Armin Hanisch

Go To C#



- Der Weg zum Profi
- Tipps für Umsteiger
- Arbeitsbuch und Referenz

ADDISON-WESLEY

Pearson Education

ADDISON-WESLEY

Armin Hanisch

Go To C#

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Ein Titeldatensatz für diese Publikation ist
bei Der Deutschen Bibliothek erhältlich.

Die Informationen in diesem Buch werden ohne Rücksicht auf einen
eventuellen Patentschutz veröffentlicht.
Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.
Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter
Sorgfalt vorgegangen. Trotzdem können Fehler nicht vollständig ausgeschlossen werden.
Verlag, Herausgeber und Autoren können für fehlerhafte Angaben
und deren Folgen weder eine juristische Verantwortung noch
irgendeine Haftung übernehmen.
Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und
Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der
Speicherung in elektronischen Medien.
Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten
ist nicht zulässig.

Fast alle Hardware- und Softwarebezeichnungen, die in diesem Buch erwähnt werden,
sind gleichzeitig auch eingetragene Warenzeichen oder sollten als solche betrachtet werden.

Umwelthinweis:
Dieses Buch wurde auf chlorfrei gebleichtem Papier gedruckt.
Die Einschrumpffolie – zum Schutz vor Verschmutzung – ist aus umweltverträglichem
und recyclingfähigem PE-Material.

10 9 8 7 6 5 4 3 2 1

05 04 03 02

ISBN 3-8273-1932-3

© 2002 by Addison Wesley Verlag,
ein Imprint der Pearson Education Deutschland GmbH
Martin-Kollar-Straße 10–12, D-81829 München/Germany
Alle Rechte vorbehalten
Einbandgestaltung: Barbara Thoben, Köln
Lektorat: Christiane Auf, cauf@pearson.de
Korrektorat: Susanne Franz, Ottobrunn
Herstellung: Monika Weiher, mweiher@pearson.de
Satz: reemers publishing services gmbh, Krefeld, www.reemers.de
Druck und Verarbeitung: Bercker Graphischer Betrieb, Kevelaer
Printed in Germany

Inhaltsverzeichnis

Vorwort 17

Basics – Teil I 19

1 Einführung 21

1.1 Noch eine neue Programmiersprache? 21
1.2 An wen richtet sich das Buch? 21
1.2.1 Leserkenntnisse 22
1.2.2 Systemvoraussetzungen 22
1.3 Konventionen 23
1.4 Gliederung 24
1.5 Die CD zum Buch 25

2 Das .NET-Framework 27

2.1 Was ist .NET? 27
2.1.1 Die Bausteine von .NET 27
2.1.2 Die Rolle von Windows 28
2.2 Das .NET-Framework im Überblick 28
2.2.1 Die Vorteile des .NET-Frameworks 29
2.2.2 Das Common Type System 32
2.2.3 Die Common Language Specification (CLS) 33
2.2.4 Die .NET-Laufzeitumgebung (CLR) 34
2.3 .NET aus Anwendungssicht 35
2.3.1 Übersetzung 35
2.3.2 Laden und Ausführung 36
2.3.3 Ressourcenverwaltung 37
2.4 .NET aus Entwicklersicht 37
2.4.1 Ein klarer Schnitt? 38
2.4.2 Die Bequemlichkeit der Infrastruktur 39
2.4.3 Die Namensräume 40
2.4.4 Wo anfangen? 40
2.5 Zusammenfassung 43

3 C# Sightseeing Tour 45

3.1 Das Problem 45
3.1.1 Ein Werkzeug für .NET 46
3.2 Designziele von C# 47
3.2.1 Produktivität 47
3.2.2 Stabiler und sicherer Code 47
3.2.3 Praxistauglichkeit 48
3.2.4 Modularität 49
3.2.5 Nutzung aller .NET-Möglichkeiten 49

3.2.6 Echte Objektorientierung 50
3.2.7 Komponentenorientierte Entwicklung 50
3.3 C# im Überblick 52
3.3.1 Wofür C#? 52
3.3.2 Am Anfang war »Hallo, Welt« 54
3.3.3 Grundprinzipien von C#-Programmen 55
3.3.4 Datentypen in C# 57
3.3.5 Kontrollstrukturen & Operatoren 59
3.3.6 Klassen & Interfaces 59
3.3.7 Attribute & Reflection 60
3.3.8 Strukturierte Fehlerbehandlung 60
3.3.9 Interoperabilität mit COM & Win32 61
3.3.10 Bedingte Kompilierung / Präprozessor 62
3.3.11 Namespaces 62
3.3.12 Formatierte Stringausgabe in C# 63
3.4 Zusammenfassung 64

4 Namensräume 65

4.1 Grundlagen 65
4.2 Definition eines Namensraums 66
4.2.1 Hierarchische Namensräume 67
4.3 Nutzen eines Namensraums mit using 68
4.4 Namenskonflikte beim Import mit using 69
4.4.1 Aliase 70
4.5 Die wichtigsten Namensräume 71
4.6 Zusammenfassung 72
4.7 Übungen 73

5 Daten, Typen, Variablen 75

5.1 Wert- und Referenztypen 75
5.2 Alles ist ein Objekt 75
5.2.1 Boxing und Unboxing 76
5.2.2 Boxing und Literale 76
5.3 Einfache Datentypen 77
5.3.1 Bool 78
5.3.2 Char 79
5.3.3 String 79
5.3.4 Nummerische Literale 80
5.4 Bezeichner 81
5.5 Konstanten 82
5.5.1 Echte Konstanten 82
5.5.2 Konstantenausdrücke 83
5.5.3 Read-Only Variablen 84
5.6 Variablen 85
5.6.1 Deklaration 85
5.6.2 Initialisierung von Variablen 86

5.6.3 Variablenkategorien in C# 86
5.6.4 Standardwerte 91
5.7 Zusammenfassung 92
5.8 Übungen 93

6 OOP-Grundlagen 95

6.1 OOP-Konzepte 96
6.1.1 Abstraktion 97
6.1.2 Kapselung 97
6.1.3 Kohäsion und Kopplung 98
6.1.4 Vererbung 99
6.1.5 Polymorphie 100
6.2 Terminologie 100
6.2.1 Klasse (class) 100
6.2.2 Feld / Attribut (field, member variable) 101
6.2.3 Eigenschaften (property) 101
6.2.4 Methode (method) 102
6.2.5 Instanziierung (instantiation, creation) 103
6.2.6 Konstruktoren (constructor) 104
6.2.7 Objekt (object, class instance) 105
6.2.8 Freigabe / Destruktoren (destructor) 105
6.2.9 Alles ist ein Objekt 106
6.3 Zusammenfassung 107
6.4 Übungen 108

7 Klassen in C# 109

7.1 Klassendefinition 109
7.2 Zugriff auf Klassen und Mitglieder 109
7.2.1 Zugriff auf Klassen 110
7.2.2 Zugriff auf Klassenmitglieder 111
7.3 Methoden 112
7.3.1 Definition 113
7.3.2 Parameterübergabe 113
7.3.3 Überladen von Methoden 117
7.3.4 Variable Anzahl von Parametern 119
7.4 Instanzen erzeugen 120
7.5 Konstruktoren und Destruktoren 121
7.5.1 Konstruktoren 121
7.5.2 Destruktoren 123
7.6 Eigenschaften 125
7.6.1 Deklaration von Eigenschaften 127
7.6.2 OOP-Vorteile von C#-Eigenschaften 128
7.6.3 Der spezielle Bezeichner value 130
7.6.4 Nur-Lese-Eigenschaften 130
7.7 Statische Klassenmitglieder 131
7.7.1 Das Schlüsselwort static 132

7.7.2 Zugriff auf statische Klassenmitglieder 132
7.7.3 Statische Konstruktoren 134
7.8 Übungen 136

8 Anweisungen & Operatoren 137

8.1 Anweisungen 137
8.1.1 Schlüsselwörter 138
8.2 Ablaufsteuerung 139
8.2.1 Aus der Gruft – GOTO 139
8.2.2 Verzweigungen – IF / ELSE 140
8.2.3 Bedingte Schleifen – DO & WHILE 142
8.2.4 Festgelegte Schleifen – FOR 143
8.2.5 Schleifen in Auflistungen – FOREACH 144
8.2.6 Raus hier – BREAK & CONTINUE 144
8.2.7 Mehrfachauswahl – SWITCH 145
8.3 Synchronisation, Überlaufprüfung 146
8.3.1 lock 146
8.3.2 checked, unchecked 146
8.4 Operatoren 146
8.4.1 Arithmetische Operatoren 147
8.4.2 Logische Operatoren 148
8.4.3 Relationale Operatoren 151
8.4.4 Zuweisungen 152
8.4.5 Typ-Operatoren 152
8.4.6 Der Bedingungsoperator 155
8.4.7 Zugriffsoperatoren 156
8.5 Übungen 156

Going professional – Teil II 157

9 Abzählen in C# – Enums 159

9.1 Die Grundlagen 159
9.2 Enums implementieren 159
9.3 Basisdatentypen für Aufzählungen 162
9.3.1 Aufzählungen als Bitflags 163
9.4 Ausgabe von Aufzählungstypen 164
9.5 Zusammenfassung 167
9.6 Übungen 168

10 Structs und Arrays 169

10.1 Arrays 169
10.1.1 Deklaration 169
10.1.2 Erzeugung und Initialisierung 172
10.1.3 Konvertierungen bei Arrays 173
10.1.4 Die Klasse System.Array 174

10.2 Structs 175
10.2.1 Mehr als nur Daten 175
10.2.2 Deklaration und Initialisierung von structs 176
10.2.3 Verschachtelte structs 179
10.2.4 Layout von structs 180
10.3 Zusammenfassung 181
10.4 Übungen 181

11 Vererbung 183

11.1 Einfache Vererbung 183
11.1.1 Zugriff über die Basisklasse 187
11.1.2 Zugriff auf Mitglieder der Basisklasse 187
11.1.3 Zugriff auf Konstruktoren der Basisklasse 188
11.2 Steuern der Vererbung 188
11.2.1 Die Notwendigkeit virtueller Methoden 188
11.2.2 Virtuelle Methoden 189
11.2.3 Der Unterschied zwischen new und override 191
11.2.4 Abstrakte Klassen 193
11.2.5 Verhindern der Vererbung 194
11.3 Was alle erben – System.Object 195
11.4 Übungen 198

12 Fehlerbehandlung 199

12.1 Versuch und Fehler – try/catch 199
12.1.1 Explizites catch 199
12.1.2 Generelles catch 200
12.2 Die Hierarchie von Exceptions 201
12.3 Ausnahmen – System.Exception & Co 203
12.4 Schutzblocks mit finally 205
12.5 Kaskadierende Exceptions 206
12.6 Exceptions selbst gemacht 208
12.6.1 Ableitung der Exception-Klasse 208
12.6.2 Auslösen einer Exception 209
12.7 Zusammenfassung 211
12.8 Übungen 212

13 Schnittstellen (Interfaces) 213

13.1 Probleme klassenbasierter Vererbung 213
13.2 Was sind Interfaces 216
13.2.1 Deklaration einer Schnittstelle in C# 217
13.2.2 Implementieren einer Schnittstelle 218
13.2.3 Schnittstellen mit Eigenschaften 219
13.3 Arbeiten mit Schnittstellen 219
13.3.1 Benutzen einer Schnittstelle 219
13.3.2 Testen auf eine bestimmte Schnittstelle 222
13.3.3 Typumwandlungen für Schnittstellen 223

13.4 Implementieren mehrerer Schnittstellen 223
13.4.1 Definitionen der Klassen und Schnittstellen 224
13.4.2 Explizite Schnittstellenimplementierung 227
13.5 Zusammenfassung 231
13.6 Übungen 232

14 Ein- und Ausgabe 233

14.1 Dateioperationen 233
14.1.1 Die Klasse Path 235
14.1.2 File und Directory 237
14.2 Das Stream-Konzept 240
14.2.1 Streamklassen 240
14.2.2 Lesen und Schreiben einer Textdatei 242
14.2.3 Codierung von Dateien 243
14.2.4 Peek – 'Nur mal einen Blick drauf werfen' 245
14.2.5 Lesen und Schreiben von Binärdateien 246
14.2.6 Stream-Eigenschaften 247
14.3 Lesen und Schreiben von XML-Dateien 248
14.3.1 Zugriffsarten 248
14.3.2 Die Beispieldatei 249
14.3.3 XML-Dateien laden und anzeigen 250
14.3.4 Suchen und Ändern im Dokument 254
14.3.5 Einfügen und Löschen von Knoten 255
14.3.6 XmlDocument-Instanzen in Dateien speichern 256
14.3.7 Lesen von XML-Dateien mit XmlReader 257
14.3.8 Schreiben von XML-Dateien mit XmlWriter 261
14.4 Übungen 264

15 Strings 265

15.1 String-Literale 265
15.2 System.String im Überblick 266
15.2.1 Methoden der String-Klasse 267
15.3 Strings formatieren 270
15.3.1 Standard-Formate 270
15.3.2 Standard-Formatierung 271
15.4 Die StringBuilder-Klasse 275
15.4.1 Erzeugen eines StringBuilders 276
15.4.2 Eigenschaften der Klasse 276
15.4.3 StringBuilder-Methoden 277
15.5 String oder StringBuilder? 277
15.6 Übungen 279

16 Konvertierungen 281

16.1 Implizite Konvertierungen 281
16.1.1 .NET-Datentypen 282
16.2 Explizite Konvertierungen 283
16.3 Benutzerdefinierte Konvertierungen 283

16.3.1 Der Beispieltyp ClockTime 283
16.3.2 Implizite Konvertierungsroutine 285
16.3.3 Explizite Konvertierungsroutine 286
16.3.4 Implizit oder explizit? 286
16.3.5 Auflösungskonflikte 287
16.3.6 Die komplette Struktur 287
16.4 Zusammenfassung 289
16.5 Übungen 290

17 Windows Forms 291

17.1 Einführung 291
17.2 Das Programmiermodell 292
17.2.1 Wo bleibt die Formulardatei? 297
17.2.2 Läuft und läuft und läuft – Application.Run 297
17.3 Steuerelemente 299
17.3.1 Steuerelemente hinzufügen 299
17.3.2 Ereignisbehandlung für Steuerelemente 304
17.4 Wichtige Klassen 306
17.4.1 MessageBoxen einst und jetzt 307
17.4.2 Menüs 307
17.4.3 Bestehende Ereignisse zuweisen 312
17.5 Komponenten zur Laufzeit erzeugen 312
17.5.1 Steuerelemente 312
17.5.2 Menüs 313
17.6 Zusammenfassung 315
17.7 Übung 316

18 Indexer und Auflistungen 317

18.1 Indexer 317
18.1.1 Deklaration eines Indexers 317
18.1.2 Zugriff über einen Indexer 320
18.1.3 Designüberlegungen 321
18.1.4 Indexer und Eigenschaften im Vergleich 324
18.2 Auflistungen 324
18.2.1 Auflistungsklassen 325
18.2.2 Auflistungen, Indexer & foreach 329
18.3 Zusammenfassung 334

19 Fehlersuche mit C# 335

19.1 Das Attribut [Conditional] 335
19.2 Die Klassen Debug und Trace 335
19.2.1 Mitglieder von Debug und Trace 337
19.2.2 Asserts 338
19.2.3 Ausgabe der Diagnosemeldungen 339
19.2.4 Das Listener-Konzept für Debug-Output 341

19.2.5 Einen eigenen Listener implementieren 342
19.2.6 Kommunikation mit dem Debugger 346
19.3 Zusammenfassung 346

20 Delegates & Events 347

20.1 Delegates 347
20.1.1 Deklaration eines Delegate 347
20.2 Ereignisse 351
20.2.1 Ereignisinformationen definieren 352
20.2.2 Ereignisauslösende Klasse & Delegate 353
20.2.3 Das Ereignisobjekt definieren 354
20.2.4 Die Ereignisbehandlungsroutine 354
20.2.5 Klasse mit dem Abonnenten erstellen 355
20.2.6 Callback-Funktion des Abonnenten 356
20.2.7 Die Callback-Methode anmelden 356
20.2.8 Abschlussarbeiten 357
20.2.9 Die Testklasse bauen 359
20.2.10 Das große Ganze 359
20.3 Zusammenfassung 362

Reference – Teil III 363

21 Garbage Collection 365

21.1 Grundlagen 365
21.1.1 Die alte Welt – Reference Counting 366
21.1.2 Die neue Welt – Garbage Collection in .NET 366
21.2 Finalize, Dispose, Close oder was? 367
21.2.1 Konzepte für die Freigabe von Objekten 367
21.2.2 Sichere Speicherfreigabe mit using 368
21.3 GC-Generationen 371
21.4 Die Garbage Collection steuern 371
21.5 Weitere Methoden der Klasse GC 373
21.5.1 Finalisierungen unterdrücken 373
21.5.2 Destruktoren doch wieder ausführen 374
21.5.3 Objekte am Leben erhalten 375
21.6 Zusammenfassung 375

22 Reflection in C# 377

22.1 Type, GetType() und typeof() 377
22.1.1 Die Beispielklasse 378
22.1.2 Her mit dem Typ: das Type-Objekt 379
22.1.3 Members only – Klassenmitglieder auflisten 379
22.1.4 Parameter fischen 381
22.1.5 Aktivieren – Dynamischer Methodenaufruf 383

22.2 Reflection überzeugend demonstriert 384
22.2.1 Die Problemstellung 384
22.2.2 Der Lösungsansatz 385
22.3 Zusammenfassung 388

23 Attribute 391

23.1 Grundlagen 391
23.2 Vordefinierte Attribute 392
23.3 Was passiert bei der Benutzung eines Attributes? 395
23.4 Benutzerdefinierte Attribute 395
23.4.1 Definition der Attributklasse 395
23.4.2 Das Attribut AttributeUsageAttribute 396
23.4.3 Optionale Parameter in eigenen Attributen 398
23.5 Abrufen von Attributinformationen 399
23.5.1 Abfragen eines Attributes 400
23.5.2 Abrufen der Informationen mit Reflection 400
23.5.3 Feststellen, ob ein Attribut definiert ist 401
23.6 Zusammenfassung 402

24 Überladen von Operatoren 403

24.1 Wie funktioniert Operatorüberladung? 403
24.2 Was kann überladen werden? 404
24.2.1 Unäre Operatoren 404
24.2.2 Binäre Operatoren 405
24.2.3 Relationale Operatoren 405
24.2.4 Ein Beispiel zur Operatorüberladung 406
24.2.5 Überladen oder nicht überladen? 408
24.3 Zusammenfassung 409

25 Serialisierung 411

25.1 Attribute oder ISerializable? 411
25.2 Die Beispielklasse 412
25.3 Attributgesteuerte Serialisierung 413
25.3.1 Serializable und NonSerialized 413
25.3.2 Formatter-Klassen 414
25.3.3 Der Prozess der Serialisierung 415
25.3.4 Deserialisierung 417
25.4 ISerializable implementieren 422
25.5 XML-Serialisierung 424
25.5.1 Die Beispielklasse 425
25.5.2 Serialisierung in eine XML-Datei 426
25.5.3 Deserialisierung aus einer XML-Datei 427
25.5.4 Steuerung der XML-Serialisierung 428
25.6 Zusammenfassung 429

26 Assemblies 431

26.1 Die Probleme der Vergangenheit 431
26.2 Was sind Assemblies? 432
26.2.1 Private und öffentliche Assemblies 434
26.2.2 Assemblies angucken 436
26.3 Öffentliche Assemblies erstellen 437
26.3.1 Der prinzipielle Ablauf 437
26.3.2 Beispiel-Bibliothek erstellen 437
26.3.3 Schlüsselpaar erzeugen 438
26.3.4 Signieren des Assemblies 438
26.3.5 Der globale Assembly-Cache 441
26.3.6 Assembly im GAC eintragen 442
26.4 Öffentliche Assemblies benutzen 444
26.5 Versionsverwaltung bei Assemblies 446
26.5.1 Aufbau der Versionsnummern 446
26.6 Informationen im Programm abrufen 447
26.7 Konfigurationsdateien 448
26.8 Zusammenfassung 454

27 Unsicherer Code 455

27.1 Zeiger in C# 455
27.2 Das Schlüsselwort unsafe 456
27.3 Nutzung von Zeigern 457
27.3.1 Deklaration von Zeigern 457
27.3.2 Zeiger-Operatoren 457
27.3.3 Beispiel für Zeigervariablen 458
27.3.4 Typumwandlungen von Zeigern 459
27.3.5 sizeof und Zeigerarithmetik 461
27.4 Der Operator -> 463
27.5 Zusammenfassung 464

28 XML-Dokumentation 465

28.1 XML-Kommentare 465
28.1.1 Eingabe von XML-Kommentaren 466
28.1.2 Unterstützte XML-Elemente 467
28.1.3 Generieren der Dokumentationdatei 468
28.1.4 Syntaxprüfung des Elementinhaltes 471
28.2 Zusammenfassung 472

29 Lösungen zu den Übungen 473

29.1 Kapitel 4 473
29.1.1 Die Fragen 473
29.1.2 Die Antworten 473
29.2 Kapitel 5 474
29.2.1 Die Fragen 474
29.2.2 Die Antworten 474

29.3 Kapitel 6 476
29.3.1 Die Fragen 476
29.3.2 Die Antworten 476
29.4 Kapitel 7 478
29.4.1 Die Fragen 478
29.4.2 Die Antworten 478
29.5 Kapitel 8 479
29.5.1 Die Fragen 479
29.5.2 Die Antworten 480
29.6 Kapitel 9 481
29.6.1 Die Fragen 481
29.6.2 Die Antworten 481
29.7 Kapitel 10 482
29.7.1 Die Fragen 482
29.7.2 Die Antworten 482
29.8 Kapitel 11 484
29.8.1 Die Fragen 484
29.8.2 Die Antworten 484
29.9 Kapitel 12 485
29.9.1 Die Fragen 485
29.9.2 Die Antworten 485
29.10 Kapitel 13 487
29.10.1 Die Fragen 487
29.10.2 Die Antworten 487
29.11 Kapitel 14 489
29.11.1 Die Fragen 489
29.11.2 Die Antworten 489
29.12 Kapitel 15 489
29.12.1 Die Fragen 489
29.12.2 Die Antworten 490
29.13 Kapitel 16 490
29.13.1 Die Fragen 490
29.13.2 Die Antworten 490
29.14 Kapitel 17 492
29.14.1 Die Fragen 492
29.14.2 Die Antworten 492

30 Die Werkzeuge 493

30.1 Der Compiler 493
30.1.1 Grundlagen 493
30.1.2 Compiler-Optionen 493
30.2 ILDASM und ILASM 496
30.2.1 ILDASM 496
30.2.2 Die GUI-Oberfläche 497
30.2.3 Anzeige des IL-Codes 498
30.3 ILASM 499

30.4 Debugger 499
30.4.1 Aufruf des Debuggers 499
30.4.2 Die Kommandos 500
30.5 WinCV 502
30.6 Visual Studio.NET 503
30.7 Zusammenfassung 503

31 Referenzteil 505

31.1 Datentypen 505
31.1.1 Werttypen 505
31.1.2 Implizite Typumwandlungen 506
31.1.3 Referenztypen 507
31.2 Anweisungen 509
31.2.1 Auswahlanweisungen 509
31.2.2 Schleifenanweisungen 510
31.2.3 Verzweigungsanweisungen 510
31.2.4 Fehlerbehandlung 511
31.2.5 Codeausführung 511
31.2.6 Namensraumnutzung 512
31.3 Präprozessorkommandos 513
31.3.1 #if, #else, elif, endif 513
31.3.2 #define, #undef 514
31.3.3 #warning, #error 514
31.3.4 #line 515
31.3.5 #region, #endregion 515
31.4 Namensräume 515
31.4.1 MS-spezifische Namensräume 515
31.4.2 .NET-Namensräume 515
31.5 Vordefinierte Attribute 519

Stichwortverzeichnis 523

Vorwort

»Möchten Sie nicht ein Buch über C# schreiben?« Als Redner auf einer Software-Konferenz ist man auf allerlei Fragen gefasst, aber neben dem Beruf auch noch Autor werden? Im Nachhinein freue ich mich, damals zugesagt zu haben. Es hat Spaß gemacht, ein Buch über eine neue Programmiersprache zu schreiben und zu versuchen, auf dem schmalen Grat zwischen »Kenne ich schon, langweilig« und »Ja, das ist neu und interessant« nicht abzurutschen. Ob es mir gelungen ist, entscheiden Sie als Leser.

Ich habe mit Absicht den trockenen Ton mancher Fachbücher vermieden und theoretische Erklärungen auf das notwendige Minimum beschränkt. Sämtliche Beispielprogramme finden Sie auf der beiliegenden CD. Nur soweit der Programmcode zum Verständnis nötig ist, wurde er auch im Buch abgedruckt, Sie werden also nicht von seitenlangen Listings erschlagen und ich wünsche Ihnen viel Spaß bei der Arbeit mit C# und der .NET-Laufzeitumgebung.

Besonderer Dank gilt für die Betreuung meiner Lektorin Christiane Auf sowie dem gesamten Team von Addison-Wesley, das dieses Buch erst möglich gemacht hat. Ein dickes Dankeschön auch an Heinz, Evelyn, Dave und Wolfgang für wertvolle Anregungen, die Zeit zum Probelesen und Eure Ideen. Unsere Katze hat ihr Bestes gegeben, mich vom Schreiben abzuhalten und durch Liegen auf der Tastatur ganze Sätze umzustellen. Ich hoffe, alle Eingriffe wieder rückgängig gemacht zu haben. Vor allem aber möchte ich mich bei meiner Frau Andrea bedanken. Nicht nur, dass sie es mit jemandem aushält, bei dem Hobby und Beruf nahtlos ineinander übergehen, sondern auch für das Verständnis, die frischen Espressi und die Massagen nach langen Schreibabenden. HDM!

Für Anregungen, Kommentare, konstruktive Kritik und Hinweise auf leider entgangene Fehler ist der Verfasser dankbar. Senden Sie diese entweder über den Verlag oder per Email an die Adresse armin.hanisch@addison-wesley.de.

Ergolding, im Winter 2001
Armin Hanisch



Basics – Teil I

1 Einführung

1.1 Noch eine neue Programmiersprache?

Zugegeben, das ist kein besonders kreativer Einstieg für ein Buch über eine neue Programmiersprache. Dennoch werden Sie sich vielleicht genau diese Frage gestellt haben, als Microsoft das .NET SDK vorstellte. Schließlich gibt es bereits erprobte Sprachen für die Entwicklung von Anwendungen unter Windows: Visual C++, Visual Basic, Delphi und viele mehr.

Eigentlich sollte die Frage genau andersherum gestellt werden: warum keine neue Programmiersprache? Werkzeuge zur Softwareentwicklung sind, wie der Name bereits ausdrückt, Werkzeuge. Ändern sich 'Materialien' oder Anforderungen, sollten Sie auch ein passendes Werkzeug benutzen. Um es mit Abraham Maslow zu sagen: 'Wenn Sie nur einen Hammer kennen, sieht jedes Problem wie ein Nagel aus'.

In diesem Buch erfahren Sie unter anderem, warum die Entscheidung von Microsoft für die Entwicklung einer neuen Programmiersprache (oder die konsequente Weiterführung erprobter Konzepte, je nach Blickwinkel) durchaus sinnvoll war und welche Vorteile dies für Sie als Entwickler mit sich bringt. Sie lernen C# in all seinen Facetten auch in seinen fortgeschrittenen Anwendungen kennen und erfahren natürlich auch eine Menge über die darunter liegende .NET-Laufzeitumgebung. Nach der Lektüre dieses Buches werden Sie mir sicherlich zustimmen, dass C# das Zeug dazu hat, die beliebteste Sprache für .NET-Anwendungen zu werden.

1.2 An wen richtet sich das Buch?

Dieses Buch ist für alle gedacht, die an C# und der Nutzung des .NET-Frameworks von Microsoft mit C# interessiert sind. Es ist keine komplette Behandlung der .NET-Laufzeitumgebung (dann hätten Sie mit diesem Buch noch drei Folgebände erhalten), der Schwerpunkt liegt auf der Besprechung von C# als moderne, objektorientierte Programmiersprache und deren Möglichkeiten im .NET-Umfeld. Dennoch werden Sie in diesem Buch auch Informationen zu sprach-

übergreifenden Themen der .NET-Laufzeitumgebung finden, solange der Bezug zu C# gegeben ist. In gewisser Hinsicht sind Teile des Buches also der Versuch, eine Grenze zwischen der Programmiersprache C# und den Klassenbibliotheken der .NET-Laufzeitumgebung zu ziehen. Bei mehr als 7000 Klassen musste ich irgendwo aufhören, Dinge aus der .NET-Laufzeitumgebung in ein C#-Buch zu packen.

1.2.1 Leserkenntnisse

Wie bei jedem Buch gehe auch ich von bestimmten Grundannahmen über Sie als Leser, Ihre Erfahrung in der Entwicklung von Software und die Ihnen zur Verfügung stehende Ausrüstung aus.

Die gute Nachricht: Sie müssen kein Programmier-Guru sein, um dieses Buch benutzen zu können. Die Zielgruppe sind Personen, die bereits mit einer anderen Programmiersprache Erfahrung gesammelt haben. Dieses Buch ist also nicht für Einsteiger in die Programmierung gedacht, sondern wie der (bei OOP-Entwicklern für Gänsehaut sorgende) Titel nahelegt, für interessierte Entwickler und Umsteiger von anderen Programmiersprachen. Sie benötigen allerdings lediglich Kenntnisse einer prozeduralen Programmiersprache. Natürlich haben Sie es leichter, wenn Sie schon mal eine Klasse gesehen haben, aber dies ist keine Voraussetzung. Aus diesem Grund werde ich die grundlegenden Konstrukte von C#, wie Ablaufsteuerung oder Variablen, die auch in anderen Sprachen vorhanden sind, eher gestrafft behandeln. Damit habe ich dann mehr Platz für die Grundlagen der objektorientierten Programmierung, da hier bei vielen Umsteigern von beispielsweise Visual Basic oder C noch nicht der entsprechende Hintergrund vorhanden ist. Leser aus 'echten' OOP-Sprachen bitte ich an dieser Stelle um Nachsicht für etwas ausführliche oder vereinfachende Erläuterungen. Ansonsten benötigen Sie nur noch etwas Neugierde und einen gewissen Sinn für Humor, um meine über das Buch verstreuten Anmerkungen und Exkurse zu genießen …

1.2.2 Systemvoraussetzungen

Die Anforderungen an das Ihnen zur Verfügung stehende Computersystem sind schnell geklärt: Sie sollten Windows 2000 oder Windows XP zur Verfügung haben, wenn möglich mehr als 128 MByte RAM und das .NET-Framework SDK und die Visual Studio.NET Entwicklungsum-

gebung. Letzteres (VS.NET) ist für die meisten Beispiele in diesem Buch nicht notwendig, so dass Sie beinahe alle Quellcodes auch nur mit dem .NET Framework SDK nachvollziehen können. Aus diesem Grunde auch die Vielzahl von Beispielen als reine Konsolenanwendung. Eine Auflistung lässt sich auch ohne den Ballast einer grafischen Umgebung demonstrieren und eine neue Programmiersprache ist schon genug, da muss es nicht auch noch ein neues GDI-Interface sein.

1.3 Konventionen

Für dieses Buch gelten die folgenden typografischen Konventionen:

Format	Bedeutung
Fließtext	Der normale Text des Buches
Kursiv	Wichtige Begriffe, zusätzliche Anmerkungen
`Listing`	Diese Schrift mit festem Zeichenabstand wird für Quelltext verwendet
F5	Diese Schriftart dient der Kennzeichnung von Tastenkombinationen
Änderungen	Fettgedruckt sind wichtige Hinweise und vor allem auch Änderungen am Quelltext im Laufe der einzelnen Kapitel
KAPITÄLCHEN	Kapitälchen dienen der Angabe von Menüpunkten und Dialogen



Dieses Symbol »Achtung« zeigt Ihnen für die jeweils angegebene Programmiersprache Informationen, die besonders wichtig sind, oder wo sich C# stark von der Programmiersprache unterscheidet oder dieser besonders ähnelt.

Hinweise sind Exkurse und zusätzliche Informationen, Kommentare des Autors und andere Dinge, die zwar interessant, aber für das Verständnis des Buches nicht unbedingt notwendig sind.



Tipps beinhalten kleine Ratschläge zum Herangehen an bestimmte Problemstellungen, Abkürzungen, Hinweise auf bestimmte Themen der Online-Hilfe und mögliche Veränderungen des Beispielcodes als Ausgangspunkt für eigene Experimente.



Mit diesem Symbol werden Übungsfragen am Ende vieler Kapitel gekennzeichnet, für die Sie in einem eigenen Kapitel die entsprechenden Lösungen finden.

Noch ein Wort zu den Übungen: In den fortgeschritteneren Kapiteln finden Sie keine Übungsaufgaben mehr, sondern nur noch eine Zusammenfassung des Kapitels. Da ich davon ausgehe, dass Sie bereits über Erfahrung mit einer anderen Programmiersprache verfügen, sind Sie sicher in der Lage, die auf der CD gelieferten Beispiele für eigene Experimente umzubauen. Sinnvolle Übungen und Beispiele würden ab einer gewissen Ebene zu umfangreich und speziell, um über das Niveau von ‚Spielzeugcode' hinaus zu reichen. Haben Sie eine Idee für ein Beispiel zu einem Thema oder würden Sie gerne mehr Übungsaufgaben haben? Lassen Sie es mich wissen!

Exkurse, Zusatzinformationen

Einige Themen verdienen etwas ausführlichere Erläuterungen, welche aber nicht unbedingt für das Verständnis von C# notwendig sind. Aus diesem Grund werden diese in einem kleinen Exkurs behandelt, der durch einen Kasten eingerahmt ist.

Im Buch verwende ich durchgängig die Form 'Softwareentwickler' oder 'Entwickler' für alle an der Programmierung beteiligten Personen. Dies geschieht rein aus Gründen der leichteren Lesbarkeit (der/die Entwickler/in ist schrecklich) und schließt natürlich ausdrücklich alle weiblichen Fachkolleginnen ein. Ein Großteil meiner x86-Assemblerkenntnisse stammt von einer netten Kommilitonin im Biologie-Anfangssemester (ja, ich wollte mal was anderes als EDV machen), die wahrscheinlich bei Michael Abrash selbst gelernt hat.

1.4 Gliederung

Das Buch gliedert sich thematisch in drei Teile. Im ersten Abschnitt erhalten Sie grundlegende Informationen über den Aufbau der .NET-Laufzeitumgebung und das Design der Sprache C# sowie einen generellen Überblick. Hier finden Sie auch die grundlegenden Informationen, so dass Sie Programme in C# zum Großteil bereits lesen können.

Der zweite Teil behandelt fortgeschrittene Konzepte von C# und die wichtigsten Basisklassen der .NET-Laufzeitumgebung. Mit dem Inhalt dieser ersten beiden Abschnitte sollten Sie ca. 90% ihrer Programmierung mit C# erledigen können.

Der dritte Abschnitt widmet sich den Themen, die C# im Gegensatz zu Visual Baisc, C++ oder Java grundlegend anders handhabt oder die Sie für weit fortgeschrittene Programmierung benötigen, wie z.B. das Erstellen eigener Attribute, die Steuerung der Garbage Collection und das Arbeiten mit Zeigern in C#.

1.5 Die CD zum Buch

Auf der beiliegenden CD finden Sie die entsprechenden Beispielcodes aus dem Buch. Damit Sie die Beispiele auch ohne Visual Studio.NET nachvollziehen können, habe ich nur die Quellcodes ohne die entsprechenden VS.NET-Projekte gespeichert. Benutzen Sie Visual Studio, erstellen Sie einfach ein entsprechendes Projekt und fügen die .cs-Dateien dem Projekt hinzu.

Alle Quellcodes und Beispielprogramme wurden vom Autor nach bestem Wissen und Gewissen getestet, allerdings kann keine Garantie für eine Fehlerfreiheit übernommen werden. Wenn Sie die Tipparbeit vermeiden wollen, dann ohne Garantie für Fehlerlosigkeit des Beispielquellcodes.

Ich habe darauf verzichtet, auf die beiliegende CD das .NET Framework SDK in der Version Beta2 zu kopieren. Diese wäre bei Erscheinen des Buches aufgrund der Verfügbarkeitsplanung von Microsoft bereits veraltet. Ich verweise Sie daher auf die msn-Site von Microsoft, wo Sie unter www.microsoft.com/dotnet alle aktuellen Versionen finden.

2 Das .NET-Framework

Das .NET-Framework ist keine Microsoft-spezifische Umgebung. Microsoft hat die Hersteller von Compilern und Programmiertools eingeladen, am Ausbau mitzuwirken und weitere Sprachen .NET-fähig zu machen. Welche Bestandteile der .NET-Umgebung das möglich machen und wie diese zusammenarbeiten, ist Inhalt dieses Kapitels.

2.1 Was ist .NET?

Diese Frage dürfte in den letzten Monaten im Zusammenhang mit der Diskussion von Microsoft-Technologien eine gute Chance auf den ersten Platz in der Liste der am häufigsten gestellten Fragen haben. Je nachdem, wem Sie diese Frage stellen, erhalten Sie eine andere Antwort. Die Extreme reichen von ».NET ist nur dann .NET, wenn es sich um verteilte Objekte und den Einsatz des BizTalk-Servers mit dem Versand von XML-Dateien per SOAP handelt« über ».NET ist eine Vision für die Nachfolge von Visual Basic« bis zu ».NET? Das sind 7.000 Klassen mit 150.000 Methoden und damit nicht lernbar«.

2.1.1 Die Bausteine von .NET

Die Wahrheit berührt beide Extreme. Die .NET-Initiative besteht aus einer Reihe von Blöcken, die jeweils einen ganz bestimmten Bereich der EDV berühren und daher auch je nach Berufsgruppe zu verschiedenen Antworten führen. Ein Entwickler wird diese Frage anders beantworten als der Leiter eines Rechenzentrums, der die nächste Generation seiner Infrastruktur plant. Daher ist .NET nicht einfach die nächste Generation von Windows oder nur die Programmiersprache C#. Die nachfolgende Abbildung soll verdeutlichen, welche Bereiche zusammen .NET ausmachen.

Thema dieses Buches ist die Programmiersprache C# und damit der markierte Block, das .NET Framework oder die .NET-Laufzeitumgebung, wie ich diesen Teil für den Rest des Buches bezeichnen möchte.

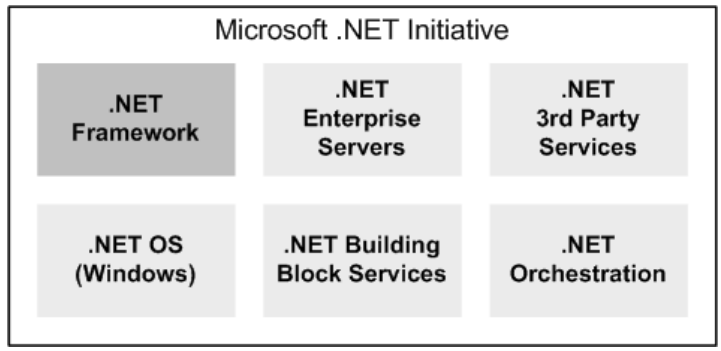


Abbildung 2.1: Übersicht über .NET

2.1.2 Die Rolle von Windows

Wie aus der Übersicht zu sehen ist, steht bei dem Betriebssystem für .NET immer noch Windows und natürlich lauert unter der Schicht der .NET-Laufzeitumgebung immer noch die alte Windows-API. Microsoft hat allerdings .NET ausdrücklich betriebssystemunabhängig angelegt. Dass dies kein reines Marketing ist, sieht man an den Fortschritten des Mono-Projektes, einer GPL-Implementierung der CLR und eines C#-Compilers für Linux. Interessenten finden weitere Informationen auf der Mono-Homepage unter *http://mononet.sourceforge.net* im Web.

2.2 Das .NET-Framework im Überblick

Bevor ich im Urlaub in eine Region fahre, die ich noch nicht kenne, suche ich mir zwei Landkarten: eine Überblickskarte der Region und eine möglichst detaillierte für die Einzelheiten. Genauso werde ich hier verfahren. Als Erstes möchte ich Ihnen das .NET-Framework in groben Blöcken vorstellen, welche wir dann Stück für Stück weiter zerlegen.

Dies sind die Bausteine des .NET-Frameworks, wobei für unsere Zwecke noch die entsprechenden Entwicklungswerkzeuge (Compiler) hinzukommen, gleich ob in Form von Visual Studio.NET oder als Kommandozeilenversion.

Alles, was Sie im obigen Diagramm sehen, ist unabhängig von einer bestimmten Programmiersprache und alle .NET-Sprachen nutzen die gleichen Services aus diesem Framework.

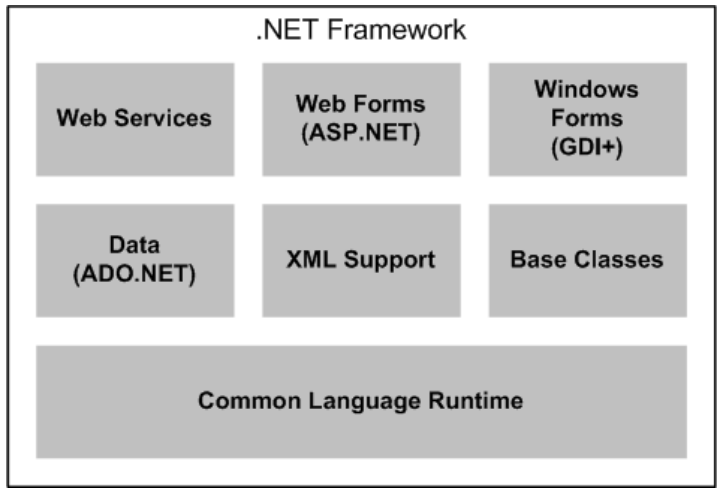


Abbildung 2.2: Der Grobaufbau des .NET-Frameworks

Prinzipiell könnte man den Umfang also bildlich darstellen, indem Sie sich vorstellen, dass die VB-Laufzeitbibliotheken und die C++ Bibliotheken und alles andere, was Sie bis jetzt an Zusatzkomponenten benötigt haben, in einer großen Bibliothek kompatibel für alle Programmiersprachen vorliegen. Die gesamte .NET-Laufzeitumgebung besitzt einen Umfang von weit über 7.000 Klassen mit ca. 150.000 Methoden und Eigenschaften! Trotz dieses gewaltigen Umfangs werden Sie feststellen, dass die Einarbeitung leichter ist als alles, was bisher an Bibliotheken für Windows zu haben war.

2.2.1 Die Vorteile des .NET-Frameworks

Der folgende Abschnitt fasst die hauptsächlichen Vorteile der .NET-Laufzeitumgebung als kleinen Vorgriff in einer Übersicht zusammen. Der Begriff in Klammern hinter dem Untertitel stellt den entsprechenden englischen Begriff bzw. das immer wieder zitierte Schlagwort in den technischen Unterlagen dar, so dass Sie die einzelnen Themen zuordnen können.

Automatische Speicherverwaltung (garbage collection)

Speicher für Ressourcen, den Sie während der Programmausführung anfordern, muss nicht mehr explizit freigegeben werden. Darum kümmert sich die automatische Verwaltung des Arbeitsspeichers. Auch die

Zuteilung des Speichers erfolgt unter der Kontrolle der .NET-Laufzeitumgebung, so dass die Gefahr von Speicherlecks weitgehend gebannt ist. Das Entfernen nicht mehr benötigter Referenzen auf Speicherbereich erfolgt mit minimalen Performance-Einbußen und ist auch in der Lage, gegenseitige zirkuläre Referenzen aufzulösen, was bei COM nicht der Fall war.

Sprachunabhängiger Zwischencode (intermediate language)

Die Übersetzung in einen sprachunabhängigen Zwischencode erlaubt nicht nur eine relativ kompakte Speicherung der ausführbaren Codestücke, sondern vor allem eine schnelle Umsetzung in den jeweiligen Maschinencode der Zielplattform und eine sprachneutrale Nutzung übersetzter Bibliotheken.

IL nach native code-Umsetzung (just in time-compiler, JIT)

Dieser Punkt räumt mit dem Leistungsnachteil bisheriger Systeme auf, die einen Zwischencode benutzten (beispielsweise Java Bytecode). Die hochoptimierten Befehle der Zwischensprache (intermediate language) sind auf einer entsprechend tiefen Ebene des Systems angesetzt (virtuelles Ziel der IL ist eine Stackmaschine), so dass eine schnelle Umsetzung in die Maschinensprache des Zielsystems möglich ist. Für extrem leistungskritische Anwendungen ist bei der Installation der Anwendung eine Vorübersetzung in den plattformabhängigen CPU-Code möglich (preJIT).

Daher wird ein Assembly nie als Zwischencode ausgeführt, sondern es erfolgt immer eine Übersetzung. Entweder bei der Installation oder zur Ladezeit des Assemblys. Dann bestehen zwei Möglichkeiten: einmal der so genannte Econo-JIT, der eine möglichst schnelle, wenn auch nicht optimale Umsetzung gewährleistet, deren Ergebnis nach der Ausführung wieder verworfen wird, zweitens der ›normale‹ JIT, dessen hochoptimierte Ausgabe im Cache des Systems vorgehalten wird, so dass bei der nächsten Ausführung nur noch verglichen werden muss, ob diese Version bereits umgesetzt wurde.

Mehrere Anwendungen pro Prozessraum (application domains)

Da die .NET-Laufzeitumgebung den Speicher verwaltet, den eine Anwendung anfordern kann, ist diese auch immer über die entsprechenden Zugriffe informiert und kann verschiedene Anwendungen bzw. Ausführungseinheiten in einen Prozessraum laden und in ver-

schiedenen ›Subprozessen‹ (application domains) isolieren. Da aufgrund der Speicherverwaltung keine Anwendung in den Speicherbereich einer anderen Anwendung schreibt, stellt dies kein Risiko dar. Der durch den Wegfall der nötigen Kontext- und Prozesswechsel bei einer Kommunikation der Anwendungen entstehende Leistungsgewinn ist aber erheblich.

Objektorientiertes Programmiermodell (base classes)

Die Klassen der .NET-Laufzeitumgebung bilden eine komplett objektorientierte und konsistente Schicht zur Programmierung von Anwendungen. Ein Zugriff beispielsweise auf die Windows-API ist nicht mehr notwendig, die Klassen kapseln das darunter liegende Betriebssystem. Bei einer späteren Portierung der .NET-Laufzeitumgebung auf ein anderes Betriebssystem muss der Code kaum mehr angepasst werden (wenn entsprechend sauber programmiert wurde).

Selbstbeschreibende Ausführungseinheiten (manifests)

Die Ausführungseinheit einer Anwendung sind die sog. Assemblies ('Baugruppen' ist die entsprechende Übersetzung von Microsoft). Diese sind selbstbeschreibend und enthalten entsprechende Metadaten, so dass eine Eintragung in die Registry nicht mehr nötig ist und ein Assembly Auskunft über die benötigten Sicherheitseinstellungen für eine Ausführung geben kann.

Einfache Installation (xcopy deployment)

Für eine Anwendung ist keine umfangreiche Installation und Registrierung von Bibliotheken mehr nötig, prinzipiell reicht das Kopieren der entsprechenden Dateien in das Verzeichnis der Anwendung. Die Lokalisierung ist ebenso einfach über entsprechend konfigurierbare Unterordner mit länderspezifischen Ressourcen möglich.

Durchgängige Code-Sicherheit (code access / role based security)

Durch die Übersetzung in IL-Opcodes und die im Assembly enthaltenen Metadaten kann die .NET-Laufzeitumgebung beim Laden einer Anwendung feststellen, ob bestimmte Aktionen nicht ausgeführt werden dürfen. Die Berechtigungen können nicht nur rollenbasiert für einzelne Benutzer, sondern erstmals auch code- und lokationsabhängig (z.B. für Downloads aus dem Internet) vergeben werden.

Erweiterte Laufzeit-Typinformationen (reflection)

Jede Anwendung besitzt den Zugriff auf sämtliche Informationen zu den enthaltenen Datentypen über eine reflection genannte Schnittstelle. Diese erlaubt das Auslesen von Informationen über die eigenen und auch fremde Klassen (bei entsprechender Sicherheitseinstellung). Sogar neue Typen und komplette Assemblies können zur Laufzeit dynamisch erzeugt werden.

2.2.2 Das Common Type System

Um diese Unabhängigkeit von einer bestimmten Programmiersprache zu garantieren, muss natürlich Einigkeit darüber bestehen, welche Datentypen in einer Umgebung zur Verfügung stehen und wie diese Datentypen aufgebaut sind.

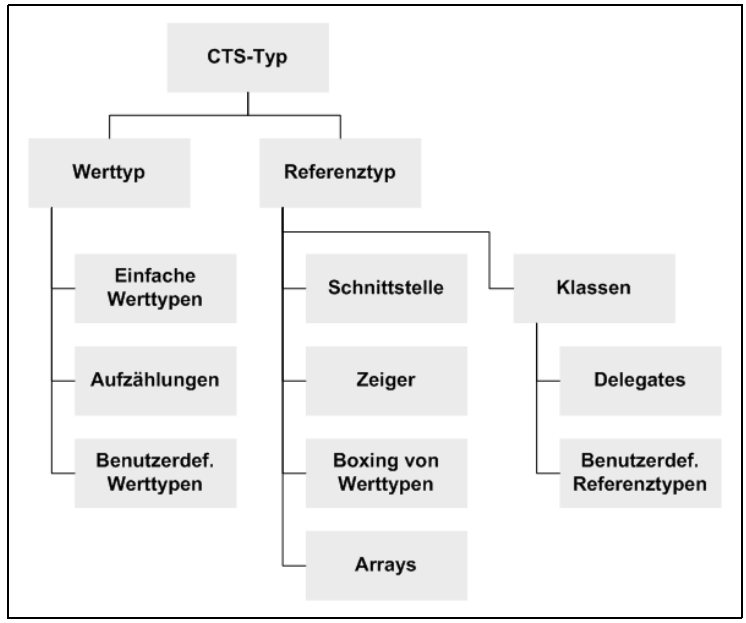


Abbildung 2.3: Der komplette CTS-Typbaum

Dies betrifft nicht nur einfache Typen wie einen Integer, sondern auch komplexe OOP-Typen wie eine Klasse oder eine Schnittstelle. Microsoft hat hier versucht, beim Design der möglichen Datentypen sowohl

Flexibilität als auch Einfachheit zu vereinen und Probleme anderer Umgebungen (z.B. einfache Typen in Java, Copy Constructors in C++) zu vermeiden. Das Ergebnis dieser Überlegungen ist das Common Type System (CTS), die Basis der möglichen Datentypen. Grundsätzlich existieren in der .NET-Welt nur zwei Typen von Typen (welch Formulierung!): Werttypen und Referenztypen. Alle anderen Datentypen sind nur speziellere Ableitungen. Auf Seite 32/Abb. 2.3 sehen sie den kompletten Typbaum des CTS.

2.2.3 Die Common Language Specification (CLS)

Nicht jede Programmiersprache ist in der Lage, alle möglichen Datentypen aus diesem gemeinsamen, maximal möglichen Typsystem zu unterstützen. Was also tun? Die Lösung ist eine gemeinsame Basis, sozusagen der kleinste gemeinsame Nenner aller .NET-Sprachen, um eine Sprachunabhängigkeit zu gewährleisten. Diese Minimalforderung heißt Common Language Specification (CLS) und muss von allen Programmiersprachen unterstützt werden, welche die .NET-Umgebung nutzen wollen.

Damit das nicht zu abstrakt wird, hier ein paar praktische Beispiele: Einige Sprachen, wie z.B. Visual Basic, kennen keine vorzeichenlosen Typen. Ein Integer in VB ist ein `int` in C# oder C++, das Konzept eines `uint` ist VB fremd. Daher ist für eine CLS-kompatible Klasse gefordert, für alle nach außen sichtbaren Variablen nur vorzeichenbehaftete Datentypen zu verwenden.

Als zweites Beispiel dient die Unterscheidung von Groß- und Kleinschreibung bei vielen Programmiersprachen. Ob ich in Java oder C++ eine Methode `Machwas` oder `machWas` nenne, ist ein großer Unterschied, schließlich handelt es sich bei diesen Sprachen um zwei verschiedene Methoden. Auch hier ist VB wieder wesentlich toleranter: Es gibt keinen Unterscheid zwischen Groß- und Kleinschreibung bei Bezeichnern. Daher ist für die Kompatibilität mit der CLS Voraussetzung, dass keine zwei Bezeichner im gleichen Gültigkeitsbereich vorkommen, die sich nur durch ihre Schreibweise unterscheiden.

Natürlich können Sie in C# oder C++ zwei Variablen `int Zahl` und `uint zahl` verwenden und damit eine perfekte .NET-Anwendung entwickeln. Allerdings ist dann eben nicht sichergestellt, dass eine Klasse,

die beide Variablen veröffentlicht, mit jeder anderen Sprache der .NET-Welt benutzt werden kann. Wie Sie später noch sehen werden, gibt es Möglichkeiten in C#, Programmelemente entsprechend zu markieren, so dass Sie der Compiler auf eine eventuelle Nichteinhaltung der CLS-Spezifikationen hinweist.

2.2.4 Die .NET-Laufzeitumgebung (CLR)

Um für die nachfolgenden Ausführungen eine Basis zu schaffen, müssen wir das .NET-Framework wie ein Bergsteiger von unten nach oben durchlaufen. Beginnen wir also unsere .NET-Expedition im Basiscamp am Fuße der .NET-Laufzeitumgebung.

Spricht man von der .NET-Laufzeitumgebung, meint man oft das komplette Framework. Allerdings wird unterschieden zwischen dem, was für eine .NET-Anwendung minimal vorhanden sein muss und welche Teile optional sind. Beispiel: Für ein »Hallo, Welt« als .NET-Anwendung benötigen Sie weder ADO.NET noch Web Services oder XML. Die grundlegende Infrastruktur aber wird benötigt und diese finden Sie in der CLR. Die folgende Abbildung listet die wichtigsten Bestandteile und Aufgaben der CLR auf.

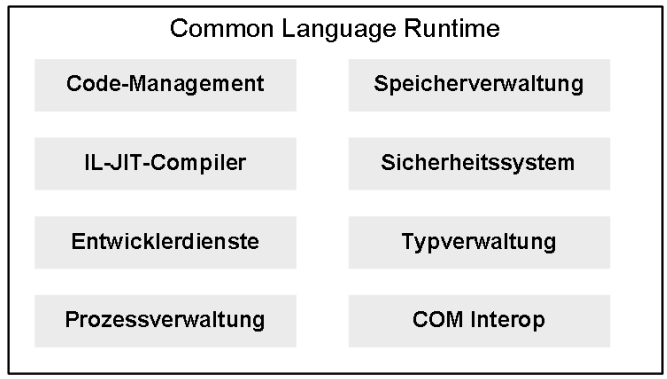


Abbildung 2.4: Aufgaben der Common Language Runtime

2.3 .NET aus Anwendungssicht

Der folgende Abschnitt betrachtet die .NET-Laufzeitumgebung aus der Sicht einer Anwendung, von der Übersetzung des Quellcodes bis zum Beenden der Anwendung.

2.3.1 Übersetzung

Wenn Sie Code in eine .NET-Programmiersprache übersetzen lassen, erstellt der Compiler eine Datei, die den IL-Code und die Metadaten (das Manifest) enthält. Wenn Sie möchten und die CLS einhalten, können Sie die Anwendung auch mit mehr als einer Programmiersprache erstellen. Nach der Übersetzung liegt die Anwendung als ein oder mehrere Assemblies vor. Ein Assembly enthält den Einsprungpunkt der Anwendung, die anderen sind meist (aber nicht notwendigerweise) als Bibliotheken (mit der Erweiterung DLL) abgelegt. Diesen Ablauf verdeutlicht die folgende Abbildung.

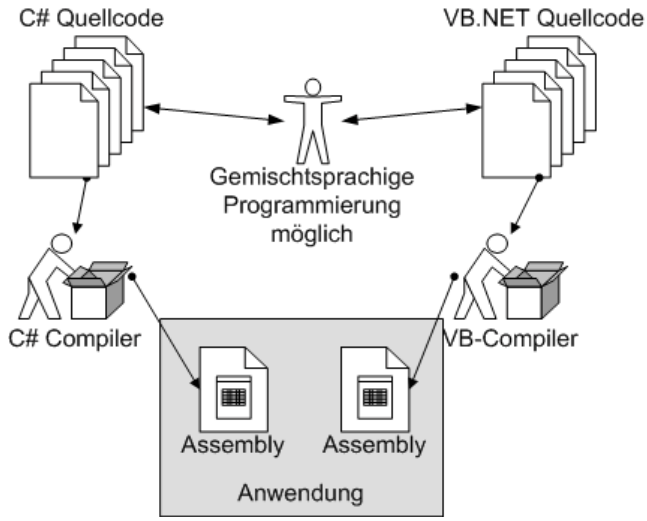


Abbildung 2.5: Erstellung einer .NET-Anwendung

2.3.2 Laden und Ausführung

Bei der Ausführung eines Assemblys wird nach der Lokalisierung des Assemblys eine Sicherheitsüberprüfung durchgeführt, ob der Zugriff erlaubt ist, der Code im derzeitigen Sicherheitskontext ausgeführt werden darf. Diese Überprüfung schließt z.B. einen Check des Speicherverhaltens des Systems ein. Wurde beispielsweise unsafe code benutzt, also in C# mit Zeigern außerhalb der Kontrolle der .NET-Laufzeitumgebung programmiert, hängt es von den Sicherheitseinstellungen des Systems ab, ob der Code ausgeführt werden darf oder nicht.

Diese Sicherheitseinstellungen können von einem Programm auch bereits im Manifest verzeichnet oder bei Start des Programms angefordert werden, so dass bei unzureichenden Rechten (z.B. kein Zugriff auf das lokale Filesystem) eine Ausführung erst gar nicht gestartet wird.

Anschließend wird, wenn nicht bereits im Cache des Systems eine native-code Überführung des Assembly vorliegt, mit Hilfe eines Just-in-time-Compilers die Umsetzung in den Maschinencode vorgenommen und das Assembly in eine sog. Application Domain geladen und ausgeführt. Hier wird dann der Startpunkt der Anwendung angesprungen und der Code ausgeführt.

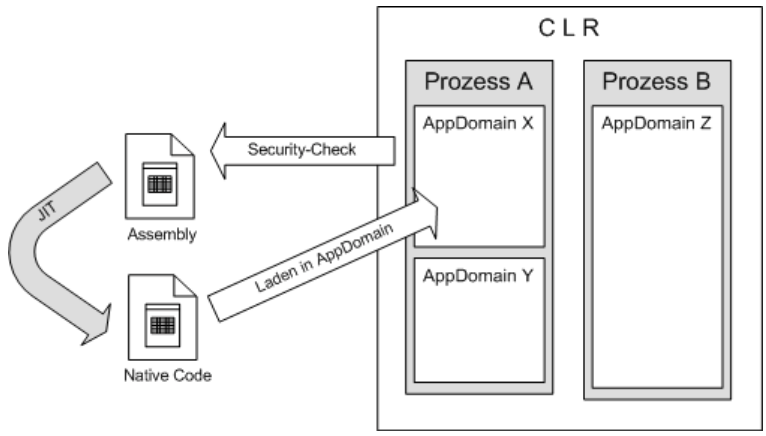


Abbildung 2.6: Security-Check und Laden eines Assemblys

2.3.3 Ressourcenverwaltung

Wird während der Laufzeit eine Ressource benötigt, also Speicher angefordert, weist die .NET-Laufzeitumgebung der Anwendung den Speicher zu und prüft in regelmäßigen Abständen, welche Referenzen nicht mehr erreicht werden können und damit nicht mehr benötigt werden. Diese werden automatisch wieder in den Speicherpool der .NET-Laufzeitumgebung zurückgegeben. Diese sog. garbage collection ist hier stark vereinfacht geschildert, mehr Details erfahren Sie im Kapitel zu diesem Thema später im Buch.

Wird Code aus einem anderen Assembly oder einer Bibliothek benötigt, überprüft die .NET-Laufzeitumgebung die Versionseinstellungen des Assemblys aus dem Manifest und stellt fest, ob das Assembly mit den vorliegenden Versionen kompatibel ist. Anschließend wird ebenfalls für dieses Assembly eine Sicherheitsüberprüfung durchgeführt. Anschließend wird der Code ausgeführt. Diese durchgängige Sicherheitsüberprüfung verhindert auch, dass ein Stück bösartiger Code mit unzureichenden Rechten eine Systemfunktion aufruft und diese die Aktion ausführen lässt.

2.4 .NET aus Entwicklersicht

Für den Entwickler stellt sich bei jedem neuen System die Frage, wie man die Möglichkeiten und Grenzen des Systems möglichst schnell in den Griff bekommt. Das .NET-Framework, mit mehr als 7.000 Klassen, über 150.000 Methoden und Eigenschaften ist nichts, was sich an einem verregneten Herbstwochenende schnell mal ausprobieren lässt. Allerdings ist der Einstieg auch leichter, als Sie nach diesen Zahlen vielleicht annehmen.

Nachdem viele Funktionen der .NET-Laufzeitumgebung nicht direkt mit der Sprache C# zu tun haben, eine Beschäftigung mit C# aber ohne die Nutzung der Infrastruktur der .NET-Laufzeitumgebung nicht sinnvoll möglich ist, möchte ich Ihnen an dieser Stelle eine kleine Orientierungshilfe geben.

Nachdem der Autor auch nach einem Jahr Beschäftigung mit der .NET-Laufzeitumgebung immer noch versucht, die 7.000 Klassen auswendig zu lernen, um nicht bei den seltener besuchten Namensräumen

dauernd mit einem Finger der linken Hand über der F1-Taste zu schweben, wächst der Wunsch nach einer großen Landkarte des gesamten Frameworks. Sollten Sie eine Idee zu einer sinnvollen Realisierung haben oder spezielle Wünsche, senden Sie mir eine Email. Zusammen mit anderen Autoren ist eine Art ».NET Enzyklopädie für erfahrene Entwickler« in Planung, mal sehen, ob man das Framework in handliche Brocken zerteilen kann.

2.4.1 Ein klarer Schnitt?

Was lief falsch bei der bisherigen Entwicklung von Windows-Anwendungen? Wieso war die Entwicklung eines komplett neuen Frameworks überhaupt notwendig? Betrachtet man die Windows-Programmierung seit den Zeiten des ersten SDK (wer erinnert sich nicht an den Klassiker von Charles Petzold zur Windows-Programmierung mit dem SDK?), so fällt auf, dass Microsoft seit diesen Zeiten das Programmiermodell beständig weiterentwickelt und aufgestockt hat. Bis zu einem gewissen Punkt ist so etwas sehr sinnvoll, sichert es doch die Investition der Entwickler in das Gelernte. Irgendwann wird der Ballast der erzwungenen Kompatibilität und der nicht mehr benötigten Techniken aber so groß, dass die ursprüngliche Konsistenz eines Programmiermodells verloren geht.

Ich habe in einem anderen Buch zu C# ein schönes Beispiel für diesen Sachverhalt gefunden: Falls Sie bisher in C++ programmieren, schreiben Sie einfach mal alle Möglichkeiten auf, in einem Windows-Programm einen String zu deklarieren. Nach einem halben Dutzend Möglichkeiten, die ich hier gar nicht aufzählen möchte, haben Sie immer noch nicht alle gefunden. Jetzt stellen Sie sich ein Programmiermodell und eine .NET-Programmiersprache vor, bei der ein String einfach nur ein `string` ist – immer. Wäre das nicht einfacher?

Das ist nur ein Beispiel dafür, warum Microsoft erkannt hat, dass es Zeit für ein Umdenken in der Softwareentwicklung wurde. Die Zeit der API-Aufrufe und Zeigeroperationen ist in einer OOP-Welt einfach vorbei, ebenso der Ausschluss eines großen Teils der Softwareentwickler durch unzureichende Möglichkeiten der Programmiersprache (sehen Sie sich bei einem VB-Kollegen einmal an, was teilweise für Verrenkungen nötig sind, um bestimmte Funktionalitäten von Windows zu erreichen). Als Einstieg können Sie sich .NET für Entwickler

einfach als eine neue Methode vorstellen, Windows-Programme zu erstellen: mit einem komplett objektorientierten Programmiermodell und einem sprachübergreifenden Typsystem.

Natürlich meint die Programmierung für die .NET-Laufzeitumgebung nicht nur Windows, aber solange die .NET-Laufzeitumgebung nicht auf eine zweite Plattform portiert wurde, bleibt der Unterbau eben Windows.

Dazu kommt noch, was im Kapitel ***C# Sightseeing Tour*** bei den Programmiersprachen erwähnt wird: C++ besitzt mittlerweile so viel Ballast und so viele Möglichkeiten für Flüchtigkeitsfehler, dass die Sprache einfach nicht mehr zeitgemäß ist (bedenken Sie, dass C++ als eine der ersten OOP-Sprachen zu Beginn der 80er Jahre entstand und zwischen den Konzepten von C++ und C# 20 Jahre Weiterentwicklung bei den Computersprachen und der Softwareentwicklung liegen).

2.4.2 Die Bequemlichkeit der Infrastruktur

Ein weiterer Vorteil von .NET für Entwickler ist die enorme Fülle an Funktionalität für grundlegende Dinge, die einem zur Verfügung gestellt werden. Dies erleichtert einfach die Arbeit und Sie müssen nicht jedes Mal das Rad von neuem erfinden. Wer beispielsweise programmiert heute noch eigene DATEI ÖFFNEN-Dialogboxen für Windows? Die Common Controls enthalten ein Standard-Dialogfeld, mit dem die Benutzer vertraut sind und das sie einfach mit den passenden Einstellungen aufrufen. Hier kann die .NET-Laufzeitumgebung einem Entwickler viel Arbeit abnehmen, von der automatischen Auswertung von Konfigurationsdateien für Anwendungen bis zu vordefinierten Klassen für Listen, Stacks, Queues und vielem mehr.

Betrachten Sie daher die .NET-Laufzeitumgebung einfach als riesengroßen Werkzeugkasten, bei dem Sie sich nach Bedarf bedienen können und von dem Sie zu Beginn auch nicht alle Werkzeuge bis ins letzte Detail kennen müssen. Microsoft hat den Einstieg durch das Konzept der Namensräume leicht gemacht, so dass Sie die gesuchte Funktionalität relativ leicht finden.

2.4.3 Die Namensräume

Die Gliederung in verschiedene, hierarchisch geordnete Namensräume erlaubt eine erste Orientierung. In Abbildung 2.7 finden Sie eine Art Landkarte der Namensräume, so dass Sie die einzelnen Blöcke der Funktionalität einordnen können.

Falls Sie nach dem Studium dieses Diagramms der Meinung sind, eine neue Brille zu benötigen, Sie finden eine Grafikdatei mit diesem Diagramm auf der beiliegenden CD zum Ausdruck in der gewünschten Größe.

2.4.4 Wo anfangen?

Keine leichte Frage, aber immerhin gibt es einige Einstiegspunkte, die Sie sich unabhängig von speziellen Interessen oder Aufgabenstellungen ansehen sollten. Dieser Abschnitt soll Ihnen als Vorschlag für eigene Erkundungen während des Kennenlernens der .NET-Laufzeitumgebung dienen, mit zunehmender Erfahrung werden Sie später sowieso die für Sie interessanten Klassen in der Online-Dokumentation nachlesen.

- `System`

  Die Klassen in diesem Namensraum bilden die essentielle Basis für die Arbeit mit der .NET-Laufzeitumgebung. `System` sollten Sie wirklich in der Online-Dokumentation intensiv durchlesen, denn hier finden Sie neben den grundlegenden Klassen auch die Datentypen und Strukturen, die an allen anderen Stellen der .NET-Laufzeitumgebung verwendet werden.

- `System.Collections`

  Falls Sie Daten strukturiert im Speicher ablegen möchten, sollten Sie einen Blick auf die Klassen dieses Namensraums werfen. Von Auflistungen über Warteschlangen, Stacks, Hashtabellen und Dictionaries ist hier alles, was Sie zu diesem Thema brauchen werden.

- `System.Data.*`

  Der Namensraum System.Data und die darunterliegenden Namensräume decken den gesamten Bereich von ADO.NET ab.

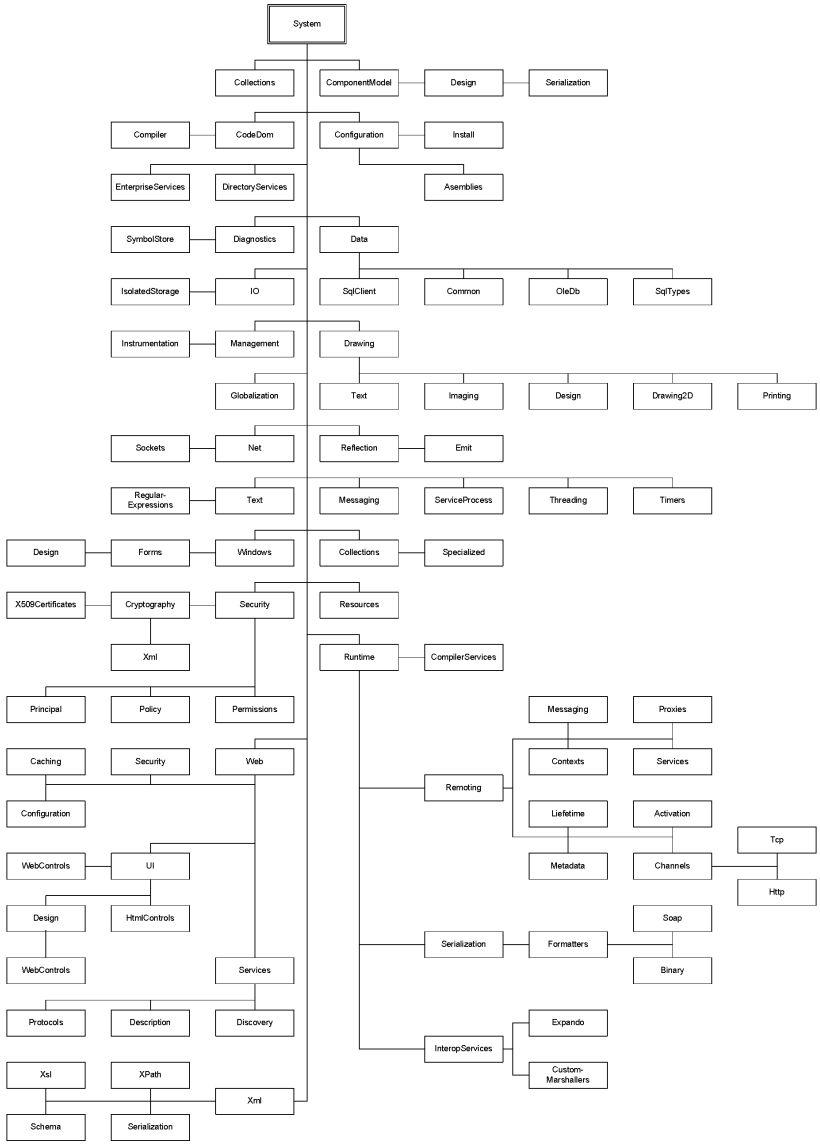


Abbildung 2.7: Die Hierarchie der Namensräume der .NET-Laufzeitumgebung

- `System.Diagnostics`

  Hier finden sie die Debug- und Trace-Klassen, die Kommunikation mit dem Debugger und andere Dinge, die im Kapitel ***Fehlersuche*** behandelt werden.

- `System.Drawing.*`

  Die Namensräume für GDI+, die neue Grafikschnittstelle von Windows.

- `System.IO`

  In diesem Namensraum befinden sich die Klassen für die Ein- und Ausgabe über Streams. Diese Klassen sollten Sie sich mit als Erstes ansehen, um mit dem streamorientierten Konzept der Ein- und Ausgabe bei .NET vertraut zu werden.

- `System.Reflection.*`

  Die Klassen in diesem Namensraum stellen Möglichkeiten zu einer sehr weitgehenden Typuntersuchtung zur Laufzeit sowie der dynamischen Erzeugung von Datentypen und Assemblies zur Laufzeit zur Verfügung. Wenn Sie untersuchen wollen, welche Methoden fremde Klassen zur Verfügung stellen, ist dies der Ansatzpunkt. Tools wie WinCV benutzen die Klassen in diesem Namensraum intensiv.

- `System.Runtime.Serialization.*`

  Hier finden Sie Klassen zur Serialisierung von Objekten, also der Speicherung auf einem persistenten Datenträger, so dass Sie später aus diesem Speicher die Instanz der Klasse erneut erzeugen können.

- `System.Text / System.Text.RegularExpressions`

  Die `StringBuilder`-Klasse zur Stringbearbeitung findet sich hier ebenso wie Klassen zum Konvertieren von Daten in textuelle Darstellung, außerdem eine sehr mächtige Klasse mit einer vollständigen Implementierung von regulären Ausdrücken.

- `System.Web`

  Dieser Namensraum bietet Klassen zur Kommunikation mit einem HTTP-Server, der Verwaltung von Cookies und dem Transfer von Dateien. Zusammen mit den darunter liegenden Namensräumen

finden Sie hier auch die Funktionalität der WebForms und ASP.NET.

- `System.Windows.Forms`

  Falls Sie GUI-Anwendungen für den Desktop entwickeln, sollten Sie sich die Klassen in diesem Namensraum genau ansehen, hier sind die Steuerelemente für die neuen Windows Forms definiert.

- `System.Xml.*`

  Dieser und die darunter liegenden Namensräume beinhalten die Unterstützung der .NET-Laufzeitumgebung für XML-Dateien und die Serialisierung von Objekten in XML-Dateien.

2.5 Zusammenfassung

Dieses Kapitel kann naturgemäß nur an der Oberfläche der .NET-Laufzeitumgebung kratzen und einige grundlegende Sachverhalte erläutern, soweit sie für den Einstieg in C# benötigt werden. In den einzelnen Kapiteln werden die entsprechenden Klassen und Strukturen dann detaillierter vorgestellt.

