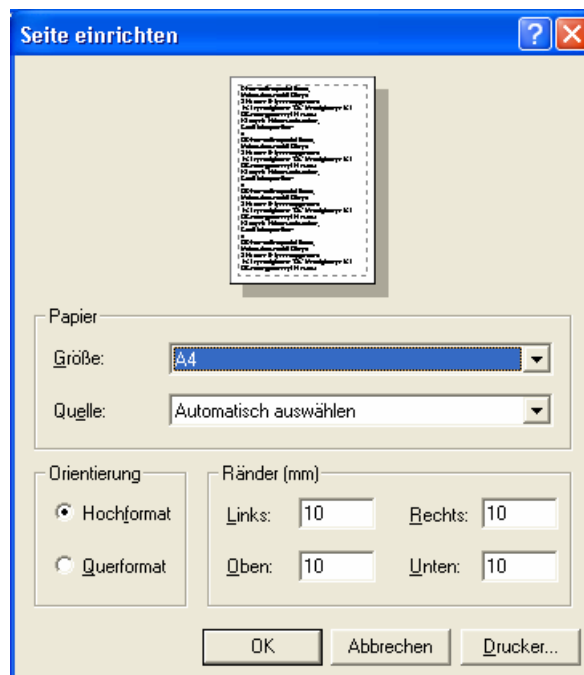
Es fällt übrigens auf, dass der Seitenaufbau durch ein **PrintPreviewDialog**-Objekt in Abhängigkeit vom zugeordneten Drucker recht lahm sein kann.

Vielleicht sind Sie daran interessiert, was ein **PrintPreviewDialog**-Objekt hinter der Bühne beim Aufruf seiner **ShowDialog()**-Methode tut:

- Es setzt die **PrintController**-Eigenschaft des Druckauftrags, auf den seine **Document**-Eigenschaft zeigt, 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 zu Aufrufen des **PrintPage**-Handlers führt.
- Es zeigt die fertigen Bilder in einem Formular mit etlichen Bedienelementen an.

20.3.2 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 = druckAuftrag;
```

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 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.

Per Menüitem kann im erweiterten Editor die folgende Ereignisbehandlungsmethode gestartet werden:

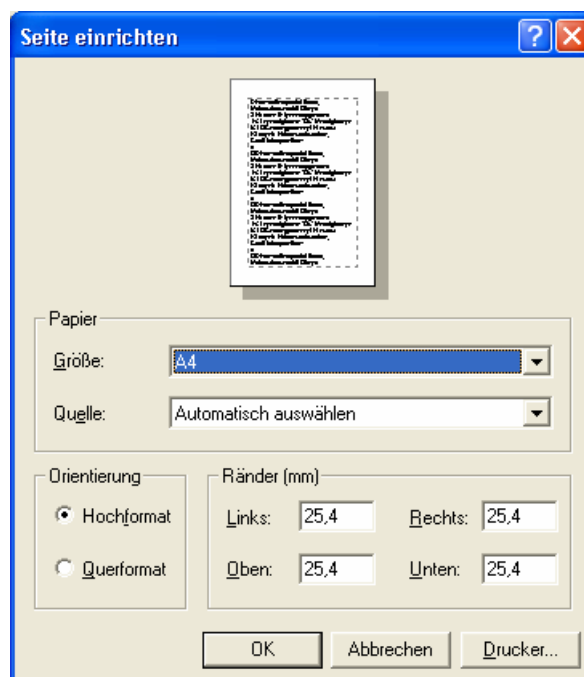
```
protected void MitPageSetupOnClick(object sender, EventArgs e) {
    pageSetupDialog.ShowDialog();
}
```

Es ist keine Auswertung des **ShowDialog()**-Rückgabewerts und keine Verarbeitung der erfragten Informationen erforderlich, weil der Seiteneinrichtungsdialog die mit **OK** quittierten Einstellungen selbständig auf das angebundene **PageSettings**- bzw. **PrinterSettings**-Objekt anwendet und auch auf das Abbrechen durch den Benutzer geeignet reagiert.

Leider enthält die Klasse **PageSetupDialog** einen ärgerlichen Fehler: Beim Öffnen des Dialogs wird falsch von der Druckermaßeinheit (1/100 Zoll) in Millimeter umgerechnet, so dass um den Faktor 2,54 verkleinerte Ränder in den Textfeldern landen. Beim voreingestellten Rand von 1 Zoll (= 25,4 mm) resultieren z.B. 10 mm (siehe oben). Beim Quittieren des Dialogs mit **OK** passiert *kein* komplementärer Fehler, so dass die unerwünscht geminderten Ränder zum **PageSettings**-Objekt übertragen werden. Leider schafft es Microsoft nicht, den seit Jahren bekannten Fehler zu beheben (KB Article ID 814355), und empfiehlt stattdessen, via Windows-Systemsteuerung das Maßsystem von **Metrisch** auf **US-Maße** zu ändern. Bei Kühnel (2006, Abschnitt 23.5.3) findet sich der weit-aus bessere Vorschlag, die **PageSetupDialog**-Eigenschaft **EnableMetric** auf den Wert **true** zu setzen:

```
pageSetupDialog.EnableMetric = true;
```

- Mit der vorgeschlagenen Korrektur werden die Randeinstellungen korrekt angezeigt:

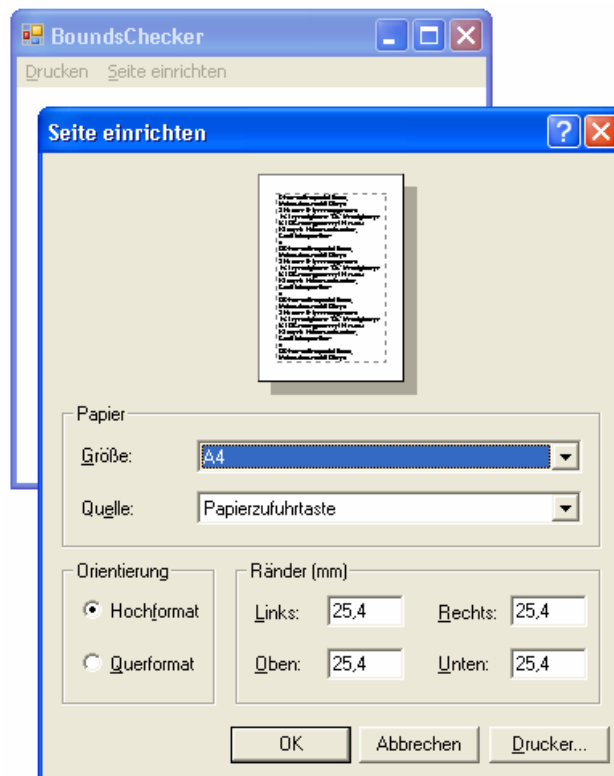


20.4 Übungsaufgaben zu Abschnitt 20

1) Erstellen Sie ein Programm, das für die **PageSettings** zu einen Druckauftrag folgende Koordinaten ausgibt:

- erste und letzte horizontale Koordinate der Seite
- erste und letzte horizontale Koordinate der bedruckbaren Seite
- erste und letzte horizontale **Graphics**-Koordinate der bedruckbaren Seite
- erste und letzte horizontale Koordinate der Seite bei Berücksichtigung der Ränder

Das Ausgabegerät und die Seiteneinstellungen sollten über einen **PageSetupDialog** wählbar sein, z.B.:



Beachten Sie die Hinweise zum **PageSetupDialog**-Bug in Abschnitt 20.3.2. Um faktische Ausdrücke zu vermeiden, sollten Sie im **PrintPage**-Handler die **Cancel**-Eigenschaft des **PrintPageEventArgs**-Objekts auf **true** setzen.

21 Zwischenablage, Ziehen und Ablegen

Mit Hilfe der Zwischenablage kann man 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). Beim Kopieren oder Ausschneiden legt die Quellanwendung in der Regel Daten in mehreren Formaten in der Zwischenablage ab (z.B. einfachen und formatierten Text). Beim Einfügen prüft die Zielanwendung, ob sie eines der verfügbaren Formate verarbeiten kann.

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 in den **Click**-Behandlungsmethoden zu den Menüitems **Bearbeiten > Kopieren** bzw. **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 aber aus didaktischen Gründen in unserem Editorprojekt vorübergehend mit den zugrunde liegenden und allgemein verwendbaren FCL-Klassen für den Datenaustausch via Zwischenablage arbeiten.

Ein solcher Datenaustausch kann zwischen *verschiedenen* Anwendungen, aber auch *innerhalb* einer Anwendung stattfinden. Speziell im zuletzt genannten Fall kommen dabei auch sehr spezielle Daten in Frage, 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 Dateien etc. per Maus (engl.: *Drag & Drop*) behandelt, weil teilweise dieselben Klassen wie beim Zwischenablagenzugriff zum Einsatz kommen.

21.1 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 Menüitem **Format > Schriftart**) mit Hilfe der **RichTextBox**-Eigenschaften **SelectionFont** und **SelectionColor** auf die aktuelle Auswahl:

```
protected void MitFontOnClick(object sender, EventArgs e) {
    Font oldFont = editor.SelectionFont;
    Color oldForeColor = editor.SelectionColor;
    DialogResult result;
    result = fontDialog.ShowDialog();
}
```

```

if (result == DialogResult.OK) {
    editor.SelectionFont = fontDialog.Font;
    editor.SelectionColor = fontDialog.Color;
} else if (result == DialogResult.Cancel) {
    editor.SelectionFont = oldFont;
    editor.SelectionColor = oldForeColor;
}
}

protected void FontDialogOnApply(object sender, EventArgs e) {
    editor.SelectionFont = fontDialog.Font;
    editor.SelectionColor = fontDialog.Color;
}

```

Später kann der Bedienkomfort z.B. durch Symbolleistenschalter für die Attribute **fett** und *kursiv* noch gesteigert werden.

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 21.4.2).

21.2 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):

DataFormats-Felder		Win32-ID	Bedeutung
Name	Inhalt		
Text	Text	1	Text in 8-Bit-ANSI-Kodierung
Bitmap	Bitmap	2	Geräteabhängiges Bitmap-Format
MetafilePict	MetafilePict	3	Traditionelles Metafile-Format
SymbolicLink	SymbolicLink	4	SYLK-Format für Tabellenkalkulationsdaten
Dif	DataInterchangeFormat	5	DIF-Format für Tabellenkalkulationsdaten (<i>Data Interchange Format</i>)
Tiff	TaggedImageFileFormat	6	TIFF-Bitmapformat
OemText	OEMText	7	Text in 8-Bit-ASCII-Kodierung
Dib	DeviceIndependentBitmap	8	Geräteunabhängiges Bitmap-Format (<i>Device Independent Bitmap</i>)
Palette	Palette	9	Farbpalette zur gemeinsamen Verwendung mit geräteunabhängigen Bitmaps mit 256 Farben
PenData	PenData	10	Format für Stift-Eingabegeräte
Riff	RiffAudio	11	RIFF-Audioformat
WaveAudio	WaveAudio	12	Wave-Audioformat
UnicodeText	UnicodeText	13	Text in Unicode-Kodierung
Enhanced-Metafile	EnhancedMetafile	14	erweitertes Windows - Metafile-Format

DataFormats-Felder		Win32-ID	Bedeutung
Name	Inhalt		
FileDrop	FileDrop	15	Liste von Dateien, z.B. per Explorer mit Strg+C angelegt
Locale	Locale	16	Windows-Kulturformat für Textdaten in der Zwischenablage
CommaSeparated Value	Csv		Austauschformat für Tabellenkalkulations- und Datenbankprogramme
Html	HTML Format		HTML-Format
Rtf	Rich Text Format		RTF-Format
Serializable	WindowsForms10-PersistentObject		Beliebiges WinForms-Objekt
StringFormat	System.String		Ergebnis bei Ablage von String -Objekten durch .NET - Anwendungen

Viele Zwischenablagenformate werden vom .NET - Framework nicht direkt unterstützt (z.B. Riff, Tiff), doch Ihre Anwendung kann diese Daten natürlich mit eigenen Algorithmen beliebig geschickt und kreativ verarbeiten.

21.3 Ein Zwischenablagen-Inspektor

Das folgende Programm zeigt die verfügbaren Formate des aktuellen Zwischenablageninhalts an. Optional erscheinen auch die .NET - Typen, die aus der Zwischenablage automatisch generiert werden können:

```
using System;
using System.Windows.Forms;
using System.Drawing;
class ClipboardInspector : Form {
    ListBox libClipboard;
    CheckBox cbAutoConv;

    public ClipboardInspector() {
        Text = "Clipboard-Inspektor";
        ClientSize = new Size(280, 190);

        Label labClipboard = new Label();
        labClipboard.Text = "Akt. Formate der Zwischenablage (Win32-ID):";
        labClipboard.Location = new Point(20, 20);
        labClipboard.Size = new Size(260, 20);
        labClipboard.Parent = this;

        libClipboard = new ListBox();
        libClipboard.Location = new Point(20, 40);
        libClipboard.Size = new Size(240, 100);
        libClipboard.Parent = this;

        Button btnAkt = new Button();
        btnAkt.Text = "Aktualisieren";
        btnAkt.Width = 80;
        btnAkt.Parent = this;
        btnAkt.Location = new Point(180, 150);
        AcceptButton = btnAkt;
        btnAkt.Click += new EventHandler(ButtonOnClick);

        cbAutoConv = new CheckBox();
        cbAutoConv.Text = "Konvertierungs-Formate";
        cbAutoConv.Parent = this;
        cbAutoConv.Width = 150;
        cbAutoConv.Checked = true;
        cbAutoConv.Location = new Point(20, 150);
        cbAutoConv.Click += new EventHandler(CheckBoxOnClick);

        ButtonOnClick(this, EventArgs.Empty);
    }
}
```

```

protected void ButtonOnClick(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);
    }
}
protected void CheckBoxOnClick(object sender, EventArgs ea) {
    ButtonOnClick(this, EventArgs.Empty);
}
[STAThread]
static void Main() {
    Application.Run(new ClipboardInspector());
}
}

```

Nachdem der WWW-Browser Firefox 2.0 eine HTML-Passage in die Zwischenablage geschrieben hat, zeigt der Inspektor folgende Formate an:



Weil wir der **Clipboard**-Inspektor möglichst früh als Hilfsmittel zur Verfügung haben wollen, müssen wir bei seiner Implementierung dem Stoff etwas vorgreifen und die statische Methode **GetDataObject()** der Klasse **Clipboard** verwenden, die eigentlich erst in Abschnitt 21.5.2 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 Methoden

string[] GetFormats()

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 Screenshots 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**:

Dataformats.Format GetFormat(string formatName)

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 Abschnitt 21 vorgestellten Methoden zum Zwischenablagenzugriff keine **ThreadStateException** werfen, muss die **Main()**-Methode das Attribut **STAThread** erhalten:

```
[STAThread]
static void Main() {
    Application.Run(new RitchisTexteditor());
}
```

21.4 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:

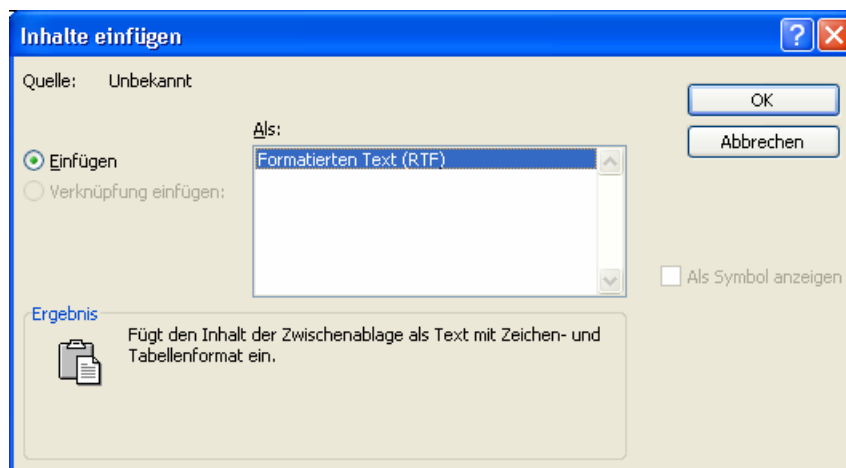
```
Clipboard.Clear();
```

21.4.1 Formatspezifische Set-Methoden der Klasse Clipboard

Seit der .NET - Version 2.0 bietet die Klasse **Clipboard** etliche Methoden, um Daten in speziellen Formaten in die Zwischenablage zu übertragen, so dass sich die meisten Routineaufgaben bequem erledigen lassen. Die folgende **Click**-Behandlungsmethode **MitCopyOnClick()** zum Menüitem **Bearbeiten > Kopieren** unseres Editors verwendet die **SetText()**-Methode, um den markierten Text des **RichTextBox**-Steuerelements (inklusive Formatierungen) in die Zwischenablage zu schreiben:

```
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:



Mit den folgenden **Clipboard**-Methoden lassen sich viele Routineaufgaben bei der Ablage von speziellen Zwischenablageformaten erledigen:

- **SetText(String s)**
Je nach Betriebssystem entsteht das Zwischenablageformat **Text** oder **UnicodeText**.
- **SetText(String s, TextDataFormat format)**
Der zweite Parameter vom Enumerationstyp **TextDataFormat** definiert das Zwischenablageformat. Es sind folgende Werte erlaubt (in Klammern das resultierende Zwischenablagenformat gemäß Abschnitt 21.2):

- **CommaSeparatedValue (Csv)**
- **Html (Html Format)**
- **Rtf (Rich Text Format)**
- **Text (Text)**
- **UnicodeText (UnicodeText)**
- **SetImage(Image bitmap)**
Im der Zwischenablage entstehen die Formate **Bitmap** und **System.Drawing.Bitmap**.
- **SetAudio(Stream audioStream)**
Im der Zwischenablage entsteht das Format **WaveAudio**.
- **SetFileDropList(StringCollection pfadnamen)**
Im der Zwischenablage landet eine Liste von Dateinamen.
- **SetData(String format, object daten)**
Mit dem ersten Parameter legt man das gewünschte Zwischenablageformat fest.

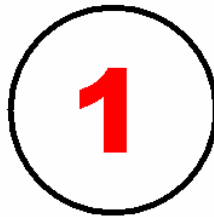
Für die formatspezifischen **Clipboard**-Methoden gilt:

- In der Zwischenablage entsteht eine *Kopie* der Daten.
- Nach dem Programmende stehen die Zwischenablagendaten anderen Programmen *weiter* zur Verfügung.

21.4.2 Die Clipboard-Methode SetDataObject()

Die seit .NET 1.0 verfügbare **Clipboard**-Methode **SetDataObject()** überträgt Instanzen von Werttypen sowie beliebige Objekte aus serialisierbaren Klassen (vgl. Abschnitt 11.2.3) in die Zwischenablage und erlaubt dabei auch eine Ablage per *Referenz*. Es sind zwei Überladungen vorhanden:

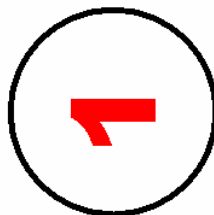
- **SetDataObject(object data);**
Instanzen von Werttypen und **String**-Objekte werden in die Zwischenablage *kopiert*. Von Objekten anderer Klassen (z.B. **Image**) landet dort eine *Referenz*, so dass beim späteren Zwischenablagenzugriff der aktuelle Objektzustand geliefert wird. Die Rastergrafik



wird durch folgende Zeilen per Referenz in die Zwischenablage übertragen und anschließend rotiert:

```
Bitmap image = new Bitmap("eins.bmp");
Clipboard.SetDataObject(image);
image.RotateFlip(RotateFlipType.Rotate90FlipXY);
```

Ein Konsument bezieht folglich aus der 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 ein erfolgreich abgerufener Zwischenablageninhalt auch *nach* Beenden des Quellprogramms weiter verfügbar ist.

- **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.

Die Methode **SetDataObject()** konvertiert nach Möglichkeit automatisch zwischen den .NET - Datentypen und den von anderen Anwendungen verwendeten Formaten. Wenn Sie z.B. ein **String**-Objekt ablegen, dann landen in der Zwischenablage neben **System.String** auch die Formate **Text** und **UnicodeText**.

Neben der Option zur Ablage per Referenz hat die Methode **SetDataObject()** vor allem die Möglichkeit zur **simultanen Ablage mehrerer Formate** zu bieten. In Abschnitt 21.4.1 wurde gezeigt, dass die **Copy**-Methode unseres Editors mit folgendem **SetText()**-Aufruf RTF-Daten in die Zwischenablage schreiben kann:

```
Clipboard.SetText(editor.SelectedRtf, TextDataFormat.Rtf);
```

Mit dem folgenden **SetDataObject()**-Aufruf

```
Clipboard.SetDataObject(editor.SelectedRtf, true);
```

erhalten wir die Zwischenablagenformate **Text**, **UnicodeText** und **System.String**, so dass z.B. MS-Word vorhandenen RTF-Text nicht mehr erkennt und beim Einfügen falsch behandelt. 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 21.4.1). Zur Methode **SetData()** besitzt die Klasse **DataObject** eine zusätzliche Überladung mit Einfluss auf die automatische Konvertierung in kompatible Formate:

```
SetData(string format, object daten)
```

```
SetData(string format, bool autoConvert, object daten)
```

Anschließend wird das komplette Objekt über die statische **Clipboard**-Methode **SetDataObject()** in die Zwischenablage befördert:

```
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 hier die in Abschnitt 21.2 beschriebenen Felder der Klasse **DataFormats**.

Unser Inspektor aus Abschnitt 21.2 zeigt, dass nun alle gewünschten Formate in der Zwischenablage landen:



Wenn **SetData()** keine automatische Konvertierung in kompatible Formate vornehmen soll, verwendet man die zweite Überladung aus obiger Liste und setzt den Parameter *autoConvert* auf **false**.

21.5 Daten aus der Zwischenablage abrufen

21.5.1 Formatspezifische Get-Methoden der Klasse Clipboard

Seit der .NET - Version 2.0 hat die Klasse **Clipboard** etliche formatspezifische Methoden zum Abrufen von Zwischenablageinhalten hinzu gewonnen, so dass sich die meisten Routineaufgaben bequem erledigen lassen. Die folgende **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.

Mit den folgenden **Get**-Methoden der Klasse **Clipboard** lassen sich viele Routineaufgaben beim Abruf von speziellen Zwischenablageformaten erledigen, wobei mit einer korrespondierenden **Contains()**-Methode die Existenz von Daten im gewünschten Format festgestellt werden kann:

- **bool ContainsText(), String GetText()**
ContainsText() liefert den Wert **true** zurück, wenn Textdaten in der Zwischenablage vorhanden sind, ansonsten **false**. Vorhandener Text lässt sich mit **GetText()** in ein **String**-Objekt befördern.
- **bool ContainsText(TextDataFormat format), 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 Textformate erlaubt (in Klammern die zugehörigen Zwischenablagenformatnamen gemäß Abschnitt 21.2):
 - **CommaSeparatedValue (Csv)**
 - **Html (Html Format)**
 - **Rtf (Rich Text Format)**
 - **Text (Text)**
 - **UnicodeText (UnicodeText)**

- **bool ContainsImage(Image bitmap), Image GetImage(Image bitmap)**
ContainsImage() liefert den Wert **true** zurück, wenn die Zwischenablage Bitmap-Daten enthält, ansonsten **false**. Aus vorhandenen Bitmap-Daten lässt sich mit **GetImage()** ein **Image**-Objekt erstellen.
- **bool ContainsAudio(), 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.
- **bool ContainsData(String format), 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.
- **bool ContainsFileDropList(), 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.

21.5.2 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 21.6) 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.¹ Folglich können wir das Objekt mit der Methode

bool GetDataPresent(string format)

bool GetDataPresent(Type typ)

bool GetDataPresent(string format, bool autoConvert)

befragen, ob Daten mit einem bestimmtem Format vorhanden sind. So wird die in Abschnitt 21.5.1 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 beide Varianten der Formatspezifikation vorgeführt (per **String** und per **Type**-Objekt).

Sollen bei der Existenzprüfung mögliche Konvertierungen unberücksichtigt bleiben, verwendet man die dritte **GetDataPresent()**-Überladung und setzt den booleschen Parameter *autoConvert* auf **false**.

¹ Nach Petzold (2003, S. 1018) handelt es sich in der Regel um ein Objekt der Klasse **DataObject**, die wir bereits kennen gelernt haben.

Zur Entnahme von Daten mit bestimmtem Format verwendet man die **IDataObject**-Methode **GetData()**, wobei das Format wie bei **GetDataPresent()** durch seinen Namen oder einen .NET - Datentyp beschrieben wird:

object GetData(string format)

object GetData(Type typ)

object GetData(string format, bool autoConvert)

GetData() nimmt u.a. folgende Konvertierungen vom angeforderten Zwischenablageformat in einen FCL - Typ vor, wenn man dies nicht mit dem *autoConvert*-Parameterwert **false** untersagt:

angefordertes Zwischenablageformat	resultierender FCL – Typ
Text, OemText, UnicodeText, SymbolicLink, DataInterchangeFormat, HTML Format, Rich Text Format, System.String	System.String (bei HTML Format und Rich Text Format mit Formatierungs-Tags im Ergebnis-String)
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

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:

```
dataObject.GetData(DataFormats.StringFormat);
```

Es resultiert hingegen ein **null**-Objekt, wenn man das automatische Konvertieren verbietet:

```
dataObject.GetData(DataFormats.StringFormat, false);
```

Weil **GetData()** den Rückgabetypp **object** besitzt, ist zur weiteren Verarbeitung in der Regel eine explizite Typumwandlung erforderlich, z.B.

```
String s = (string) dataObject.GetData(DataFormats.StringFormat);
```

Das von **GetDataObject()** gelieferte **IDataObject** hat sich übrigens schon bei dem in Abschnitt 21.3 vorgestellten Clipboard-Inspektor nützlich gemacht. In der folgenden Anweisung liefert es einen **String**-Array mit den Namen der vorhandenen Zwischenablagenformate:

```
string[] sF = Clipboard.GetDataObject().GetFormats(cbAutoConv.Checked);
```

Bei der Einsatzversion unseres Editors greifen wir in **MitPasteOnClick()** nach erfolgreichem Studium der **Clipboard**-Methoden nun wieder auf die **RichTextBox**-Methode **Paste()** zurück, die bequemer ist und dabei auch noch weitere Formate unterstützt (z.B. Bitmaps, Metadateien):

21.6 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.

Als Quelle bzw. Ziel kommen **Form**-Objekte und Steuerelemente in Frage, so dass ein Datenaustausch *zwischen* Anwendungen und auch *innerhalb* einer Anwendung realisieren lässt (z.B. zur Erweiterung einer Symbolleiste um neue Schaltflächen).

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 beliebigen 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.

21.6.1 Die Rolle der Zielanwendung

Ein **RichTextBox**-Steuerelement funktioniert als *Quelle* von Texten, beherrscht aber die mindestens ebenso wichtige Zielrolle *nicht*, so dass wir im Editorprojekt zunächst die Empfängerfunktionalität nachrüsten. In Abschnitt 21.6.2 werden wir aber auch das Verhalten des Editors in der Quellenrolle nachbessern.

Im Vergleich zu anderen Darstellungen sind mehr (zunächst belastende) technische Details zu finden, die zu praxisnahen Lösungen führen sollen. Es wird z.B. verhindert, dass aufgrund der neuen Drag & Drop - Funktionalität die **RichTextBox**-Kompetenz, ein Wort per Doppelklick komplett zu markieren, verloren geht. In den meisten Textverarbeitungsprogrammen (z.B. MS-Word oder WordPad) sind beide Mausfunktionen nutzbar.

Soll ein Steuerelement per Drag & Drop Daten entgegen nehmen, muss seine Eigenschaft **AllowDrop** auf **true** gesetzt werden (Voreinstellung: **false**), z.B.:

```
public RitchisTexteditor() {
    . . .
    editor.AllowDrop = true;
    . . .
}
```

Damit es nicht an Stelle der gewünschten Funktionalität zu Laufzeitfehlern kommt, muss die **Main()**-Methode das Attribut **STAThread** erhalten:

```
[STAThread]
static void Main() {
    Application.Run(new RitchisTexteditor());
}
```

Wie man es erwarten durfte, geht es beim Ziehen und Ablegen recht ereignisreich zu, falls **AllowDrop** den Wert **true** besitzt:

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.
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. Im Editorprojekt findet sich auch für den **DragEnter**-Ereignis eine gleich zu erläuternde Nutzungsmöglichkeit:

```
editor.DragEnter += new DragEventHandler(EditorOnDragEnter);
editor.DragOver += new DragEventHandler(EditorOnDragOver);
editor.DragDrop += new DragEventHandler(EditorOnDragDrop);
```

Den Ereignisbehandlungsmethoden wird ein Objekt der Klasse **DragEventArgs** als Aktualparameter übergeben, das folgende Eigenschaften besitzt:

Eigenschaft	Zweck												
X, Y	Mauskoordinaten bezogen auf den Bildschirm Diese müssen in der Regel mit der Control -Methode PointToClient() (siehe Abschnitt 14.4.1) in Klientenbereichskoordinaten umgerechnet werden.												
Data	Dieses Objekt erfüllt die Schnittstelle IDataObject , kann also gemäß der Beschreibung in Abschnitt 21.5.2 genutzt werden.												
AllowedEffect	Von der Quellanwendung wird über die bitweise kombinierbaren Werte der Enumeration DragDropEffects festgelegt, was die Zielanwendung mit den Daten tun darf, z.B.: <table><tr><td>None</td><td>0x00000000</td><td>Link</td><td>0x00000004</td></tr><tr><td>Copy</td><td>0x00000001</td><td>All</td><td>0x00000003</td></tr><tr><td>Move</td><td>0x00000002</td><td></td><td></td></tr></table>	None	0x00000000	Link	0x00000004	Copy	0x00000001	All	0x00000003	Move	0x00000002		
None	0x00000000	Link	0x00000004										
Copy	0x00000001	All	0x00000003										
Move	0x00000002												
Effect	Diese Eigenschaft ist vom selben Typ wie AllowedEffect , erlaubt aber einen												

Eigenschaft	Zweck								
	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.								
KeyState	Bei ihren Verarbeitungsabsichten orientiert sich die Zielanwendung an den Genehmigungen in AllowedEffects 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 Zielanwendung so interpretiert: <table border="0"> <tr> <td>Linke Maustaste</td><td>Verschieben (innerhalb eines Laufwerks)</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)	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)								
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 Editors eine Dateienliste (Zwischenablagenformat **DataFormats.FileDrop**) oder(formatierten) Text (Zwischenablagenformat **DataFormats.Text** bzw. **DataFormats.Rtf**) im **DragEventArgs**-Ereignisobjekt vorfindet, signalisiert sie per **Effect**-Eigenschaft die Bereitschaft zum Verschieben oder zum Kopieren, wobei die Benutzerabsicht via **KeyState**-Eigenschaft ausgewertet wird:

```

void EditorOnDragOver(object sender, DragEventArgs dea) {
    if ((dea.KeyState & 1) != 1)
        return;
    if (dea.Data.GetDataPresent(DataFormats.FileDrop)) {
        dea.Effect = DragDropEffects.Copy;
    } else {
        if (dea.Data.GetDataPresent(DataFormats.Text) ||
            dea.Data.GetDataPresent(DataFormats.Rtf)) {
            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;
            }
        }
    }
    if (dea.Data.GetDataPresent(DataFormats.Text) ||
        dea.Data.GetDataPresent(DataFormats.Rtf)) {
        editor.SelectionLength = 0;
        Point pt = editor.PointToClient(new Point(dea.X, dea.Y));
        int insPos = editor.GetCharIndexFromPosition(pt);
        if (pt.X > editor.GetPositionFromCharIndex(insPos).X + 10)
            insPos++;
        editor.SelectionStart = insPos;
    }
}

```

Wir fühlen uns nur angesprochen, wenn die linke Maustaste gedrückt ist, was durch einen Bit-Vergleich der **DragEventArgs**-Eigenschaft **KeyState** mit der Eins zu ermitteln ist:

```
if ((dea.KeyState & 1) != 1)
    return;
```

Am Ende der Methode wird dafür gesorgt, dass der Benutzer per Einfügemarke erkennt, wo einzufügender Text erscheinen wird:



Zunächst werden die im **DragEventArgs**-Objekt gelieferten Mauskoordinaten auf den Klientenbereich bezogen:

```
Point pt = editor.PointToClient(new Point(dea.X, dea.Y));
```

Zur so gewonnenen **Point**-Instanz liefert die **RichTextBox**-Methode **GetCharIndexFromPosition()** die Nummer des nächstgelegenen Zeichens:

```
int insPos = editor.GetCharIndexFromPosition(pt);
```

Wenn die **RichTextBox**-Eigenschaft **SelectionStart** diesen Wert erhält, erscheint dort die Einfügemarke:

```
editor.SelectionStart = insPos;
```

Befindet sich die Maus hinter dem letzten Zeichen, ist der gemeldete Indexwert um 1 zu klein. Diesen korrigiert die folgende Anweisung:

```
if (pt.X > editor.GetPositionFromCharIndex(insPos).X + 10)
    insPos++;
```

Das Verfahren zur Anzeige der Einfügeposition hat den Nachteil, dass die Ausgangsposition der Einfügemarke sowie eine eventuelle Markierung verloren gehen, was aber nur nach dem Abbruch einer (Drag & Drop) - Aktion relevant ist. Leider bietet die Klasse **RichTextBox** kein Ereignis, um auf den Abbruch geeignet reagieren zu können.

Mit dem folgenden **DragEnter**-Handler