Um sich in der .NET-Laufzeitumgebung zurechtzufinden, sollten Sie sich mit den Werkzeugen vertraut machen, die Sie für diese Aufgabe benötigen. Im Anhang finden Sie eine kurze Vorstellung der wichtigsten Tools, zusätzlich sollten Sie auf jeden Fall bei den ersten Schritten immer die Online-Dokumentation des .NET-Framework SDK offen haben.

Für das nächste Kapitel lade ich Sie zu einer Sightseeing-Tour durch C# ein …

3 C# Sightseeing Tour

Wenn ich auf dem Weg zu meinem Arbeitsplatz in München aus dem Bahnhof komme, sehe ich immer wieder eine Reihe von Bussen mit der Aufschrift *München in einer Stunde* auf der gegenüberliegenden Straßenseite warten. Ich habe in dieser Stadt sicherlich bereits Wochen verbracht und würde nicht behaupten, München zu kennen. Dennoch erfüllen diese kurzen Touren ihren Zweck. Sie erhalten eine erste Orientierung, lernen kurz die wichtigsten Plätze kennen und besitzen später die Möglichkeit, interessante Orte wieder aufzusuchen und sich die Details anzusehen.

Dieses Kapitel ist wie eine kurze Stadtrundfahrt durch C#. Sie erfahren die grundlegenden Bestandteile von C#, lernen die wichtigsten Prinzipien kennen und können bei Interesse jederzeit zu einem der vertiefenden Kapitel springen.

3.1 Das Problem

Mit dem Erfolg von Internet und Intranet, verteilten objektorientierten Anwendungssystemen und der Programmiersprache Java wurde den führenden Köpfen der Firma Microsoft schnell klar, dass sich die Anforderungen an die Softwareentwicklung erneut verschieben würden. Es begann ein strategisches Umdenken in Richtung modularer, komponentenbasierter Anwendungen, welche das Web nutzen konnten.

Die Plattform für ein solches System musste eine Infrastruktur bereitstellen, welche möglichst viele der bekannten Probleme für Softwareentwickler lösen konnte: bessere Produktivität, stabilere Software, schnellere Entwicklung, echte Wiederverwendbarkeit. Solche generellen Verbesserungen sind eine nette Umschreibung für die realen, täglichen Probleme eines Entwicklers: fehlende Synchronisierung zwischen den Headerdateien und der Implementierung, zusätzliche Dateien für Typbibliotheken, das Windows DLL-Chaos, der Kampf mit der Registry und die oft nur theoretisch vorhandene Möglichkeit zur Entwicklung mit mehreren Programmiersprachen.

Stellen Sie sich ein typisches Szenario zur Entwicklung einer Internet-Anwendung vor: Wir hätten da C++ für die Systemsoftware und COM-Komponenten, die Oberfläche bauen wir mit Visual Basic, eine Datenbank-DLL wurde mit Delphi erstellt, wir benutzen ADO, SQL, HTML, XML für die Serialisierung der Daten, natürlich gibt es Konflikte mit der Namensgebung von Bezeichnern im Team, ein armer Tester muss die VB-Runtime und die MFC-DLLs sowie ein halbes Dutzend anderer DLLs unter einen Hut bekommen (und wehe, eine Version ändert sich) – ich denke, Sie wissen, worauf ich hinaus will ...

Das .NET-Framework von Microsoft bietet eine Lösung für beinahe alle der oben geschilderten Probleme, da hier zum ersten Mal die Infrastruktur der Laufzeitumgebung ein komponentenbasiertes System darstellt. Einzelne Bausteine (die sog. Base Classes, später mehr dazu) bieten klar definierte Funktionalität, welche von jeder .NET-fähigen Sprache genutzt werden kann. Weitere Details entnehmen Sie bitte dem vorigen Kapitel ***Microsoft .NET***.

3.1.1 Ein Werkzeug für .NET

Alle verfügbaren Programmiersprachen haben ihre Vor- und Nachteile. C++ ist schnell und Sie können damit alles programmieren, was Sie je brauchen. Allerdings ist C++ aufgrund der historischen Entwicklung nicht unbedingt eine leicht erlernbare und produktive Sprache. Visual Basic ist leicht erlernbar und bei grafischen Frontends erhalten Sie relativ schnell Ergebnisse. Ich würde aber damit kein großes System mit Infrastrukturaufgaben entwickeln. Java, modern und objektorientiert, besitzt immer noch zwei Nachteile: Performance und Ressourcenbedarf sind 'suboptimal' und Sun wacht eifersüchtiger über die Sprache als mancher Drache über seine Prinzessin. Um noch für einen Satz im Märchenland zu verweilen, begab es sich aber nun, dass ein weiser Mann von einer anderen Softwareschmiede zu Microsoft kam, der als Entwickler von schnellen und produktiven Programmiersprachen in der Branche einen großen Ruf hatte. Anders Hejlsberg hatte bei Borland nach Turbo Pascal mit Delphi die einzige ernste Alternative der Windows-Welt zu Visual C++ und Visual Basic entwickelt und grübelte mit anderen nun für Microsoft an einer Sprache für das .NET-Framework.

Wie immer, wenn man sich nicht für eine Sache entscheiden kann, sah die Lösung so aus, dass das Team das Beste aus allen Welten nahm (mit einem deutlichen Schwerpunkt auf C++ und Java) und zusammen mit eigenen Ideen daraus eine neue Programmiersprache entwickelte, unbelastet von Kompatibilitätszwängen. C# war geboren.

3.2 Designziele von C#

3.2.1 Produktivität

Obwohl es banal klingt, ist eine der größten Lernerleichterungen bei der Benutzung von C# unter anderem der Ansatz des sog. 'one-stop coding'. Der gesamte Quellcode und alle Metadaten befinden sich in einer einzigen Datei. Wer einmal versucht hat, als Einsteiger den Haufen von Dateien für ein Projekt unter C++ in den Griff zu bekommen, wird C# auf Anhieb mögen.

Die Lernkurve für Entwickler, die von anderen modernen Sprachen kommen, ist relativ flach. C# bietet aufgrund der Herkunft eine weitgehende Orientierung an der Syntax von C/C++, ohne deren kryptische Details zu übernehmen. Das Design der Sprache ist wesentlich schlanker und eleganter als C++, gleichzeitig aber auch wesentlich strukturierter als Visual Basic. Wenn Sie von Java oder Delphi kommen, werden Sie sich in C# sofort heimisch fühlen. Im Gegensatz zu Java erlaubt C# den Zugriff auf andere APIs und das System und bietet damit wesentlich mächtigere Möglichkeiten durch den Wegfall des Zwangs zur Beschränkung auf eine Sprache. Das Entwicklungsteam von C# bietet Delphi-Programmierern einen wesentlich weniger gängelnden Compiler, der aber die häufigsten Fehler dennoch verhindert. Mehr dazu im nächsten Abschnitt, wo Sie einige dieser Möglichkeiten kennen lernen.

3.2.2 Stabiler und sicherer Code

Eine Programmiersprache, die vor allem auch für die Komponentenentwicklung dient, muss den Entwickler aktiv bei der Produktion von stabilem und sicherem Code unterstützen. Serversysteme in Produktionsumgebungen laufen nonstop mit vielleicht zwei oder drei Tagen Wartungsstop pro Jahr. Da ist jeder nicht geplante Neustart des Servers schnell eine entsprechende finanzielle Belastung für das Unternehmen.

C# unterstützt die Erstellung von stabilem Code durch eine Reihe von Merkmalen:

- Striktes Typkonzept, welches z.B. unsichere Typumwandlungen verhindert und so 'Spezialitäten' wie in vielen C++ Programmen verhindert.
- Variablen dürfen nur dann benutzt werden, nachdem ihnen ein Wert zugewiesen wurde. Dies verhindert die Verwendung nicht initialisierter Variablen.
- Arithmetische Berechnungen können im checked-Modus auf Überlauf geprüft werden. So werden Probleme durch zu klein dimensionierte Datentypen zu einem Teil verhindert.
- Die automatische Speicherverwaltung (garbage collection) entlastet den Entwickler durch den Wegfall von Programmcode zur Speicherverwaltung. Ebenso kann die Testzeit, die bis jetzt zur Suche nach Speicherlecks aufgewendet wurde, für sinnvollere Arbeiten (oder sogar Freizeit?) verwandt werden.

Stabiler Code allein reicht für eine gute Anwendung oder Komponente nicht aus, wenn nicht sichergestellt ist, dass der stabile Code auch der ist, den Sie ausführen lassen wollen (jedes Opfer eines stabil laufenden trojanischen Pferdes kann ein Lied davon singen). Daher bietet C# ebenfalls Unterstützung für die Codesicherheit, beispielsweise durch die kryptografische Signatur von gemeinsam genutzten Komponenten, die Rechtevergabe für die Codeausführung aufgrund der Herkunft des Codes (lokal, Internet).

3.2.3 Praxistauglichkeit

C# ist keine im Elfenbeinturm entstandene Programmiersprache. Der größte Teil des .NET-Frameworks wurde in C# geschrieben und bereits in der Beta-Phase wurden von vielen Entwicklern bei Microsoft und in anderen Firmen auf diese Weise Millionen von Codezeilen getestet. Die Sprache bietet in einer konsistenten Umgebung alles, was in der Praxis für eine erfolgreiche Entwicklung nötig ist, ohne neuere Erkenntnisse der Softwareforschung zu vernachlässigen.

Praxistauglichkeit bedeutet aber auch den Schutz bestehender Fähigkeiten. Niemand benutzt eine neue Programmiersprache, wenn alles, was im Unternehmen bis dato erstellt wurde, auf einmal virtuelles Altmetall ist. Falls es notwendig sein sollte (und in der Übergangszeit wird es ganz sicher notwendig sein), erlaubt C# den problemlosen Zugriff auf COM-Komponenten. In C# erstellte .NET-Komponenten lassen sich ohne Probleme von COM-Clients aus aufrufen und auch die von vielen so geliebten Pointer sind nicht verbannt (wenn auch eingeschränkt). Sie sind in der Lage, DLLs zu importieren, das Windows-API und all die kleinen Tricks zu nutzen, auf die Entwickler von Zeit zu Zeit zurückgreifen.

3.2.4 Modularität

Kaum ein Projekt in der Softwareentwicklung wird noch von einem einzigen Entwickler durchgeführt. In der Vergangenheit war die reine Koordinierung der vielen Include-Dateien bei C++-Projekten oft schon ein Stolperstein. Schließlich liegt der Quellcode für die Textverarbeitung, mit der dieses Buch entsteht (Sie ahnen, welche?), nicht in einer Datei mit 100 MByte. Auch hier erleichtert C# den Entwicklern die Arbeit, da alle Informationen für die Übersetzung in der Quellcodedatei enthalten sind. Keine Includes, keine IDL-Dateien, keine Header, alles ist im Quellcode über entsprechende Sprachmöglichkeiten gelöst. Eine Klasse liegt in einer Datei und nutzt Code aus anderen Klassen. Natürlich können Sie mehr als eine Klasse in einer Datei erstellen, allerdings können Sie eine Klasse nicht über mehrere Dateien veteilen. Zusammen mit dem Konzept der Namensräume, die sich über mehr als eine Datei und damit mehr als ein Assembly erstrecken können, haben Sie damit flexible Möglichkeiten, auch große Codeprojekte sinnvoll zu strukturieren.

3.2.5 Nutzung aller .NET-Möglichkeiten

C# ist die Programmiersprache, mit der beinahe das gesamte .NET-Framework und die Basisklassen erstellt wurden. Natürlich ist jede andere .NET-kompatible Sprache (von Visual Basic.NET bis zu Fujitsu COBOL) in der Lage, die Möglichkeiten beispielsweise des gemeinsamen Typsystems zu nutzen. Allerdings wurde aufgrund der Historie der einen oder anderen Programmiersprache oftmals eine kleine Korrektur

notwendig oder bestimmte Konzepte aus der Sprachspezifikation stehen nicht zur Verfügung (wenn eine Sprache keine Eigenschaften als Bestandteil ihrer Grammatik unterstützt, können diese in dieser Sprache auch nicht verwendet werden).

Ein Großteil der .NET-Laufzeitumgebung wurde mit C# erstellt und das C#-Team hat hausintern bei Microsoft bereits Millionen von Codezeilen gesehen. Obwohl es Dinge gibt, welche Sie in C# nicht tun können (die in managed C++ möglich sind), sind diese Sachen genau die Relikte aus 'alten Zeiten', die Sie mit .NET auch nicht mehr benötigen werden.

3.2.6 Echte Objektorientierung

In C# ist jeder Datentyp ein Objekt und stammt von einer gemeinsamen Basisklasse `object` ab. Sogar die einfachen, skalaren Datentypen wie `int` können über einen Typcast zu Objekten umgewandelt werden. Dies geschieht allerdings ohne den in anderen Systemen wie Smalltalk üblichen Overhead durch einen Mechanismus, der sich boxing bzw. unboxing nennt und den ich in der Sprachreferenz genauer erläutere.

Die Unterstützung der Komponentenentwicklung bietet C# vor allem auch durch eine bessere Definition von überschriebenen Methoden in Klassen. Über entsprechende Konstruktionen lassen sich die Versionierung von Methoden und die Vererbung gezielter als beispielsweise in C++ steuern.

Moderne, objektorientierte Konstruktionen wie Schnittstellen, Eigenschaften und typsichere Funktionszeiger (*delegates*) sind integraler Bestandteil der Sprache und sorgen für eine konsistente OOP-Umgebung. Alles in C# ist Bestandteil einer Klasse, es gibt keinen herumliegenden prozeduralen Code oder solche Dinge wie ein Standard-Codemodul in Visual Basic.

3.2.7 Komponentenorientierte Entwicklung

Die Geschichte der Programmiersprachen dauert jetzt nahezu ein halbes Jahrhundert und langsam beginnen Entwickler zu verstehen, welche Konzepte die Erstellung moderner Softwaresysteme erleichtern und die Produktivität erhöhen. Wie überall stellen sich viele Erkennt-

nisse erst in der täglichen Praxis heraus. Eine Reihe von Sprachen hat es trotz akademisch höchst niveauvoller Ideen nie in die breite Masse der Entwickler geschafft (wie viele Ihrer Kollegen schreiben in Smalltalk?). Einzelne Fähigkeiten dieser Sprachen und Konzepte der Softwareentwicklung wurden nach und nach in bestehende Programmiersprachen und Entwicklungssysteme übernommen.

Nachdem ich für keinen Hersteller einer Programmiersprache arbeite, kann ich ohne den Vorwurf der Einseitigkeit ein weiteres Designziel angeben, das die Entwickler von C# meiner Meinung nach im Auge hatten: Einfachheit und Konsistenz.

Für alle passionierten Fotoamateure unter den Softwareentwicklern: C++ ist eine dieser vor einigen Jahren noch modernen »Display und Knöpfchen«-Spiegelreflexkameras, die vor dem ersten Foto erst über einen Touchscreen programmiert werden mussten und vor lauter Funktionalität kaum noch Bilder machen konnten. C# dagegen ist wie meine geliebte EOS-50, Hebel statt Touchscreen, Retro-Design ohne Ballast, aber mit allen Möglichkeiten, die wirklich sinnvoll sind.

C# ist in gewisser Hinsicht die Rückkehr zu den Wurzeln der objektorientierten Programmierung. Was C++ versprochen und Java teilweise gehalten hat, wird mit C# tatsächlich möglich: eine wirklich komponentenorientierte OOP-Sprache mit einer von Ballast befreiten Syntax. Weg von Bitlevel-Details (C++ war für eine echte OOP-Sprache viel zu tief angesiedelt mit all den Zeigern, Relikten aus der C-Ära), wartbar (eine Klasse in einer Datei, keine Includes) und ausgerichtet auf schnittstellenbasierte Komponentenerstellung. Die Syntax und Semantik der Sprache sind straff genug, um die üblichen Flüchtigkeitsfehler von Entwicklern zu verhindern, lassen aber im Gegensatz zu anderen Sprachen genügend Freiraum für die Ausdrucksmöglichkeiten der Entwickler.

Man mag über bestimmte Details der .NET-Laufzeitumgebung geteilter Meinung sein, aber ich denke, die Sprache C# ist eine sehr gut gelungene Entrümpelung von C++ mit einem Schuss Java und Delphi. Natürlich fehlen mir auch bei C# immer noch ein paar Dinge (warum zum Beispiel ist nicht wie in Eiffel eine Sprachunterstützung für Design by Contract enthalten, das hätte der Qualität der entstehenden Programme sicher gut getan), aber insgesamt gesehen mag ich C#.

3.3 C# im Überblick

3.3.1 Wofür C#?

Viele Programmiersprachen besitzen eine Anwendergemeinde aus einem relativ begrenzten Einsatzgebiet. So werden mathematische Anwendungen teilweise immer noch mit Fortran geschrieben, Systemprogrammierung unter Unix erfolgt mit C, der GUI-Teil vieler Windows-Anwendungen ist zum großen Teil fest in der Hand von Visual Basic. Es gibt sogar Programmiersprachen, welche extra für einen ganz bestimmten Einsatzzweck in Forschung und Industrie entworfen wurden und dort eine außerhalb dieses Bereichs kaum wahrgenommene Nische besetzen (oder kennen Sie das ursprüngliche zur Steuerung von Radioteleskopen entwickelte FORTH?). Daher stellen Sie sich völlig zu Recht die Frage, was denn alles mit C# entwickelt werden kann. Die einzige Voraussetzung ist das Vorhandensein der .NET-Laufzeitumgebung, die (noch) auf die Windows-Welt beschränkt ist. Ansonsten sind Ihrer Kreativität keinerlei Grenzen gesetzt. Anstatt nun hunderte von möglichen Anwendungsszenarien aufzuzählen, möchte ich lieber den Ansatz von Visual Studio.NET aufgreifen und Ihnen die möglichen Einsatzgebiete anhand einer Einordnung in Anwendungskategorien schildern.

Kommandozeilen-Programme

Hier handelt es sich um Programme, die Sie in einem Kommandofenster des jeweiligen Windows-Betriebssystems ausführen, analog zu Programmen wie dem MS-DOS Editor EDIT.EXE oder dem Programm FDISK.EXE. Haupteinsatzgebiet solcher Anwendungen sind der Bereich von Hilfsprogrammen oder Programmen für Administratoren (ja, es gibt Leute, die können Windows auch ohne Maus bedienen).

GUI-Programme

Dies sind die 'klassischen' Windows-Anwendungen, wie sie sich die meisten Anwender vorstellen, von Freecell bis zu Word für Windows. Hier werden Sie im Verlauf des Buchs erfahren, warum C# die ideale Mischung aus der Mächtigkeit von C++ und der Produktivität von Visual Basic mit der Modernität von Java darstellt. Das Stichwort für diese Art von Anwendungen heißt Windows Forms und Sie werden später noch eine ganze Menge darüber erfahren.

ASP.NET-Anwendungen

Konnten Sie vor der .NET-Zeit Active Server Page-Skripte fast nur mit VBScript oder JavaScript erstellen (einige wenige Entwickler nutzten zusätzliche Script-Engines wie ActivePerl), so bietet ASP.NET, die Nachfolgeversion, die Auswahl aus jeder verfügbaren .NET-Sprache. Der Code wird, gleich für welche Sprache, auch kompiliert und nicht mehr interpretiert, so dass die Leistung einer solchen Anwendung wesentlich höher liegt.

.NET-Komponenten

Komponenten sind nichts anderes als in DLL kompilierte Klassen und Strukturen, welche von anderen Anwendungen wie eine Bibliothek benutzt werden können. Der Unterschied zu Steuerelementen liegt darin, dass Komponenten normalerweise nicht über eine Benutzeroberfläche verfügen.

Steuerelemente

Steuerelemente (*controls*) sind im Formulardesigner von Visual Studio.NET verwendbare Komponenten, welche meist über eine Benutzeroberfläche verfügen und zusätzliche Funktionalität zur Verfügung stellen. Diese Elemente können Sie sowohl für die Windows Forms (allein stehende GUI-Anwendungen) als auch für Web Forms (ASP.NET Anwendungen für Browser-basierte Clients) erstellen.

Web Services

Web Services sind über das Internet erreichbare Anbieter von Diensten, also eine Art von Remote-Komponenten mit einer sprach- und plattformunabhängigen Schnittstelle. Damit können Sie eine genau definierte Funktionalität für die Benutzer dieser Web Services anbieten und so Anwendungen erstellen, welche ihren Funktionsumfang aus vielen verschiedenen Stellen im Internet beziehen und nur noch die Koordination erledigen. Im Gegensatz zu am menschlichen Benutzer orientierten Dienstangeboten sind die 'Kunden' der Web Services also selbst wieder andere Anwendungen. Ein Feld für völlig neue Geschäftsideen …

3.3.2 Am Anfang war »Hallo, Welt«

Es gibt Gegenden in diesem Universum, in denen Sie für das folgende Programm mit lebenslangem Programmierverbot bestraft werden. Ich werde allerdings ausgerechnet in meinem ersten Buch keinen Bruch mit der Tradition riskieren, daher also Vorhang auf für »Hello, World«.

```
using System;
// das erste programm in C#

class World
{
    public static int Main()
    {
        Console.WriteLine("Hallo, Welt!!");
        return (0);
    }
}
```

Listing 3.1: Die C#-Version von Hallo, Welt

Zum Test dieses Programms erstellen Sie den Quellcode in einem Texteditor Ihrer Wahl (bei dieser Länge dürfte es sogar noch Notepad tun) und speichern die Datei anschließend als *hello.cs* ab. Rufen Sie nun den C#-Compiler mit der folgenden Kommandozeile auf, um eine ausführbare Datei *hello.exe* zu erhalten.

```
csc hello.cs
```

Starten Sie das Programm und bewundern Sie gebührend die Ausgabe. Das war's! Sie haben gerade das erste Programm in einer neuen Programmiersprache erstellt.

Lassen Sie mich anhand dieses Listings kurz auf einige Aspekte der C#-Syntax eingehen, bevor ich Ihnen die anderen Konzepte der Sprache vorstelle. Wie Sie erkennen, ist die Syntax stark an die Familie der C-artigen Sprachen angelehnt, zu der neben C++ auch Java gehört. Daher werden sich alle Entwickler, die mit einer dieser Programmiersprachen bereits Erfahrungen gesammelt haben, schnell einlesen. Für Umsteiger aus anderen Sprachen wie Visual Basic oder Delphi ist der rein syntaktische Umstieg etwas größer. Sie werden aber schnell merken, dass in C# eine größere Prägnanz als beispielsweise in Visual Basic

möglich ist. Diese Kürze des Ausdrucks geht aufgrund der in C# vorgenommenen 'Entrümpelung' der C-Syntax aber nicht zu Lasten der Lesbarkeit.

Ein weiteres Merkmal dieses ersten Listings ist das bereits im Abschnitt *Produktivität* angesprochene 'One-Stop-Coding'. Alles, was der Compiler für die Übersetzung des Quellcodes benötigt, findet er in dieser einen Datei. Auch bei Projekten, deren Komplexität über 'Hallo, Welt' hinausgeht, benötigen Sie keine Headerdateien, keine IDL-Definitionen, keine Registrierungsdateien oder andere externe Dateien. Diese potentiellen Fehlerquellen sind damit bereits ausgeschlossen.

Abschließend noch der Punkt, der hauptsächlich den Umsteigern von VB und anderen eher prozeduralen Sprachen auffallen dürfte. Auch eine simple Ausgabe im Konsolenfenster ist in eine Klasse integriert. Hier ist C# sehr mit Java verwandt, alles ist objektorientiert. Jeglicher Code liegt in einem eigenen Datentyp (warum ich nicht Klasse geschrieben habe und was Sie anstelle einer Klasse noch benutzen können, erfahren Sie etwas später).

3.3.3 Grundprinzipien von C#-Programmen

Die nachfolgenden Punkte sollen Ihnen eine schnelle Orientierung erlauben, worin sich C# im Vergleich zu Ihrer bevorzugten bisherigen Programmiersprache unterscheidet.

Dateistruktur

C#-Programme bestehen aus einer Quellcodedatei. Wie im letzten Abschnitt geschildert, gibt es keine Headerdateien, Include-Dateien oder zusätzliche Dateien für Typbibliotheken. Der Code ist Bestandteil eines Datentyps (einer Klasse oder einer Struktur). Es gibt keinerlei globalen Code oder Variablen außerhalb einer Klasse. C# ist streng objektorientiert.

C# gehört zur Gruppe der Freiformsprachen. Dies bedeutet, dass Sie den Quelltext so formatieren können, wie Sie es für optimal halten. So lange Sie keine Schlüsselwörter trennen, wird *white space* (der durch Leerzeichen, Einrückungen und Zeilenschaltungen gebildete Leerraum) vom Compiler ignoriert. Im Gegensatz zu Visual Basic können

Sie also den Code so gestalten, dass er für Sie leicht lesbar ist. Irgendwelche Zeilenfortführungszeichen (wie der unselige Unterstrich in VB) sind damit unnötig.

Kommentare

Kommentare in C# existieren in zwei Formen: bis zum Ende der Zeile oder vom Anfangszeichen des Kommentars bis zum Endezeichen des Kommentars. Zwei aufeinanderfolgende Schrägstriche bilden einen Kommentar bis zum Ende der aktuellen Zeile, die Zeichenfolge »/*« leitet einen Blockkommentar ein, der durch die Zeichenfolge »*/« wieder beendet wird.

Als Besonderheit bietet C# die Möglichkeit, XML-Kommentare zu erstellen. Mit deren Hilfe sind Sie in der Lage, aus dem Quelltext eine XML-Datei mit einer Dokumentation Ihrer Klassen zu erstellen, welche die Bedeutungen der einzelnen Routinen und Parameter enthält. In Visual Studio.NET wird Ihnen dies durch eine entsprechende Editorfunktionalität sogar noch erleichtert. Geben Sie dazu einfach drei aufeinanderfolgende Schrägstriche ein.

Codeblöcke

Die grundlegende Einheit für Anweisungen ist wie in C der Block. Dies ist eine durch geschweifte Klammern eingeschlossene Reihe von Anweisungen und Operatoren. Eine einzelne Anweisung benötigt die Klammern nicht. Manchmal wird aber aus Gründen der Lesbarkeit auch eine einzelne Anweisung in geschweifte Klammern gesetzt.

Die Definition von Variablen, Objekten und anderen Strukturen erfolgt nach dem Motto »Wo, Was, Wer«. Zuerst kommt die (optionale) Angabe eines Gültigkeitsbereichs über einen sog. Modifikator. Anschließend folgt der Datentyp, dann der Name der Variable oder der Funktion.

```
public int nCount;
private object[] objArray;
string Name;
```

Listing 3.2: Beispiele für die Deklaration von Variablen

VB / Delphi



Diese Art der Deklaration von Typen ist für C- und Java-Entwickler die 'natürliche' Reihenfolge, für Visual Basic und Delphi genau verdreht. Sollten Sie also zu Beginn Ihrer C#-Laufbahn Fehlermeldungen bekommen, suchen Sie zuerst an dieser Stelle, ob Sie nicht eine Angabe vergessen oder den Typ nach dem Bezeichner angegeben haben.

Bei C# werden Groß- und Kleinschreibung unterschieden. *test* und *Test* sind also zwei verschiedene Bezeichner! Diese Eigenschaft der Sprache wird sogar oft dazu benutzt, einen Datentyp mit einem Großbuchstaben zu deklarieren und für eine Variable dieses Typs dann Kleinbuchstaben zu verwenden. Da dies Programmierfehler aber erleichtert (wie schnell hat man einen Großbuchstaben klein geschrieben), werde ich versuchen, in diesem Buch eine andere Konvention einzuhalten. Mehr dazu im Kapitel 5 ***Daten, Typen, Variablen***.

Sämtliche Kontrollstrukturen besitzen den gleichen allgemeinen Aufbau mit Kontrollstruktur, einen geklammerten Ausdruck, Anweisungsblock. Damit ist zumindest für VB-Entwickler die Welt viel einfacher und konsistenter geworden.

In C# gibt es bedingt durch die Herkunft der Sprache nur Funktionen. Eine Prozedur oder Subroutine wie in anderen Sprachen ist einfach eine Funktion, welche keinen Wert zurückliefert. Wenn eine Funktion keinen Wert liefern soll, dann wird dies durch den Pseudo-Datentyp *void* deklariert. *Void* ist also nur ein Platzhalter für das Nichts (wie philosophisch von C#).

3.3.4 Datentypen in C#

Wert- und Referenztypen

C# unterscheidet zwischen zwei grundlegenden Datentypen: Werttypen (value types) und Referenztypen (reference types). Werttypen werden immer auf dem Stack angelegt und enthalten die tatsächlichen Werte der Variablen. Alle einfachen, skalaren Typen mit Ausnahme von string sind Werttypen. Eine Referenzvariable enthält dagegen eine Referenz auf den Inhalt der Variablen und wird auf dem Heap angelegt. Objekte, Klassen und auch Strings sind Referenztypen.

Aufzählungstypen

Aufzählungen (enumerations) sind ein integraler Bestandteil von C# (ich wundere mich immer noch, welchen Grund Sun wohl hatte, dieses Konzept aus Java zu streichen). Ein Aufzählungstyp ist ein Datentyp, der nur Werte aus der bei der Deklaration aufgeführten Menge akzeptiert. Im Vergleich zu anderen Sprachen ist der Compiler hier auch strenger mit der Typprüfung. Sie können nicht einfach anstelle des Aufzählungstyps eine Zahlenkonstante mit dem Ordinalwert übergeben, sondern nur ein Mitglied Ihrer definierten Aufzählung. Dies macht Code durch die symbolischen Bezeichner leichter lesbar und durch die Typsicherheit auch fehlerfreier.

Felder

Natürlich unterstützt C# auch ein- und mehrdimensionale Felder (arrays). Diese Felder können innerhalb einer Dimension auch verschiedene Längen besitzen, d.h. bei einem zweidimensionalen Feld muss die zweite Dimension nicht überall den gleichen maximalen Index besitzen (obwohl das den Code nicht notwendigerweise leichter verständlich machen wird).

Einfache Datentypen

Wie jede andere Programmiersprache bietet auch C# eine Reihe von vordefinierten Datentypen. Die Tabelle Datentypen listet die einfachen Datentypen und die darunter liegenden .NET-Typen auf. Details zur Implementierung der einzelnen Typen erfahren Sie im Kapitel 5 ***Daten, Typen, Variablen***.

C#-Typ	.NET-Typ	Beschreibung
sbyte	System.Sbyte	8 Bit mit Vorzeichen
byte	System.Byte	8 Bit ohne Vorzeichen
short	System.Int16	16 Bit mit Vorzeichen
ushort	System.Uint16	16 Bit ohne Vorzeichen
int	System.Int32	32 Bit mit Vorzeichen
uint	System.Uint32	32 Bit ohne Vorzeichen
long	System.Int64	64 Bit mit Vorzeichen

Tabelle 3.1: Die einfachen Datentypen in C#

C#-Typ	.NET-Typ	Beschreibung
ulong	System.Uint64	64 Bit ohne Vorzeichen
float	System.Single	Gleitkommazahl einfach genau
double	System.Double	Gleitkommazahl doppelt genau
decimal	System.Decimal	Dezimalzahl mit 28 signifikanten Stellen
char	System.Char	Ein Unicode-Zeichen
bool	System.Boolean	Ein logischer Wert (TRUE | FALSE)

Tabelle 3.1: Die einfachen Datentypen in C# (Forts.)

3.3.5 Kontrollstrukturen & Operatoren

Hier gibt es bei C# eigentlich wenig Neues für Leute mit Erfahrung in C/C++ oder Java. Einzig die `foreach`-Schleife und die Notwendigkeit eines `break` bei jedem `case`-Zweig wären hier aufzuführen. Details zu den einzelnen Möglichkeiten der Ablaufsteuerung und der möglichen Operatoren finden Sie im Kapitel 8 ***Anweisungen und Operatoren***.

3.3.6 Klassen & Interfaces

Die Unterstützung beim Design von Klassen durch die jeweilige Programmiersprache wurde bei C# im Vergleich zu beispielsweise C++ wesentlich erweitert. Gültigkeit, Sichtbarkeit und Zugriff auf Mitglieder einer Klasse lassen sich explizit und in größerem Umfang als bei C++ angeben. Die gesamte Syntax ist wesentlich einfacher und klarer als bei den meisten anderen Sprachen, darüber hinaus übernimmt C# das Konzept von Eigenschaften (properties) als integralen Bestandteil der Sprache.

Durch die Ausrichtung auf komponentenorientierte Programmierung sind die Möglichkeiten zur Versionierung von Klassen besser als in anderen Sprachen. So lassen sich Klassen beispielsweise als `sealed` erklären, d.h. es ist von diesen Klassen keine Ableitung per Vererbung mehr möglich. Durch den Verzicht auf mehrfache Vererbung (*multiple inheritance*) und die strikte Ausrichtung auf schnittstellenbasierte Programmierung werden die Fehlerquellen und die Komplexität solcher Konstrukte in C++ vermieden. C# folgt damit der aktuellen Entwicklung der Theorie zur objektorientierten Softwareentwicklung. Details zu

diesen Konzepten finden alle Umsteiger von VB und anderen nicht rein objektorientierten Sprachen im Kapitel 6 ***OOP-Grundlagen***.

3.3.7 Attribute & Reflection

Stellen Sie sich die Möglichkeiten von Reflection in C# als eine Art turbogeladene Version von RTTI (*run time type information*) vor. Mit Reflection sind Sie in der Lage, in Ihren Anwendungen auch spät (zur Laufzeit) gebundene Objekte nach Typinformationen, Methoden, Parameterzahl und anderen Informationen zu fragen. Damit können Sie nicht nur eigenen Code, sondern auch andere Klassen und Methoden nach benötigten Informationen durchsuchen und dynamisch erzeugen.

Attribute stellen eine Möglichkeit dar, Metadaten in einem Quelltext unterzubringen, ohne die Definition der Sprache unnötig aufzublähen und bieten damit eine Möglichkeit, Zusatzinformationen im Code unterzubringen, für die Sie in anderen Sprachen IDL-Dateien, Compileroptionen und andere unschöne Dinge benötigen.

Zusammen bilden diese beiden Konzepte ein elegantes, erweiterbares und leicht erlernbares Konzept für die Erweiterung Ihres C#-Codes um beliebige Metadaten. Mehr dazu erfahren Sie im Referenzteil.

3.3.8 Strukturierte Fehlerbehandlung

In Sprachen wie Java und Delphi längst eine Selbstverständlichkeit, bietet die .NET-Laufzeitumgebung eine Weiterentwicklung dieses Konzepts. Auch C# bietet die Möglichkeit der Behandlung und des Auslösens von Ausnahmen (exceptions). Diese können der Fehlerbehandlung dienen (`try/catch`) oder als eine Art Schutzblock für Ressourcen fungieren (`try/finally`). Aufgrund der Einbindung in die .NET-Umgebung aber sind diese `exception`-Objekte in der Lage, beliebig zwischen Komponenten in verschiedenen Sprachen hin- und hergeschickt zu werden. Als Entwickler sind Sie natürlich in der Lage, eigene, dem jeweiligen Problemraum angepasste Exceptions aus den vordefinierten Klassen abzuleiten.

3.3.9 Interoperabilität mit COM & Win32

Keine Programmiersprache wird Erfolg haben, wenn Sie als Entwickler sofort allen alten, noch bestehenden und mühsam entwickelten Code als Geschichte erklären müssen. Daher bieten C# und das .NET-Framework die Möglichkeit, bestehende Komponenten unter COM nach und nach mit neuen Komponenten aus der .NET-Welt zu ersetzen.

Nutzung von COM

Bedingt durch die internen Unterschiede in Implementierung und Design lassen sich .NET-Komponenten nicht direkt von COM-Clients aus aufrufen. Andererseits besteht für .NET-Komponenten keine direkte Möglichkeit, auf bestehende COM-Server zuzugreifen. Microsoft löst dieses Problem durch logische Adapterstecker, einem CCW (COM callable wrapper) für den Zugriff von COM nach .NET und dem CRW (common runtime wrapper) für die Aktivierung von COM-Servern durch .NET-Komponenten. Damit ist ein gleitender Übergang in die Welt von C# kein Problem mehr. Wenn Sie sich für die (gar nicht so schrecklichen) Details interessieren, lesen sie das Kapitel im Referenzteil.

Aufrufe des Win32-API

Eigentlich wollte ich diesen Absatz gar nicht schreiben. Die Möglichkeit des Aufrufs einer strikt plattformabhängigen API in einem als gekapselten, prinzipiell plattformenabhängigen System lässt jedem halbwegs erfahrenen Entwickler Schauer über den Rücken laufen, weil klar ist, was passieren wird. Andererseits wäre die Beschreibung von C# ohne dieses Thema unvollständig.

Im Grunde ist es ganz einfach: Sie definieren für Ihre Funktion ein Attribut, das angibt, welche DLL zu importieren ist und geben dann die Parameter wie in der API-Deklaration an für eine `external`-Funktion an.

```
// windows runterfahren
[dllimport(dll="kernel32.dll")]

private static external bool ExitWindowsEx(uint uFlags, uint
dwReserved);
```

Listing 3.3: Aufruf einer Win32-Funktion mit C#

3.3.10 Bedingte Kompilierung / Präprozessor

Was mich zu meiner Zeit als C-Neuling an der Uni beinahe zum Wahnsinn getrieben hätte, war der Präprozessor von C (und auch von C++) und dessen Makros. Ein im Prinzip mächtiges Werkzeug wurde nach und nach zu einem Monster, dessen Fehlerpotential durch Makros nur von wirklichen Profis im Zaum zu halten war (und auch dies nur mit unökonomisch hohem Aufwand). Bei C# sind Makros durch den Präprozessor nicht mehr notwendig. Es werden Präprozessorkommandos zur bedingten Kompilierung und zur Strukturierung des Quellcodes unterstützt, mehr ist aufgrund der erweiterten Möglichkeiten und der größeren Klarheit von C# im Vergleich zu C++ auch gar nicht notwendig.

3.3.11 Namespaces

Namespaces sind kein neues Konzept von C#. Wenn Sie von C++ kommen, werden Sie mit den Namespaces von C# keine Probleme haben. Umsteiger von Java können sich die Namespaces wie packages vorstellen, allerdings ohne die Beschränkung der Bestandteile des Package-Names auf Verzeichnisse.

Haben Sie bisher mit einer anderen Sprache als C++ oder Java programmiert, keine Angst! Wie Sie in den nächsten Absätzen erfahren werden, sind Namespaces sehr leicht begreifbar und bieten Ihnen eine zusätzliche Strukturierungsmöglichkeit für Ihren Quellcode. Sie erweitern nämlich einfach die Benennungsmöglichkeiten für Ihre Klassen.

Stellen Sie sich vor, Sie entwickeln ein Kalenderprogramm mit Datumsberechnungen für Archäologen. Natürlich nennen Sie diese Klasse `Datum`. In einem Nachbarteam arbeitet ein Kollege ebenfalls an einer Klasse für Datumsberechnung, allerdings für den Kalender der Inkas. Auch er nennt seine Klasse `Datum`. In einer Sprache ohne Namespaces hätten Sie nun ein Problem. In C# lässt sich dieses Problem relativ einfach lösen.

```
namespace Gregorian
  {
    public class Datum {
      // code steht hier
      }
```

```
  }


namespace Inca
  {
    public class Datum {
      // code steht hier
      }
  }
```

Listing 3.4: Definition gleichnamiger Klassen in unterschiedlichen Namensräumen

Damit erhalten Sie zwei Vorteile: Zum einen besitzen Sie damit eine weitere Möglichkeit, logisch zusammengehörige Klassen in einem namespace zu sammeln. Zum anderen vermeiden Sie damit die oben angesprochenen Konflikte bei der Namensgebung von Klassen und müssen sich nicht irgendwelche obskuren Präfixe aus Vorname und Blutgruppe des Entwicklers für die Benennung von Klassen ausdenken! Wenn Sie sich fragen, wo denn dabei das Problem liegt, arbeiten Sie einfach mal ein Jahr lang in einem Programmierteam mit 20 Entwicklern.

3.3.12 Formatierte Stringausgabe in C#

Dieser Abschnitt ist nur eine Art literarischer Zeiger auf das Kapitel 15 ***Strings***. In den folgenden Kapiteln werden Sie immer wieder Ausgaben finden, die so aussehen, wie der folgende Code:

```
Console.WriteLine("Attribut {0}, Name = {1}",
                  i, attr[i]);
```

Diese Art der Ausgabeformatierung entspricht von der Funktion (und in weiten Teilen auch syntaktisch) der `printf`-Notation von C und C++. Wieso das C#-Team auf einer eigenen Kennung für einzusetzende Paramter bestand, entzieht sich der Kenntnis des Autors, alle Details zu diesen Formatierungen finden Sie aber im Kapitel über die Strings.

3.4 Zusammenfassung

Damit bin ich am Ende des ersten Streifzugs quer durch die Sprache C# angekommen. Sie sollten einen ersten Überblick über die Sprache und die Gründe für ihre Entwicklung bekommen haben und einordnen können, wie groß die Unterschiede zu der bisher von Ihnen benutzten Programmiersprache liegt.

Hier eine kleine Zusammenfassung der Vorteile von C# gegenüber anderen Programmiersprachen für die Entwicklung von .NET-Anwendungen

- Eine konsistente und leicht erlernbare Syntax
- Sicherheit und Vermeidung von Seiteneffekten durch in und out Parameter
- Typsichere Aufzählungen
- Strukturen als Werttypen ohne unnötigen Ballast
- Compiler verhindert eine Vielzahl von Flüchtigkeitsfehlern durch entsprechende Prüfungen (zwangsweise Variableninitialisierung, bool-Ausdruck bei if, etc.)
- Komplett objektorientiertes Typsystem, durch Boxing können einfache Datentypen ohne wesentlichen Performanceverlust als Objekte behandelt werden
- Unterstüzung des Entwicklers bei der Fehlersuche durch Debug- und Trace-Klassen, Asserts und Schnittstellen zum Debugger
- Komplette Nutzung der .NET-Umgebung, nicht auf CLS-Kompatibiliät beschränkt

4 Namensräume

Das Konzept der Namensräume (*namespaces*) ist bei vielen Entwicklern bereits bekannt, da einige Programmiersprachen ein solches Konzept bereits seit Jahren unterstützen. Auch hier hat sich das C#-Team nach dem Motto 'nimm das Gute und lass das weniger Gute weg' bei den bestehenden Sprachen umgesehen.

C++ / Java



Im Vergleich zu C++ wurden die Namensräume mit zusätzlicher Funktionalität ausgestattet, als C++-Entwickler sollten Sie also auf ziemlich viele vertraute Dinge stoßen. Im Gegensatz zu Java sind die einzelnen Hierarchiestufen eines Namensraums nicht an einzelne Verzeichnisse gebunden.

Sie werden sich möglicherweise wundern, dass dieses Thema gleich zu Beginn des Buches vor allen anderen Themen behandelt wird. Bedingt durch die Strukturierung der .NET-Laufzeitumgebung in verschiedenen Namensräumen ist das Verständnis dieses Konzepts Voraussetzung für fast alle weiteren Kapitel. Daher möchte ich sichergehen, dass Sie über eine gewisse Sicherheit im Umgang mit Namensräumen verfügen, bevor wir die restlichen Themen in Angriff nehmen.

4.1 Grundlagen

Allen Lesern, die bis jetzt mit Sprachen ohne die Unterstützung von Namensräumen gearbeitet haben, möchte ich gleich zu Beginn eines versprechen: Sie werden diese Dinger wirklich mögen! Denn Namensräume lösen eines der ältesten Probleme im Software-Universum: das der Namenskonflikte bei der Benennung von Datentypen. Die .NET-Laufzeitumgebung macht selbst ausgiebig Gebrauch von diesem Konzept. Anders wären die Tausende von Klassen, die in dieser Umgebung stecken, auch kaum zu handhaben gewesen.

Stellen Sie sich vor, Ihre Firma entwickelt eine Klasse mit dem Namen `Ueberweisung`, welche innerhalb des deutschen Bankensystems den Transfer von einem Konto zu einem anderen Konto bewerkstelligt. Aufgrund einer Expansion Ihres Kunden sind Sie nun plötzlich

gezwungen, auch Überweisungen im europäischen Ausland zu handhaben. Wie wird diese Klasse genannt? In C# ganz einfach ebenfalls `Ueberweisung`! Hier können Sie nämlich für jede Klasse und all ihre Unterstützungsklassen und -daten einen eigenen Namensraum definieren.

Dies möchte ich Ihnen zeigen, indem wir zuerst einen Blick darauf werfen, was Microsoft Visual Studio.NET eigentlich macht, wenn Sie ein neues Projekt anlegen.

4.2 Definition eines Namensraums

Die von Ihnen entwickelten Klassen eines Projekts kommen alle in einen eigenen Namensraum, den VS.NET mit dem Namen Ihres Projekts erzeugt.

```
namespace Projekt1
{
}
```

Listing 4.1: Ein 'leerer' Namensraum

Alles innerhalb dieser geschweiften Klammern gehört nun zum Namensraum PROJEKT1. Stellen Sie sich das analog zum Dateisystem Ihres Computers vor: Wenn Sie einen neuen Ordner ORDNER1 anlegen, gehören alle darin erstellten Dateien in diesen Ordner. Genauso verhält es sich mit den Klassen Ihres Projekts:

```
namespace Projekt1
{
   public class Testklasse
   {
   }
}
```

Listing 4.2: Eine Klasse in einem Namensraum

Der Name der Klasse lautet nicht `Testklasse`, sondern eigentlich `Projekt1.Testklasse`. Jeder Zugriff von außerhalb des Namensraums muss auch über diesen kompletten Namen erfolgen. Ein anderes System, das ähnliche Strukturen erlaubt und welches Sie sicher auch kennen, ist das *Domain Name System* (DNS) des Internet. Ebenso wie bei

Domainnamen bei der kompletten Angabe des Pfades zum Rechner von einem vollqualifizierten Domainnamen (*fully qualified domain name*) gesprochen wird, so heißt in der C#-Terminologie die Kombination aus Namensraum und Datentyp ebenfalls vollqualifizierter Name (*fully qualified name*).

4.2.1 Hierarchische Namensräume

Jetzt komme ich auf mein Beispiel mit der Überweisung zurück. Für die innerdeutsche Überweisung können Sie nun einen Namensraum `Deutschland` erstellen und dort die Klasse `Ueberweisung` unterbringen, für die europäischen Überweisungen einen Namensraum `Europa`. Wenn aber das Projektteam zwei Abteilungen weiter ebenfalls einen Namensraum `Europa` erstellt? Macht nichts, denn wie die Domainnamen im Internet sind auch die Namensräume bei C# hierarchisch aufbaubar. Sie erstellen einen Namensraum mit dem Namen Ihrer Abteilung oder Firma (in meinem Beispiel `XYZCorp`) und schließen darin die beiden anderen Namensräume ein.

```
namespace XYZCorp
{

   namespace Banking
   {
      namespace Deutschland
      {
         public class Ueberweisung
         {
         }
      }

      namespace Europa
      {
         public class Ueberweisung
         {
         }
      }

   }
}
```

Listing 4.3: Geschachtelte Namensräume

Damit ist jede Klasse nun eindeutig ohne die Gefahr einer Namenskollision ansprechbar. Einmal als `XYZCorp.Banking.Deutschland.Ueberweisung` und einmal als `XYZCorp.Banking.Europa.Ueberweisung`!

Sie ahnen es sicher schon, es muss einfach eine Möglichkeit geben, bei Bedarf ohne den gesamten Pfad des Namensraums bis zur Klasse auszukommen. Schließlich funktioniert das bei Dateinamen und Domainnamen auch. Diese Möglichkeit existiert und ihr Name ist `using`.

4.3 Nutzen eines Namensraums mit using

Fast alle Beispiele in C# beginnen mit einer oder mehreren Zeilen in der Form:

```
using System;
using System.Reflection;
using System.Text;
```

Listing 4.4: Angabe benutzter Namensräume mit using

Mit dieser Angabe werden alle Namen (und damit natürlich die Datentypen und Klassen) dieses Namensraums in den derzeit gültigen Namensraum importiert und ich kann die Namen dieser Klassen benutzen, ohne jedes Mal den vollqualifizierten Namen angeben zu müssen (was, nebenbei bemerkt, den Code auch nicht gerade lesbarer macht).

Auch hierzu möchte ich Ihnen gleich ein Beispiel zeigen:

```
namespace MeinProjekt
{
   using System;

   public class Test
   {
      public static void Main()
      {
         Console.WriteLine("Hallo, Welt!!");
         Console.ReadLine();
      }
   }
}
```

Listing 4.5: Hallo, Welt mit Namensräumen

Wie Sie anhand der beiden fettgedruckten Zeilen erkennen, kann ich nun die Klasse `Console` benutzen, ohne den kompletten Namen `System.Console` angeben zu müssen. Ohne die Benutzung von Namensräumen würde der Code so aussehen:

```
namespace MeinProjekt
{
   public class Test
   {
      public static void Main()
      {
         System.Console.WriteLine("Hallo, Welt!!");
         System.Console.ReadLine();
      }
   }
}
```

Listing 4.6: Hallo, Welt ohne Namensräume

Nachdem es Namensräume wie

- System.Runtime.InteropServices.CustomMarshalers
- System.EnterpriseServices.CompensatingResourceManager
- System.ComponentModel.Design.Serialization

gibt, wird schnell klar, warum ein `using <namespace>` wesentlich weniger Tipparbeit bedeutet.

Java



Im Gegensatz zu Java ist es nicht möglich, eine bestimmte Klasse zu importieren. Sie können nur das gesamte Paket (den `namespace`) angeben, den Sie importieren möchten.

4.4 Namenskonflikte beim Import mit using

Frei nach Murphys Gesetz geben zwei Entwickler in zwei unterschiedlichen Teams einer Klasse in ihrem Namensraum genau dann den gleichen Namen, wenn diese beiden Namensräume zusammen in einem Projekt Ihres Teams benutzt werden müssen. Anders ausgedrückt, was passiert eigentlich, wenn Sie zwei Namensräume importieren und dort jeweils eine Klasse mit dem gleichen Namen definiert ist?

```
namespace Projekt1
{
   using ABCGmbH.Banking;
   using XYZCorp.Banking;

   public class Test
   {
      public static void Main()
      {
        // aus ABCGmbH.Banking
        Ueberweisung uew1;

        // aus XYZCorp.Banking;
        Ueberweisung uew2;
      }
   }

}
```

Listing 4.7: Ein Namenskonflikt trotz Namensräumen

Findet der Compiler beim Durchsuchen der importierten Namensräume mehr als einen Datentyp mit dem gleichen Namen, erhalten Sie eine Fehlermeldung. Jetzt besitzen Sie zwei Möglichkeiten. Einmal können Sie bei beiden Typen den vollqualifizierten Namen angeben und damit den Konflikt auflösen. Zum anderen können sie einen Alias definieren.

4.4.1 Aliase

Ein Alias dient eigentlich dazu, die (zumindest mir) angeborene Faulheit eines Entwicklers zu unterstützen (wobei ich dies natürlich effizienten Kräfteeinsatz nenne). Anstelle der kompletten Angabe des Namensraumes können Sie im Code später mit dem Alias arbeiten. Dieser wird natürlich kürzer gewählt als der Namensraum und damit sparen Sie sich wieder Schreibarbeit.

```
namespace Projekt1
{
   using EnBank = Einsteinet.ASP.Banking.Deutschland;

   public class Test
   {
```

```
      public static void Main()
      {
        // kurzer Alias statt vollquailifiziert
        EnBank.Ueberweisung uew2;
      }
   }

}
```

Listing 4.8: Aliase machen den Code kürzer und besser lesbar

Damit habe ich aber auch eine Möglichkeit, den Konflikt aus dem letzten Abschnitt mit minimalem Aufwand aufzulösen: durch die Definition eines Alias für jeden Namensraum, der importiert wird.

```
namespace Projekt1
{
   using abc = ABCGmbH.Banking;
   using xyz = XYZCorp.Banking;

   public class Test
   {
      public static void Main()
      {
        // aus ABCGmbH.Banking
        abc.Ueberweisung uew1;

        // aus XYZCorp.Banking;
        xyz.Ueberweisung uew2;
      }
   }

}
```

Listing 4.9: Aliase zur Konfliktauflösung beim Namensraumimport

4.5 Die wichtigsten Namensräume

Namensraum	Klassen, Inhalt
System	Der grundlegende Namensraum für Basisklassen, der die Fehlerbehandlung (System.Exception) sowie die allgemeinen Datentypen und die Klasse object definiert.

Tabelle 4.1: Einige Namensräume aus der .NET-Laufzeitumgebung

Namensraum	Klassen, Inhalt
System.IO	Ein- und Ausgabeklassen, Dateibehandlung, StreamReader usw.
System.Collection	Hier finden sich u.a. die Klassen für Auflistungen, Stacks, verbundene Listen.
System.Text	Namensraum für die StringBuilder-Klasse, die regulären Ausrücke und die Textmanipulation.
System.Windows.Forms	Die Windows Forms für die Erstellung von GUI-Anwendungen sind in diesem Namensraum zu finden.
System.Web	Das Gegenstück zu WinForms. Hier finden Sie die ASP.NET-Klassen für die WebForms.
System.Reflection	Klassen zur Ermittlung von Informationen über Datentypen zur Laufzeit.

Tabelle 4.1: Einige Namensräume aus der .NET-Laufzeitumgebung (Forts.)

Diese Aufzählung könnte ich noch seitenweise so weitertreiben. Erst bei einer Art Baumansicht der von Microsoft erstellten Namensräume wird langsam klar, welchen Umfang die .NET-Laufzeitumgebung überhaupt besitzt und dass es ohne diese Möglichkeit einer Strukturierung oberhalb der Ebene von Klassen keine Chance gäbe, so ein System zu organisieren. In Kapitel 2 ***Das .NET-Framework*** finden Sie eine Art Landkarte mit allen Namensräumen der .NET-Laufzeitumgebung unterhalb von `System`.

4.6 Zusammenfassung

Namensräume bieten eine flexible Möglichkeit, Konflikte durch gleich benannte Bezeichner zu vermeiden. Zusätzlich bieten sie ein hierarchisches Strukturierungsmittel für Codebibliotheken.

Die Grenzen eines Namensraums sind nicht auf eine Datei bzw. ein Assembly beschränkt, sondern mehrere Assemblies können Typen für einen Namensraum definieren.

Normalerweise werden die Bezeichnungen für einen Namensraum so vergeben, dass diese ungefähr einem umgedrehten Internet-Domainnamen des Unternehmens entsprechen und damit von links nach rechts

immer spezieller werden. Der Vorschlag von Microsoft lautet: Firma (evtl. mit Land), Bereich, Technologie, Gruppe.

Beispiele:

- XYZCorp.Imaging.AlphaBlending
- com.abc.text.parser
- Serialisierung.Xml

4.7 Übungen



1. Wie können Sie zwei Methoden mit gleichen Namen in Ihrer Anwendung benutzen, von denen jede in einem eigenen Namensraum vorhanden ist?
2. Suchen Sie in der Online-Dokumentation, in welchem Namensraum die Klasse BinaryFormatter vorhanden ist.
3. Welche Funktion hat die folgende Quellcodezeile?

```
using regexp = System.Text.RegularExpressions;
```

5 Daten, Typen, Variablen

5.1 Wert- und Referenztypen

C# unterscheidet zwei grundlegende Typen von Daten: Werttypen (value types) und Referenztypen (reference types). Werttypen enthalten die tatsächlichen Daten, während die Referenztypen einen Verweis auf die Adresse der Daten speichern. Werttypen werden auf dem Stack angelegt und übergeben, während die Referenztypen auf dem Heap erzeugt werden und beim Aufruf eben nur die Referenz übergeben wird.

Werttypen sind in C# alle einfachen, skalaren Datentypen sowie structs und enums. Also alles, was einfach nur einen Wert enthält oder eine Verpackung für einen Wert darstellt, sei es ein Zähler in einer Schleife oder die Darstellung einer Koordinate als Struktur von X- und Y-Koordinate. Alle anderen Datentypen sind Referenztypen. Dazu gehören Instanzen vom Typ `object` ebenso wie Strings, selbstdefinierte Klassen und Arrays. Gemeinsam ist allen Datentypen, dass sie vom gemeinsamen Vorfahr `object` abstammen.

5.2 Alles ist ein Objekt

An dieser gemeinsamen Abstammung krankten viele bisherige Programmiersprachen. Entweder wurde das Konzept konsequent realisiert, was für einfache Typen wie einen `int` einen enormen internen Aufwand für die Verwaltung der Objektreferenzen bedeutete (und sich in einer dementsprechenden Performance niederschlug) oder die Designer der Programmiersprache brachen aus Gründen der Performance mit diesem Konzept. Dies führte dann dazu, dass die skalaren Datentypen auf wundersame Art und Weise im System 'einfach da' waren und die Konsistenz des Programmiermodells darunter litt. Ein Beispiel für den ersten Ansatz ist Smalltalk, ein Beispiel für den zweiten Ansatz bietet Java. Die Entwickler von C# waren mit beiden Möglichkeiten unzufrieden und wandten einen Trick an, der ebenso einfach wie elegant ist. Das Zauberwort hierfür heißt `boxing` bzw. `unboxing`.

5.2.1 Boxing und Unboxing

Wird ein Werttyp als Referenztyp behandelt, beispielsweise durch Übergabe als Parameter an eine Funktion, welche einen Parameter vom Typ `object` erwartet, dann belegt die Laufzeitumgebung eine Objektbox und legt dort eine Kopie des Wertes ab. Damit existiert eine Referenz und die Funktion kann ohne Probleme aufgerufen werden.

Obwohl dies nahezu immer automatisch geschieht, können Sie dies manuell jederzeit in Code programmieren.

```
int x = 5;
object obj;
    obj = x;
```

Listing 5.1: Boxing eines Werttyps

Sie haben nun eine Referenz auf eine Objektbox und darin den Werttyp. Schön, aber wie bekommen Sie nun das Gegenteil, also den `int` wieder zurück eine Variable mit dem Werttyp `int`? Dies geschieht mit `unboxing`, indem die Box sozusagen ausgeleert und der Werttyp wieder herausgeholt wird.

```
int x = 5;
object obj;
    obj = x;
    // unboxing
    int m = (int) obj;
```

Listing 5.2: Unboxing eines Werttyps

Der Typcast (die erzwungene Umwandlung in einen `int`) ist hier notwendig, da die Referenz auf ein `object` ja jeden beliebigen Typ speichern könnte. Die Laufzeitumgebung führt hier eine Typprüfung durch. Ist in der Objektbox kein `int` enthalten, wird eine Exception ausgelöst.

5.2.2 Boxing und Literale

Da in der .NET-Umgebung alle Datentypen wie bereits geschildert von `object` abstammen, lassen sich auch alle einfachen Werttypen mit der gerade gezeigten Methode in Referenztypen umwandeln. Nach-

dem Literale ebenfalls einen Datentyp besitzen, ist es in C# völlig in Ordnung, folgende Anweisungen zu schreiben:

```
Using System;
class Test
{
    public static int Main()
    {
        Console.WriteLine( 3.ToString() );
        return (0);
    }
}
```

Listing 5.3: Behandlung eines einfachen Typs als Objekt

Schließlich ist der Literalwert 3 nichts anderes als eine direkte Instanz eines Integers. Damit bietet C# ein wirklich vollständiges und konsistentes Programmiermodell objektbasierter Datentypen.

5.3 Einfache Datentypen

Wie jede andere Programmiersprache stellt auch C# eine Reihe von vordefinierten Datentypen. Diese werden aber nicht von C# zur Verfügung gestellt, sondern von der darunter liegenden .NET-Umgebung, genauer dem Common Type System. Jeder Datentyp in C# entspricht daher einem im System-Namespace definierten .NET-Typ. Die Tabelle Datentypen listet die einfachen Datentypen und die darunter liegenden .NET-Typen auf.

C#-Typ	.NET-Typ	Beschreibung
sbyte	System.Sbyte	8 Bit mit Vorzeichen
byte	System.Byte	8 Bit ohne Vorzeichen
short	System.Int16	16 Bit mit Vorzeichen
ushort	System.Uint16	16 Bit ohne Vorzeichen
int	System.Int32	32 Bit mit Vorzeichen
uint	System.Uint32	32 Bit ohne Vorzeichen
long	System.Int64	64 Bit mit Vorzeichen

Tabelle 5.1: Die einfachen Datentypen in C#

C#-Typ	.NET-Typ	Beschreibung
ulong	System.Uint64	64 Bit ohne Vorzeichen
float	System.Single	Gleitkommazahl einfach genau
double	System.Double	Gleitkommazahl doppelt genau
decimal	System.Decimal	Dezimalzahl mit 28 signifikanten Stellen
char	System.Char	Ein Unicode-Zeichen
bool	System.Boolean	Ein logischer Wert (TRUE | FALSE)

Tabelle 5.1: Die einfachen Datentypen in C# (Forts.)

Da die C#-Bezeichnungen für die Datentypen also nur einen Alias für die tatsächliche Bezeichnung des Typs der .NET-Laufzeitumgebung darstellen, können Sie natürlich auch anstelle eines `int` ohne Probleme einen System.Int32 deklarieren. Aus Gründen der Lesbarkeit des Codes ist die aber nicht empfehlenswert und ich werde in allen Listings auch die C#-Bezeichnungen der Datentypen benutzen (außerdem ist es weniger Tipparbeit).



C++

Im Gegensatz zu C++ sind die vorzeichenbehafteten und die vorzeichenlosen Datentypen in der .NET-Laufzeitumgebung tatsächlich eigenständige, voneinander unabhängige Datentypen. Im Zuge der 'Sprachentrümpelung' wurde auch gleich mit den Modifikatoren `signed`, `unsigned`, `short` und `long` kurzer Prozess gemacht. Als Entwickler müssen Sie sich explizit für einen der oben angegebenen Datentypen entscheiden.

5.3.1 Bool

Der Datentyp `bool` ist ein eigener Datentyp, nicht einfach eine Variante eines ganzzahligen Datentyps. Daher müssen beispielsweise Ausdrücke innerhalb einer `if`-Anweisung auch zu einem echten `bool`-Wert (als Literal ausgedrückt durch die Konstanten `true` und `false`) evaluieren. Dieser Datentyp ist nicht mit den ganzzahligen Typen austauschbar.

5.3.2 Char

Für den Datentyp `char` existieren einige Besonderheiten, die Ihnen als Programmierer vielleicht die eine oder andere kleine Überraschung bereiten könnten, deshalb an dieser Stelle ein kleiner Einschub. Das Begrenzungszeichen für Literale `char`-Angaben ist das einfache Anführungszeichen (`'`), das für string die doppelten Anführungszeichen (`"`). Achten Sie darauf, diese beiden nicht zu verwechseln.

Ein `char` enthält ein Unicode-Zeichen und liegt damit als Zahl im Bereich von 0 bis 65535 und gleichzeitig im Bereich von `ushort`. Dennoch sind nicht alle Operationen, die mit einem `ushort` möglich sind, auch mit einem `char` möglich, um Fehler im Programmcode zu verhindern. Der 'offizielle' Wertebereich von `char` ist der Unicode-Zeichensatz (ISO 464), also zwei Bytes. Aus diesem Grunde existieren keine impliziten Umwandlungen für `char`, obwohl sich Werte aus `sbyte`, `byte` und `ushort` durchaus in einem `char` speichern ließen.

Literale Angaben für einen `char` müssen immer als Zeichenliterale geschrieben werden. Die Angabe eines nummerischen Wertes ist nur in Verbindung mit einem Typcast möglich. So würde beispielsweise die Deklaration `const char LF = (char)10` der Deklaration `const char LF = '\x000A'` entsprechen.

Mögliche Angaben für Zeichenliterale:

- Als einzelnes Zeichen: `'A'`
- Als Umwandlung aus einem int: `(char) 65`
- Als hexadezimales Literal: `'\x41'`
- Als Unicode-Wert: `'\u0041'`

5.3.3 String

In C# sind Zeichenketten ein unveränderlicher Datentyp. Dies bedeutet, dass eine Zeichenkette (z.B. eine Variable vom Typ `string`) nach der Erzeugung nicht mehr verändert werden kann. Alle Zeichenkettenfunktionen bearbeiten daher nicht den übergebenen Parameter, sondern liefern eine neue Instanz eines Strings mit dem verarbeiteten Inhalt zurück.

Als Literale werden Strings in doppelte Anführungszeichen eingeschlossen. Innerhalb dieser Anführungszeichen werden aber bestimmte Zeichen anders behandelt als der Rest. Was ich meine, ist das von C und C++ bekannte Konzept der Darstellung von Steuerzeichen mit einem vorangestellten Backslash. Damit lassen sich beispielsweise Zeilenvorschübe (`\n`), Tabulatoren (`\t`) und andere Zeichen ausgeben.

```
const string Pfad = "C:\\WINNT\\SYSTEM32"

const string Verbatim = @"C:\WINNT\SYSTEM32"

Console.WriteLine("Das ist ein Text.");
```

Listing 5.4: Beispiele für String-Literale

Wie Sie beim ersten Beispiel erkennen, hat dieses Konzept der Einleitung von Steuerzeichen durch einen Rückwärtsschrägstrich aber auch Nachteile (zumindest in der Windows-Welt, den Unix-Entwicklern mit dem Schrägstrich als Pfadtrenner ist diese Problematik völlig fremd). Bei einer Windows-Pfadangabe muss jeder Backslash zweimal angegeben werden, da die Angabe `\\` für einen Rückwärtsschrägstrich in der Ausgabe sorgt. In C# hat man daher die 'Verbatim-Strings' eingeführt, bei denen keinerlei Behandlung von Sonderzeichen oder Steuerzeichen vorgenommen wird. Daher kommen diese Angaben (Zeile 2 bei den Beispielen oben) mit einem einfachen Backslash aus. Gekennzeichnet werden die Verbatim-Strings durch ein vorangestelltes `@`-Zeichen.

Das Verkettungszeichen für Strings ist auch in C# der Operator +. Weitere Details erfahren sie in Kapitel 15 ***Strings***.

5.3.4 Nummerische Literale

Die Angabe literaler Werte ist für Programmierer aus dem C-Umfeld sehr vertraut. Nach der Angabe des Zahlenwertes folgt ein Suffix, welches den zu verwendenden Datentyp angibt. Gleitkommawerte ohne ein Suffix werden automatisch als double aufgefasst.

```
int zahl = 42;
// einfache genauigkeit mit F
float f = 3.1415726F;
// ohne suffix = double
```

```
double ln = 2.302585092994045684;
// M = decimal
decimal netto = 1000000.00M;
// U = ohne vorzeichen
ushort p = 59468U;
```

VB



Für Visual Basic-Programmierer sieht die Sache etwas anders aus. Obwohl die .NET-Umgebung allen Sprachen ein gemeinsames Typsystem zur Verfügung stellt, kennt Visual Basic vom Sprachkonzept her keine vorzeichenlosen Datentypen. Ein Integer in VB ist immer vorzeichenbehaftet. Es hindert Sie allerdings niemand daran, den Datentyp direkt über den .NET-Datentyp zu deklarieren, um so auf die gleichen Datentypen wie C# zugreifen zu können.

```
Dim x as System.Uint32
```

Listing 5.5: Deklaration eines vorzeichenlosen Typs in VB.NET

5.4 Bezeichner

Bezeichner sind, wie der Name bereits ausdrückt, Namen für Konstanten, Variablen, aber auch Strukturen, Klassen und für alles, was in C# einen Namen erhalten kann. Der erste Buchstabe eines Bezeichners ist ein Buchstabe oder der Unterstrich. Danach können Buchstaben, Ziffern und andere Unicode-Literale folgen. Unicode-Zeichen werden in der Notation \uhhhh angegeben, wobei 'h' für eine hexadezimale Ziffer steht. Dies ist die einfache Version. Aufgrund der Unterstützung von Unicode durch die .NET-Laufzeitumgebung und C#, sind die lexikalischen Details etwas komplexer als hier dargestellt. Sollten Sie also vorhaben, Ihren Variablen oder Funktionen exotischere Namen zu geben, werfen Sie vorher einen Blick in die Sprachreferenz zu C#, welche Sie auf der MSDN-Website von Microsoft oder ausgedruckt und gebunden als Buch im Fachhandel finden.

Die .NET-Laufzeitumgebung unterstützt die Programmierung mit mehr als einer Sprache und C# ist nur ein Mitglied der großen .NET-Familie. Daher kann es bei der Softwareentwicklung durchaus vorkommen, dass Sie auf Bezeichner für Variablen oder Funktionen stoßen, die in C# als Schlüsselwörter reserviert sind. Solche Bezeichner können in C# verwendet werden, wenn ihnen das @-Zeichen vorangestellt wird.

Falls möglich, sollten Sie dies in eigenen Programmen so weit als möglich vermeiden, um die Verwirrung beim Lesen von Quelltexten zu minimieren. Es ist laut der Sprachreferenz auch möglich, das vorangestellte @-Zeichen auch für Bezeichner zu verwenden, welche keine Schlüsselworte darstellen, davon wird aber aus stilistischen Gründen strikt abgeraten.

5.5 Konstanten

Aufgrund der Flexibilität des gemeinsamen Typsystems werden Sie zwischen den eingebauten Datentypen von C# und den von Ihnen definierten Datentypen kaum jemals einen Unterschied feststellen. Sie sind in der Lage, alles mit benutzerdefinierten Datentypen zu tun, was Sie auch mit den integrierten Datentypen tun können, mit einer Ausnahme: Bei Konstanten können Sie nur die integrierten Typen als Literale schreiben, nicht die benutzerdefinierten Datentypen.

5.5.1 Echte Konstanten

Eine echte Konstante ist das, was Sie wahrscheinlich vermuten: die literale Angabe eines Wertes in Verbindung mit dem Schlüsselwort `const`. Sehen Sie dazu das Listing 5.6 an.

```
using System;
// eine menge konstanten
class Konstanten
{
   public const double Pi = 3.14159265553;
   public const float G = 9.81F;
   public const int Magic = 42;
}
// testklasse für die konstanten
class test
{
   public static int Main()
   {
      Console.WriteLine("Pi = {0}",Konstanten.Pi);
   }
}
```

Listing 5.6: Konstantendefinitionen

Sie erkennen bei diesem Listing auch, dass ein `const` immer auch `static` impliziert, d.h. der Wert der Konstanten wird natürlich nur einmal für alle Instanzen der Klasse gespeichert. Daher greife ich bei der Ausgabe auch direkt auf den Namen der Klasse zurück, ohne erst eine Instanz dieser Klasse (ein Objekt) zu erzeugen. Mehr dazu erfahren Sie im Kapitel 6 ***OOP-Grundlagen***.

Ab und zu tritt aber nun der Fall ein, dass sich eine Konstante zwar direkt als literaler Wert angeben ließe, dies aber aus Gründen der Verständlichkeit und Lesbarkeit des Codes nicht möglich ist. Auch dafür gibt es in C# eine Lösung.

5.5.2 Konstantenausdrücke

Sie können Konstanten auch mit mathematischen Ausdrücken erstellen. Dazu muss dem Compiler aber zum Zeitpunkt der Übersetzung der Wert des Ausdrucks bekannt sein, dass dieser in eine Konstante umgerechnet werden kann. Ein klassisches Beispiel hierfür sind Auflistungen von Fehlercodes, die ab einem gemeinsamen Basiswert gezählt werden.

```
class ErrorCodes
{
 public const ushort ERRORBASE = 50000U;
 public const ushort INVALID_PASSWORD = ERRORBASE + 1;
 public const ushort INVALID_USERNAME = ERRORBASE + 2;
 // viele weitere fehlermeldungen
 public const ushort ACCOUNT_LOCKED   = ERRORBASE + 999;
}
```

In einem solchen Konstantenausdruck in C# dürfen die folgenden Elemente enthalten sein:

- Literale Angaben (z.B. 35, 12.5F oder das Literal `null`)
- Bezüge auf Konstante in `class`- und `struct`-Typen
- Bezüge auf Mitglieder einer Aufzählung (`enum`)
- Unterausdrücke in Klammern
- Typumwandlungen (casts), wenn das Ergebnis ein einfacher Typ ist

- Die folgenden unären Operatoren: `+`, `-`, `!` und `~`
- Die folgenden binären Operatoren, wenn die beiden Operanden aus der Menge der oben genannten Elemente stammen: +, –, *, /, %, <<, >>, &, |, ^, &&, ||, ==, !=, <, >, >=, <=
- Der Bedingungsoperator `?:`

Kann der Ausdruck zum Zeitpunkt der Übersetzung nicht in eine echte Konstante aufgelöst werden, so erhalten Sie einen Compilerfehler. Aufgrund der integrierten Typsicherheit und Überprüfung von C# erhalten Sie ebenfalls einen Compilerfehler, wenn bei der Auswertung des Ausdruckes ein nummerischer Überlauf zustande kommt. Obwohl ich dies nicht empfehle, können Sie diese Überprüfung durch die Verwendung eines `unchecked`-Blocks umgehen.

5.5.3 Read-Only Variablen

Ein Problem, auf das Sie bei der Verwendung von Konstanten stoßen können, ist folgendes Szenario: Sie möchten gerne eine Konstante benutzen, aber deren Wert wird erst bei der Übersetzung des Codes berechnet oder kann aufgrund von Anforderungen an die Anwendung erst nach dem Start dieser Anwendung festgelegt werden. In allen anderen Programmiersprachen müssten Sie nun auf die Vorteile einer Konstantendeklaration verzichten. Irgendwann weisen Sie dann dieser 'Konstanten' versehentlich einen Wert zu und Ihre Anwendung geht den Weg alles Irdischen …

C# bietet Ihnen mit den so genannten `readonly`-Variablen eine Möglichkeit, diese Fehlerquelle zu umgehen.

```
class Test
{
   public readonly string domain;
   public Test(string d)
   {
      domain = d;
   }
}
```

Listing 5.7: Benutzung einer readonly-Variablen

Diese Methode erlaubt es Ihnen, den Wert der so definierten Variablen ein einziges Mal zu setzen (normalerweise in einem Konstruktor, wie in diesem Beispiel), danach kann auf diese Variable nur noch lesend zugegriffen werden. Eine schöne Ergänzung zur Vermeidung von Flüchtigkeitsfehlern beim Programmieren!