```
void EditorOnDragEnter(object sender, DragEventArgs dea) {
    editor.Focus(); // Caret() sichtbar machen
}
```

gelingt es, die Einfügestelle bei importiertem Text (aus einer fremden Anwendung stammend) sichtbar zu machen. Zur Quellfunktionalität unserer eigenen Anwendung kommen wir schließlich erst später.

In der **DragDrop**-Behandlungsmethode zeigt sich deutlich die Parallelität zur Zwischenablagentechnik. Sie erhält per **DragEventArgs**-Parameter ein Objekt, das die **IDataObject**-Schnittstelle erfüllt, also potentiell Daten in verschiedenen Formaten enthält und mit der Methode **GetDataPresent()** befragt werden kann (vgl. Abschnitt 21.5.2). Bei diesen Anfragen eignen sich zur Bezeichnung der Formate die in Abschnitt 21.2 behandelten **DataFormats**-Felder. Zu importierender (RTF-)Text ersetzt die aktuelle Markierung (**RichTextBox**-Eigenschaft **SelectedRtf** bzw. **SelectedText**), welche aufgrund der **DragOver**-Behandlungsmethode grundsätzlich die Länge 0 hat:


```

void EditorOnDragDrop(object sender, DragEventArgs dea) {
    if (dea.Data.GetDataPresent(DataFormats.Rtf)) {
        editor.SelectedRtf = (string) dea.Data.GetData(DataFormats.Rtf);
    } else
    if (dea.Data.GetDataPresent(DataFormats.StringFormat)) {
        editor.SelectedText =
            (string) dea.Data.GetData(DataFormats.StringFormat);
    } else
    if (dea.Data.GetDataPresent(DataFormats.FileDrop)) {
        String[] fns = (string[]) dea.Data.GetData(DataFormats.FileDrop);
        FileStream fs = null;
        try {
            fs = new FileStream(fns[0], FileMode.Open,
                               FileAccess.Read, FileShare.Read);
            editor.Text = "";
            editor.LoadFile(fs, RichTextBoxStreamType.PlainText);
            if (editor.Text.Substring(2, 4).IndexOf("rtf1") == 0)
                editor.Rtf = editor.Text;
            fileName = fns[0];
            Text = Path.GetFileName(fileName) + " - " + strAppName;
            editor.Modified = false;
        } catch {
            MessageBox.Show("Fehler beim Öffnen der Datei " + fns[0],
                             strAppName, MessageBoxButtons.OK, MessageBoxIcon.Error);
        } finally {
            fs.Close();
        }
    }
}

```

Liegen **FileDrop**-Daten vor, benimmt sich der Editor ungefähr so wie beim Öffnen einer Textdatei. RTF-Daten werden am charakteristischen Textanfang erkannt.

Für das **DragLeave**-Ereignis hätte sich im Editorprojekt durchaus ein Nutzen gefunden, doch wird es offenbar vom **RichTextBox**-Steuerelement *nicht* gefeuert.

21.6.2 Die Rolle der Quellenanwendung

Die (Drag & Drop)-Quellenanwendung ergreift die Initiative durch Aufruf der **Control**-Methode **DoDragDrop()**, die folgende Aktualparameter erhält:

- **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 Editor übergibt ein Objekt der Klasse **DataObject**, das neben dem formatierten Text noch begleitende Informationen enthält, und erlaubt sowohl das Kopieren als auch das Verschieben:

```

DataObject dataObject = new DataObject();
RitchSelection rs = new RitchSelection();
rs.Start = selStart;
rs.Length = selLength;
dataObject.SetData("Selected in RitchisTextBox", rs);
dataObject.SetData("Rich Text Format", editor.SelectedRtf);
DragDropEffects ddResult = editor.DoDragDrop(dataObject,
        DragDropEffects.Copy | DragDropEffects.Move);

```

Das **DataObject** enthält in einem speziellen Datenformat Start und Länge einer Textmarkierung, wobei die folgende Struktur als Datentyp dient:

```
struct RitchSelection {
    public int Start, Length;
}
```

Zweck ist, bei internem Transport das Einfügen in die Markierung zu verhindern. Dazu wird in einer Erweiterung des **DragOver**-Handlers die Markierungsinformation ausgewertet und ggf. über den **DragDropEffect None** das Einfügen verhindert:

```
if (dea.Data.GetDataPresent("Selected in RitchisTextBox")) {
    RitchSelection sel =
        (RitchSelection)dea.Data.GetData("Selected in RitchisTextBox");
    if (editor.SelectionStart < sel.Start + sel.Length &&
        editor.SelectionStart > sel.Start)
        dea.Effect = DragDropEffects.None;
}
```

Weil das Ziehen und Ablegen mit dem Drücken der linken Maustaste startet, rufen wird **DoDragDrop()** oft in einer **MouseDown**-Ereignismethode

```
editor.MouseDown += new MouseEventHandler(EditorOnMouseDown);
```

auf, wenn die linke Taste betroffen und markierter Text vorhanden ist:

```
protected void EditorOnMouseDown(object sender, MouseEventArgs e) {
    int selStart, selLength;
    bool within = false;
    if (e.Button == MouseButtons.Left && editor.SelectionLength > 0) {
        selStart = editor.SelectionStart;
        selLength = editor.SelectionLength;
        int textLength = editor.Text.Length;
        DataObject dataObject = new DataObject();
        RitchSelection rs = new RitchSelection();
        rs.Start = selStart;
        rs.Length = selLength;
        dataObject.SetData("Selected in RitchisTextBox", rs);
        dataObject.SetData("Rich Text Format", editor.SelectedRtf);
        DragDropEffects ddResult = editor.DoDragDrop(dataObject,
            DragDropEffects.Copy | DragDropEffects.Move);
        if (editor.Text.Length > textLength)
            within = true;
        if (ddResult == DragDropEffects.Move) {
            if (within) {
                if (editor.SelectionStart > selStart) { //nach rechts verschoben
                    editor.SelectionStart = selStart;
                } else { // nach links verschoben
                    editor.SelectionStart = selStart + selLength;
                }
            } else {
                editor.SelectionStart = selStart;
            }
            editor.SelectionLength = selLength;
            editor.Cut();
        }
        if (!within) {
            dataObject.SetData("Drag & Drop Canceled", new Object());
            editor.SelectionStart = selStart;
        }
    }
}
```

Der **DoDragDrop()**-Aufruf kehrt erst dann zurück, wenn die Aktion abgeschlossen oder per **Esc**-Taste abgebrochen wurde.

Ist der Text nach Ausführung des **DragDrop**-Handlers länger, dann hat ein *interner* Texttransport stattgefunden:

```
if (editor.Text.Length > textLength)
    within = true;
```

Nach einer Verschiebung (Rückgabewert **DragDropEffects.Move**) ist der Text am alten Ort zu löschen. Dies geschieht über die **RichTextBox**-Methode **Cut()**, damit die Aktion per **Undo()** zurück genommen werden kann. Bei den Positionsberechnungen ist die Bewegungsrichtung (vor oder zurück) und das Ziel (intern oder extern) zu berücksichtigen.

Nach einem Texttransport in eine andere Anwendung wird leider trotzdem das Ereignis **DragDrop** in der eigenen Anwendung ausgelöst, so dass der Text auch hier erscheint. Um dies zu verhindern, fügen wir ein Signal als eigenes Format in das **DataObject** ein

```
dataObject.SetData("Drag & Drop Canceled", new Object());
```

und nutzen diese Information im **DragDrop**-Handler:

```
if (dea.Data.GetDataPresent("Drag & Drop Canceled"))
    return;
```

Der **DoDragDrop()**-Aufruf im **MouseDown**-Handler verhindert leider das Markieren eines Worts per Doppelklick. Um das Problem zu lösen, wird die Methode **DoDragDrop()** nur dann aufgerufen, wenn seit dem letzten Mausklick eine längere Zeit vergangen, wenn also kein Doppelklick vorliegt:

```
long diff = DateTime.Now.Ticks - lastClick;
lastClick = DateTime.Now.Ticks;
if (diff < 3000000) {
    lastClick = DateTime.Now.Ticks;
    return;
}
```

Unser Editor beherrscht nun das interne Verschieben und Kopieren von Textpassagen, die kopierende oder verschiebende Übergabe an andere Anwendungen sowie den kopierende oder verschiebende Textimport, wobei ein gründlicher Betatest allerdings noch aussteht. Leider befindet sich die Einfügemarke nach einer erfolgreichen oder abgebrochenen (Drag & Drop) - Aktion nicht immer an der erwarteten Stelle. Den vollständigen Quellcode mit dem aktuellen Entwicklungsstand finden Sie im Ordner

...\BspUeb\Zwischenablage\RitchisTextEditor

21.7 Übungsaufgaben zu Abschnitt 21

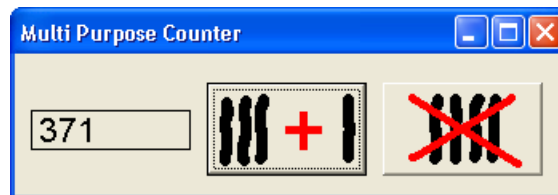
1) Wie Sie sicher wissen, kann die Windows-Zwischenablage nur *ein* Element aufnehmen. Ist es trotzdem möglich, dass sich dort z.B. gleichzeitig ein Bild und ein Text befinden?

22 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. 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 dem Abschnitt 8.4.2.3, 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.



22.1 Ressourcendateien und Generatoren

Das .NET-Framework benutzt für Ressourcen zwei äquivalente und bijektiv 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. Weitere Informationen finden sich z.B. im SDK-Hilfethema *Hinzufügen und Bearbeiten von Ressourcen (Visual C# Express)*.

22.1.1 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).

In der RESX-Datei folgen auf die XML-Schemadefinition zur Beschreibung der Struktur

```

<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>

```

zunächst einige **<resheader>**-Tags mit einer Namensangabe und einem eingebetteten **<value>**-Tag, 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>**-Tag mit Namensangabe und einem eingebetteten **<value>**-Tag, z.B. bei einer Zeichenfolge

```

<data name="TitelText">
  <value>Multi Purpose Counter</value>
</data>

```

Außer bei Zeichenfolge-Ressourcen enthält das **<data>**-Tag 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,
Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
mimetype="application/x-microsoft.net.object.bytearray.base64">
  <value>
    Qk300gAAAAAADYAAAAoAAAAZAAADIAAABABgAAAAAAAAAADEdGAAxA4AAAAAAAAAAAAA20ns20ns
    20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns
    . . .
    20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns20ns
  </value>
</data>

```

Mit einem Objekt der Klasse **ResXResourceWriter** aus dem Namensraum **System.Resources** können Sie 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"));
        Console.WriteLine("Fertig!"); Console.ReadLine();
        rsxw.Close();
    }
}

```

Zum *Lesen* einer RESX-Datei per Programm (z.B. in Ihrem Ressourcen-Editor) steht die FCL-Klasse **ResXResourceReader** bereit.

Viele C#-Entwicklungsumgebungen wie das Visual Studio und SharpDevelop bieten eine umfangreiche Unterstützung für das Erstellen, Bearbeiten und Verwenden von RESX-Dateien (vgl. Abschnitt 22.3), so dass man in der Regel nicht zu einem XML-Editor greifen wird.

22.1.2 TXT-Datei für Zeichenfolge-Ressourcen

Bei Zeichenfolgen-Ressourcen kann man eine einfach editierbare Textdatei erstellen, z.B.:

```

; Zeichenfolgen für Universalrechner - deutsch
Math_Greeting = Willkommen beim Universalrechner
Math_Formula_Label = Formel und Ergebnisse:
Math_Clear_Button = Zurücksetzen
Math_Calc_Button = Berechnen
Math_Calc_Error = Eingabefehler

```

22.1.3 RESOURCES-Datei

Aus einer RESX- oder TXT-Datei lässt sich mit dem Konsolenprogramm **resgen.exe** aus dem .NET – SDK (als Exe-Assembly vorhanden im Ordner ...\\Microsoft Visual Studio 8\\SDK\\v2.0\\Bin) eine äquivalente Binärdatei mit der Namensweiterung **.resources** erstellen, z.B.:

```

>"C:\\Programme\\Microsoft Visual Studio 8\\SDK\\v2.0\\Bin\\resgen" mpc.resx mpc.resources
Liest in den 3-Ressourcen von "mpc.resx".
Ressourcendatei wird geschrieben... Fertig.

```

resgen.exe beherrscht folgende Konvertierungen zwischen den Ressourcen-Dateiformaten:

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 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"));
        Console.WriteLine("Fertig!"); Console.ReadLine();
        rsw.Close();
    }
}

```

Die Inhalte einer RESOURCES-Datei müssen per Compiler in ein Assembly oder ein Satellitenassembly befördert werden (siehe Abschnitt 22.2).

22.2 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 leicht 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();
reset.Image = (Bitmap) rm.GetObject("Reset");

```

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 liegende 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 **ResourceManager**-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 **RE-SOURCES**-Datei automatisch zu Erstellen und an den Compiler zu übergeben.

22.3 Ressourcen-Verwaltung mit der Visual C# 2005 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.

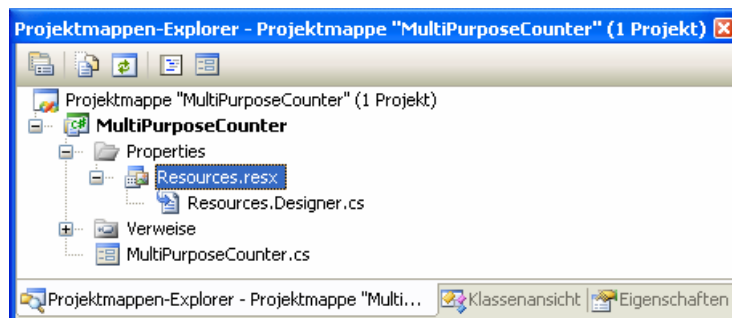
Bei einem neuen Projekt bietet die **Ressourcen**-Registerkarte der Projekteigenschaften einen bequemen Weg, um die Standardressourcendatei zu erstellen:



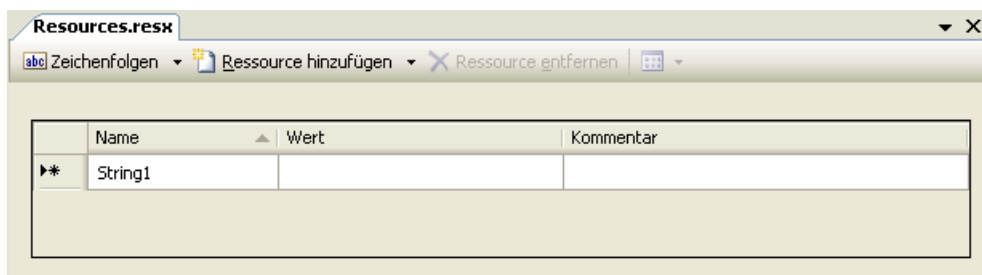
Anschließend finden sich im Projekt-Unterordner **Properties**

- die RESX-Datei **Resources.resx**
Mit diesem Dateityp haben wir uns in Abschnitt 22.1.1 beschäftigt. Später lässt die Entwicklungsumgebung durch das Hilfsprogramm **resgen.exe** im Projekt-Unterordner **...\obj\Debug** die korrespondierende Binärdatei **MPC.Properties.Resources.resources** erstellen.
- die Quellcodedatei **Resources.Designer.cs**
Hier erstellt die Entwicklungsumgebung eine Hilfsklasse für den bequemen Zugriff auf die Ressourcen, wobei im Wesentlichen die in Abschnitt 22.2 beschriebene Technik zum Einsatz kommt.

Die neuen Dateien erscheinen auch im Projektmappenexplorer:



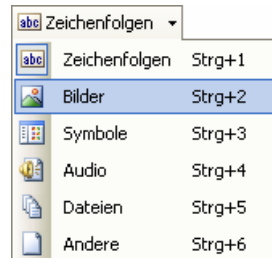
Von hier aus kann die Bearbeitung der Ressourcendatei per Doppelklick gestartet werden:



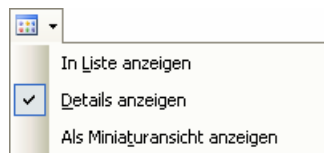
Für unser Zählprogramm tragen wir zunächst die Zeichenfolgen-Ressource ein:

	Name ▲	Wert	Kommentar
✎	TitelText	Multi Purpose Counter	
*			

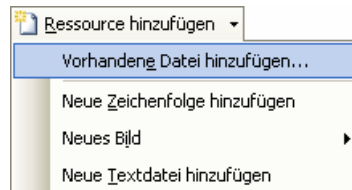
Um die Bitmap-Ressourcen zu ergänzen, wählen wir in der linken Liste die Option **Bilder**



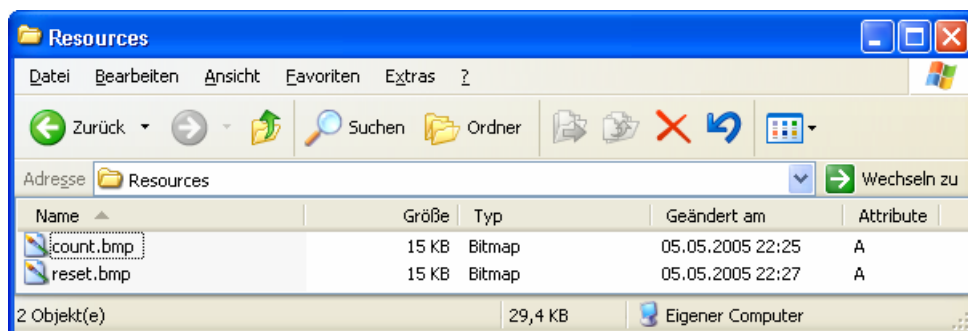
und in der rechten Liste die Option **Details anzeigen**:



Nun fügen wir eine **Ressource aus einer vorhandenen Datei** hinzu



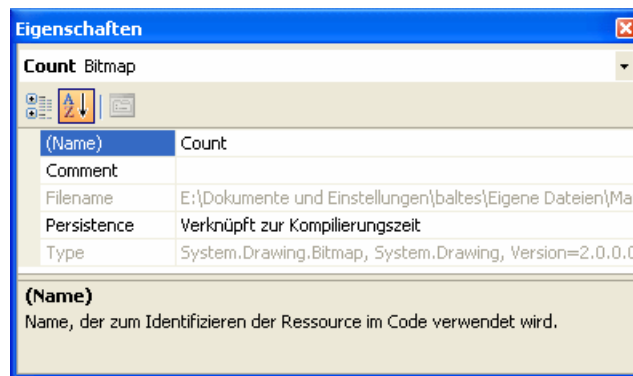
und öffnen die beiden Bitmapdateien. Diese landen in dem von Visual C# 2005 Express Edition angelegten Projekt-Unterrordner **Resources**



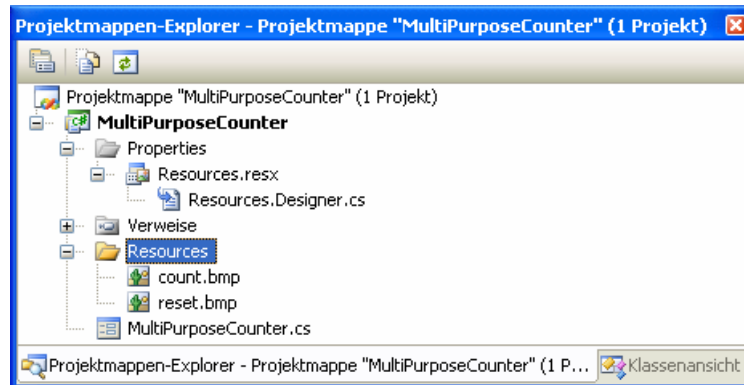
und werden in der Ressourcenliste angezeigt:

Name	Dateiname	Typ	Größe	Kommentar
count	Resources\count.bmp	Windows-Bitmap	100 x 50	
reset	Resources\reset.bmp	Windows-Bitmap	100 x 50	

Ihren identifizierenden Namen erhalten die Ressourcen via Eigenschaftsfenster, z.B.:



Im Projektmappenexplorer zeigt sich nun folgendes Bild:



In der Datei **Resources.Designer.cs** steckt eine automatisch erstellte Hilfsklasse namens **Resources**, die über statische Eigenschaften einen bequemen Zugriff auf die Ressourcen ermöglicht und dabei erwartungsgemäß ein Objekt der Klasse **ResourceManager** einsetzt (vgl. Abschnitt 22.2), das über die **Resources**-Eigenschaft (!) **ResourceManager** angesprochen wird:

```
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 unserer eigenen Klassendefinition sind die Ressourcen nun sehr bequem ansprechbar

```
Text = Resources.TitelText;
Bitmap bmp = Resources.Count;
reset.Image = Resources.Reset;
```

Weil die Entwicklungsumgebung die Hilfsklasse in einen Unternamensraum steckt, müssen wir diesen in unserer Klassendefinition berücksichtigen, z.B. per **using**-Direktive:

```
using MPC.Properties;
```

Das resultierende Exe-Assembly ist nun ohne begleitende Bitmap-Dateien lauffähig.

Den vollständigen Quellcode mit dem aktuellen Entwicklungsstand finden Sie im Ordner
...\BspUeb\Ressourcen\Multi Purpose Counter\vs

22.4 Übungsaufgaben zu Abschnitt 22

1) Integrieren Sie bei unserem Editorprojekt die Bitmap-Ressourcen in das Assembly.

23 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 Win32-API) und von Komponenten aus dem Common 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.

23.1 Funktionen in nativen DLLs nutzen

Obwohl das .NET-Framework eine ziemlich vollständige Kapselung des Win32-APIs liefert, kommt doch gelegentlich der direkte Aufruf einer API-Funktion in Frage. Die **Platform Invocation Services** (Kurzname: *PInvoke*) in der CLR erlauben die Nutzung von Funktionen in DLLs mit nicht-verwaltetem Code.

Der PInvoke-Einsatz ist nicht ganz kostenfrei zu haben: Neben Fixkosten von ca. 10 - 30 x86-Instruktionen pro Aufruf schlägt noch der Aufwand der Parameter-Konvertierung (siehe unten) zu Buche.

23.1.1 DLLs importieren

In C# wird eine nicht verwaltete DLL-Funktion folgendermaßen genutzt:

- Man schreibt einen Methodenprototypen mit den Modifikatoren **static** und **extern**.
- Dieser Methode wird das Attribut **DllImport** mit dem DLL-Dateinamen als Konstruktor-Parameter vorangestellt.

Im folgenden Beispiel wird auf diese Weise die Funktion **MessageBox()** aus der Win32-API-DLL **user32.dll** aufgerufen:

```
using System;
using System.Runtime.InteropServices;

class Win32MessageBox {
    [DllImport("user32.dll")]
    static extern int MessageBox(int hWnd, string text, string titel, int typ);

    static void Main() {
        MessageBox(0, "MessageBox-Vergleich", "Win32-API", 0);
        System.Windows.Forms.MessageBox.Show("MessageBox-Vergleich", ".NET");
    }
}
```

23.1.2 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 unter Windows NT/2K/XP Unicode (Win32-Datentyp **LPWSTR**) verwendet wird.

Für die erforderlichen Typanpassungen enthält die CLR einen so genannten **Marshaler**, der die wichtigsten Datentypen aus dem Win32-API und dem Common 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 aus dem .NET-SDK zeigt wichtige Typzuordnungen:

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
DOUBLE	double	Double

Beim Aufruf der API-Funktion **GetWindowsDirectory()** (aus der Datei **kernel32.dll**) bildet der Marshaler den ersten Parameter ab und reicht den zweiten unverändert durch:

```
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(s); Console.ReadLine();
    }
}
```

Wird eine .NET-Klasse oder -Struktur an eine C-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);
    }
}

```

23.1.3 Win32-Fehlercode ermitteln

Viele Win32-Funktionen signalisieren den (Miss)erfolg bei der Ausführung über einen booleschen Rückgabewert. Wenn Sie darüber hinaus bei negativem Ausgang am Windows-Fehlercode interessiert sind, müssen Sie ...

- im **DLLImport**-Attribut der Win32-Funktion den Konstruktor-Parameter **SetLastError** auf den Wert **true** setzen,
- nach dem Funktionsaufruf die statische 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** umzubenennen, resultiert erwartungsgemäß der Win32-Fehler 5 (Fehler beim Zugriff):

Fehler 5 bei MoveFile

23.2 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 aktuellen Windows-Betriebssysteme, und es ist 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-Komponententechnologie vermittelt werden. Das Thema ist recht anspruchsvoll und verlangt nach übereinstimmender

Expertenmeinung ca. ein halbes Jahr Einarbeitungszeit. Wir beschränken uns zudem auf die *Verwendung* von COM-Komponenten in .NET - Projekten. Über die ebenfalls mögliche *Entwicklung* von COM-Komponenten in C# oder einer anderen .NET – Sprache informieren z.B. Troelsen (2001) und MacBeth (2004).

23.2.1 Einige Begriff und Ideen aus der COM-Technologie

Mit der COM-Technologie wird die Entwicklung von

- objektorientierter und
- auf binärer Ebene programmiersprachenunabhängig wieder verwendbarer

Software ermöglicht.

Im Unterschied zu der ebenfalls binär wieder verwendbaren Software in klassischen Funktions-DLLs (z.B. mit dem Win32-API) erleichtern COM-Komponenten die Nutzung durch ein objektorientiertes Programmiermodell mit Schnittstellen (interfaces), Eigenschaften (properties), Methoden und Ereignissen.

Während in C++ erstellte Klassen nur in C++ - Programmen wieder verwendet werden können, 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** (**D**istributed **C**omponent **O**bject **M**odel) sind auch Zugriffe über Netzwerke möglich. Die mit Windows 2000 eingeführte Erweiterung **COM+** unterstützt Unternehmenslösungen mit Transaktionstechnik.

Die ausgeprägt komponentenorientierte .NET - Technologie kann in vielerlei Hinsicht als Weiterentwicklung und Vereinfachung der COM-Technologie aufgefasst werden.

23.2.1.1 Typbibliotheken und Interfaces

Gegenüber dem Nutzer einer Komponente (gemeint sind Softwareentwickler!) werden im guten objektorientierten Sinn alle Implementierungsdetails verborgen. Er muss nur die in einer **Typbibliothek** veröffentlichten **Schnittstellen** (engl.: *interfaces*) einer Komponente kennen. Solche **Metadaten** befinden sich bei einer COM-Komponente entweder in einer separaten Datei (mit der Namensendung **.tlb**) oder auch innerhalb der DLL- oder EXE-Datei mit den COM-Klassen (speziell beim Erstellen mit Visual Basic). Bei einer .NET - Komponente sind bekanntlich die Metadaten grundsätzlich im Assembly enthalten.

23.2.1.2 COM-Server und -Klienten

Als **COM-Server** wird jede ausführbare Datei (EXE oder DLL) bezeichnet, die eine COM-Klasse enthält. Es sind folgende Typen zu unterscheiden:

- **In-Process-Server** (In-Proc-Server)
Sie residieren in einer DLL-Datei und werden im Adressraum des Klienten ausgeführt, so dass der Aufruf einer In-Proc-Server - Methode kaum aufwendiger ist als der Aufruf einer klienteneigenen Methode.
- **Out-Of-Process-Server** (Out-Of-Proc-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.

Für den Klienten ist es irrelevant, ob ein COM-Objekt im selben Prozess existiert, in einem anderen Prozess derselben Maschine oder auf einem anderen Rechner.

23.2.1.3 Registrierung

Aus welcher ausführbaren Datei eine COM-Klasse zu laden ist, erfährt das COM-Laufzeitsystem aus der Windows-Registry. Im Zweig HKCR\CLSID findet sich für jede COM-Klasse ein nach ihrer CLSID benannter Schlüssel mit den erforderlichen Angaben. Für die in Abschnitt 23.2.2.2 vorzustellende COM-Klasse handelt es sich um den Schlüssel

```
[HKEY_CLASSES_ROOT\CLSID\{3F6C504D-6663-447D-AF9A-B3073DF61CC1}]
```

Als Standardwert des ProgID-Teilschlüssels findet sich hier für Menschen lesbare Programm-Identifikation. 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.:

```
Windows Registry Editor Version 5.00
```

```
[HKEY_CLASSES_ROOT\EuroKonverter.Betrag]
@="EuroKonverter.Betrag"
```

```
[HKEY_CLASSES_ROOT\EuroKonverter.Betrag\Clsid]
@="{3F6C504D-6663-447D-AF9A-B3073DF61CC1}"
```

Außerdem ist beim COM-Einsatz ein Interface- und ein Typelib-Key beteiligt (siehe unten).

Immerhin können die für COM-Komponenten erforderlichen, recht umfangreichen Registry-Eintragen vom Windows-Hilfsprogramm **regsvr32.exe** automatisch erledigt werden, z.B.:

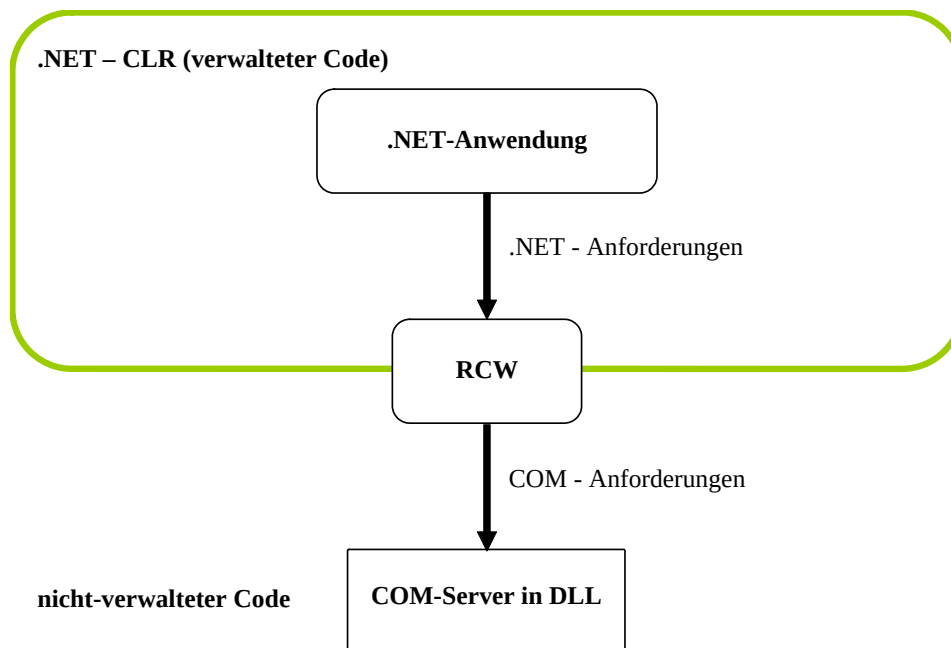
```
regsvr32 EuroKonverter.dll
```

Die fehlerträchtige Abhängigkeit der COM-Technik von zahlreichen Registry-Keys stellt jedoch einen zentralen Kritikpunkt dar, den das .NET-Framework überwunden hat.

23.2.2 COM-Server in .NET - Anwendungen nutzen

23.2.2.1 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 und den COM-typischen Verwaltungsaufwand erledigt.

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 bietet das .NET-Framework Werkzeuge zur automatischen Erstellung solcher Proxy-Assemblies. Im SDK ist das Hilfsprogramm

...\\Microsoft Visual Studio 8\\SDK\\v2.0\\Bin\\TlbImp.exe

enthalten, welches per Kommandozeilenparameter einige Kontrolle über das resultierende Assembly ermöglicht (z.B. Name, Signatur, Namensraum). Beim Einsatz einer .NET - Entwicklungsumgebung (wie Visual C# 2005 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 23.2.2.3).

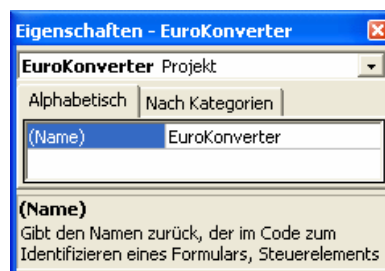
23.2.2.2 COM-Server mit Visual Basic erstellen

Zur Demonstration der .NET-COM - Kooperation erstellen wir nun mit Visual Basic 6.0 einen einfachen COM-Server, registrieren diesen beim Betriebssystem, erzeugen einen RCW und nutzen die COM-Komponente in einem .NET - Programm.

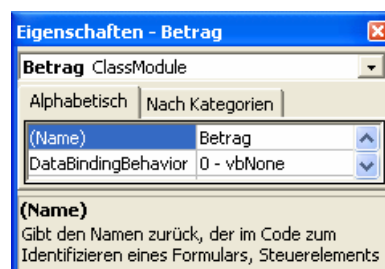
Nach dem Start von Visual Basic 6.0 wählen wir Projekttyp **ActiveX-DLL**, so dass ein prozessinterner COM-Server entsteht:



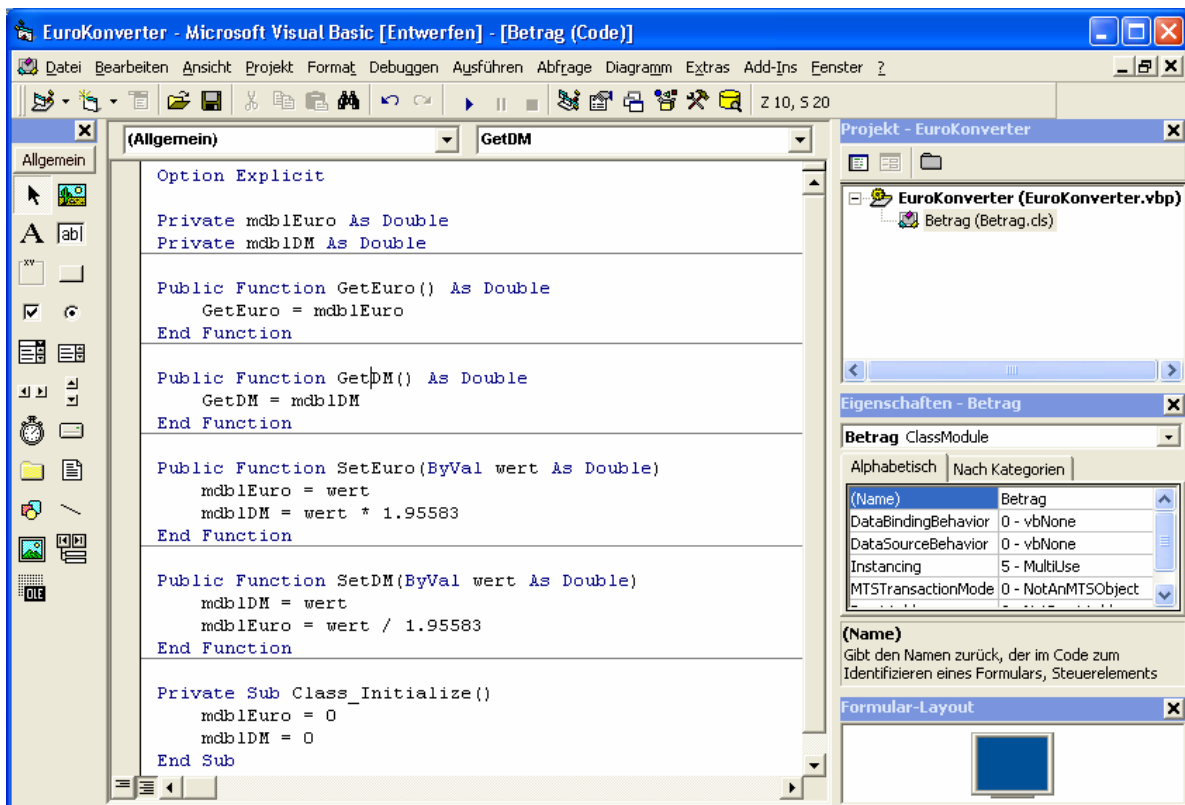
Geben Sie dem Projekt via Projekt-Explorer den Namen EuroKonverter (**Eigenschaften** aus dem Kontextmenü wählen):



Vereinbaren Sie per Eigenschaftsfenster für die zu erstellende Klasse den Namen Betrag:



Im folgenden Quellcode wird ein COM-Objekt realisiert, das Methoden zum Setzen und Abfragen seines Betrages in Euro oder DM anbietet:



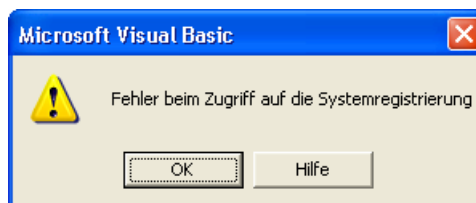
Sichern Sie das Projekt mit

Datei > Projekt speichern unter

und veranlassen Sie die Erstellung des COM-Servers mit

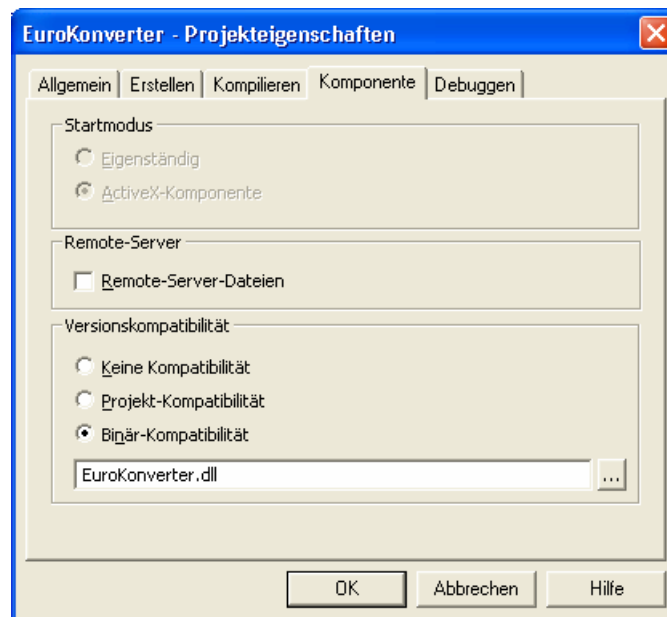
Datei > EuroKonverter.dll erstellen

Weil Visual Basic beim Erstellen automatisch den COM-Server in der Registry eintragen möchte, kommt es ohne Administrator-Rechte zu einer Fehlermeldung, wobei die DLL aber trotzdem entsteht:



Vermutlich werden Sie den COM-Server vor seiner endgültigen Registrierung ohnehin an einen anderen Ort kopieren.

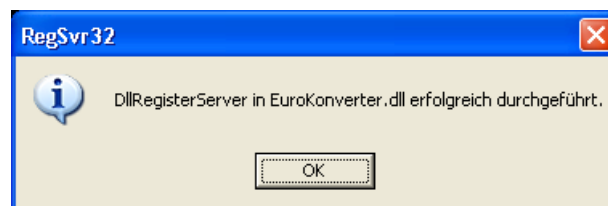
Wählen Sie nun auf der Eigenschaften-Registerkarte **Komponente** die binäre Versionskompatibilität mit der eben erstellten Datei **EuroKonverter.dll**, damit Visual Basic nicht bei jedem Übersetzen neue GUIDs erstellt:



Zum Registrieren eines COM-Servers dient das Windows-Hilfsprogramm **regsvr32.exe**, das natürlich nur mit Administrator-Rechten erfolgreich ausgeführt werden kann, z.B.:

regsvr32 EuroKonverter.dll

Sie sollten folgende Erfolgsmeldung erhalten:



Im folgenden Registry-Key wird die Klasse mit der ProgID EuroKonverter.Betrag deklariert:

Windows Registry Editor Version 5.00

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{089F2AC7-B1CB-4359-B35D-CC128FD01D69}]
```

```
@="EuroKonverter.Betrag"
```