5.6 Variablen

Variablen sind Speicherpositionen für Werte. So weit, so einfach. Bei einer typsicheren Sprache wie C# kommen nur einige Eigenschaften dazu. Eine Variable besitzt nämlich nicht nur einen Bezeichner (den Namen) und einen Inhalt, sondern auch einen definierten Datentyp (.NET-Typ oder benutzerdefiniert) und die Laufzeitumgebung garantiert auch, dass eine Variable nur Daten dieses Types enthalten kann.

Insbesondere für VB-Programmierer bedeutet dies: es gibt keine Variants! Jede Variable in einem C#-Programm besitzt immer einen bestimmten Datentyp. Eine Umwandlung in einen anderen Datentyp kann nur durch Typumwandlungen (type casts) erfolgen, welche einen Datentyp vor der Zuweisung in einen anderen umwandeln. Aber dazu später mehr.

5.6.1 Deklaration

Um eine Variable zu deklarieren, gehen Sie vor wie in einer anderen Programmiersprache auch. C# folgt hierbei dem Muster Wo, Was, Wer. Dies bedeutet: zuerst steht ein optionaler Modifikator (das »Wo«), welcher den Geltungsbereich der Variablen oder andere Spezifikationen angibt (z.B. `public` oder `readonly`), danach folgt der Typ der Variablen (das »Was«), dann schließlich der Name der Variablen (das »Wer«).

```
public float verbrauch;
private readonly double g;
```

Listing 5.8: Beispiele für Variablendeklarationen

5.6.2 Initialisierung von Variablen

Die Laufzeitumgebung von .NET sorgt dafür, dass keine Zugriffe auf nicht initialisierte Variablen möglich sind. Wird eine Variable verwendet, ohne dass zuvor eine Zuweisungsoperation stattgefunden hat, gibt der Compiler eine Fehlermeldung aus. Alle Probleme mit zufälligen Werten durch die Verwendung von nicht initialisierten Variablen in Sprachen wie C sollten damit der Vergangenheit angehören.

Wie bei vielen anderen Programmiersprachen auch, lassen sich die Startwerte der Variablen direkt bei der Deklaration angeben. C++- und Java-Programmierern wird das nur ein mildes Lächeln entlocken, für VB-Umsteiger ist dieses Konzept aber neu, daher an dieser Stelle ein paar Beispiele.

```
public uint EineZahl = 4242;
private string domain = "DE";
private float mwst;
public int problem = mwst * 2; // Fehler!
// der wert von mwst ist zur Compilezeit nicht bekannt
```

Listing 5.9: Initialisierungen von Variablen bei der Deklaration

Als Initialwert lassen sich wie gezeigt nicht nur nummerische Literale, sondern auch alle anderen Konstantenausdrücke verwenden, die im Abschnitt ***Konstantenausdrücke*** erläutert wurden.

5.6.3 Variablenkategorien in C#

C# unterscheidet verschiedene Kategorien von Variablen:

- Statische Variable
- Instanzvariable
- Lokale Variable
- Array-Elemente
- Wertparameter
- Referenzparameter
- Ausgabeparameter

Statische Variable

Statische Variablen werden mit dem Schlüsselwort `static` deklariert und existieren ab der Erzeugung des Datentyps, in dem sie deklariert wurden bis zur Beendigung des Programms. Eine statische Variable enthält den Standardwert des jeweiligen Datentyps (siehe ***Standardwerte***)

Instanzvariable

Alle Variablen, welche ohne das Schlüsselwort `static` deklariert werden, gelten als Instanzvariablen. Da in C# jeder Code in einer Klasse definiert werden muss, gibt es nur zwei mögliche Typen von Instanzvariablen: die in einer Klasse und die in einer Struktur. Instanzvariablen werden erzeugt bzw. sind verfügbar, wenn die Klasse bzw. die Struktur erzeugt wurde und sind verfügbar, solange die Klasse bzw. Struktur verfügbar ist. Für Klassen bedeutet dies, dass keine Referenz mehr auf die Klasse existiert und ein eventuell definierter Destruktor aufgerufen wurde.

Lokale Variable

Eine lokale Variable kann innerhalb eines Blocks, innerhalb einer `for`-Schleife (siehe Kasten ***Gültigkeit von Schleifen-Variablen***) oder innerhalb eines `switch`-Blocks deklariert werden. Lokale Variablen lassen sich in C# nur eindeutig deklarieren. Dies bedeutet, dass der Compiler (und natürlich auch der Entwickler) jederzeit in der Lage sein muss, eindeutig eine Variable einem bestimmten Gültigkeitsbereich zuzuordnen. Ist dies nicht der Fall, erzeugt der Compiler eine Fehlermeldung und die Variable muss umbenannt werden.

Etwas akademischer ausgedrückt: Ein Gültigkeitsbereich für Variablen wird durch einen Block definiert, der alle geschachtelten Blöcke dieses Bereichs einschließt. Daher ist die Deklaration einer Variablen in einem Block mit dem gleichen Namen wie im umschließenden Block nicht zulässig.

Genug der Theorie, sehen Sie sich die folgenden Beispiele an, dann wird dieses Konzept sehr schnell klar. Nehmen wir an, Sie haben eine Klasse zur Darstellung und Speicherung von Positionen erstellt.

```
// funktionierende deklaration
class Position {
   // felder der klasse
   float laenge
   float breite;

   // konstruktor der klasse
   public Position(float laenge, float breite)
   {
      // eindeutigkeit durch präfix herstellen
      this.laenge = laenge;
      this.breite = breite;
   }
}
```

Listing 5.10: Eindeutigkeit einer lokalen Variablenreferenz durch Präfix

Da in C# die Parameter einer Funktion Vorrang vor den Feldern der Klasse besitzen, bezieht sich die Angabe bereits auf den entsprechenden Parameter des Konstruktors. Um daher bei der Zuweisung zu den Feldern der Klasse Eindeutigkeit herzustellen, wird das Präfix `this` verwendet (welche die Instanz selbst bezeichnet). In der folgenden Klassendefinition tritt ein Problem auf, welches zu einer Fehlermeldung des Compilers führt.

```
// fehlerhafte deklaration
class Position {
   // felder der klasse
   float laenge
   float breite;

   // konstruktor der klasse
   public Position(float laenge, float breite)
   {
      // eindeutigkeit durch präfix herstellen
      this.laenge = laenge;
      this.breite = breite;
   }
   private float PolDistanz()
   {
      float poldist;
      // nachfolgende deklaration kann nicht
      // eindeutig aufgeloest werden.
      float breite;
```

```
      // hier berechnung der poldistanz
      return poldist;
   }
}
```

Listing 5.11: Fehler durch mehrdeutige lokale Variablenreferenz

Hier kann die Bedeutung des Bezugs `breite` nicht eindeutig aufgelöst werden. Es existiert in der Klasse ein Feld `breite`, welches auch in der Methode verfügbar ist, die Methode selbst definiert aber ebenfalls eine lokale Variable `breite`. Es existiert auch keine Möglichkeit, dies durch einen Präfix eindeutig zu kennzeichnen, daher ist die innere Deklaration ungültig und der Name der Variablen muss geändert werden.

Gültigkeit von Schleifenvariablen

Die Gültigkeit von Schleifenvariablen bei einer `for`-Schleife ist ein Sonderfall, da hier die Laufvariable der Schleife auch innerhalb der `for`-Schleife definiert werden kann. Nach den Regeln für die Gültigkeit einer Variablen innerhalb des Blocks der Definition kann auf diese Variable nur in der Anweisung oder dem Anweisungsblock zugegriffen werden, der auf die `for`-Anweisung folgt.

```
using System;
class Beispiel  {
   for( int i = 0; i < 10; i++)
      {
         if (i == 8) break;
         Console.WriteLine("I hat den Wert {0}",i);
      }
      if (i != 10) // FEHLER! i nicht mehr gültig!
         { ... weiterer code ... }
}
```

Lokale Variablen sind ab dem Augenblick verfügbar, in dem die Programmausführung den Block der Deklaration erreicht und werden in dem Augenblick nicht mehr verfügbar, in dem die Programmausführung den Deklarationsblock verlässt.

Darüber hinaus existiert noch eine zusätzliche Sicherung in C#: Die Verwendung einer lokalen Variablen im Deklarationsblock an einer Stelle *vor* der tatsächlichen Deklaration in diesem Block wird als Fehler betrachtet.

Array-Elemente

Elemente eines Arrays sind verfügbar, solange das Array verfügbar ist. Die Notwendigkeit der Zuweisung eines Initialwertes an eine Variable wird bei Array-Elementen anders gehandhabt als bei den restlichen Variablen. Hier gelten alle Elemente von Arrays als initial zugewiesen. Sowohl Arrays aus Werttypen als auch Arrays aus Referenztypen erlauben den Zugriff auf ihre Elemente, auch ohne vorherige Initialisierung.

```
using System;
class Test {
   public static void Main()
   {
      int[] v = new int[10];
      // der nachfolgende code ist erlaubt
      Console.WriteLine("v[0] = {0}",v[0]);
   }
}
```

Listing 5.12: Array-Mitglieder müssen nicht initialisiert werden

Dies ist keine Besonderheit, welche die Entwickler von C# für Arrays vorgesehen haben, sondern aufgrund der Flexibilität möglicher Operationen mit Arrays ist der Compiler nicht in der Lage, in allen möglichen Situationen sicherzustellen, dass den Elementen ein initialer Inhalt zugewiesen wurde. Um also nicht reproduzierbare Ergebnisse zu vermeiden, verzichtet der Compiler ganz auf eine Überprüfung dieser Zuweisungen.



Wertparameter

Verwechseln Sie Wertparameter nicht mit Werttypen!

Jeder Parameter einer Funktion, der nicht mit einem der beiden Schlüsselworte `ref` oder `out` deklariert wurde, ist ein Wertparameter (für die Profis: *by value passing*). Diese Variablen sind verfügbar, sobald die Funktion, für die der Parameter deklariert wurde, aufgerufen wird und werden mit der Rückkehr der Programmausführung von dieser Funktion ungültig.

Die Bedingung der initialen Zuweisung eines Inhalts vor der ersten Verwendung wird durch die Belegung mit dem Wert, der beim Aufruf der Funktion angegeben wurde, erfüllt.

Referenzparameter

Alle Parameter einer Funktion, welche mit dem Schlüsselwort `ref` deklariert wurden, sind Referenzparameter. Referenzparameter sind Variablen, die keinen neuen Speicher für einen Inhalt belegen, sondern stattdessen auf den Speicherplatz der übergebenen Variablen zeigen und somit auch immer den direkten Inhalt der übergebenen Variablen besitzen.

Ein Referenzparameter muss einen initialen Inhalt zugewiesen bekommen haben, bevor er als Referenzparameter verwendet werden kann. Innerhalb einer Struktur gilt das Schlüsselwort `this` als gültiger Wert für einen Referenzparameter auf die Struktur selbst.

Ausgabeparameter

Wird ein Parameter mit dem Schlüsselwort `out` deklariert, gilt er als Ausgabeparameter. Ebenso wie ein Referenzparameter sind Ausgabeparameter keine neuen Speicher für einen Inhalt, sondern zeigen stattdessen auf den Speicherplatz der übergebenen Variablen und besitzen somit auch immer den direkten Inhalt der übergebenen Variablen. Allerdings unterscheiden sich die Ausgabeparameter von Referenzparametern in den folgenden Punkten.

- Ausgabeparameter müssen nicht initialisiert werden, um verwendet werden zu können (warum, erfahren Sie in den nächsten drei Punkten).
- Nach dem Aufruf einer Funktion gelten alle Variablen, welche als Ausgabeparameter übergeben wurden, als mit einem initialen Wert versehen.
- Innerhalb einer Funktion gelten alle Variablen, welche als Ausgabeparameter übergeben wurden, als nicht initialisiert.
- Jede Variable, welche als Ausgabeparameter übergeben wurde, muss vor dem Ende der Funktion einen Inhalt zugewiesen bekommen.

5.6.4 Standardwerte

Alle statischen Variablen, Instanzvariablen und Array-Elemente bekommen einen Standardwert zugewiesen. Für Werttypen ist dies der Wert des Standardkonstruktors, für Referenztypen ist dies `null`.

In der Regel wird diese Initialisierung so durchgeführt, dass der Speichermanager im entsprechenden Speicherbereich der Variablen alle Bits auf Null setzt, bevor der Speicherbereich einer Anforderung zugewiesen wird. Dennoch kann mit Ausnahme von Array-Elementen keine Variable in einem Ausdruck benutzt werden, ohne dass sie im Programmcode initialisiert wurde (siehe ***Initialisierung von Variablen***).

5.7 Zusammenfassung

Auch an dieser Stelle wieder eine kurze Zusammenfassung des Kapitels:

- C# unterscheidet bei der technischen Implementierung der Datentypen zwischen den beiden Kategorien Werttypen und Referenztypen. Ein Werttyp ist ein direkter Speicherort der Daten und Werttypen werden beim Aufruf von Methoden auf dem Stack als Kopie übergeben. Referenztypen speichern in ihren Variablen nicht die Daten, sondern einen typischeren Zeiger (die Referenz) auf den Inhalt. Beim Aufruf von Methoden werden diese Referenzen übergeben und nicht eine Kopie der Daten.
- Als Boxing bzw. Unboxing bezeichnet man den Prozess der Umwandlung eines Werttyps in einen Referenztyp und wieder zurück. Durch das 'Einpacken' in eine Objektbox kann der Inhalt der Variablen über eine Referenz angesprochen werden. Damit ist es möglich, auch die einfachen Datentypen vom Datentyp `object` abzuleiten.
- In C# muss jede Variable vor ihrer ersten Verwendung initialisiert werden. Ist dies nicht der Fall oder wird eine Variable in einem Block vor dem Zeitpunkt ihrer Deklaration verwendet, gibt der Compiler eine Fehlermeldung aus.
- Mit Hilfe des Modifikators `ref` können Werte an eine Methode per Referenz übergeben und auch Werttypen geändert werden. Der Modifikator `out` bietet eine sichere Möglichkeit, eine Funktion mehr als einen Wert liefern zu lassen.

5.8 Übungen



1. Welche Vorteile bietet das sog. Boxing?
2. Welche Bedeutung besitzt die Angabe von 'M' hinter einer Zahlenkonstanten?
3. Warum sollten Sie vermeiden, mehrere Strings beispielsweise in einer Schleife mit dem +-Operator zu verketten? Worin liegt der Nachteil dieser Methode?
4. Sie möchten gerne sicherstellen, dass ein Wert während der Programmlaufzeit nicht unabsichtlich verändert werden kann. Eine Deklaration als Konstante scheidet aber aus, da Sie den Initialwert erst zur Laufzeit des Programms bestimmen können. Welche Möglichkeiten haben Sie, um das gewünschte Ergebnis zu erreichen?
5. Sie schreiben eine Methode, die zwei Werte liefert. Welche Möglichkeiten stehen Ihnen bei C# zur Realisierung zur Verfügung?
6. Warum müssen Sie bei einer Variablen, die als `out`-Paramter übergeben wird, nicht vorher eine Initialisierung durchführen?

6 OOP-Grundlagen

Sicher kennen Sie den Spruch 'Gut Ding will Weile haben'. Ein geradezu klassisches Beispiel für dieses Prinzip ist die objektorientierte Programmierung. Immerhin ist die erste Sprache, der man das Attribut objektorientiert verleihen kann, bereits über 30 Jahre alt. Simula wurde Ende der 60er Jahre für Simulationssysteme entwickelt und es dauerte bis in die Mitte der 80er Jahre, bis das Thema allmählich einen breiteren Raum in der Softwareliteratur einnahm. Aber erst seit einigen Jahren ist die objektorientierte Programmierung in den allgemeinen Werkzeugkasten der Softwareentwicklung gewandert.

Zum Thema OOP wurden bereits ganze Regale an Büchern geschrieben und manche Diskussionen über OOP und ihre Details nehmen Ausmaße und einen geradezu religiösen Eifer an, der die mittelalterlichen Kreuzzüge als eine kleinere Meinungsverschiedenheit erscheinen lässt.

Sie werden in diesem Buch von einem weiteren dieser Traktate verschont, der Sinn dieses Kapitels ist lediglich, mit dem Schwerpunkt C# eine solide Grundlage für die nachfolgenden Informationen zu schaffen. Dem Titel des Buches folgend erhalten Sie Informationen für Umsteiger mit Vorkenntnissen. Viele andere Programmiersprachen bieten aber keine vollständige Objektorientierung, daher werde ich an dieser Stelle etwas ausführlicher als sonst und gehe stärker auf die Grundlagen der objektorientierten Programmierung ein. Erfahrene Programmierer, die regelmässig Klassen entwerfen, können dieses Kapitel überspringen, für VB-Programmierer und Umsteiger älterer prozeduraler Sprachen ist die Lektüre sicher empfehlenswert.

OOP ist natürlich nicht im Vakuum oder im Elfenbeinturm entstanden, sondern knüpft an bereits vorhandene Techniken der Softwareentwicklung an. Aufbauend auf den Gedanken der modularen Programmierung, dass sich ein Softwaresystem aus mehreren, modularen und eigenständigen Einheiten zusammensetzt, bieten objektorientierte Programmiersprachen bereits sprachliche Möglichkeiten, diesen Ansatz zu optimieren. Traditionelle (prozedurale) Programmiersprachen boten zwar bereits Möglichkeiten zur Bildung komplexer Datentypen, wie

Sie dies bei den structs und Arrays kennen gelernt haben. Dennoch wurden immer noch getrennt davon Routinen programmiert, welche diese Daten bearbeiteten. Bei komplexen Systemen wurde diese künstliche logische Schranke zwischen den Daten und dem Code, der diese Daten bearbeitete, ein echtes Hindernis.

Der OOP-Ansatz benutzt eigenständige Objekte, eine Kombination aus Daten und dem Code, der diese Daten bearbeitet, als grundlegende Bausteine einer Anwendung. Vergleichbar einem struct, der auch gleich die Funktionen zur Bearbeitung seiner Datenfelder enthält, ist ein Objekt eine Zusammenfassung von Code und Daten, die auch zusammengehören. Insofern stellt ein struct die Vorstufe zu einem Objekt dar, wenn auch mit einigen Einschränkungen, wie Sie gleich erfahren werden.

Im Gegensatz zur Meinung vieler Einsteiger führt die objektorientierte Programmierung nicht automatisch zu leichter erstellbarem Code oder weniger Schreibarbeit. Fast alle objektorientierten Anwendungen verfügen aber über wesentlich leichter wartbaren Code, letztlich das einzige Kriterium für eine Anwendung, deren Lebenszeit länger als eine einmalige Ausführung ist.

Eine echte OOP-Sprache unterstützt bzw. realisiert die folgenden Prinzipien der objektorientierten Softwareentwicklung, welche die Vorteile dieser Methode erläutern.

6.1 OOP-Konzepte

Falls Sie zu den Lesern gehören, die entweder die Praxis dem theoretischen Hintergrund vorziehen oder aber bereits mit den nachfolgenden Konzepten vertraut sind, können Sie gleich zum nächsten Kapitel weiterspringen. An dieser Stelle auch wieder ein Rat an die Umsteiger von VB: Auch wenn Microsoft damit geworben hatte, dass Visual Basic 6 eine objektorientierte Programmierung erlaubte, fehlten der Sprache doch einige Möglichkeiten, die zur Grundausstattung einer 'echten' OOP-Sprache gehören, wie zum Beispiel Vererbung.

6.1.1 Abstraktion

Das Konzept der Abstraktion war letztlich einer der Auslöser für die Entwicklung einer Sprache wie Simula. Es erlaubt dem Entwickler, sich auf die Modellierung eines Problems in Begriffen und Objekten der realen Welt zu konzentrieren. Wenn Sie als Architekt für ein neues Bürogebäude das Verhalten einer Gruppe von Aufzügen simulieren wollen, wollen Sie sich nicht mit Aufzugsarrays, Kapazitätsfeldern und Funktionen für Start und Stop herumschlagen. Sie wollen eine Reihe von 'Aufzugsobjekten' haben, deren Eigenschaften (Daten) und deren Verhalten (Code) sie einstellen können, um die Beförderung der Personen zu optimieren. Anstelle von Datenstrukturen und davon getrenntem Code können Sie sich mit Kollegen in vertrauten Begriffen austauschen und erledigen so Ihre Arbeit schneller und (hoffentlich) auch besser. Je höher der Abstraktionsgrad, den eine Programmiersprache unterstützt, umso einfacher ist es, Dinge des realen Lebens im Computer nachzubilden.

6.1.2 Kapselung

Kapselung ist die Weiterentwicklung der modularen Programmierung. Jedes Modul sollte möglichst eigenständig und unabhängig von den anderen sein, um ungewollte Seiteneffekte zu vermeiden. Genau dies ist auch das Ziel in der objektorientierten Entwicklung mit C# und anderen Sprachen. Objekte sind möglichst vollständig gekapselte Einheiten mit Eigenschaften und Verhalten, die ein Entwickler ansonsten als 'black box' ansehen kann. Details der Implementierung eines Objekts bleiben anderen Entwicklern, die dieses Objekt benutzen, verborgen. Eine Kommunikation mit dem Objekt erfolgt nur über definierte Schnittstellen. So sind Sie in der Lage, jederzeit interne Details zu verändern, ohne dass dies Auswirkungen auf fremden Code hat.

Soweit die schöne neue Welt. Leider hat die Sache auch hier einen Haken: Kapselung ist eine Möglichkeit, ein Feature von OOP, kein Automatismus, den Sie als Entwickler geschenkt bekommen. Dies ist ein Prinzip, das im Code der Objekte auch implementiert werden will. Verfolgen Sie daher immer das Prinzip, ein System so zu verlassen, wie Sie es vorgefunden haben. Ändert Ihr Objekt eine Einstellung, sollte diese bei der Freigabe des Objekts wieder rückgängig gemacht werden. Lassen Sie keinen direkten Zugriff auf interne Daten des Objekts zu,

schaffen Sie dafür extra 'Zugriffseigenschaften', welche die internen Daten kapseln und vor dem Zugriff anderer Objekte oder Funktionen verbergen.

6.1.3 Kohäsion und Kopplung

Zwei Begriffe, die auch in der Dokumentation zum .NET-Framework immer wieder auftauchen, sind Kohäsion (*cohesion*) und Kopplung (*coupling*). Ideal sind Komponenten (Objekte) mit hoher Kohäsion und niedriger Kopplung. Was bedeutet das?

Kohäsion ist die Fähigkeit eines Objekts, eine Aufgabe und genau diese Aufgabe zu erledigen. Versuchen Sie nicht, die eierlegende Wollmilchsau als Objekt zu erschaffen, sondern konzentrieren Sie sich auf die Entwicklung von unabhängigen Objekten, die jeweils eine Aufgabe gut erledigen und nicht mehrere schlecht. Ein riesiges Kalender/Urlaubs/Feiertags/Anzeige-Objekt ist wesentlich weniger verwendbar als mehrere Objekte für Kalenderdaten, Urlaubsplanung, Feiertagsberechnung und die Anzeige dieser Daten. Kleine Module mit großer Kohäsion sind besser wieder verwendbar und das ist letztlich alles, was in der Softwareentwicklung zählt – das Rad nicht zweimal zu erfinden.

Kopplung ist die gegenseitige Abhängigkeit von Objekten oder Komponenten. Ein Beispiel für ein System mit starker Kopplung ist COM. COM benötigt eine schnelle Verbindung (am besten im gleichen Prozessraum) und ist ein Protokoll, das eine gewisse Binärkompatibilität voraussetzt. Alle COM-Komponenten benötigen das Vorhandensein der Registry. Darüber hinaus sind Aufrufe auch synchron, d.h. das aufrufende Objekt wartet, bis das aufgerufene Objekt seine Arbeit erledigt hat (jeder, der schon mal vergebens auf ein CreateObject auf einem Remote-Server gewartet hat, kennt das). Ein lose gekoppeltes System sind internet-basierende Mailsysteme. Es gibt eine, genau eine Schnittstelle (nämlich SMTP zum Senden), die Operation der Mailzustellung erfolgt dann asynchron und es werden keinerlei Annahmen über den Client gemacht. Wer genügend Selbstüberwindung aufbringt, kann seine Mails sogar per Telnet verfassen ...

Natürlich muss es für Objekte, die kommunizieren, eine gewisse Kopplung geben, nämlich die gemeinsame Schnittstelle. Diese sollte aber minimiert werden auf möglichst nur eine Schnittstelle ohne Hintertüren

oder Alternativen. Wenn ein Objekt beispielsweise eine Eigenschaft `MailCount` besitzt, dann sollten Sie die Anzahl der Nachrichten auch nur so erhalten und nicht noch alternativ über ein internes Feld oder eine zusätzliche Funktion `GetMailCount`.

6.1.4 Vererbung

Vererbung ist die Fähigkeit eines Objekts, das allgemeine Verhalten (Daten und Code) eines anderen Objekts zu übernehmen und dann zu spezialisieren oder zu erweitern. Dies erlaubt ebenfalls die Wiederverwendung von bereits vorhandenem Code und spart damit Entwicklungszeit. Wie oft haben sich z.B. Visual Basic Programmierer gewünscht, ein Formular von einem anderen abzuleiten und nur noch den einen Button zu ergänzen ...

Angenommen Sie programmieren einen Simulator für angehende Fluglotsen. Im Luftraum kann sich eine ganze Menge tummeln: Segelflieger, Propellermaschinen, große Jets, Ultraleichtflugzeuge und so weiter. Jedes dieser Objekte nun komplett von Grund auf zu entwickeln, wäre ein unverhältnismäßig großer Aufwand. Besser wäre es, ein allgemeines Objekt zu entwickeln, ausgestattet mit allem, was die Flugzeuge gemeinsam haben, beispielweise Spannweite, max. Geschwindigkeit, max. Flughöhe, Räderzahl im Fahrweg, Code zur Ausgabe der Kennung und zur Darstellung des Symbols und der ganze Rest.

Nun können Sie davon für ein Flugzeug mit Antrieb ein Objekt ableiten und ergänzen nur noch die Daten und Code für den Antrieb. Ein weiterer Vorteil: Ändern Sie später das Objekt für das allgemeine Flugzeug, erbt die Propellermaschine z.B. automatisch das neue Feld Passagierzahl, ohne eine einzige Zeile Code.

Diese hierarchische Struktur der Vererbung besitzt auch C#. Alle Objekte und Datentypen stammen von `object` ab. Jedes Objekt kann aufgrund dieser Vererbung wie sein Vorfahr behandelt werden. Akzeptiert eine Funktion in C# einen Parameter vom Typ `object`, kann auch jedes davon abgeleitete Objekt übergeben werden.

6.1.5 Polymorphie

Polymorphie bedeutet 'Mehrgestaltigkeit'. Dies ist das fortgeschrittenste der Konzepte der objektorientierten Programmierung und erlaubt es, Code abhängig vom jeweils tatsächlich vorhandenen Objekt auszuführen.

So lässt sich die Methode 'MaleSymbol' unseres allgemeinen Flugzeugobjekts erben und ergänzen oder ändern. Eine Propellermaschine soll schließlich ein anderes Symbol auf dem Radarschirm zeigen als ein Jet. Haben Sie nun ein Array mit 20 Flugzeugobjekten, so können Sie in einer Schleife jeweils einfach den Code für 'MaleSymbol' aufrufen und jedes Objekt 'weiß', welches Symbol es anzuzeigen hat. Dies erleichtert die Softwareentwicklung erheblich.

In den nächsten Kapiteln werden Sie für jedes dieser Prinzipien Beispiele sehen und erfahren, wie diese OOP-Prinzipien in C# implementiert wurden.

6.2 Terminologie

Bevor Sie weiterlesen, hier zuerst einige Begriffe aus der OOP-Welt, deren Bedeutung Sie kennen sollten und die für das Verständnis der nachfolgenden Absätze wichtig sind. In Klammern finden Sie den jeweils in der Literatur verwendeten englischen Fachausdruck.

6.2.1 Klasse (class)

Eine Klasse ist die Definition eines Objektes, also das Rezept für ein Objekt. In C# ist dies die Beschreibung in Code, mit welchen Eigenschaften, Attributen und Methoden das Objekt ausgestattet sein wird. Stellen Sie sich den Zusammenhang zwischen einer Klasse (der Definition) und einem Objekt so vor wie den Zusammenhang zwischen einem Bauplan für ein Haus und dem tatsächlichen Gebäude. Eine einfache Klasse in C# finden Sie im nächsten Absatz.

```
public class Klasse {
   // hier klassenmitglieder eintragen
}
```

Listing 6.1: Die Minimal-Klasse in C#

Die Klasse werde ich im Verlauf dieses Kapitels nach und nach mit Leben füllen, so dass Sie am Ende des Kapitels eine komplette Klasse kennen gelernt haben.

6.2.2 Feld / Attribut (field, member variable)

Ein Objekt ist also eine Zusammenfassung von Code und Daten. Die Daten werden in Feldern bzw. Attributen gespeichert. Eine Klasse 'Säugetier' besitzt beispielsweise ein Attribut 'Augenfarbe' oder 'Gewicht'. Diese internen Datenfelder einer Klasse werden aber in der Regel nicht öffentlich zugänglich gemacht, sondern über Eigenschaften anderen Objekten zur Verfügung gestellt. Näheres erfahren Sie weiter unten bei Kapselung.

```
public class Klasse {
   // hier klassenmitglieder eintragen
   private int fNummer;  // feld
}
```

Listing 6.2: Ein einfache Felddefinition

Hier habe ich ein Feld vom Datentyp `int` definiert, welches den Namen `fNummer` besitzt. Noch können wir damit nicht viel anfangen, aber das ändert sich bald. Auf die Bedeutung des Schlüsselwortes `private` gehe ich im nächsten Kapitel genauer ein, an dieser Stelle nur soviel: Alles, was `private` deklariert wurde, ist außerhalb der Klasse nicht sichtbar.

6.2.3 Eigenschaften (property)

Eine Eigenschaft ist eine Möglichkeit des Zugriffs auf interne Daten und Code einer Klasse. In der Regel dienen Eigenschaften als definierte Schnittstelle, so dass interne Details der Datenspeicherung vor Benutzern der Klasse und deren Objekten verborgen bleiben können. Stellt eine Klasse beispielsweise eine Eigenschaft 'Nachrichtenanzahl' zur Verfügung, ist es für die Benutzer des Objektes unerheblich, ob diese Anzahl intern in einem Attribut gespeichert, über eine Methode zur Zeit der Abfrage erst berechnet oder extern von einem anderen Server geholt wird.

```
public class Klasse {
   // hier klassenmitglieder eintragen
   private int fNummer;  // feld
   // eine eigenschaft definieren
   public int ID // Name der eigenschaft
   {
      get {
         // lesen der eigenschaft
         return fNummer;
      }
      set {
         // setzen der eigenschaft
         fNummer = value;
       }
   }
}
```

Listing 6.3: Eine Eigenschaft kommt zur Klasse dazu

Nun besitzt die Klasse eine Eigenschaft mit dem Namen `ID` und dem Datentyp `int`. Diese Eigenschaft kann gelesen und geschrieben werden, dafür existieren innerhalb der Definition die beiden Abschnitte `get` und `set`. Möchten Sie eine Eigenschaft erstellen, welche vom Anwender der Klasse nur gelesen werden kann, dann lassen Sie einfach den `set`-Abschnitt weg.

6.2.4 Methode (method)

Eine Methode ist ein Stück Code einer Klasse. Methoden werden auch oft als Mitglieder oder Mitgliedsfunktionen einer Klasse bezeichnet. Ich bleibe beim Begriff Methode, der sich eingebürgert hat. Alle Funktionen und Prozeduren einer Klassen sind also Methoden dieser Klasse. In C# existieren Schlüsselwörter, welche diese Methoden selektiv nach 'außen' sichtbar machen oder verbergen. Sie können also genau bestimmen, welche Methoden Ihrer Klasse von anderen aufgerufen werden können. Auch hier bitte ich um etwas Geduld, was die Steuerung dieser Sichtbarkeit und anderer Dinge angeht. Ich werde im nächsten Kapitel sehr detailliert darauf eingehen.

```
public class Klasse {
   // hier klassenmitglieder eintragen
   private int fNummer;  // feld
   // eine eigenschaft definieren
```

```
   public int ID // Name der eigenschaft
   {
      get {
         // lesen der eigenschaft
         return fNummer;
      }
      set {
         // setzen der eigenschaft
         fNummer = value;
       }
   }
   // eine methode
   public double Quadrat(double n) {
      return n*n;
   }
}
```

Listing 6.4: Die erste Methode der Klasse – nicht sehr ausgefallen

Details zur Deklaration der einzelnen Methoden und den möglichen Parametern finden Sie in Kapitel 8 ***Klassen in C#***.

6.2.5 Instanziierung (instantiation, creation)

Dies ist der Prozess des Hausbauens, Staub und Steine. Nein, ernsthaft, der Prozess des tatsächlichen Aufbaus eines Objekts im Speicher anhand der Klassendefinition heißt Instanziierung. Danach ist eine Instanz einer Klasse, ein Objekt, vorhanden. Die Analogie zum Hausbau als der Erzeugung eines tatsächlichen physischen Hauses aus dem Bauplan ist also durchaus gerechtfertigt, vor allem, da für eine Instanz einer Klasse ja auch Ressourcen (wie Arbeitsspeicher) belegt und zugeordnet werden müssen.

Also muss unsere Klasse nun instanziiert (ein Objekt aus der Klasse erzeugt) werden. Dies kann die Klasse à la Münchhausen nicht von selbst, sondern eine andere Klasse erzeugt in irgendeiner Methode eine Instanz unserer Klasse. Daher brauchen wir eine 'Versuchsklasse', welche eine Instanz unserer Klasse erzeugt.

```
using System;
public class Test {
   public static void Main() {
      Klasse obj;
```

```
        // eine instanz erzeugen
        obj = new Klasse();
        // eine methode aufrufen
        Console.WriteLine( obj.Quadrat( 3.5 ) );
    }
}
```

Listing 6.5: Eine Instanz der Klasse erzeugen

6.2.6 Konstruktoren (constructor)

Was genau passiert beim Erzeugen eines Objektes (der Instanz einer Klasse)? Nachdem die Speicherverwaltung den Speicher angefordert und das Objekt 'gebaut' hat, wird der Konstruktor einer Klasse aufgerufen. Ein Konstruktor?

Dabei handelt es sich im Prinzip um eine Methode, aber eine mit speziellen Aufgaben. Bei der Erzeugung eines Objekts ist nämlich der ideale Zeitpunkt für Initialisierungsaufgaben, das Belegen von Variablen, Öffnen von Dateien oder der Erledigung anderer Aufgaben, welche die Klasse zur Herstellung ihrer Arbeitsfähigkeit braucht. Daher nennt man diese Methode eben Konstruktor.

Für jede Klasse kann es mehr als einen Konstruktor geben. Damit sind Sie in der Lage, flexibel auf die jeweiligen Anforderungen zu reagieren. Bei unserer Beispielklasse möchten Sie vielleicht die Eigenschaft ID bei der Erzeugung des Objektes gleich mit einem Wert füllen, ein anderes Mal vielleicht nicht, dann muss aber eine interne Routine aufgerufen werden, welche einen Wert generiert. Unterschieden werden die einzelnen Konstruktoren wie andere Methoden anhand der Typsignatur. In C# wird der Konstruktor einer Klasse erstellt, indem Sie eine Methode erzeugen, welche den Namen der Klasse trägt.

```
public class Klasse {
   // hier klassenmitglieder eintragen
   private int fNummer;  // feld
   // konstruktoren stehen meist zu beginn
   public void Klasse(int x) {
      fNummer = x;
   }
   // eine eigenschaft definieren
   public int ID // Name der eigenschaft
```

```
   {
      get {
         // lesen der eigenschaft
         return fNummer;
      }
      set {
         // setzen der eigenschaft
         fNummer = value;
       }
   }
   // eine methode
   public double Quadrat(double n) {
      return n*n;
   }
}
```

Listing 6.6: Die Klasse besitzt nun einen Konstruktor

6.2.7 Objekt (object, class instance)

Als Objekt wird die physische Instanz einer Klasse bezeichnet, also das Ding, welches im RAM liegt und nach dem Rezept der Klassendefinition entstanden ist. Jedes Objekt besitzt seinen eigenen Speicher für Eigenschaften. Um bei unserem Beispiel zu bleiben, mit unserem Bauplan lassen sich drei Häuser bauen. Diese drei Häuser belegen drei physische Grundstücke, können aber durchaus jedes in einer anderen Farbe (Eigenschaft) gestrichen und mit anderen Ziegeln (Eigenschaft) gedeckt sein.

6.2.8 Freigabe / Destruktoren (destructor)

Die Freigabe eines Objekts bedeutet nichts anderes, als dass dieses Objekt nicht mehr benutzt wird und der Speichermanager des Betriebssystems das Objekt zerstören und dessen Speicher wieder freigeben kann. Beim .NET-Framework übernimmt diese Aufgabe die automatische Speicherverwaltung (garbage collection).

Auch bei der Zerstörung eines Objektes kann es sinnvoll sein, Aufräumarbeiten durchzuführen oder z.B. Datenbankverbindungen abzubauen. Daher existiert in der objektorientierten Programmierung auch das Gegenstück zu einem Konstruktor, der so genannte Destruktor.

Einen Destruktor in C# erstellen Sie, indem Sie eine Methode mit dem Namen der Klasse, aber einem vorangestellten ~ erzeugen. Aufgrund der automatischen Speicherverwaltung ist aber nicht sicher, wann dieser Destruktor aufgerufen wird, da der Speichermanager der .NET-Umgebung dies eigenständig entscheidet. Verlassen Sie sich im Gegensatz zur bisherigen COM-Programmierung unter Windows also nicht darauf, dass und wann der Destruktor Ihrer Klasse aufgerufen wird. Im Normalfall brauchen Sie daher in C# keinen Destruktor zu programmieren, da sich die automatische Speicherverwaltung um das 'Aufräumen' nicht mehr benötigter Objekte kümmert.

Microsoft empfiehlt in seinen Dokumenten, eine entsprechende Methode (z.B. mit dem Namen `Dispose()`) zu erstellen, welche die 'Aufräumarbeiten' unabhängig vom Destruktor bei Bedarf durchführt. Mehr dazu erfahren Sie im Kapitel 17 ***Windows Forms*** und im Kapitel 21 ***Garbage Collection***.

6.2.9 Alles ist ein Objekt

Vor C# hatten viele objektorientierte Programmiersprachen ein Problem. Eine wirklich konsequente Objektorientierung (jeder Datentyp ist ein Objekt, alles stammt von einem gemeinsamen 'Urahn' ab) war nur auf Kosten der Performance möglich (z.B. Smalltalk). Andere Programmiersprachen wie Java wichen diesem Problem dadurch aus, dass die einfachen Datentypen wie ein `int` kein Objekt, sondern auf wundersame Art und Weise 'einfach da' waren, was das ganze schöne OOP-Konzept wieder verwässert.

C# setzt diesen objektorientierten Ansatz konsequent um. Hier ist alles ein Objekt! Ein Objekt hat, wie Sie gerade erfahren haben, Methoden und jedem Objekt stehen alle Methoden seiner Vorfahren zur Verfügung. Das gemeinsame Basisobjekt `object` definiert eine Methode `ToString`, welche (oh Wunder) für eine Ausgabe des Objekts als Zeichenkette sorgt. Nachdem in C# alles, auch die einfachen Datentypen, von `object` abstammt, ist der Code der folgenden Zeile völlig legal:

```
System.Console.WriteLine( 3.Tostring() );
```

Weitere Informationen, wie die .NET Laufzeitumgebung die Umwandlung zwischen Wert- und Referenztypen erledigt, finden Sie unter Boxing in Kapitel 5 ***Daten, Typen, Variablen***.

6.3 Zusammenfassung

Ich habe Ihnen in diesem Kapitel die wichtigsten Grundlagen der objektorientierten Programmierung vorgestellt und die häufigsten Begriffe erläutert. Für einen Einsteiger in die Materie ist das eine ganze Menge neue Information auf einmal. Lassen Sie uns daher das Material zusammenfassen.

- Die in der objektorientierten Programmierung vorherrschenden Konzepte der Softwareentwicklung zielen darauf ab, logisch zusammengehörenden Code und Daten zu funktionalen Objekten zusammenzufassen. Darüber hinaus soll die objektorientierte Programmierung die Wiederverwendung von Code bzw. Funktionalität erleichtern. Durch Abstraktion wird es dem Entwickler und seinen Kunden ermöglicht, sich in Begriffen des Problemraums auszudrücken und Details der Implementierung zu verbergen.
- Die grundlegende Konstruktion in der objektorientierten Programmierung ist die Klasse. Als 'Bauplan' für ein Objekt enthält sie die Datenfelder, Eigenschaften und Methoden, über die ein Objekt verfügt. Je nach Planung sind einige dieser Klassenmitglieder nach außen hin sichtbar oder nicht.
- Die Instanz einer Klasse nennt man Objekt. Wird dieses Objekt erstellt, wird ein eventuell in der Klasse vorhandener Konstruktor aufgerufen. Für den Abbau des Objekts ist in C# die Garbage Collection verantwortlich, welche vor der Zerstörung eines Objektes noch dessen Destruktor aufruft. In C# ist allerdings der Zeitpunkt dieses Aufrufs nicht festgelegt. Es kann sogar sein, dass ein Destruktor (Finalizer) gar nicht aufgerufen wird.

6.4 Übungen



1. Erläutern Sie kurz den Unterschied zwischen einer Klasse und einem Objekt.
2. Welchen Vorteil bieten Eigenschaften gegenüber öffentlichen Feldern?
3. Sollte die Kopplung oder die Kohäsion einer Klasse maximiert werden?
4. Welche Designregeln sollten Sie für die Entscheidung zwischen Eigenschaften und Methoden zur Änderung von Daten der Klasse beachten?

7 Klassen in C#

In diesem Kapitel setze ich voraus, dass Sie mit den Grundbegriffen der objektorientierten Programmierung in einer anderen Programmiersprache vertraut sind oder das Kapitel ***OOP-Grundlagen*** gelesen haben.

7.1 Klassendefinition

Die allgemeine Form der Definition einer Klasse in C# folgt diesem Schema:

```
<sichtbarkeit> class <klassenname>
{
   [ <mitgliedsdefinition> ]
}
```

Listing 7.1: Allgemeine Form einer Klassendefinition

Kleiner Hinweis an die C++-Fraktion: Nach der Definition der Klasse steht **kein** Strichpunkt.

Für den Namen der Klasse gelten die üblichen Regeln für Bezeichner, die ich in Kapitel 8 ***Anweisungen und Operatoren*** erläutere.

7.2 Zugriff auf Klassen und Mitglieder

Die Sichtbarkeit einer Klasse innerhalb des Programms und das Thema Zugriff auf Mitglieder einer Klasse werden in den meisten Büchern zum Thema OOP immer erst relativ spät erläutert. Daher an dieser Stelle zuerst die Erklärung der so genannten Zugriffsmodifikatoren (der englische Ausdruck *accessibility modifiers* klingt irgendwie vertrauter).

Eines der Grundprinzipien der objektorientierten Programmierung ist die Kapselung, das bedeutet die größtmögliche Abschottung einzelner Module und Klassen untereinander, so dass Objekte von außen nur noch den Zugriff auf Mitglieder der Klasse besitzen, der unbedingt notwendig ist.

7.2.1 Zugriff auf Klassen

Für die nachfolgenden Beispiele verwende ich die folgende Klasse, welche ich im Laufe dieses und der nächsten Kapitel nach und nach erweitern werde.

```
class Datum {
   int jahr;
   byte monat;
   byte tag;
}
```

Listing 7.2: Die (noch unfertige) Ausgangsbasis für die Datumsklasse

Bereits jetzt fällt auf, dass an dieser Stelle etwas Entscheidendes fehlt: Ich habe keinerlei Informationen, wer alles wie auf die Klasse und ihre Mitglieder zugreifen darf. Wäre der Zugriff unbeschränkt möglich, würde dies das Prinzip der Kapselung verletzen. Wäre jeder Zugriff auf Daten der Klasse nur für Mitglieder der Klasse möglich (was in diesem Beispiel auch der Fall ist), wäre die Klasse nicht zu gebrauchen und Sie und ich könnten uns mangels Arbeit faul an den Strand legen …

Aus diesem Grund lassen sich sowohl für Klassen als auch für Mitglieder einer Klasse entsprechende Modifikatoren für den Zugriff angeben. Beginnen wir mit der Klasse:

```
public class {}
```

Definiert eine Klasse mit 'öffentlichem' Zugriff, d.h. jede andere Klasse kann auf diese Klasse zugreifen

```
internal class {}
```

Definiert die Klasse als 'intern'. Der Zugriff auf diese Klasse ist auf das Assembly beschränkt, in welchem die Klasse definiert wurde, da ein Assembly die grundlegende Einheit einer Anwendung innerhalb der .NET-Laufzeitumgebung ist.

```
abstract class {}
```

Dieses Schlüsselwort gehört zwar nicht direkt zu den Modifikatoren für den Zugriff auf eine Klasse. Ich erwähne `abstract` und `sealed` hier aber nur kurz der Vollständigkeit halber, da ich im Abschnitt über Vererbung

und Versionierung von Klassen detaillierter darauf eingehe. Eine abstrakte Klasse enthält in ihren Methoden keinen Code und muss daher immer vor ihrer Verwendung abgeleitet werden, da eine Nachfahrenklasse dann den eigentlichen Code für die Methode implementiert. Damit sind abstrakte Klassen sozusagen eine Vorstufe zu Interfaces.

```
sealed class {}
```

Eine `sealed` (versiegelte) Klasse entspricht in Java einer *final class* und schließt die Vererbungskette ab. Von dieser Klasse können Sie also keinen Nachfahren mehr ableiten.

7.2.2 Zugriff auf Klassenmitglieder

Hier ist die mögliche Bandbreite größer, Sie können den Zugriff auf die Methoden und Eigenschaften der Klasse feiner regeln:

```
class Datum {
   [ <modifier> ] bool IstSchaltjahr() {
   // code für IstSchaltjahr
   }
}
```

Listing 7.3: Allgemeine Form der Definition eines Klassenmitglieds

Hier kann `<modifier>` die folgenden Werte annehmen:

private

Dies ist der Standard für Mitglieder einer Klasse, wenn kein Modifikator angegeben wird. Damit ist der Zugriff auf dieses Mitglied der Klasse nur innerhalb der Klasse möglich. Auf diese Art und Weise wird das Prinzip der Kapselung durch die Annahme der restriktivsten Zugriffsmöglichkeit als Standard unterstützt. Sie können also nicht versehentlich interne Felder Ihrer Klasse freigeben, indem Sie vergessen, einen Zugriffsmodifikator anzugeben.

protected

Bedeutet wie bei Java, Delphi, etc. die Beschränkung des Zugriffs auf die definierende Klasse oder alle davon abgeleiteten Klassen.

internal

Beschränkt den Zugriff auf alle Klassen im gleichen Assembly (dem Quelltextmodul). Dies war bei Delphi beispielsweise bei der Definition einer Klasse ebenfalls implizit möglich, C# schafft hier eine saubere Definition.

protected internal

Wie Sie unschwer erkennen können, handelt es sich bei diesem Modifikator um eine Mischform der beiden letzten Angaben. Damit kann auf diese Klasse entweder von der definierenden Klasse, jeder abgeleiteten Klasse oder innerhalb des gleichen Quellcode-Moduls (Assembly) zugegriffen werden.

Für C++ entspricht `protected internal` in etwa den `friend`-Klassen, wenn auch nicht komplett. Dieser Zugriff dürfte aber für eine leichtere Portierung von Quellcode mit `friend`-Klassen ausreichen.

public

Dies ist der unbeschränkte, öffentliche Zugriff für alle Klassen aus allen Assemblies.

7.3 Methoden

Neben Datenfeldern enthalten Klassen auch den zur Verarbeitung der Felder benötigten Code. Im Gegensatz zur prozeduralen Programmierung werden also nicht Werte aus Strukturen oder Variablen abgefragt und mit Prozeduren und Funktionen bearbeitet, sondern führen Methoden einer Klasse aus. Sie sagen also dem Objekt gewissermaßen, was es zu tun hat. Damit benötigen Sie wesentlich weniger Informationen über den internen Aufbau eines Objekts und verringern damit die Kopplung zwischen Objekten und Modulen in Ihrer Anwendung.



VB

Wenn Sie von Visual Basic umsteigen, werden Sie sich jetzt möglicherweise zwei Fragen stellen: was sind Methoden und wo sind meine `Sub`s und `Function`s geblieben? Zur Abwechslung die Antwort auf die zweite Frage zuerst: In C# ist alles eine Funktion, es gibt also keinen Unterschied zwischen einer Prozedur und einer Funktion. Soll eine Funktion keinen Wert zurückliefern, geben Sie als Rückgabetyp `void` an.

`void` steht als Platzhalter für 'kein Rückgabetyp'. Nun zu Frage eins: Wie ich Ihnen bereits bei unserer ***C# Sightseeing Tour*** geschildert habe, ist C# eine vollständig objektorientierte Sprache. Daher existiert aller Code auch nur als Teil einer Klasse (einer Objektdefinition, mehr im Kapitel ***OOP-Grundlagen***) und Funktionen, welche Bestandteil einer Klasse sind, werden als Methoden bezeichnet.

Ebenso wird bei jedem Aufruf einer Methode auch die Liste der Parameter angegeben, notfalls eben die leere Liste, wenn keine Parameter übergeben werden. Diese Änderung wird übrigens auch in Visual Basic.NET Einzug halten. Auch hier werden Sie in Zukunft immer die Klammern angeben und auch beim Aufruf ohne Speichern des Rückgabewertes die Liste der Parameter in Klammern setzen. Damit ist wieder eine VB-Spezialität Geschichte, die viele Umsteiger von anderen Sprachen eine Menge Nerven gekostet hat …

7.3.1 Definition

Bei der Deklaration und dem Aufruf von Methoden folgt C# überwiegend dem Modell von C++ bzw. Java. Jede Methode besitzt einen Namen, einen Datentyp für das zurückgelieferte Ergebnis und eine Liste von Parametern. Das allgemeine Schema sieht so aus:

```
<modifikator>  <rückgabetyp>  <name>  <parameter>
```

Listing 7.4: Allgemeines Schema zur Definition einer Methode

Der Modifikator bestimmt die Sichtbarkeit der Methode, deren mögliche Werte ich bereits oben erläutert habe.

7.3.2 Parameterübergabe

Die Parameter für einen Methodenaufruf werden in runde Klammern eingeschlossen an die Methode übergeben. Im Gegensatz zu anderen Programmiersprachen wie Visual Basic müssen bei einer leeren Parameterliste ebenfalls die Klammern (dann als leeres Klammerpaar) angegeben werden.

Für die nachfolgenden Beispiele werde ich an dieser Stelle eine kleine Funktion definieren, welche die höchste erlaubte Geschwindigkeit auf

einer Autobahn für ein bestimmtes Land ausgibt. Als Kennung für das Land verwende ich das internationale KFZ-Kennzeichen.

```
int MaxAutobahn(string land) {
   switch land {
      case "CH":
         return 120; break;
      case "F":
         return 130; break;
      }
}
```

Listing 7.5: Bestimmung der erlaubten Höchstgeschwindigkeit

Nun, für diese zwei Länder klappt das ganz gut. Wenn Sie beispielsweise mit `speed = MaxAutobahn("CH")` das Limit für die Schweiz wissen möchten, erhalten Sie in der Variablen `speed` den Wert `120`.

Jetzt mache ich die Sache komplizierter. Was passiert bei der Abfrage des Wertes für Spanien? Dieser Fall ist nicht definiert, ich muss den Code also um die folgenden Zeilen am Ende der switch-Anweisung ergänzen.

```
      default:
         return -1;
         break;
```

Damit erhalten Sie bei unbekannten Ländern eine 'Höchstgeschwindigkeit' von -1 und können diesen Fall abfangen. Eine andere (und bessere) Möglichkeit wäre das Auslösen einer Exception, allerdings behandle ich das Thema Fehlerbehandlung erst nach den Klassen. Daher bleiben wir für dieses Mal bei einem speziellen Rückgabewert.

Kommen wir zurück nach Deutschland, wo noch der Grundsatz 'freie Fahrt für freie Bürger' herrscht und uns jedes Jahr Tausende von Auto-Touristen beschert, die ebenfalls einmal ihrem Bleifuss frönen möchten.

Eine Möglichkeit würde so aussehen:

```
      case "D":
         return 0;
         break;
```

Interessanter sind aber andere Fälle wie eben auch in Italien, wo die Geschwindigkeit abhängig von Hubraum oder Leistung des Fahrzeugs ist. Hier stehen Sie vor dem Problem, in einer Methode mit einem Rückgabewert zwei Informationen liefern zu müssen. Dafür existieren die so genannten `out`-Parameter.

Der Modifikator out

Ein `out`-Parameter ist ein Parameter, welcher in der Methode gesetzt werden kann, auf den Sie aber innerhalb der Methoden keinen Zugriff besitzen. Wie der Name bereits ausdrückt, dient er nur dem Transport von zusätzlichen Informationen nach außen, um den Einsatz zusätzlicher Eigenschaften oder Variablen zu verhindern.

```
int MaxAutobahn(string land, out bool variabel)
{
   switch land {
      case "I":
         return 130;  variabel = true;  break;
      case "F":
         return 130;  variabel = false; break;
      case "D":
         return   0;  variabel = false; break;
      default:
         return -1;   variabel = false; break;
      }
}
```

Listing 7.6: Einsatz eines out-Parameters

Aufgrund der Anforderung des C#-Compilers, dass keine nicht initialisierten Variablen verwendet werden dürfen, muss in diesem Beispiel der Parameter `variabel` vor Verlassen der Methode einen Wert zugewiesen bekommen. Da der `out`-Parameter innerhalb der Methode nicht auf der rechten Seite eines Ausdrucks (also gelesen) verwendet werden darf, ist diese Anforderung beim Eintritt in die Methode erfüllt.

Dieser Modifikator muss beim Aufruf der Methode ebenfalls aus Gründen der Typsicherheit angegeben werden, damit wird der Code auch etwas mehr selbstdokumentierend, da Sie beim Lesen des Aufrufs bereits die Information bekommen, dass es sich hier um einen `out`-Parameter handelt.

```
int speed;
bool v;
speed = MaxAutobahn("I", out v);
```

Listing 7.7: Der Modifikator out muss auch beim Aufruf angegeben werden

Nachdem die Höchstgeschwindigkeit auf Autobahnen in Italien variabel ist (siehe die Definition der Methoden im vorhergehenden Abschnitt), besitzt die Variable `v` nach dem Aufruf den Wert `true`.

Generell sollten Sie für den Transport von Informationen aus Methoden heraus entsprechende `out`-Parameter wählen, wann immer möglich. Die Rückmeldung von Fehlern oder Problemen durch unerwartete Parameter sollte nach Möglichkeit nicht über einen speziellen Rückgabewert, sondern über eine `Exception` erfolgen. Dieses Thema behandle ich im nächsten Kapitel.

Der Modifikator ref

Ab und zu ist es aber im Gegensatz zu `out`-Parametern geradezu notwendig, nicht nur auf einen Parameter zuzugreifen, sondern den Wert dieser Variablen innerhalb der Methode auch ändern zu können. Dazu muss ich etwas weiter ausholen, wie Parameter in C# genau an die Methoden übergeben werden. In C# kann innerhalb einer Methode der Inhalt einer als Parameter übergebenen Variablen normalerweise nicht verändert werden, wenn diese einen Werttyp darstellt. Ein kleiner Codeausschnitt soll das verdeutlichen:

```
// code ...
void Machwas(int p) {
   // code ...
  p = 34;
}
int x = 5;
// Aufruf der Methode
Machwas(x);
// x hat immer noch den Wert 5!
```

Listing 7.8: Parameterübergabe für Werttypen

Der Grund liegt darin, dass Parameter in C# per default als Wert (*by value*) übergeben werden. Die einzige Ausnahme in C# sind Parameter, welche aufgrund ihres Datentyps bereits Referenztypen darstellen (also alle Objekte, Strings und eigenen Klassen).

```
// code ...
void Machwas(object o) {
   // code ...
  o.x = 34;
}
objekt = New ObjektKlasse();
objekt.x = 5;
// Aufruf der Methode
Machwas(objekt);
// objekt.x besitzt nun den Wert 34!
```

Listing 7.9: Parameterübergabe für Referenztypen

Um eine Übergabe als Referenz (*by reference*) auch für Werttypen zu erhalten, verwenden Sie das Schlüsselwort `ref` bei der Deklaration der Parameterliste und dem Aufruf der Methode!

```
void Platziere(ref float x, ref float y) {
   x = Round(x);
   y = Round(y);
}
```

Listing 7.10: Deklaration einer Methode mit ref-Parametern

Aus Gründen der Codesicherheit ist die Vorgabe einer Übergabe einer Wertekopie auf dem Stack eine sinnvolle Annahme. Viele VB-Programmierer wissen dies aus eigener leidvoller Erfahrung, da die Standardeinstellung in VB bisher genau das Gegenteil war. Erst mit VB.NET wird sich dieses Verhalten den anderen Sprachen der .NET-Umgebung anpassen.

C# ist hier sogar besonders typsicher, da sie den Modifikator `ref` auch beim Aufruf der Methode angeben müssen! Andernfalls erwartet C# nämlich eine Wertübergabe.

```
float wx;
float wy;
// code, der wx und wy belegt
Platziere(ref wx, ref wy);
// die angabe von "ref" ist nötig!
```

7.3.3 Überladen von Methoden

Ebenso wie Java, Delphi, C++ und viele andere OOP-Sprachen erlaubt es auch C#, mehrere Methoden innerhalb einer Klasse mit dem

gleichen Namen zu definieren. »Toll, wie soll der Compiler das dann auseinanderhalten«, werden Sie sich nun fragen, wenn Sie dieses als Überladen (*overloading*) bezeichnete Konzept noch nicht kennen gelernt haben. Dies ist ganz einfach möglich. Eine Methode wird auch für Sie als Entwickler nicht nur durch ihren Namen identifiziert. Sie besitzt zusätzlich ja einen Rückgabedatentyp und eine Liste von Parametern, welche ebenfalls einen bestimmten Datentyp besitzen. Die Information aus Anzahl und Typ der Parameter wird als Signatur einer Methode bezeichnet und diese muss zusammen mit dem Namen innerhalb einer Klasse eindeutig sein. Damit kann der Compiler bei Aufrufen die einzelnen Methoden auseinanderhalten.

Hier ein Beispiel für das Überladen von Methoden. Sie haben als Entwickler für hochwertige mathematische Spezialsoftware folgende Klasse entwickelt

```
class Rechnen
{
   public int Addiere(int x, int y) {
      return x + y;
   }
}
```

Listing 7.11: Die Methode Addiere ohne Überladungen

Ihr Konkurrent bringt nun eine Klasse auf den Markt, welche zusätzlich zu `int` sogar in der Lage ist, Fliesskommazahlen zu addieren und Sie müssen nachziehen. Dank Überladen können Sie die Klasse wie folgt ergänzen:

```
class Rechnen
{
   public int Addiere(int x, int y) {
      return x + y;
   }
   public double Addiere(double x, double y) {
      return x + y;
   }
}
```

Listing 7.12: Die Methode Addiere mit einer Überladung

Alle Anwender dieser Klasse sind nun in der Lage, ohne Lernaufwand sofort auch `double`-Werte zu addieren, da der Compiler anhand der Typsignatur die richtige Methode herausfindet. Bestehender Code muss in keiner Weise geändert werden!

```
int summe = Addiere(5, 7);
double ergebnis = Addiere(3.45, 7.001);
```

Sie fragen sich, warum der Rückgabetyp nicht zur Signatur zählt? Nun, Sie können in C# jede Methode auch aufrufen, ohne den Rückgabewert zu verwenden. Damit würde bei gleicher Parametersignatur das Unterscheidungsmerkmal wegfallen und der Compiler wäre nicht in der Lage, herauszufinden, welche Methode Sie aufrufen möchten.

```
class Rechnen
{
   public int Addiere(int x, int y) {
      return x + y;
   }
   public double Addiere(int x, int y) {
      return x + y;
   }
}

Addiere(4, 8);
Addiere(2, 1);
// funktioniert nicht, da nicht unterscheidbar
```

Listing 7.13: Eine nicht auflösbare Überladung

7.3.4 Variable Anzahl von Parametern

Auch in C# ist es möglich, eine Methode zu schreiben, die eine variable Anzahl von Parametern aufnimmt. Damit sind Sie in der Lage, Methoden wie `WriteLine()` zu erzeugen, bei denen der Benutzer nicht von vorneherein auf eine bestimmte Anzahl von Parametern festgelegt ist. Dazu verwenden Sie den Modifikator `params`. Bevor ich Ihnen ein entsprechendes Beispiel zeige, möchte ich noch auf die bestehenden Einschränkungen hinweisen. Schließlich ist eine variable Anzahl von Parametern etwas, bei dem Sie als Entwickler etwas mehr Aufwand bei der Erstellung der Methode haben, nachdem Sie nicht sagen können, wie viele Parameter der Benutzer der Methode übergibt.

Es kann nur ein Schlüsselwort `params` in der Definition einer Methode geben und der mit diesem Modifikator ausgestattete Parameter muss der letzte in der Liste der Methodenparameter sein.

```
public static void UseParams(params int[] list)
   {
      for ( int i = 0 ; i < list.Length ; i++ )
         Console.WriteLine(list[i]);
      Console.WriteLine();
   }
```

Listing 7.14: Eine Methode mit variabler Parameterzahl

Mögliche Aufrufe für diese Methode sehen Sie im nachfolgenden Listingausschnitt. Sie müssen lediglich darauf achten, dass alle übergebenen Parameter vom gleichen Typ sind.

```
UseParams(1, 2, 3);
int[] myarray = new int[3] {10,11,12};
UseParams(myarray);
```

Listing 7.15: Mögliche Aufrufe der Methode

Können Sie nicht sicher vorher sagen, welchen Datentyp der Benutzer als Parameter übergibt, existiert trotzdem eine Möglichkeit. Alle Datentypen erben von dem Datentyp `object`, also deklarieren Sie die Methode einfach so, dass als Parameter ein Array vom Datentyp `object` mit einer variablen Anzahl akzeptiert wird:

```
public static void IchNehmAlles(params object[] parm)
```

Damit sind Sie nun frei in der Wahl der Parameter, müssen aber dafür selbst in der Methode überwachen, ob an der richtigen Position auch der korrekte Datentyp übergeben wird. Wie Sie mehr über einen Datentyp herausfinden, erläutere ich in Kapitel 22 ***Reflection*** und Kapitel 23 ***Attribute***.

7.4 Instanzen erzeugen

Eine neue Instanz einer Klasse erzeugen Sie über den Operator `new` und die Angabe eines Klassennamens. Haben sie für die Klasse einen Konstruktor definiert, übergeben Sie die entsprechende Anzahl von Parametern.

```
class Machwas
{
   public readonly DateTime erzeugt;
   // der konstruktor
   public Machwas()
   {
      erzeugt = DateTime.Now;
   }
}

class TestKlasse
{
   public static void Main()
   {
      Machwas mw = new Machwas();
      Console.WriteLine("mw erzeugt um: {0}",
         mw.erzeugt.ToString());
   }
}
```

Listing 7.16: Erzeugen einer Klasseninstanz mit new

7.5 Konstruktoren und Destruktoren

Wird ein Objekt erzeugt (also die Instanz einer Klasse im Speicher erstellt), sind in vielen Fällen noch Initialisierungen notwendig. Ebenso möchten Sie beim Zerstören des Objekts noch Aufräumarbeiten durchführen. Dafür dienen auch in C# Konstruktoren und Destruktoren.

C++ / Delphi



Destruktoren sind nicht dasselbe wie Destruktoren in C++ oder Delphi und werden bei C# auch aufgrund der anderen Speicherverwaltung wesentlich seltener benutzt!

7.5.1 Konstruktoren

Ein Konstruktor wird, falls vorhanden, automatisch aufgerufen, wenn eine neue Instanz einer Klasse erzeugt wird. Die Definition eines Konstruktors in C# ist eine `public`-Methode (Sie wollen den Konstruktor auch von außen aufrufen können) mit dem Namen der Klasse, welche

keinen Rückgabetyp besitzt (auch nicht `void`, gar nichts) und welcher daher auch keinerlei Werte zurückliefert. Dazu benutze ich wieder die Datum-Klasse.

Datumsberechnung

Die .NET-Laufzeitumgebung bietet mit dem Namensraum System.Globalization eine ganze Reihe von Kalendern an, in denen Sie ein Datum ausdrücken können. Vom gregorianischen Kalender über den moslemischen Hijri-Kalender bis zum thailändischen Kalender ist eigentlich alles da, was Sie für internationale Anwendungen benötigen.

```
class Datum {
   // die internen Datenfelder
   private int jahr;
   private byte monat;
   private byte tag;

   public bool IstSchaltjahr()
   {
      return (jahr % 400 == 0) ||
             ((jahr % 100 != 0) && (jahr % 4 == 0));
   }

   public Datum(int jahr, byte monat, byte tag)
   {
      this.jahr = jahr;
      this.monat = monat;
      this.tag = tag;
   }

}
```

Listing 7.17: Deklaration eines Konstruktors

Wird für eine Klasse kein Konstruktor angegeben, so setzt die .NET-Laufzeitumgebung intern einen leeren Default-Konstruktor ein. Sie können Konstruktoren wie andere Methoden ebenfalls überladen, je nach Parametersignatur wird dann der passende Konstruktor verwendet. Je nach Anforderung kann für einen Konstruktor auch eine Sichtbarkeit von `protected` oder `private` angegeben werden. Wobei ich gestehen muss, für `protected` noch den Sinn verstehen zu können, bei einem `private`-Konstruktor (welcher eine Klasse nicht instanziierbar

macht, da auf den Konstruktor nicht zugegriffen werden kann) fehlt mir sowohl ein sinnvolles Beispiel als auch eine sinnvolle Erklärung …

Auch ein `struct` kann einen Konstruktor besitzen, dieser darf aber nicht parameterlos sein. Darüber hinaus kann ein `struct` nicht mit Vererbung benutzt werden.

7.5.2 Destruktoren

Destruktoren sind ebenfalls wie Konstruktoren spezielle Methoden, welche dann aufgerufen werden, wenn das Objekt zerstört (also die Instanz aufgelöst und der belegte Speicher freigegeben) wird. Hier zeigt sich wieder die Prägnanz einer Sprache aus der C-Familie. Der Name des Destruktors lautet ebenfalls wie der Name der Klasse, nur mit einem vorangestellten Bit-Komplement-Operator (der ~). Syntaktisch gesehen ist der Destruktor also das Komplement eines Konstruktors und dies trifft den Sachverhalt genau.

Im Gegensatz zu Konstruktoren kann es pro Klasse nur einen einzigen Destruktor geben und dieser darf keine Parameter in seiner Deklaration enthalten.

```
class Datum {
   // die internen Datenfelder
   private int jahr;
   private byte monat;
   private byte tag;

   public bool IstSchaltjahr()
   {
      return (jahr % 400 == 0) ||
             ((jahr % 100 != 0) && (jahr % 4 == 0));
   }

   public Datum(int jahr, byte monat, byte tag)
   {
      this.jahr = jahr;
      this.monat = monat;
      this.tag = tag;
   }

   public ~Datum()
```

```
    {
        // was immer auch aufgeräumt werden muss
    }
}
```

Listing 7.18: Deklaration eines Destruktors für die Klasse

An dieser Stelle unterscheidet sich C# ganz erheblich von anderen Sprachen wie C++ oder Delphi und ähnelt sehr Java. Nachdem Entwickler nie Zeit haben, hier die Kurzform für die Java-Fraktion: Destruktoren in C# arbeiten wie `finalize()` in Java und sind ebenfalls nicht-deterministisch.



C++ / Delphi

Für den Rest der Welt muss ich etwas ausholen. Die Speicherverwaltung der .NET-Laufzeitumgebung besitzt eine automatische Freigabe von nicht mehr benutztem Speicher, die so genannte garbage collection. Im Gegensatz zur COM-Welt arbeitet diese Speicherverwaltung nicht über Referenzzähler und kann damit auch Objekte freigeben, welche nur noch gegenseitig aufeinander verweisen, aber sonst nicht mehr benötigt werden. Den Zeitpunkt, zu dem eine solche 'Müllsammlung' durchgeführt wird, legt aber die .NET-Laufzeitumgebung selbst fest. Erst dann wird auch der Destruktor eines Objekts aufgerufen, da erst dann die Zerstörung des Objekts erfolgt. Sie können also nicht sicher sein, wann der Destruktor der von Ihnen programmierten Klasse aufgerufen wird bzw. ob er während der Laufzeit der Anwendung überhaupt aufgerufen wird. Dieses Verhalten wird als nicht-deterministische Freigabe (*non-deterministic finalization*) bezeichnet. Sollten Sie sich in Ihrem Code also darauf verlassen, dass der Destruktor des Objekts sicher aufgerufen wird, müssen Sie diesen Code bei einer Portierung umbauen.

Wenn Sie einen Blick in die Dokumentation zur .NET-Laufzeitumgebung werfen, werden Sie feststellen, dass Microsoft für viele Klassen eine eigene Methode zum Freigeben belegter Ressourcen oder dem Schließen von Dateien und Netzwerkverbindungen angelegt hat. Die Namen dieser Methoden lauten meist `Close()` oder `Dispose()`. Diesen Ansatz können Sie auch in eigenen Anwendungen verwenden.

C# erleichtert durch dieses Konzept der Destruktoren die Programmierarbeit wesentlich, da hier auch evtl. Finalisierungen einer Basis-

klasse aufgerufen werden. In C++ für die .NET-Laufzeitumgebung müssten Sie beispielsweise die `Finalize()`-Methode des Typs `object` überschreiben und dann auch `Finalize()` der Basisklasse aufrufen. Der C#-Compiler nimmt Ihnen diese Arbeit ab. Der Code

```
public ~Datum()
{
   // aufräumen
}
```

wird implizit vom Compiler umgebaut in die folgende Konstruktion:

```
protected override Finalize()
{
   try
   {
      // aufräumen
   }
   finally
   {
      base.Finalize();
   }
}
```

Weitere Details zum Thema Speicherverwaltung und einige Methoden, mit denen das Verhalten der systemeigenen Müllabfuhr etwas detaillierter gesteuert werden kann, finden Sie in Kapitel 21 ***Garbage Collection***.

7.6 Eigenschaften

Sie können für den Zugriff auf eine Klasse natürlich die Datenfelder über public nach außen hin sichtbar machen und dann direkt damit arbeiten (dies ist in der Tat etwas, das viele Entwickler völlig normal finden).

```
class Konto
{
   // jeder greift direkt auf feld "fsaldo" zu
   public decimal fsaldo;
   // ... weiterer code der klasse ...
}
```

Listing 7.19: Die schlechte Methode zuerst – direkter Zugriff auf Felder

Diese Art des Datenzugriffs ist allerdings suboptimal, freundlich ausgedrückt. Sie verletzen damit gleich zwei Prinzipien der objektorientierten Programmierung, nämlich das der Kapselung und der minimalen Kopplung. Durch den direkten Zugriff wird zuviel Information über den inneren Aufbau des Objekts nach draußen gegeben, um später Änderungen und Weiterentwicklungen noch ungefährdet durchführen zu können und zum anderen wird durch die Benutzung des öffentlichen Feldes einer Klasse mit einem bestimmten Datentyp die Kopplung zwischen den beiden Objekten erhöht.

Daher hat sich in vielen Sprachen das 'Eigenschaften'-Entwurfsmuster für das Design eines solchen Zugriffs entwickelt, welches für den lesenden und schreibenden Zugriff auf solche Felder extra eine Methode entwickelt, über die dann der Zugriff erfolgt.

```
class Konto
{
   // saldo ist jetzt private
   private decimal fsaldo;

   // methode zum lesen
   public decimal getSaldo() {
      return fsaldo;
   }

   // methode zum schreiben
   public void setSaldo(decimal betrag) {
      // überprüfung des zugriffs
      if ( <validierung> )
         fsaldo = betrag;
   }
}
```

Listing 7.20: Zugriff auf das Feld über Warpper-Methoden

Diese Idee ist im Prinzip gut, die Ausführung aber, wie bei vielen Dingen im wirklichen Leben nicht besonders berauschend. Neben dem Aufwand, diese Methoden zu codieren, bieten Sprachen wie C++ keine Sicherheit, dass auch nur über diese Methoden und nicht direkt auf das Datenfeld zugegriffen wird. Bei Änderungen müssen Sie beide Methoden synchron halten, falls sich der Datentyp ändert, ist dieser beim Parameter der Set-Methode anzupassen und und und …

7.6.1 Deklaration von Eigenschaften

Fanfaren, Trommelwirbel, Auftritt Eigenschaften (*properties*). Dieses Konzept ist einer der wenigen Fälle, in denen tatsächlich Visual Basic der technologische Vorreiter war (wobei VB-typisch gleich wieder ein paar Inkonsistenzen eingeführt wurden, die C# an dieser Stelle behebt). Für den Anwender einer Klasse sieht der Zugriff auf Eigenschaften so aus, als würde direkt das entsprechende Datenfeld manipuliert. Die Möglichkeiten gehen allerdings noch viel weiter. Nirgendwo ist festgelegt, dass dieses Datenfeld auch physikalisch existieren muss. Sie können den Wert der Eigenschaft beim Lesen beispielsweise berechnen und beim Schreiben in mehrere verschiedene Datenfelder verteilen.

Für den Anfang ergänze ich die Klasse um den entsprechenden Code zur Erzeugung der Eigenschaft.

```
class Konto
{
   // saldo ist jetzt private
   private decimal fsaldo;

   public decimal Saldo
   {
      get
      {
         return fsaldo;
      }

      set
      {
         if ( <validierung> )
            fsaldo = value;
      }
   }
}
```

Listing 7.21: Die Klasse besitzt nun eine Eigenschaft saldo

Wieso fsaldo?



Zur Unterscheidung im Code zwischen dem Feld einer Klasse und der definierten Eigenschaft haben Sie mehrere Möglichkeiten. Eine davon ist die Nutzung der Eigenschaft von C#, Groß- und Kleinschreibung

zu unterscheiden. Das interne Feld heißt dann z.B. `saldo`, die Eigenschaft `Saldo`. Persönlich finde ich das gar nicht empfehlenswert, auch wenn diese Notation in vielen Stellen der Literatur zu C# auftaucht. Sehr schnell haben Sie nämlich einmal **s**aldo und **S**aldo verwechselt. Falls Sie beispielsweise beim Lesen der Eigenschaft anstelle von `return saldo;` ein `return Saldo;` stehen haben, bekommen Sie beim ersten Lesen der Eigenschaft einen Stacküberlauf!

Daher benutzen viele Entwickler ein entsprechendes Präfix für die internen Felder (so wie hier das `fsaldo` für *field saldo* steht). Taucht dann ein Tippfehler auf, kann der Compiler den falsch geschriebenen Bezeichner anmeckern. Damit ist dies eine einfache Möglichkeit, bereits allein durch die richtige Wahl der Bezeichner schwer zu entdeckende Bugs zu verhindern.

7.6.2 OOP-Vorteile von C#-Eigenschaften

Im Gegensatz zum ‚Eigenschaften'-Entwurfsmuster (beispielsweise in C++) bieten die Eigenschaften von C# auch beim konsistenten Entwurf von OOP-Anwendungen viele Vorteile. Sie können Eigenschaften überschreiben, indem Sie die Eigenschaft in Ihrer Klasse als `virtual` deklarieren (falls Ihnen `virtual` nichts sagt, bitte ich um etwas Geduld bis zum Abschnitt ***Vererbung***). Für abstrakte Klassen, welche nur die Eigenschaften und Methoden einer Klasse spezifizieren, aber nichts über die Implementierung aussagen, sind abstract-Eigenschaften möglich.

Ein anderer Vorteil ist die Verringerung der Kopplung zwischen einzelnen Klassen. Wenn Sie den Zugriff auf Felder Ihrer Klasse nur noch über Eigenschaften zulassen, sind Sie bei einer Veränderung der internen Ablage der Eigenschaftswerte völlig frei. Sie haben damit eine definierte Schnittstelle zur Kommunikation mit Ihrer Klasse, nur ohne den Aufwand, die entsprechenden Methoden zum Lesen und Schreiben selbst programmieren zu müssen.

Darüber hinaus kann die Leistung Ihrer Anwendung bei geschickter Ausnutzung von Eigenschaften deutlich gesteigert werden. Als kleines Beispiel möchte ich Ihnen eine Klasse vorstellen, die in einem geografischen Informationssystem eingesetzt werden soll. Mit Hilfe dieser Klasse können Sie aus einer gegebenen Position (ausgedrückt in Koor-

dinaten aus geografischer Länge und Breite) die nächste Stadt berechnen lassen. Da hierfür (nicht nur in unserem Beispiel) eine zeitaufwändige Suche über die Koordinaten in einer Datenbank nötig ist, sollte der Wert auf jeden Fall zwischengespeichert werden, solange sich die Koordinaten nicht ändern. Mit Hilfe von Eigenschaften und einem `bool`-Flag ist dies überhaupt kein Problem, wie der folgende Listingausschnitt zeigt.

Alle Leser mit geografischer Bildung mögen die grobe Vereinfachung auf Gleitkommazahlen für Breite und Länge verzeihen, aber dies ist kein Buch für Kartographen, sondern für Softwareentwickler …

```
// Einsatz von Eigenschaften und
// Zwischenspeicherung von Werten

public struct Koordinaten
{
   public double Laenge;
   public double Breite;
}

public class Test
{
   protected string fStadt;
   protected Koordinaten fKoord;
   private bool bNeu = FALSE;

   public Test(Koordinaten k)
   {
      fKoord = k;
      bNeu = TRUE;
   }

   public koordinaten Position
   {
      get
      {
         return fKoord;
      }

      set
      {
         fKoord = value;
```

```
            bNeu = TRUE;
         }
      }

      public string NaechsteStadt
      {
         get
         {
            if (bNeu)
               // zeitintensiver aufruf
               fStadt = BerechneStadt();
            else
               // ausgabe des wertes aus dem cache
               return fStadt;
         }

      }

}
```

Listing 7.22: Performanceverbesserung durch Zwischenspeichern

7.6.3 Der spezielle Bezeichner value

Wie Sie aus der Definition des Schreibzugriffs auf die Eigenschaft sehen können, existiert ein spezieller Bezeichner `value`, welcher den Wert enthält, der der Eigenschaft zugewiesen werden soll. Der Datentyp dieses Bezeichners entspricht dem der Eigenschaft.

7.6.4 Nur-Lese-Eigenschaften

Falls Sie eine Eigenschaft erstellen möchten, deren Inhalt nur gelesen werden kann, lassen Sie einfach den entsprechenden `set`-Teil aus der Deklaration aus.

```
class Konto
{
   // saldo ist jetzt private
   private decimal fsaldo;

   public decimal Saldo
   {
      get
      {
```

```
            return fsaldo;
        }

        set
        {
            if ( <validierung> )
                fsaldo = value;
        }
    }

    // eigenschaft nur lesbar
    public bool hatGuthaben {
        get {
            return (fsaldo > 0);
        }
    }
}
```

Listing 7.23: Erweiterung der Klasse um eine Nur-Lese-Eigenschaft

7.7 Statische Klassenmitglieder

Jede Instanz einer Klasse besitzt ebenfalls ihre eigene Instanz der Mitglieder dieser Klasse. Dazu ein Beispiel: Im letzten Abschnitt habe ich Ihnen die Klasse Konto vorgestellt. Wird mit `new` aus dieser Klasse eine Instanz erzeugt, besitzt jede Instanz eine eigene Kopie des Feldes `fsaldo`. Schließlich hätten Sie etwas dagegen, dass eine Buchung auf ein Konto alle anderen Instanzen Ihrer Geschäftskonten auf den gleichen Wert setzt, es sei denn, Sie haben gerade ein völlig revolutionäres Buchhaltungskonzept erfunden …

Bei bestimmten Gelegenheiten macht es aber keinen Sinn, wenn Daten oder Code für jede Instanz einer Klasse gesondert angelegt werden. Bei unserer Konto-Klasse könnte die Notwendigkeit einer Euro-Umrechnung auftreten, daher sollte ich wohl besser irgendwo in der Klasse die offiziellen Umrechnungskurse für die einzelnen Währungen in der Klasse unterbringen. Die Werte sind aber nun wirklich für jede Instanz der Klasse gleich und müssen daher eigentlich nur einmal gespeichert werden. Diese Daten gehören logisch gesehen also eher zur gesamten Klasse als zu einer bestimmten Instanz.

7.7.1 Das Schlüsselwort static

Solche Mitglieder einer Klasse werden als statische Mitglieder (*static members*) bezeichnet. Einige andere Programmiersprachen bezeichnen solche Konstruktionen auch als Klassenfunktionen (*class functions*), was sprachlich beinahe besser passt. Ein weiteres Beispiel ist der Startpunkt jeder .NET-Anwendung in C#:

```
public static void Main() {}
```

Auch dieser Code muss nur einmal im Speicher vorhanden sein, unabhängig davon, wie viele Instanzen der Klasse ich erzeuge. Wie Sie erkennen, wird bei der Deklaration eines `static` einfach dieses Schlüsselwort nach der Sichtbarkeit und vor dem Datentyp angegeben.

Konstanten sind statisch

Es gibt eine Gruppe von statischen Mitgliedern einer Klasse, welche auch ohne `static` nur einmal für alle Instanzen einer Klasse zusammen gespeichert werden: Konstanten. Da sich der Wert einer Konstante nie ändert, reicht eine Kopie für alle Instanzen der Klasse aus. Das Schlüsselwort `const` impliziert immer auch `static`.

7.7.2 Zugriff auf statische Klassenmitglieder

Wenn ein `static` zur Klasse und nicht zu einer Instanz (einem Objekt) gehört, dann kann auf dieses Feld oder diese Methode auch nur über die Klasse zugegriffen werden.



C++

Im Gegensatz zu C++ verlangt C# aus Gründen der Typsicherheit eine klare Unterscheidung zwischen Instanzmitgliedern und statischen Mitgliedern einer Klasse. Sie müssen also beim Aufruf einer statischen Methode auch wirklich den Namen der Klasse angeben und können dies nicht über ein Objekt tun, wie in C++ möglich. Damit wird bereits beim Lesen des Codes klar, welche Methode eine Instanzmethode ist und welche als statische Methode zur Klasse gehört.

Als Beispiel möchte ich einen 'Instanzzähler' für eine Klasse vorstellen. Die Klasse führt in einem `static`-Feld einen Zähler für die Anzahl der Instanzen dieser Klasse, welche über eine Nur-Lesen-Eigenschaft veröffentlicht wird.

```
// Instanzenzaehlung über statische Felder
using System;
namespace CSharpBuch
{
   class Konto
   {
      // das Feld für den Kontostand
      private decimal fSaldo;

      // eine readonly-Variable für den zeitpunkt
      // der erzeugung des objekts
      public readonly DateTime erzeugt;

      // anzahl der bestehenden instanzen
      protected static int fInstanceCount;

      // eine statische nur-lese-eigenschaft
      public static int InstanceCount
      {
         get
            {
               return fInstanceCount;
            }
      }

      // der konstruktor
      public Konto()
      {
         saldo = 0;
         erzeugt = DateTime.Now;
         fInstanceCount++;
      }

      // der destruktor
      public ~Konto()
      {
         if (fInstanceCount > 0)  // sicher ist sicher
           fInstanceCount--;
      }

      // die eigenschaft "Saldo"
      public decimal Saldo
      {
         get
         {
```

```
            return fSaldo;
         }

         set
         {
            fSaldo = value;
         }
      }

 }


// eine Klasse zum testen
class TestClass
{

   // auch "Main()" ist ein static member
   static void Main(string[] args)
   {
      Konto[] konto = new Konto[10];
      int i;
      for(i = 0; i <= konto.GetUpperBound(0); i++)
      {
         konto[i] = new Konto();
         Console.Write(konto[i].erzeugt.ToString());
         Console.WriteLine(" : Count = {0}",
                           Konto.InstanceCount);
      }

      Console.ReadLine();
   }
}

}
```

Listing 7.24: Statische Felder, readonly-Variablen, Eigenschaften

7.7.3 Statische Konstruktoren

Sie können für eine Klasse auch einen statischen Konstruktor definieren. Dieser kann beispielsweise zur Berechnung von statischen Feldern dienen, welche als `readonly` deklariert sind, weil der Wert zur Zeit der Kompilierung noch nicht feststand.

```
class Machwas
{

  public Machwas()
   {
      // der normale konstruktor
   }

  // keine modifikatoren zulässig
  static Machwas()
   {
      // der statische konstruktor
   }

}
```

Listing 7.25: Eine Klasse mit einem statischen Konstruktor

Nachdem statische Konstruktoren zur Klasse gehören, sind bei ihnen Modifikatoren für die Sichtbarkeit nicht zulässig. Die Benutzung von statischen Konstruktoren bietet sich für einmalige Initialisierungen an, die für die gesamte Klasse und nicht für einzelne Instanzen dieser Klasse durchgeführt werden sollen.

Die .NET-Laufzeitumgebung garantiert, dass ein statischer Konstruktor nach dem Start der Anwendung und vor der Erstellung der Instanz dieser Klasse ausgeführt wird, aber nicht mehr. Der Aufruf kann zeitlich also von der tatsächlichen Erstellung einer Instanz dieser Klasse deutlich getrennt sein.

7.8 Übungen



1. Welcher der beiden Modifikatoren ist restriktiver: protected oder private?
2. Warum sind Klassenmitglieder ohne Modifikator per default private?
3. Welche Einschränkung hat C# bei Methoden mit variabler Parameterzahl?
4. Wozu dient der Bezeichner `value` und wo wird dieser definiert?
5. Erstellen Sie eine Klasse mit überladenen Konstruktoren zur Speicherung einer Strecke in einem zweidimensionalen Koordinatensystem.

8 Anweisungen & Operatoren

Ein Programm in C# besteht aus Klassen, innerhalb derer Methoden definiert werden, die wiederum aus Anweisungen bestehen. Diese Anweisungen schließlich bestehen aus Schlüsselwörtern und Ausdrücken. Ein Ausdruck enthält Bezeichner, Literale und Operatoren. Damit ist die Blitztour durch die Grammatik von C# auch schon beendet. Datentypen und Variablen sowie Konstanten habe ich Ihnen bereits vorgestellt. In diesem Kapitel möchte ich Ihnen die Anweisungen und Operatoren von C# erläutern. C++-, Delphi- und Java-Programmierer werden hier viel Vertrautes entdecken. VB-Programmier kann ich an dieser Stelle bereits beruhigen: Sie werden sehen, dass Sie sich in C# Code wesentlich prägnanter und klarer ausdrücken können, ohne dass die Lesbarkeit leidet oder Sie viel umdenken müssen. Es gibt Entwickler, die C# als 'das bessere VB' bezeichnen …

8.1 Anweisungen

Wie in allen C-ähnlichen Sprachen, sind auch in C# Anweisungen frei formatierbar, d.h. Sie können Anweisungen beliebig einrücken und ohne Fortsetzungszeichen auf mehrere Zeilen aufteilen (*wie oft habe ich in VB diesen Unterstrich vergessen!*). Anweisungen enden alle mit einem Semikolon.

Überall, wo in C# eine Anweisung stehen kann, kann auch ein Anweisungsblock stehen. Ein Anweisungsblock ist einfach eine Reihe von Anweisungen, welche in geschweifte Klammern eingeschlossen wird.

```
{
   x = 67;
   y = x + 2;
   Machwas(x,y);
}
```

Listing 8.1: Ein Anweisungsblock in C#

Warum dieses Konzept benötigt wird, soll ein kleines Beispiel illustrieren, auf das ich später bei den Möglichkeiten zur Verzweigung des Programmablaufs noch einmal zurückkommen werde.

Die allgemeine Form für eine Verzweigung lautet wie folgt:

```
if  <bedingung> <anweisung> [ else <anweisung>];
```

Die beiden eckigen Klammern bedeuten, dass dieser Tel der Anweisung optional ist und die Ausdrücke in spitzen Klammern sind symbolische Platzhalter. In C#-Code sieht das folgendermaßen aus:

```
if (alter >= 18)
   System.Console.WriteLine("Wahlberechtigt!");
```

Wollen Sie nun bei einer Erfüllung der Bedingung mehr als eine Anweisung ausführen lassen, setzen Sie die Anweisung in geschweifte Klammern.

```
if (alter >= 18)
   {
   fuehrerschein_moeglich = TRUE;
   System.Console.WriteLine("Wahlberechtigt!");
   }
```

Programmierer von Sprachen aus der Pascal/Modula-Familie kennen das Konzept als `begin/end` Block, für VB-Programmierer ergibt sich dadurch eine Vereinfachung durch den Wegfall vieler verschiedener Schlüsselwörter zum Abschluss von Strukturen wie `end if`, `loop`, `end select` und so weiter.

Jede Anweisung in C# kann bei Bedarf eine Marke (*label*) zugewiesen bekommen. Ein Beispiel dafür finden Sie bei der Besprechung der `switch`-Anweisung.

8.1.1 Schlüsselwörter

C# besitzt die in Tabelle 8.1 aufgeführten Schlüsselwörter.

abstract	do	in	protected	true
as	double	int	public	try
base	else	interface	readonly	typeof
bool	enum	internal	ref	uint
break	event	is	return	ulong
byte	explicit	lock	sbyte	unchecked
case	extern	long	sealed	unsafe
catch	false	namespace	short	ushort
char	finally	new	sizeof	using
checked	fixed	null	stackalloc	virtual
class	float	object	static	void
const	for	operator	string	volatile
continue	foreach	out	struct	while
decimal	goto	override	switch	
default	if	params	this	
delegate	implicit	private	throw	

Tabelle 8.1: Die Schlüsselwörter von C#

Diese reservierten Wörter sind nicht als Bezeichner verwendbar (mit Ausnahme der im Kapitel über Variablen und Datentypen genannten Methode).

8.2 Ablaufsteuerung

8.2.1 Aus der Gruft – GOTO

Nachdem in C# jede Anweisung eine Marke zugeordnet bekommen kann (für die Namen dieser Marken gelten die gleichen Bedingungen wie für Bezeichner, nur dass der erste Buchstabe ein Doppelpunkt sein muss), können diese in C# mit der `goto`-Anweisung auch direkt angesprungen werden.

```
goto hierher;
// viieeel code dazwischen
hierher:
   if (x == 4)
      Machwas();
```

Listing 8.2: Ein Label als Sprungziel

Ja, Sie haben richtig gelesen: C# hat ein `goto`. Aus Gründen, die allein Microsoft wissen mag, hat das C#-Team um Anders Hejlsberg à la Jurassic Park einen Dinosaurier aus der Frühzeit der Programmierung auferstehen lassen. Und um ehrlich zu sein, befürchte ich, dass sich `goto` in C# im Lauf der Zeit auch so verhalten wird wie der alte T.Rex im Kino – er wird seine Erfinder in dunklen Nächten jagen. Damit lassen sich auch in C# Quelltexte portieren, die Anwärter auf einen vorderen Platz beim *obfuscated C contest* wären (*www.ioccc.org*).

Um das leidige Kapitel abzuschließen und meinem alten C-Lehrer die Ehre zu geben: Es mag auch in einer modernen, objektorientierten Programmiersprache wie C# Szenarios geben, die den Einsatz von `goto` unter Umständen gerechtfertigt erscheinen lassen: bei der Codierung von Automaten (*state machines*), beispielsweise in `switch`-Blocks oder Statuswechseln in Turing-Maschinen. Aber auch für diesen Zweck existieren Alternativen, die ich bevorzuge.

Der C#-Compiler versucht sein Bestes, den Missbrauch von `goto` zu verhindern. So wird zum Beispiel der Versuch abgewiesen, mitten in einen Block zu springen. Allerdings kann aus technischen Gründen nicht jede mögliche Konstruktion abgefangen werden, die eventuell Probleme verursacht.

8.2.2 Verzweigungen – IF / ELSE

Verzweigungen sind auch in C# einfach nur Verzweigungen ohne besondere Geheimnisse.

```
// testen einer variablen
if (redezeit == 0)
   Console.WriteLine("Bedanken und aufhören!");
else if (redezeit < 0)
       {
         Console.WriteLine("Überzogen!");
```

```
        mikro_aus = TRUE;
      }
   else
      Console.WriteLine("Weiterreden!");
```

Listing 8.3: if/else in Aktion

Delphi



Hier lauert einer der wenigen Stolpersteine für Delphi-Entwickler beim Umstieg auf C#. Die Anweisung vor dem `else` wird entweder als Block mit geschweiften Klammern codiert oder aber als einzelne Anweisung mit einem Strichpunkt abgeschlossen. Denken Sie also daran: auch vor einem `else` steht in C# ein Strichpunkt.

Es gibt allerdings einen Unterschied zu C bzw. C++: die Bedingung für die Verzweigung muss einen bool-Wert ergeben. Damit fallen für viele C-Programme typische Ausdrücke wie das folgende Beispiel in Zukunft weg:

```
int i = 45;
if (i)
   {
      // code
   }
```

Listing 8.4: Ein echter Klassiker in C und C++

Sie erhalten im Austausch dafür aber auch einen echten Vorteil im Bereich der Fehlervermeidung. Da der Compiler strikt einen bool-Typ verlangt, sind Bugs durch eine Zuweisung aufgrund vergessenem zweiten Gleichheitszeichen nicht mehr möglich, Sie erhalten dann eine Fehlermeldung des Compilers.

```
if (i = 35)   // kein bool-Typ: Fehler!
   {
      // code
   }
```

Listing 8.5: Auch ein Klassiker, aber unter den Bugs in C / C++

8.2.3 Bedingte Schleifen – DO & WHILE

Wenn Sie in C# eine Anweisung oder einen Block mehrmals wiederholt ausführen lassen möchten, haben Sie zwei Möglichkeiten, abhängig davon, ob Sie bereits wissen, wie oft diese Wiederholung geschehen soll. Falls ja, verwenden Sie die `for`-Schleife, falls nicht, dann benutzen Sie entweder `do` oder `while`.

`while` arbeitet so, wie Sie das wahrscheinlich auch von anderen Sprachen her kennen: ein Ausdruck wird zu einem bool-Wert evaluiert und falls dieser `true` ergibt, wird die Anweisung (eine einzelne Anweisung oder ein Block) ausgeführt. Anschließend wiederholt sich das Ganze, bis die Auswertung des Ausdrucks den Wert `false` ergibt.

```
string s;
while ( (s = Console.ReadLine()) != null )
{
   // ausgeben, bis alle Zeilen gelesen wurden
   Console.WriteLine(s);
}
```

Listing 8.6: While-Schleifen testen die Schleifenbedingung zuerst

Lesern aus der C- und Java-Ecke wird dieser Code wahrscheinlich nur ein Gähnen entlocken, für VB-Umsteiger möchte ich an dieser Stelle doch noch einmal den Ausdruck hinter `while` erläutern, da dieses Konzept in C-ähnlichen Sprachen üblich ist.

Nach `while` steht ein Ausdruck, der einen `bool`-Wert ergeben muss. Die Zuweisung von `s = Console.ReadLine()` liefert entweder den String aus der Eingabe oder `null`, falls keine Zeilen mehr zur Verfügung stehen (da das Ende der Datei erreicht ist). Dieses Ergebnis wird mit dem Literal `null` verglichen und liefert damit einen `bool`-Wert. Somit haben Sie eine Zuweisung und einen Vergleich in einem Ausdruck erreicht. Diese Art der Codierung werden Sie in C#-Code noch häufig sehen.

`do`-Schleifen funktionieren ähnlich, arbeiten allerdings als so genannte *annehmende* Schleifen, d.h. zuerst wird der Schleifenkörper einmal ausgeführt, dann wird die Bedingung getestet. Diese Art der Wiederholung eignet sich vor allem dann, wenn Sie eine Aktion ausführen möchten, welche selbst erst die Bedingung testbar macht. Ein Beispiel

dafür sind kommandogestuerte Programme (wie ein ftp-Client), bei denen Sie das Kommando erst dann kennen, wenn es vom Benutzer eingegeben wurde. Sie benötigen also mindestens eine Eingabe, an der Sie die Bedingung testen können.

```
string s;
do {
   s = Console.ReadLine();
   // ausgeben, bis QUIT gelesen wurde
   Console.WriteLine(s);
} while ( s != "QUIT" );
```

Listing 8.7: do-Schleifen führen erst den Schleifenkörper aus

8.2.4 Festgelegte Schleifen – FOR

Schleifen, die mit dem Schlüsselwort `for` eingeleitet werden, erlauben die Ausführung von Anweisungen für eine bekannte Anzahl von Wiederholungen.

```
for (int i = 1; i <= 10; i++ ) {
   Console.WriteLine("{0} Quadrat ist {1}",i,i*i);
   }
```

Listing 8.8: For-Schleifen für eine festgelegte Anzahl von Durchläufen

Wie Sie hier erkennen, ist es völlig legal, die Laufvariable für die Schleife erst im Ausdruck für `for` zu definieren. Sie können jeden der drei Teile innerhalb des `for`-Ausdrucks auch weglassen, wenn Sie sich selbst um die entsprechende Aktion kümmern:

```
int i = 0;
for ( ; i <= 10; i++ ) {
   Console.WriteLine("{0} Quadrat ist {1}",i,i*i);
   }
```

Listing 8.9: For mit manueller Initialisierung

Lassen Sie alle drei Teilangaben weg, dann haben Sie eine Endlosschleife!

```
for ( ; ; ) {
   // dieser Code läuft dauernd
   }
```

Listing 8.10: Endlosschleifen für schnelle CPUs :-)

8.2.5 Schleifen in Auflistungen – FOREACH

Hier werden sich zur Abwechslung einmal die Visual Basic-Programmierer zu Hause fühlen. `foreach` entspricht von der Funktionalität her im Großen und Ganzen der Anweisung `for each` in VB. Diese Schleife läuft nacheinander durch alle Elemente eines Arrays oder einer Auflistung und spart erstens die Definition einer Laufvariablen, zweitens müssen Sie vorher nicht wissen, wie viele Elemente die Auflistung oder das Array hat.

```
int myArray[] = new int[] {2,3,5,7,11,13,17,19};
foreach ( int x in myArray )
{
   Console.WriteLine("Element besitzt den Wert {0}", x);
}
```

Listing 8.11: Beispiel für eine foreach-Schleife

Dieses Konstrukt funktioniert sowohl mit Arrays als auch mit Auflistungen, da Arrays in der .NET-Umgebung über die Auflistungsklassen in `System.Array` definiert werden. Dies bedeutet natürlich auch, dass Sie diesen Mechanismus für eigene Auflistungen einsetzen können, wenn Sie diese entsprechend entwerfen. Mehr Informationen dazu erhalten Sie im Abschnitt ***Auflistungen selbst gebaut***.

8.2.6 Raus hier – BREAK & CONTINUE

In allen Schleifenkonstrukten sind Sie in der Lage, den Schleifenkörper vorzeitig zu verlassen, entweder für einen erneuten Durchgang oder zum kompletten Verlassen der Schleifenanweisung.

Das Schlüsselwort `continue` bricht die Ausführung des Schleifenblocks ab (es muss hier ein Block sein, da bei einer einzelnen Anweisung ein `continue` keinen Platz hätte und auch keinen Sinn macht) und beginnt einen neuen Durchlauf.

Das Schlüsselwort `break` bricht die Ausführung des Schleifenblocks ebenfalls ab, verlässt aber anschließend die Schleife und übergibt die Programmausführung an die nächste Anweisung nach der Schleife.

8.2.7 Mehrfachauswahl – SWITCH

Einfache Verzweigungen habe ich Ihnen bereits beim Absatz zur `if`-Anweisung vorgestellt. Für eine Abfolge von mehreren Alternativen wäre das aber nur ziemlich umständlich mit `if`-Abfragen kodierbar. Daher existiert in C# ebenfalls die in vielen Programmiersprachen vorhandene `switch`-Anweisung einer Mehrfachauswahl (je nach Programmiersprache `case`, `select` oder `switch` genannt).

```
switch ( <testausdruck> )  {
   case <ausdruck>:
      <anweisung | block>
      break;
[ default:
      <anweisung | block>
      break; ]
}
```

Listing 8.12: Die allgemeine Syntax für switch

Hier existieren (teilweise aus Gründen der Codesicherheit) einige Neuerungen in C# im Vergleich zu anderen Sprachen. Der Datentyp des Ausdrucks hinter dem Schlüsselwort `switch` muss entweder einen der Integer-Typen, `char` oder `string` ergeben. Letzteres dürfte vor allem VB-Programmierer freuen, die diese Möglichkeit schon immer nutzen konnten (die andere Sprachen wie C oder Delphi schmerzlich vermissen ließen).

Ein weiterer Unterschied ist die Tatsache, dass **kein** automatisches 'Durchfallen' durch die einzelnen case-Blöcke wie in C/C++ möglich ist. Falls Sie das `break` weglassen, erhalten Sie einen Compilerfehler! Sie können allerdings mit einem `goto` zwischen den einzelnen `case`-Blöcken hin- und herspringen, auch rückwärts (*ich hatte Ihnen doch versprochen, dass es einen sinnvollen Einsatz für goto gibt*).

Die einzige Möglichkeit, mehrere Vergleiche durch 'Durchfallen' zu erledigen, ist die Definition von leeren `case`-Blöcken, d.h. nur `case`-Anweisungen ohne Code direkt hintereinander, was aber wieder der intuitiven Lesbarkeit des Quelltextes entspricht.

```
// s ist vom Typ string
switch ( s )  {
   case "Karl":
   case "Franz":
   case "Hans":
      Console.WriteLine("männlicher Name");
      break;
   case "Gabi":
   case "Andrea":
      Console.WriteLine("weiblicher Name");
      break;
   default:
      Console.WriteLine("Geschlecht für Name unbek.");
      break;
}
```

Listing 8.13: Ein Beispiel für einen switch-Block

8.3 Synchronisation, Überlaufprüfung

8.3.1 lock

Die Anweisung `lock` dient der Synchronisierung von Zugriffen in Anwendungen mit mehreren Threads. Damit können Sie Codeblöcke bilden, auf die immer nur ein Thread zugleich zugreifen darf. Die .NET-Laufzeitumgebung kümmert sich dann um die Serialisierung der einzelnen Threads.

8.3.2 checked, unchecked

C# bietet für nummerische Berechnnugen die Möglichkeit, Überlaufprüfungen zuzulassen oder abzuschalten. Dies errreichen Sie, indem Sie die entsprechenden Passagen im Quellcode mit einem `checked` oder `unchecked` versehen. Damit werden dann bei einem mit `checked` überprüften Codeblock auch Überläufe von Integerwerten gemeldet.

8.4 Operatoren

Dieser Abschnitt lässt sich schnell abhandeln, wenn Sie bisher mit C++/Java programmiert haben. War Ihre Sprache bisher VB, Pascal, COBOL oder eine andere Programmiersprache, sollten Sie sich für diesen Abschnitt etwas Zeit nehmen.

Operatoren lassen sich in verschiedene Gruppen einteilen: Anzahl der Operanden oder Art der durchgeführten Operation. Ich habe mich für eine Einteilung in funktionale Gruppen entschieden, nach denen ich als Umsteiger von einer anderen Programmiersprache suchen würde.

- Arithmetische und Bit-Shift Operatoren: `+ - * / ++ -- % << >>`
- Logische Operatoren: `& | && || ^ ~ !`
- Relationale Operatoren: `== != <= >= < >`
- Zuweisungen: `= += -= *= /= %= &= |= ^= <<= >>=`
- Typ-Operatoren: `new (TYP) is sizeof typeof as`
- Bedingungsoperator: `?:`
- Zugriffsoperatoren: `. []`

8.4.1 Arithmetische Operatoren

Die Grundrechenarten

Die Operatoren der vier Grundrechenarten handle ich in einem Satz ab, da es wohl kaum eine Programmiersprache gibt, bei der diese Operatoren anders funktionieren. Sowohl +, –, * und / besitzen jeweils zwei Operanden und wenden die entsprechende Rechenart darauf an. Der Multiplikationsoperator und der Divisionsoperator besitzen eine höhere Priorität als die Addition und die Subtraktion. Die Rechenregel *Punkt vor Strich* funktioniert also. Falls Sie das bei Microsoft für eine Selbstverständlichkeit halten, starten Sie den Windows-Taschenrechner in der Standard-Ansicht und geben Sie 4 + 5 * 2 ein!

Modulo (%)

Der Modulo-Operator berechnet wie bei C++ und Java den Rest einer ganzzahligen Division, bei vielen anderen Programmiersprachen heißt dieser Operator `mod`. Das klassische Beispiel für diesen Operator ist die Berechnung von Schaltjahren, hier also die Variante in C# (die verwendeten relationalen Operatoren erläutere ich weiter unten).

```
bool Schaltjahr(uint jahr) {
   return (jahr % 400 == 0) ||
          ((jahr % 100 != 0) && (jahr % 4 == 0));
   }
```

Listing 8.14: Schaltjahresberechnung in C#

Im Gegensatz zu Visual Basic besitzt C# keinen Operator für eine ganzzahlige Division. Diesen müssen Sie über einen entsprechenden Ausdruck nachbilden, indem Sie den ganzzahligen Anteil der Division benutzen.

++ und --

Die beiden Operatoren ++ und -- lassen sich sowohl als Präfix- als auch als Postfix-Operatoren einsetzen. Dies bedeutet, dass Sie diese Operatoren entweder vor oder nach dem Bezeichner setzen können. Verfügbar ist dieser Operator für alle einfachen nummerischen Werttypen und alle Aufzählungstypen.

```
int i = 4;
if (++i == 5) Console.WriteLine("i besitzt den Wert 5")
Console.WriteLine(i = {0}", i);
```

Listing 8.15: ++ als Präfix-Operator

Ausgabe auf dem Bildschirm:

```
i besitzt den Wert 5
i = 5
```

Wird der Operator nach dem Bezeichner verwendet, wird der Wert auch erst nach der Operation erhöht, in welcher der Ausdruck verwendet wird:

```
int i = 4;
if (i++ == 5) Console.WriteLine("i besitzt den Wert 5")
Console.WriteLine(i = {0}", i);
```

Listing 8.16: ++ als Postfix-Operator

Ausgabe auf dem Bildschirm:

```
i = 5
```

8.4.2 Logische Operatoren

Logisches und bitweises UND

Der Operator `&` führt eine bitweise UND-Verknüpfung der beiden Operanden durch, während der logische Operator `&&` ein logisches

UND für beide Ausdrücke berechnet. Ich möchte zuerst ein Beispiel für den Einsatz des bitweisen Operators `&` zeigen:

```
const cFett            = 0x01;
const cUnterstrichen   = 0x02;
const cKursiv          = 0x04;
const cDurchgestrichen = 0x08;
const cSchattiert      = 0x10;

uint flags;

// code

bool IstFett = (flags & cFett) == cFett;
```

Hier wird überprüft, ob der in `flags` gespeicherte Wert das Bit für Fettdruck gesetzt hat, in dem die Variable mit dem Bitwert logisch UND-Verknüft wird. Ist das Bit gesetzt, sind beide Bits 1 und damit ist das Ergebnis auch wieder der Bitwert für Fettdruck.

Ein Beispiel für den Einsatz des logischen Operators `&&` sahen Sie bereits bei der Berechnung des Schaltjahres. Für alle VB-Programmierer an dieser Stelle noch ein Hinweis, der leicht zu dem einen oder anderen schwer aufzuspürenden Bug führen kann: Der Operator `&&` arbeitet mit einer Kurzschlussauswertung (*short circuit evaluation*). Der entsprechende Ausdruck wird dazu von links nach rechts ausgewertet und sobald ein Teilausdruck den Wert `false` liefert, wird die Auswertung abgebrochen, da damit der Gesamtausdruck bei einer UND-Verknüpfung nicht mehr `true` ergeben kann.

```
if (<ausdruck1> && <ausdruck2> )
   <anweisung>;
```

Listing 8.17: Kurzschlussauswertung bei && in C#

Ergibt `<ausdruck1>` `false`, wird `<ausdruck2>` nicht mehr ausgewertet. Steht nun in diesem Ausdruck eine Funktion, welche auch noch andere Dinge erledigt, unterbleibt die Ausführung dieser Funktion!

Logisches und bitweises ODER

Diese Operatoren arbeiten analog zu den UND-Operatoren, führen aber eine ODER-Verknüpfung durch. Was ich für den Operator `&&`

zum Thema Kurzschlussauswertung erwähnt habe, gilt hier ebenso. Nur, wenn der erste Ausdruck ein `false` liefert, wird auch der zweite Teilausdruck evaluiert.

```
if (antwort == 42) || BoolFunc(antwort)
```

Hier wird `BoolFunc` nur dann aufgerufen, wenn `antwort` nicht den Wert `42` besitzt, da ansonsten die ODER-Bedingung bereits erfüllt wäre.

Ein bitweises Oder wird meist zum Setzen von Flags in Bitmasken benutzt. Um auf das Beispiel zum bitweisen Und noch einmal zurückzukommen: Mit einem bitweisen Oder können Sie einzelne Bits setzen:

```
flags = flags | cFett; // setzt flag fuer fettdruck
```

Logisches NICHT

Der Operator `!` liefert die Negation des nachfolgenden Ausdrucks und entspricht damit dem `not` anderer Programmiersprachen.

Exklusives ODER

Der Operator `^` berechnet eine bitweise exklusive ODER-Verknüpfung der beiden Operanden. Es darf also entweder das eine Bit **oder** das andere gesetzt sein, aber nicht beide zugleich. Wenn an dieser Stelle ein unmoralisches Beispiel gestattet ist: Sie können als Softwareentwicklerin zu einer Party entweder mit dem Ehemann oder mit dem Freund erscheinen, ohne Probleme zu bekommen.

Und wo liegt der Nutzen?

Solche Referenzen sind ja ganz nett zum Nachschlagen, aber als Trainer werde ich in Softwareentwicklerkursen meist an dieser Stelle gefragt, wo der praktische Nutzen für ein bitweises Exklusiv-Oder liegt. Nun, damit lassen sich mit etwas Tüfteln sehr speicherplatzsparende Dinge erledigen. Testen Sie einmal den folgenden Code:

```
class Geheimnis
{
public static void Main() {
   byte a = 56;
   byte b = 120;
   a = b ^ a;
```

```
    b = b ^ a;
    a = b ^ a;
    Console.WriteLine("a = {0}, b = {1}", a, b);
    }
}
```

Erraten Sie die Lösung? Genau, damit können Sie den Inhalt zweier Variablen tauschen, ohne eine Hilfsvariable zu benötigen! Wer immer Ihnen also erzählt, zum Tauschen von Werten benötigt man einen Zwischenspeicher, hat bei den logischen Operatoren nicht ganz aufgepasst …

Bitweises Komplement

Um alle Bits in einem ganzzahligen Werttyp umzudrehen, verwenden Sie den Operator ~, welcher das sog. Bitkomplement berechnet.

```
byte b = 170;
Console.WriteLine("170 bitweise gedreht ist {0}", ~b);
```

Listing 8.18: Können Sie die Ausgabe im Kopf berechnen?

8.4.3 Relationale Operatoren

Die Vergleichsoperatoren sind wieder in fast allen Sprachen gleich und werden daher nur kurz in einer Tabelle vorgestellt.

Operator	Bedeutung
==	ist gleich
!=	ist ungleich
<=	kleiner oder gleich
>=	größer oder gleich
<	kleiner
>	größer

Tabelle 8.2: Die relationalen Operatoren von C#

Viele Bugs in C++- und C-Programmen haben ihren Ursprung in der Verwechslung des Gleichheitsoperators == und des Zuweisungsoperators =. Bei C# verhindert dies zum Teil der Compiler durch eine

wesentlich striktere Typprüfung. So akzeptiert die `if`-Anweisung nur einen `bool`-Typ, daher wird `if (x = 5)` gar nicht erst angenommen, sondern nur das korrekte `(if x ==5)`.

8.4.4 Zuweisungen

Zuweisungen werden in C# für alle Datentypen durch den Operator = vorgenommen. Nun sind Entwickler prinzipiell nicht besonders von der Tipparbeit beim Erstellen von Programmen begeistert. Daher existierten bereits in C Operatoren, welche Zuweisung für einfache arithmetische Operationen abkürzten.

Der Operator += entspricht beispielsweise einer Additionszuweisung. Anstelle des Ausdrucks `x = x + 5` können Sie also `x += 5` schreiben. Dies ist für die Operationen der folgenden Tabelle möglich.

Operator	entspricht folgender Operation
+=	Addition
-=	Subtraktion
*=	Multiplikation
/=	Division
%=	Modulo
&=	Bitweises UND
|=	Bitweises ODER
^=	Bitweises Exklusiv-ODER
<<=	Bitverschiebung nach links
>>=	Bitverschiebung nach rechts

Tabelle 8.3: Erweiterte Zuweisungsoperatoren in C#

8.4.5 Typ-Operatoren

Typumwandlungen (casts)

Mit diesem Operator (der in runden Klammern gesetzten Angabe eines Typs) sind Sie in der Lage, den Wert eines Ausdrucks in den gewünschten Datentyp umzuwandeln (falls zulässig). Ist die Durchführung dieser Typumwandlung nicht zulässig, wird eine Exception ausgelöst.

```
byte LF = 10;
string s;
s = "Zeile 1" + char(LF) + "Zeile 2";
```

Hier wird der Bytewert LF in einen Zeichenwert umgewandelt und kann so bei der Verkettung der Strings benutzt werden.

Größenberechnung – sizeof

Der Operator `sizeof` berechnet die physikalische Größe des als Parameter übergebenen Typs. Ein Ausdruck `int i = sizeof(uint)` liefert in `i` den Inhalt `4`, da ein uiNT 4 Byte groß ist.

Objektinstanziierung – new

Sollten Sie noch nichts mit objektorientierter Programmierung, Typ-Informationen und Instanzen von Klassen programmiert haben, empfehle ich Ihnen, zuerst einen Blick in die Kapitel 6 ***OOP-Grundlagen*** und Kapitel 7 ***Klassen in C#*** zu werfen, bevor Sie sich die Beispiele der folgenden vier Operatoren ansehen.

Mit Hilfe des Operators `new` wird eine neue Instanz einer Klasse (ein Objekt) im Speicher erzeugt. Als Parameter bekommt der Operator den Namen der Klasse oder des Interfaces übergeben.

Typauswertung – typeof

`typeof` liefert genau das, was Sie aufgrund des Namens auch erwarten würden, den Typ des Operanden. Meist wird dies in Kombination mit dem `is`-Operator benutzt.

```
if (typeof(obj) is string)
   {
      // hier code für stringbehandlung
   }
```

Typvergleich – is

In vielen Fällen ist die erfolgreiche Codeausführung davon abhängig, ob ein Objekt einen bestimmten Typ besitzt, d.h. eine Instanz dieser oder jener Klasse ist, ein bestimmtes Interface unterstützt oder nicht. Diesen Vergleich kann der Operator `is` erledigen (siehe das obige Beispiel bei `typeof`).

Eine detailliertere Erklärung dieser Operatoren liefere ich Ihnen mit einer Einführung in die 'Selbstbetrachtung' von Typen (*reflection*) bei C# in Kapitel 22 ***Reflection in C#*** und in Kapitel 23 ***Attribute***.

Typumwandlung für Objekte – as

Der Operator `as` arbeitet ähnlich wie `is`, allerdings wird kein Vergleich durchgeführt, sondern eine Typkonvertierung in die gewünschte Klasse bzw. das gewünschte Interface. Ist diese erfolgreich, liefert `as` als Ergebnis eine Instanz des angegebenen Typs, andernfalls ist das Ergebnis dieser Umwandlung `null`.

Hier ein kleines Beispiel für die Verwendung der Typoperatoren, welches die Konzepte verdeutlicht.

```
// test für new, is, as
interface IMachwas {
   void SagHallo();
}

interface IWasAnderes {
   void Schweige();
}

class Macher : IMachwas {
   private string name;
   public void Macher(string besucher) {
      if (name != "")
         name = besucher;
      else
         name = "Fremder";
   }

   public void SagHallo() {
      Console.WriteLine("Hallo, {0}!", name);
   }
}

class Schweiger : IWasAnderes {
   public void Schweige() {
      Console.WriteLine("Stille ...");
   }
}
// testen der klassen
```

```
class Test {
   public static void Main() {
      object[] arr = {new Macher("Franz"),
                      new Schweiger(),
                      new Schweiger(),
                      new Macher("Gabi")
                      };

      foreach (Object obj in arr ) {
         if (obj is IMachwas) {
            IMachwas im = (IMachwas) obj;
            im.SagHallo();
         }
      }
}
```

Listing 8.19: Typabfrage und cast mit is

Anstelle der Anweisungen in der `foreach`-Schleife ließe sich nun durch die Verwendung des Operators `as` auch der folgende Code einsetzen, welcher funktional gleichwertig (aber meiner Meinung nach leichter lesbar) ist.

```
// as anstelle von is und typecast
IMachwas im = (obj as IMachwas);
if (im != null)
   im.SagHallo();
```

Listing 8.20: Typabfrage mit cast und as

8.4.6 Der Bedingungsoperator

Das ist auch bei C# mein Liebling unter den Operatoren, schließlich hat mich sein Gegenstück in C zu Anfang meines C-Selbststudiums einige Haare gekostet, bis ich das Prinzip begriffen hatte. Ich hoffe, Sie haben es auf Grund meiner Erklärung leichter. Alle C/C++-Programmierer können diesen Abschnitt getrost überspringen.

Dieser Operator `?:` besitzt als einziger Operator in C# drei Operanden und wählt auf Grund eines `bool`-Ausdrucks als ersten Operanden einen der beiden folgenden Operanden als Ergebnis. Damit sparen Sie sich den Einsatz einer `if`-Anweisung.

```
int steuersatz = (ermaessigt == true) ? 7 : 16;
```

Ist der Ausdruck `ermaessigt == TRUE` wahr, dann erhält die Variable `steuersatz` den Inhalt `7`, andernfalls den Wert `16`. Ohne diesen Operator würde der Code so aussehen, da er nicht in einer Anweisung zu erledigen wäre:

```
int steuersatz;
if (ermaessigt == true)
   steuersatz = 7;
else
   steuersatz = 16;
```

8.4.7 Zugriffsoperatoren

Der Operator . dient dem Zugriff auf Methoden und Eigenschaften einer Klasse, zur Trennung von Namensraum und Klasse und so weiter. Allgemein definiert ist der Operator . als Trennzeichen innerhalb hierarchischer Datenstrukturen in C# und entspricht damit bei Klassen und Methoden dem C++-Operator `->`.

Der `[]`-Operator ist kein Operator im eigentlichen Sinne, sondern definiert in C# ein Array (ein- oder mehrdimensional). Für den Zugriff auf ein Element über seinen Index dient dann die eckige Klammer.

8.5 Übungen

Ü

1. Warum liefert der Code `if(x = 3) { Console.Write("ungerade"); }` eine Fehlermeldung bei der Übersetzung?
2. Sie wollen bei der Erfüllung einer bestimmten Bedingung in einer `while`-Schleife die Schleifenabarbeitung abbrechen und die Schleife komplett verlassen. Benutzen Sie dazu `break` oder `continue`?
3. Wie lässt sich vor einer Typumwandlung feststellen, ob eine Objektvariable auch von einem bestimmten Datentyp ist?
4. Sie wollen in einer `foreach`-Schleife feststellen, wie oft diese bereits durchlaufen wurde. Wie würden Sie den Code formulieren?
5. Was ist das Ergebnis von `8 ^ 3` in C#?



Going professional – Teil II

9 Abzählen in C# – Enums

9.1 Die Grundlagen

C# ist eine typsichere Programmiersprache mit einer für die C/C++-Familie relativ strikten Typprüfung zur Vermeidung von Fehlern im Programmcode. Daher fällt auch die Implementierung von Aufzählungstypen etwas 'bürokratischer' aus, als es manchem gestandenen C-Veteranen lieb sein mag. Alle anderen Leser werden mir nach der Lektüre dieses Abschnitts sicher zustimmen, dass die Vorteile der C#-Implementierung bei weitem überwiegen.

Benötigt werden Aufzählungen immer dann, wenn eine Methode oder eine Variable in einem Programm nur ein definiertes Set von Werten verarbeiten soll. Ein Steuerelement zur Bildverarbeitung soll vielleicht nur Rot, Grün oder Blau als Farbwert akzeptieren, in einem Kalenderprogramm soll als Kalender nur der gregorianische, der julianische oder der Kalender der französischen Revolution verwendet werden. Oder, damit wären wir bei meinem Beispiel für diesen Abschnitt, der Wert eines Feldes beim alten Schülerspiel *Schiffe versenken* kann einen von vier möglichen Werten annehmen: Wasser, Schiff, bereits getroffenes Wasser und ein bereits getroffenes Schiffsfeld.

9.2 Enums implementieren

In C#-Code sieht die erste Implementierung dann so aus:

```
using System;
public class Versenken
{
   public enum Feldwert
   {
      Wasser,
      Schiff,
      Wassertreffer,
      Schiffstreffer
   }
}
```

Listing 9.1: Eine einfache Aufzählung

Damit kann der Vergleich beim Spiel und die Deklaration des Spielfeldes nun mit den Werten dieser Aufzählung erfolgen.

```
// spielfeld deklarieren
Feldwert[,] spielfeld = new Feldwert[9,9];
// ...
// code zum testen
   switch spieldfeld[x, y] {
      case Wasser:
         // code fuer 'daneben'
         break;
      case Schiff:
         // code fuer einen Treffer
         break;
      case Wassertreffer:
         // hier waren wir schon mal
         break;
      case Schiffstreffer:
         // hier haben wir schon getroffen
         break;
   }
}
```

Listing 9.2: Verwenden von Aufzählungstypen

Im Gegensatz zu beispielsweise C lässt sich der Compiler auch nicht beschummeln. Wird ein Datentyp als Aufzählung deklariert, werden auch nur Werte aus diesem Aufzählungstyp akzeptiert, nicht die entsprechenden Ordinalzahlen. Der einzige Weg an dieser Prüfung vorbei führt über eine explizite Typumwandlung, damit die Absicht im Code auch ausgedrückt wird.

Warum das so ist, zeigt eine kleine Erweiterung des Codes. Die zugewiesenen Werte lassen sich nämlich bei der Definition des Aufzählungstyps vergeben und müssen nicht in aufeinander folgender Reihe liegen. Als Standard beginnt C# bei der Vergabe von Wertentsprechungen mit der 0 und erhöht den Wert jeweils um eins. Für den Ausdruck gelten die gleichen Regeln wie für Konstantenausdrücke, die im Kapitel 8 ***Anweisungen und Operatoren*** erläutert wurden. Insbesondere darf ein Mitglied der Aufzählung nie seinen eigenen Wert referenzieren (zirkuläre Definition).

```
public enum Feldwert
{
   Wasser = 0,
   Schiff = 2,
   Wassertreffer = 4,
   Schiffstreffer = 8
}
```

Listing 9.3: Aufzählung mit definierten Ordinalwerten

Aufzählungstypen und 0



Nachdem die .NET-Laufzeitumgebung alle Variablen initialisiert, in dem diese auf 0 gesetzt werden, ist der Standardwert für eine Variable jedes Aufzählungstyps ebenfalls der Ordinalwert 0. Daher sollte jeder Aufzählungstyp ein Mitglied mit diesem Wert definieren. Ist dies nicht der Fall, kann dies zu Problemen führen.

0 ist auch der einzige nummerische Literalwert, der einem Aufzählungstyp direkt ohne eine Typumwandlung zugewiesen werden kann. Dies ist ein Entgegenkommen des C#-Teams, da die Zuweisung von 0 an einen Aufzählungstyp meist den Standardwert meint.

Es ist völlig legal, mehr als einem Mitglied eines Aufzählungstyps den gleichen Wert zuzuweisen. Beide Mitglieder besitzen dann den gleichen Aufzählungswert. Ebenfalls kann die explizite Zuweisung eines Wertes auch bei einem anderen als dem ersten Mitglied der Aufzählung gestartet werden. Für alle nachfolgenden Mitglieder, für die keine explizite Angabe des Wertes mehr erfolgt, wird wieder der Wert um eins erhöht. Bevor die Erläuterung noch komplexer wird, hier ein Beispiel, an dem Sie beide Konzepte erkennen können.

```
enum MyErrors {
   ERR_Base = 0,
   ERR_UnknownUser = ERR_BASE,
   ERR_InvalidPassword,
   ERR_PasswordExpired,
   ERR_OutOfMemory = ERR_BASE +100,
   ERR_InvalidOpcode,
   ERR_InvalidHandle
}
```

Listing 9.4: enum für Fortgeschrittene

Die Wertefolge für diesen Aufzählungstyp lautet also 0, 1, 2, 3, 100, 101, 102, 103.

9.3 Basisdatentypen für Aufzählungen

In Zeiten, in denen in Discount-Supermärkten Rechner mit 40 GByte-Festplatten verramscht werden, mag die folgende Anmerkung etwas anachronistisch wirken, aber auch für die Mitglieder eines Aufzählungstyps kann der zugrunde liegende Basisdatentyp bestimmt werden, welcher den Speicherbedarf eines Mitglieds festlegt. Der Standard ist auch hier `int`. Gültige Werte sind `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` und `ulong`. Anders ausgedrückt, alle einfachen nummerischen Ganzzahltypen. Im Normalfall können Sie dies getrost ignorieren, es sei denn, Sie definieren einen Aufzählungstyp, bei dem die Menge der Mitglieder den verfügbaren Wertebereich eines `int` überschreitet (ich würde Ihnen dann gerne das Papier für Ihre Listings liefern!).

Angegeben wird der Basisdatentyp nach dem Namen des Aufzählungstyps, getrennt durch einen Doppelpunkt (ich komme auf diese Schreibweise noch beim Thema Vererbung zurück).

```
public enum Feldwert : byte
{
   Wasser = 0,
   Schiff = 1,
   Wassertreffer = 2,
   Schiffstreffer = 4
}
```

Listing 9.5: Aufzählung mit Angabe des Basisdatentyps

Ein Mitglied dieser Aufzählung kann über eine Typumwandlung in seinen zugrunde liegenden Basisdatentyp konvertiert werden und umgekehrt.

```
int x = (int) Schiff;
```

9.3.1 Aufzählungen als Bitflags

Sie können die Mitglieder der Aufzählung natürlich über einen entsprechenden Basistyp und die passende Wahl der Attributwerte auch zur Definition von bitweisen Kennzeichen benutzen, die miteinander über ein bitweises Oder bzw. ein bitweises Und verknüpft werden können.

```
public enum FileAttr : byte
{
   faDefault  = 0x00,
   faReadOnly = 0x01,
   faHidden   = 0x02,
   faSysFile  = 0x04,
   faVolumeID = 0x08,
   faFolder   = 0x10,
   faArchive  = 0x20
}
```

Listing 9.6: Eine Bitmaske als Aufzählungstyp

Ein versteckter Ordner lässt sich nun für eine Verwendung dieser Aufzählungsbits mit der Kombination `FileAttr.faFolder | FileAttr .faHidden` beschreiben. Die logischen bitweisen Operatoren `|`, `&`, `^` und ~ sind für alle selbstdefinierten Aufzählungsdatentypen gültig und arbeiten dementsprechend.

In einigen Büchern und Webseiten zum Thema C# finden Sie manchmal auch den Hinweis auf das Attribut `[Flags]`, welches der Aufzählung vorangestellt wird. Dieses Attribut (ich behandle Attribute und das Thema Metadaten in Kapitel 23 ***Attribute***) ist ein Hinweis an das .NET-Laufzeitsystem (und natürlich an Entwickler). Erstens darauf, dass die Werte dieser Aufzählung normalerweise per `|` (bitweises ODER) verbunden werden und zweitens, dass viele Codebrowser und Entwicklungsumgebungen Bitflag-Aufzählungen dann anders darstellen. Wie Sie im Listing 9.5 erkennen, ist die Verwendung dieses Attributs aber nicht zwingend.

Der Unterschied zur Laufzeit liegt beispielsweise in der Ausgabe der Daten über die Methode `ToString()`. Nehmen wir an, Sie haben folgende Codezeilen:

```
FileAttr attr = faReadOnly | faSysFile | faHidden;
Console.WriteLine("Wert = {0}", attr.ToString() );
```

Ohne die Angabe des Attributes `[Flags]` erhalten Sie folgende Ausgabe:

```
Wert = 7
```

Mit Angabe von `[Flags]` bei der Definition erhalten Sie diese Ausgabe:

```
Wert = faReadOnly,faSysFile,faHidden
```

9.4 Ausgabe von Aufzählungstypen

So schön das Konzept der Aufzählungstypen bei allen anderen Sprachen, die ich kenne, auch ist: erst bei C# habe ich eine vernünftige Möglichkeit gefunden, diese Namen eines `enum` auch wieder dem Benutzer auszugeben. Um allen Einsteigern in die Programmierung die Problemstellung zu verdeutlichen, hier ein kleines Szenario: Sie haben eine neue Anwendung erstellt, mit deren Hilfe der Anwender nach Dateien mit bestimmen Attributen suchen kann (vergessen wir `DIR /a` für einen Moment). In einer Auswahlliste möchten Sie nun dem Anwender die möglichen Werte präsentieren. Sehr oft wird dann auf eine Hilfsliste mit den Bezeichnungen als String-Array ausgewichen, in welchen die Einträge sozusagen doppelt geführt werden. Nach Murphys Gesetz wird mit tödlicher Sicherheit bei der nächsten Änderung der Aufzählungsnamen im Code die Beschreibung nicht angepasst. Das Ergebnis wird vollends unbrauchbar, sobald sich die Werte der Mitglieder der Aufzählung ändern und diese nicht in der entsprechenden Textausgabe synchron mit geändert werden. Darüber hinaus kann ja auch ein neues Mitglied (im Beispiel z.B. `faCompressed = 0x40`) hinzukommen. Auch hier muss der Code wieder manuell angepasst werden. Ich denke, Sie wissen, worauf ich hinaus will.

Praktischerweise definiert die `enum`-Klasse eine Methode `GetValues`, welche das (nach Werten sortierte) Auslesen alle Mitgliederwerte des Aufzählungstyps in ein Array erlaubt. Ich verwende die Aufzählung aus Listing 9.5:

```
Array werte = Enum.GetValues( typeof(FileAttr) );
// werte[0] enthält nun 0, werte[6] enthält 32
```

Wie Sie erkennen, wird als Parameter ein `Type`-Objekt übergeben. Für diesen Datentyp werden die Mitglieder aus der Aufzählung ausgelesen. Wenn Sie keinen gültigen Typ (ein `enum`) übergeben, wird eine exception ausgelöst.

Die Namen der Aufzählung können Sie genauso einfach ermitteln. Dafür verwenden Sie die Methode `GetNames` der enum-Klasse. Diese liefert ein Array aus Strings, in welchem Sie die Namen der Aufzählungsmitglieder finden. Praktischerweise sind diese ebenfalls nach dem Wert der Mitglieder sortiert, so dass die beiden Methoden synchron arbeiten.

```
string[] namen = Enum.GetNames( typeof(FileAttr) );
// name[0] enthält nun z.B. "faDefault"
```

Was noch fehlt, ist eine Methode, um für ein einzelnes gegebenes Mitglied der Aufzählung den Namen zu ermitteln. Damit lassen sich dann auch Konstrukte wie das nachfolgende endlich ad acta legen, die wohl schon jeder Entwickler öfter (mehr oder weniger begeistert) geschrieben hat.

```
// attr ist vom Datentyp FileAttr
switch attr {
   case faDefault:
      Console.WriteLine("faDefault");
      break;
   case faReadOnly:
      Console.WriteLine("faReadOnly");
      break;
   case faHidden:
      Console.WriteLine("faHidden");
      break;
   case faSysFile:
      Console.WriteLine("faSysFile");
      break;
   case faVolumeID:
      Console.WriteLine("faVolumeID");
      break;
   case faFolder:
      Console.WriteLine("faVolumeID");
      break;
   case faArchive:
      Console.WriteLine("faArchive");
      break;
}
```

Listing 9.7: Die »alte« Art der Ausgabe von enum-Werten vor C#

Jetzt reicht dafür eine einzige Zeile:

```
Console.WriteLine(Enum.GetName(typeof(FileAttr),attr));
```

Listing 9.8: So einfach funktioniert es mit C# und Enum.GetName

Ich denke, damit wird auch einem weniger erfahrenen Leser sofort klar, wie viel Potential zur Zeitersparnis und vor allem zur Fehlervermeidung in diesen neuen Möglichkeiten steckt. Stellen Sie sich vor, Sie müssten mit den Farben, für die in der .NET-Laufzeitumgebung Namen definiert sind (welche mir persönlich nebenbei bemerkt wenig helfen, da die Namen teilweise merkwürdig gewählt sind und die Farben nicht aus der Palette der 216 browser-sicheren Farben stammen), eine Listbox füllen. Ohne Methode, um aus dem `enum` den Namen eines Mitglieds zu erhalten, würde das eine völlig sinnlose Tipparbeit.

Enums und Java

Das Konzept von Aufzählungstypen (`enums`) wird von Java nicht unterstützt. Meine persönliche Meinung ist, dass es sich hierbei um ein durchaus nützliches Konstrukt bei Programmiersprachen handelt, aber ich habe Java nicht erfunden und die einzige Sprache, für die ich jemals einen Compiler geschrieben habe, besaß den hohen Marktanteil von immerhin zwei Installationen (1985 auf einem Uni-Rechner unter MS-DOS 2.1!). Also werde ich die Entscheidung des Java-Teams nicht werten, möchte aber für alle Java-Entwickler einige Bemerkungen zu diesem Thema einfügen, welche das Thema verdeutlichen.

Einem `enum` am nächsten kommt in Java noch das Entwurfsmuster der so genannten *static finals*. Obwohl es einige Java-Profis gibt, die `enums` ablehnen, da Probleme hinsichtlich Erweiterbarkeit, Serialisierung und Wertebereichen mit Sprüngen (wie in C# problemlos möglich) nicht geklärt sind, hat der Ersatz durch eine entsprechend gestaltete Klasse in Java auch Vorteile. An dieser Stelle eine von Brad Appleton programmierte Klasse, die dem `enum`-Konzept so nahe als (mit vernünftigem Aufwand) möglich kommt.

```
class Color {
   protected static Vector
      colorVector = new Vector();
   private int    colorNumber;
   private String colorName;
```

```
    protected Color(String name) {
       this.colorName = name;
       this.colorNumber = colorVector.Size();
       colorVector.addElement(this);
    }
    public String name() {
       return this.colorName;
    }
    public int ord() {
       return this.colorNumber;
    }
    public static final Color RED =
           new Color("RED");
    public static final Color YELLOW =
           new Color("YELLOW");
    public static final Color BLUE =
           new Color("BLUE");
    // usw ...
}
```

9.5 Zusammenfassung

`Enum` bietet eine typsichere Möglichkeit, einen eigenen Datentyp mit einer definierten Anzahl von möglichen Werten zu erstellen.

Über eine Typumwandlung können Sie bei Bedarf auf den jedem Mitglied eines Aufzählungstyps zugewiesenen nummerischen Ordinalwert zugreifen. Mit Hilfe der Methoden der Klasse `Enum` können Sie sehr leicht auf Namen und Werte der Aufzählung zugreifen.

Aufzählungstypen eignen sich auch gut als mögliche Werte einer Bitmaske, mit Hilfe des Attributes [Flags] können Sie eine entsprechende Verwendung der Aufzählung markieren.

Bei Bedarf lässt sich für eine Aufzählung ein Basisdatentyp angeben, der dann für die Speicherung der einzelnen Werte einer Aufzählung benutzt wird.

9.6 Übungen

Ü

1. Können Sie bei der Erstellung einer Aufzählung Sprünge in den zugewiesenen nummerischen Werten haben?
2. Wie lässt sich die Bezeichnung eines Aufzählungsmitglieds ermitteln?
3. Sie möchten gerne 8 verschiedene Werte in einer Aufzählung speichern und wollen nur den minimal erforderlichen Platz für die Aufzählung verbrauchen? Wie würden Sie diese Aufzählung erstellen?
4. Können Sie der Variablen eines Aufzählungstyps anstelle eines Mitglieds der Aufzählung auch einfach den nummerischen Wert zuweisen?

10 Structs und Arrays

Dieses Kapitel stellt eine Weiterführung des Kapitels über die einfachen Datentypen dar. Sie erfahren hier, wie in C# zusammengesetzte Datentypen, Aufzählungen und Arrays benutzt werden.

10.1 Arrays

Arrays als Felder von Werten, welche über einen Index angesprochen werden, gab es schon in den ersten Programmiersprachen, lange bevor ein Entwickler die erste Zeile für das .NET-Framework geschrieben hat. Aus diesem Grund sollten Sie sich diesen Abschnitt des Kapitels in Ruhe durchlesen, da Sie (nicht nur) als C- oder C++-Entwickler eine Reihe neuer Konzepte kennen lernen, durch die sich Arrays bei C# von denen anderer Sprachen vor der .NET-Ära unterscheiden.

Wichtigster Unterscheid zu beispielsweise C++ ist die Tatsache, dass Arrays als Referenztypen durch das Speichermanagement von .NET verwaltet werden und nach Gebrauch durch die Garbage Collection 'entsorgt' werden. Sie müssen sich als Entwickler also einzig um das Erzeugen des Arrays kümmern. Sobald keine Variable mehr auf das Array zugreift, wird es aus dem Speicher entfernt.

10.1.1 Deklaration

Arrays werden in C# als Referenztypen behandelt und auf dem Heap erzeugt. Abgesehen von anderen Vorteilen eliminiert dies wieder einen möglichen Grund für die Verwendung von Pointern wie bei C oder C++. Beim Aufruf von Methoden wird eine Referenz auf das Array auf dem Stack übergeben und das war's (ein unabsichtlicher Stacküberlauf durch Arraydaten auf dem Heap kommt also bei C# nicht mehr vor). Im Array selbst werden dann die Daten direkt gespeichert, wenn es sich um Werttypen wie ordinale Datentypen oder structs handelt. Enthält das Array Referenztypen wie Klassen oder Strings, dann speichert das Array nur die Referenz auf die eigentlichen Daten.

Die allgemeine Syntax für die Deklaration eines Arrays lautet wie folgt:

```
<datentyp>[] <bezeichner> [ = <initialisierung> ];
// Beispiel
double[] tagesumsaetze;
```

Listing 10.1: Allgemeine Syntax für die Deklaration eines eindimensionalen Arrays

Linguistischer Ausflug

An diesem Beispiel lässt sich ebenfalls die Entwicklung der Syntax moderner Programmiersprachen im Vergleich zu älteren Sprachen erkennen.

Fangen wir bei einem 'uralten' Basic (einem Commodore CBM4032 aus meiner Oldtimer-Sammlung) an:

```
Dim tagesumsaetze(31)
```

Damit wird einfach nur an den Bezeichner in runden Klammern der maximale Index angehängt. Eine Definition der Variable allein ohne sofortige Erzeugung des Arrays ist hier nicht möglich. Selbstdefinierte Datentypen waren 1981 sowieso noch Science Fiction bei Basic-Dialekten. In Visual Basic wird ein Array folgendermaßen deklariert:

```
Dim tagesumsaetze As integer(31)
```

Diese Anweisung deklariert die Variable für das Array und erzeugt dieses auch gleich. Immerhin wird hier aus der Deklaration schon deutlich, dass es sich um ein Array aus `Integer`-Werten handelt. In Pascal (und Delphi) sieht die Deklaration so aus:

```
Var tagesumsaetze : array[1..31] of integer;
```

Hier lassen sich bereits Unter- und Obergrenze eines Arrays festlegen und aus der Deklaration geht klar hervor, dass es sich um ein Array handelt. In Delphi ist sogar die Deklaration einer reinen Array-Variablen ohne die Erzeugung des Arrays über eine Zeigervariable (einen Pointer) möglich:

```
Type TUmsatzArray = array[1..31] of integer;
Var tagesumsaetze : ^TUmsatzArray;
```

Da in C# alle Arrays Referenztypen sind, muss jedes Array erst instanziiert werden, die Array-Variable selbst ist also immer nur die Referenz. C# (und auch Java) deklariert ein Array wie folgt:

```
int[] tagesumsaetze;
```

Da dies nur die Variable (die Referenz) auf ein Array deklariert, ist hier auch noch keine Angabe der gültigen Indexwerte notwendig. C# ist bei der Deklaration strenger als Java, welches auch die alte C-Variante `int tagesumsaetze[];` als Deklaration zulassen würde.

Mehrdimensionale Arrays

Natürlich lassen sich in C# auch Arrays mit mehr als einer Dimension deklarieren. Dazu werden in den Klammern die Dimensionen einfach durch Kommata getrennt.

```
int[,] schachbrett; // array mit zwei dimensionen
double[,,] raumwuerfel; // array mit drei dimensionen
```

Listing 10.2: Deklaration mehrdimensionaler Arrays

Jagged Arrays (Arrays von Arrays)

An dieser Stelle zuerst ein Lob an das Team, welches bei Microsoft für die Förderung und Promotion von C# zuständig ist (für den schönen Titel *Technology Evangelist* gibt es leider keine Übersetzung). In fast jedem Vortrag höre ich als einer der neuen Features von C#, dass Arrays mit ungleichmäßig großen Dimensionen unterstützt werden, die so genannten jagged arrays. Klingt gut, nur ist das als einfaches Array von Arrays mit C auch schon seit Mitte der 70er möglich (klingt aber nicht so innovativ).

Nachdem die zweite bzw. alle nachfolgenden Dimensionen also unterschiedlich groß sein können, reicht die Deklaration aus Listing 9.1 nicht mehr aus. Ein jagged array wird daher so deklariert:

```
double[][] jahresumsatz;
```

Dies deklariert eine Referenzvariable für ein Array, dessen Mitglieder wieder aus einem Array bestehen. Da die zweite 'Dimension' (die Arrays) also unabhängig von der ersten Dimension ist, können diese Arrays auch unterschiedlich groß werden (in unserem Beispiel jeweils 30 oder 31 und einmal 28 bzw. 29 Einträge).

Hier zum Abschluss noch eine echte Kopfnuss aus der C# Sprachreferenz. Was wird hier deklariert?

```
double[] [,,] [,] messwert;
```

Das ist ein Array ([]) aus dreidimensionalen Arrays ([,,]), welche jeweils ein zweidimensionales Array ([,]) von double-Werten speichern. Nun hätte ich nur noch gerne ca. 4 GByte an RAM, um dieses Ding auch entsprechend groß zu erzeugen …

10.1.2 Erzeugung und Initialisierung

Dies geschieht über das Schlüsselwort new und die Angabe der entsprechenden gültigen Indexwerte für das Array.

```
tagesumsatze = new double[31];
```

Dies reserviert den Speicherplatz für ein double-Array mit 32 Einträgen. Diese werden von der .NET-Laufzeitumgebung mit 0 vorbelegt, müssen aber vor einer Verwendung aufgrund der Notwendigkeit der Zuweisung eines Initialwertes erst einen Wert zugewiesen bekommen. Im Gegensatz zu manch anderen Programmiersprachen beginnt ein Array immer mit dem Index 0 und besitzt als höchsten Index also bei einem int[N] den Wert N-1. Für die Größe der einzelnen Dimensionen gelten die gleichen Regeln wie für Konstantenausdrücke in Kapitel 8, ***Anweisungen und Operatoren***.



VB / Delphi

Hinweis für VB-Programmierer: Eine Möglichkeit, wie in VB mit option base den Anfangsindex auf 0 oder 1 zu setzen, ist in der .NET-Umgebung nicht möglich. Die schlechte Nachricht für Delphi-Programmierer: Damit ist auch die Deklaration von Arrays mit einem definierten Anfangsindex wie Delphi nicht möglich. In beiden Sprachen ist bei einer Portierung von Code damit Überarbeitungsarbeit angesagt.

Aus Gründen der Performance werden auch Arrays mit Referenztypen auf 'alle Bits auf 0' (bei Referenzen also null) und nicht auf den im Array deklarierten Datentyp initialisiert. Falls Sie also ein Array aus Instanzen einer Klasse deklarieren, können Sie nach der Erzeugung dieses Arrays mit new nicht auf die einzelnen Elemente zugreifen, da diese den Wert null enthalten. Warum dies so ist? Nun, andernfalls müsste der Compiler für jedes Element des Arrays einen Aufruf für den jeweiligen Konstruktor einbauen und diesen bei der Erzeugung des Arrays aufrufen. Aus Performancegründen unterbleibt dies aber. Als Vorgriff auf das Kapitel über Klassen hier der Ausweg: Sie definieren sich in der Klasse eine static-Methode, welche als Funktionsergebnis

eine Instanz der Klasse liefert und rufen diese dann für jedes Element des Arrays auf (was möglich ist, da `static`-Methoden auch ohne eine Instanz der Klasse aufgerufen werden können, aber dazu später mehr).

Sie können die Werte für ein Array natürlich wie z.B. in C++ auch gleich bei der Deklaration angeben. Dazu wird in geschweiften Klammern eine Liste der entsprechenden Werte für jedes Array-Element angegeben. Die Betonung liegt hier auf **jedes Element**. Sie dürfen also nicht nur einen Teil angeben, sondern müssen wirklich alle Mitglieder oder gar keins angeben (wir leben in einer digitalen Welt).

```
int[] prim20 = new int[8] {1, 2, 3, 5, 7, 11, 13, 17};
```

Listing 10.3: Deklaration mit Initialisierung der Array-Werte

Für ein mehrdimensionales Array werden die Werte entsprechend geschachtelt. Um beispielsweise ein zweidimensionales Array aus Koordinaten zu füllen, gehen Sie wie folgt vor:

```
int[,] koordinaten = new double[6,2]
   {  {1.0, 2.5}, {1.8, 4.7},
      {1.4, 9.8}, {8.3, 5.9},
      {4.7, 6.2}, {3.1, 6.0} } ;
```

Listing 10.4: Initialisierung eines mehrdimensionalen Arrays

Nachdem der Compiler anhand der Liste mit den Initialwerten ja die entsprechenden Größen der einzelnen Dimensionen selbst berechnen kann, dürfen Sie das Schlüsselwort `new` und die Angabe der Dimensionsgröße auch weglassen.

```
int[] prim20 = {1, 2, 3, 5, 7, 11, 13, 17};
```

Listing 10.5: Deklaration mit Initialisierung ohne new

10.1.3 Konvertierungen bei Arrays

Auch bei der Konvertierung von Array-Inhalten, wie sie bei einer Zuweisung eines Array-Wertes an einen anderen vorkommen, muss die Typsicherheit von C# gewährleistet werden. Daher gelten hier entsprechende Regeln. Ebenso bei der Übergabe von ganzen Arrays als Parameter eines Methodenaufrufs muss eine erfolgreiche Konvertierung möglich sein.

Eine implizite Konvertierung ist dann möglich, wenn sowohl das Quell- als auch das Ziel-Array über die gleiche Anzahl an Dimensionen verfügen und eine implizite Referenzkonvertierung für den Elementtyp des Quell-Arrays zu dem Elementtyp des Ziel-Arrays besteht. Damit lässt sich beispielsweise ein Array aus `string` als Parameter einer Methode an ein Array des Typs `object` übergeben, da `object` ein Basistyp von der Klasse `string` ist und damit eine implizite Konvertierung existiert. Mehr zu diesem Thema erfahren Sie in Kapitel 16 ***Konvertierungen***.

10.1.4 Die Klasse System.Array

Die .NET-Laufzeitumgebung liefert mit der den Arrays zugrunde liegenden Klasse `System.Array` eine ganze Menge an Funktionalität, welche die Unterstützung anderer Sprachen für Arrays übertreffen. Die gut zwei Dutzend Methoden und Eigenschaften sollte Sie auf jeden Fall in der Dokumentation zum .NET-Framework einmal durchblättern.

So können Sie Arrays sehr einfach mit der Methode `Sort()` sortieren lassen, während `Reverse()` die Reihenfolge der Inhalte des Arrays umdreht. Das übliche Programmierbeispiel zum Thema Arrays, nämlich das Sortieren der Inhalte eines Arrays, kann ich Ihnen an dieser Stelle also nicht mehr verkaufen. Allerdings beschränkt sich `Sort()` auf eindimensionale Arrays, was aber in der Mehrzahl der Fälle ausreichen sollte.