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{089F2AC7-B1CB-4359-B35D-CC128FD01D69}\Implemented Categories]
```

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{089F2AC7-B1CB-4359-B35D-CC128FD01D69}\Implemented Categories\{40FC6ED5-2438-11CF-A3DB-080036F12502}]
```

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{089F2AC7-B1CB-4359-B35D-CC128FD01D69}\InprocServer32]
```

```
@="E:\C#\BspUeb\Interop\COM\COMServer\EuroKonverter.dll"
"ThreadingModel"="Apartment"
```

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{089F2AC7-B1CB-4359-B35D-CC128FD01D69}\ProgID]
```

```
@="EuroKonverter.Betrag"
```

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{089F2AC7-B1CB-4359-B35D-CC128FD01D69}\Programmable]
```

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{089F2AC7-B1CB-4359-B35D-CC128FD01D69}\TypeLib]
```

```
@="{B993F56A-6208-4596-AF86-D367032A79A6}"
```

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{089F2AC7-B1CB-4359-B35D-CC128FD01D69}\VERSION]
```

```
@="1.0"
```

Über diesen Registry-Key lässt sich per ProgID die CLSID ermitteln:

```
Windows Registry Editor Version 5.00
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\EuroKonverter.Betrag]
@="EuroKonverter.Betrag"
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\EuroKonverter.Betrag\Clsid]
@="{089F2AC7-B1CB-4359-B35D-CC128FD01D69}"
```

Wie der folgende TypeLib-Key zeigt, steckt die von Visual Basic erzeugte COM-Typbibliothek in der DLL:

```
Windows Registry Editor Version 5.00
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\TypeLib\{B993F56A-6208-4596-AF86-D367032A79A6}]
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\TypeLib\{B993F56A-6208-4596-AF86-D367032A79A6}\1.0]
@="EuroKonverter"
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\TypeLib\{B993F56A-6208-4596-AF86-D367032A79A6}\1.0\0]
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\TypeLib\{B993F56A-6208-4596-AF86-D367032A79A6}\1.0\0\win32]
@="E:\C#\BspUeb\Interop\COM\COMServer\EuroKonverter.dll"
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\TypeLib\{B993F56A-6208-4596-AF86-D367032A79A6}\1.0\FLAGS]
@="0"
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\TypeLib\{B993F56A-6208-4596-AF86-D367032A79A6}\1.0\HELPDIR]
@="E:\C#\BspUeb\Interop\COM\COMServer"
```

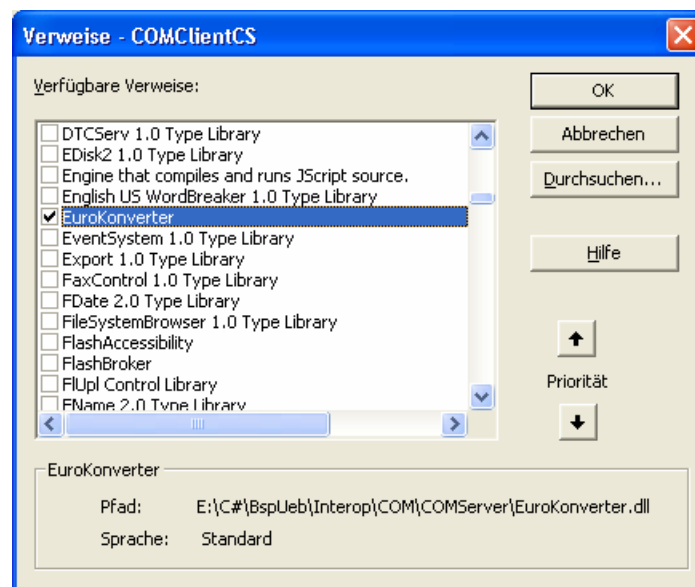
Es wird auch noch ein Interface-Key benötigt:

```
Windows Registry Editor Version 5.00
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\Interface\{70E883A7-8148-4393-9570-A9FB7B2C17F1}]
@="Betrag"
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\Interface\{70E883A7-8148-4393-9570-A9FB7B2C17F1}\ProxyStubClsid]
@="{00020424-0000-0000-C000-000000000046}"
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\Interface\{70E883A7-8148-4393-9570-A9FB7B2C17F1}\ProxyStubClsid32]
@="{00020424-0000-0000-C000-000000000046}"
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\Interface\{70E883A7-8148-4393-9570-A9FB7B2C17F1}\TypeLib]
@="{B993F56A-6208-4596-AF86-D367032A79A6}"
"Version"="1.0"
```

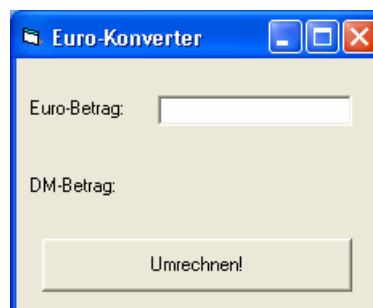
Bevor wir den Einsatz des COM-Servers im .NET-Framework behandeln, soll noch seine Funktionstüchtigkeit im klassischen Kontext bewiesen werden. Dazu legen wir ein neues VB6-Projekt vom Typ **Standard-Exe** an. Um hier ein Objekt unserer COM-Klasse erzeugen zu können, muss der registrierte COM-Server über

Projekt > Verweise

in die Referenzenliste des Projekts aufgenommen werden, z.B.:



Zum Befehlsschalter im folgenden Formular



verfassen wir diese **Click**-Ereignisroutine:

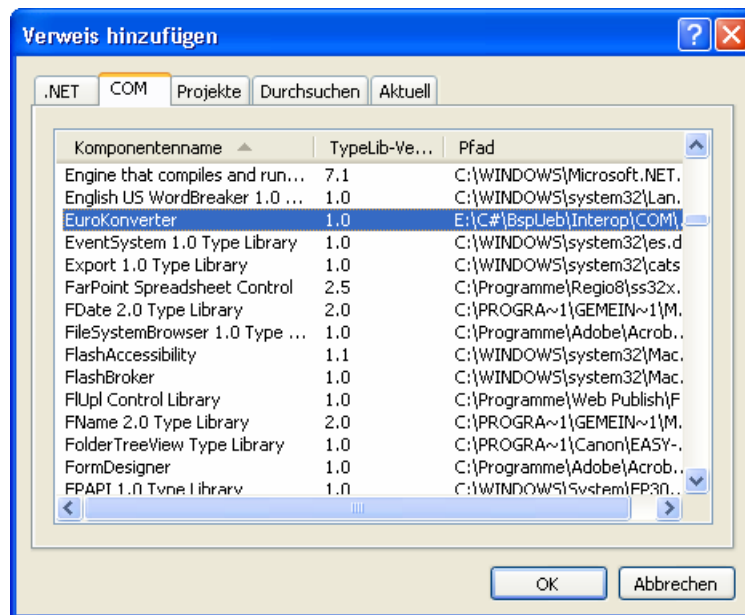
```
Private Sub Umrechnen_Click()  
    Dim b As New EuroKonverter.Betrag  
    b.SetEuro euro.Text  
    LabelErgebnis.Caption = b.GetDM  
End Sub
```

Das Objekt der Klasse `EuroKonverter.Betrag` erfährt den Ausgangsbetrag durch den Aufruf seiner Methode `SetEuro()` und liefert als Rückgabewert seiner Methode `GetDM()` das Konvertierungsergebnis. Mehr Quellcode ist für den bescheidenen, aber immerhin funktionstüchtigen COM-Klienten nicht erforderlich:



23.2.2.3 COM-Server in C# verwenden

Wir legen ein neues C# - Projekt an und fügen per Projektmappen-Explorer einen Verweis auf den COM-Server EuroKonverter hinzu:



Daraufhin entsteht im Projekt-Unterordner `...\obj\Debug` das Proxy-Assembly **Interop.EuroKonverter.dll** mit der Proxy-Klasse `BetragClass` im Namensraum `EuroKonverter`.

Wer mehr Kontrolle benötigt (z.B. beim Klassennamen oder Namensraum), der kann das in Abschnitt 23.2.2.1 beschriebene 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 EuroKonverter;

class COMClientCS : Form {
    Label lbErgebnis;
    TextBox tbEuro;
    BetragClass betrag = new BetragClass();

    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;
    cbUmrechnen.Click += new EventHandler(ButtonOnClick);
}

void ButtonOnClick(object sender, EventArgs e) {
    if (tbEuro.Text.Length > 0) {
        double d;
        try {
            d = double.Parse(tbEuro.Text);
        } catch {
            lbErgebnis.Text = "Falsche Eingabe!";
            return;
        }
        betrag.SetEuro(d);
        lbErgebnis.Text = betrag.GetDM().ToString();
    }
}

static void Main() {
    Application.Run(new COMClientCS());
}
}

```

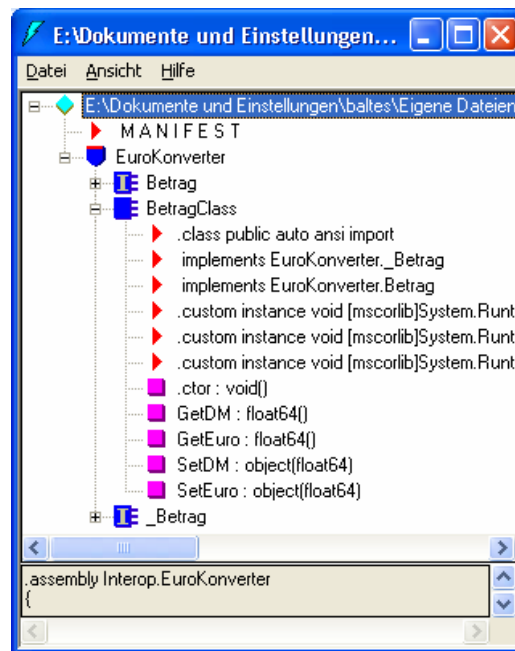
Der automatisch angelegte Namensraum zum RCW-Assembly wird importiert:

```
using EuroKonverter;
```

Für die Währungskonvertierung wird ein Objekt der Proxy-Klasse `BetragClass` erzeugt, womit indirekt ein COM-Objekt entsteht:

```
BetragClass betrag = new BetragClass();
```

Bei Bedarf kann man ein Proxy-Assembly z.B. mit dem SDK-Hilfsprogramm **ILDasm.exe** inspizieren:

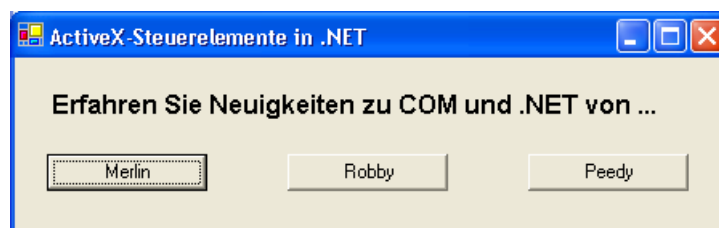


Unser COM-Klient nutzt in seiner `ButtonOnClick()`-Methode die Methoden `SetEuro()` und `GetDM()` des COM-Objekts.

Ruft der Klient eine Server-Methode auf, dann muss deren Adresse im Speicher ermittelt werden. Diese so genannte **Bindung** (engl.: *binding*) kann (wie beim eben beschriebenen Verfahren) zur Übersetzungszeit erfolgen (= frühe Bindung) oder (mit erheblich größerem Zeitaufwand) zur Laufzeit (= späte Bindung). Weitere Informationen zur späten Bindung finden Sie z.B. in Liberty (2002, S. 597ff) oder Troelsen (2002, S. 622ff).

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.

Ein C# - Projekt mit der folgenden Bedienoberfläche

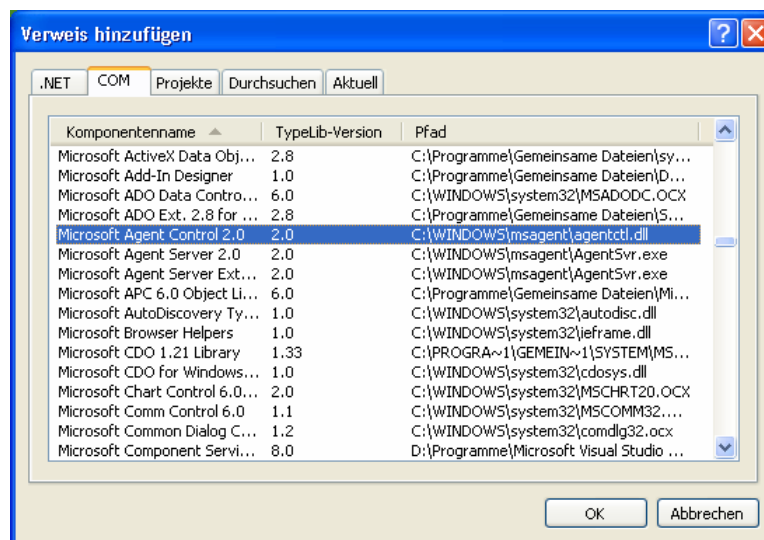


erhält in der Visual C# 2005 Express Edition per Projektmappen-Explorer einen Verweis auf das ab Windows 2000 in der Regel installierte **Microsoft Agent Control 2.0**¹:

¹ Nötigenfalls kann die Software von einem Microsoft-Server herunter geladen werden:

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

Dort stehen über den bei einer Windows-Installation meist ausschließlich vorhandenen *Merlin* hinaus noch weitere Agenten zur Verfügung, und auch eine Sprachausgabe mit gut verständlicher deutscher Aussprache ist vorhanden.



Nach dem Abschicken dieser Dialogbox entsteht im Projekt-Unterordner `...\obj\Debug` das Proxy-Assembly **Interop.AgentObjects.dll** mit dem RCW.

Die Formularklasse erhält Referenzinstanzvariablen vom Schnittstellentyp **IAgentCtlCharacterEx**, den der RCW aus der COM-Typbibliothek übernommen hat:

```
IAgentCtlCharacterEx merlin, robby, peedy;
```

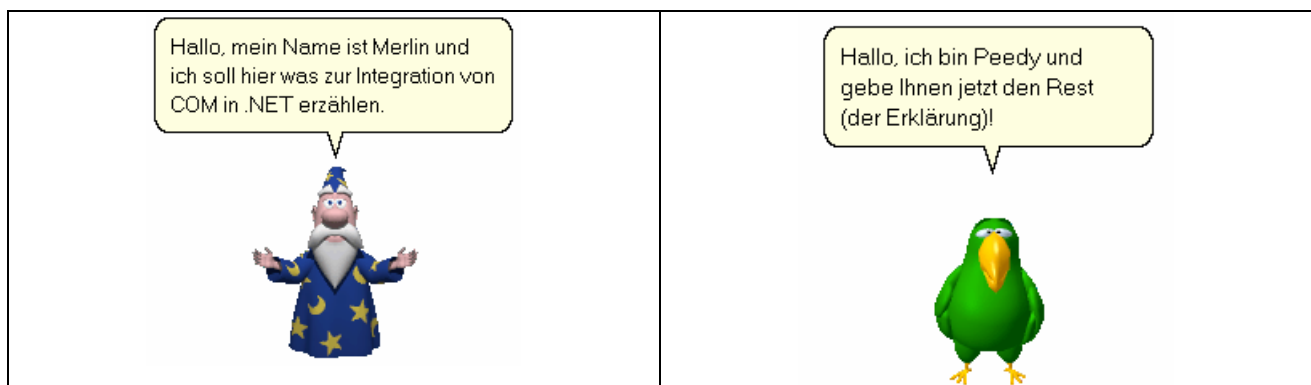
Ein Objekt der ebenfalls importierten Klasse **AgentClass** übernimmt die Aufgabe, drei Akteure (Merlin, Robby und Peedy) auf ihren Auftritt vorzubereiten:

```
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:



24 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.

24.1 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.

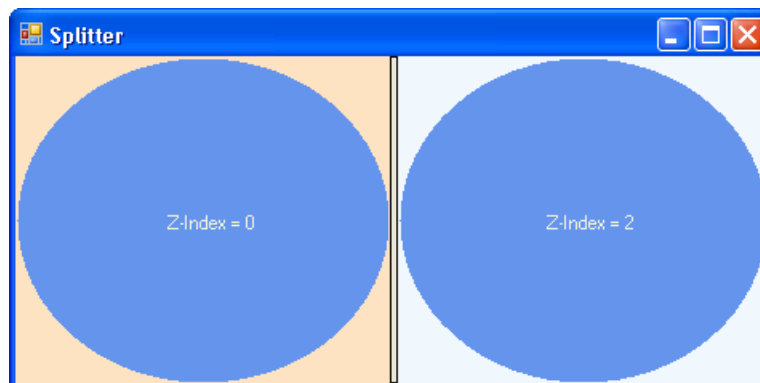
Neben dem älteren Steuerelement **Splitter** ist seit .NET 2.0 die bequemere und in der Regel zu bevorzugende Klasse **SplitContainer** verfügbar. Anschließend wird trotzdem auch die ältere Klasse vorgestellt, weil sie in vielen vorhandenen C# - Quellen anzutreffen ist und nur .NET 1.0 benötigt, also eine optimale Kompatibilität bietet

24.1.1 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** andockt, dem das ältere **Panel**-Element nach der in Abschnitt 8.4.4.3 beschriebenen Regel für **Dock**-Konkurrenten den Vorrang überlässt.
- Zuletzt wird ein weiteres **Panel**-Element rechts andockt, 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;

public class SplitterDemo : Form {

    public SplitterDemo() {
        Text = "Splitter";
        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 = 5;
        split.BorderStyle = BorderStyle.FixedSingle;

        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.Beige,
                    pan.Width / 2 - 24, pan.Height / 2 - 5);
    }

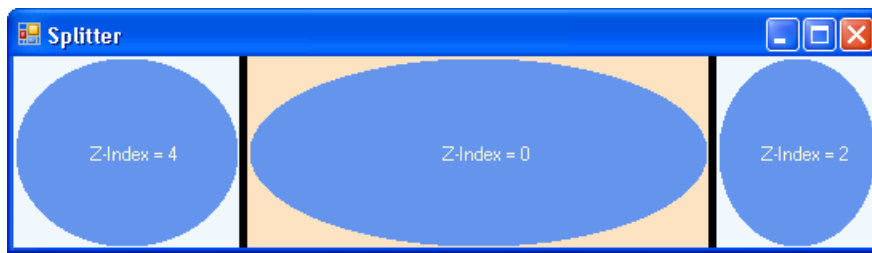
    void PanelOnResize(object sender, EventArgs e) {
        ((Panel) sender).Refresh();
    }

    public static void Main() {
        Application.Run(new SplitterDemo());
    }
}

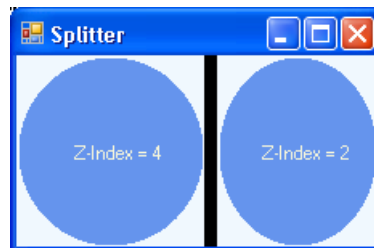
```

Für eine Aktualisierung der Anzeige bei einer Größenänderung der **Panel**-Objekte wird in diesem Programm über einen **Resize**-Handler gesorgt. Bei unseren aus **Form** abgeleiteten Fensterklassen haben wir denselben Effekt über den Wert **true** für die geschützte (!) Eigenschaft **ResizeRedraw** erreicht.

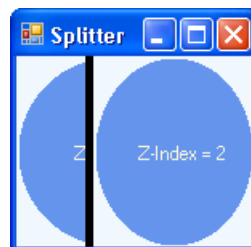
Es dürfte Ihnen keine Mühe machen, auch am linken Rand ein Paar aus **Splitter**- und **Panel**-Objekt einzubauen, so dass drei größenvariable Flächen entstehen, von denen die *mittlere* (mit **DockStyle.Fill**) auf horizontale Änderungen der Fenstergröße reagiert:



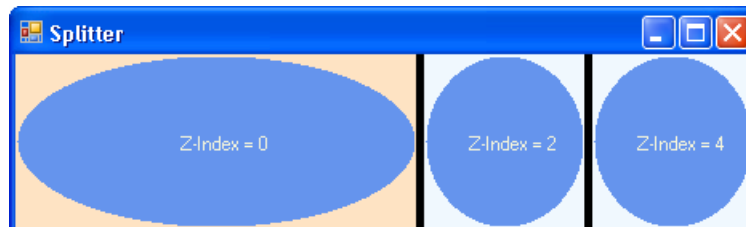
Bei zunehmender Platznot verschwindet irgendwann das **Panel** mit **DockStyle.Fill**:



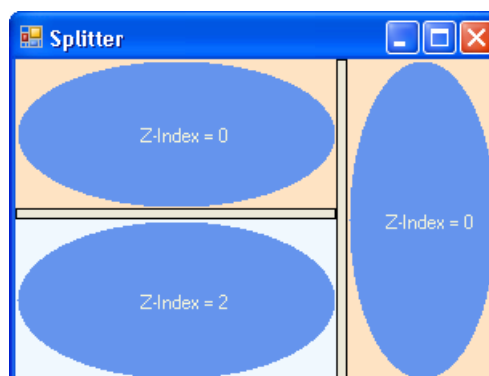
Bei weiter schrumpfender horizontaler Fensterausdehnung wird das **Panel** mit dem Z-Index 4 vom **Panel** mit dem Z-Index 2 überdeckt:



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 Containerrand, an den er angedockt 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**.

24.1.2 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;

public class SplitContainerDemo : Form {
    SplitContainer sc;

    public 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);
    }

    public static void Main() {
        Application.Run(new SplitContainerDemo());
    }
}

```

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 wird ermittelt oder festgelegt, ob **Panel1** bzw. **Panel2** kollabieren (verschwinden) soll.
- **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**.

24.2 Baumansichten

Die Präsentation von hierarchisch strukturierten Daten (z.B. Laufwerke und Verzeichnisse eines Rechners) wird im .NET-Framework durch die Klassen aus dem Namensraum **System.Windows.Forms** 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 dar.

24.2.1 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 **MainMenu**, **MenuItem** und **MenuItemCollection** (vgl. Abschnitt 16) erinnert fühlen, liegen Sie richtig und können den folgenden Quellcode sicher mühelos verstehen:

```
using System;
using System.Windows.Forms;

public class TreeViewEinfach : Form {
    public TreeViewEinfach() {
        Text = "TreeViewEinfach";

        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");
    }

    public static void Main() {
        Application.Run(new TreeViewEinfach());
    }
}
```

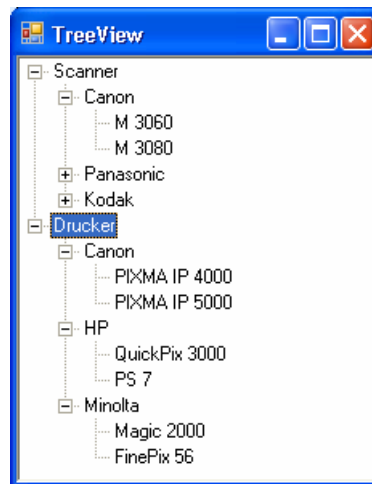
Per Indexzugriff auf die Unterknoten erspart man sich explizite Referenzvariablen, muss aber strikt auf die Reihenfolge der Knoten achten. Z.B. wird in folgender Anweisung

```
tv.Nodes[1].Nodes[2].Nodes.Add("Magic 2000");
```

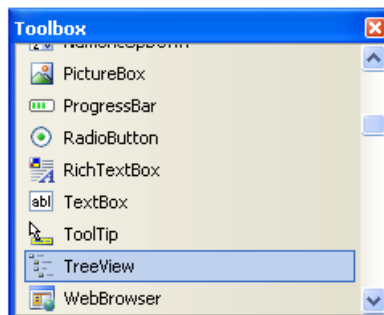
die **Add()**-Methode des 3. Unterknotens (*Minolta*) im 2. Stammknoten (*Drucker*) aufgerufen. Mit expliziten Referenzvariablen vom Typ **TreeNode** kann eventuell die Lesbarkeit des Quellcodes verbessert werden, z.B.:

```
TreeNode minolta = tv.Nodes[1].Nodes[2];
minolta.Nodes.Add("Magic 2000");
minolta.Nodes.Add("FinePix 56");
```

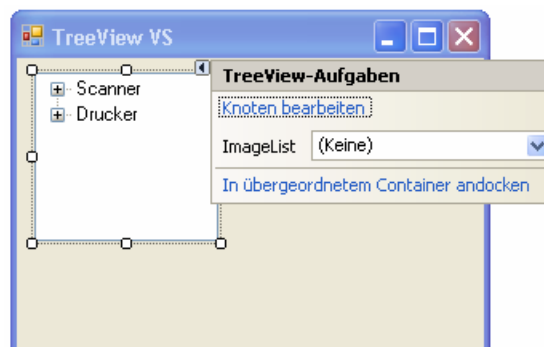
Gemessen am Programmieraufwand erhalten wir eine brauchbare Optik und Funktionalität:



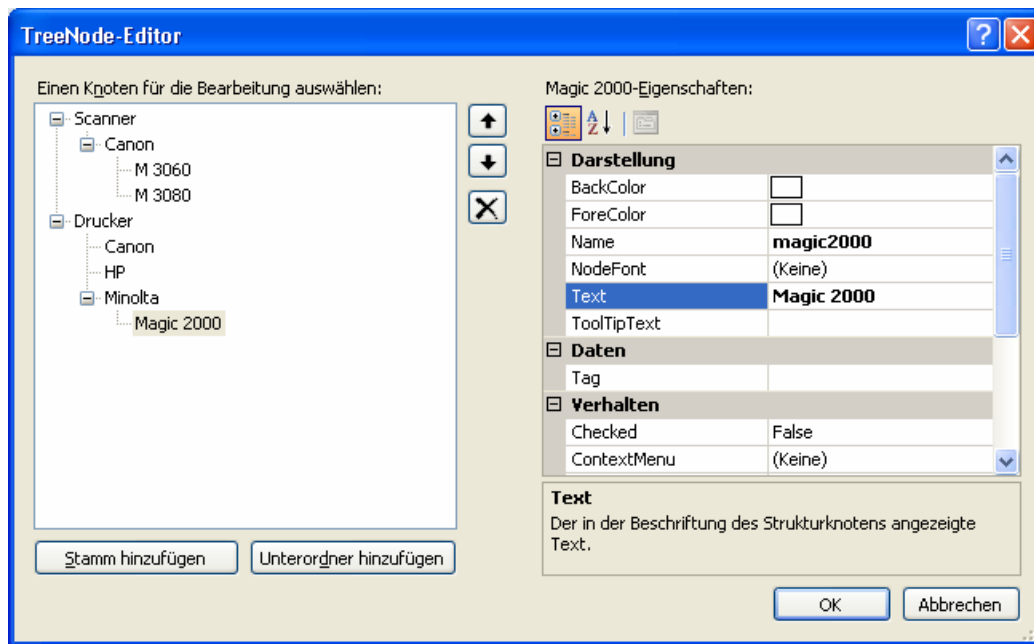
In der Visual C# 2005 Express Edition unterstützt der Formulardesigner nach dem Einfügen eines **TreeView**-Steuerelements



das Erstellen einer Baumansicht über den **TreeNode**-Editor, erreichbar über den Eintrag **Knoten bearbeiten** im **TreeView**-Aufgabenmenü, das nach einem Mausklick auf den kleinen Pfeil am oberen Rand des Steuerelements erscheint:



Das Anlegen, Verschieben und Löschen von Stamm- und Unterknoten gelingt auf intuitivem Weg. Allerdings ist die Liste der **TreeNode**-Eigenschaften unvollständig:



24.2.2 Datenerfassung

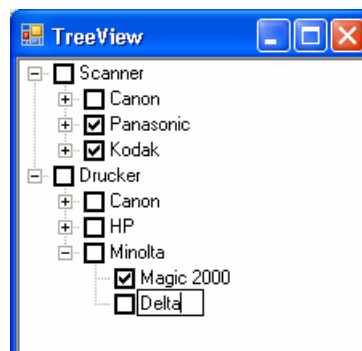
Ein **TreeView**-Objekt eignet sich nicht nur zur Präsentation, sondern auch zur Erfassung von Daten:

- 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
 - **TreeNodeCollection** (z.B. **Add()**) und
 - **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.

24.2.3 Verzeichnisbaum

Wir betrachten nun ein besonders wichtiges 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 (2003, S. 932ff) die aus **TreeView** abgeleitete Klasse **DirectoryView**:

```
using System;
using System.IO;
using System.Windows.Forms;

public class DirectoryView : TreeView {

    public DirectoryView() {
        Nodes.Clear();
        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) {
        tn.Nodes.Clear();
        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:

```
public static void Main() {
    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 über Ein-/Ausgabe per Datenstrom bekannt sein dürfte:

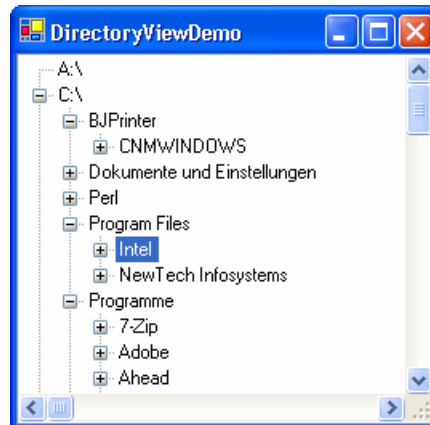
```
String[] drives = Directory.GetLogicalDrives();
```

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 `AddDirectory()` zum Erstellen der Unterknoten aufgerufen. Durch den rekursiven Selbstaufwurf (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.

24.2.4 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. Im der erweiterten Version des `DirectoryInfo`-Konstruktors verschafft uns die statische **DriveInfo**-Methode einen Array mit **DriveInfo**-Objekten, welche die im lokalen System vorhandenen Datenträger repräsentieren. Über den Typ eines Laufwerks informiert die **DriveInfo**-Eigenschaft **DriveType** mit Werten vom gleichnamigen Enumerationstyp:

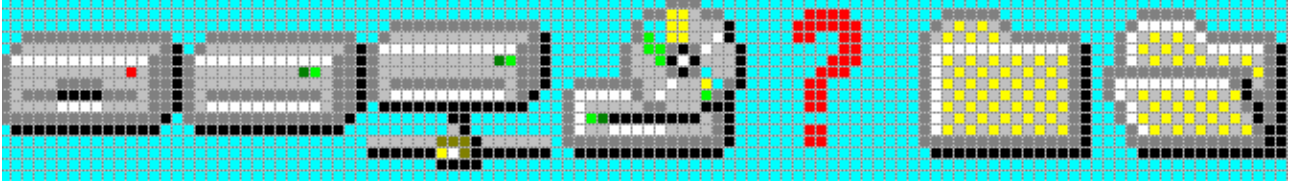
```
public DirectoryView() {
    BeginUpdate();
    Nodes.Clear();
    DriveInfo[] drives = DriveInfo.GetDrives();
    int imageIndex;
    foreach (DriveInfo d in drives) {
        switch (d.DriveType) {
            case DriveType.Removable: imageIndex = 0; break;
            case DriveType.Fixed: imageIndex = 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;
    }
    EndUpdate();
}
```

```

ImageList = new ImageList();
ImageList.Images.AddStrip(Image.FromFile("dirview.bmp"));
ImageList.TransparentColor = Color.Cyan;
}

```

Im folgenden Streifen sind sieben Bitmaps mit jeweils 16×16 Pixeln für die wichtigsten Laufwerkstypen sowie für geschlossene und geöffnete Ordner enthalten:

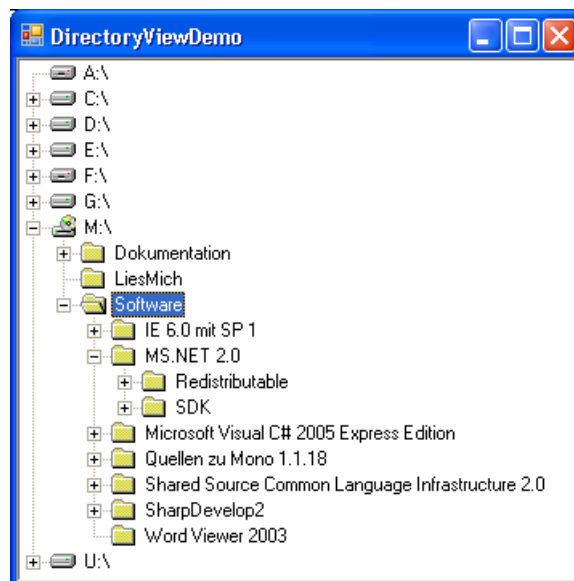


Die im erweiterten Konstruktor unserer `DirectoryView`-Klasse verwendeten Anweisungen bzw. Klassen zur Verwaltung der Bitmaps sind mit den Erfahrungen aus dem Abschnitt 19.2 über Symbolleisten leicht zu verstehen. Das **ImageList**-Objekt wird der **ImageList**-Instanzeigenschaft der (aus **TreeView** abgeleiteten) Klasse `DirectoryView` zugewiesen und so mit der Strukturansicht verknüpft. 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 aus der Methode `AddDirectories()` zum Erstellen eines Ordnerknotens:

```
TreeNode tnDir = new TreeNode(Path.GetFileName(s), 5, 6);
```

Unser `DirectoryView`-Steuerelement besitzt nun eine akzeptable Optik:



Anschließend wird endlich das schon in Abschnitt 24.2.3 erwähnte Performanzproblem beim Erstellen eines **DirectoryInfo**-Objekts durch einen bedarfsgerechten und ereignisgesteuerten Baumaufbau behoben.

24.2.5 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	Standardbehandlungsmethode	Delegatentyp
BeforeSelect	OnBeforeSelect()	TreeViewCancelEventHandler
BeforeExpand	OnBeforeExpand()	TreeViewCancelEventHandler
BeforeCollapse	OnBeforeCollapse()	TreeViewCancelEventHandler
AfterSelect	OnAfterSelect()	TreeViewEventHandler
AfterExpand	OnAfterExpand()	TreeViewEventHandler
AfterCollapse	OnAfterCollapse()	TreeViewEventHandler

Bei einer eigenen, aus **TreeView** abgeleiteten Klasse empfiehlt sich das Überschreiben der Standardbehandlungsmethode. Ansonsten ist eine Methode zu definieren, deren Signatur dem zugehörigen Delegatentyp entspricht.

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();
}
```

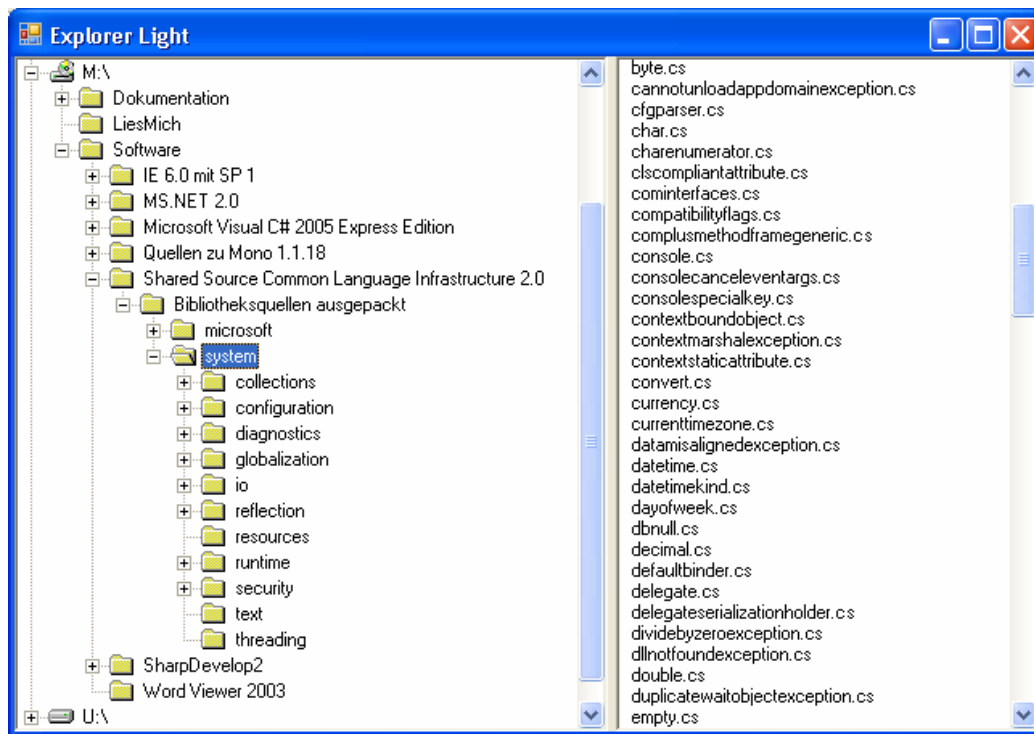
wird für einen zu expandierenden Knoten *K* zunächst die Liste der Unterknoten aktualisiert. Somit kann der Benutzer durch Reduzieren und anschließendes Erweitern eines Knotens eine Aktualisierung erzwingen. Dann werden die Unterknoten der nächsten Schachtelungsstufe ermittelt, damit das Steuerelement die Unterknoten von *K* ggf. mit einem Plussymbol ausstattet. Mit dem rekursiven Selbstaufzuruf am Ende der Methode **AddDirectories()** verschwindet unser Performanzproblem.

Den vollständigen Quellcode der Klasse **DirectoryView** finden Sie im Ordner

...\BspUeb\WinForms\TreeView\DirectoryView\Fertig

24.2.6 Explorer Light

Abschließend soll als Anwendungsbeispiel für unser neues Steuerelement **DirectoryView** eine Light-Variante des Windows-Explorers erstellt werden:



Bei der Raumaufteilung kommt die in Abschnitt 24.1.2 behandelte Klasse **SplitContainer** zum Einsatz:

```
using System;
using System.Drawing;
using System.Windows.Forms;
using System.IO;

public class ExplorerLight : Form {
    SplitContainer sc;
    DirectoryView dv;
    Panel fileList;

    public 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.Invalidate();
}

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;
}

public static void Main() {
    Application.Run(new ExplorerLight());
}

```

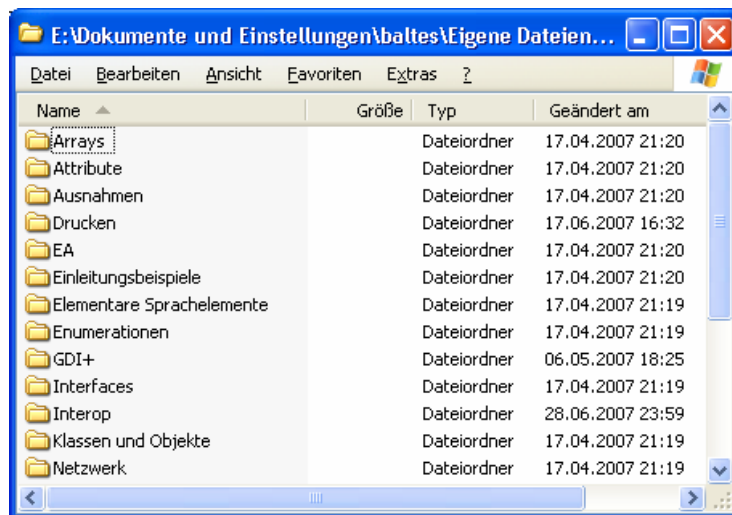
In den linken **SplitContainer**-Bereich setzen wir **DirectoryView**-Objekt mit Knoten für alle Datenträger und Ordner des lokalen Rechners. Rechts platzieren wir ein oben angedocktes **Panel**-Steuerelement mit einem **Paint**-Handler, welcher die Namen der Dateien im markierten Knoten auflistet und die **Panel**-Höhe neu berechnet. In einem **AfterSelect**-Handler für das **DirectoryView**-Objekt veranlassen wir die Aktualisierung der Dateienliste beim Wechsel der Knotenmarkierung.

24.3 Listen und Tabellen

Zum Präsentieren einer Tabelle bzw. Liste mit mehreren Spalten (Variablen) und einer beliebigen Anzahl von Einträgen (Fällen) sind die Steuerelemente **ListBox** und **ComboBox** wenig geeignet, weil die spaltenbündige Anzeige kaum gelingt. Als Alternative bietet das .NET – 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 die Itembezeichnung. In weiteren Spalten erscheinen die zu einem Item erzeugten 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.
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 für eine Auflistung im Stil des Windows-Explorers konstruiert wurde. Hier wird die **Details**-Ansicht genutzt:



24.3.1 Konstruktion

Um mit den Klassen **ListView**, **ListViewItem** und **ListViewSubItem** vertraut zu werden, erstellen wir zunächst ein einfaches Beispielprogramm:

```
using System;
using System.Windows.Forms;

public class ListViewDemo : Form {
    public 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());
        }
    }

    public static void Main() {
        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 Klasse **ListViewSubItem** verwaltet und die bei Kollektionsklassen üblichen Methoden bietet.

Listenunterelemente (**ListViewSubItem**-Objekte) besitzen ebenfalls eine **Text**-Eigenschaft. Bei mehrspaltigen Listen erscheinen die **Text**-Werte der Subitems sukzessive ab der zweiten Spalte.

Das Beispielprogramm zeigt für diverse Brotaufstriche die Namen linksbündig, die Preise und Kalorien 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

24.3.2 Datenerfassung

Wie ein **TreeView**-Objekt (vgl. Abschnitt 24.2.2) 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 bei **TreeView**-Steuerelement) auch eine Mehrfachauswahl möglich ist.
- Zur besseren Unterstützung der Mehrfachauswahl lassen sich per **CheckBoxes**-Eigenschaft Kontrollkästchen aktivieren, z.B.:

```
listView.CheckBoxes = true;
```

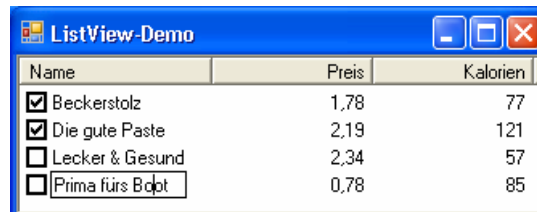
Bei aktivierten Kontrollkästchen signalisiert das **ListView**-Ereignis **ItemCheck** einen Zustandswechsel.

- Man kann per **LabelEdit**-Eigenschaft das Verändern der Itembezeichnungen durch den Benutzer erlauben, z.B.:

```
listView.LabelEdit = true;
```

Den Text der Subitems kann man nicht auf diese Weise frei geben. In der SDK-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.

24.3.3 Ereignisse

Analog zu den Verhältnissen bei der Baumansicht (vgl. Abschnitt 24.2.5) tritt bei der Listenansicht nur die Klasse **ListView** als Ereignisemittend in Erscheinung. Neben dem bereits erwähnten Ereignis **ItemCheck** sind auch die folgenden Ereignisse oft von Interesse:

Ereignis	Erläuterung
ItemActivate	Durch Aktivieren eines Items (mit einem einfachen oder doppelten Mausklick, je nach Wert der Activation -Eigenschaft) kann z.B. der Benutzer des Windows-Explorers signalisieren, dass er ein Dokument öffnen oder ein Programm starten möchte.
SelectedIndexChanged	Die Auswahl hat sich geändert.
ColumnClick	Der Benutzer hat auf einen Spaltenkopf geklickt.

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 markiert sind.

24.3.4 Verzeichniseinträge auflisten

In der folgenden **ListView**-Ableitung, die teilweise auf einem Vorschlag von Petzold (2003, 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.Drawing;
using System.Diagnostics;
using System.Threading;

public class DirEntryList : ListView {
    String strDir;
```

```

public DirEntryList(String dir) {
    Dir = dir;
    View = View.Details;
    Sorting = SortOrder.Ascending;
    Columns.Add("Name", 120, HorizontalAlignment.Left);
    Columns.Add("Größe", 120, HorizontalAlignment.Right);
    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;
    strDir = dir;
    Items.Clear();
    DirectoryInfo directoryInfo = new DirectoryInfo(dir);
    FileInfo[] afi;
    try {
        afi = directoryInfo.GetFiles();
    } catch {
        return;
    }
    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);
    }
    DirectoryInfo[] adi;
    try {
        adi = directoryInfo.GetDirectories();
    } catch {
        return;
    }
    foreach (DirectoryInfo di in adi) {
        ListViewItem lvi = new ListViewItem(di.Name);
        lvi.SubItems.Add("");
        lvi.SubItems.Add(di.LastWriteTime.ToString());
        Items.Add(lvi);
    }
}

override protected void OnItemActivate(EventArgs ea) {
    base.OnItemActivate(ea);
    foreach (ListViewItem lvi in SelectedItems) {
        if (lvi.SubItems[1].Text == "")
            ShowEntries(Path.Combine(strDir, lvi.Text));
        else
            ThreadPool.QueueUserWorkItem(new WaitCallback(StartProcess),
                Path.Combine(strDir, lvi.Text));
    }
}

protected void StartProcess(object path) {
    try {
        Process.Start((String)path);
    } catch {
        return;
    }
}
}

```

Zu Demonstrations- und Testzwecken ergänzen wir noch eine **Main()**-Methode:

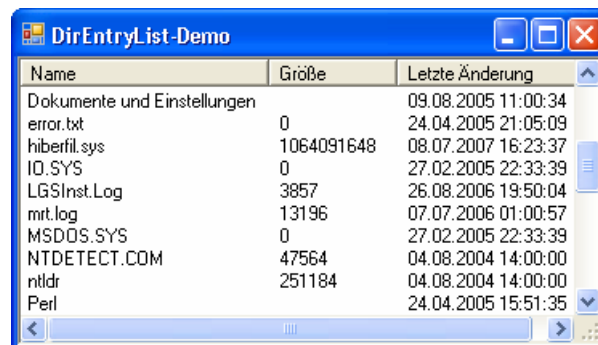
```
public static void Main() {
    Form frm = new Form();
    frm.Text = "DirEntryList-Demo";
    DirEntryList fv = new DirEntryList("C:\\");
    fv.Parent = frm;
    fv.Dock = DockStyle.Fill;
    fv.ShowEntries();
    Application.Run(frm);
}
```

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 11.3 gearbeitet.

Die überschreibende Behandlungsmethode für das Ereignis **ItemActivate** sorgt dafür, dass alle markierten Dateien mit Hilfe der statischen Methode **Start()** aus der Klasse **Process** (Namensraum **System.Diagnostics**) gestartet bzw. geöffnet werden. 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 zu öffnende Verzeichniseintrag ein Ordner, dann werden seine Verzeichniseinträge vom aufgelistet. So gelingt der Wechsel zu einem Unterordner, während der umgekehrte Weg mangels Eintrag für das übergeordnete Verzeichnis nicht zur Verfügung. Es macht aber sicher keine großen Probleme, den Rückweg z.B. wie im Windows-Explorer über einen Symbolschalter zu realisieren.

Optisch ist das **DirEntryList**-Steuerelement noch verbesserungsbedürftig:



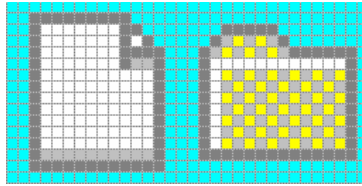
Weil Microsoft in der englischen Dokumentation (Stand: .NET 3.0) keine Ordnung für die **GetFiles()** - Produktion garantieren möchte, hat die **Sorting**-Eigenschaft den Wert **SortOrder.Ascending** erhalten, was allerdings zum *gemeinsamen* Sortieren von Dateien und Ordnern führt. In Abschnitt 24.3.6 sorgen wir dafür, dass zunächst die Unterordner und danach die Dateien (jeweils sortiert) erscheinen.

24.3.5 Bilder für die Listenitems

Wie die Knoten im **TreeView**-Steuerelement (vgl. Abschnitt 24.2.4) 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:

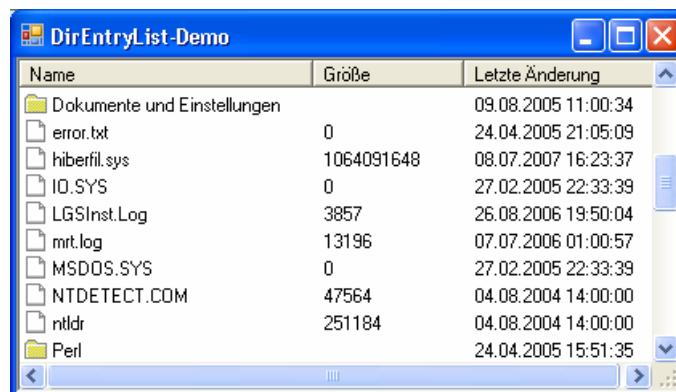


Mit der Transparenz-Deklaration für die Hintergrundfarbe des Streifens sorgen wir für eine ansprechende Optik.

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:



24.3.6 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**, 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.
- Außerdem wird in **DirEntryList** ein Memberobjekt der Klasse **DirEntryComparer** erzeugt und der 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);
```

- Bisher enthalten die **ListViewItem**-Objekte Subitems für den Datei- bzw. Ordnernamen, die Größe und den letzten Schreibzugriff, wobei alle drei Eigenschaften den Typ **String** besitzen. Beim Sortieren sind aber Zeichenfolgen mit formatierten Datums- und Zeitangaben (z.B. 20.08.2007 15:52) 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.:

```
return del.SortOrder > 0 ?
    comparer.Compare((lx.Tag as FileInfo).Length, (ly.Tag as FileInfo).Length):
    comparer.Compare((ly.Tag as FileInfo).Length, (lx.Tag as FileInfo).Length);
```

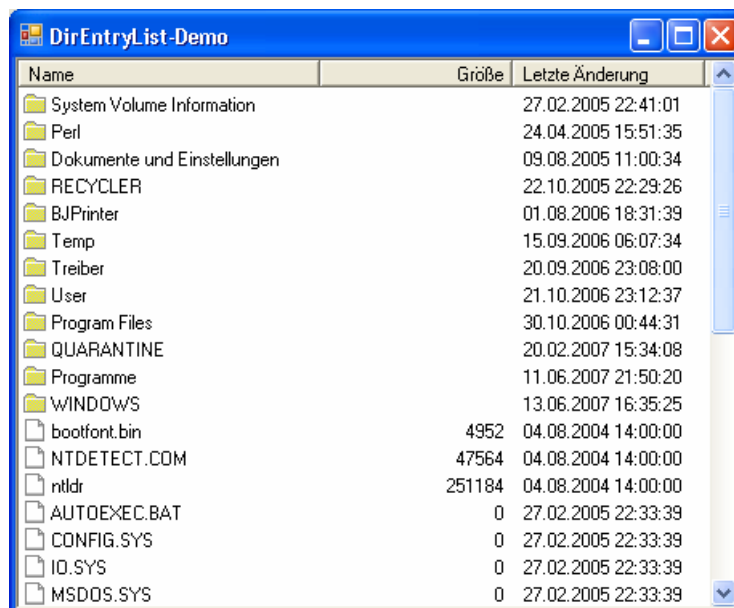
Hier wird per Tag-Eigenschaft auf die angehefteten **FileInfo**-Objekte zugegriffen. Den sortierrichtungsabhängigen Größenvergleich überlassen wir einem Objekt der Klasse

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 berücksichtigt.

Nun bietet unser Steuerelement den gewohnten Sortierkomfort:



Den vollständigen Quellcode der Klasse **DirectoryView** finden Sie im Ordner

...BspUeb\WinForms\ListView\DirListEntry\Fertig

24.3.7 Explorer Midrange

Mit dem neu entwickelten Steuerelement **DirectoryView** (siehe Abschnitt 24.2.3ff) und **DirEntryList** (siehe Abschnitt 24.3.4ff) können wir eine verbesserte Variante unseres Explorer-Nachbaus erstellen:

```

using System;
using System.Drawing;
using System.Windows.Forms;

public class ExplorerMidrange : Form {
    SplitContainer sc;
    DirectoryView dv;
    DirEntryList del;

    public 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);

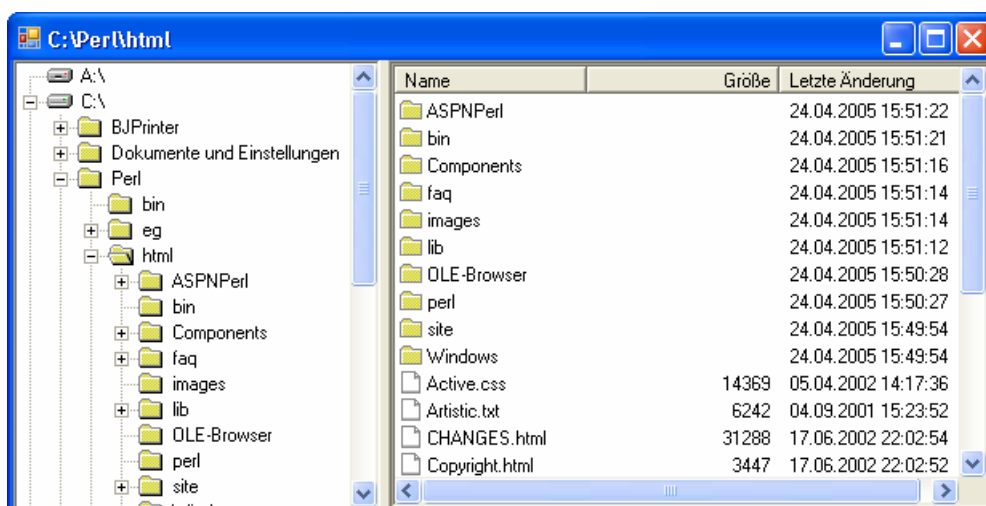
        del = new DirEntryList(dv.SelectedNode.FullPath);
        del.Parent = sc.Panel2;
        del.Dock = DockStyle.Fill;
    }

    void DirectoryViewOnAfterSelect(object sender, TreeViewEventArgs tvea) {
        String s = tvea.Node.FullPath.Replace("\\\\", "\\");
        del.ShowEntries(s);
        Text = s;
    }

    public static void Main() {
        Application.Run(new ExplorerMidrange());
    }
}

```

Wir haben mit begrenztem Aufwand ein brauchbares und leicht ausbaufähiges Programm erstellt:



Zur Projektorganisation ist noch anzumerken, dass die beiden Klassen `DirectoryView` und `DirEntryList` gemeinsam in einem Bibliotheks-Assembly mit integrierten Ressourcen verfügbar gemacht werden sollten.

24.4 Übungsaufgaben zu Abschnitt 24

1) Präsentieren Sie dem Benutzer einen Baumeditor, der das Einfügen, Entfernen und Benennen von Knoten erlaubt.

25 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 Kühnel (2006, Kap. 26) 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)**, das für die Erstellung, Pflege und Abfrage von Datenbanken verantwortlich ist. Bekannte Beispiele für diese Softwaregattung sind: Microsoft SQL Server (Express Edition), MySQL, Oracle, DB2 von IBM usw. Wir werden in diesem Kurs mit der kostenlos verfügbaren Microsoft SQL Server 2005 Express Edition arbeiten, die zusammen mit der Visual C# 2005 Express Edition ausgeliefert und per Voreinstellung mit installiert wird. 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.

Das .NET – Framework enthält eine Bibliothek namens **ADO.NET** (*Active 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.

25.1 Relationale Datenbanken

Heute arbeitet praktisch jedes DBMS mit der **relationalen Datenbankstruktur**, die sich gegenüber älteren Bauformen (z.B. hierarchische Datenbank, Netzwerk-Datenbank) weitgehend durchgesetzt hat.

25.1.1 Tabellen

Bei einer relationalen Datenbank sind die Daten in **Tabellen**¹ angeordnet, wobei die Spalten auch **als Felder** oder **Variablen** und die Zeilen auch als **Datensätze** (engl: *records*) oder **Fälle** bezeichnet werden:

¹ Oft wird mit dem Begriff *Relation* eine Beziehung zwischen zwei Tabellen bezeichnet (siehe unten). Manche Autoren verwenden ihn jedoch synonym zum Begriff *Tabelle*, was durchaus mit der mathematischen Definition einer Relation als Teilmenge eines kartesischen Produktes aus mehreren Mengen harmonisiert: Jede von den m Variablen einer Tabelle steuert die Menge ihrer potentiellen Ausprägungen bei. Ein Element des kartesischen Produktes dieser m

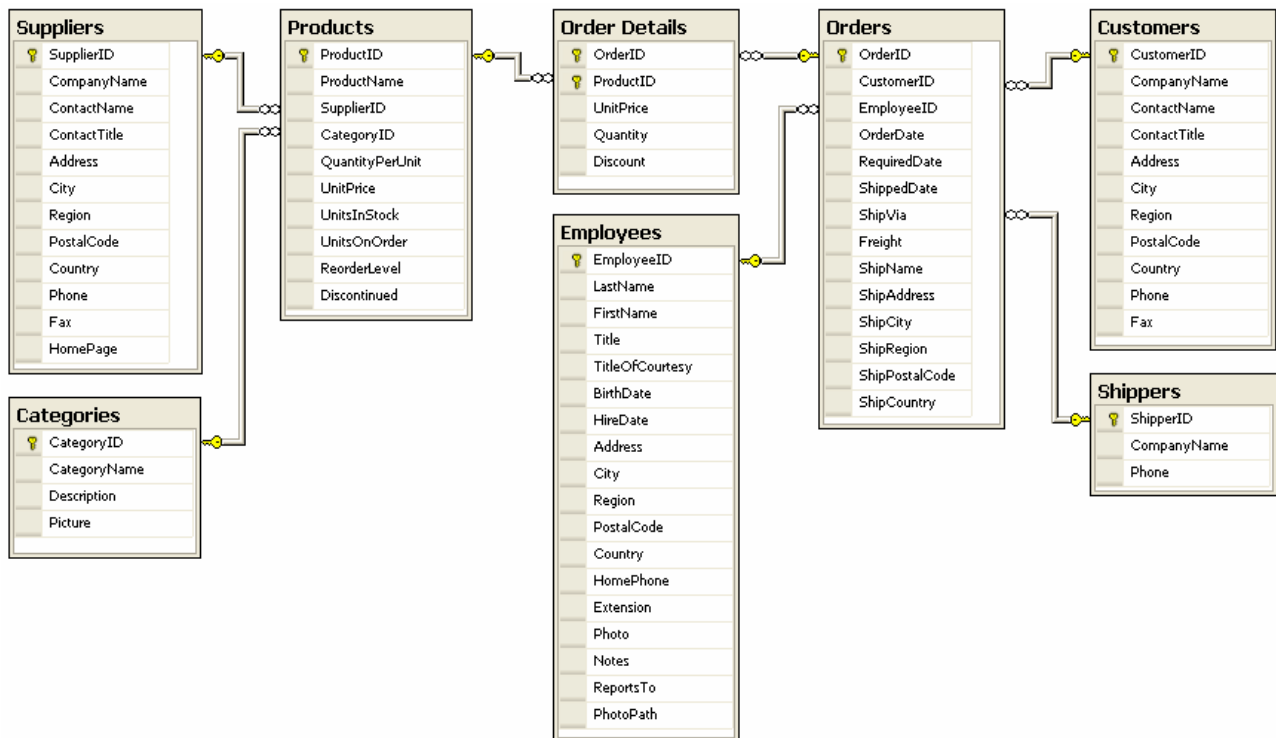
- Jede Zeile der Tabelle enthält die zu einem Fall verfügbaren Informationen, wobei jede Tabelle Fälle eines bestimmten Typs enthält (z.B. Kunden, Artikel, Lieferanten, Reklamationen). Welche Tabellen benötigt werden, hängt von Anwendungsbereich ab.
- Jede Spalte (jedes Feld) enthält für alle Fälle die Werte zu einem Merkmal. Alle Werte in einer Spalte besitzen denselben **Datentyp** (z.B. Zahl, Datum, Zeichenfolge).
- Oft dient eine bestimmte Spalte als **Primärschlüssel** und enthält eindeutige Werte (kein Wert kommt mehrmals vor). 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 DBMS vereinbaren, die Werte einer Primärschlüsselspalte ausgehend von einem Startwert automatisch zu vergeben.

Wir betrachten in Abschnitt 25 die von Microsoft angebotene Beispieldatenbank zu einer erfundenen Firma namens **Northwind** (siehe Abschnitt 25.3.2.2 zum Bezug und zur Installation der Datenbank). Wie das Visual Studio als SQL Server - Frontend genutzt werden kann, erfahren Sie in Abschnitt 25.5.3. In der **Northwind**-Datenbank befindet sich u.a. eine Tabelle mit dem Personal der Firma:

	EmployeeID	LastName	FirstNa...	Title	TitleOfCourtesy	BirthDate	HireDate	Address	City	Region	PostalCode	Country
▶	1	Davolio	Nancy	Sales Repres...	Ms.	08.12.1948 ...	01.05.1992...	507 - 20th Av...	Seattle	WA	98122	USA
	2	Fuller	Andrew	Vice Presiden...	Dr.	19.02.1952 ...	14.08.1992...	908 W. Capit...	Tacoma	WA	98401	USA
	3	Leverling	Janet	Sales Repres...	Ms.	30.08.1963 ...	01.04.1992...	722 Moss Bay...	Kirkland	WA	98033	USA
	4	Peacock	Margaret	Sales Repres...	Mrs.	19.09.1937 ...	03.05.1993...	4110 Old Red...	Redmond	WA	98052	USA
	5	Buchanan	Steven	Sales Manager	Mr.	04.03.1955 ...	17.10.1993...	14 Garrett Hill	London	NULL	SW1 8JR	UK
	6	Suyama	Michael	Sales Repres...	Mr.	02.07.1963 ...	17.10.1993...	Coventry Hou...	London	NULL	EC2 7JR	UK
	7	King	Robert	Sales Repres...	Mr.	29.05.1960 ...	02.01.1994...	Edgeham Holl...	London	NULL	RG1 9SP	UK

1 von 9 Die Zelle ist schreibgeschützt.

Das folgende Datenbankdiagramm zeigt die wichtigsten **Northwind**-Tabellen mit den zugehörigen Feldern und den Primärschlüsseln (einfach oder zusammengesetzt):

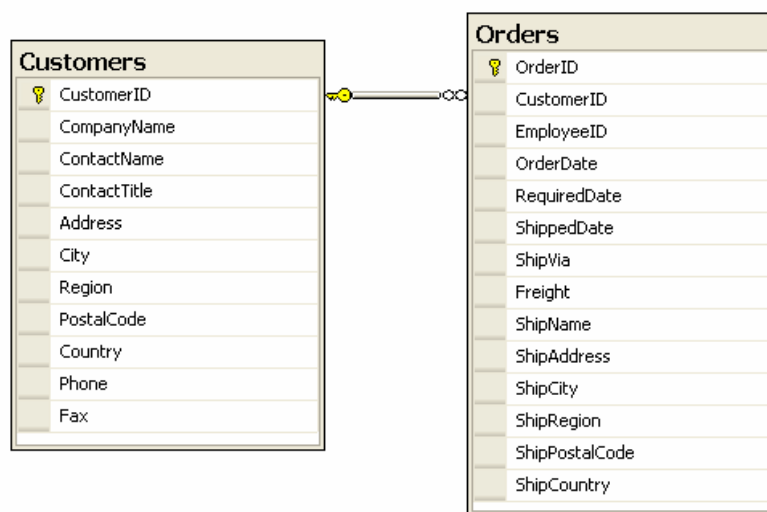


Mengen ist ein m -Tupel mit m speziellen Variablenausprägungen, also eine Tabellenzeile. Eine Tabelle mit n Zeilen kann also in der Tat als Teilmenge des kartesischen Produkts (sprich: als Relation) aufgefasst werden.

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.

25.1.2 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 (als 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.

25.2 SQL

Ein DBMS interagiert mit anderen Softwaresystemen über die **Structured Query Language** (SQL). Microsofts SQL-Server 2005 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.

25.2.1 Überblick

Wie der Name *SQL* nahe legt, ist die *Abfrage* von Informationen klarer Einsatzschwerpunkt, wenngleich der gesamte Sprachumfang die beiden folgenden Teilmengen umfasst:

- **DDL (Data Definition Language)**

Mit den DDL-Befehlen (z.B. CREATE TABLE, CREATE INDEX) kann man die Struktur einer Datenbank erstellen oder verändern.

- **DML (Data Manipulation Language)**

Es werden zwei Arten von „Manipulationen“ unterschieden:

- **Auswahlabfragen**

Der außerordentlich wichtige Befehl SELECT ermöglicht das Abrufen, Auswählen, Suchen und Auswerten von Datensätzen. Als Abfrageergebnis erhält man eine Menge von Datenzeilen (engl. *rowset*), die sich bei Verwendung von Beziehungen strukturell von den Zeilen der Datenbanktabellen unterscheiden können. Obwohl der SELECT-Befehl meist der DML zugerechnet wird, findet bei einer Abfrage offenbar keine Manipulation statt.

- **Aktionsabfragen**

Die folgenden Aktionsbefehle haben tatsächlich eine Veränderung von Datenbankinhalten zur Folge:

- **INSERT**
Hängt einen Datensatz am Ende einer Tabelle an
- **DELETE**
Löscht einen Datensatz
- **UPDATE**
Ändert die Werte von Fällen

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 25.4.7).

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.

25.2.2 Spalten abrufen

Will man im 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
```

25.2.3 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')
```

25.2.4 Daten aus mehreren Tabellen zusammenführen

Sobald die FROM-Klausel *mehrere* Tabellen enthält, muss den Namen der abgerufenen Spalten ein Tabellename 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 (vom DBMS überwachten) 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
```

25.2.5 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
```

25.2.6 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>

25.2.7 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
```

25.3 Microsoft SQL Server 2005 Express Edition

25.3.1 Eigenschaften

Microsofts SQL Server 2005 Express Edition ist ein voll funktionstüchtiges, zeitlich unbefristet verwendbares Datenbankmanagementsystem (DBMS). Im Vergleich zu den kostenpflichtigen Varianten von Microsofts SQL Server 2005 bestehen folgende Einschränkungen:

- maximale Datenbankgröße 4 GB
- maximal 1 GB Hauptspeicher nutzbar
- nur 1 CPU nutzbar
- Mitwirkung bei der Datenreplikation nur als Klient

Von der guten Integration mit der Visual C# 2005 Express Edition (siehe Abschnitt 25.5.3) haben wir u.a. folgende Vorteile:

- bequeme Datenbankprogrammierung
- Ablage von kompletten .NET - Objekten in einer Datenbank

Einer Verwendung der SQL Server 2005 Express Edition in Ihren eigenen Programmen stehen keine lizenzrechtlichen Probleme im Wege, weil Microsoft eine Verbreitung zusammen mit Ihren Anwendungen nach einer Registrierung (<http://www.microsoft.com/sql/editions/express/default.msp>) erlaubt.