Die Suche nach Elementen gestaltet sich ebenfalls dank der .NET-Laufzeitumgebung und der `System.Array`-Klasse wesentlich einfacher als in nicht .NET-fähigen Programmiersprachen: `IndexOf()` und `LastIndexOf()` liefern den Index der ersten bzw. letzten Fundstelle in einem Array und mit der Methode `BinarySearch()` wird ein schneller binärer Suchalgorithmus gleich mitgeliefert.

`System.Array` implementiert Arrays kompatibel mit der `foreach`-Anweisung. Sie können also Arrays mit einer `foreach`-Schleife durchlaufen, wenn der Datentyp der Laufvariablen vom gleichen Typ wie die Elemente des Arrays sind.

```
int[] lottozahlen = new int[5];
foreach (int x in lottozahlen)
   Console.WriteLine("Zahl = {0}", x);
```

Listing 10.6: Ausgabe aller Elemente eines Arrays mit foreach

Die Größe eines Arrays lässt sich über die Eigenschaft `Length` der Klasse feststellen, die Größe einer bestimmten Dimension über die Methoden `GetLowerBound()` und `GetUpperBound()`. Wie Sie sehen, existiert eine Methode `GetLowerBound()`, um die untere Grenze einer Arraydimension festzustellen, obwohl laut Sprachreferenz von C# alle Arrays immer mit dem Index 0 beginnen. Dies trifft zwar für C# zu. Das dem .NET-Framework zugrunde liegende Typsystem erlaubt aber durchaus auch Indizes mit einem anderen Startwert als 0. Als Investition in die Zukunft (und in ihre Haare auf dem Kopf) empfehle ich Ihnen daher, sich grundsätzlich der Methoden `GetLowerBound()` zu bedienen und nicht einfach anzunehmen, dass der Index eines Arrays immer mit 0 beginnt.

Was bei solchen Annahmen herauskommen kann, konnten wir beim Aufschrei der VB-Gemeine erleben, als `TRUE` in VB.NET auf einmal `1` statt `-1` sein sollte. Hätten alle VB-Programmierer konsequent auf `not false` oder wenigstens `true` statt dem Zahlenwert `-1` getestet, wäre das überhaupt kein Problem gewesen.

10.2 Structs

In C# haben `structs` ähnlich wie bei C++ mehr mit Klassen als mit den alten zusammengesetzten Typen oder Records von Basic oder Delphi gemeinsam. Alle Einsteiger in die Welt der objektorientierten Programmierung sollten an dieser Stelle also vielleicht einen mutigen Schritt vorwärts zum Kapitel ***OOP-Grundlagen*** und dann ***Klassen*** tun. Anschließend können Sie diesen letzten Teil der Datentypenkapitel dann nachholen.

10.2.1 Mehr als nur Daten

`Structs` in C# sind zusammengesetzte Datentypen, welche auch Code (Methoden) enthalten können, die aber als Werttypen und nicht wie Klassen, Arrays oder Strings als Referenztypen angelegt werden. Damit ist auch gleich der größte Unterschied erklärt: `structs` liegen als Werttypen auf dem Stack, sind also ideal dafür geeignet, Daten zu speichern, welche nicht zu groß sind, in großer Zahl vorkommen und für die eine eigene Klasse einfach zuviel Aufwand wäre. `structs` können nichts von einem anderen Datentyp erben, ebenso kann von einem `struct` selbst

nichts geerbt werden. Ansonsten können Sie wie bei C++ `structs` zusätzlich zu den Daten auch mit Code (Methoden) ausstatten und `structs` können Konstruktoren besitzen.

Als Faustregel: Wenn es nur um zusammengesetzte Datentypen (wie Koordinaten, Angaben von Wert und Maßeinheit oder Netzwerkpaketstrukturen) geht, dann ist ein `struct` die richtige Wahl. Ist die Datenmenge größer als so ungefähr 128 Byte oder benötigen Sie zu den Daten auch verarbeitenden Code bzw. werden später von diesen Daten spezialisiertere Formen abgeleitet, dann sollten Sie eine Klasse erstellen.



VB / Delphi

Ein `struct` entspricht bei Visual Basic in etwa dem selbstdefinierten Datentyp mit `Type`. Allerdings sind die Möglichkeiten eines `struct` wesentlich erweitert, da in VB ein `Type` nur Daten enthalten kann.

Das Gleiche gilt für Delphi: ein `struct` ist mit einem `record` vergleichbar. Hier könnte man die Methoden eines `struct` noch mit Zeigern oder Prozedurtypen nachbilden, welche als Felder im `record` gespeichert werden.



C++

Der Unterschied zwischen `struct` und `class` ist in C# wesentlich größer als in C++. Im Gegensatz zu C++, wo ein `struct` im Prinzip nichts anderes als eine Klasse ohne Zugriffsschutz durch Sichtbarkeitsmodifikatoren darstellt, sind ein `struct` und eine `class` in C# in internem Aufbau und Möglichkeiten durchaus verschieden.

10.2.2 Deklaration und Initialisierung von structs

Für einen `struct` gelten in C# die gleichen Regeln für Sichtbarkeit und Zugriffsgrenzen wie für Klassen. Um das gleiche Standardverhalten wie in C++ zu erreichen, müssen wir also alle Felder der Struktur als `public` deklarieren (mehr zum Thema Zugriff und Sichtbarkeit in Kapitel 7 ***Klassen in C#***).

Der einfachste Fall der Deklaration eines `struct` ist nur die Deklaration der darin enthaltenen einfachen Datentypen. An dieser Stelle finden Sie auch eine Art Weltpremiere in der C#-Literatur. Ich werde keine

Koordinatenstruktur mit `x`- und `y`-Koordinate als Beispiel bringen, sondern etwas völlig Neues (und zum Zeitpunkt der Entstehung dieses Buches auch wesentlich Aktuelleres): einen Währungs-struct, der Betrag und Währung speichert!

Es ist mir bis heute unverständlich, wie viel Mühe sich Softwarehersteller von Microsoft bis Oracle mit der Definition toller Datentypen für die rundungssichere Speicherung von Währungsbeträgen geben, aber dabei nie einen Blick über den kulturellen Tellerrand Amerikas werfen. Was soll ich mit einem Währungsbetrag anfangen, bei dem nicht die entsprechende Währung dabei ist. Immerhin, seit der Einführung des Euro ist zumindest die Frage DM oder EUR nicht mehr aktuell …

```
// beispiel für einen minimal-struct in C#
public struct Geld {
   public decimal betrag;
   public string waehrung;
}
```

Listing 10.7: Der erste struct in C#

Was passiert nun, wenn ich eine Variable mit diesem Datentyp deklariere?

```
Geld guthaben;
```

Hier ist einer der Punkte, in denen sich ein `struct` und eine Klasse deutlich unterscheiden. Für eine Klasse benötige ich zuerst eine Instanz dieser Klasse (ein Objekt), bevor ich auf die Eigenschaften und Methoden dieser Klasse zugreifen kann. Daher muss vor einem Zugriff auf eine Objektvariable erst mit `new` eine Instanz erzeugt werden. Da `structs` Werttypen sind, enthält die Variable guthaben im Gegensatz zu Referenztypen keine Referenz auf den `struct`, sondern die Daten selbst. Die .NET-Laufzeitumgebung initialisiert diese Felder alle mit einem Nullwert, also 0 für nummerische Datentypen und einem null für den String.

Der Zugriff auf die Struktur ist also auch ohne ein `new` möglich, bringt aber gleich die erste Überraschung: Da der Wert für den String `waehrung` auf `null` initialisiert wurde, muss hier erst eine Instanz mit `new` erzeugt werden, da Strings ja Referenztypen sind (mehr zu Strings erfahren Sie in Kapitel 15 ***Strings***).

```
Geld guthaben;
guthaben.betrag = 1500.00;
guthaben.waehrung = new String("DEM");
```

Listing 10.8: Zugriff auf die Felder eines struct

Konstruktoren bei struct

Wäre schön, wenn diese Startwerte als Initialisierung gleich bei der Deklaration erledigt werden könnte, nicht wahr? Dies ist einer der Vorteile eines C#-`Struct` im Vergleich zu z.B. selbstdefinierten Datentypen in Visual Basic. C#-`Structs` können auch Code speichern, in diesem Fall einen Konstruktor, welche beim Erzeugen der Struktur aufgerufen wird.

```
// beispiel für einen minimal-struct in C#
public struct Geld {
   public decimal betrag;
   public string waehrung;

   public Geld(decimal b, string w) {
      betrag = b;
      waehrung = w;
   }
}
```

Listing 10.9: struct in C# mit Konstruktor

Damit kann der Code aus Listing 10.3 wie folgt geschrieben werden:

```
Geld guthaben = new Geld(1500.00, new String("DEM") );
```

Listing 10.10: Deklaration eines struct mit Konstruktor

Zusammengefasst: Wird ein `struct` deklariert, ohne `new` aufzurufen, setzt die .NET-Laufzeitumgebung alle Bits in allen Feldern auf `0`. Daher muss eine Anwendung mit einem auf `0` initialisierten `struct` ebenso zurechtkommen wie mit einem durch einen Konstruktor initialisierten `struct`. Enthält die `struct` nur Werttypen, kann dieser sofort benutzt werden, alle Werttypen besitzen dann den Wert `0`.



Aus Gründen der Fehlervermeidung ist bei einem `struct` auch kein Konstruktor ohne Parameter (ein sog. *default constructor*) erlaubt. Dieser könnte die Felder bei der Deklaration automatisch auf unterschied-

liche, von 0 verschiedene Werte initialisieren, was sich unter Umständen in nur schwer auffindbaren Bugs äußert. Einen Destruktor gibt es bei einem `struct` ebenfalls nicht.

10.2.3 Verschachtelte structs

Es ist ohne weiteres möglich, `structs` zu verschachteln. Auf die Felder der inneren Struktur wird dann über den Namen der äußeren Struktur, gefolgt von einem Punkt (also der üblichen Schreibweise für hierarchische Datentypen) zugegriffen. Die Deklaration des inneren `struct` kann dabei natürlich außerhalb oder innerhalb der umgebenden `struct` stehen, wie die beiden folgenden Beispiele zeigen.

Im ersten Beispiel nehme ich an, dass beide `structs` irgendwann noch einmal in der Anwendung gebraucht werden, daher deklariere ich den `struct` Konto außerhalb des `struct` Geld.

```
// beispiel für geschachtelte structs
public struct Konto {
   public ulong kontonr;
   public ulong blz;
}

public struct Geld {
   public decimal betrag;
   public string waehrung;
   public Konto konto;

   public Geld(decimal b, string w) {
      betrag = b;
      waehrung = w;
      konto.kontonr = 0;
      konto.blz = 0;
   }
}
// zugriff auf die felder
Geld guthaben;
guthaben.konto.kontonr = 12012056;
guthaben.konto.blz = 700000000;
```

Listing 10.11: Geschachtelte Structs in C#

10.2.4 Layout von structs

Normalerweise legt die .NET-Laufzeitumgebung fest, wie und in welcher Reihenfolge die Felder einer Struktur im Speicher angeordnet werden. Falls Sie dies für einen bestimmten Zweck (wie den Aufruf von Win32-API Funktionen) ändern wollen, müssen Sie zwei Attribute benutzen. Informationen darüber, was Attribute sind und wie Sie diese benutzen oder selbst definieren, finden Sie in Kapitel 23 ***Attribute***.

Wenn Sie also einen `struct` mit einem manuellen Layout versehen wollen, geben Sie vor der Deklaration des `struct` das Attribut `StructLayout` mit dem passenden Mitglied der Aufzählung `LayoutKind` an.

- **Auto** Die .NET-Laufzeitumgebung kümmert sich um das Layout (Standard).
- **Sequential** Die Felder der Struktur werden in der Reihenfolge, in der sie bei der Deklaration angegeben wurden, mit dem jeweiligen Datentyp, nacheinander im Speicher abgelegt.
- **Explicit** Kontrolliert die genaue Lage innerhalb des Layouts der Struktur über die Angabe eines Offsets für jedes Feld. Wird dieser Wert gewählt, ist die Angabe des Attributs `FieldOffset` für jedes Feld der Struktur zwingend.

Als Beispiel stelle ich Ihnen die Deklaration einer Struktur für den Aufruf der API-Funktion zum Auslesen der Systeminformationen vor, bei der für jedes Feld der Struktur explizit das Layout angegeben ist.

```
[StructLayout(LayoutKind.Explicit)]
public struct SYSTEM_INFO {
   [FieldOffset(0)] public ulong OemId;
   [FieldOffset(4)] public ulong PageSize;
   [FieldOffset(16)] public ulong ActiveProcessorMask;
   [FieldOffset(20)] public ulong NumberOfProcessors;
   [FieldOffset(24)] public ulong ProcessorType;
}
```

Listing 10.12: Eine Struktur mit detaillierter Layout-Angabe

10.3 Zusammenfassung

Strukturen (`structs`) sind eine Möglichkeit, logisch zusammengehörige einfache Datentypen zu einem komplexen benutzerdefinierten Datentyp zusammenzufassen. Strukturen können auch ineinander verschachtelt werden.

Strukturen sind Werttypen und werden daher nicht auf dem Heap angelegt, stehen auch ohne den Aufruf eines Konstruktors zur Verfügung und sind für kleine Dateneinheiten gedacht.

Strukturen können wie Klassen auch über Methoden verfügen, besitzen aber keine Möglichkeiten der Vererbung.

Für den Aufruf bestimmter Systemfunktionen lässt sich das physikalische Speicherlayout einer Struktur mit Attributen angeben und so die Standardeinstellung der .NET-Laufzeitumgebung umgehen.

10.4 Übungen



1. Was ist ein `char[,] [,,] eingabe` für eine Struktur?
2. Worauf sollten Sie bei sehr großen Strukturen achten?
3. Erstellen Sie ein kleines Programm, mit dem Sie ein Array mit einer benutzerdefinierten Anzahl von `int`-Werten einlesen und daraus die jeweils höchste und niedrigste Zahl ermitteln. Ziehen Sie auch die Online-Hilfe für die Klasse `System.Math` zu Rate.
4. Wie können Sie einer Struktur bei der Deklaration gleich für jedes Feld einen Initialwert zuweisen?

11 Vererbung

Vererbung ist eines der großen Grundprinzipien der objektorientierten Programmierung. Durch die Möglichkeit, eine Klasse von einer bestehenden abzuleiten, erhalten Sie die Möglichkeit, auf vorhandener Arbeit aufzubauen und Relationen der Art »X ist wie ein Y, aber …« auszubauen. In diesem Kapitel werde ich Ihnen die Möglichkeiten von C# vorstellen, die Sie zum Ableiten von Klassen haben. Natürlich werde ich es auch nicht unterlassen können, wieder den einen oder anderen kleinen Exkurs zu einzelnen Themen unterzubringen.

Noch ein kleiner Hinweis zur Terminologie für OOP-Einsteiger: Vererbung kann entweder nach unten gerichtet (eine Klasse wird von einer anderen abgeleitet) oder nach oben gerichtet (eine Klasse erbt von einer Basisklasse) ausgedrückt werden. Beide Formen sind funktional gleich, da es nur zwei Ausdrucksweisen für die gleiche Beziehung zwischen Basisklasse und erbender neuer Klasse sind.

11.1 Einfache Vererbung

Um eine Klasse von einer anderen abzuleiten, müssen Sie angeben, von welcher Klasse die neue Klasse abgeleitet wird (erbt). Alle bisherigen Beispiele sahen aber doch nur so aus wie die Klasse unten, werden Sie nun einwenden.

```
public class Testklasse
{
   // mitglieder
}
```

Listing 11.1: Eine Klasse, die implizit von object abgeleitet wird

Stimmt, dennoch erbt auch diese Klasse und zwar implizit von `object`. In C# ist alles ein Objekt, Klassen ohne die Angabe einer Basisklasse werden daher automatisch von der Klasse `object` abgeleitet (der Compiler setzt das für Sie ein. Nett, nicht?). Natürlich können Sie das auch selbst angeben und damit stelle ich auch gleich den 'Vererbungs-Operator' vor:

```
public class Testklasse : object
{
   // mitglieder
}
```

Listing 11.2: Eine Klasse mit expliziter Angabe der Basisklasse



C++

In C# ist eine Ableitung nur von einer Basisklasse möglich (*single inheritance*). Sie können Schnittstellen (*interfaces*) benutzen, um von verschiedenen Basisklassen zu erben. Auch wenn viele C++-Entwickler dies nicht gerne lesen, Mehrfachvererbung ist ein Raubtier, das nur von wenigen gezähmt werden kann. Für diese Gurus ist es ein wertvolles Werkzeug, für den Rest der Welt (mich eingeschlossen) bedeutet es mehr Probleme als Nutzen. Bisher konnte ich alle Anforderungen in Projekten sehr gut (und wesentlich wartungsärmer) durch Schnittstellen erfüllen. Im Web liegt eine Veröffentlichung von Tom Carghill mit dem schönen Titel »multiple inheritance considered harmful«. Lesenswert!

Damit wäre aber in diesem Beispiel außer Schreibarbeit nichts gewonnen, denn die neue Klasse `Testklasse` erbt einfach die gesamte Funktionalität der Klasse `object` und führt selbst nichts Neues ein (machen das nicht in der realen Welt auch die meisten Erben?). Daher möchte ich für die folgenden Beispiele eine andere Basisklasse definieren.

```
public class Saeugetier
{
   protected string fName;
   private int fLaufbeine;

   public float Gewicht;

   public Saeugetier(string name, float gew, int beine)
   {
      fName = name;
      fLaufbeine = beine;
      Gewicht = gew;
   }

   public int Laufbeine
   {
      get { return fLaufbeine; }
```

```
      set { fLaufbeine = value; }
   }

   public void Laufe()
   {
      Console.WriteLine("{0} läuft auf {1} Beinen.",
                        fName, fLaufbeine);
   }

}
```

Listing 11.3: Die Basisklasse Saeugetier

Dies soll unsere Basisklasse für Säugetiere darstellen. Jedes Säugetier besitzt einen Namen, ein Gewicht und eine Anzahl Laufbeine.

Ich gebe zu, hier fehlt eine Überprüfung auf entweder 2 oder 4 Laufbeine, es sei denn, die nächste Expedition der Kryptozoologen findet ein Exemplar einer sechsbeinigen Waldantilope. Ich möchte die Beispiele allerdings nicht mit Prüfroutinen voll stopfen, welche den Blick auf die eigentlichen Konzepte verstellen (in einer Produktivanwendung würden Sie doch niemals die Fehlerprüfung weglassen, versprochen?). Eine letzte Anmerkung, bevor Sie mir Brehms Tierleben nachwerfen: Nachdem ich ursprünglich mal Ozeanologie studieren wollte, ist mir durchaus bewusst, dass es Meeressäuger ohne Laufbeine gibt. Diese werden im Kapitel über Interfaces noch sehr wertvoll für die Beispiele werden, Flipper ist also nicht vergessen. Nun aber zurück zu C# …

Um die Klasse der Katzen (systematisch sind die Katzen natürlich eine Familie) von der Klasse der Säugetiere abzuleiten, benutze ich den 'Vererbungsoperator' und gebe die Basisklasse an. In der neuen Klasse kann ich die Möglichkeiten der Basisklasse nun um die gewünschte neue Funktionalität ergänzen.

```
public class Katze : Saeugetier
{

   // katzen laufen immer auf vier beinen,
   // daher nur im base-konstruktor anzahl laufbeine
   public Katze(string name, float gew) :
      base(name, gew, 4)
      {
      }
```

```
    // neue Methode Laufe, "new" unterdrückt warnung
    // des compilers, siehe Text unten
    new public void Laufe()
    {
       Console.WriteLine("Katzen laufen auf " +
                         "{0} Samtpfoten.", Laufbeine);
    }

}
```

Listing 11.4: Die abgeleitete Klasse Katze

Die Methode `Laufe()`, welche in der Klasse `Katze` deklariert wurde, ersetzt die Methode `Laufe()` der Basisklasse. Der C#-Compiler gibt hier allerdings eine Warnung aus, da dies ja auch aus Versehen geschehen könnte. Dafür dient das Schlüsselwort `new` an dieser Stelle. Dies markiert für den Entwickler sichtbar, dass hier eine neue Version der Methode eingeführt wurde, welche die alte überschreibt.

To protect or not to protect …

Die Deklaration der internen Datenfelder als `protected` legt fest, dass nur diese Klasse und alle davon abgeleiteten Klassen Zugriff auf diese Felder besitzen. Damit besitzt eine abgeleitete Klasse aber auch Zugriff auf interne Strukturen Ihrer Basisklasse, was die Prinzipien der Kapselung und der minimalen Kopplung verletzt. Wie Sie sehen, greife ich im Konstruktor der abgeleiteten Klasse auch nicht auf das Feld `fLaufbeine` zu, welches ausdrücklich als `private` deklariert wurde, sondern auf die Eigenschaft `Laufbeine`. Dadurch erhalten Sie die Kapselung Ihrer Klasse und geben nicht interne Strukturen, sondern abstrakte Schnittsellen für Daten frei. Wie die Eigenschaft `Laufbeine` intern gespeichert wird, ist für den Nutzer der Klasse völlig unerheblich. Hätte ich den Zugriff auf das Feld `fLaufbeine` erlaubt, indem ich es als `protected` (oder `public`) deklariert hätte, könnte ich den Namen und den Typ des Feldes nicht mehr folgenlos ändern, da eine abgeleitete Klasse möglicherweise bereits auf diese Informationen aufbaut. Sie können also bereits beim Design Ihrer Klasse durch einfache Änderungen dafür sorgen, dass manche Probleme erst gar nicht entstehen. Dieser Ansatz konsequent weitergedacht, führt uns dann später zu den Schnittstellen (*interfaces*).

11.1.1 Zugriff über die Basisklasse

Damit ist eine Instanz von `Katze` auch ein `Saeugetier`. Warum? Nun, Sie können jede abgeleitete Klasse auch mit dem Typ der Basisklasse referenzieren (nicht nur in C#), da eine `Katze` ja eben auch ein `Saeugetier` ist. Die Klasse `Katze` bietet die gesamte Funktionalität von `Saeugetier`, nur darüber hinaus noch zusätzliche Funktionalität. Damit ist die Behandlung einer `Katze`-Instanz als `Saeugetier`-Instanz völlig legal.

```
public class Test
{
   Katze k = new Katze("Neko", 3.8);
   Saeugetier s = k;
}
```

Listing 11.5: Zugriff auf eine Klasse über den Typ der Basisklasse

Mit dieser Möglichkeit wird in der objektorientierten Programmierung das Prinzip der Polymorphie realisiert. Ich werde darauf weiter unten bei den virtuellen Methoden zurückkommen, erinnern Sie mich daran …

11.1.2 Zugriff auf Mitglieder der Basisklasse

Nachdem C# nur eine einfache Vererbung unterstützt, ist es leicht, auf Mitglieder der Basisklasse zuzugreifen. Dazu benutzen Sie das Schlüsselwort `base`, welches eine Referenz auf die Basisklasse liefert.

Java

`base` entspricht im Großen und Ganzen dem Schlüsselwort `super` in Java.



```
public class Katze : Saeugetier
{

   public Katze(string name, float gew) :
      base(name, gew, 4)
      {
         // zugriff auf basisklasse
         base.Laufe();
      }

   // neue Methode Laufe, "new" unterdrückt warnung
   new public void Laufe()
   {
```

```
      Console.WriteLine("Katzen laufen auf " +
                        "{0} Samtpfoten.", Laufbeine);
   }

}
```

Listing 11.6: Zugriff auf die Basisklasse mit base

11.1.3 Zugriff auf Konstruktoren der Basisklasse

Wenn Sie eine Instanz einer Klasse erzeugen, wird für diese Klasse ein evtl. vorhandener Konstruktor aufgerufen. Nachdem die Klasse `Katze` ja ebenfalls ein `Saeugetier` ist (sie wurde davon abgeleitet), muss beim Erzeugen einer `Katze`-Instanz auch der Konstruktor für `Saeugetier` aufgerufen werden. Für die sog. default-Konstruktoren (ein Konstruktor ohne Parameter) kann der Compiler die Aufrufe für die gesamte Vererbungshierarchie bis zu `object` hinauf automatisch erledigen. Besitzt der Konstruktor aber Parameter, müssen Sie explizit einen Konstruktor der Basisklasse aufrufen, und zwar im Konstruktor der abgeleiteten Klasse, wie Sie bei Listing 11.4 sehen.

11.2 Steuern der Vererbung

11.2.1 Die Notwendigkeit virtueller Methoden

Warum ist die bisher mögliche Vererbung nicht ausreichend? Dazu deklariere ich ein Array von Säugern.

```
public class Test
{
   Saeugetier[] tiere = new Saeugetier[3];

   tiere[0] = new Saeugetier( "Mammut", 5800F, 4 );
   tiere[1] = new Katze( "Neko", 3.8 );
   tiere[2] = new Saeugetier( "Franz", 73.5, 2 );

   foreach(Saeugetier s in tiere)
   {
      s.Laufe();
   }

}
```

Listing 11.7: Ein Array von Säugetieren

Wird diese Anwendung gestartet, erzeugt diese die folgende Ausgabe:

```
Mammut läuft auf 4 Beinen.
Neko läuft auf 4 Beinen.
Franz läuft auf 2 Beinen.
```

Was ist mit den Samtpfoten passiert? Der Grund liegt in der konsequenten Anwendung der Vererbungshierarchie. Ich habe das Array als ein Array von `Saeugetier` deklariert, also wird eine angelegte Instanz von `Katze` als `Saeugetier` behandelt, mit allen Konsequenzen, eben auch dem Aufruf der Methode `Laufe()` der Basisklasse `Saeugetier` und nicht von `Katze.Laufe()`. Es wird also immer der Datentyp des Referenztyps (hier `Saeugetier[]`) betrachtet und nicht der tatsächliche Typ des Objekts.

11.2.2 Virtuelle Methoden

Zur Lösung dieses Problems bieten objektorientierte Programmiersprachen die so genannten virtuellen Methoden (*virtual methods*) an. Hier wird intern eine Tabelle mit den Adressen der Methoden für den tatsächlichen Datentyp des Objekts angelegt und diese Methode wird dann auch aufgerufen. Zur Demonstration werde ich den Code nun entsprechend umbauen.

```
/*
   Goto C# -- VirtualMethods
   Autor: Armin Hanisch
*/

// unsere basisklasse, erbt implizit von object
public class Saeugetier
{
   protected string fName;
   private int fLaufbeine;

   public float Gewicht;

   public Saeugetier(string name, float gew, int beine)
   {
      fName = name;
      fLaufbeine = beine;
      Gewicht = gew;
   }
```

```
   public int Laufbeine
   {
      get { return fLaufbeine; }
      set { fLaufbeine = value; }
   }

   virtual public void Laufe()
   {
      Console.WriteLine("{0} läuft auf {1} Beinen.",
                        fName, fLaufbeine);
   }

}


// die abgeleitete Katze-Klasse
public class Katze : Saeugetier
{

   // katzen laufen immer auf vier beinen,
   // daher nur im base-konstruktor anzahl laufbeine
   public Katze(string name, float gew) :
      base(name, gew, 4)
      {
      }

   override public void Laufe()
   {
      Console.WriteLine("Katzen laufen auf " +
                        "{0} Samtpfoten.", Laufbeine);
   }

}

// die Testklasse zum ausprobieren
public class Test
{
   Saeugetier[] tiere = new Saeugetier[3];

   tiere[0] = new Saeugetier( "Mammut", 5800F, 4 );
   tiere[1] = new Katze( "Neko", 3.8, 4 );
   tiere[2] = new Saeugetier( "Franz", 73.5, 2 );

   foreach(Saeugetier s in tiere)
   {
```

```
        s.Laufe();
    }

}
```

Listing 11.8: Die Klassen besitzen die virtuelle Methode Laufe()

Sie sollten mich doch an das Thema Polymorphie erinnern? Danke! Jetzt kann ich dieses Prinzip nämlich an einem praktischen Beispiel erklären. Wörtlich übersetzt bedeutet Polymorphie 'Vielgestaltigkeit' und genau das macht auch unser Code. Obwohl ich einfach ein Array vom Typ `Saeugetier` deklariert habe, 'weiß' die .NET-Laufzeitumgebung, welche Methode passend zum Typ der Instanz aufzurufen ist. Hätte ich nun auch noch eine Klasse Mensch abgeleitet und die entsprechenden virtuellen Methoden überschrieben, könnte ich einfach für ein Objekt die Methode `Laufe()` aufrufen. Solange es von `Saeugetier` abgeleitet ist, kümmert sich der Compiler um den Rest. Damit fallen diese endlosen Unterscheidungen des Datentyps vor dem Aufruf weg, welche bei der prozeduralen Programmierung immer eine Quelle für Fehler waren.

Die Ausgabe der Testklasse entspricht jetzt auch den Erwartungen:

```
Mammut läuft auf 4 Beinen.
Katzen laufen auf 4 Samtpfoten.
Franz läuft auf 2 Beinen.
```

Der C#-Compiler muss jetzt nur noch wissen, ob in der abgeleiteten Klasse eine Methode deklariert wurde, welche mit dem Schlüsselwort `override` die virtuelle Methode der Basisklasse überschreibt. Ist dies der Fall, wird die Adresse dieser Methode in die internen Tabellen des Objekts im Speicher eingetragen und zukünftig verwendet (die C#-Compilerbauer mögen mir diese Vereinfachung verzeihen).

11.2.3 Der Unterschied zwischen new und override

Sie können einmal im obigen Listing bei der Klasse `Katze` das Schlüsselwort `override` weglassen, aber in der Basisklasse `Saeugetier` die Methode `Laufe()` trotzdem als virtual deklarieren. Was passiert dann?

Die Ausgabe gleicht wieder der ersten Ausgabe. Es wird jetzt die Version der Basisklasse aufgerufen. Allerdings erhalten Sie einen Hinweis

des Compilers (in Visual Studio.NET der blaue Kringel unter dem Methodennamen), dass die Methode `Katze.Laufe()` nun die Methode `Saeugetier.Laufe()` (die Basisklassen-Methode) ausblendet und Sie sich zwischen `new` und `override` entscheiden sollten, um die gwünschte Absicht auch im Quelltext deutlich zu machen. Als Standard entscheidet der Compiler also, die neue Methode als unabhängig von der Methode der Basisklasse zu betrachten, was die sicherste Annahme darstellt.

Das Schlüsselwort `new` an dieser Stelle bietet Ihnen also die Möglichkeit, eine neue, von der Basisklasse unabhängige Methode, Eigenschaft oder Struktur einzuführen, welche den gleichen Namen besitzt. Um dann noch Zugriff auf die ursprüngliche Methode oder Eigenschaft zu bekommen, müssen Sie diese dann über den Verweis auf die Basisklasse mit `base.<Name>` ansprechen.



C++

Sie müssen für jede überschriebene Methode das Schlüsselwort override benutzen. virtual wird nur für die oberste Ebene einer Spezialisierung benutzt.

Das Schlüsselwort `override` legt zusammen eine Überschreibung fest, welche ab diesem Punkt in der Vererbungshierarchie sozusagen eine Spezialisierung darstellt. Dennoch kann das Objekt als Instanz der Basisklasse behandelt werden und wird die passende Methode aufrufen, da die neue Adresse in der Methodentabelle des Objekts eingetragen wird.

Wollen Sie an einer bestimmten Stelle in Ihrer Objekthierarchie einen neuen Punkt für Spezialisierungen durch überschriebene Methoden setzen, also sozusagen eine neue 'Basismethode' deklarieren, können Sie die Kombination aus den beiden Schlüsselwörtern benutzen und eine Methode als `virtual override` deklarieren.

Natürlich ist die Kombination aus `new` und `override` nicht erlaubt, da sich dies widersprechen würde.

11.2.4 Abstrakte Klassen

Niemand hindert Sie, genau wie im letzten Abschnitt erläutert, einfach das `override` für die Methode Ihrer abgeleiteten Klasse zu vergessen. Damit wäre dies eine Quelle für fehlerhaften Code, denn im schlimmsten Fall unterscheiden sich die Implementierungen der beiden Klassen deutlich. Es existiert aber in der objektorientierten Programmierung eine Möglichkeit, den Entwickler einer abgeleiteten Klasse zu zwingen, alle Methoden auch tatsächlich zu implementieren und zu überschreiben. Das Mittel hierzu heißt abstrakte Klassen.

Java



Möchten Sie eine abstrakte Klasse in C# erstellen, müssen Sie das Schlüsselwort `abstract` auch zusätzlich zu den Methoden bei der Klasse verwenden.

Eine abstrakte Klasse ist, wie der Name vermuten lässt, eine Art abstraktes Schema, wie die Klasse im Hinblick auf Eigenschaften und Methoden auszusehen hat. Vor weiteren Erklärungen möchte ich Ihnen eine abstrakte Säugetier-Klasse zeigen.

```
public abstract class Saeugetier
{
   protected string fName;
   private int fLaufbeine;

   public float Gewicht;

   public Saeugetier(string name, float gew, int beine)
   {
      fName = name;
      fLaufbeine = beine;
      Gewicht = gew;
   }

   public int Laufbeine
   {
      get { return fLaufbeine; }
      set { fLaufbeine = value; }
   }

   // keine implementierung!!
```

```
    abstract public void Laufe();

}
```

Listing 11.9: Eine abstrakte Klasse

Wie Sie erkennen, gibt es für die Methode `Laufe()` keinerlei Code. Die abstrakte Klasse legt nur fest, dass eine abgeleitete Klasse eine Methode mit dieser Signatur und diesem Namen implementieren muss.

Damit ist eine abstrakte Klasse nur eine Art von Schema oder Vertrag, den jede erbende Klasse zu erfüllen hat. Sie können eine abstrakte Klasse komplett mit abstrakten Mitgliedern ausstatten oder manche Mitglieder bereits in der abstrakten Klasse implementieren. Da der Klasse in jedem Fall ein Teil des nötigen Codes fehlt, bedeutet dies auch, dass Sie für diese Klasse keine Instanz erzeugen können. Sie muss zuerst in einer anderen Klasse abgeleitet werden, welche alle Methoden implementiert. Von dieser Klasse können Sie dann eine Instanz erzeugen.

Abstrakte Klassen sind somit auch der programmiertechnische Vorläufer von Schnittstellen (*interfaces*), welche ich Ihnen im nächsten Kapitel vorstellen möchte.

11.2.5 Verhindern der Vererbung

Viele Firmen leben davon, Komponenten für andere Entwickler zu erstellen oder ganze Klassenbibliotheken zu programmieren. Hier besteht in einer komponentenorientierten Internetumgebung natürlich ein gewisses Interesse daran, von bestimmten Klassen keine Ableitungen mehr erstellen zu lassen. Aber auch für das Codemanagement kann es in größeren Projekten sinnvoll sein, Klassen sozusagen zu 'versiegeln'.

Genau dies leistet das Schlüsselwort `sealed`. Es entspricht dem `final` in Java und verhindert, dass von dieser Klasse eine andere Klasse abgeleitet werden kann.

```
// versiegelte finale klasse
public sealed class Testklasse
{
```

```
   // mitglieder
}

// erzeugt einen fehler
public class GehtNicht : Testklasse
{
}
```

Listing 11.10: Sealed verhindert das Erben von dieser Klasse

11.3 Was alle erben – System.Object

Der Datentyp `object` stammt wie alle anderen Datentypen aller .NET-Sprachen von der Klasse `System.Object` (mit einem großen O) ab. Alles, was `System.Object` kann, kann damit auch jeder andere Datentyp in der .NET-Laufzeitumgebung.

Diese Klasse stellt unter anderem fünf Methoden zur Verfügung, die ich Ihnen kurz erläutern möchte, da zumindest ein Teil dieser Methoden auch in Ihren Anwendungen häufig benutzt oder überschrieben werden wird.

- `ToString()`
- `Equals()`
- `GetHashCode()`
- `GetType()`
- `Finalize()`

Die Methode `ToString()` liefert eine textuelle Darstellung des Objekts. Das Standardverhalten ist relativ einfach. Ohne Überschreiben liefert `ToString()` einfach den Namen der Klasse zurück. Normalerweise werden Sie daher in Ihren Anwendungen diese Methode überschreiben, um eine Ausgabe zu erhalten, mit der Sie beispielsweise auch einzelne Instanzen der Klasse auseinanderhalten können.

```
// beispiel für ToString
public class Testklasse
{
   readonly DateTime erzeugt;
   string fID;
```

```
   Testklasse(string ID)
   {
      this.erzeugt = DateTime.Now();
      this.fID = ID;
   }

   public override string ToString()
   {
      return fID + " von " + erzeugt.ToString();
   }

}
```

Listing 11.11: Überschreiben von ToString() für bessere Ausgaben

Die Methode `Equals()` benötigen Sie, um die Inhalte zweier Objekte zu vergleichen. Nachdem Objekte Referenztypen sind, vergleicht der normale Gleichheitsoperator == die Übereinstimmung der Referenz, nicht des Inhalts.

```
Katze k1 = new Katze("Miez", 4.0);
Katze k2 = new Katze("Maunz", 3.5);
Katze k3 = new Katze("Miez", 4.0);

if (k1 == k3)
   {
      // nur dann wahr, wenn k1 und k3
      // auf dasselbe objekt zeigen,
      // was hier nicht der Fall ist
   }
```

Listing 11.12: Indirekt kennt C# doch Zeiger, sie heißen halt Referenzen

Sie können Ihre eigenen Vergleiche bauen, indem Sie einfach die Methode `Equals()` überschreiben. Das Standardverhalten dieser Methode entspricht dem des Gleichheitsoperators, also ein Vergleich der Referenzen. Dies lässt sich aber schnell ändern. Ich verwende die Klasse `Saeugetier`, so erben alle davon abgeleiteten Klasse automatisch diese Funktionalität:

```
public class Saeugetier
{
   protected string fName;
   private int fLaufbeine;
```

```
   public float Gewicht;

   public Saeugetier(string name, float gew, int beine)
   {
      fName = name;
      fLaufbeine = beine;
      Gewicht = gew;
   }

   public int Laufbeine
   {
      get { return fLaufbeine; }
      set { fLaufbeine = value; }
   }

   public void Laufe()
   {
      Console.WriteLine("{0} läuft auf {1} Beinen.",
                        fName, fLaufbeine);
   }

   public bool Equals(object obj)
   {
      return (fName == (Saeugetier)obj.fName)
             && (LaufBeine ==
                (Saeugetier)obj.Laufbeine);
   }

}
```

Listing 11.13: Die Methode Equals wird überschrieben

Aus Gründen der Höflichkeit habe ich auf einen Gewichtsvergleich verzichtet. Beachten Sie, dass der Parameter für `Equals()` als `object` übergeben wird. Aus Gründen der Typsicherheit kann ich nicht einfach zwei Klassen vergleichen, sondern muss vorher über eine Typumwandlung (*type cast*) erst den Parameter `obj` als Instanz der Klasse `Katze` behandeln. Falls es sich nicht um eine solche Instanz handelt, wird ein Laufzeitfehler ausgelöst (s. ***Fehlerbehandlung***).

Jetzt können Sie zwei Instanzen auf Übereinstimmung des Inhalts vergleichen:

```
Katze k1 = new Katze("Miez", 4.0);
Katze k2 = new Katze("Maunz", 3.5);
Katze k3 = new Katze("Miez", 4.0);

if ( k1.Equals(k3) )
   {
      // jetzt ist diese bedingung erfüllt
   }
```

Listing 11.14: Vergleich mit Equals()

Wenn Sie für diesen Vergleich auch noch den Operator == benutzen möchten, bitte ich Sie um Geduld bis zum Abschnitt ***Überladen von Operatoren*** weiter hinten in diesem Buch.

Zum Vergleich indirekt dazu gehört auch die Methode `GetHashCode()`. Diese liefert einen für eine Instanz eines Objekts eindeutigen Integerwert zurück. Für die .NET-Laufzeitumgebung sind zwei laut `Equals()` 'gleiche' Objekte verpflichtet, den gleichen Wert bei `GetHashCode()` zu liefern. Falls Sie also die eine Methode überschreiben, müssen Sie dies in fast allen Fällen auch für die andere tun. Wenn Sie in Ihrer Klasse eine eindeutige Kennung speichern, die in einen `int` passt, können Sie diesen Wert für `GetHashCode()` benutzen, ansonsten müssen Sie sich einen entsprechenden Algorithmus überlegen.

Die letzte Methode in diesem Abschnitt ist `Finalize()`. Diese Methode ist für eine Freigabe von vom Objekt belegten nichtverwalteten Ressourcen gedacht. Eine detaillierte Behandlung finden Sie bei ***Konstruktoren und Destruktoren***.

11.4 Übungen

Ü

1. Worauf müssen Sie achten, wenn Sie die Methode `Equals` in einer Ihrer Klassen überschreiben?
2. Wie erhalten Sie Zugriff auf die Klassenmitglieder der Basisklasse ? Erstellen Sie eine Klasse, deren Konstruktor einen `int` als Parameter erwartet und diesen an den Konstruktor der Basisklasse weitergibt.
3. Was sind abstrakte Klassen und wo liegt der Unterschied zu mit `sealed` gekennzeichneten Klassen?

12 Fehlerbehandlung

12.1 Versuch und Fehler – try/catch

Einer der Vorteile bei der Fehlerbehandlung innerhalb von C# (und der gesamten .NET-Laufzeitumgebung) ist die Tatsache, dass der Code so geschrieben werden kann, als würde alles glatt gehen. Keine eingestreuten, tief verschachtelten Abfragen auf Fehlercodes, sondern Sie ordnen in Ihrer Klasse den Code in zwei Blöcken an.

12.1.1 Explizites catch

Der eine, mit dem vorangestellten Schlüsselwort `try`, enthält den auszuführenden Code, der andere die bei einem Laufzeitfehler abzuarbeitenden Programmteile. Dieser wird durch ein vorangestelltes `catch` gekennzeichnet. Hierzu gleich ein Beispiel, bei dem zur Laufzeit eine ganze Menge schief gehen kann:

```
/*
   Goto C# -- ExceptionDemo
   Autor: Armin Hanisch
*/

using System;

class Test
{
   double zahl1, zahl2;

   public static void Main()
   {
      try
      {
         zahl1 = double.Parse(Console.ReadLine());
         zahl2 = double.Parse(Console.ReadLine());

         Console.WriteLine("Ergebnis = {0}",
                           zahl1 / zahl 2);
      }

      catch (Exception e)
```

```
        {
            Console.WriteLine("Fehler! Exception: " +
                              e.Message);
        }

        Console.WriteLine("Hier geht's weiter");
    }
}
```

Listing 12.1: Eine einfache Form der Fehlerbehandlung

Der Code innerhalb des `try`-Blocks wird ausgeführt. Tritt kein Fehler auf, wird der nachfolgende `catch`-Block übersprungen und es geht bei der Konsolenausgabe weiter. Damit können Sie den Code Ihrer Anwendung so programmieren, dass der normale Ablauf eingehalten wird und müssen nicht nach jeder Aktion alle möglichen Fehler prüfen.



C++

In C# sind für eine Ausnahmebedingung nur von der Klasse System.Exception abgeleitete Klassen möglich. Sie können also in C# keine int-Exception auslösen.

Tritt aber innerhalb des `try`-Blocks ein Laufzeitfehler auf (der Benutzer gibt keine gültige Zahl ein, als zweite Zahl wird 0 eingegeben, etc.), so wird von der .NET-Laufzeitumgebung zuerst geprüft, ob der Code innerhalb eines solchen Blocks ausgeführt wird. Falls ja, wird nach dem dazugehörigen `catch`-Block gesucht. Die genauen Regeln dazu liefere ich im nächsten Abschnitt dieses Kapitels. Anschließend wird der Code zur Fehlerbehandlung in diesem `catch`-Block ausgeführt. Dieser Code kann die Ausnahmebedingung (*exception*) entweder für bearbeitet erklären oder neu auslösen.

12.1.2 Generelles catch

Sie können den `catch`-Block auch so schreiben, dass jede Exception von diesem Block behandelt werden soll. Dann können Sie die Angabe der Exception-Klasse weglassen:

```
try
    {
    }
```

```
catch
   {
   }
```

Listing 12.2: Eine generelle Catch-Anweisung

Dies sollte allerdings nur dann verwendet werden, wenn es wirklich nicht anders geht. Da hier keine Information vorliegt, welche Exception aufgetreten ist, können Sie unter Umständen auch nicht angemessen in diesem `catch`-Block auf den Fehler reagieren.

Mein Favorit ist immer noch ein Programm, welches mir die unvergessliche Dialogbox »Application Error! Error 9999: invalid error number« beschert hat (nein, ich verrate nicht, welche Anwendung das war …).

12.2 Die Hierarchie von Exceptions

Als Parameter für den `catch`-Block wird der Typ der Exception angegeben. Stehen mehrere zur Auswahl, sucht die .NET-Laufzeitumgebung nach dem besten passenden Block. Dies ist der `catch`-Block mit der Exception für den aufgetretenen Fehler, die von der Hierarchie her die genaueste Übereinstimmung liefert.

```
/*
   Goto C# -- ExceptionDemo
   Autor: Armin Hanisch
*/

using System;

class Test
{
   int zahl1, zahl2;

   public static void Main()
   {
      try
      {
         zahl1 = int.Parse(Console.ReadLine());
         zahl2 = int.Parse(Console.ReadLine());

         Console.WriteLine("Ergebnis = {0}",
                           zahl1 / zahl 2);
      }
```

```
        catch (DivideByZeroException e)
        {
           Console.WriteLine("Division durch 0!");
        }

        catch (ArithmeticException e)
        {
           Console.WriteLine("Arithmetikfehler!");
        }

        catch (Exception e)
        {
           Console.WriteLine("Fehler! Exception: " +
                             e.Message);
        }

        Console.WriteLine("Hier geht's weiter");
    }
}
```

Listing 12.3: Fehlerbehandlung mit mehreren Catch-Blocks

Dies ist zugleich auch eine der wenigen Stellen bei C#, wo die Reihenfolge von eigentlich unabhängigen Blöcken für die Programmausführung und die Kompilierung entscheidend ist. Nachdem die .NET-Laufzeitumgebung die `Catch`-Blöcke vom Spezifischen zum Allgemeinen hin durchsucht, müssen Sie diese Blöcke auch in der richtigen Reihenfolge anordnen.

Hätte ich den Block, der die `ArithmeticException` abfängt, vor den Block mit der `DivideByZeroException` gestellt, könnte ich den Code gar nicht übersetzen lassen, da sofort ein Compilerfehler erzeugt werden würde. Schließlich beinhaltet eine `ArithmeticException` auch die `DivideByZeroException`, daher würde dieser Block nie ausgeführt werden.

Keine Exceptions bei double / float

Wann immer ich irgendetwas Neues lerne, sei es beim Programmieren, Kochen oder einer anderen Tätigkeit, kommt der Moment, an dem ich denke: »Jawohl, langsam hast Du die Sache verstanden«. Sie ahnen, was kommt? Richtig, exakt eine Sekunde später schlägt Mr. Murphy unerbittlich zu und es passiert etwas, mit dem ich bis dahin garantiert nicht gerechnet hatte.

Genau dieses Erlebnis hatte ich auch im Herbst 2000, als ich die ersten Spielereien mit C# versucht habe. Von fast allen anderen Programmiersprachen war ich gewohnt, dass eine Division durch Null nicht nur bei Integerwerten, sondern auch bei Gleitkommazahlen einen Fehler auslöst bzw. das Ergebnis (wie in C++) undefiniert ist. Genau das ist in C# (und auch anderen Sprachen wie Java) nicht der Fall, da Sie einem beinahe 20 Jahren alten Standard (IEEE 754) folgen. Vielleicht hätte ich damals an der Schule und in der Uni in Mathematik doch besser aufpassen sollen …

Langer Rede kurzer Sinn: wie auch Java definiert C# für diese Fälle spezielle Konstanten, mit denen Sie dann das Ergebnis der Berechnung vergleichen können, da eine Operation mit dem Typ `double` oder `float` nie eine Exception auslöst (weshalb ich im letzten Beispiel den Datentyp `int` benutzt habe, falls Sie sich das schon gefragt haben). Hier also die möglichen Resultate:

Die Operation 1.0 / 0.0 (bzw. double durch 0.0) ist definiert als +unendlich und entspricht in C# der Konstanten `double.PositiveInfinity`.

Die Operation -1.0 / 0.0 (bzw. double durch 0.0) ist definiert als -unendlich und entspricht in C# der Konstanten `double.NegativeInfinity`.

Die Operation 0.0 / 0.0 ist mathematisch nicht möglich und wird durch die Konstante `double.NaN` (*not a number*) ausgedrückt.

Für die leichtere Abfrage existieren die entsprechenden Methoden `IsNaN()`, `IsPositiveInfinity()`, `IsNegativeInfinity()` bzw. `IsInfinity()`. Natürlich gilt alles, was ich hier für `double` geschrieben habe, analog für `float`. Die Definitionen für diese unendlichen Konstanten wurden durch eben diese Operationen getätigt und ja, der Vergleich der beiden Konstanten `double.PositiveInfinity` und `float.PositiveInfinity` ergibt `true`.

12.3 Ausnahmen – System.Exception & Co

In der .NET-Laufzeitumgebung sind die gängigsten Ausnahmen (*exceptions*) vordefiniert und hierarchisch geordnet. An oberster Stelle steht die Klasse `System.Exception`, von der alle anderen Exceptions abgeleitet werden. Die Tabelle zeigt Ihnen die wichtigsten vordefinierten Exceptions. Diese werden alle von der Klasse `SystemException` abgeleitet, welche wiederum von `Exception` erbt. Diese Klassen wiederum sind die Basisklassen für spezifischere Fehlerbedingungen, bei-

spielsweise finden Sie die `DivideByZeroException` als Nachfahre von `ArithmeticException`. Details dazu finden Sie in der Online-Dokumentation zum .NET Framework SDK:

SystemException	OutOfMemoryException
AppDomainUnloadedException	RankException
ArgumentException	AmbiguousMatchException
ArithmeticException	ReflectionTypeLoadException
ArrayTypeMismatchException	ExternalException
BadImageFormatException	InvalidComObjectException
CannotUnloadAppDomainException	InvalidOleVariantTypeException
LicenseException	MarshalDirectiveException
WarningException	SafeArrayRankMismatchException
ConfigurationException	SafeArrayTypeMismatchException
InstallException	RemotingException
ContextMarshalException	ServerException
DataException	SerializationException
DBConcurrencyException	CryptographicException
SqlException	PolicyException
SqlTypeException	SecurityException
RegistrationException	VerificationException
ServicedComponentException	XmlSyntaxException
ExecutionEngineException	TimeoutException
FormatException	StackOverflowException
IndexOutOfRangeException	SynchronizationLockException
InvalidCastException	ThreadAbortException
InvalidOperationException	ThreadInterruptedException
InvalidProgramException	ThreadStateException
InternalBufferOverflowException	TypeInitializationException
ManagementException	TypeLoadException

Tabelle 12.1: Die von System.Exception abgeleiteten Ausnahmefehler

SystemException	OutOfMemoryException
MemberAccessException	TypeUnloadedException
MulticastNotSupportedException	UnauthorizedAccessException
NotImplementedException	XmlSchemaException
NotSupportedException	XmlException
NullReferenceException	XsltException

Tabelle 12.1: Die von System.Exception abgeleiteten Ausnahmefehler (Forts.)

12.4 Schutzblocks mit finally

In vielen Fällen reicht es nicht aus, einfach nur einen Laufzeitfehler abfangen zu können, sondern es sind bei der Behandlung des Fehlers auch alle bereits belegten Ressourcen wieder freizugeben, Dateien zu schließen und andere Dinge. Wird dies nicht gut geplant, kann der entsprechende Aufruf unterbleiben und Ihre Anwendung wird dadurch Speicher verlieren oder gar instabil.

Aus diesem Grund gibt es einen dritten Block für die Behandlung von Laufzeitfehlern in C#, den `finally`-Block. Wie der Name sagt, wird dieser Block nach dem Code und einer evtl. Fehlerbehandlung ausgeführt, ganz gleich, ob ein Fehler auftrat oder nicht. Damit können Sie sicherstellen, dass die Methode das System in einem konsistenten Zustand verlässt. Hier ein kleines Listing-Skelett, welches dieses Konzept verdeutlicht.

```
using System;

class Test
{
   public static void Main()
   {
      try
      {
         // Datei öffnen
      }

      catch (Exception e)
      {
         // Behandlung von laufzeitfehlern
```

```
        }

        finally
        {
           // Datei schließen
        }

    }
}
```

Listing 12.4: Ressourcenschutz mit finally

Im `catch`-Block können Sie sich ganz um die Behandlung des Fehlers kümmern, ohne darauf achten zu müssen, dass die Datei noch geschlossen werden muss. Auch im normalen `try`-Codeblock wird die Datei nicht wieder geschlossen. Gleichgültig, ob ein Fehler auftritt oder nicht, sorgt der `finally`-Block am Ende dafür, dass die Datei geschlossen wird. Dies zeigt auch gleich den Haupteinsatzzweck einer solchen Konstruktion, nämlich zum Schutz vor Ressourcenverlust und Speicherlecks.

Ein solcher `finally`-Block kann mit oder ohne einen `catch`-Block verwendet werden. Da dieser `finally`-Block sozusagen eine Art Sicherheitszone darstellt, löst jeder Versuch, die Programmausführung mit `return`, `goto` oder einer anderen Anweisung aus dem `finally`-Block herauszuführen, einen Fehler aus. Tritt während der Abarbeitung des `finally`-Blocks eine Exception auf, so wird diese vom dazugehörigen `try`-Block übernommen.

Zum Thema 'Aufräumarbeiten' sei auf Kapitel 7 ***Klassen in C#*** verwiesen, in dem es auch um Destruktoren geht.

12.5 Kaskadierende Exceptions

Laufzeitfehler verhalten sich in C# wie Gasbläschen in Sekt (natürlich nur in übertragenem Sinne): sie blubbern nach oben. Konkret bedeutet dies, dass bei einem Laufzeitfehler in einer Methode zuerst in dieser Methode nach einem try/catch gesucht wird. Das folgende Beispiel soll dies verdeutlichen. Hier werden in einer Routine Notendurchschnitte für ein Array mit Schulnoten von Eins bis Sechs berechnet und ausgegeben. Natürlich habe ich dafür gesorgt, dass ein Fehler auftritt …

```
/*
   Goto C# -- ExceptionBubble
   Autor: Armin Hanisch
*/

using System;

class Test
{
   int zahl1, zahl2;

   double GetDurchschnitt(int[] werte)
   {
      int anzahl = werte.GetLength(0);
      int summe = 0;

      foreach(int w in werte)
         summe += w;

      return (double)summe / anzahl;
   }

   public static void Main()
   {
      int[] mess1 = {3, 4, 2, 1, 5, 2, 1, 4};
      int[] mess2 = {};

      try
      {
         Console.WriteLine( "Durchschnitt 1 = {0}",
                            GetDurchschnitt(mess1) );

         Console.WriteLine( "Durchschnitt 2 = {0}",
                            GetDurchschnitt(mess2) );
      }

      catch (ArithmeticException e)
      {
         Console.WriteLine("Arithmetikfehler!");
      }

      catch (Exception e)
      {
         Console.WriteLine("Fehler! Exception: " +
                           e.Message);
```

```
        }

    }
}
```

Listing 12.5: Exceptions laufen in der Aufrufhierarchie nach oben

Der erste Aufruf von `GetDurchschnitt()` in `Main()` klappt problemlos, der zweite mit einem leeren Array liefert für die Länge des Arrays den Wert Null und produziert damit eine Division durch Null. Innerhalb der Methode `GetDurchschnitt()` ist aber kein `catch`-Block definiert, also wird aus der Methode heraus in `Main()` gesprungen und dort der entsprechende Exception-Block aktiviert.

Wird keine solche Konstruktion gefunden, wird innerhalb des übergeordneten Blockes gesucht, bis die Suche schließlich auf der Ebene der Anwendung mit der Standardroutine für Laufzeitfehler endet (die zeigt Ihnen dann normalerweise die Dialogbox mit der Fehlermeldung an, abhängig von der Konfiguration der .NET-Laufzeitumgebung).

12.6 Exceptions selbst gemacht

12.6.1 Ableitung der Exception-Klasse

Im letzten Beispiel wäre eine Fehlermeldung, welche eher im Problemraum des Anwenders liegt, angebrachter gewesen. Anstelle von »**Division durch Null**« wäre der Hinweis, dass das übergebene Array leer ist und damit kein Durchschnitt von Null Werten berechnet werden, sicher sinnvoller. Die .NET-Laufzeitumgebung kennt aber keine 'Dein Array ist leer'-Exeption. Was also tun?

Ich würde diesen Abschnitt nicht schreiben, wenn es dafür nicht eine Lösung gäbe. Sie können problemlos eigene Exception-Klassen von denen der .NET-Laufzeitumgebung ableiten und diese dann mit einer besser passenden Meldung an den Benutzer ausstatten. Dazu erben wir von der passenden Exception und programmieren einfach eine neue Klasse.

```
/*
   Goto C# -- UserException
   Autor: Armin Hanisch
```

```
*/

// die selbst definierte exception
public class CountZeroException : Exception
{
   // der default-konstruktor
   public CountZeroException()
   {
   }

   // der konstruktor nur mit message
   public CountZeroException(string message) :
      base(message)
   {
   }

   // konstruktor mit message und inner exception
   public CountZeroException(string message,
                             Exception inner) :
      base(message, inner)
   {
   }
}
```

Listing 12.6: Die erste selbstdefinierte Exception-Klasse

Jetzt fehlt nur noch der Umbau der Methode `GetDurchschnitt()`, um bei Bedarf diese neue `exception` auslösen zu können.

12.6.2 Auslösen einer Exception

Dazu wird der Code des vorigen Beispiels wie folgt abgeändert:

```
// unsere testklasse
class Test
{
   int zahl1, zahl2;

   double GetDurchschnitt(int[] werte)
   {
      int anzahl = werte.GetLength(0);
      int summe = 0;

      if (anzahl == 0)
```

```
            throw(
               new CountZeroException("Anzahl ist Null!")
                 );
         else
            {
               foreach(int w in werte)
                  summe += w;

               return (double)summe / anzahl;
            }
      }

      public static void Main()
      {
         int[] mess1 = {3, 4, 2, 1, 5, 2, 1, 4};
         int[] mess2 = {};

         try
         {
            Console.WriteLine( "Durchschnitt 1 = {0}",
                               GetDurchschnitt(mess1) );

            Console.WriteLine( "Durchschnitt 2 = {0}",
                               GetDurchschnitt(mess2) );
         }

         catch (ArithmeticException e)
         {
            Console.WriteLine("Arithmetikfehler!");
         }

         catch (Exception e)
         {
            Console.WriteLine("Fehler! Exception: " +
                              e.Message);
         }

      }
}
```

Listing 12.7: Klare Fehlermeldung durch passende Exception

Über das Schlüsselwort `throw` lässt sich natürlich auch jede bereits vordefinierte Exception auslösen.

Befinden Sie sich gerade in einem `catch`-Block bei der Bearbeitung einer Exception und möchten diese erneut auslösen, um so beispielsweise die Fehlerbehandlung in der Aufrufhierarchie nach oben zu eskalieren, lassen Sie einfach den Parameter mit dem Exception-Objekt weg. Schließlich weiß die .NET-Laufzeitumgebung durch den aktiven `catch`-Block, welche Exception gemeint ist.

```
catch (ArithmeticException e)
{
   Console.WriteLine("Arithmetikfehler!");

   // Exception erneut auslösen
   throw;
}
```

Listing 12.8: Erneutes Auslösen einer Exception

12.7 Zusammenfassung

Die strukturierte Fehlerbehandlung mit Hilfe von Ausnahmebedingungen, den Exceptions, bietet eine wesentliche bessere Methode als die dauernde Prüfung aus Fehlercodes. Der Programmcode wird dadurch leichter lesbar und besser strukturiert.

Mit Hilfe des Schlüsselwortes `finally` sind Sie in der Lage, auch beim Auftreten einer Exception noch Code auszuführen, um damit beispielsweise Dateien zu schließen oder Ressourcen freizugeben.

Exception-Blöcke können geschachtelt werden und werden hierarchisch mit dem Klassenbaum der definierten Exception-Klassen verglichen, so dass Sie durchaus auch eine allgemeine Fehlerprüfung definieren können.

Eigene Exception-Klassen lassen sich jederzeit durch die Ableitung von einer der bestehenden Klassen erzeugen. Damit sind programm- und aufgabenspezifische Fehlermeldungen in jeder Anwendung leicht zu erzeugen.

12.8 Übungen



1. Erstellen Sie eine Methode, die prüft, ob eine als Parameter übergebene Datei auch vorhanden ist und bei Nichtvorhandensein eine entsprechende Exception auslöst (benutzen Sie dazu die Klasse `System.IO.File`).
2. Was passiert, wenn Sie in Ihrer Anwendung eine Exception auslösen, diese aber nicht behandeln?
3. Unter welchen Umständen kann es sinnvoll sein, 'stille Exceptions' zu haben, also bestimmte Exceptions einfach ohne Reaktion abzufangen und wie würden Sie dies programmieren?
4. Wie fangen Sie eine Divison durch 0 bei einer `double`-Variablen ab?

13 Schnittstellen (Interfaces)

In diesem Kapitel stelle ich Ihnen die Weiterentwicklung der reinen klassenbasierten Vererbung vor, die schnittstellenbasierte Programmierung (*interface based programming*). Wie Sie aus den vorherigen Kapiteln erfahren haben, ist auch die objektorientierte Programmierung nicht frei von diversen Fallstricken. Einige Probleme werde ich in dieser Einführung anreißen, so dass Sie verstehen, warum Schnittstellen (*interfaces*) ein so flexibles und mächtiges Werkzeug darstellen.

Noch ein kleiner Hinweis in eigener Sache: Wir alle sind Gewohnheitstiere. Da ich ab und zu auch mal englischsprachige Trainings für Entwickler halte, meine Kommentare in meinen eigenen Programmen ebenfalls auf Englisch schreibe (dann muss ich beim Lesen des Codes nicht dauernd die Sprache wechseln), kann es durchaus sein, dass das eine oder andere Mal anstelle von Schnittstelle noch der Begriff *interface* auftaucht. Nehmen Sie's meiner Lektorin also nicht übel, Sie kann nichts dafür, es ist mein Fehler …

13.1 Probleme klassenbasierter Vererbung

Wenn Sie eine Klasse erstellen, können Sie bestimmte Details der Implementierung vor den Benutzern dieser Klasse verbergen, indem Sie diese als `private` deklarieren und damit das Prinzip der Kapselung verwirklichen. Dennoch sind Klassenhierarchien von aufeinander aufbauenden Klassen mögliche Problemfälle einer solch reinen klassenbasierten Vererbung. Für die Vererbung können Sie Mitglieder von Klassen als `protected` deklarieren. Damit sind diese Mitglieder nur innerhalb der Klasse selbst oder in allen davon abgeleiteten Klassen sichtbar. Genau damit beginnen aber die Probleme.

Wenn Sie (oder ein anderer Entwickler) nun eine Klasse von der Basisklasse ableiten und dabei auf Mitglieder der Basisklasse zugreifen, nutzen Sie interne Details der Implementierung der Basisklasse. Eine Änderung der Basisklasse bedeutet damit implizit auch immer eine Änderung und Anpassung aller davon abgeleiteten Klassen. Den Aufwand bei einer Klassenhierarchie, die auch nur einen winzigen Teil der .NET-Laufzeitumgebung ausmacht, können Sie sich sicher ausmalen.

Andererseits ist es aber auch kaum möglich, das ursprüngliche Design der Basisklasse auf alle Zeiten einzufrieren, da Sie dann die Klassenhierarchie nicht wieder erweitern können.

Ein anderes Problem ist eine Weiterentwicklung einer Methode in einer der Basisklassen. Falls Sie zusätzliche (oder weniger) Parameter benötigen, ändern Sie damit die Signatur der Methode und alle abgeleiteten Klassen müssen geändert werden. Die einzige Alternative ist meist eine Änderung des Namens (wir alle kennen Win32-Aufrufe wie `ExitWindowsEx`), die aber ab einer bestimmten Zahl von Änderungen nur noch wenig Ähnlichkeit mit dem ursprünglichen Namen besitzen.

Bei Feldern gibt es in C# immerhin den Ausweg über Eigenschaften, welche die tatsächliche Implementierung verbergen, in der Basisklasse sind Sie damit nur noch auf den Namen der Eigenschaft festgelegt. Bei Methoden oder Ereignissen haben Sie diese Möglichkeit allerdings nicht.

Das erfolgreiche Design einer größeren Klassenhierarchie ist daher von den Anforderungen durchaus mit dem Schachspiel zu vergleichen. Die Regeln und möglichen Spielzüge sind in einer halben Stunde erklärt. Bis jemand Schach wirklich spielen kann, vergehen in der Regel Jahre (ich arbeite seit fast 8 Jahren mit Klassen und lerne seit 25 Jahren Schach und würde nicht sagen, dass ich eines davon wirklich beherrsche). In kleinen und mittleren Systemen dagegen lässt sich die klassenbasierte Vererbung gut einsetzen. Ebenso nützlich ist sie bei Szenarien, in denen das Wachstum bzw. die Evolution einer Klassenhierarchie von vorneherein gut abschätzbar und sehr gering ist.

Dieses Problem bezeichne ich als das *Affe mit der Banane-Szenario*. Denn was war eigentlich der Grund für Vererbung in der objektorientierten Programmierung? Die Wiederverwendung von Code, eben das Nutzen von Funktionalität in einer abgeleiteten Klasse, die von der Basisklasse bereits zur Verfügung gestellt wird und nur partiell erweitert oder ergänzt wird. Genau dieser Ansatz führte aber bei großen Klassenhierarchien zu der oben beschriebenen engen Kopplung durch die gegenseitige Abhängigkeit von Implementierungsdetails. Sie sollten aber nicht ein Prinzip in der objektorientierten Programmierung verwirklichen, indem Sie ein anderes (das der möglichst geringen Kopplung) brechen. Ansonsten haben Sie am Ende ein System, bei dem Sie ent-

weder alles oder gar nichts aus einer riesigen, unflexiblen Klassenhierarchie benutzen. Sie können also nicht nur die Banane haben, sondern müssen auch den Affen nehmen, der die Banane hält, den Baum, auf dem der Affe sitzt und den ganzen Urwald drum herum …

Die eigentliche Anforderung der damaligen OOP-Pioniere war in Wirklichkeit nicht die Wiederverwendbarkeit von Code (*code reusability*). Gemeint war tatsächlich die Wiederverwendung von Funktionalität, also eine Art Zusammenstellen von Objekten aus klar definierten Blöcken von Funktionalität (Was will ich?), nicht Zugriff auf bestehenden Code (Wie will ich das?). Schließlich interessiert mich als Autofahrer auch nicht, wie die Implementierung meines Verkehrsservices für das Autoradio aussieht. Was mich interessiert, ist die Funktionalität der Verkehrshinweise. Die Komponenten des Systems werden als echte *black box* betrachtet. Die gegenseitige Kopplung wird auf ein Minimum (nämlich die Beschreibung einer 'Schnittstelle') reduziert. Damit kann die Funktionalität aber problemlos wieder verwendet werden. Die Schnittstelle 'Verkehrsservice' garantiert mir unabhängig vom Sender, dass während der Ausstrahlung des entsprechenden Pilottons eine Verkehrsdurchsage erfolgt und dass die Durchsage endet, bevor der Pilotton abgeschaltet wird.

Daher sind Schnittstellen (*interfaces*) eine grundlegende Weiterentwicklung des Konzepts der objektorientierten Programmierung. Was vorher zusammen mit der Implementierung in einer Klasse vorhanden war, kann jetzt als reine Beschreibung einer Art von Vertrag zwischen Implementierung und Nutzung separat erstellt werden. Diese Trennung zwischen Beschreibung der Funktionalität und deren tatsächlicher Implementierung ermöglicht eine ganz andere Art der Softwareentwicklung. Schnittstellenbasierte Programmierung unterscheidet sich deutlich von der rein klassenbasierten objektorientierten Programmierung, da hier durch die geringere Kopplung und stärkere Kapselung eine andere Art des Designs möglich wird. Der Aspekt einer *black box*-Nutzung wird auch dadurch unterstützt, dass eine Schnittstelle immer vollständig oder überhaupt nicht implementiert wird. Der Benutzer einer solchen Schnittstelle kann sich also darauf verlassen, dass eine angebotene Methode auch mit einer entsprechenden Funktionalität ausgestattet ist.

Da eine Schnittstelle eine reine Beschreibung ist ('ich garantiere Dir, dass Du meine Methode `Machwas()` aufrufen kannst und diese einen `string` liefert'), bilden Schnittstellen eine ideale Basis für Polymorphie. Jede Klasse kann eine eigene Version von `Machwas()` implementieren, abhängig von den Anforderungen an diese Klasse. Ein Aufruf kann nun unabhängig vom Objekt erfolgen, es wird dann jeweils die 'passende' Version von `Machwas()` aufgerufen.

Ein weiterer Vorteil ist die Weiterentwicklung von Softwaresystemen und Klassen. Ergibt sich die Notwendigkeit einer erweiterten Funktionalität oder von zusätzlichen Parametern, müssen Sie keine Klassen ändern, sondern definieren einfach ein zusätzliches Interface. Dessen Methoden können genau den gleichen Namen besitzen, aber die verbesserte Funktionalität oder andere Signaturen enthalten. Wenn eine Klasse nun beide Schnittstellen implementiert, können alte Clients ohne Probleme und ohne eine einzige Zeile Code zu ändern immer noch weiterarbeiten. Neue Clients nutzen die neue Schnittstelle und kommen in den Genuss der verbesserten Funktionalität, ohne dass sich der Entwickler neue Namen und Methoden merken müsste. Diese gesamte Versionisierung einer objektorientierten Entwicklung ist über Schnittstellen also wesentlich besser steuerbar. Über die in diesem Kapitel vorgestellten Möglichkeiten des Tests auf die Unterstützung einer bestimmten Schnittstelle lässt sich die passende Funktionalität leicht auswählen.

Die mit einer schnittstellenbasierten Programmierung verbundenen Vorteile lassen sich nicht über Nacht erschließen. Wenn Ihnen das alles noch etwas unklar vorkommt oder Sie bisher keinen rechten Grund gesehen haben, in Ihren Anwendungen anstelle der klassenbasierten Vererbung Schnittstellen zu benutzen, brauchen Sie sich keine Sorgen zu machen. Erst mit zunehmender Praxis und in größeren Projekten erschließen sich die Vorteile dieser Art der Softwareentwicklung.

13.2 Was sind Interfaces



C++ / Java / Delphi

Um es für Umsteiger mit Vorkenntnissen wieder kurz zu machen, Schnittstellen sind für C++-Entwickler abstrakte Klassen mit vollständig abstrakten virtuellen Methoden ohne jegliche Felder und Imple-

mentierung. Schnittstellen in Java sind ziemlich das Gleiche wie die Schnittstellen in C#. Schnittstellen in Delphi entsprechen ebenfalls prinzipiell den C#-Schnittstellen.

Ab hier also etwas ausführlicher: Eine Schnittstelle ist ein Vertrag zwischen zwei Softwareparteien. Insofern ist der Begriff Schnittstelle völlig richtig gewählt. Diese Schnittstelle definiert genau, welche Funktionalität eine Klasse zur Verfügung stellt, die diese Schnittstellen implementiert. Keine Chance, an irgendeinem offiziellen Weg vorbei auf Interna einer Klasse zuzugreifen. Alles, was erlaubt ist, sind die in der Schnittstelle aufgeführten Möglichkeiten. Darüber hinaus ist eine Schnittstelle auch eine 'Alles oder Nichts'-Geschichte: Sie werden entweder komplett oder überhaupt nicht implementiert. Dies bedeutet, dass Sie sich als Benutzer einer Schnittstelle darauf verlassen können, dass die gesamte Funktionalität und nicht nur ein Teil auch vorhanden ist und genutzt werden kann.

13.2.1 Deklaration einer Schnittstelle in C#

Die Deklaration einer Schnittstelle folgt in C# dem Muster einer Klassendefinition, allerdings enthält die Schnittstelle keinen Implementierungscode, sondern nur die Beschreibung (den 'Vertrag') der Funktionalität:

```
public interface ILaufen
{
   void Laufe();
   double GetGeschwindigkeit();
}
```

Listing 13.1: Eine einfache Schnittstellendefinition

Eine Schnittstelle darf keine Felder oder Implementierungscode enthalten. Erlaubt sind nur Methoden, Eigenschaften, Ereignisse und Indexer als Deklarationsrümpfe. Wie bereits in der Einführung erläutert, ist damit eine Art der Programmierung möglich, welche sich auf die Beschreibung von Verhalten und Eigenschaften beschränkt und die tatsächliche Implementierung einer Klasse außen vor lässt. Diese Möglichkeiten sind allerdings wie vieles andere in der objektorientierten Programmierung kein Automatismus, sondern eine dem Entwickler angebotene Möglichkeit. Doch genug der grauen Theorie, ich möch-

ten Ihnen jetzt zeigen, wie eine Klasse von einer solchen Schnittstelle erben, d.h. diese Schnittstelle implementieren, kann.

13.2.2 Implementieren einer Schnittstelle

Das Implementieren einer Schnittstelle geschieht einfach über die Erstellung der entsprechenden Methoden in der Klasse. Dazu geben Sie den Namen der Schnittstelle (wie bei einer Vererbung durch einen Doppelpunkt getrennt) hinter dem Namen der Klasse an. Erbt die Klasse von einer Basisklasse und Sie implementieren zusätzlich eine Schnittstelle oder implementieren mehr als eine Schnittstelle, trennen Sie die Bezeichnungen nach dem Doppelpunkt durch Kommata.

```
public class ImplKlasse : ILaufen
{
   // konstruktor und klassenmethoden
   public ImplKlasse()
   {
   }

   protected Machwas()
   {
   }

   // die methoden der schnittstelle
   public void Laufe()
   {
      Console.WriteLine("Läuft und läuft ...");
   }

   public double GetGeschwindigkeit();
   {
      double ergebnis;
      // berechnung hier
      return ergebnis;
   }
}
```

Listing 13.2: Implementieren einer Schnittstelle

Damit die Methoden auch für die Implementierung der Schnittstelle genutzt werden können, musste ich diese als public deklarieren, sonst wären sie ja nicht von 'außerhalb' der Klasse sichtbar.

13.2.3 Schnittstellen mit Eigenschaften

Nachdem die Definition einer Schnittstelle keinen Code enthalten darf, kann auch bei der Deklaration einer Schnittstelle mit einer oder mehreren Eigenschaften nur das 'Skelett' dieser Eigenschaft angegeben werden.

```
interface ILaufen
{
   void Laufe();

   double Schrittlaenge
   {
      get;
      set;
   }
}
```

Listing 13.3: Eine Schnittstelle mit einer Eigenschaft

Hier erzeuge ich für die Definition der Schnittstelle eine Eigenschaft mit dem Namen `Schrittlaenge`, welche gelesen und geschrieben werden kann (daher `get` und `set`) vom Datentyp `double`. Damit ist festgelegt, dass eine Implementierung der Schnittstelle diese Eigenschaft zur Verfügung stellen muss. Nicht festgelegt dagegen ist die interne, physische Realisierung dieser Eigenschaft. Diese ist für einen Nutzer der Schnittstelle aber auch gar nicht von Bedeutung. Sie sehen also, Schnittstellen unterstützen sehr schön das OOP-Prinzip der Kapselung und Abstraktion von Daten. Nach dem Motto 'Je krasser, desto Beispiel' erklärt: Sie können den aktuellen Wert der Eigenschaft auch beim Setzen auf einem Laserdrucker ausgeben und beim Lesen per OCR mit dem Scanner wieder einlesen. Für eine Klasse, welche die Schnittstelle benutzt, macht dies keinen Unterschied.

13.3 Arbeiten mit Schnittstellen

13.3.1 Benutzen einer Schnittstelle

Sie nutzen eine Schnittstelle wie eine Klasse, indem Sie eine Referenz darauf erstellen und die entsprechenden Methoden aufrufen. Diese Referenz erhalten Sie, indem Sie eine Instanz einer Klasse erstellen,

welche die gewünschte Schnittstelle implementiert. Intuitiv würden Sie vielleicht Folgendes versuchen:

```
public class Testklasse
{
   public static void Main()
   {
      // instanz der klasse erstellen
      ImplKlasse obj = new ImplKlasse();

      // instanz der schnittstelle erstellen
      // Funktioniert so nicht!! Siehe Text.
      ILaufen il = new ImplKlasse();
   }
}
```

Listing 13.4: Dieser Ansatz funktioniert nicht

Wenn Sie dies versuchen, erhalten Sie eine Fehlermeldung beim Übersetzen des Codes. Warum? Nun, die Klasse `ImplKlasse` könnte ja mehr als eine Schnittstelle implementieren. Außerdem ist C# eine typsichere Sprache und auch bei Schnittstellen können Sie nicht einfach einer Variablen eines Typs einen Inhalt eines anderen Typs zuweisen. Hier kommen wieder die Typ-Operatoren aus dem Kapitel ***Anweisungen und Operatoren*** ins Spiel. Mit Hilfe des Operators `as` lässt sich ein Objekt in einen anderen Klassen- oder Schnittstellen-Typ umwandeln.

```
public class Testklasse
{
   public static void Main()
   {
      // instanz der klasse erstellen
      ImplKlasse obj = new ImplKlasse();

      // instanz der schnittstelle erstellen
      // das funktioniert!
      ILaufen il = ( new ImplKlasse() ) as ILaufen;
   }
}
```

Listing 13.5: Typwandlung mit dem as-Operator

Die beiden Klammern um `new ImplKlasse()` sind nicht nötig, ich schreibe den Code aber trotzdem gerne so. Überfliegt jemand, der vielleicht neu in C# ist, den Code, kann er sofort erkennen, dass ich eine neue Instanz in den Typ der Schnittstelle umwandle.

Jetzt können Sie die entsprechenden Methoden, z.B. `il.Laufe()` aufrufen und nutzen. Wesentlich interessanter ist aber die Tatsache, dass Sie an eine Methode, die einen Parameter vom Typ `ILaufen` erwartet, ohne Probleme auch eine Instanz von `ImplKlasse` übergeben können.

```
public class Testklasse
{
   public static void Main()
   {
      // instanz der klasse erstellen
      ImplKlasse obj = new ImplKlasse();

      // schnittstelle erstellen
      ILaufen il = ( new ImplKlasse() ) as ILaufen;

      Machwas(il);    // funktioniert sowieso
      Machwas(obj);   // funktioniert ebenfalls!
   }

   void Machwas(ILaufen intf)
   {
      if(intf.GetGeschwindigkeit() < 20)
         Console.WriteLine("Noch laufbar.");
      else
         Console.WriteLine("Zu schnell für Menschen");
   }
}
```

Listing 13.6: Die Instanz der Klasse kann als Schnittstelle betrachtet werden

Hier passiert Folgendes: Da die Klasse ja die Implementierung der Schnittstelle enthält, kann eine Instanz dieser Klasse als Schnittstelle aufgefasst werden. Damit werden dann aber auch alle nicht in der Definition der Schnittstelle enthaltenen Mitglieder der Klasse (Felder, Methoden, Eigenschaften, etc.) ausgeblendet, da die Methode nur an der Implementierung der Schnittstelle interessiert ist.

Was aber, wenn Sie (oder ich) durch einen Schreibfehler eine Klasse angeben, welche die entsprechende Schnittstelle gar nicht implementiert? Dann erhält der Benutzer bei der Ausführung des Codes einen Laufzeitfehler. Es muss also eine Möglichkeit bestehen, eine Klasse sozusagen zu fragen, ob sie eine bestimmte Schnittstelle unterstützt.

13.3.2 Testen auf eine bestimmte Schnittstelle

Wer mit C++ bereits etwas in die COM-Programmierung unter Windows hineingeschnuppert hat, dem wird jetzt `QueryInterface()` wie eine Leuchtreklame vor dem geistigen Auge erscheinen. Richtig getippt! Allerdings haben Sie sicher schon festgestellt, dass C# wesentlich eleganter und einfacher ist als C++ (oder mussten Sie sich in C# schon mit Sachen wie `AddRef()` und `Release()` herumschlagen?).



Delphi

Die Typoperatoren `is` und `as` funktionieren wie in Delphi, nur mit dem Unterschied, dass ein fehlgeschlagenes `as` keine Exception auslöst, sondern eine `null`-Referenz liefert. Etwas weniger dramatisch also …

In C# wird dazu der Typoperator `is` benutzt, mit dem festgestellt werden kann, ob ein Objekt eine bestimmte Schnittstelle ‚ist', d.h. diese Schnittstelle implementiert. Ergibt der Vergleich `true`, können Sie das Objekt gefahrlos überall dort einsetzen, wo die entsprechende Schnittstelle erwartet wird.

```
public class Testklasse
{
   public static void Main()
   {
      // instanz der klasse erstellen
      ImplKlasse obj = new ImplKlasse();

      if (obj is ILaufen)
      {
         // nutzen der schnittstelle
      }
   }
}
```

Listing 13.7: Testen auf eine bestimmte Schnittstelle

13.3.3 Typumwandlungen für Schnittstellen

Sie können ein Objekt auch in eine entsprechende Schnittstelle konvertieren, indem Sie eine Typumwandlung (*type cast*) durchführen. Auch hier bietet C# wieder sowohl den Anhängern der C/C++-Welt als auch für die Java/Delphi-Fraktion eine passende Variante an. Als Beispiel möchte ich meine Instanz `obj` der Klasse `ImplKlasse` in die Schnittstelle `ILaufen` konvertieren und dann einer Variablen dieses Typs zuweisen.

```
ILaufen il = ((ILaufen) m);
```

Listing 13.8: Die C++-Methode einer Typumwandlung

```
ILaufen il = (m as ILaufen);
```

Listing 13.9: C# folgt Java/Delphi-Methode bei der Typumwandlung

Abgesehen von der Syntax unterscheiden sich beide Möglichkeiten deutlich in ihrem Verhalten. Angenommen, das Objekt implementiert die gewünschte Schnittstelle nicht (Sie haben vielleicht vergessen, eine Codezeile vorher mit `is` zu testen), dann löst die erste Variante mit der C-artigen Typumwandlung eine Exception aus. Die zweite Variante mit dem Operator `as` liefert dagegen einen `null`-Wert zurück. So können Sie also nach der Zuweisung auch noch testen, ob die erhaltene Referenz `null` oder eine echte Referenz ist. Funktionieren werden in C# beide Varianten, wenn Sie auch nach Möglichkeit den `as`-Operator benutzen sollten.

13.4 Implementieren mehrerer Schnittstellen

Bisher habe ich in der Klasse nur eine Schnittstelle implementiert. Mit C# sind Sie aber nicht darauf begrenzt, sondern Sie können in einer Klasse mehr als eine Schnittstelle implementieren (wenn die Klasse auch ab einer gewissen Anzahl ziemlich unübersichtlich werden dürfte).

Sie erinnern sich noch an das Kapitel mit den Klassen und den vergessenen Meeressäugern? Ich hatte Ihnen doch versprochen, dass ich mich um Flipper kümmere …

Als Beispiel für die Implementierung mehrerer Schnittstellen möchte ich ebenfalls wieder auf eine analoge Situation in der Natur zurückgreifen, welche das Konzept sofort einleuchtend erscheinen lässt. Jedes Säugetier besitzt ein Gewicht, eine Augenfarbe und eine bestimmte Geschwindigkeit der Fortbewegung in seinem Medium. Die meisten Säugetiere können laufen, es gibt aber auch die Meeressäugetiere, die nicht mehr laufen, dafür aber schwimmen und tauchen können. Zusätzlich gibt es Säuger, welche sowohl laufen als auch schwimmen und tauchen können (der Mensch weniger elegant tauchen, der Seelöwe eher wacklig laufen, aber immerhin). Diese Zusammenhänge lassen sich ideal über Schnittstellen in einer objektorientierten Programmiersprache nachbilden.

13.4.1 Definitionen der Klassen und Schnittstellen

Dazu definiere ich zuerst die Basisklasse Saeuger:

```
class Saeuger
{
   private double fGeschwindigkeit;
   private double fGewicht;
   private string fName;
   private Color fAugenfarbe;

   public Saeuger()
   {
      fGewicht = 0;
      fGeschwindigkeit = 0;
      fAugenfarbe = Color.Black;
      // GUID als 'default name'
      fName = System.Guid.NewGuid().ToString();
   }

   public Saeuger(string name,
                  double gewicht,
                  double geschwindigkeit,
                  Color augenfarbe)
   {
      fGewicht = gewicht;
      fGeschwindigkeit = geschwindigkeit;
      fAugenfarbe = augenfarbe;
      fName = name;
   }
```

```
    public double Geschwindigkeit
    {
        get { return fGeschwindigkeit; }
        set { fGeschwindigkeit = value; }
    }

    public double Gewicht
    {
        get { return fGewicht; }
        set { fGewicht = value; }
    }

    public Color Augenfarbe
    {
        get { return fAugenfarbe; }
        set { fAugenfarbe = value; }
    }

    public string Name
    {
        get { return fName; }
        set { fName = value; }
    }
}
```

Listing 13.10: Die Basisklasse für alle Säugetiere

Das einzig Neue an dieser Klassendefinition dürfte eine Zeile im Standard-Konstruktor `Saeuger()` sein, mit dem der Inhalt des Feldes `fName` gesetzt wird. Nachdem kein Parameter übergeben wurde, bekommt diese Instanz einfach einen globalen, eindeutigen Bezeichner (eine GUID). Mehr Informationen dazu erhalten Sie in der Dokumentation der `struct System.Guid`.

Jetzt komme ich zur Definition der einzelnen Schnittstellen für Laufen und Schwimmen. Da das Laufen oder Schwimmen je nach Säugetier mit verschiedenen Körperteilen ausgeführt wird, habe ich dafür eine Aufzählung definiert. Außerdem werden beim Menschen die oberen und unteren Extremitäten noch dazu verschieden bezeichnet, daher handelt es sich um eine Flags-Aufzählung, so dass Arme und Beine kombinierbar sind. Mehr Informationen dazu finden Sie in Kapitel 9 ***Abzählen in C#-Enums*** und in Kapitel 10 ***Structs und Arrays***.

```
// die aufzaehlung fuer die extremitaeten
[Flags]
public enum Extremitaeten : byte
{
   Arme        = 0x01,
   Beine       = 0x02,
   Ruderbeine  = 0x04,
   Flossen     = 0x08,
   Fluegel     = 0x10
}

// das interface ISchwimmen für schwimmer/taucher
interface ISchwimmen
{
   void Schwimme();
   void Tauche();

   Extremitaeten Koerperteil();

   double Tauchtiefe
   {
      get;
   }
}

// das interface ILaufen fuer landratten
interface ILaufen
{
   void Laufe();
   Extremitaeten Koerperteil();

   double Schrittlaenge
   {
      get;
      set;
   }
}
```

Listing 13.11: Die zwei Schnittstellen

Als erste Implementierung und kleine Wiederholung eine Klasse Delphin, welche nur die eine Schnittstelle `ISchwimmen` implementiert.

```
class Delphin : Saeuger, ISchwimmen
{
   // standard-konstruktor
   public Delphin() : base()
   {
   }

   public Delphin(string name,
                  double gewicht,
                  double geschwindigkeit,
                  Color augenfarbe)
   : base(name, gewicht, geschwindigkeit, augenfarbe)
   {
   }

   public void Schwimme()
   {
      Console.WriteLine("ISchwimmen.Schwimme von " +
                        "Delphin {0}.", this.Name);
   }

   public void Tauche()
   {
      Console.WriteLine("ISchwimmen.Tauche von " +
                        "Delphin {0}.", this.Name);
   }

   public Extremitaeten Koerperteil()
   {
       return Extremitaeten.Flossen;
   }

   public double Tauchtiefe
   {
       get { return 450; }
   }
}
```

Listing 13.12: Die Klasse Delphin

13.4.2 Explizite Schnittstellenimplementierung

Für die Klasse Mensch ist bereits ein Problem absehbar. Beide Schnittstellen definieren eine Methode `Koerperteil()`, welche ein Mitglied der Aufzählung `Extremitaeten` zurückliefert. Damit ist die Signatur

beider Methoden gleich und ich kann diese normalerweise nicht in meiner Klasse unterbringen, ohne eine Fehlermeldung beim Übersetzen zu erhalten.

Die Lösung hierfür lautet explizite Schnittstellenimplementierung. Bisher haben wir die Schnittstellen dadurch implementiert, dass ein öffentliches (`public`) Mitglied der Klasse mit der für die Schnittstellen-Deklaration passenden Signatur erstellt wurde und sich die Schnittstelle dieses dann 'irgendwie selber gesucht' hat. Dieser Ansatz hat zwei Nachteile. Erstens ist dieses Mitglied (gleich, ob Methode, Eigenschaft oder Ereignis) dann auch Bestandteil der Klasse selbst, obwohl sie vielleicht nur im Kontext der Schnittstelle Sinn macht und zweitens lässt sich so eben keine zweite Schnittstelle implementieren, die eine gleichnamige Definition enthält.

Bei einer expliziten Schnittstellenimplementierung wird dem Namen des Klassenmitglieds einfach der Name der Schnittstelle vorangestellt. Dabei darf aber auch kein Modifikator wie `public` oder `private` mehr stehen, denn jetzt ist dieses Mitglied der Klasse nur für die Implementierung der Schnittstelle verfügbar und nicht mehr für die Klasse selbst. Damit besitzen wir wieder eine saubere Trennung zwischen Grundfunktionalität der Klasse und Implementierung der Schnittstelle.

Hier die entsprechend aufgebaute Klasse Mensch, bei der ich gleich alle Methoden und Eigenschaften der Schnittstellen explizit implementiert habe, nicht nur die Methode `Koerperteil()`.

```
class Mensch : Saeuger, ISchwimmen, ILaufen
{
   private double fSchrittlaenge;

   public Mensch() : base()
   {
   }

   public Mensch(string name,
                 double gewicht,
                 double geschwindigkeit,
                 Color augenfarbe)
   : base(name, gewicht, geschwindigkeit, augenfarbe)
```

```
{
}

void ISchwimmen.Schwimme()
{
   Console.WriteLine("ISchwimmen.Schwimme von " +
                     "Mensch {0}.", this.Name);
}

void ISchwimmen.Tauche()
{
   Console.WriteLine("ISchwimmen.Tauche von " +
                     "Mensch {0}.", this.Name);
}

Extremitaeten ISchwimmen.Koerperteil()
{
   return Extremitaeten.Arme | Extremitaeten.Beine;
}

Extremitaeten ILaufen.Koerperteil()
{
   return Extremitaeten.Beine;
}

void ILaufen.Laufe()
{
   Console.WriteLine("ILaufen.Laufe von Mensch.");
}

double ILaufen.Schrittlaenge
{
   get
   {
      return fSchrittlaenge;
   }

   set
   {
      fSchrittlaenge = value;
   }
}

double ISchwimmen.Tauchtiefe
{
```

```
      get { return 130; }
   }
}
```

Listing 13.13: Die Klasse Mensch

Alles, was noch fehlt, ist eine kleine Testklasse, damit wir die Funktion der Klasse auch ausprobieren können, dann ist der Code vollständig (Sie finden das komplette Beispiel auf der CD zum Buch).

```
class Testklasse
{
   static void Main(string[] args)
   {
      Mensch m = new Mensch("Franz", 75.0,
                            26.0, Color.Brown);
      Delphin d = new Delphin("Flipper", 200,
                            45, Color.Gray);

      ILaufen il = (new Mensch()) as ILaufen;

      // liefert false, da die Klasse Delphin
      // die schnittstelle ILaufen nicht implementiert
      if (d is ILaufen)
         ((ILaufen)d).Laufe();
      else
         Console.WriteLine(d.Name + " läuft nicht.");

      if (m is ILaufen)
      {
         ILaufen intf = (m as ILaufen);
         intf.Laufe();
         Console.WriteLine(m.Name + " benutzt " +
                     intf.Koerperteil().ToString() );
      }
      else
         Console.WriteLine(d.Name + " läuft nicht.");

      if (m is ISchwimmen)
      {
         ISchwimmen intf = (m as ISchwimmen);
         intf.Schwimme();
         Console.WriteLine(m.Name + " benutzt " +
                     intf.Koerperteil().ToString() );
      }
```

```
      else
         Console.WriteLine(d.Name + " schwimmt nicht.");

      Console.ReadLine();

   }
}
```

Listing 13.14: Die Testklasse für die Implementierung der Schnittstellen

13.5 Zusammenfassung

Schnittstellen bieten einen Ansatz in der objektorientierten Programmierung, bei dem die Kopplung zwischen einzelnen Klassen wesentlich geringer ist als bei der üblichen klassenbasierten Vererbung. Eine Schnittstelle als Vertrag zwischen zwei Softwarekomponenten garantiert, dass die aufgeführte Funktionalität auch komplett vorhanden ist (der Compiler erzeugt einen Fehler, wenn Sie vergessen, ein Mitglied einer Schnittstelle zu programmieren).

Als rein abstrakte Beschreibungen von Funktionalität können Sie daher auch keine Instanz einer Schnittstelle erzeugen, sondern nur Instanzen von Klassen, welche eine oder mehrere Schnittstellen implementieren.

Bei der Implementierung einer oder mehrerer Schnittstellen in einer Klasse können Sie mit Hilfe der expliziten Schnittstellenimplementierung eine klare Trennung zwischen Funktionalität der Klasse und Implementierung der Schnittstelle vornehmen. Damit ist dann auch die Implementierung gleichnamiger Mitglieder aus mehr als einer Schnittstelle möglich.

Mit Hilfe der beiden Operatoren `is` und `as` können Sie sowohl testen, ob ein Objekt eine bestimmte Schnittstelle implementiert als auch mit einer sicheren Typumwandlung Objekte als Referenz zu dieser Schnittstelle konvertieren.

13.6 Übungen

Ü

1. Entwerfen Sie eine Schnittstelle für eine Gerätesteuerung mit den Methoden An, Aus, Vorwärmen (Parameter double), den zwei Eigenschaften Werkstücke und Brennzeit sowie dem öffentlichen `double`-Feld Abkuehlzeit.
2. Wie lassen sich zwei Methoden gleichen Namens aus zwei verschiedenen Schnittstellen in einer Klasse implementieren?
3. Wo liegt der Unterscheid zwischen einer abstrakten Klasse und einer Schnittstellendefinition?
4. Angenommen, eine Klasse `abc` implementiert eine Schnittstelle `xyz`. Von `abc` leiten Sie nun eine neue Klasse ab. Muss diese neue Klasse ebenfalls wieder die Methoden der Schnittstellen implementieren?

14 Ein- und Ausgabe

Aufgrund der Herkunft von C# besitzt die Sprache selbst keine Schlüsselwörter für die Ein- oder Ausgabe von Daten, wie dies bei Visual Basic u.a. mit `Open` und `Close` der Fall ist. Diese Erbschaft von C sorgt für ein flexibles und nur durch die Möglichkeiten der Laufzeitumgebung begrenztes Repertoire an Möglichkeiten, ohne Einschränkung durch die Syntax der Sprache. C# benutzt hier die entsprechenden Klassen der .NET-Laufzeitumgebung. Insofern ist dieses Kapitel nicht C#-spezifisch, sondern gilt für alle Programmiersprachen, die mit der .NET-Laufzeitumgebung arbeiten. Da Sie aber als Umsteiger sicher zuerst einen Überblick über die entsprechenden Konzepte suchen, hier also die Möglichkeiten zum Lesen und Schreiben von Dateien.

14.1 Dateioperationen

Für die Arbeit mit den Objekten des Dateisystems stellt die .NET-Laufzeitumgebung entsprechende Klassen zur Verfügung, die entweder von `Object` (`File`, `Directory` und `Path`) oder von `MarshallByRefObject` (`FileInfo` und `DirectoryInfo`) abgeleitet werden.

Die große Namensverwirrung bei Microsoft: Vor langer, langer Zeit (also DOS und Windows 3.x) hießen Objekte zur Verwaltung eines hierarchischen Dateisystems noch Verzeichnisse (*directory*). Irgendwann beschloss Microsoft, dass dieser Begriff für den Endanwender zu technisch klang und ersetzte ihn durch den aus der Papierbürowelt bekannten Ordner (*folder*). In allen anderen Betriebssystemen, die mir bekannt sind (sowie der dazugehörigen Literatur) spricht man dagegen weiter vom Verzeichnis (*directory*). Nachdem Microsoft mit .NET anscheinend wirklich einen plattformunabhängigen Weg gehen möchte, hat sich das .NET-Team dazu entschlossen, die Klassen für den Zugiff auf Verzeichnisse wieder `DirectoryInfo` und `Directory` zu nennen. Womit alle Plattformen wieder glücklich vereint wären, nur Entwickler und Benutzer damit komplett verwirrt sind, wie denn nun diese Dinger genannt werden sollen …

Damit existieren für den Zugriff auf Verzeichnisse und Dateien zwei Klassen: `File` und `FileInfo` sowie `Directory` und `DirectoryInfo`. Worin liegt der Unterscheid, welche Klasse sollten Sie benutzen?

Die Antwort hängt davon ab, was Sie mit der Datei oder dem Verzeichnis (Sie wissen schon, dem früheren Ordner) vorhaben. Alle Methoden der Klassen `File` und `Directory` sind komplett statisch und werden nie instantiiert. Bei einem einzelnen Zugriff ergibt sich dadurch eine etwas kürzere Ausführungszeit Ihres Codes, da sich die .NET-Laufzeitumgebung die Erzeugung des Objekts im Speicher sparen kann. Sobald Sie mit einer Datei aber beispielsweise mehrere Operationen nacheinander ausführen, ändert sich das Bild: jetzt bietet `FileInfo` die deutlich bessere Leistung.

Bei den beiden Info-Klassen werden nämlich Instanzen für die Dateisystemobjekte angelegt, die alle nötigen Informationen zwischenspeichern. So wird bei der Benutzung der Klasse `File` beispielsweise bei jedem Zugriff die komplette Sicherheitsüberprüfung (Rechte, Zugriff, Codesicherheit) durchgeführt, bei einer Instanz der Klasse `FileInfo` nur beim Erzeugen des Objektes. Daher sollten Sie für Mehrfachoperationen immer die entsprechende Info-Klasse benutzen. Benötigen Sie dagegen den Dateinamen nur für den Konstruktor einer `StreamReader`-Instanz, können Sie genausogut die Klasse `File` benutzen.

Die Klasse `Path` nimmt eine Sonderstellung ein und dient in der Hauptsache der Bearbeitung von Dateipfaden und deren Namen. Hier stehen nur die statischen Methoden zur Verfügung. Die Bezeichnung 'Pfad' bedeutet eines der folgenden Elemene aus der Liste:

- Absoluter Laufwerkspfad (z.B. @»C:\programme\dingsbums\programm.exe«)
- Absoluter Verzeichnispfad (z.B. @»C:\programme«)
- Relative Pfadangabe mit oder ohne Datei (z.B. @»images\bild1.gif«)
- UNC-Pfad mit oder ohne Datei (z.B. @»\\Server14\Freigabename«)



Denken Sie daran, dass Sie einen Verbatim-String benutzen müssen, um Pfadangaben mit nur einem Backslash schreiben zu können (wie im Beispiel oben durch ein vorangestelltes »@« gekennzeichnet). Andernfalls müssten Sie, analog zu C++, den letzten Pfad als »`\\\\Server14\\Freigabename`« schreiben.

14.1.1 Die Klasse Path

Diese Klasse dient der einfacheren Bearbeitung von Pfadnamen und Vergleichen bei Dateinamen. So können Sie beispielsweise auf einfache Art und Weise feststellen, ob eine Datei eine bestimmte Dateinamenerweiterung besitzt:

```
If (".exe" == Path.GetExtension(dateivar))
{
   // ausführbare datei
}
```

Wie aus dem Codefragment ersichtlich, liefert die Methode `GetExtension` den Punkt mit. Wird keine Erweiterung gefunden oder ein leerer String als Parameter übergeben, liefert die Methode entweder eine Null-Referenz oder den speziellen String `Empty`, der eine symbolische vordefinierte Nur-Lese-Variable für einen Leerstring darstellt.

Die folgenden Tabellen liefern die wichtigsten Methoden und Eigenschaften der Klasse:

Eigenschaft	Beschreibung
DirectorySeparatorChar	Das Trennzeichen für Verzeichnisse und Dateinamen, bei Windows der Rückwärtsschrägstrich (Backslash, »\«), bei Unix der Schrägstrich (»/«) und bei MacOS der Doppelpunkt (»:«)
AltDirectorySeparator Char	Gibt ein alternatives Trennzeichen für Verzeichnisse und Dateinamen an, bei Windows und Mac ist dies der Schrägstrich (»/«), bei Unix der Rückwärtsschrägstrich (»\«).
InvalidPathChars	Plattformabhängig, liefert eine Auflistung der Zeichen, die in einem Dateipfad nicht vorkommen dürfen.

Tabelle 14.1: Eigenschaften der Klasse Path

Eigenschaft	Beschreibung
PathSeparator	Trennt eine Liste von Pfadangaben. Da die .NET-Laufzeitumgebung derzeit nur auf Windows implementiert ist, liefert diese Eigenschaft derzeit nur das Semikolon (»;«).
VolumeSeparatorChar	Liefert das Zeichen, mit dem Laufwerke oder log. Platten (*volumes*) vom Pfad getrennt werden, bei Windows also wie beim Mac der Doppelpunkt (»:«), bei Unix-Systemen der Schrägstrich (»/«).

Tabelle 14.1: Eigenschaften der Klasse Path (Forts.)

Methode	Beschreibung
ChangeExtension	Ändert die Erweiterung einer Datei auf den angegebenen Inhalt.
Combine	Kombiniert zwei Pfadangaben zu einer. **Vorsicht: Die Fehlerprüfung ist nur sehr rudimentär, effektiv Pfad1 + Pfad2 …**
GetDirectoryName	Liefert den Verzeichnispfad einer Angabe.
GetExtension	Liefert die Erweiterung der Datei im angegebenen Pfadnamen.
GetFileName	Liefert aus einem Pfadnamen den Namen und die Erweiterung der Datei.
GetFileNameWithoutExtension	Liefert nur den Dateinamen ohne Erweiterung aus einem Pfadnamen.
GetFullPath	Erweitert den angegebenen Pfad zu einem vollständig qualifizierten Pfadnamen, aus der Angabe »bilder\sonne.jpg« wird bei einem aktuellen Verzeichnis von »c:\galerie« also die Rückgabe »c:\galerie\bilder\sonne.jpg«. **Vorsicht: Es erfolgt keine Überprüfung, ob die Datei vorhanden ist!**
GetPathRoot	Liefert das Stammverzeichnis des angegebenen Pfadnamens.

Tabelle 14.2: Methoden der Klasse Path

Methode	Beschreibung
GetTempFileName	Sehr praktisch: Liefert einen eindeutigen Namen für eine temporäre Datei. Wird kein Verzeichnis angegeben, wird das Betriebssystemverzeichnis für temporäre Dateien benutzt. **Vorsicht: Diese Methode legt die Datei mit einer Länge von 0 Byte an, Sie benötigen also die entsprechenden Berechtigungen im angegebenen Verzeichnis.**
GetTempPath	Liefert den Pfad für das Verzeichnis für temporäre Dateien.
HasExtension	Stellt fest, ob ein Pfadname über eine Dateinamenerweiterung verfügt.
IsPathRooted	Stellt fest, ob ein Pfadname über die Angabe eines Stammverzeichnisses verfügt.

Tabelle 14.2: Methoden der Klasse Path (Forts.)



Achten Sie bei allen Operationen mit Dateien darauf, dass es außerhalb der Windows-Welt durchaus Betriebssysteme gibt, die Groß- und Kleinschreibung bei Dateinamen unterscheiden. Windows NT bzw. seine Nachfolger 2000 und XP behalten die Schreibweise bereits bei und falls die .NET-Laufzeitumgebung einmal auf andere Betriebssysteme portiert werden sollte (Stichwort Mono-Projekt), dann kann es für Sie nur von Vorteil sein, wenn Sie diese mögliche Fehlerquelle bereits beim Dateizugriff eliminiert haben.

14.1.2 File und Directory

Diese beiden Klassen (und damit auch die beiden Info-Klassen mit den entsprechenden Instanzmethoden) bieten eine ganze Reihe von Eigenschaften und Methoden für die Bearbeitung von Dateien, Verzeichnissen und der Erzeugung von Streams aus Dateien. Anstelle einer trockenen Aufzählung, welche Sie sich einfacher in der Online-Dokumentation der .NET-Entwicklungsumgebung ansehen können, hier ein kleines Programm zur Verdeutlichung einiger Methoden und Eigenschaften.

```
// Goto C# -- Armin Hanisch
// abfragen von dateiinformationen

using System;
```

```
using System.IO;

namespace Dateioperationen
{
   class DateiInfo
   {
      static void Main(string[] args)
      {
         string[] dateien =
             Directory.GetFiles(@"c:\","*.sys");

         foreach(string datei in dateien)
         {
            FileInfo fi = new FileInfo(datei);
            Console.WriteLine(
                   "Datei {0}\n  ist {1} Byte " +
                   "groß und wurde erstellt am {2}",
                   datei,
                   fi.Length,
                   fi.CreationTime);
         }
         Console.ReadLine();
      }
   }
}
```

Listing 14.1: Suchen nach Dateien und Abfragen von Dateiinformationen

Nachdem mehr als einmal auf die Informationen der Datei zugegriffen wird, benutze ich eine Instanz der Klasse `FileInfo`, vor allem auch, da die statischen Methoden der Klasse `File` keine Möglichkeit bieten, auf die Größe der Datei zuzugreifen, ohne erst einen Stream zu öffnen und dessen Länge zu lesen. Wollen Sie innerhalb eines Verzeichnisses nach allen Dateien suchen, lassen Sie bei der Methode `GetFiles` einfach den zweiten Parameter weg.

Auch hier noch einmal die Anmerkung, dass die obige Suchmaske zwar auch eine Datei MSDOS.SYS (in Großbuchstaben) findet, allerdings nur, solange das unter der .NET-Laufzeitumgebung liegende Betriebssystem nicht zwischen Groß- und Kleinschreibung unterscheidet. Bei einem Vergleich der gelieferten Daten mit Hilfe des für die Klasse `String` überladenenen Operators == müssen Sie allerdings sehr wohl die Daten und die Variable beide einheitlich in Großbuchstaben

oder Kleinbuchstaben konvertieren, um alle Dateien in der Abfrage zu behandeln. Der bessere Weg ist hier eindeutig die Verwendung der für diesen Zweck vorgesehenen Methoden wie `String.Compare`, die einen Parameter IgnoreCase vom Typ bool bieten, mit dem unabhängig von Groß- und Kleinschreibung geprüft werden kann. Mehr dazu finden Sie in Kapitel 15 ***Strings***.

Um die einzelnen Unterschiede zwischen den rein statischen Klassen und den Instanzklassen noch einmal zu verdeutlichen, finden Sie eine Gegenüberstellung der beiden Klassen FileInfo und File (analog gilt diese Tabelle auch für DirectoryInfo und Directory). So finden Sie schnell die Unterschiede und können die jeweils am besten passende Klassen benutzen.

Klasse FileInfo	Klasse File
Methode AppendText	Methode AppendText
Methode CopyTo mit Ziel	Methode Copy mit Quelle und Ziel
Methode Create	Methode Create
Methode CreateText	Methode CreateText
Methode Delete	Methode Delete
Eigenschaft Attributes	GetAttributes / SetAttributes
Methode MoveTo mit Ziel	Methode Move mit Quelle und Ziel
Methode Open	Methode Open
Methode OpenRead	Methode OpenRead
Methode OpenText	Methode OpenText
Methode OpenWrite	Methode OpenWrite
Methode Refresh	(keine Entsprechung)
Methode ToString	(keine Entsprechung)
Eigenschaft CreationTime	GetCreationTime / SetCreationTime
Directory	(keine Entsprechung)
DirectoryName	(keine Entsprechung)
Eigenschaft Exists	Methode Exists

Tabelle 14.3: Unterschiede der Klassen File und FileInfo

Klasse FileInfo	Klasse File
Eigenschaft Extension	Über Path.GetExtension möglich
FullName	Über Path.GetFullname möglich
Eigenschaft LastAccessTime	GetLastAccessTime SetLastAccessTime
Eigenschaft LastWriteTime	GetLastWriteTime SetLastWriteTime
Eigenschaft Length	(keine Entsprechung)
Eigenschaft Name	Wird als Parameter angegeben

Tabelle 14.3: Unterschiede der Klassen File und FileInfo (Forts.)

14.2 Das Stream-Konzept

Eingabe und Ausgabe in der .NET-Welt laufen nicht wie bisher bei vielen Betriebssystemaufrufen unter Windows ab. Die .NET-Laufzeitumgebung setzt konsequent auf das Konzept der Streams. Ein Stream entspricht von der Funktion her genau der Übersetzung des Wortes: ein Strom von Bytes, der in eine bestimmte Richtung fließt, also gelesen oder geschrieben werden kann. Damit ist die tatsächliche Implementierung eines Streams unabhängig von einer Datei oder einem anderen Datenträger, Streams lassen sich daher auch zur Zwischenspeicherung von Daten einsetzen oder der Datenstrom fließt von einem Rechner über ein TCP/IP-Netzwerk zu einem anderen Rechner. Die .NET-Laufzeitumgebung stellt als Ausgangsbasis verschiedene Streamklassen zur Verfügung, die alle verschiedene physikalische Ausprägungen eines Streams implementieren.

14.2.1 Streamklassen

Die Klasse `Stream` bildet die Basis für alle I/O-Operationen und beschreibt eine grundlegende Transportmöglichkeit für Daten von oder zu einem persistenten Speicher. Von dieser sehr allgemeinen Klasse werden in der CLR dann spezielle Klassen abgeleitet, die für das Lesen aus einer Datei, Schreiben in eine Datei, jeweils Test oder Binärdatei und alle die anderen Möglichkeiten der Kommunikation mit einem Speichermedium vorhanden sind.

Abgeleitet von der Basisklasse Stream finden sich in der .NET-Laufzeitumgebung die folgenden Klassen:

Klasse	Beschreibung
BufferedStream	Die Klasse BufferedStream dient der Beschleunigung von Schreib- und Lesevorgängen durch das Einrichten eines Puffers. Dieser Puffer besitzt eine Standardgröße von 4 KByte.
FileStream	Ein FileStream dient dem Lesen und Schreiben von Daten aus einem Dateispeicher, gleich ob physikalische lokale Datei, UNC-Pfad, Standardeingabe, Standardausgabe oder named pipe. Aus Gründen der Performance verwendet ein FileStream automatisch Puffer. Die Nutzung der Lese- und Schreibroutinen kann entweder synchron oder asynchron benutzt werden. Die Klassen File, Directory und Path dienen meist als Hilfsklassen zur Erzeugung von FileStreams.
MemoryStream	MemoryStreams benutzen als Speicher für die Daten Arbeitsspeicher im Computer und nicht einen persistenten Datenträger. Damit sind solche MemoryStreams ideal für das Zwischenspeichern von Daten. Eine Instanz kann entweder mit einem Array von Bytes oder mit einem ‚leeren' Array initialisiert werden. Nur leer erzeugte Instanzen von MemoryStreams erlauben auch das Lesen und die nachträgliche Änderung der Größe des Streams.
NetworkStream	Ein NetworkStream implementiert eine streamorientierte Ein- und Ausgabe für die Kommunikation über Sockets. Hierbei sind sowohl synchrone als auch asynchrone Aufrufe möglich. Aufgrund der Natur einer Netzwerkübertragung kann in einem NetworkStream nicht positioniert werden. Der Aufruf von Seek führt also zu einem Fehler.
CryptoStream	Die CLR benutzt für Verschlüsselungsroutinen einen streamorientierten Ansatz. Alle Verschlüsselungsobjekte, die einen CryptoStream implementieren, können direkt Stream-Objekte benutzen, so dass die Ausgabe eines Objektes ohne eine Zwischenspeicherung die Eingabe eines anderen Objektes ergibt.

Tabelle 14.4: Die Streamklasse in der .NET-Laufzeitumgebung

Für den Zugriff auf eine Datei können Sie mit den Methoden `Open`, `OpenRead`, `OpenText` und `OpenWrite` der Klassen `File` bzw. `FileInfo` aus dem Dateiobjekt schnell einen passenden Stream mit den Standardeinstellungen erzeugen.

Die Zeile

```
StreamReader sr = File("datei.txt").OpenText();
```

liefert eine auf Nur-Lesen eingestellte `StreamReader`-Instanz zurück, welche die Zeicheneinstellung UTF-8 benutzt.

14.2.2 Lesen und Schreiben einer Textdatei

Um eine Textdatei zu lesen, wird nur sehr wenig Code benötigt, da die .NET-Laufzeitumgebung mit den Streamfunktionen das Lesen und Schreiben von Text sehr einfach macht. Prinzipiell ist der Unterschied zur Ausgabe auf der Konsole nur das Erzeugen des Streams vor der Benutzung von `ReadLine()` oder `WriteLine()`.

```
// Goto C# -- Armin Hanisch
// Lesen und Schreiben von Dateien mit FileStreams

using System;
using System.IO;

namespace InputOutput
{
   class Testklasse
   {
      static void DumpFile(string fname)
      {
         StreamReader sr = new StreamReader(fname);

         string zeile;
         int nr = 1;

         do
         {
            zeile = sr.ReadLine();
            Console.WriteLine("{0}: {1}", nr++, zeile);
         } while (zeile != null);

         sr.Close();
      }

      static void Main(string[] args)
      {
         DumpFile(@"h:\samplecode\sprachen.txt");
```

```
            Console.ReadLine();
        }

    }
}
```

Listing 14.2: Dateizugriffe über einen StreamReader

Vorteilhafterweise löst die Methode `ReadLine()` der Klasse StreamReader keine Exception aus, wenn aus der Datei nichts mehr gelesen werden kann oder die Datei von vornherein leer ist. Stattdessen wird für die Stringvariable eine Null-Referenz zurückgegeben. Diese dient dann wie im Code als Abbruchkriterium für die Schleife.

14.2.3 Codierung von Dateien

Im vorliegenden Beispiel habe ich den Konstruktor der Klasse StreamReader mit einem einzelnen Parameter benutzt. Alternativ ist aber die Angabe eines Encodings möglich. Dies ist eine Festlegung, wie die Daten in die Textdatei geschrieben werden sollen. Sie mögen nun einwenden: »Text ist Text, was soll da schon anders sein?«. Allerdings schafft ein Blick auf die Klasse `System.Text.Encoding` gleich mehr Klarheit: Dies ist eine abstrakte Basisklasse für davon abgeleitete Klassen, welche dann beim Lesen und Schreiben die eigentliche Codierung der Zeichen erledigen. Die .NET-Laufzeitumgebung benutzt als Standard die Unicode-Kodierung UTF-8. Möchten Sie nun eine Datei zum Austausch mit einem Altsystem als reine ASCII-Datei speichern, geben Sie dies beim Erzeugen des StreamWriters mit an:

```
FileStream fs = new FileStream("ausgabe.txt",
                               FileMode.CreateNew,
                               FileAccess.Write,
                               FileShare.ReadWrite);
StreamWriter sr = new StreamWriter(fs, Encoding.ASCII);
```

Die Erstellung einer `StreamWriter`- oder `StreamReader`-Instanz über einen darunterliegenden FileStream hat im Vergleich zu dem letzten Beispiel den Vorteil, dass Sie die Attribute der Datei feiner steuern können. Dateimodus und gleichzeitiger Zugriff können hier eingestellt werden. Das Schließen mit der Methode `Close` schließt auch gleichzeitig den darunterliegenden Stream.

Nun können Sie mit den Methoden der Klasse StreamWriter in die Datei schreiben:

```
sw.WriteLine("Ein ue = ü");
sw.WriteLine("Ein oe = ö");
sw.WriteLine("Ein ae = ä");
sw.WriteLine("Ätsch, völlig überflüssige Umlaute!");
sw.Close();
```

Nach dem Schließen der Datei enthält diese den folgenden Text:

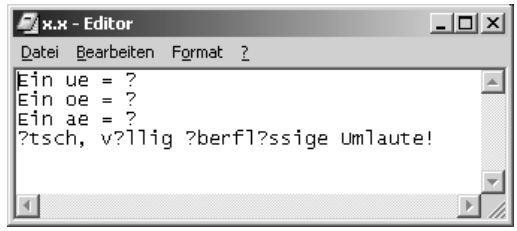


Abbildung 14.1: Der Inhalt einer Datei mit ASCII-Encoding

Wie Sie unschwer erkennen können, hat der Encoder für die Klasse ASCII alle Umlaute (die ja einen Bytewert von mehr als 127 besitzen und damit nicht im ASCII-Standard enthalten sind) durch ein Fragezeichen ersetzt. Auf diese Weise ist sichergestellt, dass die erzeugte Datei dem angegebenen Code-Standard entspricht, wenn auch die Lesbarkeit leidet.

Die .NET-Laufzeitumgebung enthält Klassen für die Ausgabe in folgenden Code-Standards:

- ASCII
- Unicode
- BigEndianUnicode
- UTF-7
- UTF-8

14.2.4 Peek – 'Nur mal einen Blick drauf werfen'

Eine andere Methode, um festzustellen, ob aus einem Stream noch gelesen werden kann, ist die Verwendung der Methode Peek (bzw. PeekChar bei einem BinaryReader). Hier wird nicht aus dem Stream gelesen, sondern nur 'geguckt', was das nächste Zeichen im Stream denn wäre. Damit wird der Lesezeiger nicht bewegt und so kann ebenfalls geprüft werden, ob das Ende eines Streams erreicht ist, da diese Methode dann in allen implementierenden Klassen den Wert -1 liefert.

Da das Laufzeitsystem eine ganze Palette an Klassen für die Ein- und Ausgabe zur Verfügung stellt, lässt sich der Code mit der gleichen Funktionalität auch anders formulieren. Für die Erzeugung eines StreamReaders wird nicht unbedingt vorher ein FileStream benötigt. Alternativ könnten Sie auch die Methode OpenText() der Klasse File verwenden:

```
StreamReader sr2 = File.OpenText("dateiname.txt");
```

Natürlich bringt auch die auf das Lesen von Textdateien spezialisierte Klasse StreamReader selbst eine Methode mit:

```
StreamReader sr3 = new StreamReader("dateiname.txt");
```



Welchen Ansatz Sie benutzen, hängt von der Problemstellung ab. Wenn Sie mit dem Stream nichts mehr vorhaben und nur den Pfad für die Datei benötigen, ist der Einsatz des StreamReader-Konstruktors mit Angabe eines Pfades und evtl. einer Codierung (z.B. ASCII oder UTF-8) natürlich sinnvoller.

Abgesehen davon existieren in der .NET-Laufzeitumgebung noch Unterschiede. So sind beispielsweise alle Methoden der Klasse File statisch und bei jedem Auruf einer Methode erfolgt eine Sicherheitsüberprüfung des Codes durch die CLR. Falls Sie also mit der Datei mehr vorhaben, empfiehlt sich der Einsatz der Klasse `FileInfo`, welche die gleichen Methoden bietet, diesmal allerdings als Instanzmethoden und damit ohne die Überprüfung bei jedem Methodenaufruf.

14.2.5 Lesen und Schreiben von Binärdateien

Für das Lesen und Schreiben von binären Dateien ließe sich natürlich der `FileStream` direkt verwenden, allerdings ist die Methode `Read` mit den Parametern `offset`, `count` und der Angabe der Daten als offenes Bytearray nicht unbedingt ideal. Daher existieren analog zu den Klassen `StreamReader` und `StreamWriter` auch die beiden Klassen `BinaryReader` und `BinaryWriter`, die für binäre Daten verwendet werden.

Das folgende Codefragment legt eine binäre Datendatei mit vier Integerwerten an und benutzt dazu die 18-fach (!) überladene Methode `Write` der Klasse `BinaryWriter`. Beim Lesen einer Datei haben Sie es dann leichter. Analog den Zugriffsmethoden der Windows-API für die Registry hat Microsoft hier für jeden Datentyp eine eigene Methode (von `ReadBoolean` bis `ReadULong`) gebaut.

```
static void CreateBinaryFile()
{
   FileStream fs = new FileStream("binaer.dat",
                   FileMode.OpenOrCreate, FileAccess.Write);
   BinaryWriter bin = new BinaryWriter(fs);

   int x;
   bin.Seek(0, SeekOrigin.Begin);
   x = 257;    bin.Write(x);
   x = -1;     bin.Write(x);
   x = 20560;  bin.Write(x);
   x = 0;      bin.Write(x);
   bin.Flush();
   bin.Close();
}
```

Listing 14.3: Methodencode zum Schreiben von Binärdaten in eine Datei

Das Ergebnis ist eine 16 Byte große Datei, deren Inhalt als Hexdump Abbildung 14.2 zeigt.

Da ein Integer (derzeit noch) 32 Bit breit ist, ergeben sich die insgesamt 16 Byte der Datei aus den vier Schreiboperationen. Wenn Sie schon mal Assembler programmiert haben oder das eine oder andere Mal auf Fehlersuche in diversen Dateien waren, erkennen Sie die Binärdarstellung der vier Integerwerte aus dem Code leicht wieder.

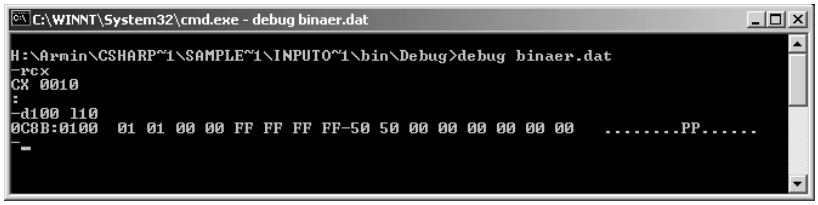


Abbildung 14.2: Der Hexdump der Binärdatei

14.2.6 Stream-Eigenschaften

Abhängig von der Implementierung und dem Modus beim Anlegen des Streams sind verschiedene Operationen nicht möglich. So können Sie bei einem `NetworkStream`, der über eine Netzwerkverbindung Daten liefert, aus naheliegenden Gründen kein `Seek` durchführen (wie schön wäre es, Werbung in Internet-Seiten einfach per Seek auszublenden …). Ein Stream, der als `TextReader` zum Lesen einer Textdatei erzeugt wurde, kann natürlich nicht geschrieben werden und so weiter. Mit Hilfe der folgenden Eigenschaften können Sie überprüfen, welche Möglichkeiten der Stream jeweils unterstützt.

- `CanRead` – liefert `true`, falls aus dem Stream gelesen werden kann.
- `CanWrite` – liefert `true`, falls in den Stream geschrieben werden kann.
- `CanSeek` – liefert `true`, wenn der Stream Positionierungen über Seek unterstützt.
- `DataAvailable` – liefert `true`, falls aktuell Daten aus dem Stream gelesen werden können.
- `Length` – liefert die Länge des Streams in Byte.
- `Position` – liefert die Position im Datenstrom, an der sich der Lese- bzw. Schreibzeiger befindet.

Diese Eigenschaften gelten für alle Stream-Klassen, die in diesem Abschnitt besprochen wurden. Weitere Eigenschaften und Methoden zu den einzelnen Stream-Klassen finden Sie in der Online-Dokumentation der .NET-Laufzeitumgebung.

14.3 Lesen und Schreiben von XML-Dateien

Über das Thema XML und .NET ließe sich allein ein ganzes Buch schreiben und Sie werden an verschiedenen Stellen dieses Buches einzelne Abschnitte finden, die das Thema XML jeweils mit den dort auftretenden Aufgabenstellungen behandeln. Daher werde ich an dieser Stelle nur kurz vorstellen, wie Sie XML-Dateien lesen und schreiben können. Andere Themen, wie beispielsweise die Serialisierung von Objekten in XML-Dateien werden bei den dazu gehörigen Kapiteln behandelt. Die entsprechenden Klassen finden Sie im Namensraum `System.Xml`, daher muss dieser zu Beginn der Datei mit `using` importiert werden.

Bei allen XML-erfahrenen Lesern möchte ich mich bereits an dieser Stelle entschuldigen, wenn hier eine ganze Reihe von Dingen vereinfacht dargestellt wird oder zu kurz kommt. Da die XML-Fähigkeiten der .NET-Laufzeitumgebung aber zu den grundlegenden Klassen gehören und die Möglichkeiten weit über das hinausgehen, was an dieser Stelle im Rahmen eines C#-Buchs gezeigt werden kann, sehen Sie die folgenden Abschnitte einfach als kleinen Streifzug an. Für alle, die tiefer in die Materie einsteigen wollen, sei der Hinweis erlaubt, dass eine sprachunabhängige ».NET-Referenz« bereits in Planung ist.

14.3.1 Zugriffsarten

Die .NET-Laufzeitumgebung unterstützt verschiedene Arten des Zugriffs auf XML-Dateien. Die beiden verschiedenen Ansätze sind einmal das Document Object Model des W3C, welches letztendlich dem Aufbau einer baumartigen Struktur aus XML-Knoten im Speicher entspricht. Dieser Ansatz hat den Vorteil, stets das gesamte Dokument im Zugriff zu haben und 'live' auf Änderungen der Struktur reagieren zu können. Nachteil ist aber eindeutig der entsprechend große Bedarf an Arbeitsspeicher bei komplexeren Dokumenten. Ein Beispiel dafür bietet der Ressourcenhunger des BizTalk-Servers.

Der andere Ansatz ist eine Art Streaming der XML-Dokumente, eine Methode, die bereits seit längerem beipielsweise vom SAX-Protokoll verfolgt wird. Hier sind die Anforderungen an die Ressourcen wesentlich geringer, allerdings steht auch nicht immer das komplette Dokument als Baum zur Verfügung und die Daten können nur gelesen wer-

den. Falls Sie also innerhalb des Dokumentes navigieren müssen oder einen Schreibzugriff benötigen, bleibt Ihnen nur der DOM-Ansatz. In allen anderen Fällen bietet der `XmlReader` und die davon abgeleitete Klasse `XmlTextReader` eine schnelle Möglichkeit, Daten im XML-Format einzulesen.

14.3.2 Die Beispieldatei

Alle Beispiele benutzen die Datei books.xml, die Sie auch in den Beispielen der .NET-Entwicklungsumgebungen und auf der CD zu diesem Buch finden. Nahezu alle anderen Bücher nutzen ebenfalls diese Datei, so dass der Transfer von Beispielen und das Weiterbauen relativ leicht fallen sollten. Diese Datei books.xml stellt einen Ausschnitt aus dem Sortiment eines fiktiven Buchhandels dar und speichert Informationen über Autoren, Titel, Preise und andere Buchdaten. Hier finden Sie den Inhalt der Datei:

```
<?xml version='1.0' ?>
<bookstore>
  <book genre="autobiography"
        publicationdate="1981" ISBN="1-861003-11-0">
    <title>The Autobiography of Benjamin
           Franklin</title>
    <author>
      <first-name>Benjamin</first-name>
      <last-name>Franklin</last-name>
    </author>
    <price>8.99</price>
  </book>
  <book genre="novel" publicationdate="1967"
        ISBN="0-201-63361-2">
    <title>The Confidence Man</title>
    <author>
      <first-name>Herman</first-name>
      <last-name>Melville</last-name>
    </author>
    <price>11.99</price>
  </book>
  <book genre="philosophy" publicationdate="1991" ISBN="1-
861001-57-6">
    <title>The Gorgias</title>
    <author>
```

```
      <name>Plato</name>
    </author>
    <price>9.99</price>
  </book>
</bookstore>
```

Listing 14.4: Die Beispieldatei books.xml für alle nachfolgenden Beispiele

14.3.3 XML-Dateien laden und anzeigen

Im folgenden Beispiel soll eine kleine XML-Datei gelesen und der Inhalt in halbwegs strukturierter Form in einer Windows Forms Anwendung angezeigt werden. Zum Lesen der XML-Daten benutze ich an dieser Stelle die von `XmlNode` abgeleitete Klasse `XmlDocument`, also den DOM-Ansatz. Dazu benötigen wir für die Anzeige ein kleines Formular mit einer Listbox und einer Schaltfläche für das Anzeigen des DATEI ÖFFNEN-Dialogs. Um die Listbox bei einer Größenänderung des Formulars automatisch anzupassen, setzen Sie die Eigenschaft `Anchors` der Listbox auf allen vier Seiten auf aktiv.

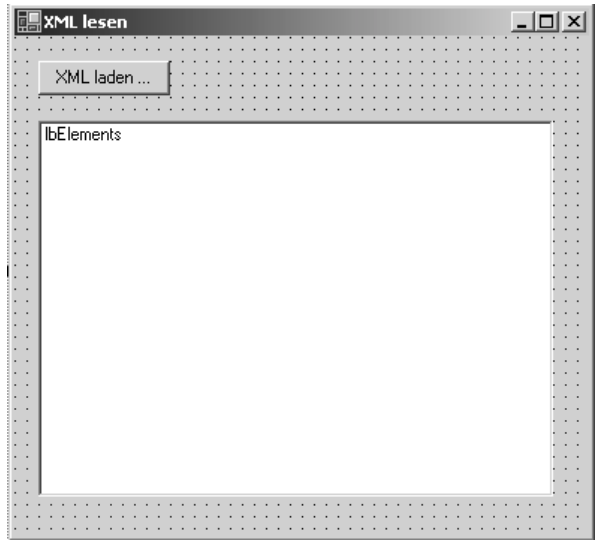


Abbildung 14.3: Das Formular zur Anzeige der XML-Daten

Da wir auf die Klassen des `Xml`-Namensraumes zugreifen werden, muss der Abschnitt mit den `using`-Anweisungen um die entsprechende Zeile ergänzt werden. Zusätzlich wird noch der Namensraum `System.Text` importiert, da wir für das Programm noch eine Instanz eines String-Builders benötigen.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Text;
using System.Xml;
```

Listing 14.5: Die using-Anweisungen für das Programm

Nach einem Klick auf die Schaltfläche soll bei einem OK die ausgewählte Datei geöffnet und angezeigt werden. Dazu wird in der Ereignisbehandlungsroutine für die Schaltfläche `btnLoad` folgender Code eingetragen:

```
// ereignisbehandlungsroutine für das laden der datei

private void btnLoad_Click(object sender, System.EventArgs e)
{
   OpenFileDialog dlgOpen = new OpenFileDialog();
   if (DialogResult.OK == dlgOpen.ShowDialog())
   {
      // listbox leeren
      lbElements.Items.Clear;

      // datei laden und anzeigen
      XmlDocument doc = new XmlDocument();
      doc.Load(dlgOpen.FileName);
      ShowDoc(doc, lbElements);
   } else
      MessageBox.Show("Dann eben nicht ...");
  }
}
```

Listing 14.6: Dieser Code wird beim Klick auf »XML laden…« ausgeführt

Zuerst wird eine neue Instanz eines Standarddialogs (DATEI ÖFFNEN) erzeugt und angezeigt. Klickt der Benutzer auf die Schaltfläche OK des Dialogs, wird eine neue Instanz der Klasse `XmlDocument` erzeugt. Mit Hilfe der Methode `Load` wird die Datei geladen und im Speicher als Baumstruktur aufgebaut.



Die Methode `Load` der Klasse `XmlDocument` ist mehrfach überladen und akzeptiert vier verschiedene Quellen für die XML-Datei:

- `public virtual void Load(Stream);`
  - Lädt die XML-Datei aus dem angegebenen Stream.
- `public virtual void Load(string);`
  - Lädt den Inhalt des XML-Dokuments aus der angegebenen URL bzw. dem angegebenen Dateinamen.
- `public virtual void Load(TextReader);`
  - Lädt den Inhalt der XML-Datei aus dem angegebenen TextReader.
- `public virtual void Load(XmlReader);`
  - Lädt den Inhalt der XML-Datei aus dem angegebenen XmlReader, einem XML-Strom.

Darüber hinaus besteht noch die Möglichkeit, mit der Methode `LoadXml` direkt einen XML-String zu übergeben und aus diesem das Dokument im Speicher aufzubauen.

Alle Überladungen der Methode `Load` und `LoadXml` führen allerdings keine Überprüfung des Dokumentes auf Gültigkeit anhand einer Document Type Definition oder eines XML-Schemas durch. Falls Sie beim Laden eine Überprüfung des Dokuments auf Gültigkeit durchführen lassen wollen, benutzen Sie dazu einen `XmlValidatingReader` und geben den gewünschten Typ der Überprüfung an. Mehr Informationen dazu finden Sie in der Online-Dokumentation.

So weit, so gut. Die .NET-Laufzeitumgebung hat also die XML-Datei geladen und die entsprechende Baumstruktur der einzelnen Knoten im Speicher aufgebaut. Auf das erste Element erhalten Sie über die Zeile

```
XmlNode root = doc.FirstChild;
```

Zugriff. Die Methode `FirstChild` liefer eine Instanz der Klasse `XmlNode` zurück, welche die Basis für die weitere Arbeit mit der Datei bildet. So lässt sich unter anderem mit der Eigenschaft `HasChildNodes` ferststellen, ob dieser Knoten des Dokumentbaums selbst wieder über Unterknoten verfügt. Damit Sie den folgenden Code nachvollziehen können, werfen Sie am besten noch einmal einen Blick auf die Beispieldatei BOOKS.XML.

```
// anzeige der xml-datei in der listbox

private void ShowDoc(XmlDocument doc, ListBox box)
{
   // ersten knoten holen
   XmlNode root = doc.FirstChild;
   //Display the contents of the child nodes.
   if (root.HasChildNodes)
   {
      // durch alle unterknoten laufen
      foreach(XmlElement n1 in root.ChildNodes)
      {
        // den namen des tags ausgeben
        lbElements.Items.Add(
         String.Format("Tag name: {0}, {1} Attribute",
         n1.Name, n1.Attributes.Count));
        // und alle unterknoten dieses knotens
        // eingerückt ausgeben
        foreach(XmlElement n2 in n1.ChildNodes)
        {
          lbElements.Items.Add(String.Format("\t{0}",
                              n2.InnerXml));
        }
      }
   }
}
```

Listing 14.7: Anzeige des Dokuments

Falls der oberste Knoten (das `root`-Element, im Beispiel mit dem Tag `<bookstore>`) über Unterknoten verfügt, werden diese in einer `foreach`-Schleife durchlaufen und jeweils der Name des Knotens (der Tag-Name) und die Anzahl der im Knoten vorhandenen Attribute ausgegeben. Danach erfolgt für alle Unterknoten eine eingerückte Ausgabe des XML-Codes.

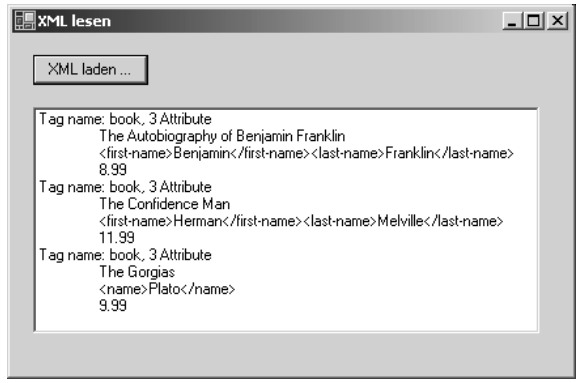


Abbildung 14.4: Die Ausgabe des ersten Beispielprogramms

14.3.4 Suchen und Ändern im Dokument

Um auf einzelne Elemente gezielt zuzugreifen, bietet die .NET-Laufzeitumgebung eine ganze Reihe von Klassen. Für das nächste Beispiel möchte ich die Ausgabe in der Listbox gerne verbessern und nur noch die einzelnen Titel ohne die ganzen XML-Tags drumherum anzeigen.

Dazu benutze ich die Methode `GetElementsByTagname` der Klasse `XmlDocument`. Diese Methode liefert eine Liste aus XML-Knoten (eine Instanz von `XmlNodelist`), die für die Ausgabe verwendet werden kann. Der resultierende Code ist noch dazu wesentlich kürzer als die erste Version der Ausgabe:

```
// zugriff auf einzelne knoten über
// GetElementsByTagName

private void ShowDoc2(XmlDocument doc, ListBox box)
{
   XmlNodeList list = doc.GetElementsByTagName("title");
   foreach(XmlElement node in list)
   {
      lbElements.Items.Add(node.InnerXml );
   }
}
```

Listing 14.8: Auslesen aller title-Knoten der Beispieldatei

Natürlich existiert auch eine Reihe von Methoden zum Selektieren von Knoten anhand der ID, zum Suchen eines einzelnen Knotens, zum Feststellen der Unterknoten eines Knotens und vieles mehr. Werfen Sie einfach einen Blick in die Online-Hilfe zu den Klassen `XmlNode` und `XmlDocument`.

14.3.5 Einfügen und Löschen von Knoten

Um unsere Beispieldatei um ein neues Buch zu erweitern, muss dem DOM-Baum im Speicher ein neuer Knoten hinzugefügt und entsprechend gefüllt werden. Dazu reicht es nicht aus, das XML-Element (den Knoten) einfach zu erzeugen. Der W3C-Standard fordert, dass explizit eine Einfügeoperation wie z.B. `AppendChild` eines Knotens zu erfolgen hat. Aber sehen wir uns das doch gleich in der Methode `NeuesBuch` an:

```
// fügt ein neues buch zum dokument hinzu

private void NeuesBuch(XmlDocument doc)
{
   // einen neuen knoten anlegen
   XmlElement element = doc.CreateElement("book");

   // jetzt das neue buch im dokument eintragen
   doc.DocumentElement.AppendChild(element);

   // die knoten title, author und price anlegen
   XmlElement title = doc.CreateElement("title");
   title.InnerText = "Zen or Maintaining " +
                     "Assembler Code";

   XmlElement author = doc.CreateElement("author");
   author.InnerText = "D. Bug";

   XmlElement price = doc.CreateElement("price");
   price.InnerText = "12.98";
   // ein attribut zu demozwecken
   price.SetAttribute("currency", "EUR");

   // jetzt unter dem neuen buchknoten einhängen
   element.AppendChild(title);
   element.AppendChild(author);
   element.AppendChild(price);
}
```

Listing 14.9: So lässt sich ein neuer Knoten einfügen

Zuerst wird mit `CreateElement` ein neuer Knoten erzeugt und als Unterknoten der Eigenschaft `DocumentElement` (also des `root`-Knotens) eingetragen. Anschließend werden die drei benötigen Unterknoten erzeugt und unterhalb des Knotens für das neue Buch eingehängt. Mit Hilfe der Eigenschaft `InnerText` eines `XmlElement` lässt sich der angezeigte Text lesen oder setzen. Analog dazu funktionieren die Eigenschaften `OuterText`, `InnerXml` und `OuterXml`. Zu Demonstrationszwecken habe ich bei dem Knoten für den Preis noch ein Attribut hinzugefügt. Wie Sie sehen, erfordert dies nur eine einzige Zeile Code.

14.3.6 XmlDocument-Instanzen in Dateien speichern

Der einfachste Weg vorneweg: Die Methode `Save` der Klasse `XmlDocument` besitzt eine Überladung, für die Sie nur einen Dateinamen angeben. Danach enthält die Datei den Inhalt der `XmlDocument`-Instanz, die diese Methode aufruft. Allerdings haben Sie sonst keinen weiteren Einfluss auf die Gestaltung des Dokumentes. Dafür ist der Code an Kürze kaum zu unterbieten:

```
// doc ist eine instanz von XmlDocument
doc.Save("datei.xml");
```

Eine andere Möglichkeit bieten die Methoden `WriteTo` und `WriteContentTo`, die als Parameter einen `XmlWriter` erwarten. Der Unterschied liegt darin, dass `WriteTo` das komplette Dokument herauschreibt und `WriteContentTo` den Inhalt des Dokumentelementes, also alle Unterknoten. Anstatt wie in manchen anderen Büchern hierfür jetzt seitenweise ein Listing zum Vergleich aufzuführen, lässt sich der Unterschied auch einfacher erklären: Ein `WriteTo` entspricht dem Inhalt der Eigenschaft `OuterXml`, während das `WriteContentTo` den Inhalt der Eigenschaft `InnerXml` schreibt.

```
MessageBox.Show(doc.OuterXml);
```

liefert also eine Messagebox mit dem Inhalt der Beispieldatei (versuchen Sie das nicht mit einer 800 KByte großen XML-Datei!).

14.3.7 Lesen von XML-Dateien mit XmlReader

Die folgenden Beispiele benutzen den Streaming-Ansatz, den auch SAX verfolgt. Allerdings bietet die Klasse XmlReader den Vorteil eines Pull-Modells bei der Programmierung, so dass Sie im Gegensatz zu SAX nicht alles abholen müssen, was ankommt, sondern selektiv und schnell durch eine XML-Datei 'durchlesen' können. Aus Gründen der Einfachheit folgen diese Beispiele wieder als Konsolenanwendung. Es sollte allerdings mittlerweile kein Problem mehr für Sie darstellen, den Beispielen eine Windows Forms-Oberfläche zu verpassen.

Zuerst die allgemeine Form der Benutzung einer `XmlTextReader`-Instanz: Nach dem Erzeugen der Instanz wird in einer Schleife jeweils der nächste Knoten gelesen, bis der Aufruf der Methode `Read` ein `false` liefert. Zu Beginn hat der Reader gar keine Daten, so dass als allererste Operation immer ein `Read` erfolgen muss.

```
// schema für den XmlTextReader
XmlTextReader rdr = new XmlTextReader("books.xml");

while (rdr.Read())
{
   // hier der code für die schleife
}
rdr.Close();
```

Listing 14.10: Der grundlegende Ablauf beim Einsatz eines XmlReaders

Für das erste Beispiel sollen alle Buchtitel ausgelesen werden. Dazu muss der entsprechende Knoten folgende Voraussetzungen erfüllen: Es muss sich um einen Knoten vom Typ `XmlNodeType.Element` handeln und dieser muss einen Unterknoten vom Typ `XmlNodeType.Text` besitzen. Der erste Entwurf für den Code nutzt die Methode `MoveToContent` der Klasse `XmlTextReader`. Diese überliest praktischerweise alle Knotentypen bis zu einem Elementknoten, so dass Leerraum und Kommentare keine Rolle spielen.

```
// Goto C# -- Armin Hanisch
// streaming-ansatz für xml-dateien mit XmlTextReader

using System;
using System.IO;
```

```
using System.Xml;

namespace XmlReaderWriter
{
   class DemoKlasse
   {
      static void Main(string[] args)
      {
         // lesen der beispieldatei
         XmlTextReader rdr =
             new XmlTextReader("books.xml");

         while (rdr.Read())
         {
            if(rdr.MoveToContent() ==
               XmlNodeType.Element)
            {
               Console.WriteLine("Buchtitel = {0}",
                   rdr.ReadElementString("title"));
            }
         }
         rdr.Close();

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

Listing 14.11: Erste Version zum Lesen einer XML-Datei mit dem XmlTextReader

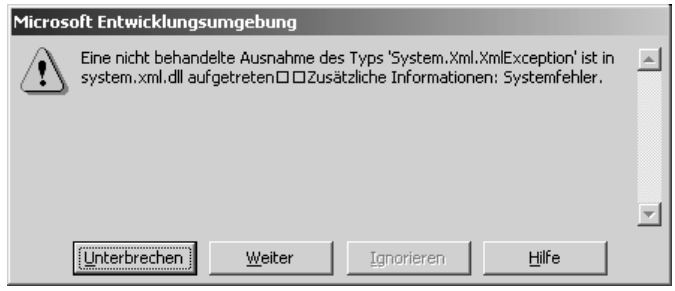


Abbildung 14.5: Fehler, da der Tagname nicht passt

In der Leseschleife wird das Ergebnis des `MoveToContent` auf einen Elementknoten getestet und dann mit der Methode `ReadElementString` der Text des »title«-Tags ausgelesen. Alternativ könnten Sie auch testen, ob die Eigenschaft `Name` der `XmlTextReader`-Instanz den Inhalt 'title' hat. Die große Überraschung folgt aber beim ersten Start des Programms.

Was ist passiert? `ReadElementString` soll einen Knoten mit dem Namen 'title' lesen. Das erste Element, auf das in unserer Beispieldatei gestoßen wird, ist aber das <bookstore>-Element, welches natürlich nicht über einen <title>-Unterknoten verfügt. Also wird einfach die evtl. auftretende Exception ignoriert und stur gelesen, bis tatsächlich in einem Element ein <title>-Tag vorkommt – sicher nicht überall die ideale Lösung, aber für unsere Zwecke hier ausreichend.

```
// Goto C# -- Armin Hanisch
// streaming-ansatz für xml-dateien mit XmlTextReader

using System;
using System.IO;
using System.Xml;

namespace XmlReaderWriter
{
   class DemoKlasse
   {
      static void Main(string[] args)
      {
         // lesen der beispieldatei
         XmlTextReader rdr =
             new XmlTextReader("books.xml");

         while (rdr.Read())
         {
            if(rdr.MoveToContent() ==
               XmlNodeType.Element)
            {
               try
               {
                  Console.WriteLine("Buchtitel = {0}",
                  rdr.ReadElementString("title"));
               }
               catch(XmlException)
               {
```

```
                    // ignorieren ....
                }
            }
        }
        rdr.Close();

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

Listing 14.12: Lesen einer XML-Datei mit dem XmlTextReader und Fehlerprüfung

Das Ergebnis entspricht dann der gewünschten Ausgabe:

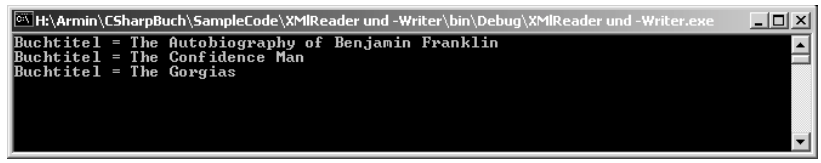


Abbildung 14.6: Die Ausgabe der Buchtitel in der Beispieldatei

Behandlung von Attributen

Attribute innerhalb eines XML-Tags werden als Ergänzung dieses Elements betrachtet und daher nicht gesondert zurückgeliefert. Sie können das Vorhandensein eines oder mehrerer Attribute mit der Eigenschaft `HasAttributes` testen und dann die einzelnen Attribute auslesen.