Weil in diesem Manuskript die meisten Aussagen über Microsofts SQL-Server für *alle* Versionen der Generation 2005 gelten, sparen wir uns in der Regel die umständliche Bezeichnung *Microsoft SQL Server 2005 Express Edition* und reden kurz über den *SQL-Server*.

Wenn Sie für diesen Kurs die Visual C# 2005 Express Edition verwenden, haben Sie vermutlich die mitgelieferte Microsoft SQL Server 2005 Express Edition zusammen mit der Entwicklungsumgebung installiert. Nötigenfalls können Sie die Installation des SQL-Servers über das Setup-Programm der Entwicklungsumgebung nachholen.

Anschließend befindet sich der SQL-Server im Ordner

C:\Programme\Microsoft SQL Server

und auf Ihrem Rechner läuft der automatisch gestartete Windows-Dienst

SQL Server (SQLEXPRESS)

Folglich ist der SQL-Server auch ohne expliziten Start durch einen Benutzer dienstbereit, zunächst aber nur für lokal angemeldete Benutzer, weil per Voreinstellung keine Netzzugriffe erlaubt sind (siehe Abschnitt 25.3.3.2).

Weitere Informationen zur Microsoft SQL Server 2005 Express Edition finden Sie z.B. bei Misner (2007).

25.3.2 Ergänzende Installationen

25.3.2.1 Microsoft SQL Server Management Studio Express

Mit dem Server Management Studio Express lassen sich Datenbanken bequem anlegen und inspizieren. Es wird *nicht* mit dem SQL Server Express ausgeliefert, kann aber kostenlos über folgenden Link bezogen werden:

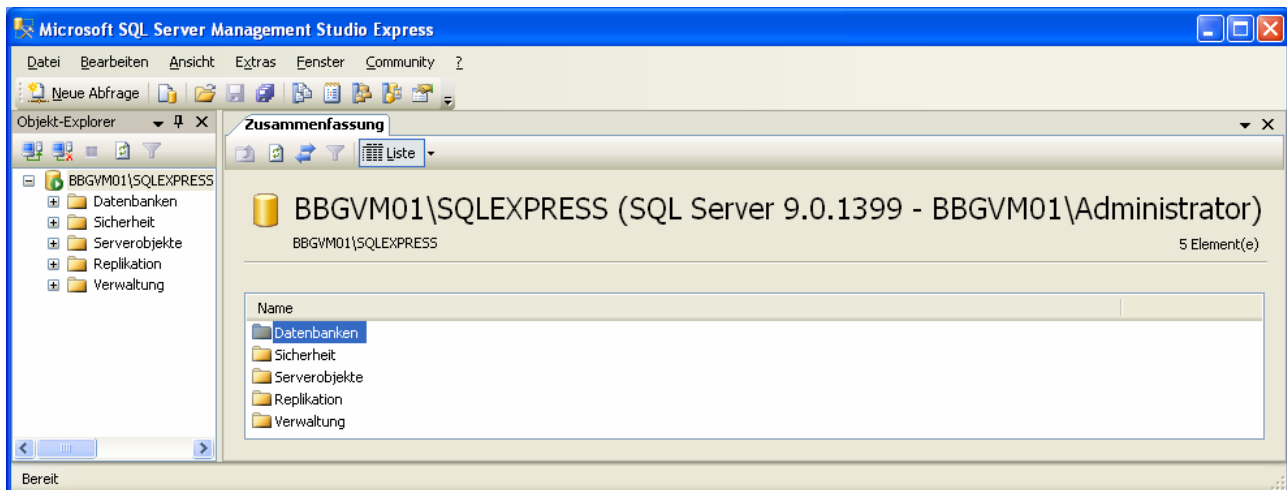
<http://www.microsoft.com/downloads/details.aspx?familyid=C243A5AE-4BD1-4E3D-94B8-5A0F62BF7796&displaylang=de>

Starten Sie die Installation mit Administratorrechten per Doppelklick auf die herunter geladene Datei **SQLServer2005_SSMSEE.msi**, und folgen Sie den Anweisungen des Installationsprogramms.

Wenn Sie das Server Management Studio starten, wird die Verbindungsaufnahme mit dem lokal installierten SQL Server angeboten, z.B.:



Nach einem Klick auf **Verbinden** haben Sie Zugang zu wichtigen Verwaltungsfunktionen des SQL-Servers:



Wir werden das Server Management Studio Express z.B. in Abschnitt 25.3.3.1 zur Konfiguration von Datenbankzugriffsrechten benutzen.

25.3.2.2 Beispieldatenbank Northwind

Über den Link

<http://msdn2.microsoft.com/de-de/library/ms143221.aspx>

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

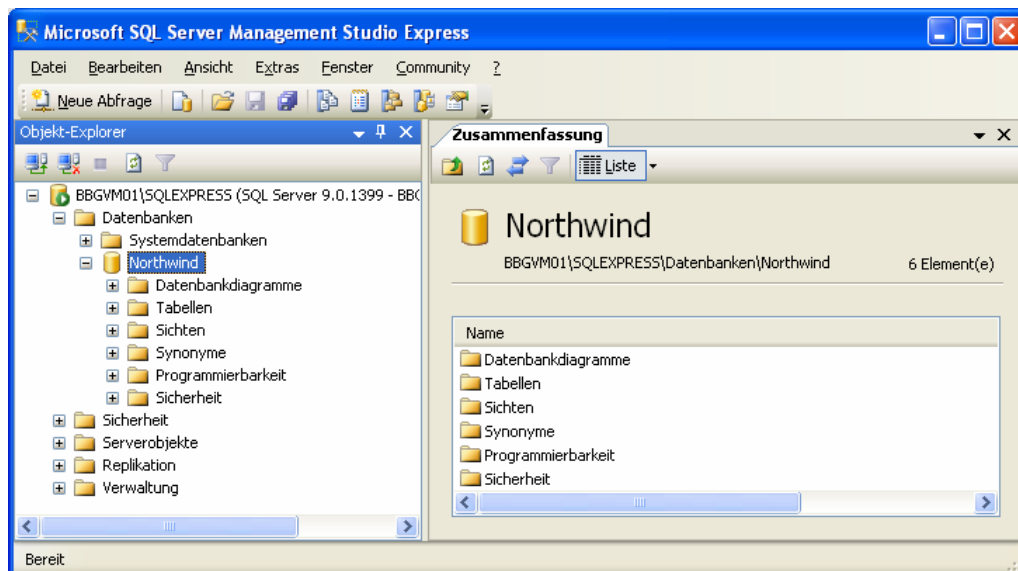
C:\SQL Server 2000 Sample Databases

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 (siehe Abschnitt 25.3.2.1), um die Beispieldatenbank **Northwind** in die Verwaltung des SQL Servers zu integrieren:

- 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 zweimal mit **OK**.

Anschließend finden Sie die Beispieldatenbank im Server Management Studio, wobei für eine Ansicht mit allen Details noch ein Aktualisieren der Ansicht (Menüitem **Ansicht > Aktualisieren** oder Taste **F5**) erforderlich ist:



Der *logische Datenbankname* (hier: **Northwind**) wird per Voreinstellung aus der **mdf**-Datei übernommen und kann bei Bedarf im Management Studio geändert werden.

25.3.3 Konfiguration

Mit der SQL Server 2005 Express Edition werden zwei Konfigurationsprogramme installiert:

- **Microsoft SQL Server Configuration Manager**
Mit diesem Programm verwaltet man Dienste und Netzwerkprotokolle zum SQL-Server.
- **Microsoft SQL Server Oberflächenkonfiguration**
Dieses Programm erlaubt die Konfiguration von Diensten, Verbindungen und speziellen Programmeigenschaften.

Die Startlinks zu den Programmen finden Sie im Unterordner **Konfigurationstools** der Programmgruppe zum SQL Server.

Für viele Konfigurationsarbeiten ist das in Abschnitt 25.3.2.1 vorgestellte SQL Server Management Studio Express am besten geeignet.

25.3.3.1 Rechteverwaltung

Bei der Verwaltung von Rechten geht es um ...

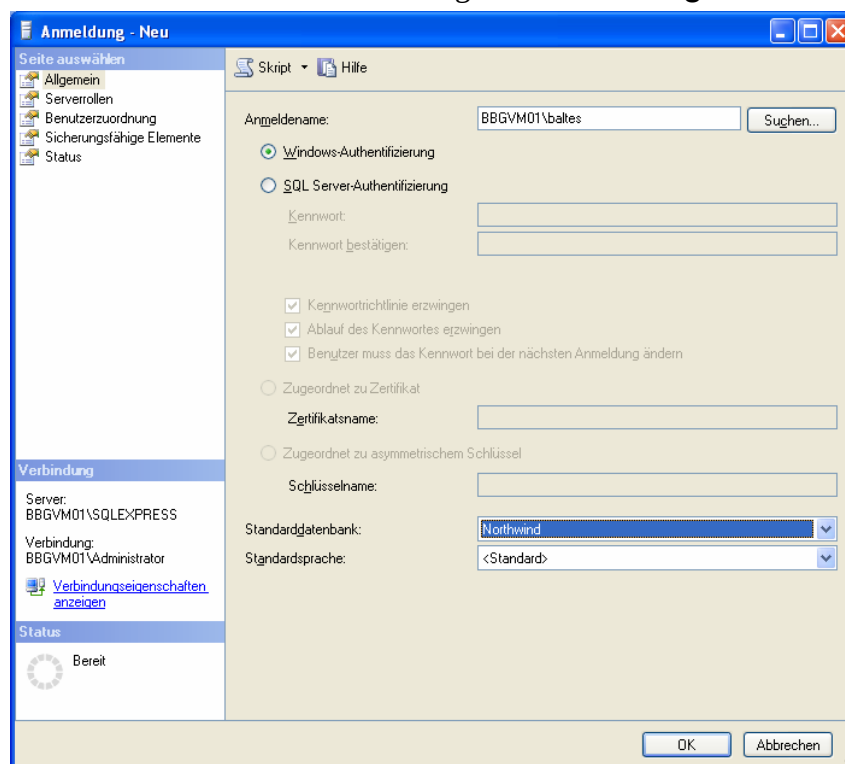
- **Authentifizierung**
Welche Benutzer, Gruppen oder Dienste können sich beim SQL Server anmelden? Eine solche Anmeldung ist z.B. dann erforderlich, wenn ein Programm im Auftrag und mit den Rechten des Benutzers mit einem SQL-Server kooperieren möchte.¹ Als Methode zur Benutzer - Authentifizierung empfiehlt 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?

¹ Die SQL Server 2005 *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 25.4.2.1). Bei dieser Einsatzart ist keine vorbereitende Sicherheitskonfiguration erforderlich.

Authentifizierung

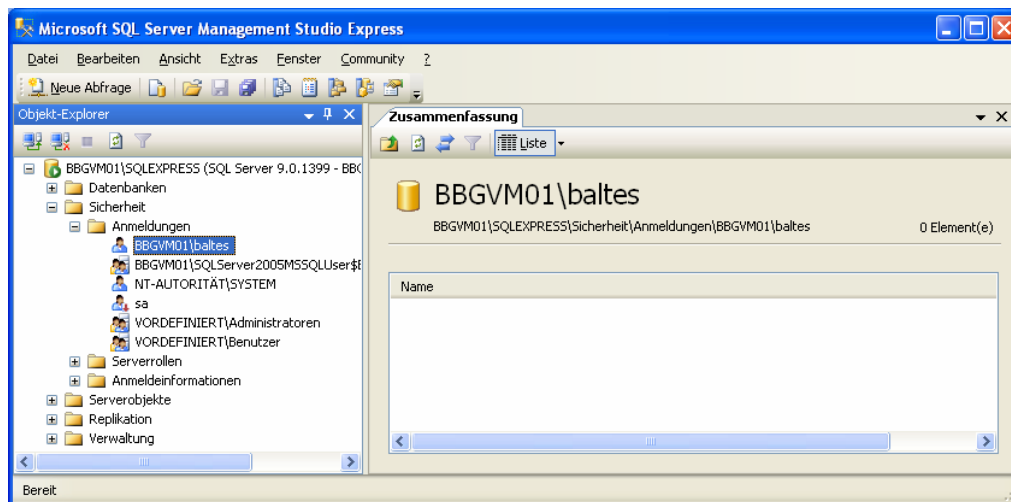
Gehen Sie folgendermaßen vor, um einem Windows-Konto den Zugriff auf den SQL Server zu erlauben:

- Starten Sie das **SQL Server Management Studio Express** mit Administratorrechten, und stellen Sie die Verbindung mit dem 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.
- Nach Aufnahme eines Benutzers sieht die Dialogbox **Anmeldung - Neu** so aus:



- Wählen Sie für den Benutzer eine **Standarddatenbank**.

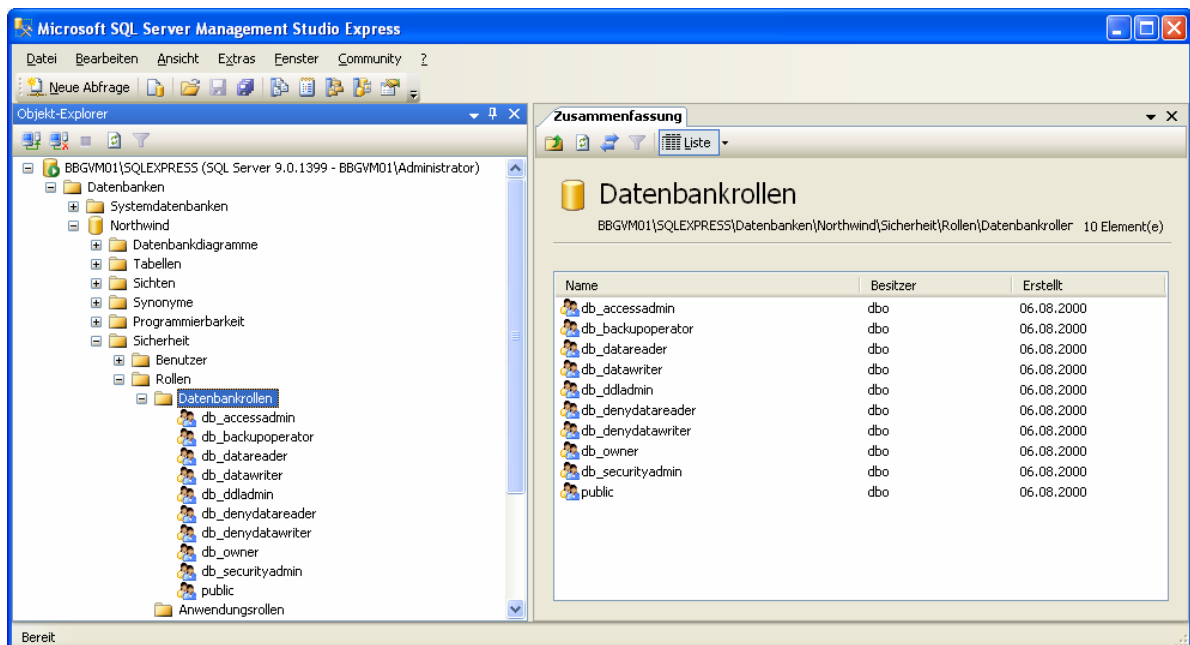
Der neue Benutzer wird im Ordner **Sicherheit > Anmeldungen** des Objekt-Explorers aufgelistet:



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 Rechners, auf dem der SQL-Server läuft.
- **Datenbankrollen**
Diese werden für jede Datenbank separat definiert und folglich vom Objekt-Explorer des Management Studios im Sicherheitsordner einer Datenbank aufgelistet, z.B.:

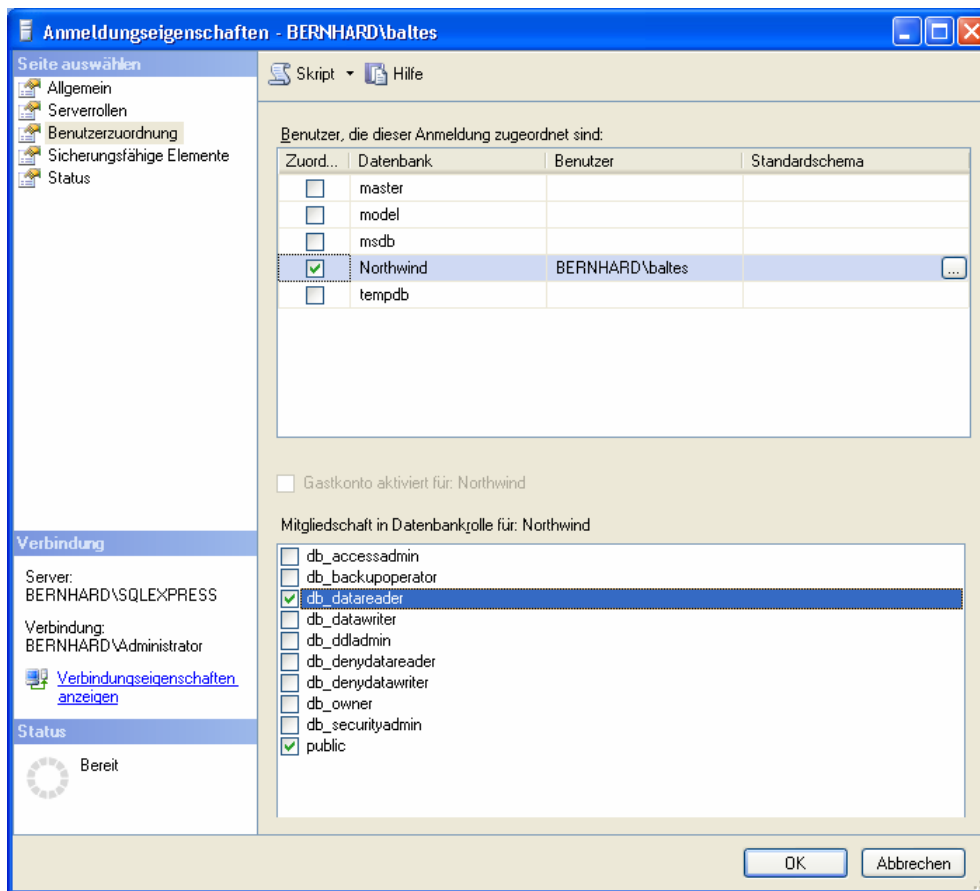


Die vordefinierten Datenbankrollen können durch Eigenkreationen ergänzt werden.

- **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.

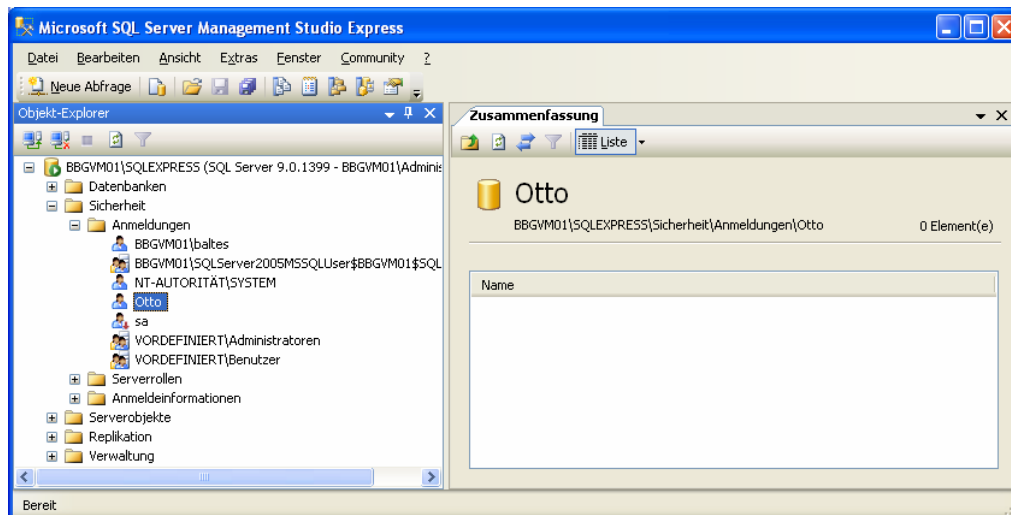
Nach der Standardinstallation unterstützt der SQL Server 2005 Express nur die Windows - Authentifizierung. Im Management Studio Express lässt sich jedoch die SQL - Authentifizierung nachträglich zuschalten:

- 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, 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.

25.3.3.2 Netzwerkzugriff via TCP/IP erlauben

Per Voreinstellung steht der SQL-Server nur lokal angemeldeten Benutzern zur Verfügung, was für die Zwecke von Abschnitt 25 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 die **SQL Server-Oberflächenkonfiguration** über ihren Link im Unterordner **Konfigurationstools** der Programmgruppe zum SQL Server.
- Klicken Sie auf **Oberflächenkonfiguration für Dienste und Verbindungen**.
- Wählen Sie in der Baumansicht am linken Rand des Dialogs den Knoten **SQLEXPRESS > Datenbankmodul > Remoteverbindungen**.
- Wählen Sie die Option **Lokale Verbindungen und Remoteverbindungen** mit der Unteroption **Nur TCP/IP**, und quittieren Sie mit einem Klick auf den Schalter **Anwenden**.
- Nun muss der Datenbankmoduldienst neu gestartet werden. Wählen Sie in der Baumansicht den Knoten **SQLEXPRESS > Datenbankmodul > Dienst**, beenden Sie den Dienst **MSSQL\$SQLEXPRESS** mit einem Klick auf den Schalter **Beenden**, und starten Sie ihn anschließend wieder mit einem Klick auf den Schalter **Starten**.

b) SQL Server Browser Dienst starten

Wenn der SQL-Server ohne Angabe einer Portnummer in der Verbindungszeichenfolge (vgl. Abschnitt 25.4.2) über Netz ansprechbar sein soll, muss der **SQL Server Browser Dienst** gestartet werden:

- Starten Sie die **SQL Server-Oberflächenkonfiguration** über ihren Link im Unterordner **Konfigurationstools** der Programmgruppe zum SQL Server.
- Klicken Sie auf **Oberflächenkonfiguration für Dienste und Verbindungen**.
- Wählen Sie in der Baumansicht am linken Rand des Dialogs den Knoten **SQLEXPRESS > SQL Server Browser > Dienst**, wählen Sie als **Starttyp** den Wert **Automatisch**, und quittieren Sie mit einem Klick auf den Schalter **Anwenden**.
- Klicken Sie auf **Starten** und dann auf **OK**.

Der Browser-Dienst hört den UDP-Port 1434 ab.

c) Firewall-Ausnahmen definieren

Ab Windows XP mit SP 2 muss für den SQL Server Dienst und für den SQL Server Browser Dienst jeweils eine Firewall-Ausnahme eingetragen werden:

- 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 Programm
C:\Programme\Microsoft SQL Server\MSSQL.1\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 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.
- Beenden Sie die Firewall-Konfiguration mit **OK**.

25.4 ADO.NET

25.4.1 Überblick

25.4.1.1 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 einem lokalen Cache zu arbeiten. Für solche Anwendungen bieten die ADO.NET – Provider (siehe Abschnitt 25.4.1.2) 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.

25.4.1.2 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. In der .NET – Version 2.0 werden Provider für die folgenden Datenbankmanagementsysteme bzw. Datenbankschnittstellen mitgeliefert:

- **SqlClient**
Dieser Provider unterstützt SQL-Server und ist speziell optimiert für Microsofts SQL Server ab Version 7.0.
- **OracleClient**
Dieser Provider unterstützt die Datenbanktechnik der Firma Oracle.
- **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.

Zum Provider **SqlClient**, den wir diesem Kurs überwiegend 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 25.4.2)
- **SqlCommand**
Repräsentiert ein SQL-Kommando (siehe Abschnitt 25.4.3)
- **SqlDataAdapter**
Diese Klasse vermittelt zwischen der Datenbankverbindung und der lokalen **DataSet**-Objekt (siehe Abschnitt 25.4.4).
- **SqlDataReader**
Erlaubt ein lesendes, sequentielles Durchlaufen der angeschlossenen Datenbank (siehe Abschnitt 25.4.9)
- **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.

25.4.1.3 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 das Abbild einer relationalen Datenbank (mit Tabellen und Beziehungen) und ermöglicht so die unverbundene Bearbeitung.
 - **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 25.1.2).
 - **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.OracleClient, System.Data.OleDb, System.Data.Odbc**

Für die Klassen der Provider für bestimmte Datenbanktechnologien (siehe oben) ist jeweils ein eigener Unternamensraum vorhanden.

In einem .NET - Programm mit Datenbankzugriff sollte man den Namensraum **System.Data** sowie den Unternamensraum zum eingesetzten Provider importieren.

25.4.2 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 konzentrieren uns anschließend auf die Verbindung zu einem SQL-Server.

25.4.2.1 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 zahlreiche Varianten in Frage¹. 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 SDK-Hilfe 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 2005 Express Edition wird per Voreinstellung mit dem Instanznamen **SQLEXPRESS** installiert.

Beispiel:

Data Source=(local)\SQLEXPRESS

¹ Es sind sogar spezialisierte Internet-Angebote mit Vorschlägen zur Bildung von korrekten Verbindungszeichenfolgen entstanden, z.B. <http://www.connectionstrings.com/>

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 25.3.3.2 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 25.3.3.2).

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.:

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.

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 gewöhnliche Netzfreigabe oder auf einem NAS-Server (*Network Attached Storage*) werden also nicht akzeptiert¹.

- **Benutzer - Authentifizierung**

Wie Sie schon aus Abschnitt 25.3.3.1 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

¹ Diese Regel mit dem gelegentlich empfohlenen Service-Startparameter -T1807 außer Kraft zu setzen, ist mir jedenfalls nicht gelungen.

- **Benutzerinstanzen der SQL Server 2005 Express Edition**

Die SQL Server 2005 *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. Insbesondere ist keinerlei Konfiguration im Sinn von Abschnitt 25.3.3.1 am SQL-Server vorzunehmen. Folglich ist eine Anwendung mit so implementiertem Datenbankzugriff leicht auf einen anderen Rechner zu übertragen. Nötigenfalls liefert man die SQL Server 2005 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

Benutzerinstanzen eignen sich gut für Datenbanken, die eng mit einer Anwendung verbunden und nur von *einer* Person benutzt werden sollen. Das Anwendungsprogramm, der SQL Server und die Datenbankdatei befinden sich auf dem Rechner, an dem der Benutzer angemeldet ist.

- **Zugriffsoptionen**

Über den Parameter **Connect Timeout** legt man fest, wie lange auf eine erfolgreiche Verbindung zum SQL Server gewartet werden soll (Voreinstellung: 15 Sekunden).

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 SDK-Hilfe zur **ConnectionString**-Eigenschaft der Klasse **SqlConnection**.

Weitere Hinweise zur Verbindungszeichenfolge

- Die Groß-/Kleinschreibung ist irrelevant.
- Die **Connection**-Klassen der Provider enthalten zu manchen Schlüsselwörtern eine schreibgeschützte Eigenschaft (z.B. die **Database**-Eigenschaft von **SqlConnection**-Objekten), deren Wert nur über den **ConnectionString** zu ändern ist.
- 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.

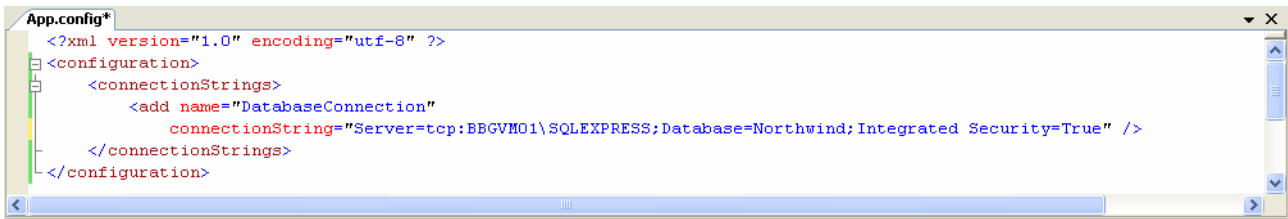
Am Anfang des aktuellen Abschnitts war eine komplette Verbindungszeichenfolge für den Zugriff auf eine lokale SQL Server - Benutzerinstanz zu sehen. Hier folgt nun ein Beispiel für den Netzwerkzugriff auf die primäre Instanz eines SQL Servers bei Verwendung der Windows - Authentifizierung:

```
dbConnection.ConnectionString = @"Server=tcp:BBGVM01\SQLEXPRESS;"+
    "Database=Northwind;Integrated Security=true";
```

Soll die Datenbankverbindung einer Anwendung flexibel konfigurierbar sein, eignet sich der Quellcode nicht zur Ablage der Verbindungszeichenfolge. Man bringt sie besser in der **Konfigurationsdatei** der Anwendung unter, die wir bei dieser Gelegenheit kennen lernen. Diese XML-Datei mit dem Namen **App.config** kann vom Visual Studio über das Item

Hinzufügen > Neues Element > Vorlage = Anwendungskonfigurationsdatei

aus dem Projekt-Kontextmenü angelegt und anschließend bearbeitet werden, z.B.:



Im Beispiel wird eine Verbindungszeichenfolge im **<connectionStrings>**-Element der Konfigurationsdatei gespeichert. Zur bequemen Verwendung der so abgelegten Verbindungszeichenfolgen im Programm bietet die Klasse **ConfigurationManager** (Namensraum **System.Configuration**) seit der .NET - Version 2.0 die Eigenschaft **ConnectionStrings** vom Typ **ConnectionStringSettingsCollection**. Statt die umständlich klingende Erläuterung zu vervollständigen, betrachten wir ein Beispiel für die angenehm einfache Anwendung:

```
dbConnection.ConnectionString =
    ConfigurationManager.ConnectionStrings["DatabaseConnection"].ConnectionString;
```

Um die Klasse **ConfigurationManager** verwenden zu können, ist ein Verweis auf das Assembly **System.configuration.dll** erforderlich.

Das Visual Studio erstellt im Ordner mit dem Exe-Assembly zu einem Projekt eine Kopie der Anwendungskonfigurationsdatei. So entsteht z.B. zur Programmdatei **Connect.exe** die Konfigurationsdatei **Connect.exe.config**.

Auf Techniken zur *verschlüsselten* Ablage von Verbindungszeichenfolge können wir hier aus Zeitgründen nicht eingehen.

25.4.2.2 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 25.4.9) ist das explizite Öffnen unumgänglich. Beim verbindungslosen Arbeiten mit der Klasse **SqlDataAdapter** (siehe Abschnitt 25.4.4) 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.

25.4.3 Die Command-Klassen

Für SELECT- oder Aktionsbefehle (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.