```
// alle attribute der xml-elemente ausgeben
private static void ShowAttributes(string filename)
{
   // lesen der beispieldatei
   XmlTextReader rdr = new XmlTextReader("books.xml");

   while (rdr.Read())
   {
      if(rdr.MoveToContent() == XmlNodeType.Element)
      {
         Console.WriteLine("Element name: <" +
                           rdr.Name + ">");
         if(rdr.HasAttributes)
         {
```

```
            while(rdr.MoveToNextAttribute())
            Console.WriteLine("   {0} = {1}",
                              rdr.Name, rdr.Value);
         }
      else
         Console.WriteLine("    <keine Attribute>");
      }
   }
   rdr.Close();
}
```

Listing 14.13: Auslesen aller Attributinformationen einer XML-Datei

Falls Sie nur das erste Attribut benötigen, existiert auch eine `bool`-Methode `MoveToFirstAttribute`. Der hier gezeigte Ansatz ist aber einfacher zu implementieren, da die Reihenfolge der Attribute in einem XML-Element nicht festgelegt ist und Sie daher nicht wissen können, welches Attribut an erster Stelle erscheint.

14.3.8 Schreiben von XML-Dateien mit XmlWriter

Jetzt haben Sie also alle Informationen für die zu erstellende XML-Datei, wollen aber nicht erst einen DOM-Baum im Speicher aufbauen. Zusätzlich beträgt die Datenmenge etwa 500 MByte, wie also bekommen Sie diese Menge an Informationen mit vernünftigem Ressourcenverbrauch aus Ihrer SQL-Datenbank in eine Datei? Die Antwort lautet `XmlWriter` und `XmlTextWriter`, die von der abstrakten Basisklasse abgeleitete Klassen, die ich Ihnen hier vorstellen möchte.

Diese Klasse arbeitet wie der `XmlTextReader` als reine Streamingklasse, nur vorwärts und ohne Pufferung, eignet sich daher auch sehr gut zum 'on the fly'-Erstellen von XML-Dateien aus vorliegenden Daten. Die Klasse bietet Ihnen eine ganze Reihe von möglichen Einstellungen, von der Anzahl der Leerzeichen pro Einrückungsebene bis zum Zeichen für das Begrenzen der Attributwerte.

Hier ein Beispiel für das Setup einer Instanz dieser Klasse:

```
// hier dateinamen anpassen
string datei = "newbooks.xml";
```

```
// null als parameter => std-encoding, also UTF-8
XmlTextWriter writer = new XmlTextWriter(datei, null);

writer.Formatting = Formatting.Indented;
writer.Indentation = 1;
writer.IndentChar = '\t';
```

Als Einrückung wird hier also das Tabulatorzeichen und zwar eines pro Stufe benutzt. Bitte denken Sie daran, dass als Einrückungszeichen nur ein Zeichen akzeptiert wird, welches aus der Menge der Leerraum-Zeichen (*whitespace*) stammt.

Anschließend kann in die Datei geschrieben werden. Hierfür stehen mehrere Methoden zur Verfügung:

- `WriteStartDocument` – schreibt den Anfang eines Dokuments (<?xml version='1.0' ?>)
- `WriteEndDocument` – schließt das Dokument ab
- `WriteStartElement` – schreibt ein »<element-name>« in die Datei
- `WriteEndElement` – schreibt ein »</element-name>« in die Datei
- `WriteElementString` – schreibt Elementname und Textinhalt in einem Aufruf
- `WriteAttributeString` – ergänzt das aktuelle Element um den Attributinhalt

Dies sind nur die wichtigsten Methoden, die auch im folgenden Beispiel Verwendung finden. Darüber hinaus finden Sie vom Schreiben von Kommentaren, CDATA-Elementen und anderem alles, was für die Erstellung komplexer XML-Dokumente nötig ist.

Hier eine Klasse, mit der eine XML-Datei für einen Eintrag in einer Bücherliste angelegt wird (mir fiel gerade kein anderes Buch ein …)

```
// Goto C# -- Armin Hanisch
// beispielklasse zum erstellen einer xml-datei

class BookClass
{
   public static void CreateXML(string filename)
```

```
    {
        // setup für den XmlTextWriter
        XmlTextWriter writer =
              new XmlTextWriter(filename, null);
        writer.Formatting = Formatting.Indented;
        writer.Indentation = 1;
        writer.IndentChar = '\t';

        // dokument starten
        writer.WriteStartDocument();

        writer.WriteStartElement("buch");

        // ein paar attribute dazu
        writer.WriteAttributeString("kategorie",
               "Softwareentwicklung");
        writer.WriteAttributeString("jahr", "2002");

        // unterelement schreiben
        writer.WriteElementString("titel", "Goto C#");

        writer.WriteStartElement("autor");
        writer.WriteElementString("vorname", "Armin");
        writer.WriteElementString("nachname", "Hanisch");
        writer.WriteEndElement();

        // endelement für "buch"
        writer.WriteEndElement();

        // dokument abschliessen
        writer.WriteEndDocument();

        // rausschreiben
        writer.Flush();
        writer.Close();

        // bescheid geben
        Console.WriteLine("Datei wurde erstellt," +
                          " weiter mit [ENTER]");
        Console.ReadLine();
    }
}
```

Listing 14.14: Eine Klasse zum Anlegen eines Bucheintrags

Hier das Ergebnis dieses Programms:

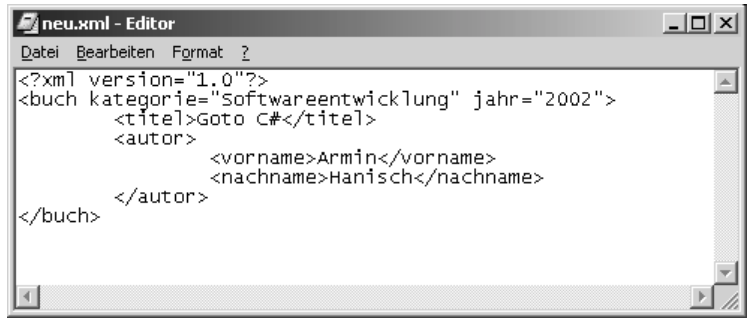


Abbildung 14.7: Die neu angelegte XML-Datei

14.4 Übungen

Ü

1. Erstellen Sie ein Programm, mit dem Sie in einem Verzeichnis alle Dateien mit den Erweiterungen *.bak*, und.*tmp.* löschen können, wenn diese älter sind als zwei Wochen.
2. Lesen Sie im Ordner System32 von Windows alle Dateien mit der Erweiterung *.exe* aus, für die eine gleichnamige *.dll* oder *.hlp*-Datei existiert.
3. Lesen Sie die Datei *books.xml* ein und setzen Sie bei den `price`-Elementen ein Attribut `currency` ein, dessen Inhalt »EUR« lautet.

15 Strings

In C# (und natürlich dem Rest der .NET-Welt) sind die Strings Referenztypen und Instanzen der Klasse `System.String`. Im Gegensatz zu anderen Umgebungen sind Strings in .NET nicht einfach eine manipulierbare Kette von Zeichen im Speicher, sondern 'unveränderbare' Datentypen.

Benutzen Sie also eine Methode, welche augenscheinlich einen String (ich benutze ab hier String als Synonym für '*eine Instanz der Klasse System.String*') ändert, liefern diese Routinen einen neuen String zurück und der alte kann von der Garbage Collection entsorgt werden. Nach seiner Erzeugung besteht also auf einen String nur noch ein Lesezugriff.

15.1 String-Literale

Was soll dabei schon interessant sein, werden Sie sich fragen. Texte in Anführungszeichen eben. Beinahe! C# unterscheidet zwei Arten von Stringliteralen:

- Strings (Beispiel: `"Alice im \nWindowsland"`)
- Verbatim-Strings (Beispiel: `@ "Von C:\windows verweht"`)

Die normalen Stringkonstanten interpretieren nämlich den Backslash mit den auch in C und C++ üblichen Zeichen (wie `\n` für eine neue Zeile). Bei Pfadangaben brechen sich die C-Benutzer dann immer die Finger: "C:\\WINDOWS\\SYSTEM32" und ähnliche Konstrukte lassen klar werden, warum so viele Entwickler dem unter Unix verwendeten Schrägstrich als Pfadtrenner nachtrauern. In C# existieren dafür die Verbatim-Strings, welche durch ein vorangestelltes @-Zeichen gekennzeichnet werden. Hier werden alle Zeichen unverändert übernommen und es erfolgt keinerlei Ersetzung oder Behandlung von Sonderzeichen.

15.2 System.String im Überblick

Als kleines Beispiel zum Einstieg (falls Sie auf »Hallo, Welt« warten, muss ich Sie enttäuschen) möchte ich Ihnen gerne demonstrieren, worin der erste Unterschied besteht, den .NET-Einsteiger bei Strings bemerken (vor allem, wenn sie von VB kommen).

```
// string-indexer und length, vergleich mit CompareTo
using System;

namespace StringSamples
{
   class TestClass
   {
      static void Main(string[] args)
      {
         // unser test-string
         String s = "Warum also haben alle " +
                    "A in diesem Satz Klammern?";

         // durch alle zeichen laufen und
         // auf dieser über den Indexer zugreifen
         for(int i = 0; i < s.Length; i++)
         {
            if(s[i] == 'a')
               Console.Write("(A)");
            else
               Console.Write(s[i]);
         }
      }
   }
}
```

Listing 15.1: Die String-Klasse im ersten Einsatz

Starten Sie das Programm, führt es zu folgender (nicht sehr sinnvollen) Ausgabe und demonstriert auch gleich die beiden einzigen Eigenschaften der Klasse `String`.

Die eine Eigenschaft, `Length`, ist auch ohne große Erläuterung zu verstehen. Interessanter ist die zweite Eigenschaft, welche Sie in C# gar nicht als Eigenschaft, sondern als Indexer erkennen, nämlich der Zugriff auf die einzelnen Zeichen eines Strings.

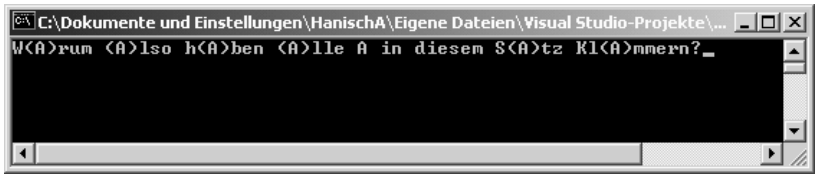


Abbildung 15.1: Die Ausgabe des Programms

In den anderen .NET-Sprachen, welche das Konzept eines Indexers ja nicht kennen, lautet der Name der Eigenschaft `Chars` und diese Eigenschaft wird mit einem Index benutzt (in VB.NET z.B. über `s.Chars(45)`).

Denken Sie daran, für die Zeichenkonstante `'a'` die einfachen Anführungszeichen zu benutzen. Verwenden Sie die doppelten Anführungszeichen, stellt dies eine Stringkonstante dar, der Indexer liefert aber einen `char` zurück und schon bekommen Sie eine der netten Meldungen, dass die Typen für diesen Operator nicht kompatibel sind. Sie könnten jetzt natürlich den Ausdruck in die Form `if(s[i].ToString() == "a")` umbauen, aber das sollte wirklich nur als Demonstration dienen und nie in Produktionscode auftauchen, versprochen?

15.2.1 Methoden der String-Klasse

Um ein paar sinnvollere Beispiele als das obige vorstellen zu können, ist natürlich etwas Vorwissen über die verfügbaren Methoden der Klasse `System.String` nötig. Daher finden Sie hier zuerst eine Übersichtstabelle mit den statischen Methoden der Klasse. Details und Beispiele folgen dann nach der Tabelle der Instanzmethoden.

Statische Methoden

Methode	Beschreibung
Compare	Führt einen Vergleich zweier Strings durch.
CompareOrdinal	Führt einen von den Sprach- und Landeseinstellungen unabhängigen Vergleich von zwei Strings durch.

Tabelle 15.1: Die statischen Methoden der String-Klasse

Methode	Beschreibung
Concat	Verbindet mehrere Strings oder Stringkonvertierungen von Objekten und liefert einen neuen String.
Copy	Liefert einen neuen String mit dem gleichen Inhalt wie der angegebene String.
Equals	Funktionalität von `Object.Equals()` wurde überladen. Stellt fest, ob beide Strings den gleichen Inhalt haben.
Format	Formatiert einen String entsprechend der Maske und liefert das Ergebnis als neuen String zurück (überladen).
Intern	Liefert eine Referenz auf einen String aus dem 'intern pool' bzw. legt eine neue Instanz dort an (Details s. Text).
IsInterned	Liefert eine Referenz auf einen String aus dem 'intern pool' oder eine Null-Referenz, falls der String noch nicht dort existiert (Details s. Text).
Join	Das Gegenteil der Instanzmethode `Split()`. Liefert einen aus den einzelnen Elementen eines Arrays zusammengesetzten String.

Tabelle 15.1: Die statischen Methoden der String-Klasse (Forts.)

Der »Intern Pool«

Die .NET-Laufzeitumgebung legt gleichlautende Stringkonstanten und Strings nur einmal in einem Speicherpool, dem sog. intern pool an. Damit wird verhindert, dass für eine zweite Stringkonstante oder Stringinstanz mit dem gleichen Inhalt unnötig Speicher verschwendet wird. Stattdessen wird einfach eine neue Referenz auf den bereits bestehenden String erzeugt. Nachdem Strings nach ihrer Erzeugung nicht mehr verändert werden dürfen, stellt das auch kein Problem dar. Mit den beiden Methoden `Intern()` und `IsInterned()` können Sie diesen Pool abfragen bzw. neue Einträge darin erzeugen.

Instanzmethoden

Für die Methoden der nachfolgenden Tabelle benötigen Sie eine Instanz eines Strings, rufen diese also nicht über den Klassennamen, sondern über die Objektvariable auf (aber das wissen Sie ja bereits aus dem Kapitel ***Klassen in C#***).

Methode	Beschreibung
Clone	Erzeugt eine Referenz auf die Instanz des Strings, welche die Methode aufruft.
CompareTo	Vergleicht die Stringinstanz mit einem anderen Objekt.
CopyTo	Kopiert einen angegebenen Bereich aus dem String in ein Array aus Zeichen.
EndsWith	Stellt fest, ob der String mit einer bestimmten Zeichenfolge endet.
Equals	Vergleicht, ob zwei Strings den gleichen Inhalt haben.
GetEnumerator	Liefert ein Aufzählungsobjekt, mit dem durch alle Zeichen des Strings iteriert werden kann.
GetHashCode	Liefert den Hashcode der Stringinstanz.
GetType	Liefert das `Type`-Objekt.
GetTypeCode	Liefert Typcode für die String-Klasse.
IndexOf	Liefert das erste Auftreten einer Zeichenfolge in dieser Stringinstanz.
IndexOfAny	Liefert das erste Auftreten eines Elements aus einem Zeichenarray in der Stringinstanz.
Insert	Fügt eine Stringinstanz an einer bestimmten Position in diese Stringinstanz ein.
LastIndexOf	Liefert das letzte Auftreten einer Zeichenfolge in dieser Stringinstanz.
LastIndexOfAny	Liefert das letzte Auftreten eines Elements aus einem Zeichenarray in der Stringinstanz.
PadLeft	Füllt die Stringinstanz auf die angegebene Länge an der linken Seite mit Leerzeichen oder einem angegeben Zeichen auf.
PadRight	Füllt die Stringinstanz auf die angegebene Länge an der rechten Seite mit Leerzeichen oder einem angegeben Zeichen auf.
Remove	Löscht ab der angegebenen Position eine Anzahl von Zeichen aus der Stringinstanz.
Replace	Ersetzt innerhalb der Stringinstanz jedes Vorkommen eines angegebenen Zeichens durch ein anderes Zeichen oder einen String.

Tabelle 15.2: Die Liste mit den Instanzmethoden der String-Klasse

Methode	Beschreibung
Split	Zerteilt die Stringinstanz in ein Array von Strings, jeweils beim Auftreten des angegebenen Trennzeichens.
StartsWith	Stellt fest, ob der String mit einer bestimmten Zeichenfolge beginnt.
Substring	Liefert einen Teilstring der Stringinstanz.
ToCharArray	Kopiert den Inhalt der Stringinstanz in ein Zeichenarray.
ToLower	Ergibt eine Kopie des Strings in Kleinbuchstaben.
ToString	Konvertiert den Wert der Instanz in einen String (für jedes Objekt verfügbar).
ToUpper	Ergibt eine Kopie des Strings in Großbuchstaben.
Trim	Entfernt alle angegebenen Zeichen vom Beginn und vom Ende der Stringinstanz.
TrimEnd	Entfernt alle angegebenen Zeichen vom Ende der Stringinstanz.
TrimStart	Entfernt alle angegebenen Zeichen vom Beginn der Stringinstanz.

Tabelle 15.2: Die Liste mit den Instanzmethoden der String-Klasse (Forts.)

15.3 Strings formatieren

Für die Formatierung von Strings können Sie die statische Methode `Format` der Klasse `System.String` verwenden. Diese erwartet als Parameter eine Formatmaske und danach eine Liste von Variablen oder Konstanten, welche entsprechend formatiert werden.

15.3.1 Standard-Formate

Diese Format-Strings ähneln der seit C bekannten Syntax der Funktion `printf`. Die Formatangaben werden bei C# allerdings in geschweifte Klammern eingeschlossen und mit einem bei `0` beginnenden Index durchnummeriert.

Die Methode `WriteLine` der Klasse `Console` besitzt eine eingebaute Unterstützung für die formatierte Ausgabe von Strings. Sie können für Zuweisungen und eigene Formatierungen aber auch jederzeit die statische Methode `Format` der Klasse `String` benutzen.

Die einfachste Form dieser formatierten Ausgabe überlässt die nummerische Formatierung dem System und steuert nur die Position der Parameter:

```
Console.WriteLine("{0} Euro waren {1} DM.",
                  5, 5 / 1.955830);
```

Diese Zeile erzeugt die folgende Ausgabe:

```
5 Euro waren 2,55645940598109 DM.
```

Wie die Ausgabe unschwer erkennen lässt, ist in vielen Fällen eine Änderung des Standardverhaltens wünschenswert. Die allgemeine Form eines solchen Platzhalters lautet `{N[,m][:<format>]}`.

Listen wir die einzelnen Bestandteile der Reihe nach auf:

- `N` ist ein bei 0 beginnender Integerindex für die Position des Arguments.
- `m` ist ebenfalls ein Integer, der die Breite des Ausgabefeldes bestimmt. Positive Werte bestimmen eine rechtsbündige Ausrichtung im Ausgabebereich, negative Werte formatieren die Ausgabe linksbündig.
- `<format>` ist eine eventuelle weitere Angabe von Formatierungscodes.

Einer der Vorteile dieser nummerischen Angabe gegenüber dem von C und C++ verwendeten System ist die beliebige Wiederholbarkeit der Ausgabe:

```
Console.WriteLine("{0} ist {0} und bleibt {0}", 5);
```

15.3.2 Standard-Formatierung

Die möglichen Formatierungscodes für einen Platzhalter zeigt die folgende Tabelle.

Format-zeichen	Beschreibung	Standardausgabe
C or c	Währungsformat	$XX,XX.XX ($XX,XXX.XX)
D or d	Decimal-Ausgabe	[-]XXXXXXX
E or e	Exponentialnotation	[-]X.XXXXXXE+xxx [-]X.XXXXXXe+xxx [-]X.XXXXXXE-xxx [-]X.XXXXXXe-xxx
F or f	Festkommaformat	[-]XXXXXXX.XX
G or g	Allgemeines Format	Entweder kurz oder exponential
N or n	Zahlenformat	[-]XX,XXX.XX
P or p	Prozentformat	Ausgabe als Prozentwert (*100 »%«)
R or r	Hin- und Zurück	Stellt sicher, dass in Strings gewandelte Zahlen bei der erneuten Konvertierung in eine Zahl das gleiche Resultat ergeben.
X or x	Hexadezimalformat	Die kleinstmögliche Hex-Darstellung.

Tabelle 15.3: Standardformate für die nummerische Formatierung

Aus `WriteLine("{0,10:p}", 0.16);` wird so die Ausgabe: `____16,00%`, zumindest bei den deutschen Systemeinstellungen (jeder der vier Unterstriche steht hier zur Verdeutlichung der tatsächlich ausgegebenen Leerstellen).

Alternativ können Sie das auszugebende Format noch genauer steuern, indem Sie einen kompletten Formatierungsstring angeben.

Zeichen	Bedeutung	Beschreibung
0	Null-Platzhalter	Gibt die Ziffer 0 aus, falls die Ausgabe weniger Zeichen umfasst als das Format.
#	Ziffernplatzhalter	Ersetzt das Zeichen durch eine Ziffer. Ist das Format breiter als die Ausgabe, werden die Zeichen ignoriert.
.	Dezimalpunkt	Gibt das länderspezifische Dezimaltrennzeichen aus.

Tabelle 15.4: Die einzelnen Elemente eines Formatierungs-Strings

Zeichen	Bedeutung	Beschreibung
,	Gruppentrenner	Trennt Zifferngruppen durch das länderspezifische Trennzeichen bzw. teilt durch 1000.
%	Prozentausgabe	Gibt das Prozentzeichen aus.
e+0 e-0	Exponential-ausgabe	Formatiert die Ausgabe in der wissenschaftlichen Notation (»E« oder »e« ist egal).
\	Literalkennung	Gibt das dahinter stehende Zeichen als literales Zeichen aus (z.B. \#).
{ }	Literalklammer	Zur Ausgabe der geschweiften Klammern verdoppeln Sie diese (»{{« gibt »{« aus).
'ABC' »ABC«	Stringliteral	Gibt die Zeichenfolge so aus, wie sie angegeben wurde.
;	Bereichstrenner	Trennt Ausgabeformate für positive, negative und Null-Werte.
	Alle anderen Zeichen	Alle anderen Zeichen werden ohne Änderung in die Ausgabe übernommen.

Tabelle 15.4: Die einzelnen Elemente eines Formatierungs-Strings (Forts.)

Damit Sie sich die einzelnen Formatierungen besser vorstellen können, finden Sie nachfolgend Programmcode und Ausgabe für verschiedene Formatierungseinstellungen.

```
public static void Main()
{
   Console.WriteLine("Ausgabeformatierungen\n");
   Console.WriteLine("000000000111111");
   Console.WriteLine("123456789012345");
   Console.WriteLine("....|....|....|");
   Console.WriteLine( "{0,15:c}", 0.16*1845.50 );
   Console.WriteLine( "{0:p}", 0.16 );
   Console.WriteLine( "{0,10:g}", 3.14159 );
   Console.WriteLine( "{0:g}", 3.14159 );
   Console.WriteLine( "{0:e}", 3.14159 );
   Console.WriteLine( "{0,12:f4}", 3.14159 );
   Console.WriteLine( "{0:x} {1:X}", 45054, 51966 );
   Console.WriteLine( "{0:}", 3.14159 );
}
```

Listing 15.2: Verschiedene Formatierungen im Programmcode …

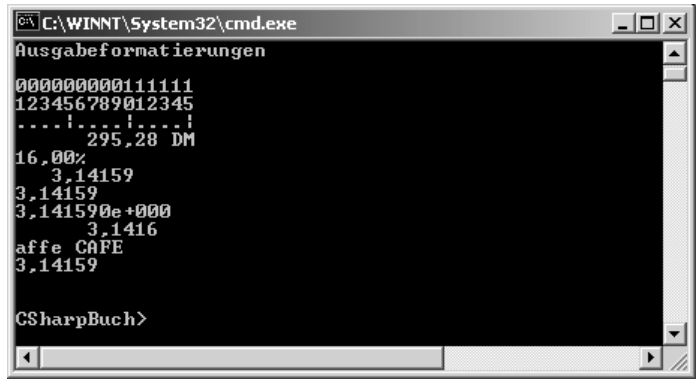


Abbildung 15.2: … und als Ausgabe im Konsolenfenster

Formatierung mit ToString()

Anstelle der Verwendung hinter dem Doppelpunkt des Platzhalters können Sie die Formatierungscodes der beiden Tabellen auch als Stringparameter an die Methode `ToString` übergeben und so das Standardverhalten der Umwandlung in einen String ändern.

```
Console.WriteLine( 5.ToString("00.00") );
```

Werteabhängige Formatstrings

Trennt man den Format-String mit dem Semikolon in einzelne Bereiche, dann wird der erste Abschnitt für die Formatierung positiver Werte, der zweite für negative Werte und der dritte für den Wert 0 benutzt. Sind weniger als drei Abschnitte vorhanden, gilt der erste Abschnitt für positive Werte und den Wert 0. Um die Möglichkeiten dieses Konzepts zu verdeutlichen, hier ein Beispiel und die entsprechende Ausgabe.

Die Trennung des Format-Strings über mehrere Zeilen ist in Ihren Programmen nicht notwendig, sie dient hier nur der Verhinderung eines Zeilenumbruchs durch das Satzprogramm. Sie können daher den FormatString in einer Zeile schreiben.

```
public static void Main()
{
   float x;

   string fmt = "Wärme 00.00°;" +
```

```
                "Kälte ##.#° unter Null;" +
                "Null Grad!";

   x = 35.5F; Console.WriteLine( x.ToString(fmt) );
   x = -5.2F; Console.WriteLine( x.ToString(fmt) );
   x = 0F;    Console.WriteLine( x.ToString(fmt) );

}
```

Listing 15.3: Werteabhängige Format-Strings



Abbildung 15.3: Die Ausgabe der Format-Strings

15.4 Die StringBuilder-Klasse

Die Klasse `StringBuilder` aus dem Namensraum `System.Text` ist die Lösung für alle Anforderungen, bei denen Sie eine Zeichenkette im Speicher wirklich bearbeiten müssen und nicht für jede Bearbeitung einen neuen String belegen möchten. Dieser String würde nach jedem Bearbeitungschritt wieder frei für die Garbage Collection und eine neue Instanz eines Strings würde wieder belegt. Ich denke, Sie können sich die Performance einer solchen Konstruktion in einer Schleife selbst ausmalen. Daher bietet die .NET-Laufzeitumgebung mit der Klasse `StringBuilder` eine praktische Alternative. Die Instanzen dieser Klasse bearbeiten wirklich einen Zeichenpuffer und belegen nicht jedesmal eine neue Instanz. Damit sollten Sie für die Verarbeitung von Zeichenketten (gerade auch in Schleifen) auf die `StringBuilder`-Klasse zurückgreifen.

15.4.1 Erzeugen eines StringBuilders

Für die Klasse existieren mehrere überladene Konstruktoren, so dass Sie den jeweils passenden verwenden können. Das kann ein Aufruf mit einem String als Startzeichenkette, einem nummerischen Wert für die Kapazität oder Kombinationen davon sein (wenn Sie dieses Buch vor dem Computer lesen, öffnen Sie doch einfach mal die Online-Hilfe zur Klasse `System.Text.StringBuilder`).

```
StringBuilder sb = new StringBuilder(8000);
StringBuilder sWork =
       new StringBuilder("Hallo, da draussen", 1024);
```

Listing 15.4: Beispiele für den StringBuilder-Konstruktor

Wenn Sie sicherstellen möchten, dass Sie für eine Operation über genügend Kapazität verfügen benutzen Sie die Methode `EnsureCapacity()`, mit der Sie prüfen können, ob die `StringBuilder`-Instanz mindestens die angegebene Zahl an Zeichen speichern kann.

```
if(sb.EnsureCapacity(1024) >= 1024)
   Machwas(sb);
else
   throw new Exception("Zu wenig Speicher für sb!");
```

Listing 15.5: Kapazitätstest mit der Methode EnsureCapacity

15.4.2 Eigenschaften der Klasse

Die Klasse `System.Text.StringBuilder` besitzt nur vier Eigenschaften, die alle selbsterklärend sind, weshalb ich sie nur kurz in einer Tabelle zusammenfasse.

Eigenschaft	Bedeutung
Capacity	Liest bzw. setzt die Kapazität der aktuellen `StringBuilder`-Instanz und entspricht der Anzahl der speicherbaren Zeichen.
MaxCapacity	Die theoretische maximale Kapazität, auf welche Sie dieses `StringBuilder`-Instanz einstellen können, normalerweise 2 Gigabyte.

Tabelle 15.5: Die Eigenschaften der Klasse StringBuilder

Eigenschaft	Bedeutung
Chars (Indexer)	In C# ist dies der Zugriff auf die einzelnen Zeichen der `StringBuilder`-Instanz über den Indexer wie bei der Klasse `String`.
Length	Die aktuelle Anzahl der Zeichen in dieser Instanz der `StringBuilder`-Klasse, also die Länge.

Tabelle 15.5: Die Eigenschaften der Klasse StringBuilder (Forts.)

15.4.3 StringBuilder-Methoden

Zur Bearbeitung einer in einem `StringBuilder` gespeicherten Zeichenkette stehen Ihnen die Methoden zur Verfügung, welche Sie vielleicht wie ich aus Gewohnheit zuerst in der `String`-Klasse gesucht haben: `Append`, `AppendFormat`, `Insert`, `Replace`, `Remove` und natürlich `ToString`.

- `Append` fügt am Ende des Zeichenpuffers neue Zeichen an. Diese Methode ist mehrfach überladen, so dass Sie den passenden Datentyp übergeben können.
- `AppendFormat` arbeitet ähnlich, bietet aber die Möglichkeit, mit einem Formatstring zuerst die Daten in eine entsprechende Form zu bringen.
- `Insert` und `Remove` fügen Zeichen an einer spezifizierten Stelle ein bzw. entfernen diese aus dem Zeichenpuffer der `StringBuilder`-Instanz.
- `Replace` tauscht einzelne Zeichen oder Zeichenketten gegen jeweils neue aus.
- `ToString` wandelt den Zeichenpuffer der `StringBuilder`-Instanz in einen String um.

15.5 String oder StringBuilder?

Diese Frage lässt sich relativ eindeutig beantworten, wenn Sie wissen, ob Sie die Zeichen des Strings bearbeiten müssen oder diese Zeichenkette nur mehr oder weniger konstant zur Ausgabezwecken benutzen. Zur Verdeutlichung habe ich eine Testklasse gebaut, in der die beiden Champions gegeneinander antreten können.

```
// laufzeitvergleich string und stringbuilder
using System;
using System.Text;

namespace StringSamples
{
   class TestClass
   {
      static void Main(string[] args)
      {
         const int cCOUNT = 10000;

         StringBuilder sb = new StringBuilder();
         String s = "";
         DateTime dt;

         // zuerst der stringbuilder
         dt = DateTime.Now;
         for(int i = 0; i < cCOUNT; i++)
         {
            sb.AppendFormat("Durchlauf {0} \n",i);
         }
         TimeSpan spanSB = DateTime.Now - dt;

         // dann der string
         dt = DateTime.Now;
         for(int i = 0; i < cCOUNT; i++)
         {
            s = s + "Durchlauf " + i.ToString()
                                 + '\x0a';
         }
         TimeSpan spanStr = DateTime.Now - dt;

         Console.WriteLine("Zeit StringBuiler = {0}",
                           spanSB);
         Console.WriteLine("Zeit String       = {0}",
                          spanStr);

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

Listing 15.6: Die Klasse für das 'Wettrennen' String gegen StringBuilder

Die Ausgabe liefert das folgende, sehr aufschlussreiche Ergebnis: Während bei 10.000 Durchläufen auf meiner Maschine fast 14 Sekunden vergehen, bis alle `Strings` im Speicher belegt, freigegeben und wieder belegt wurden, ist der `StringBuilder` mit seiner echten Zeichenverarbeitung in etwas mehr als einer Zehntelsekunde fertig.

```
Zeit für StringBuilder = 00:00:00.1101584
Zeit für String        = 00:00:13.8599296
```

Fazit: War die Verkettung von Zeichen in Schleifen schon in Visual Basic oder auch Delphi nicht besonders effizient, tritt dieser Unterschied bei C# und der .NET-Laufzeitumgebung umso deutlicher zutage. Nachdem ein String ein unveränderlicher Datentyp ist, muss die Laufzeitumgebung nach jedem Durchlauf eine neue Instanz anlegen und die alte freigeben. Innerhalb eines Durchlaufs werden durch die Verkettung noch einmal zusätzliche Speicheroperationen mit der Bildung neuer Stringinstanzen fällig. Auch wenn es anfangs noch etwas ungewohnt ist, sehen Sie hier den Vorteil der Klasse `System.Text.StringBuilder`.

15.6 Übungen



1. Erstellen Sie eine Filteranwendung, bei der eine Textdatei eingelesen wird und über einen Stream als ASCII-Datei wieder geschrieben wird, und ersetzen Sie darin alle Zeichen, die nicht im ASCII-Standard vorliegen, durch `%` und den Zeichencode in hexadezimaler Darstellung.
2. Wozu dienen Verbatim-Strings in C#?
3. Wie können Sie feststellen, ob ein bestimmter String bereits im Speicherpool der .NET-Laufzeitumgebung vorhanden ist?

16 Konvertierungen

In diesem Kapitel stelle ich Ihnen die Möglichkeiten von C# zur Konvertierung von Datentypen in einen anderen Datentyp vor. Neben den eingebauten Umwandlungen erfahren Sie auch, wie Sie für eigene Typen Umwandlungen erstellen können. Auch wenn es für viele Entwickler zum täglichen Brot gehört: Das Thema Konvertierungen oder Typumwandlungen ist für Nutzer mancher Programmiersprachen tatsächlich völlig unbekannt. VBScript etwa kennt nur einen einzigen Datentyp (Variant), bei anderen Sprachen wie C lässt Sie der Compiler nummerische Datentypen munter durcheinander zuweisen oder eine Datei durch 3 teilen, Sie werden schon wissen, was dabei herauskommt. Diese Szenarios sind eine der Hauptursachen für Fehler in Programmen. Wie Sie gleich sehen werden, hat man aus diesem Grund bei der Entwicklung von C# sehr viel Wert auf ein strenges Typkonzept gelegt.

Im Englischen wird dieser Prozess der Konvertierung als *type cast* (oder nur *cast*), also das 'Umgießen' eines Datentyps in die Form eines anderen Datentyps, bezeichnet. An vielen Stellen der deutschen Literatur wird daher auch die nur halb eingedeutschte Form Typcast verwendet. Ich benutze im Verlauf dieses Kapitels die Begriffe Konvertierung und Typumwandlung synomym.

16.1 Implizite Konvertierungen

Eine implizite Typumwandlung geschieht bei Bedarf automatisch durch die .NET-Laufzeitumgebung und den Compiler, wie in den folgenden Codezeilen:

```
int x;
byte b = 128;

// implizite typkonvertierung von byte nach int
x = b;
```

Listing 16.1: Funktionierende implizite Typumwandlung

Diese Zuweisung eines Wertes ist nur dann möglich, wenn dabei keine Information verlorengehen kann. So können Sie jederzeit einer `int`-Variablen einen `byte`-Wert zuweisen, da der Integer-Datentyp auf jeden Fall ein Byte aufnehmen kann. Umgekehrt ist dies natürlich nicht möglich, da der Wertebereich eines Bytes nur Werte zwischen 0 und 255 umfasst und damit wesentlich kleiner als der eines Integers ist.

```
int x = 1764;
byte b;

// nicht moeglich, da keine implizite
// konvertierung vorhanden
b = x;
```

Listing 16.2: Hier ist keine implizite Typumwandlung möglich

16.1.1 .NET-Datentypen

Die folgenden Typumwandlungen erfolgen implizit bei Bedarf und sind wertsicher, so dass keine Information verloren geht. Für alle anderen nummerischen Umwandlungen der Datentypen untereinander muss eine explizite Typkonvertierung vorgenommen werden.

Von	Nach
sbyte	short, int, long, float, double, or decimal
byte	short, ushort, int, uint, long, ulong, float, double, or decimal
short	int, long, float, double, or decimal
ushort	int, uint, long, ulong, float, double, or decimal
int	long, float, double, or decimal
uint	long, ulong, float, double, or decimal
long	float, double, or decimal
char	ushort, int, uint, long, ulong, float, double, or decimal
float	double
ulong	float, double, or decimal

Tabelle 16.1: Liste der impliziten Typumwandlungen

16.2 Explizite Konvertierungen

Eine explizite Typkonvertierung ist der klassische *type cast*. Hier wird vor dem Bezeichner oder dem Ausdruck ein Typkennzeichen angegeben. Der Compiler ist dann in der Lage, die entsprechende Konvertierung vorzunehmen, wenn dafür eine entsprechende Umwandlung definiert wurde.

```
int x = 1764;
byte b;

// moeglich, da keine explizite
// konvertierung vorhanden
b = (byte)x;
```

Listing 16.3: Die explizite Typumwandlung funktioniert

Nach dieser Operation besitzt `b` den Wert von 228 (= 1764-(3*256)).

16.3 Benutzerdefinierte Konvertierungen

Hier müssen Sie selbst tätig werden, da die .NET-Laufzeitumgebung ja nicht weiß, wie Sie Ihre eigenen Strukturen und Klassen ineinander oder in einen der C#-Datentypen umwandeln möchten.

16.3.1 Der Beispieltyp ClockTime

Dazu brauchen wir wieder ein Beispiel. Wie sie sicher bereits bemerkt haben, habe ich ein Faible für Zeit- und Datumsberechnungen. In diesem Kapitel möchte ich Ihnen daher zeigen, wie Sie für einen eigenen Zeit-Datentyp die entsprechenden Konvertierungen programmieren. Nachdem nicht viele Daten gespeichert werden und wir keine Vererbung benötigen, habe ich den Typ diesmal als `struct` realisiert (wenn Sie in der Dokumentation zum .NET-Framework nachsehen, finden Sie `DateTime` ebenfalls als `struct` implementiert).

```
// struktur zum speichern einer uhrzeit
struct ClockTime
{
   int fhours;
   int fminutes;
   int fseconds;
```

```
   // konstruktor nr. 1
   public ClockTime(int hrs, int min, int sec)
   {
      fhours = hrs;
      fminutes = min;
      fseconds = sec;
   }

   // konstruktor nr. 2
   public ClockTime(int seconds)
   {
      fhours = (int)seconds / 3600;
      fminutes = (int)(seconds - (fhours * 3600)) / 60;
      fseconds = seconds - (fhours * 3600) -
                 (fminutes * 60);
   }
}
```

Listing 16.4: Die Erstfassung unserer Struktur für die Zeitspeicherung

Die Notwendigkeit einer Konvertierung

Um unseren Datentyp benutzen zu können, müssen wir nur eine entsprechende Variable deklarieren und den Konstruktor aufrufen. Denken Sie daran, dass bei einer Struktur die Benutzung von `new` keine Daten auf dem Heap anlegt, da Strukturen Werttypen sind. Es wird lediglich der Code des passenden Konstruktors aufgerufen.

```
class TestClass
{
   static void Main()
   {
      ClockTime x;

      x = new ClockTime(07, 00, 00);

      x = x + 500;  // ergibt einen fehler!
   }
}
```

Listing 16.5: Es ist keine passende Konvertierung vorhanden

Wenn Sie diesen Code übersetzen lassen, erhalten Sie vom Compiler die Fehlermeldung »Der Operator '+' kann nicht auf Operanden des Typs 'Samples.ClockTime' und 'int' angewendet werden«. Da der Ope-

rator + nicht für unseren Datentyp deklariert wurde und wir noch dazu eine Integer-Konstante addieren, versucht der Compiler, eine Konvertierung zu finden, welche entweder beide Operanden in den Typ `ClockTime` oder `int` umwandelt. Da das C#-Team bei Microsoft sicherlich keine Typkonvertierung von `int` nach `ClockTime` in die .NET-Laufzeitumgebung eingebaut hat, müssen wir für eine entsprechende Routine sorgen, die `ClockTime` nach `int` konvertiert.

16.3.2 Implizite Konvertierungsroutine

Ich erweitere nun die Struktur um eine Methode, mit der eine implizite Konvertierung von `ClockTime` nach `int` möglich ist.

```
// die konvertierungs-methode
public static implicit operator int(ClockTime ct)
{
   return ct.fhours*3600 +
          ct.fminutes*60 +
          ct.fseconds;
}
```

Listing 16.6: Der Datentyp kann nun nach int konvertiert werden

Diese Konvertierungen werden von C# wie ein Operator behandelt und alle Umwandlungsmethoden müssen als statische, öffentliche Methoden deklariert werden. Der Name des Operators ist der Datentyp, in den gewandelt werden soll, während der umzuwandelnde Wert mit dem ursprünglichen Datentyp deklariert wird.

Damit wir später auch eine schöne Ausgabe bekommen, füge ich gleich noch eine weitere implizite Typumwandlung hinzu, nämlich von `ClockTime` nach `string`.

```
// konvertierung clocktime-wert in einen string
public static implicit operator string(ClockTime ct)
{
   return String.Format("{0,2:00}:{1,2:00}:{2,2:00}",
                ct.fhours, ct.fminutes, ct.fseconds);
}
```

Listing 16.7: Diese Konvertierung ist sehr einfach, einfach einen String bauen

Damit bin ich aber noch nicht fertig. Der Ausdruck `x = x + 500` kann nun zwar ausgewertet werden, da `x` implizit (lies: bei Bedarf automatisch) in einen `int` konvertiert wird, allerdings brauchen wir für die linke Seite der Zuweisung wieder einen Wert vom Type `ClockTime`. Diesen erstelle ich als explizite Methode. Im übernächsten Abschnitt erläutere ich dann die Gründe, wann Sie eine implizite und wann eine explizite Konvertierung wählen sollten.

16.3.3 Explizite Konvertierungsroutine

Diese Methode erlaubt eine Typumwandlung von `int` nach `ClockTime` und damit kann unserem Datentyp jetzt auch eine Integer-Variable zugewiesen werden.

```
public static explicit operator ClockTime(int seconds)
{
   // code to convert from int to ClockTime
   if((seconds > 0) && (seconds < 86400))
      return new ClockTime(seconds);
   else
      throw new ArgumentOutOfRangeException(
             "seconds", "0 <= seconds < 86400");
}
```

Listing 16.8: Explizite Konvertierung von int nach ClockTime

Wird ein gültiger Wert übergeben (0 bedeutet genau Mitternacht und ein Tag besitzt 86.400 Sekunden), dann erfolgt eine Umrechnung über den Konstruktor, andernfalls löse ich die Exception `ArgumentOutOfRangeException` aus.

Als explizite Konvertierung muss diese im Quelltext auch angegeben werden. Damit ändert sich die Zeile im Quelltext zu `x = (ClockTime)(x + 500)`. Ich behandle den gesamten Ausdruck `(x + 500)` also als `ClockTime` und kann damit die Zuweisung durchführen.

16.3.4 Implizit oder explizit?

Die generelle Regel lautet größtmögliche Sicherheit für den Code. Dies bedeutet, dass eine implizite Konvertierung keine Fehler auslösen darf und der Wertebereich der aufnehmenden Variablen so groß sein muss, dass keine Informationen verloren gehen.

Ist eine Prüfung nötig (nicht alle gültigen `int`-Werte beispielsweise sind auch gültige Werte für `ClockTime`), kann es zu einem Laufzeitfehler kommen. Diese Konvertierungen sollten als explizite Typumwandlungen programmiert werden, damit der Compiler nicht automatisch eine dieser Konvertierungen auswählt und Ihr Programm den Weg alles Irdischen geht.

16.3.5 Auflösungskonflikte

Im Beispiel für dieses Kapitel habe ich zwei implizite Konvertierungen in die Struktur aufgenommen, jeweils nach `int` und nach `string`. Was passiert nun bei einer Ausgabe unseres Zeitwertes? Hier der Code:

```
Console.WriteLine( x );
```

Auch hier meldet sich der Compiler mit einer Fehlermeldung zurück, dass die Angabe der Typumwandlung nicht eindeutig genug ist. Was ist passiert?

WriteLine besitzt mehrere Überladungen, um möglichst viele Datentypen ausgeben zu können. Eine davon akzeptiert einen `string`, hier ist in `ClockTime` eine implizite Konvertierung vorhanden. Eine andere Überladung akzeptiert aber einen `long`. In `ClockTime` existiert eine implizite Konvertierung von nach `int` und in der .NET-Laufzeitumgebung existiert wiederum eine implizite Konvertierung von `int` nach `long`. Damit hat der Compiler zwei gleichberechtigte Möglichkeiten und kann ohne Angabe nicht entscheiden, welche der beiden Varianten er wählen soll. Sie sehen, bei mehr als einer impliziten Typkonvertierung kann durchaus auch einmal eine explizite Angabe nötig sein.

16.3.6 Die komplette Struktur

Für den Überblick hier noch einmal die Struktur mit den Konvertierungen und der Testklasse, wie Sie sie auch auf der CD finden.

```
/*
   Goto C# -- Armin Hanisch
   benutzerdefinierte konvertierungen
*/
using System;
```

```
namespace Konvertierung
{

   struct ClockTime
   {
      int fhours;
      int fminutes;
      int fseconds;

      public ClockTime(int hrs, int min, int sec)
      {
         fhours = hrs;
         fminutes = min;
         fseconds = sec;
      }

      public ClockTime(int seconds)
      {
         fhours = (int)seconds / 3600;
         fminutes = (int)(seconds - (fhours * 3600))
                    / 60;
         fseconds = seconds - (fhours * 3600) -
                    (fminutes * 60);
      }

      public static explicit operator
      ClockTime(int seconds)
      {
         // code to convert from int to ClockTime
         if((seconds > 0) && (seconds < 86400))
                return new ClockTime(seconds);
         else
            throw new
                  ArgumentOutOfRangeException(
                  "seconds", "0 <= seconds < 86400");
      }

      public static implicit operator int(ClockTime ct)
      {
         return ct.fhours*3600 +
                ct.fminutes*60 +
                ct.fseconds;
      }

      public static implicit operator
```

```
      string(ClockTime ct)
      {
         return
            String.Format("{0,2:00}:{1,2:00}:{2,2:00}",
            ct.fhours, ct.fminutes, ct.fseconds);
      }

   }


   // die testklasse für die struktur
   class TestClass
   {
      static void Main()
      {
         ClockTime x;

         x = new ClockTime(07, 00, 00);
         x = (ClockTime)(x + 500);

         Console.WriteLine( (long)x );
         Console.WriteLine( (string)x );

         Console.WriteLine("Taste drücken ...");
         Console.ReadLine();
      }
   }
}
```

Listing 16.9: Die ClockTime-Struktur mit Typumwandlungen

16.4 Zusammenfassung

C# als typsichere Sprache erlaubt implizite und automatische Umwandlungen von einem Datentyp in den anderen nur dann, wenn keine Information verlorengeht. Für alle anderen Konvertierungen müssen Sie die Typumwandlung angeben.

Für benutzerdefinierte Datentypen ist sowohl die Erstellung von impliziten als auch expliziten Typumwandlungs-Operatoren möglich. Diese Methoden sind immer öffentlich und statisch für die Struktur oder Klasse.

Verfügen Sie über den Quellcode für beide Klassen, sollten Sie die Umwandlungen in die komplexere der beiden Klassen stecken (so wie in den Beispielen dieses Kapitels), um die schlankere Klasse nicht mit kaum gebrauchten Typkonvertierungen vollzustopfen. Ist die Typumwandlung sicher, können Sie diese als implizite Umwandlung erstellen. Besteht die Möglichkeit eines Laufzeitfehlers, da der mögliche Wertebereich des Zieltyps kleiner ist als der des Quelltyps, benutzen Sie eine explizite Typkonvertierung.

16.5 Übungen

Ü

1. Erstellen Sie eine Struktur oder Klasse für die Speicherung von Bewertungspunkten von Kollegschülern von 0 bis 15 und Konvertierungsroutinen von und zu double-Werten. Dabei entspricht der double 6 – (punkte / 3).
2. Warum müssen Sie möglicherweise auch bei einer impliziten Typumwandlung den gewünschten Typ explizit angeben, da Sie andernfalls einen Compilefehler erhalten?

17 Windows Forms

Finden Sie es auch faszinierend, zu Beginn des 21. Jahrhunderts auf einem Rechner, mit dem Sie ein Dutzend Mondfähren landen könnten, Konsolenanwendungen mit zeilenweiser Textausgabe zu entwickeln? Ich nicht, darum werde ich jetzt damit aufhören (zumindest für dieses Kapitel). Schließlich bietet die .NET-Laufzeitumgebung mit den neuen Windows Forms so viele Neuerungen und Verbesserungen, dass einem die Windows-Programmierung wieder Spaß macht.

17.1 Einführung

Vor .NET hatten Sie sich mit der Wahl der Programmiersprache auch schon festgelegt, wie die Entwicklung der grafischen Oberfläche und die Programmierung der Fenster abläuft. Mit Visual Basic war die Erstellung einfach, allerdings konnten Sie nicht alle Möglichkeiten ausnutzen, die Windows zur Verfügung stellte. C++ war toll, jedes kleinste Bit der Formatflags für die Fenstererstellung hatten Sie unter Kontrolle. Mich überkam allerdings immer eine Welle von Mitgefühl, wenn ich einen C++-Entwickler beim Design der Menüs und Dialogboxen sah. Delphi wiederum war so eine Art Zwischending, hier waren die Formulare sogar nicht einmal eigenständige Dateien, sondern zum Teil Bestandteil des Codes. Ein Wechsel der Programmiersprache führte in der Regel dazu, dass man sich mit einer völlig neuen Art der GUI-Programmierung anzufreunden hatte: MFC, VB Forms, VCL und was Ihnen noch an netten Akronymen einfällt. Damit ist dank .NET jetzt Schluss.

Microsoft hat mit den Windows Forms ein neues, für alle Sprachen einheitliches Programmiermodell geschaffen. Ausgehend von den Erfahrungen der letzten zehn Jahre wurden viele Ecken und Kanten der Win32-Programmierung geglättet, Fehler bereinigt und alles in ein konsistentes, wirklich objektorientiertes Programmiermodell verpackt. Obwohl die .NET-Laufzeitumgebung um Größenordnungen umfangreicher ist als beispielsweise die MFC, werden Sie feststellen, dass die Erstellung von GUI-Anwendungen mit den Windows Forms einfach zu erlernen ist. Außerdem macht es wesentlich mehr Spaß, da Sie sich

auf das eigentliche Problem konzentrieren können und die .NET-Laufzeitumgebung Ihnen einen Großteil der Arbeit unter der Motorhaube abnimmt.

Bevor ich mit Ihnen 'fensterln' gehe, werfen wir noch einmal einen Blick auf den schematischen Aufbau der .NET-Laufzeitumgebung.

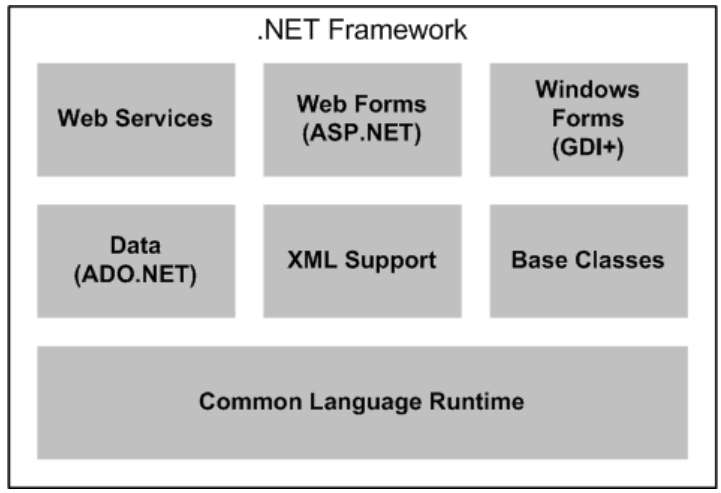


Abbildung 17.1: Der schematische Aufbau der .NET-Laufzeitumgebung

Wie Sie erkennen, existieren zwei Modelle für die Erstellung einer GUI-Anwendung: die Windows Forms und für Web-Anwendungen die WebForms. Obwohl Sie auch für Ihre Desktop-Anwendungen eine Browser-basierte Oberfläche mit den WebForms bauen können, bieten die Windows Forms immer noch mehr Möglichkeiten, während ASP.NET ohne die WebForms nicht funktioniert. In diesem Kapitel beschränke ich mich daher auf die Windows Forms, das Thema Web-Forms und ASP.NET werde ich im Kapitel über die Basisdienste der .NET-Laufzeitumgebung behandeln.

17.2 Das Programmiermodell

Die grundlegende Einheit einer GUI-Anwendung ist das Formular (form). Alle Fenster einer Anwendung werden als Formulare bezeichnet, auch wenn es sich z.B. um Dialogfenster handelt. Die entsprechende

Basisklasse, von der alle Formulare abstammen, heißt dementsprechend auch Form und ist im Namensraum `System.Windows.Forms` zu finden, der komplette Name lautet also `System.Windows.Forms.Form`. Daher finden Sie in den Beispielen auch immer u. a. das `using System.Windows.Forms` zu Beginn. In diesem Namensraum liegen auch die Klassen für die anderen Steuerelemente, von Textfeldern über Listboxen und Optionsfeldern bis hin zum Treeview.

Aber der Reihe nach. Zur Vorstellung der Konzepte möchte ich Ihnen eine minimale GUI-Anwendung mit Windows Forms vorstellen (Sie ahnen es sicher: »Hallo, Welt«, die 234. Version), danach werden wir Zug um Zug Funktionalität hinzufügen.

```
using System;
using System.Windows.Forms;
using System.Drawing;

namespace WinTest
{
    public class MeinFenster : Form
    {
        public MeinFenster()
        {
           this.Text = "Hallo, Welt!!";
        }

        static void Main(string[] args)
        {
            Application.Run( new MeinFenster() );
        }
    }
}
```

Listing 17.1: Hallo Welt als Anwendung für Windows Forms

Herzstück jeder Anwendung ist also eine Klasse, welche von der Klasse `System.Windows.Forms.Form` erbt, in diesem Beispiel die Klasse MeinFenster. Diesen Code können Sie nun mit dem C#-Compiler übersetzen und Sie bekommen eine ausführbare Datei, die nach dem Start das wundervolle Ergebnis anzeigt:

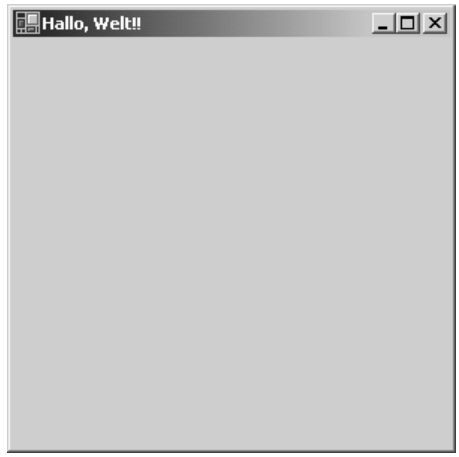


Abbildung 17.2: Das erste Fenster mit C# und .NET

Dies ist also der minimale Aufwand, den sie treiben müssen, um ein Fenster auf den Bildschirm zu bekommen. Möchten Sie diesen Code in Visual Studio.NET ausprobieren, müssen Sie allerdings etwas tricksen. Dies liegt nicht daran, dass ich etwas verschwiegen hätte, sondern an der Tatsache, dass einige Anwendungen in der Komplexität der Oberfläche doch unwesentlich über das hier gezeigte Beispiel hinausgehen. Daher legt VS.NET gleich ein passendes Grundgerüst an. Den Code im Listing oben können Sie übersetzen, indem Sie eine Konsolenanwendung als Projekt beginnen, dann Verweise auf die beiden Namensräume (DLLs) `System.Windows.Forms` und `System.Drawing` hinzufügen und in den Eigenschaften des Projekts den Ausgabetyp Windowsanwendung einstellen.

Wie umbauen, was einstellen? Hier Schritt für Schritt noch einmal die Unterschiede. Wenn Sie Visual Studio benutzen, werden Sie bei der Erstellung eines C#-Projekts (unter anderem) vor die Wahl zwischen einer Konsolenanwendung und einer Windows-Anwendung gestellt.

Entscheiden Sie sich für eine KONSOLENANWENDUNG, dann wird natürlich auch kein Namensraum für ein Fenstersystem importiert, das System legt eine leere Schablone an, die nur den Namensraum `System` benutzt und erzeugt eine Klasse mit der statischen Methode `Main`.

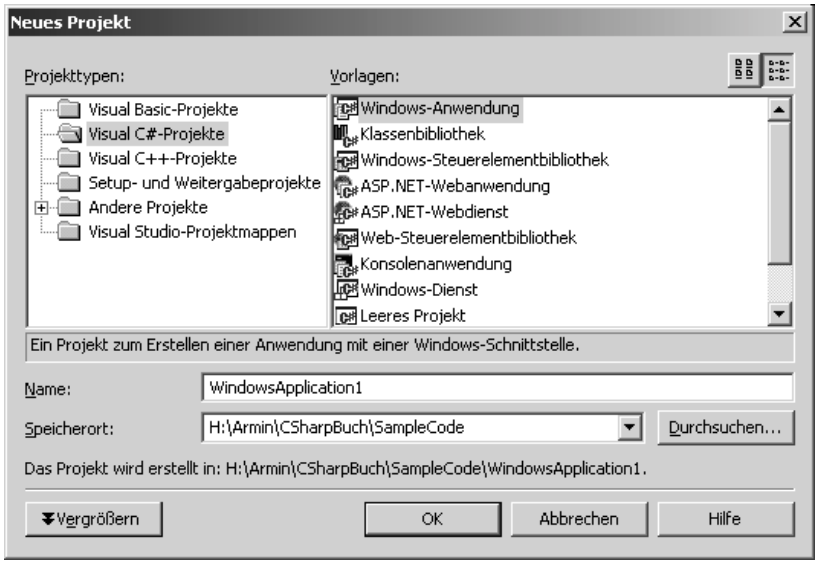


Abbildung 17.3: Der Auswahldialog für ein neues C#-Projekt

Wählen Sie dagegen den Punkt WINDOWS-ANWENDUNG, so erzeugt Visual Studio eine komplette Schablone für eine Windows Forms Anwendung mit den benötigten Namensräumen und der notwendigen Initialisierung der Steuerelemente. Dieses Grundgerüst ist etwas länger als unser WinForms-Hello.

```
// das grundgerüst einer windowsanwendung
// in visual studio.NET

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace WindowsForms1
{
   public class Form1 : System.Windows.Forms.Form
   {
      // Erforderliche Designervariable.
      private System.ComponentModel.Container
```

```
        components = null;

public Form1()
{
   // Erforderlich für die Windows Form
   // Designerunterstützung
   InitializeComponent();

   // TODO: Fügen Sie den Konstruktorcode
   // nach dem Aufruf von InitializeComponent
   // hinzu
}

// Die verwendeten Ressourcen bereinigen.
protected override void Dispose( bool disposing )
{
   if( disposing )
   {
      if (components != null)
      {
         components.Dispose();
      }
   }
   base.Dispose( disposing );
}

#region Windows Form Designer generated code
// Erforderliche Methode für die
// Designerunterstützung.
// Der Inhalt der Methode darf nicht mit dem
// Code-Editor geändert werden.
private void InitializeComponent()
{
   this.components =
        new System.ComponentModel.Container();
   this.Size = new System.Drawing.Size(300,300);
   this.Text = "Form1";
}
#endregion

// Der Haupteinstiegspunkt für die Anwendung.
[STAThread]
static void Main()
{
```

```
            Application.Run(new Form1());
        }
    }
}
```

Listing 17.2: Automatisch erzeugtes Grundgerüst einer Windows-Anwendung

Dieser Code ist alles, was Sie benötigen. Das führt auch sofort zur nächsten Frage.

17.2.1 Wo bleibt die Formulardatei?

Falls Sie bisher mit Visual Basic, Delphi oder C++ entwickelt haben, sind Sie sicher erstaunt, dass C# hier mit einer einzelnen Quellcode-Datei auskommt. Wo bleibt die Datei mit der Formulardefinition? Die Antwort ist, es gibt keine! Die gesamten Steuerelemente des Formulars und die Definition des Layouts liegen im Quellcode. Erinnern Sie sich noch an die Sightseeing-Tour zu Beginn des Buches? Darin hatte ich als einen der Vorteile von C# aufgeführt, dass diese Sprache ein One-Stop-Coding erlaubt, keine Headerdateien, kein IDL und natürlich auch keine externen Formulardateien. Wie ich später noch zeigen werde, finden Sie alles in Ihrem Quellcode (wenn auch von VS.NET gut versteckt). Wem das bekannt vorkommt, der hat sehr wahrscheinlich vorher schon einmal mit Borlands Delphi entwickelt. Das Programmiermodell der Windows Forms und die Ablage im Quelltext ist sozusagen die nächste Evolutionsstufe des Delphi-Modells (bei welcher Firma war der C#-Designer vor Microsoft?). In den nächsten Abschnitten erfahren Sie, welcher Teil des Quelltextes für die Definition des Formulars zuständig ist.

17.2.2 Läuft und läuft und läuft – Application.Run

Alte Windows-Programmierer (Leser von Charles Petzolds Standardwerk zu Beginn der 90er) werden jetzt immer noch suchenden Auges über den Quelltext irren – wo bleibt die Behandlung der Windows-Botschaften, was ist mit WM_PAINT und den ganzen Sachen? Zur Freude der VB-Umsteiger bleibt Ihnen und mir das erspart. Ich hatte ja bereits festgestellt, dass die .NET-Laufzeitumgebung einem die lästigen Details vom Halse hält, dies trifft auch auf die Nachrichtenschleife zu. Dafür ist die in Main angegebene statische Methode Run der Klasse Application zuständig. Diese bekommt als Parameter eine neue Instanz

des Hauptformulars und kümmert sich um die Bürokratie. Wir können uns den Dingen zuwenden, die mehr Spaß machen, z.B. auf eines dieser Ereignisse reagieren. Damit Sie sehen, wie einfach dies ist (ich weiß, für VB-Entwickler war es schon immer ähnlich einfach, aber denken Sie an die armen C-Entwickler!), werden wir nur bei jedem Neuzeichnen des Formulars (der berühmten WM_PAINT-Nachricht) etwas zeichnen und einen Text ausgeben. Nachfolgend finden Sie den geänderten Code.

```
using System;
using System.Windows.Forms;
using System.Drawing;

namespace WinTest
{
   public class MeinFenster : Form
   {
      public MeinFenster()
      {
         this.Text = "Hallo, Welt!!";
      }

      protected override
      void OnPaint(PaintEventArgs pe)
      {
         // Fensterrechteck füllen
         pe.Graphics.FillRectangle(
                         new SolidBrush( Color.Azure ),
                         this.ClientRectangle);

         // etwas Text darüber
         Font f = new Font("Arial",
                           36,
                           GraphicsUnit.Point);

         pe.Graphics.DrawString("Hallo, Welt",
                           f,
                           new SolidBrush(Color.Red),
                           20,
                           20);
      }

      static void Main(string[] args)
```

```
    {
        Application.Run( new MeinFenster() );
    }
 }

}
```

Listing 17.3: Eine eigene Routine für das OnPaint-Ereignis

Alles, was ich tun muss, um bei jedem Neuzeichnen meine Methode aufzurufen, ist ein Überschreiben der Methode OnPaint der Form-Klasse. Die ganze Arbeit der Umsetzung von WM_PAINT in das OnPaint-Ereignis, die Nachrichtenschleife, das Verteilen, alles dies passiert im Hintergrund, ohne dass ich mich darum kümmern müsste.



Abbildung 17.4: Die Ausgabe des neuen Quelltextes

Für alles, was Sie im Normalfall an Ereignissen eines Formulars behandeln müssten, existiert eine solche virtuelle Methode, die Sie bei Bedarf einfach überschreiben. Aber die .NET-Laufzeitumgebung wäre nicht perfekt, wenn es im Bedarfsfall nicht auch noch eine Möglichkeit geben würde, direkt in die Fensterfunktion einzugreifen und das Handling der Windows-Nachrichten selbst zu übernehmen. Aber das würde an dieser Stelle zu weit führen.

17.3 Steuerelemente

17.3.1 Steuerelemente hinzufügen

Natürlich wird eine Windows-Anwendung erst dann richtig brauchbar, wenn eine gewisse Funktionalität über Steuerelemente hergestellt

wurde (es sei denn, Sie bevorzugen den 'Unix way of life': eine Eingabezeile mit einem einsamen '>' zu Beginn). Diese Steuerelemente werden als Objektvariablen ebenfalls in den Quellcode übernommen. Ich höre schon beinahe Ihren Einwand: 'Aber die Einstellung der Eigenschaften muss doch nun in eine eigene Datei?!'. Sehen Sie sich den Quelltext zu unserem allerersten Fensterprogramm an. Der Text des Fensters wird hier im Konstruktor über in eine direkte Zuweisung gesetzt. Genau so wird jede Einstellung einer Eigenschaft eines Steuerelements als Zuweisung in C#-Code (bzw. jeder anderen Programmiersprache) vom Form Designer erstellt.

Wie die .NET-Laufzeitumgebung dies handhabt, möchte ich aber nun an einem 'richtigen' GUI-Projekt erläutern, da sich hier schön die Abläufe beim Start einer Formular-basierten Anwendung zeigen lassen. Falls Sie lieber ohne Visual Studio arbeiten, sehen Sie dann gleich, was Sie in diesem Falle manuell ergänzen müssen (es gibt Momente, in denen eine IDE richtig nützlich ist).

Ich habe in VS.NET eine neue C#-Windows-Anwendung erzeugt und auf dem Formular mit dem Namen `frmMain` eine Schaltfläche mit dem Namen `btnExit` abgelegt. Die Schaltfläche wurde nur abgelegt, ich habe noch keine Zuweisung für das Click-Ereignis gemacht, dies werde ich später nachholen.

Der Code, den VS-Net erzeugt, finden Sie im nachfolgenden Listing, welches ich nun Stück für Stück durchgehen werde.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;



namespace WinTest2
{
    public class frmMain : System.Windows.Forms.Form
    {
        private System.Windows.Forms.Button btnExit;
```

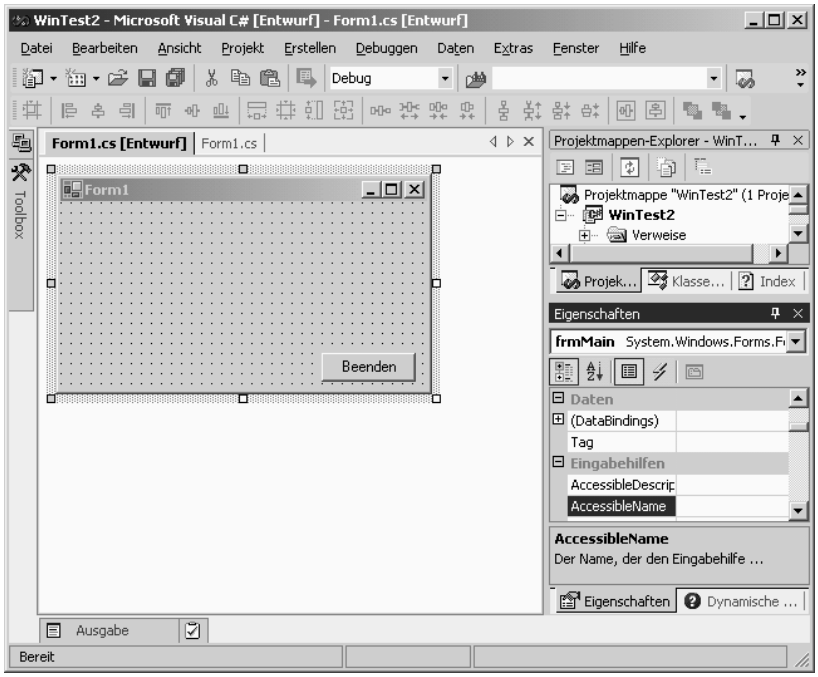


Abbildung 17.5: Ein Formular mit einer Schaltfläche in der IDE

Hier wird für das Steuerelement ein privates Feld deklariert, welches später die Objektreferenz für den Button aufnimmt.

```
/// <summary>
/// Erforderliche Designervariable.
/// </summary>
private System.ComponentModel.Container
        components = null;
```

Ein privates Feld, das der Formular-Designer benötigt

```
public frmMain()
{
    //
    // Erforderlich für die Windows
    // Form-Designerunterstützung
    //
    InitializeComponent();
```

Diesen Aufruf benötigt sowohl der Formular-Designer als auch die Laufzeitumgebung, um die einzelnen Steuerelemente des Formulars zu erzeugen und die Eigenschaften einzustellen

```
        //
        // TODO: Fügen Sie den Konstruktorcode nach
        // dem Aufruf von InitializeComponent hinzu
        //
    }


    /// <summary>
    /// Die verwendeten Ressourcen bereinigen.
    /// </summary>
    protected override
    void Dispose( bool disposing )
    {
        if( disposing )
        {
            if (components != null)
            {
                components.Dispose();
            }
        }
        base.Dispose( disposing );
    }
```

Eine Methode, die vom Formularmanager bei der Freigabe des Formulars aufgerufen wird.

```
    #region Windows Form Designer generated code
    /// <summary>
    /// Erforderliche Methode für die
    /// Designerunterstützung.
    /// Der Inhalt der Methode darf nicht mit dem
    /// Code-Editor geändert werden.
    /// </summary>
    private void InitializeComponent()
    {
        this.btnExit =
             new System.Windows.Forms.Button();
        this.SuspendLayout();
        //
        // btnExit
```

```
            //
            this.btnExit.Location =
                 new System.Drawing.Point(208, 120);
            this.btnExit.Name = "btnExit";
            this.btnExit.TabIndex = 0;
            this.btnExit.Text = "Beenden";
            //
            // frmMain
            //
            this.AutoScaleBaseSize =
                 new System.Drawing.Size(5, 13);
            this.ClientSize =
                 new System.Drawing.Size(292, 149);
            this.Controls.AddRange(
                 new System.Windows.Forms.Control[]
                     { this.btnExit} );
            this.Name = "frmMain";
            this.Text = "Form1";
            this.ResumeLayout(false);

        }
        #endregion
```

Hier werden die einzelnen Steuerelemente bei der Erzeugung des Formulars ebenfalls erstellt und mit den richtigen Eigenschaftswerten versehen. Entgegen dem Kommentar dürfen Sie hier schon ändern, wenn Sie wissen, was Sie dabei tun. Für den Großteil der Entwickler kann diese Region im Code-Editor aber zugeklappt bleiben.

```
        /// <summary>
        /// Der Haupteinstiegspunkt für die Anwendung.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.Run(new frmMain());
        }
    }
}
```

Listing 17.4: Der komplette Code für Formular und Schaltfläche

Die statische Methode Application.Run wird mit einer neuen Instanz von frmMain (unserem Hauptformular) aufgerufen und versorgt dieses mit den entsprechenden Windows-Nachrichten.

Was passiert nun in dieser Anwendung? Zuerst einmal werden einige Felder und Methoden erzeugt, damit der Formular-Designer unterstützt wird. Wir wollen ja nicht (obwohl das möglich wäre) jedes Formular von Hand malen.

Im Konstruktor für unser Formular wird nun die Methode InitializeComponent() aufgerufen, welche die einzelnen Steuerelemente als Instanzen erzeugt und deren Eigenschaft auf die im Designer eingestellten Werte setzt. Jedes Formular besitzt eine Auflistung namens Controls, in der die einzelnen Steuerelemente abgelegt sind. Ich werde darauf später beim Thema der dynamischen Erzeugung von Steuerelementen zurückkommen.

17.3.2 Ereignisbehandlung für Steuerelemente

Die Behandlung von Ereignissen für Steuerelemente folgt genau dem Schema, welches ich in Kapitel 20 ***Delegates und Events*** bereits vorgestellt habe. Wir erstellen eine Methode, die als Callback beim Eintreten des Ereignisses von der :Net-Laufzeitumgebung aufgerufen werden soll und hängen diese in die Kette der Ereignisbehandlungsroutinen ein.

Wie Sie vielleicht noch wissen, folgt die Parameterliste einer solchen Ereignisbehandlungsroutine einer bestimmten Konvention: Der erste Parameter ist vom Typ `object` und enthält das ereignisauslösende Steuerelement, der zweite Parameter ist eine Instanz der Klasse `EventArgs` oder davon abgeleitet. Als bahnbrechende, neuartige Funktion soll die Anwendung beim Klick auf die Schaltfläche beendet werden (haben Sie das nicht schon anhand des Namens vermutet?). Die entsprechende Methode der Formularklasse muss also folgendermaßen aussehen:

```
protected void btnExit_Click(object sender,
                             System.EventArgs e)
{
   Application.Exit();
}
```

Listing 17.5: Die Ereignisbehandlung für den Klick auf die Schaltfläche

Der Aufruf der Methode `Exit()` hält nicht wirklich die Anwendung an, sondern sorgt dafür, dass jede Nachrichtenschleife geleert und dann eine Rückkehr der Methode Run ausgelöst wird. Sozusagen die offizielle und brave Methode, eine Anwendung zu beenden.

Für den Designer in Visual Studio ist ebenfalls nur die Signatur der Methode ausschlaggebend, um in der Liste der möglichen Ereignisbehandlungsroutinen aufgelistet zu werden. Ich habe z.B. im Code zusätzlich zwei leere Methoden mit der gleichen Signatur deklariert:

```
protected void btnExit_Click(object sender, System.EventArgs e)
{
   Application.Exit();
}

protected void MachtNix(object sender, System.EventArgs e)
{
}

protected void MachtAuchNix(object sender, System.EventArgs e)
{
}
```

Listing 17.6: Etwas Leercode, damit der Designer was zum Anzeigen hat

Wenn ich nun in VS.NET im Designer bei den Ereignissen für die Schaltfläche die Liste aufklappe, erhalte ich folgende Ausgabe (Abbildung 17.6).

Es ist ebenfalls unerheblich, ob Sie hier zuerst einen Namen eintragen und die Eingabetaste drücken oder zuerst die Methoden programmieren und dann hier zuweisen. Notfalls generiert der Designer den entsprechenden Code in der Klasse. Jetzt muss diese Routine noch mit dem entsprechenden Ereignis verbunden werden. Auch dies erledigt der Designer bei der Zuweisung für uns. Sie wissen allerdings aus dem Kapitel über die Delegates und Ereignisse bereits, wie dieser Code aussehen wird:

```
this.btnExit.Click +=
    new System.EventHandler(this.btnExit_Click);
```

Listing 17.7: Zuweisung an den Delegate

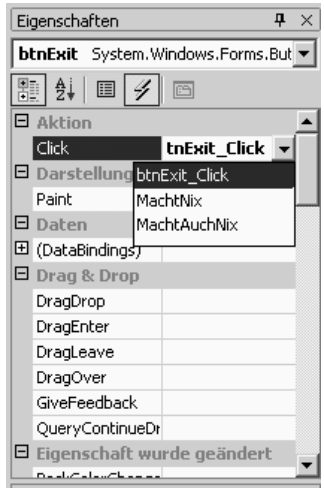


Abbildung 17.6: Die Auswahl der möglichen Event-Handler

Der Bezeichner this.btnExit.Click ist ein Delegate, den wir mit unserer Methode belegen und über den Operator += in die Kette einhängen.

17.4 Wichtige Klassen

Bei der Größe der .NET-Laufzeitumgebung verliert man leicht den Überblick, welche Klassen wofür wichtig sind. Hier eine Tabelle mit den wichtigsten Klassen für die Erstellung von GUI-Anwendungen:

Klasse	Namensraum	Bedeutung
Application	System.Windows.Forms	Die Anwendungsklasse
Form	System.Windows.Forms	Repräsentiert ein Formular
Font	System.Drawing	Klasse für Schriftart-Objekte
Graphics	System.Drawing	Die Ausgabefläche, ähnlich dem Device Context in Win32
MessageBox	System.Windows.Forms	Die Klasse für die Messageboxen
MainMenu	System.Windows.Forms	Ein Hauptmenü
MenuItem	System.Windows.Forms	Ein Menüpunkt

Tabelle 17.1: Einige der wichtigen Klassen für GUI-Anwendungen

17.4.1 MessageBoxen einst und jetzt

Die größte Umstellung für viele Entwickler dürfte bei den immer wieder gern benutzten MessageBoxen die Tatsache sein, dass es sich hier um eine Klasse handelt. Allerdings haben die Entwickler bei Microsoft dankenswerterweise die Methode `Show` als statische Methode angelegt, so dass wir nicht erst eine Instanz zu erzeugen brauchen.

Die einzelnen, überladenen Versionen von Show decken so ziemlich den gleichen Bereich ab, den es auch in der bisherigen Windows-Programmierung gab. Sie müssen sich also nicht großartig umstellen. Als kleines Beispiel habe ich der Methode `btnExit_Click()` noch eine Sicherheitsabfrage hinzugefügt, ob die Anwendung wirklich beendet werden soll.