25.4.3.1 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 25.4.3.2 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 25.4.4) 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 25.4.9).
- **ExecuteNonQuery()**
Mit dieser Methode lässt man DDL-Befehle (wie CREATE TABLE) oder Aktionsabfragen (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();
```

25.4.3.2 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 schreiben. Im folgenden Programm

```
using System;
using System.Data;
using System.Data.SqlClient;

class CommandDemo {
    SqlConnection dbConnection = new SqlConnection();
    SqlCommand selAvgSupp;

    public 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);
    }

    public void AvgPriceSupplier() {
        try {
            Console.Write("Nummer des Anbieters: ");
            int supp = Convert.ToInt32(Console.ReadLine());
            selAvgSupp.Parameters["@SuppID"].Value = supp;
            dbConnection.Open();
            Console.WriteLine("Mittl. Preis d. 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 Lieferantennummer 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 mit den üblichen Kollektionsmethoden (z.B. **Add()**) Objekte der Klasse **SqlParameter** verwaltet. 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. SDK-Hilfe zur **Parameters**-Eigenschaft der Klasse **OleDbCommand**).

25.4.4 Die DataAdapter - Klassen

Beim verbindungslosen Arbeiten spielt ein **DataAdapter**-Objekt die zentrale Rolle:

- Es führt mit Hilfe von **Command**-Objekten Auswahl- und Aktionsabfragen durch und vermittelt dabei zwischen dem 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 ein **DataSet**-Objekt kopiert werden. Mehrere Tabellen kommen z.B. dann zu Stande, wenn der **CommandText** des **SelectCommand**-Objekts eine Serie von SELECT-Kommandos enthält, z.B.:

```
SELECT * FROM Customers;SELECT * FROM Employees"
```

Die im folgenden Beispiel verwendete **Fill()** - Überladung erwartet den Namen des **DataSet**-Objekts und den gewünschten programminternen Tabellennamen:

```
dbAdapter.Fill(ds, "Employees");
```

Bei Verzicht auf die Angabe eines Tabellennamens erhält die Eigenschaft

```
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 im Beispiel das beteiligte **SelectCommand**-Objekt *mehrere* Tabellen liefern, würden diese die Namen **Employees**, **Employees1**, etc. erhalten. Ein **DataAdapter** bietet über seine Eigenschaft **TableMappings** auch eine Möglichkeit zur individuellen Benennung *mehrerer* Ergebnistabellen. Außerdem ist mir die Übertragung von Datenbank-Gültigkeitsregeln (siehe unten) auf aussagekräftig bezeichnete **DataSet**-Tabellen erst gelungen, nachdem die Korrespondenz zwischen den Standardnamen *Table*, *Table1*, ... und den gewünschten **Dataset**-Tabellennamen über die per

TableMappings-Eigenschaft ansprechbare **DataTableMappingCollection** vorgenommen wurde. Es empfiehlt sich also auch beim Zugriff auf eine einzelne Tabelle, zunächst per **Add()**-Aufruf die Zuordnung vorzunehmen und den **Fill()**-Aufruf ohne Namensangabe dahinter zu setzen, z.B.:

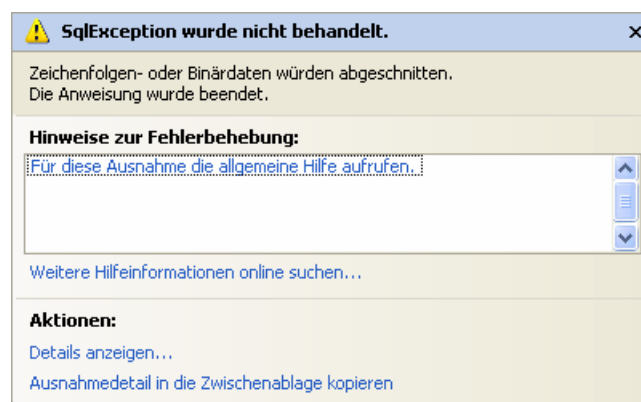
```
dbAdapter.TableMappings.Add("Table", "Employees");  
dbAdapter.Fill(ds);
```

Beim Aufruf der **DataAdapter**-Methode **Update()** werden die Aktionsabfragen 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 25.4.7 ist zu erfahren, wie die Aktionsabfragen per **CommandBuilder** entstehen und was bei **DataSet**-Objekten mit mehreren Tabellen zu beachten ist.

Die in einer Datenbank definierten Gültigkeitsregeln (z.B. Länge von Zeichenfolge) werden durch einen **Fill()**-Aufruf *nicht* zum lokalen **DataSet**-Objekt übertragen, was nach unpassenden Modifikationen zu Laufzeitfehlern beim **Update()** führt, z.B. bei einer Längenüberschreitung:

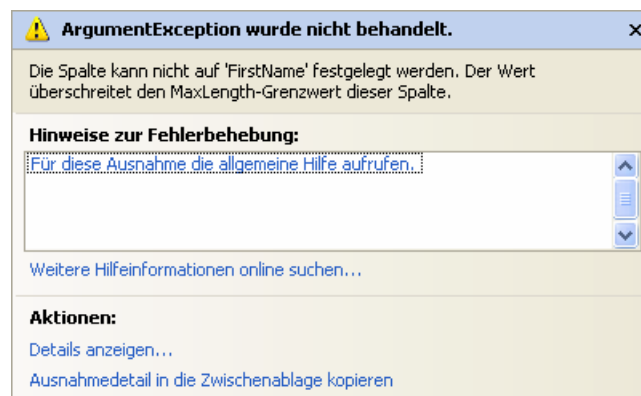


Um das so genannte *Schema* der Datenbank mit den Gültigkeitsregeln in ein **DataSet** zu übertragen, kann man die **DataAdapter**-Methode **FillSchema()** verwenden, z.B.:

```
dbAdapter.FillSchema(ds, SchemaType.Mapped);
```

Die Methode **FillSchema()** erfragt die Gültigkeitsregeln (aber keine Werte) beim DBMS und erstellt ggf. ein **DataSet**-Objekt. Wie bei der Methode **Fill()** erfolgt das Öffnen und Schließen der Verbindung automatisch. 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 den **SchemaType.Mapped** wählen.

Zwar führt ein Regelverstoß weiterhin zu einem Ausnahmefehler, auf den das Programm reagieren muss, doch wird das Problem nun ohne zeitaufwändige Datenbankverbindung erkannt:



Erfreulich ist bei den **DataAdapter**-Methoden **Fill()**, **FillSchema()** und **Update()** die Fähigkeit, bei Bedarf eine Datenbankverbindung automatisch zu öffnen und zu schließen (vgl. Abschnitt 25.4.2.2).

25.4.5 Die Klassen **DataSet**, **DataTable**, **DataRow** und **DataColumn**

In diesem (aus Zeitgründen etwas zu kurz geratenen) Abschnitt werden wichtige Klassen aus dem Provider - unabhängigen Namensraum **System.Data** behandelt. Sie speichern beim verbindungslosen Arbeiten (vgl. Abschnitt 25.4.1.1) lokale Kopien der angeforderten Datenbanktabellen samt Gültigkeitsregeln und Beziehungen.

In der folgenden Sequenz werden aus den **Northwind**-Tabellen **Customers** und **Orders** jeweils einige Spalten in ein Zweitabellen - **DataSet**-Objekt übernommen:

```
DataSet ds = new DataSet();
DataTable dtCustomers, dtOrders;

dbConnection.ConnectionString = @"Data Source=(local)\SQLEXPRESS;" +
    @"AttachDbFilename=E:\Data\NORTHWND.MDF;" +
    "Integrated Security=True;User Instance=True";
SqlCommand = new SqlCommand(
    "SELECT CustomerID, CompanyName, Country FROM Customers;" +
    "SELECT CustomerID, ShipCountry FROM Orders", dbConnection);
dbAdapter.SelectCommand = sqlCommand;

dbAdapter.TableMappings.Add("Table", "Customers");
dbAdapter.TableMappings.Add("Table1", "Orders");

try {
    dbConnection.Open();
    dbAdapter.Fill(ds);
    dbAdapter.FillSchema(ds, SchemaType.Mapped);
} 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.

Mit den beiden folgenden Zeilen

```
Console.WriteLine(
    "MaxLength(CompanyName) = " + dtCustomers.Columns["CompanyName"].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 **dtCustomers**-Eigenschaft **PrimaryKey**, die auf einen **DataColumn**-Array zeigt, versorgt worden.

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 der Customers-Tabelle geschieht:

```
public void PrintData() {
    Console.WriteLine("{0,10} {1,40} {2,20}\n", "CustomerID", "Company", "Country");
    for (int i = 0; i < dtCustomers.Rows.Count; i++)
        Console.WriteLine("{0,10} {1,40} {2,20}", dtCustomers.Rows[i][0],
            dtCustomers.Rows[i][1], dtCustomers.Rows[i]["Country"]);
}
```

Bei den Spalten einer Tabelle gelingt der Zugriff alternativ über dem numerischen Index und über die Feldnamen. Die ersten Ausgabzeilen der Methode:

CustomerID	Company	Country
ALFKI	Alfreds Futterkiste	Germany
ANATR	Ana Trujillo Emparedados y helados	Mexico
ANTON	Antonio Moreno Taquería	Mexico
AROUT	Around the Horn	UK

25.4.6 Datenbank-Beziehungen in ein DataSet übernehmen

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. 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 das **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:

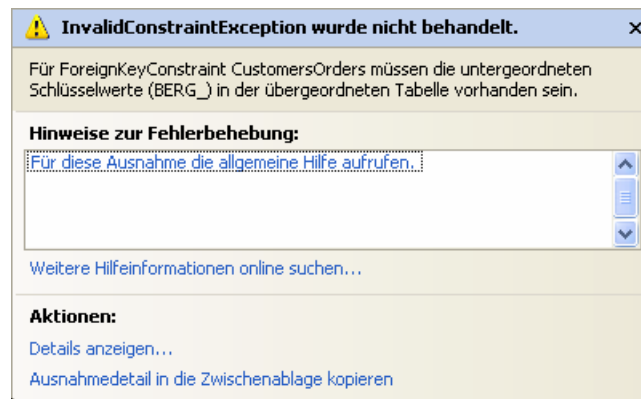
```
DataRelation co = ds.Relations["CustomersOrders"];
Console.WriteLine("\nParentKeyConstraint zur Beziehung \"CustomersOrders\": "+
    "\n Tabelle: " + co.ParentKeyConstraint.Table.TableName +
    "\n Feld:    " + co.ParentKeyConstraint.Columns[0].ColumnName +
    "\n Typ:     " + co.ParentKeyConstraint.GetType().ToString());
Console.WriteLine("\nChildKeyConstraint zur Beziehung \"CustomersOrders\": "+
    "\n Tabelle: " + co.ChildKeyConstraint.Table.TableName +
    "\n Feld:    " + co.ChildKeyConstraint.Columns[0].ColumnName +
    "\n Typ:     " + co.ChildKeyConstraint.GetType().ToString());
```

kann man Informationen zu den Gültigkeitsregeln anfordern, die aus einer Beziehung resultieren:

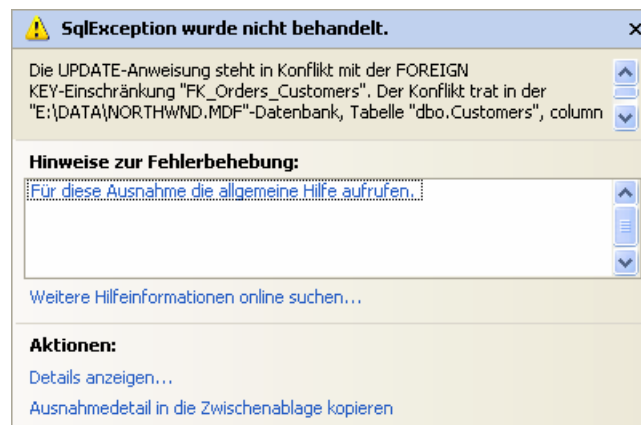
```
ParentKeyConstraint zur Beziehung "CustomersOrders":
Tabelle: Customers
Feld:    CustomerID
Typ:     System.Data.UniqueConstraint
```

```
ChildKeyConstraint zur Beziehung "CustomersOrders":
Tabelle: Orders
Feld:    CustomerID
Typ:     System.Data.ForeignKeyConstraint
```

Sind solche Gültigkeitsregeln lokal 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 auch über deren **Constraints**-Eigenschaft ansprechbar, die auf eine Kollektion von **Constraint**-Objekten zeigt. So gehört im Beispiel das **ForeignKeyConstraint** zur **Constraints**-Kollektion der **Orders**-Tabelle.

25.4.7 Datenbank-Update

Zum Aktualisieren der Datenbank durch Übertragung des aktuellen **DataSet**-Zustands ist lediglich ein Aufruf der **DataAdapter**-Methode **Update()** erforderlich, z.B.:

```
dbAdapter.Update(ds, "Personal");
```

Der **DataAdapter** benötigt jeweils ein SQL-Kommando zum Ändern von Werten (UPDATE), Einfügen von Zeilen (INSERT) und zum Löschen von Zeilen (DELETE). Man kann diese SQL-Kommandos, die zwecks Aktualisierung dem DBMS zugestellt werden, 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:

```
SqlCommandBuilder cb = new SqlCommandBuilder(dbAdapter);
```

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, was z.B. zum Erstellen mehrerer **DataTable**-Objekte sinnvoll und erlaubt ist, beachtet der **CommandBuilder** nur das *erste* SELECT-Kommando. Dementsprechend werden nur Änderungen an der korrespondierenden *ersten* Tabelle von der **Update()**-Methode berücksichtigt. Auf den **CommandBuilder** zu verzichten und die Aktionskommandos selbst zu erstellen, kann sehr aufwändig sein. Eine 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);

```

Zum Aktualisieren der Datenbank sind dementsprechend zwei **Update()**-Aufrufe erforderlich:

```

public void UpdateData() {
    try {
        dbConnection.Open();
        dbAdapterCus.Update(ds);
        dbAdapterOrd.Update(ds);
    } finally {
        dbConnection.Close();
    }
}

```

25.4.8 Zusammenspiel der Klassen beim verbindungslosen Arbeiten

Im folgenden Programm werden im Sinne der verbindungslosen Datenbankbearbeitung aus der von einer SQL Server - Benutzerinstanz verwalteten Datenbank **Northwind** 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 Disconnected {
    SqlConnection dbConnection = new SqlConnection();
    SqlCommand selCommandCus, selCommandOrd;
    SqlDataAdapter dbAdapterCus, dbAdapterOrd;
    DataSet ds = new DataSet();
    DataTable dtCustomers, dtOrders;

    public Disconnected() {
        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);

        // Aktionskommandos für beide DataAdapter automatisch erstellen lassen:
        new SqlCommandBuilder(dbAdapterCus);
        new SqlCommandBuilder(dbAdapterOrd);

        // TableMappings erleichtern die Übernahme von Schema-Informationen:
        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"];

// Vereinbarung der Master-Detail-Beziehung:
DataColumn parent = dtCustomers.Columns["CustomerID"];
DataColumn child = dtOrders.Columns["CustomerID"];
ds.Relations.Add(new DataRelation("CustomersOrders", parent, child));
}

public void PrintData() {
    Console.WriteLine("{0,10} {1,40} {2,20}\n",
        "CustomerID", "Company", "Country");
    for (int i = 0; i < 5; i++)
        Console.WriteLine("{0,10} {1,40} {2,20}", dtCustomers.Rows[i][0],
            dtCustomers.Rows[i][1], dtCustomers.Rows[i]["Country"]);
    Console.WriteLine("\n\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]);
}

public void ChangeData() {
    dtCustomers.Rows[0]["Country"] = "Lummerland";
    dtOrders.Rows[0]["CustomerID"] = "TOMSP";
}

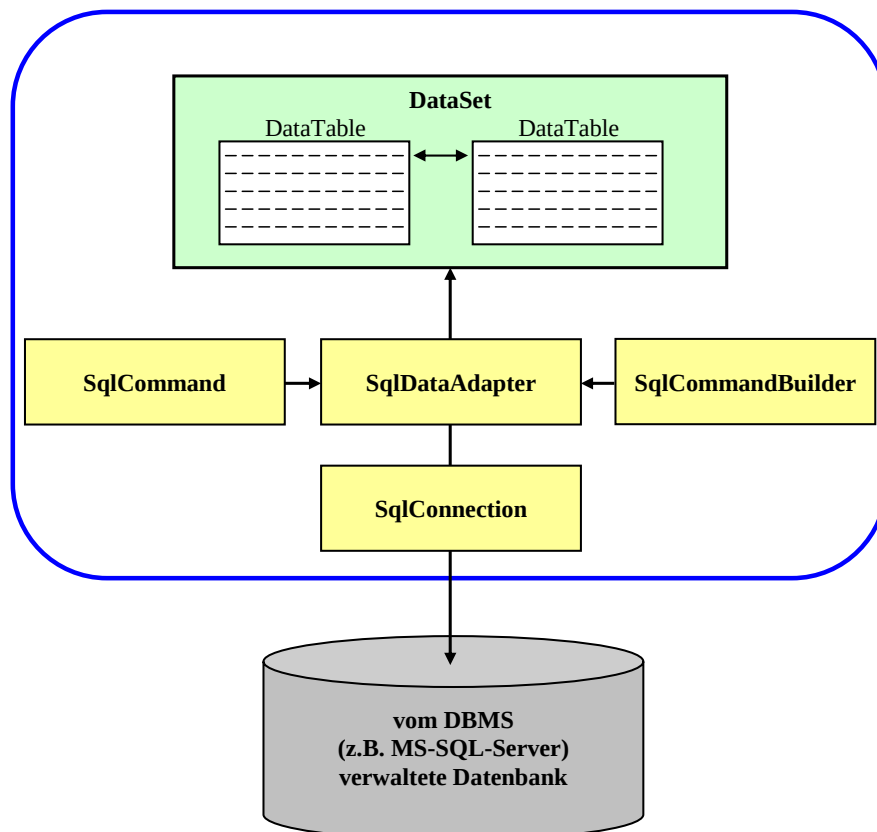
public void UpdateData() {
    try {
        dbConnection.Open();
        dbAdapterCus.Update(ds);
        dbAdapterOrd.Update(ds);
    } finally {
        dbConnection.Close();
    }
}

static void Main() {
    Disconnected dsd = new Disconnected();
    dsd.PrintData();
    dsd.ChangeData();
    dsd.UpdateData();
}
}

```

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:



25.4.9 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 dbConnector = new SqlConnection();
        dbConnector.ConnectionString =
            @"Data Source=.\SQLEXPRESS;AttachDbFilename=E:\Data\NORTHWND.MDF;" +
            @"Integrated Security=True;User Instance=True";

        String strSqlCmd = "SELECT CompanyName FROM Customers";
        SqlCommand command = new SqlCommand(strSqlCmd, dbConnector);

        dbConnector.Open();

        SqlDataReader reader = command.ExecuteReader();

        while (reader.Read()) {
            Console.WriteLine(reader["CompanyName"]);
        }

        // Schließen des Readers gibt das SqlConnection-Objekt für eine andere
        // Verwendung frei
        reader.Close();
        // Schließen der Datenbankverbindung:
        dbConnector.Close();
    }
}
  
```

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, 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.

25.5 Datenbankunterstützung im Visual Studio 2005

25.5.1 Datenquellen

Bei der Datenbankunterstützung im Visual Studio 2005 spielt das Konzept der *Datenquelle* eine zentrale Rolle, wobei es um mehrere problemspezifisch definierte Klassen geht:

- eine typisierte **DataSet**-Klasse

Eine typisierte **DataSet**-Klasse wird vom Visual Studio 2005 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 vom Visual Studio 2005 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 {
    . . .
}
```

Der **TableAdapter** ist für das Befüllen der zugehörigen **DataTable** und das Datenbank-Update zuständig.

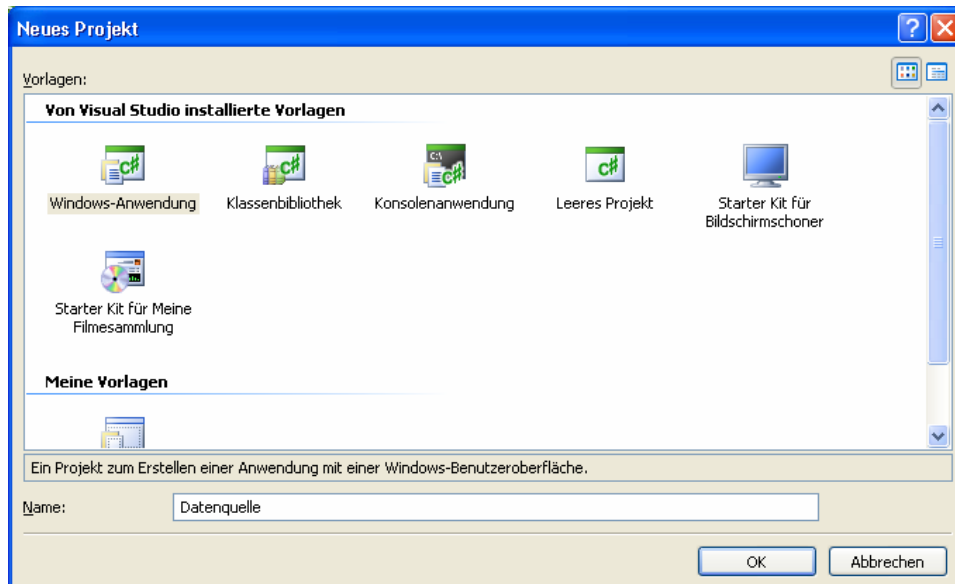
Eine Datenbank-Datenquelle sensu Visual Studio 2005 enthält also mindestens eine typisierte **DataSet**-Klasse und eine typisierte **DataTable**-Ableitung mit zugehöriger **TableAdapter**-Klasse.

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.

25.5.2 Anwendungsbeispiel

Wir erstellen eine Winforms-Anwendung mit Datenbankzugriff:

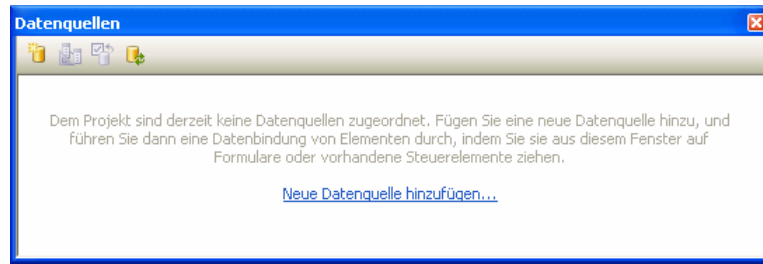
- Legen Sie ein neues Projekt auf Basis der Vorlage **Windows-Anwendung** an, z.B.:



- Öffnen Sie nötigenfalls mit dem Menübefehl

Daten > Datenquellen anzeigen

das **Datenquellen**-Fenster. Es enthält einen Link, der zum Ergänzen einer neuen Datenquelle auffordert:

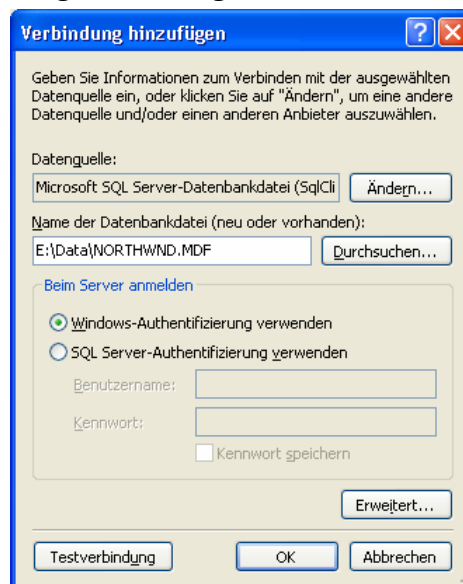


- Nach einem Klick auf diesen Link startet der Datenquellen-Konfigurationsassistent:

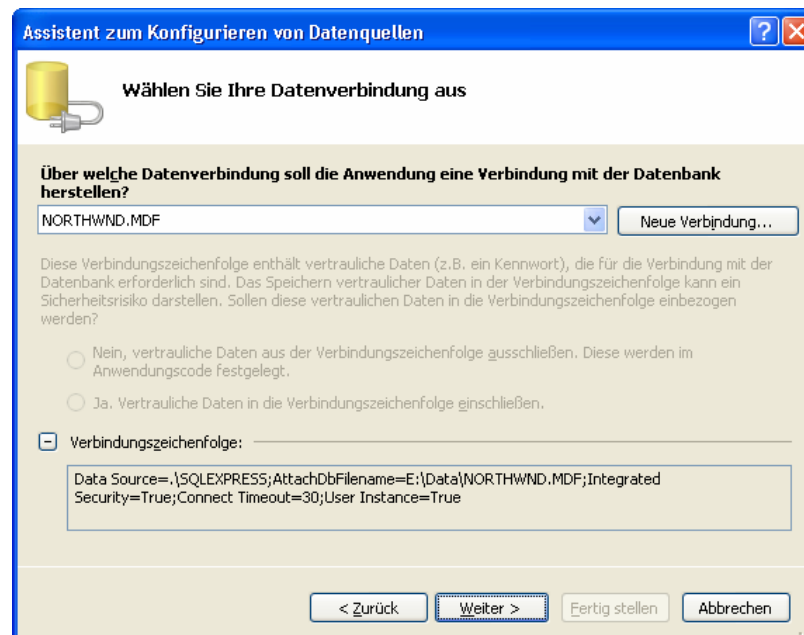


Wählen Sie den Datenquellentyp **Datenbank**. Mit den beiden anderen Optionen können wir uns in diesem Kurs aus Zeitgründen nicht beschäftigen.

- Klicken Sie im Dialog zur Auswahl einer Datenbankverbindung (siehe unten) auf den Schalter **Neue Verbindung**, um folgende Dialogbox zu erhalten:

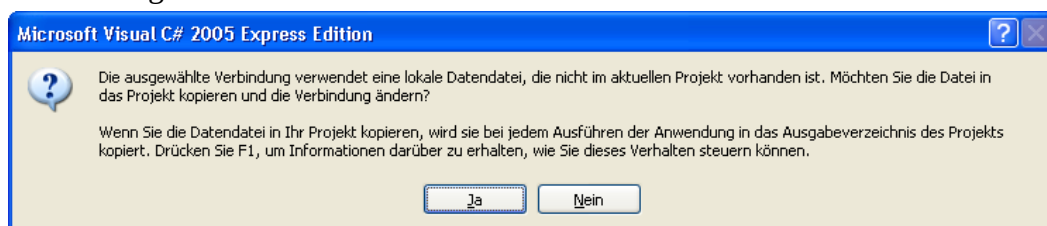


- Wählen Sie die Datenbankdatei **Northwnd.mdf**, und quittieren Sie mit **OK**, so dass im Dialog zur Auswahl einer Datenbankverbindung nun die Verbindungszeichenfolge eingesehen werden kann:



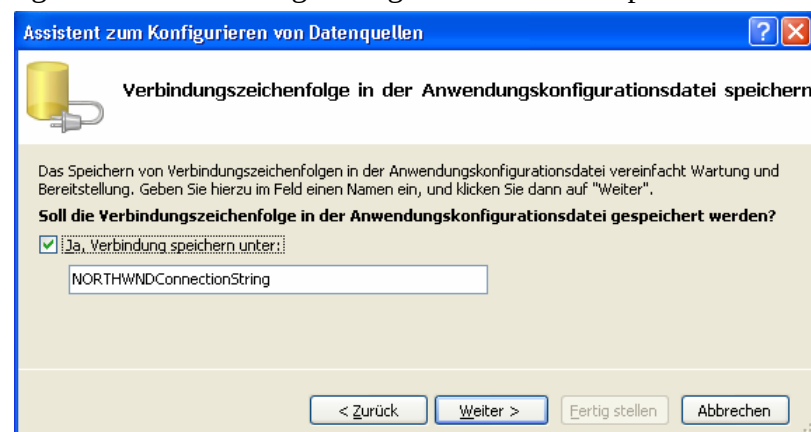
Das Ergebnis entspricht ziemlich genau unseren Erwartungen auf Basis von Abschnitt 25.4.2.1.

- Wenn Sie **Weiter** machen, erkundigt sich das Visual Studio 2005 eventuell nach Ihren Absichten zur Ablage der Datenbankdatei:

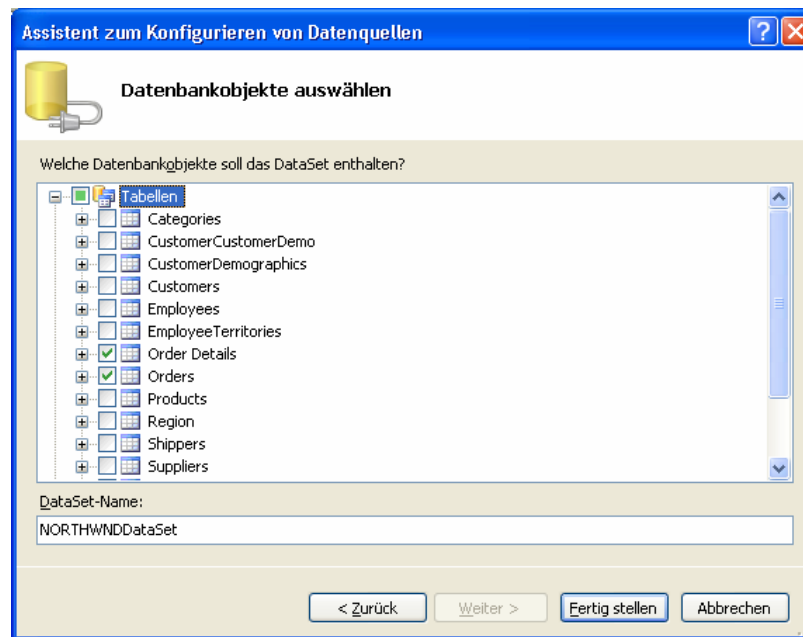


Eine Zustimmung zur projektinternen Kopie verbessert auf jeden Fall die Portabilität des Projekts.

- Die nächste Frage bezieht sich auf die in Abschnitt 25.4.2.1 behandelte Option, die Verbindungszeichenfolge in der Anwendungskonfigurationsdatei zu speichern:

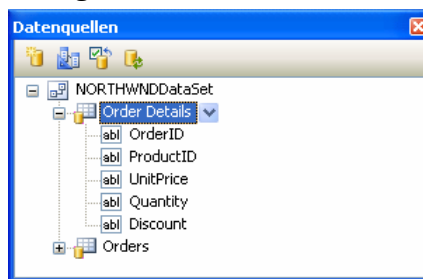


- Schließlich geht es an die Auswahl der benötigten Objekte aus der **Northwind**-Datenbank:

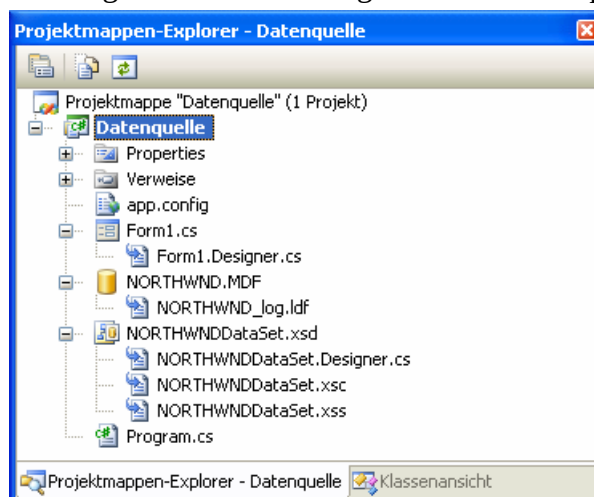


Wählen Sie die Tabellen **Order Details** und **Orders**. Von der Möglichkeit, einzelne Tabellenspalten auszuwählen, machen wir keinen Gebrauch.

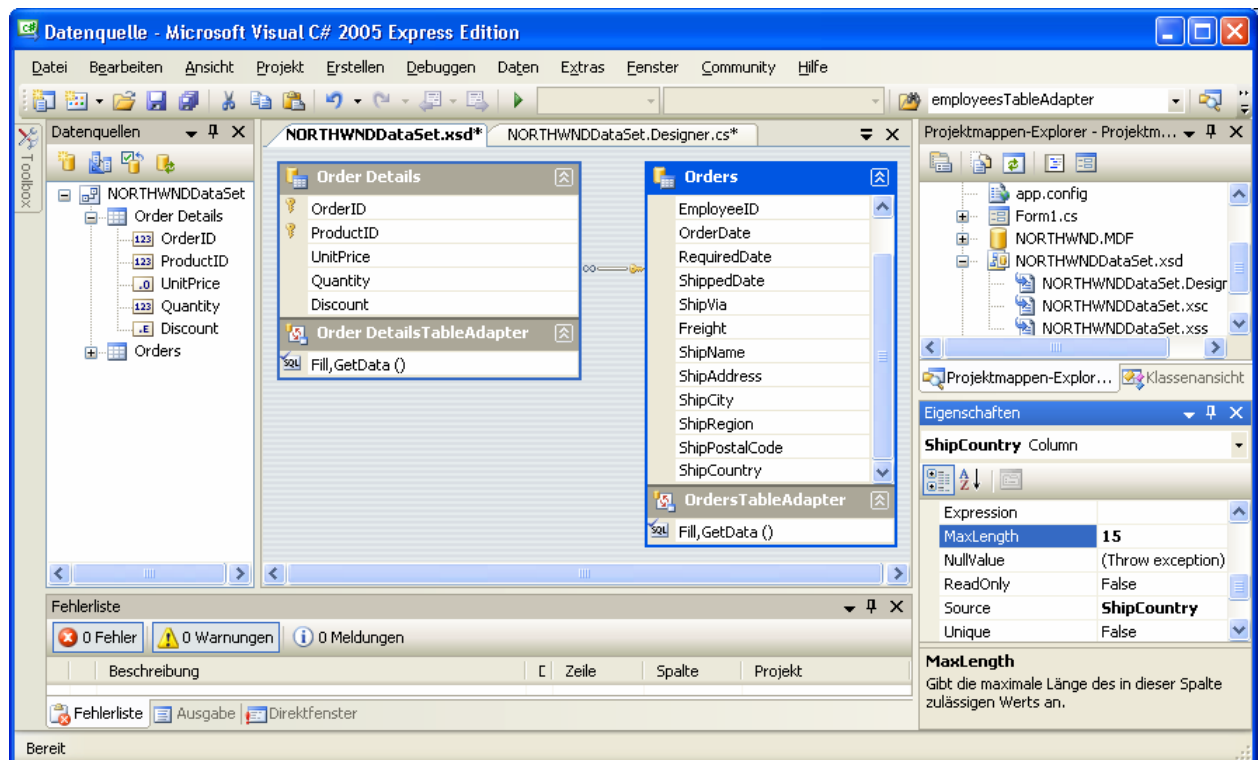
- Beenden Sie den Assistenten mit einem Klick auf den Schalter **Fertig stellen**. Im **Datenquellen**-Fenster zeigt sich nun folgendes Bild:



- Im Projektmappenexplorer zeigt sich nach dem Ergänzen der Datenquelle folgendes Bild:

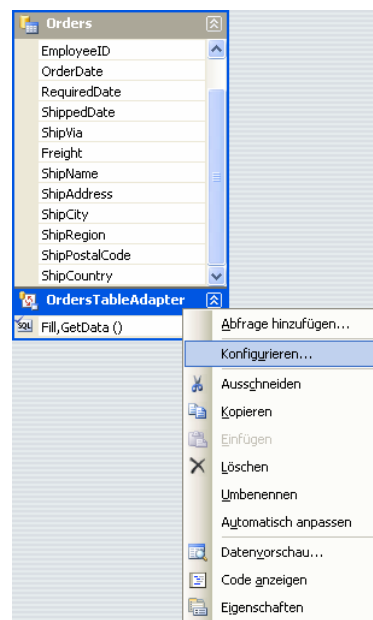


- Das Visual Studio 2005 hat mittlerweile die in Abschnitt 25.5.1 beschriebenen Klassen für die spezifischen Datenbankbedürfnisse unserer Anwendung angelegt, die nun mit Hilfe des **DataSet-Designers** konfiguriert werden können. Wählen Sie aus dem Kontextmenü zum Datenquellenfenster das Item **DataSet mit Designer bearbeiten**.

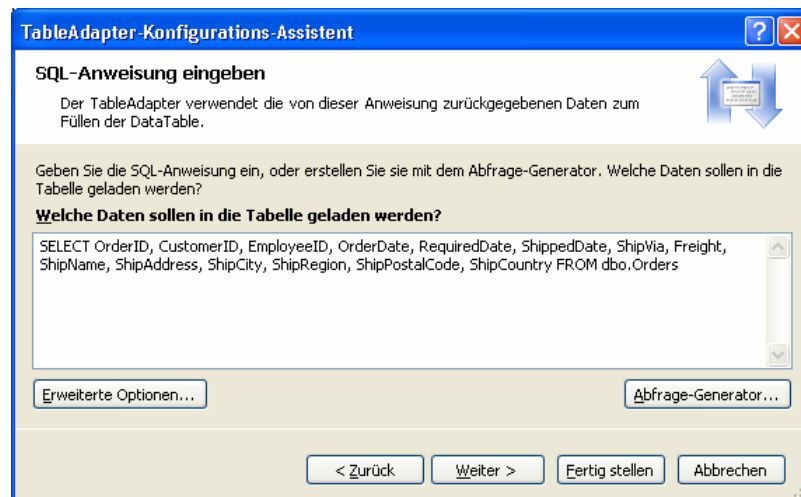


Wir können nun auf sehr intuitive Weise die Schemadatei **NORTHWNDDataSet.xsd** zum typisierten **DataSet** bearbeiten. Auf den ersten Blick wird klar, dass auch die Master-Details - Beziehung aus der Datenbank übernommen wurde.

- 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.:

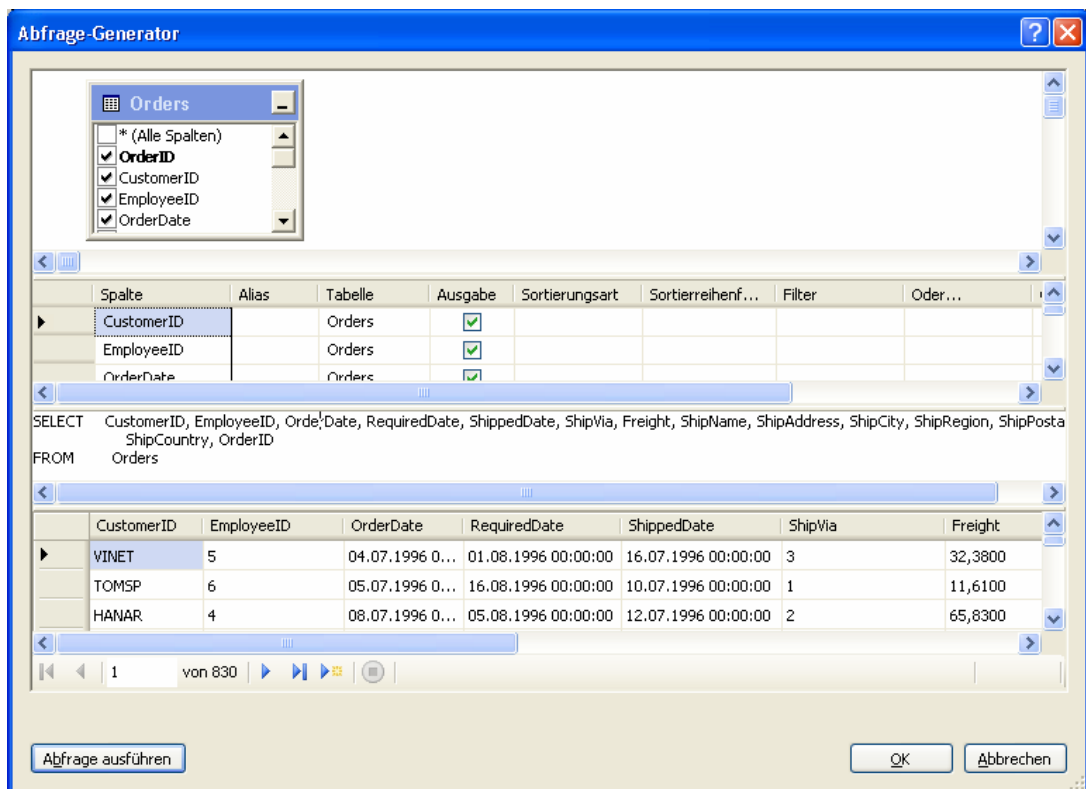


Hier ist das SELECT-Kommando zur Auswahlabfrage zu sehen und zu modifizieren:

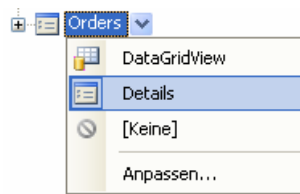


Es handelt sich um die **CommandText**-Eigenschaft eines **SqlCommand**-Objekts, wie sich durch Inspektion der vom DataSet-Designer gepflegten Quellcodedatei **NORTHWND-DataSet.Designer.cs** leicht bestätigen lässt.

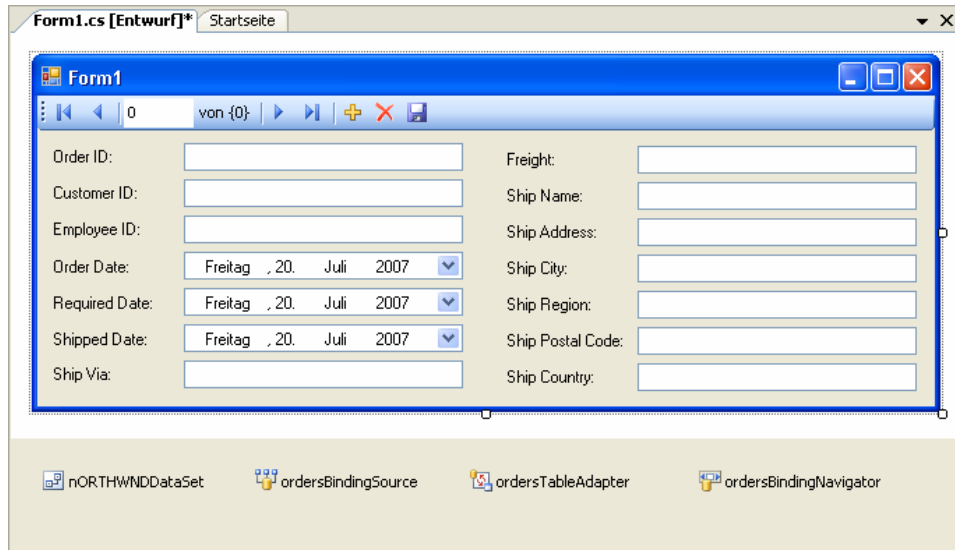
- Statt das SELECT-Kommando direkt zu verändern, kann man den **Abfrage-Generator** bemühen:



- Wir ignorieren aus Zeitgründen viele Optionen zur Assistenten-gestützten Gestaltung der **TableAdapter**-Klassen, über die z.B. die folgende Microsoft-Webseite informiert:
<http://www.microsoft.com/germany/msdn/library/visualtools/visualstudio2005/NeueDataSetFeaturesInVisualStudio2005.msp>
- 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 Orders-Tabelle anstelle der Voreinstellung **DataGridView** die Option **Details**,



so dass jede Spalte ein eigenes Steuerelement gemäß Typeinstellung im Datenquellenfenster erhält. Ziehen Sie die Orders-Tabelle auf das Formular und verbessern Sie die Anordnung der Steuerelemente, z.B.:



- Im Komponentenfach unterhalb des Formulars für die Steuerelemente ohne visuellen Auftritt tauchen vier Komponenten auf, von denen uns bisher erst zwei bekannt sind: die Objekte aus den vom Visual Studio definierten Klassen **NORTHWNDDataSet** und **OrdersTableAdapter**. Das Objekt **ordersBindingSource** 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.ordersBindingSource.DataSource = this.nORTHWNDDataSet;
this.ordersBindingSource.DataMember = "Orders";
```
 - o Um die Steuerelemente mit dem **BindingSource**-Objekt zu verbinden, wird dieses in die **DataBindings**-Liste der Steuerelemente aufgenommen, z.B.:

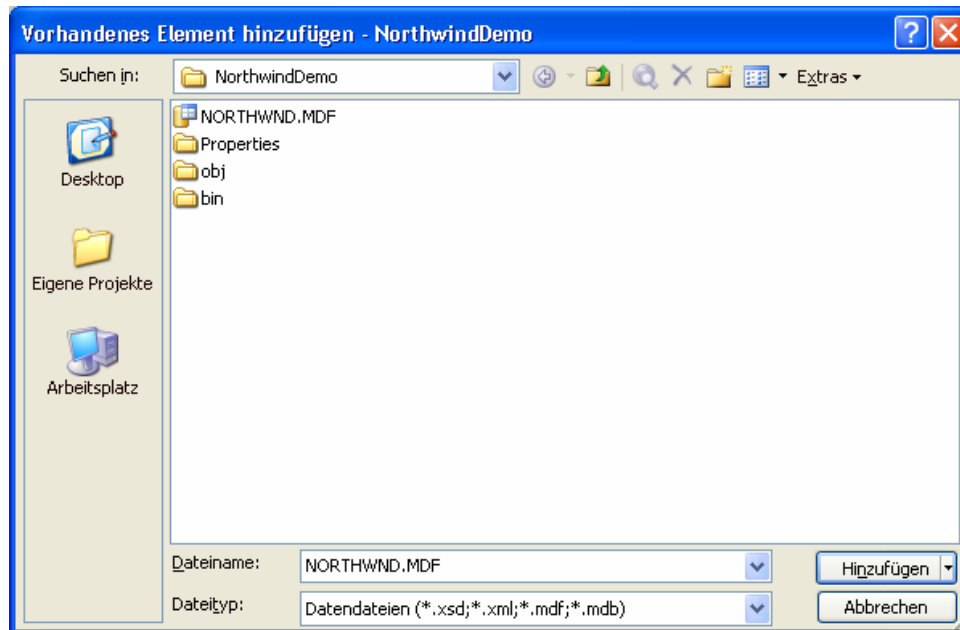

```
this.customerIDTextBox.DataBindings.Add(
    new System.Windows.Forms.Binding("Text",
    this.ordersBindingSource, "CustomerID", true));
```
- Die Referenzvariable **ordersBindingNavigator** zeigt auf ein Steuerelement aus der mit .NET 2.0 eingeführten Klasse **BindingNavigator** (Namensraum **System.Windows.Forms**). Diese Klasse realisiert eine Standardsymbolleiste zur Tabellenbearbeitung. Für die Datenbindung sorgt die Eigenschaft **BindingSource**, der ein **BindingSource**-Objekt zugewiesen wird, z.B.:


```
this.ordersBindingNavigator.BindingSource = this.ordersBindingSource;
```

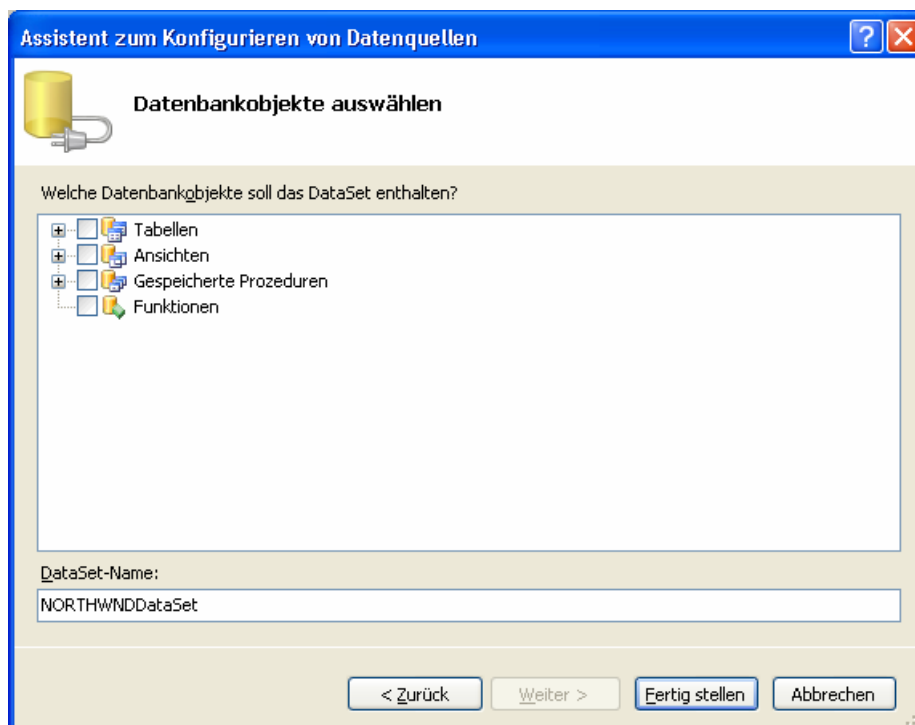
25.5.3 Datenbanken mit dem Visual Studio 2005 bearbeiten

Das Visual Studio 2005 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 Datenbank-

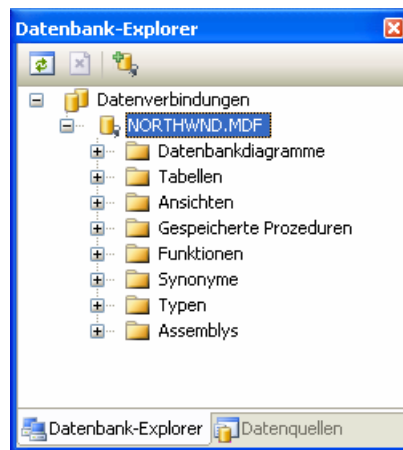
datei **NORTHWND.MDF** in den Ordner eines neuen Windows-Projekts. Nehmen Sie die Datenbank per Projektmappen-Explorer über das Kontextmenü des Projekts mit **Hinzufügen > Vorhandenes Element** in ein Projekt auf:



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:



25.5.3.1 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:

Spaltenname	Datentyp	NULL zula...
TitleOfCourtesy	nvarchar(25)	☒
BirthDate	datetime	☒
HireDate	datetime	☒
Address	nvarchar(60)	☒
City	nvarchar(15)	☒
Region	nvarchar(15)	☒
PostalCode	nvarchar(10)	☒
Country	nvarchar(15)	☒
HomePhone	nvarchar(24)	☒
Extension	nvarchar(4)	☒
Photo	image	☒
Notes	ntext	☒
ReportsTo	int	☒
PhotoPath	nvarchar(255)	☒
		☐

Jetzt lassen sich z.B. neue Spalten definieren.

Im Kontextmenü zum **Tabellen**-Zweig im Datenbank-Explorer findet sich die Option, eine neue Tabelle anzulegen.

Über die Option **Tabellendaten anzeigen** im Kontextmenü einer vorhandenen Tabelle kann man in der Designer -Zone ihre Daten anzeigen und bearbeiten, z.B.:

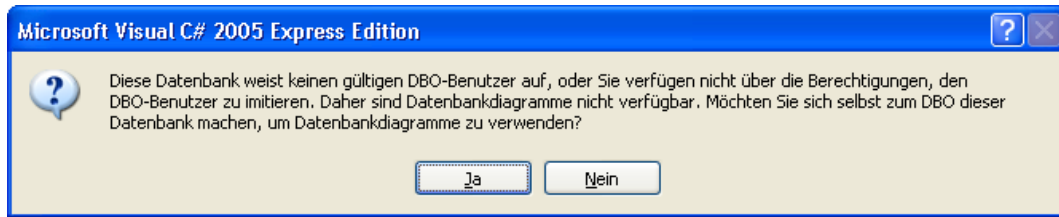
Employees: A...ORTHWN.DMF)										
	EmployeeID	LastName	FirstName	Title	TitleOfCourtesy	BirthDate	HireDate	Address	City	Region
1	1	Davolio	Nancy	Sales Representative	Ms.	08.12.1948 ...	01.05.1992 ...	507 - 20th Ave. ...	Seattle	WA
2	2	Fuller	Andrew	Vice President, Sales	Dr.	19.02.1952 ...	14.08.1992 ...	908 W. Capital ...	Tacoma	WA
3	3	Leverling	Janet	Sales Representative	Ms.	30.08.1963 ...	01.04.1992 ...	722 Moss Bay Bl...	Kirkland	WA
4	4	Peacock	Margaret	Sales Representative	Mrs.	19.09.1937 ...	03.05.1993 ...	4110 Old Redmo...	Redmond	WA
5	5	Buchanan	Steven	Sales Manager	Mr.	04.03.1955 ...	17.10.1993 ...	14 Garrett Hill	London	NULL
6	6	Suyama	Michael	Sales Representative	Mr.	02.07.1963 ...	17.10.1993 ...	Coventry House...	London	NULL
7	7	Brgel	Robert	Sales Representative	Mr.	29.05.1960 ...	02.01.1994 ...	Edgeham Hollow...	London	NULL
8	8	Callahan	Laura	Inside Sales Coordinator	Ms.	09.01.1958 ...	05.03.1994 ...	4726 - 11th Ave...	Seattle	WA
9	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.

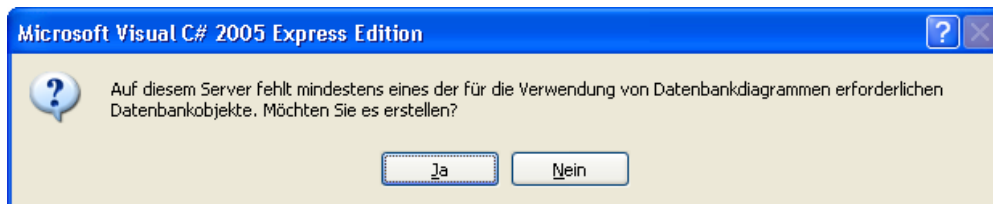
25.5.3.2 Datenbankdiagramm erstellen

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 der Datenbank.

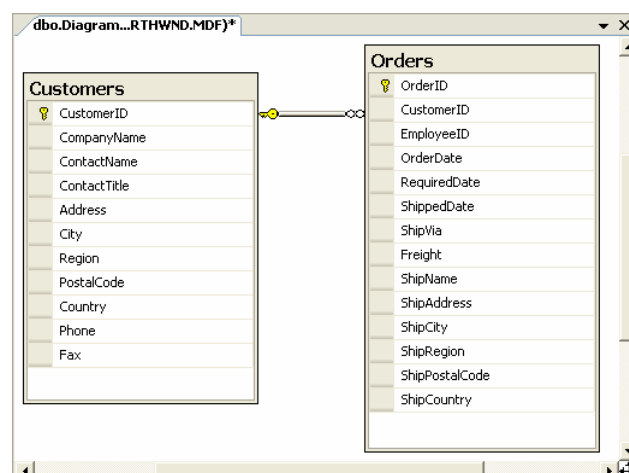
Veranlassen Sie ggf. anschließend das Erstellen fehlender Datenbankobjekte:



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 durch Verschieben von Tabellen und Relationspfeilen gestalten:

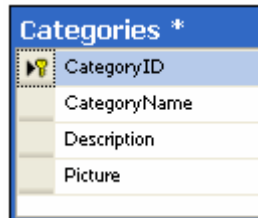


Beim Speichern des Diagramms (per **Strg-S** oder Symbolschalter angefordert) legt man seinen Namen fest.

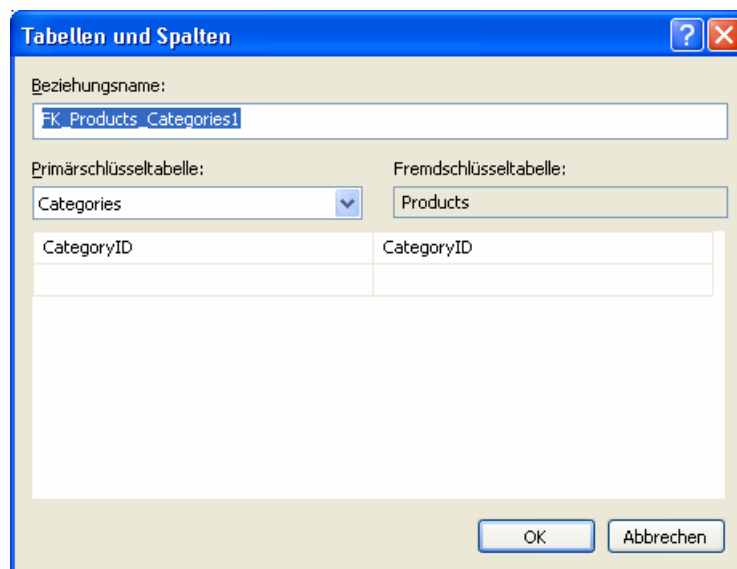
25.5.3.3 Beziehungen definieren

Über ein Datenbankdiagramm lassen sich Relationen definieren:

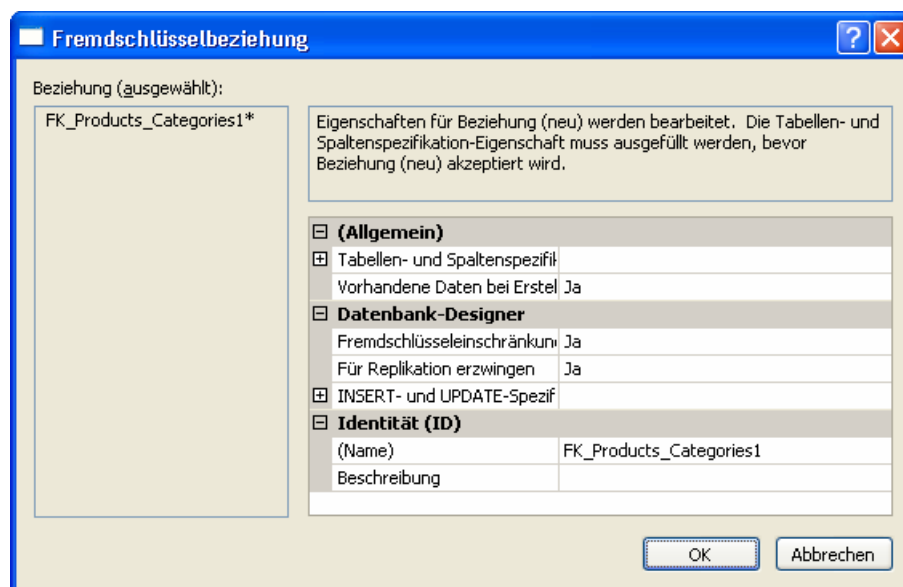
- Primärschlüssel per Klick auf das Schlüsselssymbol markieren, z.B.:



- 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**



jeweils mit **OK** quittieren.

26 Anhang

26.1 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
$x.y$	Member-Zugriff
$()$	Methodenaufruf
$[]$	Array-Index
$++, --$	Postinkrement bzw. -dekrement
<code>new</code>	Objekterzeugung
<code>checked</code>	Ganzzahl-Überlaufdiagnose
$-$	Vorzeichenumkehr
$!$	Negation
$++, --$	Präinkrement bzw. -dekrement
(Typ)	Typumwandlung
$*, /$	Punktrechnung
$\%$	Modulo (Divisionsrest)
$+, -$	Strichrechnung
$+$	Stringverkettung
$<<, >>$	Links- bzw. Rechts-Shift

Operator	Bedeutung
>, <, >=, <=, is	Vergleichsoperatoren
==, !=	Gleichheit, Ungleichheit
&	Bitweises UND
&	Logisches UND (mit unbedingter Auswertung)
^	Exklusives logisches ODER
	Bitweises ODER
	Logisches ODER (mit unbedingter Auswertung)
&&	Logisches UND (mit bedingter Auswertung)
	Logisches ODER (mit bedingter Auswertung)
? :	Konditionaloperator
=	Wertzuweisung
+=, -=, *=, /=, %=	Wertzuweisung mit Aktualisierung

26.2 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 in großen Projekten,
- die Wiederverwendung von Software.

Aufgabe 2

- Richtig
- Falsch
Allerdings sollte man in der Regel für Klasse und Datei denselben Namen wählen.
- Falsch
Die Standardbibliothek (FCL) ist für alle .NET – Sprachen identisch.
- 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.

Während ein traditioneller Compiler mit seinem Maschinencode jede zulässige Hardware unterstützen muss, kennt der JIT-Compiler die aktuelle Hardware genau und kann z.B. spezielle Befehlserweiterungen des Prozessors nutzen.

Aufgabe 5

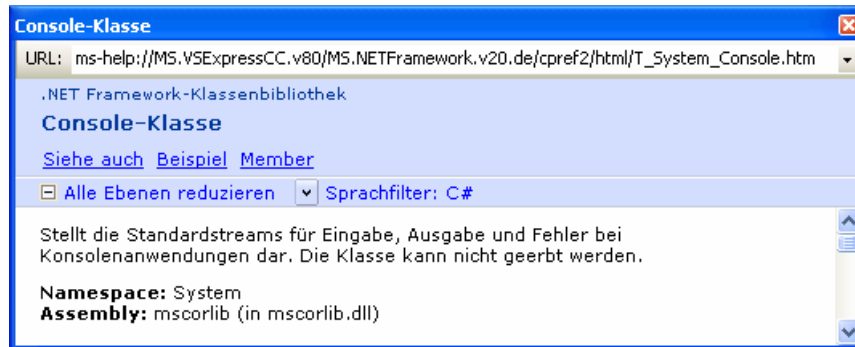
Namensräume und Assemblies sind streng genommen zwei voneinander *unabhängige* Organisationsstrukturen:

- Klassen, die zum selben Namensraum gehören, können in verschiedenen Assemblies implementiert sein.
- In einem Assembly können Klassen aus verschiedenen Namensräumen implementiert werden.

Eine gewisse Korrespondenz wird oft von den Softwareentwicklern hergestellt, z.B. bei der FCL:

Komponentenname	Version	Laufzeit	Pfad
System	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.dll
System.Configuration	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Configuration.dll
System.Configuration.Install	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Configuration.Install.dll
System.Data	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Data.dll
System.Data.OracleClient	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Data.OracleClient.dll
System.Data.SqlXml	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Data.SqlXml.dll
System.Deployment	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Deployment.dll
System.Design	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Design.dll
System.DirectoryServices	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.DirectoryServices.dll
System.DirectoryServices.Protocols	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.DirectoryServices.Protocols.dll
System.Drawing	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Drawing.dll
System.Drawing.Design	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Drawing.Design.dll
System.EnterpriseServices	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.EnterpriseServices.dll
System.Management	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Management.dll
System.Messaging	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Messaging.dll
System.Runtime.Remoting	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Runtime.Remoting.dll
System.Runtime.Serialization.Formatters.Soap	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Runtime.Serialization.Formatters.Soap.dll
System.Security	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Security.dll
System.ServiceProcess	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.ServiceProcess.dll
System.Transactions	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Transactions.dll
System.Web	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Web.dll
System.Web.Mobile	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Web.Mobile.dll
System.Web.RegularExpressions	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Web.RegularExpressions.dll
System.Web.Services	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Web.Services.dll
System.Windows.Forms	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Windows.Forms.dll
System.Xml	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Xml.dll

Allerdings befinden sich z.B. nicht alle Klassen aus dem Namensraum **System** auch im Assembly **System.dll**. Wie der SDK-Dokumentation zu entnehmen ist, befindet sich z.B. die in unseren Übungsprogrammen oft benutzte Klasse **System.Console** im Assembly **mscorlib.dll**, das auch ohne Referenz vom Compiler stets durchsucht wird:



Aufgabe 6

CLS	Microsoft hat unter dem Namen Common Language Specification 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 sicher gestellt.
IL	Ein .NET – Compiler übersetzt den Quellcode nicht in Maschinsprache, sondern in die Microsoft Intermediate Language (kurz: in IL-Code). Diesen Zwischencode übersetzt der JIT-Compiler in der CLR in Maschinencode.
COM	Diese traditionelle Komponententechnologie (Common Object Model) wird vom .NET – Framework abgelöst.
FCL	In der Framework Class Library befindet sich eine enorme Zahl von Klassen mit ausgereiften Lösungen für alle Routineaufgaben (z.B. graphische Benutzeroberflächen, Dateiverarbeitung, Netzwerkprogrammierung).

Kapitel 2 (Werkzeuge zum Entwickeln von C# - Programmen)

Aufgabe 2

Bei diesem „Programm“ lohnt sich die **using**-Anweisung für den Namensraum **System** ausnahmsweise nicht, weil der 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 **WriteLine()**-Aufruf 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 überflüssig.
 Der 3. Aufruf klappt: Statt **void** ist auch der Rückgabetyt **int** erlaubt.
 Der 4. Aufruf scheitert: Der Rückgabetyt **double** ist verboten.
 Der 5. Aufruf klappt: Diese Variante haben wir meist 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“.

Durch Klammerung wird dafür gesorgt, dass der zweite Plus-Operator zuerst ausgeführt wird und seine beiden numerischen Argumente addiert. Dieses 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 elementare 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 Flaecheninhalt betraegt: {0:f3}",
            pi * radius * radius);
    }
}

```

Aufgabe 3

Lösungsvorschlag:

```

class Prog {
    static void Main() {
        System.Console.WriteLine("Dies ist ein Zeichenketten-Literal:" +
            "\n    \"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 (= $6 \cdot 16 + 3$).

In der folgenden Anweisung wird der **char**-Variablen zeichen die Unicode-Escapesequenz für das Zeichen 'c' zugewiesen:

```
char zeichen = '\u0063';
```

Abschnitt 3.5 (Operatoren und Ausdrücke)

Aufgabe 1

Die Ausdrücke haben folgende Typen und Werte:

Ausdruck	Typ	Wert	Anmerkungen
$6/4*2.0$	double	2.0	
$(\text{int})6/4.0*3$	double	4.5	Der Typumwandlungsoperator hat die höchste Priorität und bezieht sich daher (ohne Wirkung) auf die 6.
$3*5+8/3\%4*5$	int	25	Abarbeitung mit Zwischenergebnissen: $15 + 8/3\%4*5$ $15 + 2\%4*5$ $15 + 2*5$ $15 + 10$

Aufgabe 2

erg1 erhält den Wert 2, denn:

- $(i++ == j ? 7 : 8)$ hat den Wert 8, weil $2 \neq 3$ ist.
- $8 \% 2$ ergibt 2.

erg2 erhält den Wert 0, denn:

- Der Präinkrementoperator trifft auf die bereits vom Postinkrementoperator in der vorangehenden Zeile auf den Wert 3 erhöhte Variable **i** und setzt sie auf den Wert 4.
- Dies ist auch der Wert des Ausdrucks $++i$, 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

1a1 erhält den Wert **false**, denn der Operator $\wedge$ wird aufgrund seiner höheren Priorität vor dem Operator $\&\&$ ausgewertet.

1a2 erhält den Wert **true**, weil die runden Klammern dafür sorgen, dass der Operator $\wedge$ zuletzt ausgewertet wird.

1a3 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 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\PrimitivOB

Die Lösung ohne **break** und **continue** ist deutlich komplizierter, also in der Regel nicht sinnvoll.

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\DM2EuroS

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\FloP

Bei $i = 52$ erhält man letztmals das korrekte Vergleichsergebnis.

Aufgabe 5

Lösungsvorschläge mit den beiden Algorithmus-Varianten befinden sich in den Ordnern:

...\BspUeb\Elementare Sprachelemente\GGT.Diff

...\BspUeb\Elementare Sprachelemente\GGT.Mod

Kapitel 4 (Klassen und Objekte)

Aufgabe 1

- Falsch
- Stimmt
Die Methodenüberladung ist erlaubt und üblich.
- Falsch
Es darf überhaupt kein Rückgabetyt angegeben werden.
- Grober Unsinn
Mit der Datenkapselung wird verhindert, dass die *Methoden fremder Klassen* auf Instanzvariablen zugreifen. Es war nie daran gedacht, Objekte einer Klasse voreinander zu schützen. Objekte derselben Klasse führen Methoden derselben Klasse aus, und diese sind stets zugriffsberechtigt.
- Stimmt

Aufgabe 2

Ein **readonly** – Feld kann nach der Initialisierung nicht mehr verändert werden, auch nicht von klasseneigenen Methoden.

Eine *getonly* – Eigenschaft erlaubt nur den lesenden Zugriff. Ein zugrunde liegendes Feld kann jedoch von klasseneigenen Methoden jederzeit geändert werden. Methoden fremder Klassen sollte in der Regel der direkte Zugriff auf das Feld versperrt sein (Datenkapselung).

Aufgabe 3

In der Anweisung

```
return Anzahl1;
```

wird versehentlich die **set**-Methode der Eigenschaft `Anzahl1` erneut aufgerufen. 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\Einleitungsbeispiele\Bruch\GUI

Kapitel 5 (Weitere .NETte Typen)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\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.

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** (siehe Abschnitt 5.4.1) orientiert sich der Identitätsoperator allerdings *nicht* an den enthaltenen Adressen, sondern an den Inhalten der referenzierten **String**-Objekte.

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

```
...\BspUeb\Arrays\Lotto
```

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

```
...\BspUeb\Arrays\Eratosthenes
```

Aufgabe 5

Sie finden einen Lösungsvorschlag im Verzeichnis:

```
...\BspUeb\Arrays\FloatMatrix
```

Aufgabe 6

Sie finden einen Lösungsvorschlag im Verzeichnis:

```
...\BspUeb\Zeichenfolgen\PerZuf
```

Aufgabe 7

Sie finden einen Lösungsvorschlag im Verzeichnis:

```
...\BspUeb\Zeichenfolgen\StringUtil
```

Kapitel 6 (Vererbung und Polymorphie)

Aufgabe 1

In der Vater-Klasse 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

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Vererbung und Polymorphie\Polymorphie (Figuren)

Aufgabe 4

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 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 statischen Methoden kommt das späte Binden bzw. Überschreiben nicht in Frage.

Kapitel 7 (Interfaces)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Interface\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 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 Objekten aus anderen kompatiblen Klassen) verwaltet werden. Eine Klasse „erbt“ allerdings nichts von den Schnittstellen, sondern sie gibt Verpflichtungserklärungen ab und muss die entsprechenden Implementierungs-Leistungen erbringen.

Kapitel 8 (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\DM2Euro

Kapitel 9 (Ausnahmebehandlung)

Aufgabe 1

Lösungsvorschlag:

```
using System;

class NullRefExcept {
    static void Main() {
        int[] vek = new int[5];
        vek = null;
        vek[0] = 1;
    }
}
```

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Exceptions\DuaLog\ArgumentOutOfRangeException

Kapitel 10 (Attribute)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Attribute\MyAttribute

Kapitel 11 (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 überflüssig und sehr zeitintensiv. Also sollte die folgende Zeile entfernt werden:

```
sw.AutoFlush = true;
```

Kapitel 12 (Threads)

Aufgabe 2

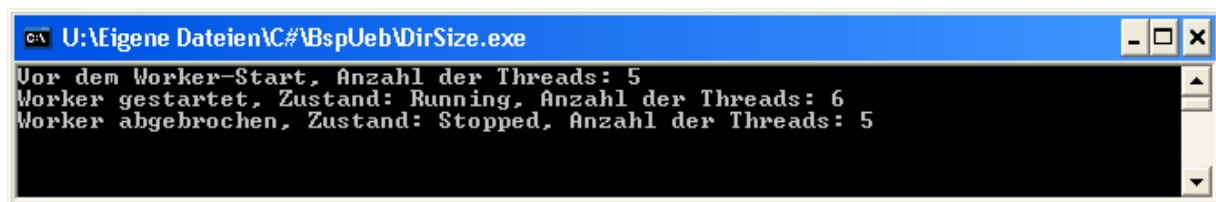
- Falsch
- Falsch
- Richtig

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Threads\DirSize

Das Beispielprogramm zeigt in der Konsole einige Informationen über die vorhandenen Threads an, z.B.:



```
U:\Eigene Dateien\C#\BspUeb\DirSize.exe
Vor dem Worker-Start, Anzahl der Threads: 5
Worker gestartet, Zustand: Running, Anzahl der Threads: 6
Worker abgebrochen, Zustand: Stopped, Anzahl der Threads: 5
```

Kapitel 13 (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 14 (Graphikausgabe 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+\Pfade und Zeichenbereiche\RoundButton

Kapitel 16 (Menüs)**Aufgabe 1**

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\WinForms\Menüs\Bearbeiten

Kapitel 17 (Dialogboxen)**Aufgabe 1 und Aufgabe 2**

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\WinForms\Dialogboxen\Weitersuchen

Kapitel 19 (Statuszeile und Symbolleisten)**Aufgabe 1**

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\WinForms\Symbolleisten\Erweitert

Kapitel 20 (Drucken)**Aufgabe 1**

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Drucken\BoundsChecker

Kapitel 21 (Zwischenablage, Ziehen und Ablegen)**Aufgabe 1**

Gemäß Abschnitt 21.4.2 lassen sich diverse Datenformate gleichzeitig in der Zwischenablage unterbringen, wobei es für die Zusammenstellung keine Restriktionen gibt.

Kapitel 24 (Variable Ansichten)**Aufgabe 1**

Sie finden einen (sehr elementaren) Lösungsvorschlag im Verzeichnis:

...\BspUeb\WinForms\TreeView\Datenerfassung

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>
- Balzert, H. (1999). *Lehrbuch der Objektmodellierung: Analyse und Entwurf*. Heidelberg: Spektrum.
- Bayer, J. (2006). *Das C# 2006 Codebook*. München: Addison-Wesley
- 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.icsharpcode.net/TechNotes/SharpDevelopCodingStyle03.pdf>
- Kühnel, A. (2006). *Visual C# 2005*. Bonn: Galileo Press. Auch als OpenBook verfügbar: http://www.galileocomputing.de/openbook/visual_csharp/
- Lammarsch, J. & Lammarsch, M. (2004). *C#*. Hannover: RRZN.
- Liberty, J. (2002). *Programming C#* (2nd ed.). Sebastopol, CA: O'Reilly.
- Louis, D. & Strasser, S. (2002). *C# in 21 Tagen*. München: Markt + Technik
- MacBeth, G. S. (2004). *C#. Programmer's Handbook*. New York: Springer.
- Microsoft Inc. (2007). *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. (2001). *Sprechen Sie Java. Eine Einführung in das systematische Programmieren*. Heidelberg: dpunkt.
- Mössenböck, H. (2003). *Softwareentwicklung mit C#. Eine kompakter Lehrgang*. Heidelberg: dpunkt.
- Petzold, C. (2002). *Windows-Programmierung mit C#*. Unterschleißheim: Microsoft Press.
- Patwardhan, N., Siever, E. & Spainhour, S. (2002). *Perl in a nutshell* (2nd ed.). Beijing: O'Reilly.
- 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.
- Spurgeon, C. E. (2000). *Ethernet. The Definitive Guide*. Sebastopol, CA: O'Reilly.
- Troelsen, A. (2002). *C# und die .NET-Plattform*. Bonn: mitp-Verlag

Stichwortregister

@

@ 287

+

+ -Operator 56

2

24.2.3 Dateiverzeichnisbaum 505

A

Abfrage

Aktion 524

Auswahl 524

parametrisierte 541

Ablaufsteuerung 98

Abort() 300

abstract 204

AcceptButton 239, 400

AcceptTcpClient() 324

ACK-Bit 313

Acos() 163

ActiveForm 402

Add()

ListBox 381

AddBezier() 371

AddLines() 371

AddRange() 381

ListBox 381

Aggregation 151

Aggregieren 526

Aktionsabfragen 524

Aktualisierungsoperatoren 89

Aktualparameter 128, 132

Algorithmen 7

Alignment 345

AllowDrop 465

Alpha-Kanal 343

AnchorStyles 244

ANSI-Textdateien 283

Antialiasing 349, 364

Anweisungen 97

Anweisungsblöcke 98

Anwendungskonfigurationsdatei 538

Anzeigebereich 374

App.config 538

Append()

StringBuilder 185

Application 220, 269

ApplicationException 261, 263

Apply 404

ARGB-Farbmodell 343

ArgumentOutOfRangeException 262, 265

Arithmetische Operatoren 78

Arithmetischer Ausdruck 78

Array 171, 209

mehrdimensional 177

ArrayList 179, 336

ASP.NET 309

Assembler 11

Assembly 12, 14

AssemblyInfo.cs 268

Assoziativität 90

AttachDbFilename 537

Attribute 266

Aufzählungen 186

Ausdrücke 77

Ausdrucksanweisungen 97

Ausgabe

im Konsolenfenster 56

Ausgabeparameter 130

Ausgabetyt 26

Auswahlabfrage 524

Auswertungsfunktionen 526

Auswertungsreihenfolge 90

Auswertungsrichtung 90

AutoCheck 378

AutoFlush 282

AutoScaleSizeBaseSize 249

AutoScroll 357, 385

AutoSize 234

B

base-Konstrukturen 196, 198

Basisklasse 193

Baumansicht 502

Bedingte Anweisung 98

Befehlsschalter 237

runde 376

Befehlszeilenargumente 105

Benutzerinstanzen 538

Bezeichner 54

Beziehung 523

Bézier-Splines 351

Bildlaufleiste 357

Bildschirmkoordinaten 339

Binärdateien 277

binäre

Operatoren 78

BinaryFormatter 284

BinaryReader 278

BinaryWriter 278

binding 494

BindingNavigator 557

BindingSource 557

Bindung 494

Bitmap 369

Bitorientierte Operatoren 85

Bitweises UND 86

Block 67

Blockanweisung 67, 98

BMP 370

bool-Literale 71

BorderStyle 235

Boxing 170

break-Anweisung 112

BringToFront() 243

Brush 346

Brushes 347

Button 237

ButtonBase 377

Bytecode 14

C

CancelButton 400

CancelEventArgs 409

CanRead 273

CanSeek 273
 CanWrite 273
 CAS 18
 case-Marke 104
 casting 86
 Casting-Operator 87
 catch-Klausel 255
 CGI 318
 CGI.pm 320
 char-Literale 72
 CheckBox 377
 checked
 Anweisung 94
 Compileroption 94
 Operator 93
 Checked
 Menüitem 394
 CIL 14
 Clear()
 ListBox 381
 ClearType 364
 ClientSize 339
 Clipboard 459
 Clipboard ID 456
 Clipping 373
 Clone() 209
 Close() 273
 StreamWriter 282
 CloseAllFigures() 373
 CloseFigure() 372
 CLR 11, 17
 CLS 14, 77
 Code Access Security 18
 Color 343
 ColorDialog 344, 413
 Columns
 ListView 511
 COM 11, 15, 267, 483
 COM+ 484
 ComboBox 380
 CommandBuilder 546
 Common Gateway Interface 318
 Common Intermediate Language 14
 Common Language Runtime 11
 Common Language Specification 14
 Common Type System 119, 169, 195
 Strukturen 169
 CompareTo()
 String 183
 Compiler 12
 COM-Server 484
 ConfigurationManager 539
 ConnectionString 536
 Console 56
 const 69, 123
 ContainerControl 235
 ContentAlignment 235
 ContextMenu 395
 continue-Anweisung 112
 Control 235
 ControlCollection 235
 controls 215
 Controls 235
 Convert 74
 Copy() 288
 Cos() 163
 CPU 11
 Create() 287
 WebRequest 314
 CreateDirectory() 289

CreateText() 287
 csc 13, 25
 csc.rsp 45
 ctor 139
 CTS 119, 169, 195
 Strukturen 169
 CurrentThread() 296

D

dangling else 101
 DashCap 344, 345
 DashStyle 344, 345
 DataColumn 544
 DataDirectory 537
 DataFormats 456, 468
 DataObject 461
 DataReader 549
 DataRelation 545
 DataRow 544
 DataSet-Designer 554
 DataTable 544
 Dateiauswahldialog 414
 Datenbankmanagementsystem 521
 Datenkapselung 117
 Datenquelle 550
 Datenströme 271
 Datentypen
 elementare 61
 DateTime 115
 DBMS 521
 DCOM 484, 485
 Deadlocks 304
 decimal 96
 declaration space 124
 Definitionstabellen 15
 Deklarationsbereich 67, 124
 Delegates 224
 Delete() 288
 Directory 289
 DeleteValue() 420
 DeMorgan 92
 Denormalisierte
 Gleitkommadarstellung 64
 deprecated 303
 Dialogboxen
 modal 399
 nicht-modal 406
 DialogResult 403
 Directory 288, 505
 DirectoryInfo 288
 DirectoryInfo 515
 Direktive
 using 19
 Dispose() 274, 334
 DLL-Hölle 15
 DllImport 481
 Dns 323
 DNS 323
 Dock 245, 497
 Document Explorer 31
 Dokumentationskommentar 54
 Domain Name System 323
 DomainUpDown 384
 do-Schleife 111
 Dots Per Inch 361
 Double
 Epsilon 97
 IsNaN() 95
 IsNegativeInfinity() 95

- IsPositiveInfinity() 95
- MaxValue 95
- Download
 - Dateien 316
- DownloadData() 321
- DownloadFile() 316
- DPI 361
- DpiY 361
- Drag & Drop 465
- DrawArc() 349
- DrawBezier() 351
- DrawBeziers() 351
- DrawClosedCurve() 350
- DrawCurve() 350
- DrawEllipse() 349
- DrawImage() 369
- DrawLine() 348
- DrawLines() 348
- DrawPath() 373
- DrawPie() 350
- DrawPolygon() 350
- DrawRectangle() 348
- DrawString() 358, 447
- DriveInfo 506
- DriveType 506
- Drucken 433
- Durchfall 104
- Dynamische Bindung 203

E

- Eigenschaft
 - Syntaxdiagramm 52
- Eigenschaften 3
 - klassenbezogene 145
- Eingabefelder 241
- Eingabefokus 239
- Einschränkende Konvertierung 87
- einstellige
 - Operatoren 78
- Elementare Datentypen 61
- else-Klausel 99
- Enabled
 - MenuItem 394
- Encoding 283
- EndCap 344
- Endlosschleife 112
- Enter()
 - Monitor 299
- Entwurfsmuster
 - Ereignisbehandlung 230
- Enumerationen 186
- Environment 161
- Epsilon 97
- Eratosthenes 189
- Ereignisse 227
- ERRORLEVEL 254
- Erweiternde Konvertierungen 86
- Escape-Sequenzen 57, 72
- Euklidischer Algorithmus 7, 114
- event 231
- Exception-Handler 255
- ExecuteNonQuery() 540
- ExecuteReader() 540
- ExecuteScalar() 540
- EXIF 370
- Exists() 288
 - Directory 289
- Exit 161
- Exit() 254

- Klasse Environment 254
- Monitor 299
- Exitcode 254
- Exklusives logisches ODER 85
- extern 481

F

- Families 356
- Farbauswahl
 - Standarddialog 343
- Farbauswahl-Standarddialog 413
- Farben 343
- FCL 11, 18
- Felder
 - klassenbezogene 144
 - verdeckte 201
- File 287
- FileAccess 276
- FileInfo 287, 515
- FileMode 276
- FileShare 277
- FileStream 272, 275
- FileSystemWatcher 289
- Fill()
 - DataAdapter 542
- FillClosedCurve() 352
- FillEllipse() 352
- FillPath() 373
- FillPie() 352
- FillPolygon() 352
- FillRectangle() 352
- FillSchema() 543
- Filter
 - SaveFileDialog 414
- finally-Block 255
- Find 410
- FixedDialog 400
- Flackern 337
- Flush() 273
- Focus() 331
- Font 354
- FontDialog 357, 411
- FontFamily 356, 362
- FontStyle 355, 378
- foreach 107
- Formalparameter 128
- Formatierte Ausgabe
 - im Konsolenfenster 56
- FormBorderStyle 238
- Formular 220
 - HTML 319
- Formulardesigner 36, 156
- Formulare 216
- for-Schleife 108
- Frame 310
- Framework Class Library 11, 18
- Fremdschlüssel 523
- Fremdschlüssel-Restriktion 523
- FromArgb() 343
- FROM-Klausel 525
- FTP-Protokoll 318
- Füllmodus 353
- Funktions-Pointer 224

G

- GAC 75

Ganzzahlarithmetik 78
 Ganzzahliliterale 69
 Garbage Collector 140
 GDI+ 333
 GenericTypographic 364
 Gerätekontext 333
 GET
 CGI-Parameter 320
 GetCurrentDirectory() 289
 GetCustomAttributes() 267
 GetDataObject() 463
 GetDataPresent() 463, 468
 GetDirectories() 289, 505, 515
 GetDriveType() 506
 GetExecutingAssembly() 476
 GetFiles() 289, 515
 GetHeight() 362
 GetHostEntry() 323
 GetLastWin32Error() 483
 GetLength()
 Array 177
 GetLogicalDrives() 505
 GetObject() 476
 GetResponse() 314
 GetResponseStream 314
 GetString() 476
 GetType() 170
 GetValue()
 RegistryKey 420
 Geviertgröße 355
 GIF 370
 Gleitkommaarithmetik 78
 Gleitkommadarstellung 63
 Gleitkommaliterale 70
 Global Assembly Cache 28
 Globale Variablen 61
 goto 104
 Graphics 333
 Graphics Device Interface 333
 GraphicsPath 371
 GROUP BY 526
 GroupBox 379
 GUI 215

H

Hallo 23
 HatchBrush 346
 Heap 124, 137
 Height 221
 Host-Name 323
 HTTP - Protokoll 316
 HttpUtility 321
 HttpWebRequest 314

I

ia64 12
 ICloneable 209
 ICMP 312
 IComparable 209, 517
 IComparer 516
 IDataObject 463, 466
 IDisposable 274
 IDL 15
 IEEE-754 63
 if-Anweisung 98
 IfModifiedSince 317
 IFormatter 284

IL 12
 ILDasm.exe 493
 ImageList 429, 507
 Indexer 187
 IndexOf()
 String 184
 IndexOutOfRangeException 174, 254
 Inflate() 341
 information hiding 117
 Initialisierung 59, 66
 Initialisierungslisten 176
 InitializeComponent() 159
 Inkonsistenz 521
 inlining 143
 INNER JOIN 525
 Innere Klassen 153
 In-Process-Server 484
 InputBox() 75
 Insert()
 StringBuilder 185
 Instanzvariablen 61, 122
 Integraler Typ 86
 Integrated Security 537
 Interaction 75
 Interface 209
 Interface Definition Language 15
 interner String-Pool 181
 Internet Control Message Protocol 312
 Interrupt() 303
 Invalidate() 337
 InvalidConstraintException 545
 IN-Vergleichsoperator (SQL) 525
 IP-Adresse 323
 IPHostEntry 323
 IsBackground 295
 ISerializable 286
 IsNaN() 95
 IsNegativeInfinity() 95
 is-Operator 204
 IsPositiveInfinity() 95
 IsStyleAvailable() 356
 Items
 ListBox 381
 ListView 512

J

jagged arrays 178
 Java 11, 14
 JIT-Compiler 17
 Join() 303
 JPEG 370

K

Kapselung 117
 Kardinale
 Splines 350
 Kindfenster 232
 Klammern 90
 Klasse
 Syntaxdiagramm 51
 Klassen 117
 innere 153
 statische 147
 klassenbezogene
 Eigenschaften 145
 Klassenmethode 25
 Klassenvariablen 61

- Kollektionen 109
- Kombinationsfeld 380
- Kommentar 54
- Komponentenfach 557
- Konditionaloperator 89
- Konfigurationsdatei 538
- Konstanten 68, 123
- Konstruktoren 137
 - statische 146
- Kontextmenü 395
- Kontrollkästchen 377
- Kontrollstrukturen 98
- Konvertierung
 - einschränkende 87
 - erweiternde 86

L

- Label 233
- Leere Anweisung 97
- Left 235
- Length 174, 183
 - Stream 273
- lexikographische Priorität 183
- LIKE-Vergleichsoperator (SQL) 525
- LinearGradientBrush 347
- LineCap 345
- LineJoin 348
- Links-Shift-Operator 86
- ListBox 380
- Listenfeld 380
- ListView 510
- ListViewItem 510
- ListViewSubItem 510
- Literale 69
- lock 297
- Logische Operatoren 84
- Logisches ODER 84
- Logisches UND 84
- Lokale Variablen 61

M

- MAC-Adresse 310
- Main() 8, 49
- MainMenu 391
- managed code 21
- Manifest 15
- Marshaler 481
- Maschinencode 11
- Master-Detail -
 - Beziehung 523
- Math-Klasse 80
- Matrix 369
- MaxValue 95
- MeasureCharacterRanges() 365
- MeasureString() 362, 363
- Mehrfachvererbung 197, 210
- Member 4
- memory leaks 140
- Menü 391
- MenuItem 391
- MenuItemCollection 391
- Menüzeile 391
- MessageBox 45
- Metadaten 15
- Methode
 - Syntaxdiagramm 52
- Methoden 126

- Aufruf 132
- Definition 127
- Modifikatoren 131
 - rekursive 148
 - Rückgabewert 131
 - statische 145
- Microsoft Intermediate Language 12
- Microsoft SQL Server 2005 Express Edition 526
- Modale Dialoge 399
- Modifikatoren
 - bei Methoden 131
- module 27
- Module 16
- Modulo 79
- Monitor 299
- Mono 436
- Mono-Projekt 20
- MouseDown 338
- MouseEventArgs 387
- MouseLeave 429
- MouseWheel 387
- Move() 288
- MoveFile() 483
- mscorlib.dll 27
- MSIL 12
- Müllsammler 140
- MulticastDelegate 225
- Multiline 242
- Multitasking 292
- Multithreading 292

N

- Namen 54
- Namensräume 18
- namespace 18
- NaN 95
- NAS 537
- Nebeneffekt 78
- Nebeneffekte 80, 85
- Negation 84
- Network Attached Storage 537
- NetworkStream 324, 326
- Netzwerkprogrammierung 309
- new-Operator 136, 137
- Next() 175
- noconfig 45
- NonSerialized 284
- Normalisieren 523
- Normalisierte
 - Gleitkommandarstellung 63
- null 125, 135
- NULL (SQL) 525
- Number
 - Klasse 254
- NumericUpDown 383

O

- Objektserialisierung 283
- Obsolete
 - Attribut 266
- ObsoleteAttribute 266
- ODBC 534
- Öffnungsmodus 276
- Offset()
 - Point 339
 - Rectangle 341
- OLE 416

OLEDB 534
 OnBeforeExpand() 508
 OnClosed() 421
 OnClosing 418
 OnMenuComplete() 428
 OnMenuStart() 428
 OnPaint() 335
 OpenFileDialog 416
 OpenType 354
 operator+ 150
 Operatoren 77
 Arithmetische 78
 bitorientierte 85
 logische 84
 überladen 149
 vergleichende 81
 Operatorentabelle 563
 Optionale Parameter 77
 Optionsfeld 377
 OracleClient 534
 ORDER BY 526
 Ordneransicht 505
 OSI-Modell 310
 out 130
 -Compiler-Option 26
 Out-Of-Process-Server 484
 override 203
 Owner 408

P

PageScale 366
 PageSettings 434
 PageSetupDialog 451
 PageUnit 366
 Paint 335
 PaintEventArgs 336
 Panel 236, 357, 379
 Parameter
 optionale 77
 Parametrisierte Abfragen 541
 params 130
 Parent 236
 partial 158
 Peek() 282
 Pen 344
 Pens 345
 PInvoke 481
 Platform Invocation Services 481
 PNG 370
 Point 339
 PointToClient() 339, 466
 PointToScreen() 339
 Polymorphie 194, 202
 Popup 394
 Port 311
 Position
 Stream 272
 POST
 CGI-Parameter 322
 Postinkrement bzw. -dekrement 79
 Postscript 355
 Potenzfunktion 80
 Pow() 80
 Präinkrement bzw. -dekrement 79
 Preemptives Zeitscheibenverfahren 302
 Primärer Thread 294
 Primärschlüssel 522, 544
 PrimaryKey 544
 PrintController 438

PrintDialog 440
 PrintDocument 435
 PrinterSettings 433
 PrintFormat 447
 PrintPage
 -Handler 445
 PrintPageEventArgs 445
 PrintPreviewControl 450
 PrintPreviewDialog 449
 Priorität 90
 Prioritäten 302
 Priority
 Thread 302
 private 122
 Process 515
 Projektmappen-Explorer 36, 156
 properties 142
 Properties 3
 protected 197
 Provider 534
 Pseudozufallszahlengenerator 174
 Puffer
 StreamWriter 282
 Pulse()
 Monitor 299
 PulseAll() 300
 Punktoperator 125, 132

Q

QUERY_STRING 320
 QueueUserWorkItem() 329, 515

R

Race Condition 297
 RadioButton 377
 RadioCheck
 MenuItem 394
 Random 146, 175
 Rank 177
 Rastergrafik 369
 RCW 485
 Read() 272
 SqlDataReader 550
 ReadByte() 272
 ReadKey() 290
 ReadLine() 74
 readonly 123
 Read-Only
 Eigenschaft 143
 ReadTimeout 328
 Records 120
 Rectangle 341
 RectangleF 370
 Redistributable 12
 Redistributable-Paket 11
 Redundanz 521
 reference
 -Compiler-Option 27, 45, 75
 referentielle Integrität 523
 Referenzparameter 129
 Referenztabellen 15
 Referenzvariablen 135
 Refresh() 337
 Region 374
 Registry 419
 RegistryKey 420
 regsvr32.exe 485, 489

Rekursive Methoden 148
Relationale Datenbanken 521
Remote Server 485
Remove()
 StringBuilder 185
Remove()
 ListBox 381
Replace()
 StringBuilder 186
Replace()
 String 184
Reservierte Wörter 55
ResetAbort() 301
resgen.exe 475
ResizeRedraw 340
ResourceManager 476
RESOURCES-Datei 475
ResourceWriter 475
Response-Datei 27
Ressourcen 473
Resume() 303
RESX-Datei 473
ResXResourceReader 475
ResXResourceWriter 474
return 106, 131
return-Anweisung 131
RichTextBox 388
Rollbalken 385
Rollen 531
Round-Robin 302
Router 311
RTF-Format 388
Rückgabewert 131
Run() 220
Runtime Callable Wrapper 485

S

SAN 537
Satellitenassembly 473
SaveFileDialog 414
Scale() 250
ScaleTransform() 368
Schaltfläche 237
 runde 376
Scheduler 302
Schleifen 107
Schließen
 von Datenströmen 273
Schnittstelle 209
Schriftarten-
 Standarddialog 411
Schriftschnitte 355
ScrollBar 385
ScrollBars 400
Scroll-Rad 386
SDK 31
SDK-Dokumentation 175
sealed 206
Seek() 273
SeekOrigin 273
Seitenansicht 449
Seiteneinrichtung 451
Seitentransformation 366
Sekundäre Fenster 399
Sekundäre Threads 294
SELECT-Befehl (SQL) 524
SelectedIndex 381
SelectedItem 382
Serialisieren
 von Objekten 283
Serializable 284
SetClip() 373
SetCurrentDirectory() 289
SetData() 461
SetDataObject() 460
SetLastError 483
SetLastWriteTime() 288
SetText() 459
SetValue()
 RegistryKey 420
Shortcut 392
Show() 220
ShowApply 412
ShowDialog() 403
ShowInTaskbar 408
Shutdown() 325
Sieb des Eratosthenes 189
Signatur 199
Sin() 163
Single-Threaded Apartment 267
Size 339
Skalarprodukt 162
Sleep() 296
SmoothingMode 349
SOAP 309
SoapFormatter 286
Socket 311, 323
Software Development Kit 31
SolidBrush 346
Sort() 209
Sorted
 ListBox 381
Späte Bindung 203
Speicherlöcher 140
Spin-Control 382
Splines
 Bézier 351
 kardinale 350
Split() 191
SplitContainer 500
Splitter 497
SQL 524
 SELECT-Befehl 524
SQL Server 2005 Express Edition 526
SQL Server Browser Dienst 533
SqlClient 534
SqlCommand 540, 547
SqlCommandBuilder 547
SqlConnection 536, 547
SqlDataAdapter 542, 547
SqlDataReader 549
SqlParameter 542
Sqrt() 162
STA 267
Stack 124
Stack-Überlauf 149
Standarddialog
 Drucken 440, 449
 Farbauswahl 343, 413
 Öffnen 416
 Schriftarten 411
 Schriftartenwahl 357
 Seiteneinrichtung 451
 Sichern 414
Standarddialoge 411
Standardkonstruktor 137
Standardressourcendatei 477
Standardschaltfläche 239
StartCap 344

startfähige Klasse 49
 Startklasse 8
 StartPosition 402
 StartsWith()
 String 184
 starvation 302
 STAThread 267, 416, 418, 459, 465
 static 143
 Statische
 Felder 144
 Klassen 147
 Konstruktoren 146
 Methoden 145
 Statische Methoden
 Verdecken 201
 StatusBarPanel 425
 Statuszeile 423
 Steuerelemente 215
 Storage Area Network 537
 Stream 272
 streams 271
 String 181, 209
 Methoden 182
 StringBuilder 185
 StringFormat 360
 String-Pool 181
 Strings
 vergleichen 183
 verketteten 182
 Ströme 271
 struct 165
 Structured Query Language 524
 Struktogramm 149
 Strukturansicht 502
 Strukturen 120, 165
 Strukturiertes Programmieren 119
 Substring()
 String 183
 Suspend() 303
 switch-Anweisung 103
 Symbolleisten 429
 Synchronisierter Block 298
 Syntaxdiagramm 50
 System.IO 271
 SystemBrushes 347
 SystemColors 344
 Systemfarben 344
 SystemPens 345

T

TabIndex 239
 TableAdapter 551
 TableLayoutPanel 247
 TableMappings 542
 target
 -Compiler-Option 26
 Tastatur
 Zugriffstaste 392
 TCP 311
 TCP/IP
 SQL-Server 533
 TcpClient 326
 TCP-Flags 313
 TcpListener 324
 Telnet 326
 TextChanged 242, 427
 Textdateien 277, 280
 TextReader 280
 TextRenderingHint 364

TextureBrush 346
 TextWriter 280
 this 125, 133, 139, 142
 Indexer 188
 Thread 293
 ThreadAbortException 300
 Threadpool 292
 ThreadPool 515
 ThreadPriority 302
 Threads 292
 throw 262
 TIFF 370
 Timeout
 NetworkStream 327
 Timer
 Threading 305
 WinForms 387
 Time-To-Live 313
 TlbImp.exe 486
 ToChar() 85
 ToInt32() 74
 ToLower() 184
 ToolBar 429
 ToolBarButton 429
 ToolTips 427
 Top 235
 ToString() 119
 StringBuilder 186
 ToUpper() 184
 TranslateTransform() 368, 369
 Transmission Control Protocol 311
 Transparentfarbe 240, 430
 TreeNode 502
 TreeNodeCollection 502
 TreeNode-Editor 503
 TreeView 502
 Trimming 361
 TrueType 354
 try-Anweisung 255
 T-SQL 524
 TTL 313
 Type 268
 Typetest-Operator 204
 Typumwandlung 86

U

Überladen
 von Methoden 133
 von Operatoren 149
 Überladung 182, 200
 Überlauf 93
 Übernehmen 404
 Überschreiben 202
 UDP 311
 UML 5
 unäre
 Operatoren 78
 Unboxing 170
 undefinierte Werte 95
 Unendlich 94
 UNICODE 416
 Unicode-Escape-Sequenzen 72
 Unicode-Zeichensatz 55
 Unified Modeling Language 5
 Uniform Resource Identifier 314
 unmanaged code 481
 Unterlauf 96
 Unterprogrammtechnik 119
 Update() 337

- DataAdapter 543
- UpDown-Steuerelement 382
- UploadValues 322
- URI 314
- URL 314
- UrlDecode() 321
- UrlEncode() 321
- URL-Kodierung 320
- User Datagram Protocol 311
- User Instance 538
- user32.dll 481
- UserAgent 314
- using
 - Direktive 19
- using-
 - Anweisung 274
- UTF8-Kodierung 282

V

- Variablen 58
 - globale 61
 - lokale 61
- Variablendeklaration 66, 97
- Verbindungsloser Datenzugriff 534
- Verbindungszeichenfolge 536
- Verbundanweisung 67, 98
- Verdecken 199
- Verdeckte Felder 201
- Vererbung 193
- Verfügbarkeit 154
- Vergleich 81
- Vergleichen
 - von Strings 183
- Vergleichsoperatoren 81
- Verketteten
 - von Strings 182
- Verkettete Liste 187
- versiegelt 206
- Versiegelte
 - Klassen 206
 - Methoden 205
- Verweis 46
- virtual 203
- Visible 220
- VisibleClipBounds 367
- Visual Basic 75
- Visual C# 2005 Express Edition 39, 106, 155, 268
- VScrollBar 385

W

- Wahrheitstafeln 84

- Wait()
 - Monitor 299
- WaitSleepJoin-
 - Zustand eines Threads 299
- WebBrowser 315, 329
- WebClient 316
- Webdienste 309
- WebResponse 314
- Wertparameter 128
- Werttypen 60
- Wertzuweisung 67
- WHERE-Klausel 525
- while-Schleife 110
- widgets 215
- Width 221
- Wiederholungsanweisungen 107
- Windows
 - Registry 419
- winexe 47
- WinForms 215
- WM_QUIT 222
- WordWrap 400
- Write() 56, 273
- WriteByte() 273
- WriteLine() 56
- Write-Only
 - Eigenschaft 143

X

- x64 12
- x86 12
- XML 473

Z

- Z-Anordnung 497
- Zeichenbereich 373
- Zeichenketten 181
- Zeichenkettenliterale 72
- Zeilenumbruch 191
- Zeitscheibenverfahren 302
- Ziehen und Ablegen 465
- ZoomFactor 400
- Zufallszahlen 146
- Zugriffsschutz 117, 154
- Zugriffstaste 392
- Zusammengesetzte
 - Anweisung 98
- Zuweisungsoperator 88
- zweistellige
 - Operatoren 78