```
protected void btnExit_Click(object sender,
                             System.EventArgs e)
{
   if (DialogResult.Yes == MessageBox.Show(
                             "Wirklich beenden?",
                             "Hinweis",
                             MessageBoxButtons.YesNo,
                             MessageBoxIcon.Asterisk))
      Application.Exit();
}
```

Listing 17.8: Eine MessageBox ist eine MessageBox ist eine MessageBox …

17.4.2 Menüs

Die Erstellung bzw. Behandlung von Menüs ist ebenfalls mit den neuen Windows Forms sehr konsistent geraten (und alle C++-Umsteiger dürfen sich über einen vernünftigen Menü-Editor freuen, mit dem sich endlich sinnvoll arbeiten lässt). Jedes Formular besitzt ein Hauptmenü, welche über die Eigenschaft Menu zugewiesen wird. Unterhalb dieses Hauptmenüs finden sich dann die einzelnen Menüpunkte, welche Instanzen der Klasse MenuItem darstellen. Natürlich können Sie kaskadierende Menüs (Untermenüs) bauen, da jedes MenuItem eine eigene Auflistung MenuItems besitzt, in der Sie die entsprechenden SubMenüs anlegen.

Über den Designer geschieht dies wieder mit wenigen Mausklicks, aber wir wollen natürlich wissen, was hinter den Kulissen passiert. Deshalb hier noch mal der Code, wobei ich die vom Designer für die Menüs eingefügten Stellen fett markiert habe:

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace WinTest2
{
   /// <summary>
   /// Zusammenfassungsbeschreibung für Form1.
   /// </summary>
   public class frmMain : System.Windows.Forms.Form
   {
      private System.Windows.Forms.Button btnExit;
      private System.Windows.Forms.MainMenu mnuMain;
      private System.Windows.Forms.MenuItem mnuDatei;
      private System.Windows.Forms.MenuItem mnuExit;
      private System.Windows.Forms.MenuItem mnuInfo;

      /// <summary>
      /// Erforderliche Designervariable.
      /// </summary>
      private System.ComponentModel.Container
              components = null;

      public frmMain()
      {
         //
         // Erforderlich für die Windows
         // Form-Designerunterstützung
         //
         InitializeComponent();

         //
         // TODO: Fügen Sie den Konstruktorcode nach
         // dem Aufruf von InitializeComponent hinzu
         //
      }
```

```
/// <summary>
/// Die verwendeten Ressourcen bereinigen.
/// </summary>
protected override void Dispose( bool disposing )
{
   if( disposing )
   {
      if (components != null)
      {
         components.Dispose();
      }
   }
   base.Dispose( disposing );
}

#region Windows Form Designer generated code
/// <summary>
/// Erforderliche Methode für die Designer-
/// unterstützung.
/// Der Inhalt der Methode darf nicht mit dem
///  Code-Editor geändert werden.
/// </summary>
private void InitializeComponent()
{
   this.btnExit =
        new System.Windows.Forms.Button();
   this.mnuMain =
        new System.Windows.Forms.MainMenu();
   this.mnuDatei =
        new System.Windows.Forms.MenuItem();
   this.mnuExit =
        new System.Windows.Forms.MenuItem();
   this.mnuInfo =
        new System.Windows.Forms.MenuItem();
   this.SuspendLayout();
   //
   // btnExit
   //
   this.btnExit.Location =
        new System.Drawing.Point(208, 120);
   this.btnExit.Name = "btnExit";
   this.btnExit.TabIndex = 0;
   this.btnExit.Text = "Beenden";
   this.btnExit.Click +=
     new System.EventHandler(this.btnExit_Click);
```

```
//
// mnuMain
//
this.mnuMain.MenuItems.AddRange(
     new System.Windows.Forms.MenuItem[]
     { this.mnuDatei} );

//
// mnuDatei
//
this.mnuDatei.Index = 0;
this.mnuDatei.MenuItems.AddRange(
     new System.Windows.Forms.MenuItem[]
     { this.mnuInfo,
       this.mnuExit} );

this.mnuDatei.Text = "Datei";

//
// mnuExit
//
this.mnuExit.Index = 1;
this.mnuExit.Text = "Beenden ...";
this.mnuExit.Click +=
  new System.EventHandler(this.btnExit_Click);

//
// mnuInfo
//
this.mnuInfo.Index = 0;
this.mnuInfo.Text = "Info ...";
this.mnuInfo.Click +=
   new System.EventHandler(this.ShowInfo);

//
// frmMain
//
this.AutoScaleBaseSize =
     new System.Drawing.Size(5, 13);
this.ClientSize =
     new System.Drawing.Size(292, 149);
this.Controls.AddRange(
     new System.Windows.Forms.Control[]
     { this.btnExit} );
```

```
            this.Menu = this.mnuMain;

            this.Name = "frmMain";
            this.Text = "Form1";
            this.ResumeLayout(false);

        }
        #endregion

        /// <summary>
        /// Der Haupteinstiegspunkt für die Anwendung.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.Run(new frmMain());
        }

        private void btnExit_Click(object sender,
                                   System.EventArgs e)
        {
            if (DialogResult.Yes == MessageBox.Show(
                                "Wirklich beenden?",
                                "Hinweis",
                                MessageBoxButtons.YesNo,
                                MessageBoxIcon.Asterisk) )
              Application.Exit();
        }

        private void ShowInfo(object sender,
                                System.EventArgs e)
        {
            MessageBox.Show("FormsDemo aus Goto C#");
        }

    }
}
```

Listing 17.9: Die Anwendung bekommt ein Menü

Hier noch einmal das Formular mit dem geöffneten Menüdesigner:

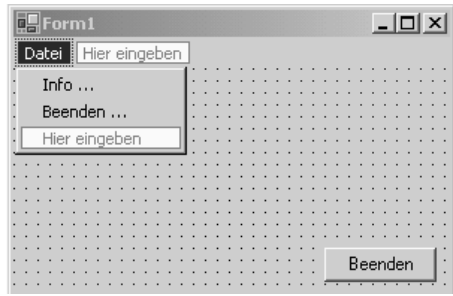


Abbildung 17.7: Die IDE mit dem geöffneten Menü-Designer

Damit besitzt unsere Windows-Anwendung schon ein Menü mit einem Hauptpunkt und darunter zwei Menüpunkten.

17.4.3 Bestehende Ereignisse zuweisen

Sehr praktisch ist die Konvention, alle Aufrufe einer Ereignisbehandlung gleich aussehen zu lassen (Sie erinnern sich, die beiden Parameter `sender` und `EventArgs`). Damit kann ich nun ohne Programmieraufwand dem Menüpunkt BEENDEN ... die Methode `btnExit_Click()` zuweisen, da die Signaturen übereinstimmen. Auf diese Art und Weise schreibe ich Methoden für Benutzeraktionen nur einmal und weise diese dann einer Schaltfläche, einem Symbol oder einem Menüpunkt zu, ohne mich zu wiederholen.

17.5 Komponenten zur Laufzeit erzeugen

17.5.1 Steuerelemente

Um zur Laufzeit programmgesteuert ein neues Steuerelement zu erzeugen, müssen wir nur dem bereits bekannten Schema folgen. Zur dynamischen Erzeugung eines Textfeldes brauche ich zuerst einmal eine Schaltfläche. Wird auf diese Schaltfläche geklickt, soll ein Textfeld dynamisch erzeugt werden, ist dieses Textfeld bereits vorhanden, soll es wieder entfernt werden. Ich nenne die Schaltfläche `btnTextfeld` und erzeuge für das `Click`-Ereignis die folgende Methode (die Referenz auf

das Textfeld lege ich in einem `private`-Feld der Klasse mit dem Namen `txt` ab).

```
private void btnTextfeld_Click(object sender,
                               System.EventArgs e)
{
   Textbox txt;
   // Textfeld-Objekt ist noch nicht vorhanden
   if (txt == null)
   {
      // Objekt erstellen und einige
      // Eigenschaften setzen
      txt = new TextBox();
      txt.Left = 20;
      txt.Top = 20;
      txt.Width = 120;
      txt.Text = "Vorgabetext";
      txt.Name = "txtBeispiel";

      // jetzt der auflistung der
      // formular-steuerelemente hinzufügen
      this.Controls.Add(txt);
   }
   else
   {
      // control entfernen
      this.Controls.Remove(txt);
      // referenz wieder auf null setzen
      txt = null;
   }
}
```

Listing 17.10: Dynamische Erzeugung von Steuerelementen

Wirklich einfach, nicht wahr? Ebenso können Sie zur Laufzeit natürlich auch Menüpunkte erstellen.

17.5.2 Menüs

Die dynamische Erstellung von Menüpunkten ist ebenso einfach. Beinahe der einzige Unterschied beschränkt sich darauf, dass beim Konstruktor eines Menüpunktes normalerweise auch eine Ereignisbehandlungsroutine (eine Instanz der Klasse `EventHandler`) mit übergeben

wird, damit die Auswahl des Menüpunktes auch etwas bewirkt. Um zu unserem Dateimenü einen neuen Punkt hinzuzufügen, reichen zwei Zeilen Code aus:

```
MenuItem mi = new MenuItem("Textfeld leeren",
                  new EventHandler(TextfeldLeeren));
mnuDatei.MenuItems.Add(0, mi);
```

Listing 17.11: Ein dynamisch erzeugter Menüpunkt

Schon haben wir dem Datei-Menü einen neuen Menüpunkt hinzugefügt. Damit der auch etwas sinnvolles erledigt, soll dieser Menüpunkt das dynamisch erzeugte Textfeld leeren, aber natürlich nur dann, wenn dieses Textfeld auch gerade vorhanden ist. Abhängig vom Zustand der Applikation ändert sich damit unser Datei-Menü.

Sie werden mir verzeihen, wenn ich diesen Punkt im Datei-Menü unterbringe. Ich hatte einfach keine Lust, für eine einzige Menüoption ein neues Bearbeiten-Menü zu erzeugen …

Dazu habe ich noch eine Methode erstellt, welche das Textfeld leert. Zu guter Letzt und um das Beispiel zu vervollständigen, wird in der Methode, die das Textfeld wieder vom Formular entfernt, auch wieder der Menüpunkt aus dem Menü gelöscht. Hier der komplette Code für die geänderte Methode `btnTextfeld_Click()`.

```
private void btnTextfeld_Click(object sender,
                               System.EventArgs e)
{
   if (txt == null)
   {
      txt = new TextBox();
      txt.Left = 20;
      txt.Top = 20;
      txt.Width = 120;
      txt.Text = "Vorgabetext";
      txt.Name = "txtBeispiel";
      this.Controls.Add(txt);

      // menüpunkt dynamisch einsetzen
      MenuItem mi = new MenuItem("Textfeld leeren",
                  new EventHandler(TextfeldLeeren));
      mnuDatei.MenuItems.Add(0, mi);
```

```
    }
    else
    {
        this.Controls.Remove(txt);
        txt = null;

        // menüpunkt wieder entfernen
        mnuDatei.MenuItems.RemoveAt(0);
    }
}
```

Listing 17.12: Erzeugen und Löschen von Menüpunkten

17.6 Zusammenfassung

Dieses Kapitel konnte natürlich nur einen kleinen Streifzug durch die Welt der Windows Forms bieten. Hauptzweck war die Verdeutlichung der neuen Prinzipien zur Erstellung von GUI-Anwendungen in C#:

- Formulare sind im Quelltext enthalten und werden nicht mehr in einer eigenen Datei abgelegt (one-stop-coding).
- Steuerelemente werden zum Zeitpunkt der Erzeugung einer Instanz der Formularklasse in einen Container eingehängt und dem Formular hinzugefügt.
- Über Ereignisbehandlungsroutinen und Delegates werden die Ereignissen mit dem Code zur Behandlung dieser Ereignisse verknüpft.
- Alle Steuerelemente besitzen einen wesentlich konsistenteren Satz an Methoden und Eigenschaften, was gegenüber den alten Windows-Steuerelementen eine deutliche Verbesserung darstellt.

Das neue Grafiksubsystem GDI+ bietet neben einer besseren Farbverwaltung mit Transparenz auch viele neue Möglichkeiten zur Transformation und Darstellung von grafischen Informationen, die über den Rahmen eines C#-Buches weit hinaus gehen.

17.7 Übung



1. Auch hier wieder ein kleines Projekt: Erstellen Sie eine Windows Forms-Anwendung, mit der über Schieberegler für eine Farbfläche der Rot-, Grün- und Blauanteil eingestellt werden kann. Dies soll für jeden Kanal nur im Vielfachen von 51 möglich sein und die kombinierte RGB-Angabe soll als Hexwert in der Form #rrggbb (wie in HTML-Seiten) in einer Textbox nach jeder Änderung angezeigt werden.

18 Indexer und Auflistungen

18.1 Indexer

Indexer sind ein neues Konzept von C#, mit dessen Hilfe Sie eine Klasse oder eine Struktur wie ein Array ansprechen können. Manchmal ist es nämlich intuitiver und lesbarer, anstelle des Zugriffs auf die Eigenschaft einer Klasse einen Index verwenden zu können.

Eine Stringklasse `String` speichert einen Text, der wiederum aus einzelnen Buchstaben besteht. Diese Buchstaben werden in einem Array gespeichert, damit Sie Zugriff auf jedes Zeichen des Strings haben. Angenommen, dieses Array trägt den Namen `Characters`, dann können Sie auf ein einzelnes Zeichen nur über `String.Characters[x]` zugreifen. Intuitiver ist aber `String[x]`. Genau diese Funktionalität liefert ein Indexer: Sie können über den Namen des Objektes bzw. der Klasse und einen Index auf Informationen in der Klasse zugreifen. Funktional gesehen ähnelt ein Indexer also einer Eigenschaft. Dieser Index muss allerdings keineswegs nummerisch sein, da auch Indexer überladen werden können. Allerdings wird jede Version des Indexers mit einem Datentyp deklariert, so dass alle Zugriffe über einen Indexdatentyp den gleichen Datentyp als Ergebnis liefern (das macht die Sache auch konsistent).

18.1.1 Deklaration eines Indexers

Die allgemeine Form eines Indexers sieht so aus:

```
public class MitIndexer
{
   public <datentyp> this[ <datentyp> <bezeichner> ]
   {
      get
      {
         // code zum lesen
      }

      set
      {
```

```
            // code zum schreiben
        }
    }
}
```

Listing 18.1: Allgemeine Deklaration eines Indexers

Da ja die Instanz der Klasse wie ein Array behandelt wird, wird der Indexer folgerichtig über das Schlüsselwort `this` deklariert. Wie bereits erwähnt, können Sie jederzeit weitere, überladene Versionen mit zusätzlichen Signaturen (Anzahl und Typ der Parameter) erstellen.

Damit können Sie in einer Klasse für Wettkampfergebnisse über einen Index z.B. den Namen des an dieser Stelle platzierten Teilnehmers erhalten.

```
class TopTen
{
   string[] rang = new string[10];

   public string this[int index]
   {
      get
      {
         if ((index > 10) || (index < 1))
            throw new ArgumentOutOfRangeException(
                 "Nur Plätze von 1 bis 10 möglich!");
         else
              return rang[index-1];
      }

      set
      {
         if ((index > 10) || (index < 1))
            throw new ArgumentOutOfRangeException(
                 "Nur Plätze von 1 bis 10 möglich!");
         else
            rang[index-1] = value;
      }
   }
}
```

Listing 18.2: Ein Indexer für einen nummerischen Index

Sehr schön wäre es jetzt, wenn ich einfach als Index einen Namen übergeben und als Ergebnis eine Zahl mit der Platzierung erhalten könnte. Genau dies funktioniert auch, ich muss dazu nur den Indexer entsprechend überladen.

```
class TopTen
{
   string[] rang = new string[10];

   public string this[int index]
   {
      get
      {
         if ((index > 10) || (index < 1))
            throw new ArgumentOutOfRangeException(
                 "Nur Plätze von 1 bis 10 möglich!");
         else
              return rang[index-1];
      }

      set
      {
         if ((index > 10) || (index < 1))
            throw new ArgumentOutOfRangeException(
                 "Nur Plätze von 1 bis 10 möglich!");
         else
            rang[index-1] = value;
      }
   }

   // die neue version mit string-index
   public int this[string name]
   {
      get
      {
         int p = Array.IndexOf(rang, name);
         if (p >= rang.GetLowerBound(0))
            return p+1;
         else
            throw new ArgumentException(
                     "Keine Platzierung für " + name);
      }

      set
```

```
      {
         if((value >= 1) && (value <= 10))
            rang[value-1] = name;
         else
            throw new ArgumentOutOfRangeException(
                        "Nur Plätze von 1 bis 10!");
      }
   }
}
```

Listing 18.3: Ein zusätzlicher Indexer mit einem String-Index

Zur Überprüfung der Beschränkung auf die ersten zehn Plätze (schließlich heißt die Klasse ja auch `TopTen`) dient jeweils ein Vergleich des Indexwertes. Das Addieren bzw. Subtrahieren von `1` ist nötig, da in C# alle Arrays immer mit dem Index `0` beginnen, wir aber Platzierungen von `1` bis `10` vergeben. Bei einer Verletzung des gültigen Index löse ich die passende Exception aus.

Der Konstruktor für `ArgumentOutOfRangeException` ist überladen, so dass Sie bei Bedarf auch den Namen des Parameters übergeben können. Werfen Sie einfach einen Blick in die Online-Hilfe …

18.1.2 Zugriff über einen Indexer

Der Zugriff erfolgt nun für die Instanz der Klasse wie über ein Array mit dem entsprechenden nummerischen Index oder über einen String-Index.

```
class TestClass
{
   static void Main(string[] args)
   {
      TopTen tt = new TopTen();

      tt[1] = "Benzino Napoloni";
      tt[2] = "Karl Bleifuss";
      tt[3] = "Emilio Commodo";

      Console.WriteLine("Großer Preis von C#");

      for(int i = 1; i <=10; i++)
         Console.WriteLine("Platz {0} = {1}", i, tt[i]);
```

```
      string driver = "Karl Bleifuss";

      Console.WriteLine("--------------------------");
      Console.WriteLine("{0} ist auf Platz {1}",
                        driver, tt[driver] );
   }
}
```

Listing 18.4: Testen der beiden Indexer

So einfach ist der Zugriff auf eine Klasse mit einer Array-ähnlichen Notation. Bei der Ausgabe sehen Sie anhand der Plätze vier bis zehn wieder, dass die .NET-Laufzeitumgebung alle Variablen auf 0 (bzw. einen leeren String) initialisiert.

Hier die Bildschirmausgabe des Programms:

```
Großer Preis von C#
Platz 1 = Benzino Napoloni
Platz 2 = Karl Bleifuss
Platz 3 = Emilio Commodo
Platz 4 =
Platz 5 =
Platz 6 =
Platz 7 =
Platz 8 =
Platz 9 =
Platz 10 =
--------------------------
Karl Bleifuss ist auf Platz 2
```

18.1.3 Designüberlegungen

Es ist ohne weiteres möglich, als Ergebnis eines Indexers komplexe Datentypen zurückzuliefern, beispielsweise eine Struktur oder ein Array. Als Beispiel für einen möglichen Einsatz hier eine Klasse, welche die Beziehung zwischen Sternzeichen und den entsprechenden Monaten implementiert.

Diese Klasse dient mehreren Zwecken. Einmal der Demonstration, wie sich mit einem Indexer auch Arrays zurückliefern lassen, zum zweiten als Beispiel für die Implementierung eines Nur-Lese-Indexers und

damit drittens auch gleich als Negativ-Beispiel, wie man es nicht machen sollte. Warum, erläutere ich Ihnen nach dem Listing.

Da nur eine `get`-Zugriffsanweisung vorhanden ist, kann dieser Indexer nur gelesen werden. Der Zugriff erfolgt über einen Indexer entweder über die Monatszahl von 1 bis 12 (liefert ein Array mit den Sternzeichennamen) oder über einen String mit dem Namen des Sternzeichens (liefert ein Array mit den entsprechenden Monaten).

Da dies hier kein Astrologie-Kurs ist, habe ich mir erlaubt, aus Platzgründen die Liste der Sternzeichen abzukürzen. Meins ist drin, das reicht …

```
class Sternzeichen
{
   public string[] this[int monat]
   {
      get
      {
         string[] arr;

         switch(monat)
         {
            case 01:
                 arr = new String[]
                 {"Steinbock", "Wassermann"};
                 break;
            case 02: arr = new String[]
                 {"Wassermann", "Fische"};
                 break;
            case 03: arr = new String[]
                 {"Fische", "Widder"};
                 break;
              :
              :
            case 12: arr = new String[]
                 {"Schütze", "Steinbock"};
                 break;
            default: arr = new String[]
                 {"unbekannt"};
                 break;
         }
         return arr;
      }
```

```
    }

    public uint[] this[string zeichen]
    {
       get
       {
          uint[] arr;

          switch(zeichen)
          {
                case "Widder":
                     arr = new uint[] {3, 4};
                     break;
                case "Stier":
                     arr = new uint[] {4, 5};
                     break;
                  :
                  :
                case "Fische":
                     arr = new uint[] {2, 3};
                     break;
                default:
                     arr = new uint[] {};
                     break;
          }
          return arr;
       }
    }
}
```

Listing 18.5: Ein Nur-Lese-Indexer. So etwas bitte nicht programmieren

Warum ist dies keine gute Idee für die Realisierung eines Indexers? Aus zwei Gründen. Erstens sollten Indexer nur dann verwendet werden, wenn die Auffassung eines Objekts als Array auch aus Modellierungsgründen Sinn macht. Dies hängt von den Anforderungen an die Anwendung ab. Zweitens, dies ist der Hauptgrund, ist ein Array als ein- oder mehrdimensionales Feld fast immer eine Variable und damit sowohl zu lesen als auch zu schreiben. Daher wird von einem Indexer beinahe immer intuitiv erwartet, auf die gleiche Art und Weise Werte auch ändern zu können, dass heißt zu schreiben. Darüber hinaus ist ein Array als Datentyp aus Gründen der Anwendungsfreundlichkeit auch

nicht der ideale Datentyp für einen Indexer. Am besten eignet sich dieses Konzept für einen Zugriff wie in einem Dictionary mit einem `int` oder einem `string` als Index.

18.1.4 Indexer und Eigenschaften im Vergleich

Ebenso wie bei der Definition einer Eigenschaft einer Klasse können Sie bei einem Indexer eine der beiden Zugriffsanweisungen `get` oder `set` weglassen, um auf diese Weise einen nur lesbaren Indexer zu erstellen. Die nachfolgende Tabelle listet noch einmal Eigenschaften und Indexer im Vergleich auf.

Eigenschaft	Indexer
Wird anhand des Namens identifiziert.	Wird anhand der Signatur identifiziert.
Der Zugriff erfolgt über einen einfachen Namen oder einen Member.	Der Zugriff erfolgt über ein Element.
Kann ein statischer Member oder ein Instanzmember sein.	Muss ein Instanzmember sein.
Ein get-Accessor einer Eigenschaft weist keine Parameter auf.	Ein get-Accessor eines Indexers hat dieselbe Liste formaler Parameter wie der Indexer.
Ein set-Accessor einer Eigenschaft enthält den impliziten value-Parameter.	Ein set-Accessor eines Indexers hat zusätzlich zum value-Parameter dieselbe Liste formaler Parameter wie der Indexer.

Tabelle 18.1: Indexer im Vergleich mit Eigenschaften

18.2 Auflistungen

Auflistungen (*collections*) sind Datenstrukturen, die logisch zusammengehörige Daten in mehrere Instanzen der gleichen Klasse speichern. So besitzt eine Instanz der Klasse `System.Windows.Forms.Menu` eine Auflistung `MenuItems`, in der die einzelnen Menüpunkte dieses Menüs enthalten sind. Kennzeichen von Auflistungen innerhalb der .NET-Laufzeitumgebung ist auch, dass sich die einzelnen Elemente mit Hilfe einer `foreach`-Schleife aufzählen lassen, so dass Sie nicht von vornherein wissen müssen, wie viele Elemente vorhanden sind und sich außerdem die Deklaration einer Laufvariablen sparen.

Viele Bücher zu anderen (vor allem nicht so modernen) Programmiersprachen widmen dem Thema Datenstrukturen für Auflistungen Dutzende von Seiten. Bei C# (und allen anderen .NET-Sprachen) hat man als Autor Pech gehabt und muss sich etwas anderes einfallen lassen. Der Namensraum `System.Collections` enthält nämlich bereits alles, was im Alltag des Softwareentwicklers anfällt.

Ein Thema werde ich allerdings nach der Besprechung der vorhandenen Auflistungen dennoch vorstellen: wie Sie Ihre eigenen Klassen so erweitern, dass Sie diese mit einer `foreach`-Schleife wie beispielsweise ein Array durchlaufen können. Zuvor aber ein kurzer Blick auf die bei .NET mitgelieferten Auflistungsklassen.

18.2.1 Auflistungsklassen

Die folgende Tabelle gibt Ihnen eine Übersicht, was die .NET-Laufzeitumgebung im Namensraum `System.Collections` so alles mitbringt (und Ihnen damit zusätzliche Zeit für einen Cappuccino oder eine Latte macchiato verschafft). Damit und mit der Klasse `Array` aus dem Namensraum `System` (die Sie bereits in Kapitel 10 ***Structs und Arrays*** kennen gelernt haben) sollten Sie die meisten Anforderungen bedienen können.

Klasse	Funktion
ArrayList	Ein Array, dessen Größe bei Bedarf dynamisch erweitert werden kann.
BitArray	Ein Feld von Bitwerten (`true` bzw. `false`) zur Darstellung von großen Binärvektoren.
HashTable	Eine Struktur aus Schlüsseln und dazugehörigen Werten, welche auf dem HashCode des Schlüssel basierend abgelegt wird.
Queue	Eine normale Warteschlange mit einer FIFO-Verarbeitung (*first in, first out*).
SortedList	Ähnlich einem HashTable, aber sortiert nach dem Schlüsselwert und über den Schlüssel oder einen nummerischen Index abrufbar.
Stack	Ein Stapelspeicher aus Objekten, also eine LIFO-Struktur (*last in, first out*)

Tabelle 18.2: Auflistungsstrukturen in System.Collections

Die folgenden Methoden werden von den Klassen zur Verfügung gestellt. Die einzelnen Spalten haben von links nach rechts die folgende Bedeutung:

- Name der Methode bzw. Eigenschaft
- Klasse Array (AR)
- Klasse ArrayList (AL)
- Klasse BitArray (BA)
- Klasse HashTable (HT)
- Klasse Queue (QU)
- Klasse SortedList (SL)
- Klasse Stack (ST)

IM bedeutet Instanzmethode der Klasse, SM ist eine statische Methode der Klasse und ein leeres Tabellenfeld bedeutet, dass die Klasse diese Methode nicht zur Verfügung stellt.

Name	AR	AL	BA	HT	QU	SL	ST
Adapter		SM					
Add		IM		IM		IM	
AddRange		IM					
And			IM				
BinarySearch	SM	IM					
Clear	SM	IM		IM	IM	IM	IM
Clone		IM	IM	IM	IM	IM	IM
Contains		IM		IM	IM	IM	IM
ContainsKey				IM		IM	
ContainsValue				IM		IM	
Copy	SM						
CopyTo	IM	IM	IM	IM	IM	IM	IM

Tabelle 18.3: Methoden der Auflistungsklassen (sm = Static, im = instance)

Name	AR	AL	BA	HT	QU	SL	ST
CreateInstance	SM						
Dequeue					IM		
Enqueue					IM		
Equals	IM	IM	IM	IM	IM	IM	IM
FixedSize		SM					
Get			IM				
GetByIndex						IM	
GetEnumerator	IM	IM	IM	IM	IM	IM	IM
GetHashCode	IM	IM	IM	IM	IM	IM	IM
GetKey						IM	
GetKeyList						IM	
GetLowerBound	IM						
GetUpperBound	IM						
GetObjectData				IM			
GetRange		IM					
GetType	IM	IM	IM	IM	IM	IM	IM
GetValue	IM						
GetValueList						IM	
IndexOf	SM	IM					
IndexOfKey						IM	
IndexOfValue						IM	
Initialize	IM						
Insert		IM					
InsertRange		IM					
LastIndexOf	SM	IM					
Not			IM				
Or			IM				
Peek					IM		IM

Tabelle 18.3: Methoden der Auflistungsklassen (sm = Static, im = instance) (Forts.)

Name	AR	AL	BA	HT	QU	SL	ST
Pop							IM
Push							IM
ReadOnly		SM					
Repeat		SM					
Reverse	SM						
Remove		IM		IM		IM	
RemoveAt		IM				IM	
RemoveRange		IM					
Reverse		IM					
Set			IM				
SetByIndex						IM	
SetAll			IM				
SetRange		IM					
SetValue	IM						
Sort	SM	IM					
Synchronized		SM		SM	SM	SM	SM
ToArray		IM			IM		IM
ToString	IM	IM	IM	IM	IM	IM	IM
TrimToSize		IM				IM	
Xor			IM				

Tabelle 18.3: Methoden der Auflistungsklassen (sm = Static, im = instance) (Forts.)

Die folgenden Eigenschaften werden von den Klassen zur Verfügung gestellt.

	AL	AR	BA	HT	QU	SL	ST
Capacity	-JA-					-JA-	
Count	-JA-		-JA-	-JA-	-JA-	-JA-	-JA-

Tabelle 18.4: Eigenschaften der Auflistungsklassen

	AL	AR	BA	HT	QU	SL	ST
IsFixedSize	-JA-	-JA-		-JA-		-JA-	
IsReadOnly	-JA-	-JA-	-JA-	-JA-	-JA-	-JA-	-JA-
IsSynchronized	-JA-	-JA-	-JA-	-JA-	-JA-	-JA-	-JA-
Item	-JA-		-JA-	-JA-		-JA-	
Keys				-JA-		-JA-	
Length		-JA-	-JA-				
Rank		-JA-					
SyncRoot	-JA-	-JA-	-JA-	-JA-	-JA-	-JA-	-JA-
Values				-JA-		-JA-	

Tabelle 18.4: Eigenschaften der Auflistungsklassen (Forts.)

18.2.2 Auflistungen, Indexer & foreach

Bereits die Klasse `System.Array` bietet die schöne Möglichkeit, alle Elemente einfach mit einer foreach-Schleife zu durchlaufen. Diese Funktionalität ist im Rahmen einer eigenen Auflistung oder auch eines Indexers für eine Klasse sehr nützlich. Die Implementierung dieser Funktionalität ist relativ einfach.

Falls Sie auch zu den Entwicklern gehören, die bei einem 'relativ einfach' oder 'im Prinzip kein Problem' bereits geistig vor der unweigerlich folgenden 'Aber-Granate' in Deckung gehen, kann ich Sie beruhigen. Kompliziert wird die Sache erst, falls mehrere Threads in einer Anwendung auf die Daten zugreifen, da das Aufzählungsobjekt (der Enumerator) keinen exklusiven Zugriff auf die Daten der Auflistung besitzt. Es sind also nur die üblichen Probleme aus dem Bereich Locking und Threadsynchronisation …

Was brauchen Sie nun für eine Klasse, die mit `foreach` aufzählbar sein soll? Einmal muss Ihre Klasse die Schnittstelle `IEnumerable` implementieren. Diese Schnittstelle besitzt eine Methode `GetEnumerator()`, welche eine Instanz einer Klasse liefert, welche wiederum die Schnittstelle `IEnumerator` implementiert.

Diese Schnittstelle wiederum besitzt zwei Methoden und eine Eigenschaft, die Sie für Ihre Auflistung implementieren müssen: `Reset()`, `MoveNext()` und die Eigenschaft `Current`.

Zusammengefasst und etwas vereinfacht ist der Ablauf also folgendermaßen:

- `foreach` testet, ob die Klasse die Schnittstelle IEnumerable implementiert.
- Ihre Klasse erzeugt einen Enumerator und liefert diesen an `foreach`.
- `foreach` ruft die Methode `Reset` auf.
- `foreach` ruft die Methode `MoveNext` auf.
- Ihre Klasse stellt beim ersten Aufruf von `MoveNext` nach einem `Reset` den internen Zeiger auf das erste Element und liefert `true`, falls in der Auflistung Daten enthalten sind.
- `foreach` liest die Eigenschaft `Current` des Enumerator-Objekts und wiederholt anschließend den Aufruf von `MoveNext` solange, bis ein `false` geliefert wird.

Als Beispiel hier noch mal das ganze in C# ausprogrammiert, mit dem Indexer aus meinem Beispiel über Indexer. Damit können wir dann die Klasse auch in einer Schleife durchlaufen. Die Änderungen gegenüber der ursprünglichen Version sind fett hervorgehoben.

```
using System;
using System.Collections;

namespace Indexers_and_Collections
{
   class TopTen : IEnumerable
   {
      string[] rang = new string[10];

      // eine interne klasse für den Enumerator
      class MyEnumerator : IEnumerator
      {
         string[] rang = new string[10];
         int enumIndex;
```

```
   public MyEnumerator(string[] topten)
   {
      topten.CopyTo(rang,0);
      enumIndex = -1;
   }

   public void Reset()
   {
      // immer eins weniger als der start
      enumIndex = -1;
   }

   public bool MoveNext()
   {
      // wird vor dem ersten zugriff einmal
      // aufgerufen, daher Reset = -1!
      enumIndex ++;
      if (enumIndex < rang.GetLength(0))
         return true;
      else
         return false;
   }

   public object Current
   {
      get
      {
         return rang[enumIndex];
      }
   }
}

// erzeugt und liefert den Enumerator
public IEnumerator GetEnumerator()
{
   return new MyEnumerator(rang);
}

public string this[int index]
{
   get
   {
      if ((index > 10) || (index < 1))
         throw new
         ArgumentOutOfRangeException(
```

```
                    "Nur Plätze von 1 bis 10 möglich!");
            else
                 return rang[index-1];
         }

         set
         {
            if ((index > 10) || (index < 1))
               throw new
               ArgumentOutOfRangeException(
                    "Nur Plätze von 1 bis 10 möglich!");
            else
               rang[index-1] = value;
         }
      }

      public int this[string name]
      {
         get
         {
            int p = Array.IndexOf(rang, name);
            if (p >= rang.GetLowerBound(0))
            {
               return p+1;
            }
            else
               throw new
                   ArgumentException(
                   "Keine Platzierung für " + name);
         }

         set
         {
            if((value >= 1) && (value <= 10))
               rang[value-1] = name;
            else
               throw new
                   ArgumentOutOfRangeException(
                   "Nur Plätze von 1 bis 10!");
         }
      }

   }

    class TestClass
```

```
    {
        static void Main(string[] args)
        {
         TopTen tt = new TopTen();

         tt[1] = "Benzino Napoloni";
         tt[2] = "Karl Bleifuss";
         tt[3] = "Emilio Commodo";

         Console.WriteLine("Großer Preis von C#");

         foreach(string sName in tt)
            Console.WriteLine(sName);

         string driver = "Karl Bleifuss";

         Console.WriteLine("--------------------");
         Console.WriteLine("{0} ist auf Platz {1}",
                           driver, tt[driver] );

        }
    }
}
```

Listing 18.6: Der Indexer ist jetzt aufzählbar

Falls Sie die Platznummer ebenfalls ausgeben möchten, ändern Sie einfach die Zeile, in der die Eigenschaft `Current` das aktuelle Element der Auflistung liefert.

```
return (enumIndex+1).ToString() + ". = " +
       rang[enumIndex];
```

Achten Sie darauf, bei solchen Sachen keine Seiteneffekte einzubauen! Ein wiederholter Aufruf von `Current` muss sicherstellen, dass immer dasselbe Element geliefert wird, bis `Reset()` oder `MoveNext()` ausgeführt wird. Auch hier wird das `+1` wieder benötigt, um vom Array-Index `0` bis `9` auf die freundlichere und übliche Top Ten Liste von `1` bis `10` zu kommen.

Die Auflistungsklassen und viele andere Klassen der .NET-Laufzeitumgebung implementieren die Schnittstellen `IEnumerable` und `IEnumerator`. Dies reicht vom Array über Druckereinstellungen bis zur Auflis-

tung MenuItems in der Klasse für die Menüs einer GUI-Anwendung. Im Referenzteil erhalten Sie dazu eine Liste.

18.3 Zusammenfassung

Indexer bieten eine Möglichkeit, Instanzen von Klassen als logisches Array aufzufassen und über einen Index eines beliebigen ordinalen Datentyps oder Strings anzusprechen.

Indexer ähneln einer Eigenschaft und obwohl ein Indexer wie bei Eigenschaften durchaus auch als Nur-Lese-Indexer implementiert werden kann, sollte dies vermieden werden. Ein Array ist eine Datenstruktur, welche sowohl gelesen als auch geschrieben werden kann, daher sollte bei einem reinen indexbasierten Lesen eine entsprechende Methode in der Klasse implementiert werden.

Auflistungen sind für alle Standardanwendungsfälle bereits in der .NET-Laufzeitumgebung enthalten, so dass nur noch für spezielle Anforderungen eigene Datenstrukturen oder Klassen entwickelt werden müssen.

Diese Klassen können über eine Implementierung der Schnittstellen IEnumerable und IEnumerator aufzählbar gemacht werden, so dass diese in einer `foreach`-Schleife durchlaufen werden können.

19 Fehlersuche mit C#

Welcher Entwickler kennt das nicht? Während das Programm langsam aus dem Stadium des ersten Prototyps in die lange Kindheit einer in Entwicklung befindlichen Betaversion wächst (*welche meist abrupt am Abend vor der Auslieferung an den Kunden durch einen Wechsel der Versionsnummer endet*), häufen sich im Code eine ganze Reihe von Testausgaben, Hilfsroutinen und anderen Dingern an, die für die Entwicklung zwar von Nutzen sind, später aber nicht mehr benötigt werden.

19.1 Das Attribut [Conditional]

Es geht allerdings auch anders: der erste Schritt dazu sind die meist seit Jahrzehnten von den einzelnen Programmiersprachen unterstützten Möglichkeiten der bedingten Kompilierung (sogar VB ist seit Jahren in diese Riege aufgerückt). Auch bei C# steht diese Möglichkeit zur Verfügung, allerdings nicht über Compilerschalter. Hier wird ein Attribut benutzt, nämlich `[Conditional("<name>")]`. Wird ein Datentyp oder eine Methode mit diesem Attribut gekennzeichnet, ist der entsprechende Code nur dann im ausführbaren Programm enthalten, wenn das gleichnamige Symbol beim Aufruf des Compilers angegeben wird.

Beim C#-Compiler lässt sich dieses Symbol durch den Schalter `/D:<name>` definieren. In der IDE von Visual Studio finden Sie diese Einstellung bei den Projekteigenschaften, abhängig von der jeweiligen Konfiguration (Debug oder Release).

Ist das entsprechende Symbol nicht definiert, werden Aufrufe für entsprechend gekennzeichnete Methoden ignoriert und der Code nicht in die ausführbare Datei aufgenommen.

19.2 Die Klassen Debug und Trace

Im Namensraum `System.Diagnostics` finden Sie die beiden Klassen `Debug` und `Trace`, die das zugrunde liegende Konzept noch etwas ausweiten. Hier sind alle Möglichkeiten der Ausgabe von Hilfsmeldungen, Statusinformationen und anderen Dinge enthalten, welche dem Entwickler das Leben (zumindest die Fehlersuche) erleichtern.

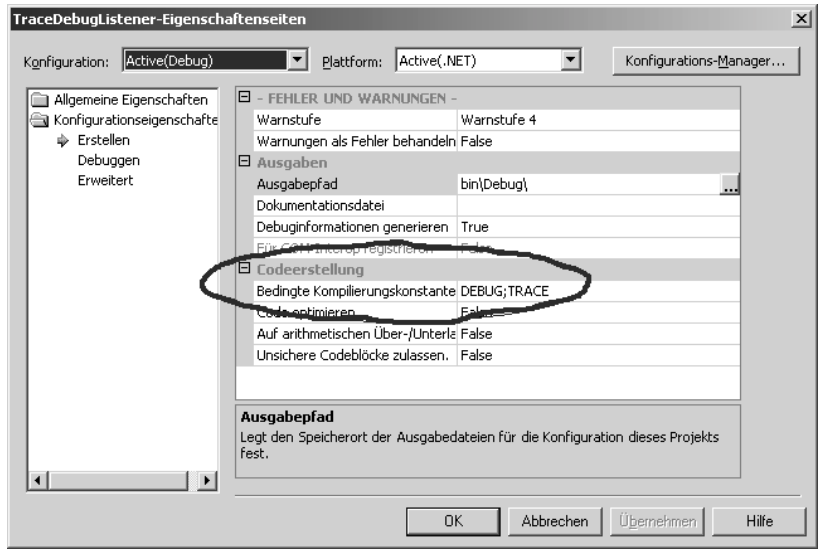


Abbildung 19.1: Die Einstellung der Compiler-Symbole

Bei diesen Klassen handelt es sich allerdings nicht um die plumpe Umsetzung der Funktionalität der VB-Anweisung `Debug.Print`, sondern um eine abstrakte Plattform zur Weitergabe solcher Debug-Meldungen. Natürlich können Sie einfach nur `Debug.WriteLine` wie früher in VB benutzen und müssen sich dann nur merken, dass es jetzt `WriteLine` anstelle von `Print` heisst. Das hieße aber nur einen Bruchteil der Fähigkeiten der beiden Klassen zu benutzen. Doch alles der Reihe nach …

Zuerst die wichtigste Frage: Warum hat Microsoft in die .NET-Laufzeitumgebung zwei Klassen eingebaut, wenn es doch eine vielleicht auch getan hätte? Die Funktion jeder Klasse ist abhängig von der Definition des entsprechenden Compiler-Symbols. Wird bei der Übersetzung weder `DEBUG` noch `TRACE` angegeben, wird gar keiner der für die beiden Klassen geschriebenen Aufrufe in das Programm integriert.

Nach der Empfehlung des C#-Teams sollte die Benutzung der `Debug`-Klasse nicht für ausgelieferte Software benutzt werden (da dann auch das Symbol `DEBUG` nicht mehr definiert wird). Die Klasse `Trace` dagegen ist dafür gedacht, auch in ausgelieferter Software beispielsweise Asserts (mehr dazu später) und anderes zur Verfügung zu stellen. Daher sollten

Sie darauf achten, die Methoden dieser Klasse nicht zu intensiv einzusetzen, um einem eventuellen Performanceverlust der Anwendung durch zu intensive 'Selbstbeobachtung' vorzubeugen.

Alle Eigenschaften und Methoden der beiden Klassen sind als `static` definiert, Sie benötigen also keine Instanz dieser Klassen, sondern können die Methoden direkt über den Klassennamen ansprechen.

19.2.1 Mitglieder von Debug und Trace

An Eigenschaften bleibt nicht viel zu erklären, hier geht es in der Hauptsache um die Einstellung von Einrückungen für die Ausgabe der entsprechenden Diagnosemeldungen.

- `AutoFlush` – stellt ein, ob nach jedem Schreibvorgang mit `Flush` der Puffer geleert werden soll oder nicht.
- `IndentLevel` – legt die Stufe der Einrückungen fest.
- `IndentSize` – bestimmt die Anzahl der Leerzeichen pro Einrückungsstufe.
- `Listeners` – liefert die Auflistung der für die Diagnoseausgabe definierten Listener (mehr dazu weiter unten).

Die Methoden der beiden Klassen beschäftigen sich ebenfalls mit der Ausgabe von Diagnosemeldungen. Wie Sie das Medium für die Ausgabe dieser Meldungen selbst steuern können, werde ich gleich im nächsten Abschnitt erläutern.

Methode	Beschreibung
Assert	Überprüft eine Bedingung und gibt eine Meldung aus, wenn diese Bedingung zu `false` evaluiert wird; löst einen Fehler aus.
Close	Leert die Ausgabepuffer und schließt dann die Listener.
Fail	Ein Assert, das immer fehlschlägt.
Flush	Leert die Ausgabepuffer und schreibt die Daten für die einzelnen Listener.
Indent	Erhöht die Einrückungsstufe um 1.

Tabelle 19.1: Methoden der beiden Klassen Trace und Debug

Methode	Beschreibung
Unindent	Erniedrigt die Einrückungsstufe um 1.
Write	Gibt Informationen an die Listener aus.
WriteIf	Gibt Informationen an die Listener aus, wenn die angegebene Bedingung erfüllt ist.
WriteLine	Gibt Informationen an die Listener mit Zeilenvorschub aus.
WriteLineIf	Gibt Informationen an die Listener mit Zeilenvorschub aus, wenn die angegebene Bedingung erfüllt ist.

Tabelle 19.1: Methoden der beiden Klassen Trace und Debug (Forts.)

19.2.2 Asserts

Ein `Assert` ist eine Annahme, die während der Programmausführung getestet wird und bei einer Auswertung zu `false` eine Meldung und eine Exception auslöst. Asserts sollten Sie für alle Punkte in einer Methode benutzen, die zwingend für die korrekte Funktion dieser Methode vorhanden sein müssen.

Dabei sollten Sie allerdings den Einsatz dieser Annahmen auf die nicht zu erwartenden Ausnahmen beschränken und die normale Fehlerprüfung im Programmcode erledigen. Hier ein Beispiel für einen weniger guten Einsatz von `Assert`:

```
static void Machwas(string filename)
{
   Trace.Assert(File.Exists(filename),
                "Datei existiert nicht");

   // arbeiten mit der angegebenen Datei
}
```

Listing 19.1: Assert weniger gut eingesetzt

An dieser Stelle täte es auch eine Überprüfung im normalen Code:

```
static void Machwas(string filename)
{
   if( File.Exists(filename)
   {
      // arbeiten mit der angegebenen Datei
```

```
   }
   else
      throw new FileNotFoundException(
            "Datei nicht vorhanden", filename);

}
```

Listing 19.2: Prüfung und Exception

Setzen Sie `Assert` also überall da ein, wo Sie entsprechend überrascht wären, falls die geprüfte Bedingung nicht wahr sein sollte.

```
static void Machwas(double schub)
{
   Trace.Assert(schub > 0.0,
                "Negativer Schub?");

   // code
}
```

Listing 19.3: Assert gut eingesetzt

Die Konzepte von C# zur Unterstützung eines ‚design by contract' sind damit mehr oder weniger ausgeschöpft, weitere Konstruktionen sind in der Sprache selbst nicht vorhanden (einer der wenigen Nachteile von C#). Weitere Informationen zum Thema Design by Contract finden Sie im Standardwerk von Bertrand Meyer zu diesem Thema oder Sie suchen in einer der Internetsuchmaschinen nach dem Begriff.

19.2.3 Ausgabe der Diagnosemeldungen

Bis jetzt habe ich nur allgemein von einer Ausgabe von Diagnose- oder Prüfmeldungen mit den beiden Klassen geschrieben, jetzt wird es Zeit, zu erläutern, wie Sie den Ort der Ausgabe steuern können.

Betrachten wir zuerst das Standardverhalten: Der folgende Code nutzt die Möglichkeiten der Klasse Debug zur Ausgabe von Diagnosemeldungen.

```
// Goto C# -- Armin Hanisch
// beispiel für debug und trace

using System;
using System.IO;
```

```
using System.Diagnostics;

namespace TraceDebug
{
   class Class1
   {
      static int Machwas(int count)
      {
         Debug.Indent();
         Debug.WriteLine("Entering Machwas()");
         int summe = 1;
         for(int i = 1; i <= count; i++)
         {
            summe += summe*(i-1);
            Debug.WriteLine(
               String.Format("loop = {0}, summe = {1}",
               i, summe));
         }
         Debug.WriteLine("Leaving Machwas()");
         Debug.Unindent();
         return summe;
      }

      static void Main(string[] args)
      {
         Debug.Listeners.Add(
            new TextWriterTraceListener(Console.Out) );
         Debug.AutoFlush = true;
         Debug.WriteLine("Entering Main");
         Console.WriteLine("Ergebnis = {0}",
                           Machwas(5) );
         Debug.WriteLine("Leaving Main");
         Console.ReadLine();
      }
   }
}
```

Listing 19.4: Beispielklasse für die Debug-Ausgaben

Hier die Ausgabe des Programms:

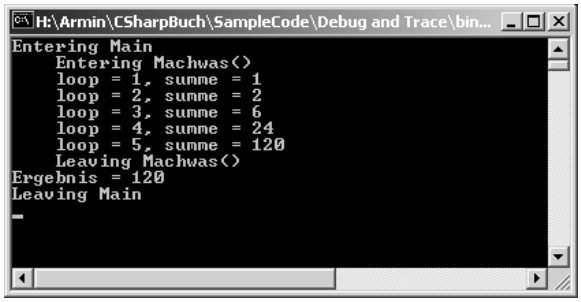


Abbildung 19.2: Die Ausgabe mit den Diagnosemeldungen gemischt

Wie Sie erkennen können, werden in der Ausgabe des Konsolenprogramms Diagnosemeldungen und Programmausgabe gemischt. Dies liegt daran, dass ich mit der Zeile

```
Debug.Listeners.Add(
      new TextWriterTraceListener(Console.Out) );
```

einen Listener erstellt habe, der die Standardausgabe als Ziel benutzt. Dies ist auch das Standardverhalten der beiden Klassen. Ein Listener?

19.2.4 Das Listener-Konzept für Debug-Output

Wie aus dem letzten Beispiel ersichtlich, ist die Ausgabe in die Konsole nicht unbedingt immer die sinnvollste Möglichkeit für Diagnosemeldungen. Programmausgabe und Meldungen werden bunt gemischt und bei einer GUI-Anwendung mit Windows Forms ist eine Konsole standardmäßig nicht angelegt. Daher hat das .NET-Team das Konzept abstrahiert und eine allgemeine Möglichkeit geschaffen, eine ganze Kette von Lauschern (listener) an die Ausgabe anzuhängen.

Zuständig dafür ist die Eigenschaft `Listeners` der Klassen `Debug` und `Trace`, die den Datentyp `TraceListenerCollection`, einer ebenfalls rein statischen Klasse, mit Methoden wie `Add`, `Clear` und `Remove` zur Verwaltung der Listener in der Auflistung. Über den Indexer der Klasse (die Eigenschaft `Item` in anderen .NET-Sprachen) können Sie auf die Liste der aktuell angemeldeten Listener zugreifen (wobei Ihnen die Eigenschaft `Count` die Anzahl liefert).

Was ein Listener mit der Ausgabe anstellt, bleibt dem Entwickler überlassen. Für einen eigenen Listener müssen Sie diesen von der abstrakten Klasse `TraceListener` ableiten und zumindest die Methoden `Write` und `WriteLine` mit jeweils vier Überladungen implementieren. Ob Sie `Fail`, `Close` und `Flush` implementieren, hängt von der Art des Listeners ab, den Sie erstellen. Damit wären auch schon alle Methoden eines Listeners erwähnt. Sie besitzen für die Ausgabe von Diagnosemeldungen vier Möglichkeiten (jeweils als `Write` ohne Zeilenvorschub und als `WriteLine` mit Zeilenvorschub):

- object – gibt die `ToString()`-Ausgabe des Objekts weiter.
- string – schreibt den übergebenen String.
- object, string – gibt `ToString()` des Objekts mit der Kategorie in string aus.
- string, string – gibt die Meldung im ersten String mit der Kategorie im zweiten String aus.

Beide Klassen tragen in die Auflistung als Standard einen `DefaultTraceListener` ein. Diese von `TraceListener` abgeleitete Klasse sendet die Ausgabedaten an die Win32 `OutputDebugString`-API und über die Methode `Log` der Klasse `Debugger` an einen evtl. angeschlossenen Debugger.

19.2.5 Einen eigenen Listener implementieren

Die .NET-Laufzeitumgebung bietet mit den Klassen `EventLogTraceListener` und `TextWriterTraceListener` zwei Ausgabemöglichkeiten. Die erste würde ich nicht empfehlen, wenn Sie nicht genau wissen, was Sie da machen. Mit dem Schreiben von Diagnosemeldungen in das Ereignisprotokoll von Windows können Sie dieses zum einen ziemlich schnell füllen und zum anderen den Administrator Ihres System in tiefe Verwirrung stürzen, wenn er auf einmal Meldungen dort findet, die er nicht erwartet …

Die zweite Möglichkeit ist eine flexbile Lösung für alle Ausgaben der Diagnosemeldungen in eine Datei oder die Standardausgabe der Konsole (`Console.Out`). Was Microsoft aber nicht bietet, ist ein GUI-Listener, der die Meldungen beispielsweise in einem eigenen Formular in einer handlichen Listbox mitloggt. Diesen bekommen Sie mit dem letzten Beispiel für dieses Kapitel geliefert.

Für die Implementierung eines eigenen Listeners muss die Klasse von der abstrakten Klasse `TraceListener` abgeleitet werden und mindestens die `Write`- und `WriteLine`-Methoden implementieren. Der Rumpf der Klasse sieht daher so aus wie im ersten Listing.

```
public class MyListener
       : System.Diagnostics.TraceListener
{
   public override void Write(Object obj)
   {
   }

   public override void Write(string s)
   {
   }

   public override void Write(Object obj, string s)
   {
   }

   public override void Write(string s1, string s2)
   {
   }

   public override void WriteLine(Object obj)
   {
   }

   public override void WriteLine(string s)
   {
   }

   public override void WriteLine(Object obj, string s)
   {
   }

   public override void WriteLine(string s1, string s2)
   {
   }
}
```

Listing 19.5: Der Rumpf für die eigene Listener-Klasse

Was noch fehlt, ist die Implementierung der tatsächlichen Funktionalität sowie der grafische Teil. Dafür verwende ich einfach ein kleines Formular mit einer Listbox. Wie die nachfolgende Abbildung zeigt, habe ich beim Fensterstil (Sizeable Toolwindow) das Control-Menü und die Schaltfläche zum Schließen abgeschaltet, schließlich wäre es ziemlich unpassend, aus Versehen im falschen Moment die Instanz des Listeners durch ein Schließen des Fensters abzuwürgen.

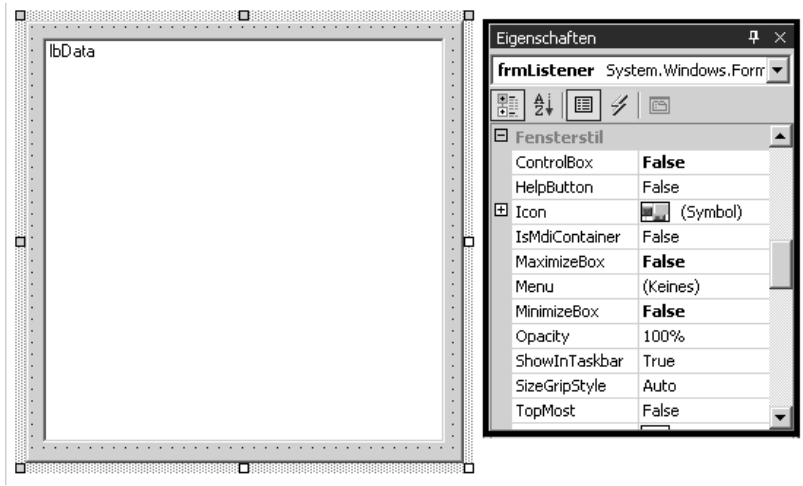


Abbildung 19.3: Das Formular für den Listener und die Fenstereigenschaften

Für eine Instanz dieses Formular wird in der neuen Listenerklasse nun ein privates Feld benötigt und im Konstruktor des Listeners muss das Formular angezeigt werden.

```
public class MyListener
       : TraceListener
{
   private frmListener gui;

   public MyListener()
   {
      gui = new frmListener();
      gui.Show();
      gui.lbData.Items.Clear();
      gui.lbData.Items.Add("Starting Listener ...");
```

```
    }

    // Write- und WriteLine ab hier
    //  :
    //  :
}
```

Listing 19.6: Initialisierung des Formulars

Was fehlt noch? Auf Seiten des Listeners nichts mehr, allerdings muss jede Anwendung, die diesen Listener benutzen möchte, ihn auch in die Auflistung `Listeners` der `Trace`- bzw. `Debug`-Klasse eintragen. Am besten passiert dies beim Laden des Startformulars, wie hier anhand des `FormLoad`-Ereignisses eines Beispielformulars gezeigt.

```
private void Form1_Load(object sender,
                        System.EventArgs e)
{
    // bei bedarf den default-listener entfernen
    Trace.Listeners.Clear();

    // neuen listener eintragen
    Trace.Listeners.Add(new MyListener());
}
```

Listing 19.7: Nutzung der neuen Listener-Klasse

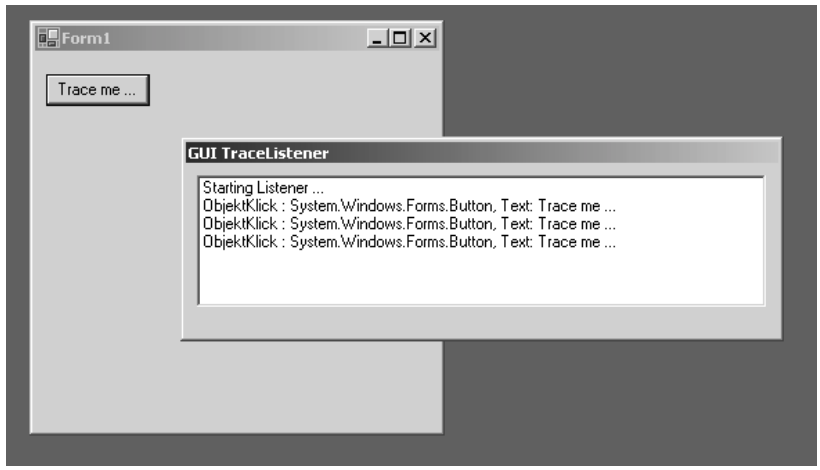


Abbildung 19.4: Die neue Listener-Klasse im Einsatz

Anschließend kann der Listener die entsprechenden Meldungen darstellen, Sie müssen dazu nicht immer wieder zu einem Konsolenfenster umschalten und vor allem können Sie in der Listbox bequem zurückblättern. Ein Beispiel für den praktischen Einsatz der neuen Listener-Klasse sehen Sie in Abbildung 19.4.

Das komplette Listing für diese Klasse finden Sie auf der CD zum Buch.

19.2.6 Kommunikation mit dem Debugger

Die Klasse `Debugger` enthält einige nützliche Routinen für die Kommunikation mit einem angeschlossenen Debugger. Die Methode `Log`, mit der Sie eine Meldung an das Ausgabelog des Debuggers senden können (wenn der Debugger dies unterstützt, wie z.B. Visual Studio.NET), haben Sie bereits kennen gelernt.

Ob der Debugger dies unterstützt, können Sie über die Eigenschaft `IsLogging` feststellen. Über die Eigenschaft `IsAttached` der Klasse können Sie feststellen, ob überhaupt ein Debugger an den Prozess Ihrer Anwendung angedockt ist. Mit der Methode `Break` schließlich lösen Sie ein Break der Anwendung und den Sprung in den Debugger programmgesteuert aus.

19.3 Zusammenfassung

Die Klassen zur Unterstützung bei der Fehlersuche bieten komfortable Möglichkeiten, während der Laufzeit der Anwendung Diagnoseinformationen zu erhalten und ohne eine Störung der Benutzeroberfläche auszugeben.

Mit Hilfe von Assertions sind Sie in der Lage, Grundannahmen Ihres Codes zu verifizieren und gegebenenfalls den Programmablauf anzuhalten. Die Unterscheidung zwischen `Debug`- und `Trace`-Klassen ermöglicht eine gute Aufteilung solcher Diagnosen.

Über eigene Klassen sind Sie in der Lage, selbst Listener für solche Ereignisse zu erstellen und erhalten so die vollständige Kontrolle über Ausgabe und Speicherung von Diagnosemeldungen. Auch mit einem evtl. angeschlossenen Debugger kann mit Hilfe entsprechender Klassen kommuniziert werden.

20 Delegates & Events

20.1 Delegates

Ähnlich wie ein Interface ist auch ein Delegate bei C# eine Art von Schablone oder Vertrag zwischen zwei Klassen, allerdings auf der Ebene einer Methode. Am ehesten vergleichbar ist das Konzept von Delegates noch mit einer typsicheren Form von Funktionszeigern oder Prozedurvariablen in Sprachen wie C++ oder Delphi. Die Möglichkeiten von Delegates gehen aber noch über die von Funktionszeigern hinaus.

Ich bin normalerweise ein Verfechter von passenden Übersetzungen in die jeweilige Sprache des Lesers. Bei vielen Ausdrücken ist dies auch problemlos möglich (z.B. 'Ereignis' statt event, 'Werttyp' anstelle von value type). Für bestimmte Fachausdrücke in der .NET-Welt hat Microsoft in der Dokumentation bereits deutsche Übersetzungen geprägt, die mehr oder weniger zutreffend sind (wenn auch der Ausdruck 'Baugruppe' statt Assembly für mich immer noch nach Schweißgerät und Maschinenöl klingt). Dennoch gibt es einige Begriffe, für die ich auch nach längerem Nachdenken keinen passenden deutschen Begriff gefunden habe, ohne den Lesefluss unnötig ins Stocken zu bringen. Dazu gehören die Delegates. Die Methode von Microsoft, einfach ‚Delegat' zu verwenden, ist genauso wenig begeisternd wie das in einigen Websites auftauchende ‚Abgesandter' (»Ihr wollt den König sprechen, Abgesandter?«). Falls Sie eine passende Übersetzung haben, senden Sie mir eine eMail. Senden Sie mir auch eine Mail, wenn Sie lieber keine Übersetzungen hätten.

20.1.1 Deklaration eines Delegate

Bei der Deklaration wird nur die Schablone für eine Methode angegeben, an welche der Delegate die Ausführung später abgibt. Übereinstimmen muss also nur die Signatur der Methode (Parametertyp und -anzahl und evtl. Rückgabetyp). Zur Deklaration verwenden Sie das Schlüsselwort `delegate`.

```
public delegate string StringOp(string s);
```

Listing 20.1: Ein Delegate für eine Stringoperation, die einen String zurückgibt

Damit habe ich den Delegate erstellt. Wie der Name aber bereits ausdrückt, führt dieser keinen Code aus, sondern wird nur (eben wie ein Funktionszeiger in C++) dazu benutzt, eine Referenz zu einer anderen Methode mit der gleichen Signatur auszubauen. Damit ich keine Instanz der Testklasse erzeugen muss, lege ich die Methoden als `static` an.

```
public class Test
{
   public static string OhneVokale(string s)
   {
      StringBuilder sb = new StringBuilder(s);
      sb = sb.Replace("a", "");
      sb = sb.Replace("e", "");
      sb = sb.Replace("i", "");
      sb = sb.Replace("o", "");
      sb = sb.Replace("u", "");
      return sb.ToString();
   }

   public static string Groß(string s)
   {
      return s.ToUpper();
   }

   public static string Laenge(string s)
   {
      return string.Fomat("Länge = {0}",
                          s.Length);
   }
```

Listing 20.2: Einige Operationen, auf die der Delegate verweisen kann

Alles, was nun noch fehlt, ist die Zuweisung an den Delegate. Dazu ergänze ich die Klasse um den folgenden Code:

```
StringOp so;

// zuweisung der referenz
so = new StringOp(Groß);

// testen des delegate
Console.WriteLine( so("Hallo, da draussen!") );
```

```
// neue zuweisung
so = new StringOp(Laenge);
Console.WriteLine( so("Hallo, da draussen!") );
```

Listing 20.3: Die Zuweisung der Referenz an den Delegate

Dieser Code produziert die folgende Ausgabe:

```
HALLO, DA DRAUSSEN
Länge = 18
```

Das in vielen C#-Büchern folgende Beispiel über die Sortierung nach verschiedenen Feldern mit Hilfe von Delegates erspare ich Ihnen an dieser Stelle und verweise stattdessen auf die .NET-Dokumentation zur Klasse `System.Array` und deren Methode `Sort()`, da ich nichts davon halte, Beispiele für Situationen aufzuführen, die in einem realen Szenario nie auftreten (oder programmieren Sie noch Ihre Sortierroutinen selbst über einen Bubblesort?).

Interessanter ist dagegen der Ansatz, die Nutzung von Delegates für den Anwender der Klassen einfacher zu machen, indem die entsprechenden Variationen als Eigenschaften einer eigenen Klasse zur Verfügung gestellt werden. Dazu lege ich die Deklaration des Delegate und der Stringoperationen in eine eigene Klasse.

```
// delegates in eigener klasse
class StringOps
{

   public delegate string StringOp(string s);

   public static string fOhneVokale(string s)
   {
      StringBuilder sb = new StringBuilder(s);
      sb = sb.Replace("a", "");
      sb = sb.Replace("e", "");
      sb = sb.Replace("i", "");
      sb = sb.Replace("o", "");
      sb = sb.Replace("u", "");
      return sb.ToString();
   }

   public static string fGroß(string s)
   {
```

```
      return s.ToUpper();
   }

   public static string fLaenge(string s)
   {
      return string.Fomat("Länge = {0}",
                          s.Length);
   }

   // jetzt die eigenschaften
   public static StringOp OhneVokale
   {
      get
      {
         return new StringOp(fOhneVokale);
      }
   }

   public static StringOp Groß
   {
      get
      {
         return new StringOp(fGroß);
      }
   }

   public static StringOp Laenge
   {
      get
      {
         return new StringOp(fLaenge);
      }
   }
}
```

Listing 20.4: Die Delegates für Stringoperationen in einer eigenen Klasse

Dies ist eine Erleichterung für den Anwender, da jetzt die entsprechenden Referenzen des Delegates wie eine statische Eigenschaft der Klasse benutzt werden können. Vergleichen Sie dazu die Routine `Main()` hier und im obigen Abschnitt.

```
// zuweisung der referenz
class Test
{
```

```
   public static void Main()
   {
      StrOps.StringOp so;

      // viel einfacher:
      so = StrOps.Groß;

      // testen des delegate
      Console.WriteLine( so("Hallo, da draussen!") );

      // neue zuweisung
      so = StrOps.Laenge;

      // noch ein test
      Console.WriteLine( so("Hallo, da draussen!") );
   }
}
```

Listing 20.5: Die Testklasse für die statischen Delegates

Das Haupteinsatzgebiet für Delegates liegt bei der Programmierung von Behandlungsroutinen für Ereignisse, weshalb ich diese beiden Themen auch in einem Kapitel zusammengefasst habe. Folgen Sie mir also in einen sehr 'ereignisreichen' Abschnitt dieses Buches …

20.2 Ereignisse

Falls Sie bisher schon Windows-Anwendungen programmiert haben, sind Sie mit dem Konzept der ereignisgesteuerten Programmierung vertraut. Hier wird ein Programm nicht in einer vorgegebenen Reihenfolge von Anweisungen abgearbeitet, sondern die Anwendung reagiert auf Ereignisse, vom Tastendruck über einen Mausklick bis zum Eintreffen von Daten über eine Netzwerkverbindung. Das klassische Beispiel in diesem Bereich liefert der unvergessene Douglas Adams in seinem Roman 'Per Anhalter durch die Galaxis'. Als sich Arthur Dent fragt, was wohl passiert, wenn er diesen Knopf auf der Konsole drückt, kommt als Reaktion auf dieses Ereignis die Nachricht »Bitte diesen Knopf nicht mehr drücken!«.

Eine Ereignisbehandlung besteht aus: Dem Ereignis, einer Routine, die das Ereignis bearbeitet, und schließlich einer Routine, die über das Ereignis informiert wird. Auch in C# folgt die Behandlung von Ereig-

nissen diesem Verleger/Abonnenten-Modell. Sie veröffentlichen Routinen, welche ein Ereignis melden und irgendwelche anderen Klassen klinken sich in die Kette der Abonnenten ein und werden beim Auftreten dieses Ereignisses benachrichtigt. Dazu werden auch die Delegates benötigt: Die an einem Ereignis interessierte Klasse definiert eine Callback-Funktion mit Hilfe des Delegates, welche dann von der Ereignisbehandlungsroutine aufgerufen wird. Klingt kompliziert? Keine Panik! Ich werde zusammen mit Ihnen Stück für Stück eine Ereignisbehandlung implementieren.

Zuvor aber noch einmal die notwendigen Schritte, damit wir auch eine entsprechende Leitlinie haben, was wann erstellt wird:

1. Ereignisinformationen definieren
2. Ereignisauslösende Klasse erstellen
3. Delegate definieren
4. Das Ereignis-Objekt definieren
5. Die Ereignisbehandlungsroutine erstellen
6. Die Klasse mit dem Abonnenten (unserem Client) erstellen
7. Die Callback-Methode für den Client erstellen
8. Callback-Methode bei der Ereignisbehandlungsroutine anmelden
9. Eine Testklasse bauen und alles austesten

Als Beispiel für die Ereignisbehandlung erstelle ich eine Würfelklasse zum Backgammon-Spiel, die bei einem Pasch (zwei Würfeln mit der gleichen Zahl) ein Ereignis 'Pasch' auslösen wird, auf das eine andere Klasse dann reagiert.

20.2.1 Ereignisinformationen definieren

Damit sind wir bei Punkt 1 unserer Liste. Um die Erstellung von Ereignisbehandlungen zu erleichtern, besitzt der Delegate per Konvention zwei Parameter: einmal das Objekt, welches das Ereignis ausgelöst hat und als zweiten Parameter ein Objekt mit Informationen über das Ereignis. Dies könnte für einen Mausklick beispielsweise die X- und Y-Position der Maus, die gedrückte Taste usw. sein, für einen Tastendruck

der Zeichencode der Taste und der Zustand der Umschalttasten. Diese Informationen über das Ereignis werden immer von der Klasse `EventArgs` abgeleitet.

Für mein Beispiel muss die Information gar nicht aufwändig sein. Alles, woran ich interessiert bin, ist die gewürfelte Zahl (nachdem bei einem Pasch die Zahlen beider Würfel gleich sind, reicht auch wirklich eine Zahl). Die von `EventArgs` abgeleitete Klasse sieht wie folgt aus:

```
public class PaschInfo : EventArgs
{
   public readonly int PaschZahl;

   // der konstruktor
   public PaschInfo(int wert)
   {
      PaschZahl = wert;
   }
}
```

Listing 20.6: Die Ereignisinformationen für unsere Würfelklasse

Das war's auch schon, nun folgt die Erstellung der ereignisauslösenden Klasse, in welcher ich dann auch den Delegate unterbringe.

20.2.2 Ereignisauslösende Klasse & Delegate

Diese Klasse übernimmt in meinem Beispiel die Aufgabe eines Würfelroboters, also werde ich später zu Demonstrationszwecken noch eine Methode ergänzen, die eine ganze Reihe von Würfen ausführt und dann bei einem Pasch das Ereignis auslöst. Zu Beginn sieht diese Klasse so aus:

```
public class Wuerfler
{
   public delegate void Pasch(object sender,
                              PaschInfo info);

}
```

Listing 20.7: Die erste Version der ereignisauslösenden Klasse

Hier sieht man die Signatur, der alle Ereignisroutinen folgen: zwei Parameter, das ereignisauslösende Objekt und ein Objekt mit den Informationen über das Ereignis. Schön, zwei Punkte auf einmal erledigt, weiter zu Punkt 4 meiner Liste.

20.2.3 Das Ereignisobjekt definieren

Hier erstelle ich eine `public`-Referenz auf ein `event`-Objekt vom gewünschten Typ (dem Delegate). Dieses Objekt wird später von den Abonnenten (den Clients) über eine Instanz des Delegate belegt. Über diese Referenz kann dann von dieser Klasse aus die Callback-Methode des Clients aufgerufen werden.

```
public class Wuerfler
{
   public delegate void Pasch(object sender,
                              PaschInfo info);

   public event Pasch OnPasch; // das event-objekt

}
```

Listing 20.8: Die Definition des Ereignis-Objekts

20.2.4 Die Ereignisbehandlungsroutine

Nun ergänze ich die Klasse um die tatsächliche Routine, welche das aufgetretene Ereignis an die Kette der Abonnenten weiterschickt. Per Konvention wird diese Methode mit einem nachfolgenden **Notify** bezeichnet, diese Vorgabe ist aber nicht zwingend (sorgt aber für eine leichtere Lesbarkeit).

```
public class Wuerfler
{
   public delegate void Pasch(object sender,
                              PaschInfo info);

   public event Pasch OnPasch; // das event-objekt

   // ein pasch fuer die zahl "wert" ist aufgetreten
   public void PaschNotify(int wert)
   {
      // ist der aufruf überhaupt notwendig?
```

```
      if (OnPasch != null)
      {
         // ereignisinformation erstellen
         PaschInfo pi = new PaschInfo(wert);
         // Ereignis auslösen
         OnPasch(this, pi);
      }
   }

}
```

Listing 20.9: Die Würfler-Klasse bekommt die Notify-Routine

Zuerst vergleiche ich die Referenz (die Adresse) von `OnPasch` mit einer `null`-Referenz. Ist diese nämlich `null`, hat sich noch kein Abonnent für das Ereignis registriert und damit ist auch keinerlei Arbeit nötig.

Anschließend wird eine Instanz der Klasse mit den Informationen über das Ereignis erzeugt (ein ganzer `int`-Wert, welche Fülle an Informationen!).

Zum Schluss wird das Ereignis-Objekt aufgerufen (dahinter versteckt sich der Delegate, welcher die Callback-Funktion des Abonnenten aufruft).

20.2.5 Klasse mit dem Abonnenten erstellen

Jetzt baue ich den Abonnenten (den Client). Diese Klasse nenne ich passenderweise `Spieler`.

```
class Spieler
{
   Wuerfler fWuerfler;  // die wuerfler-klasse
   int fPaschCount;     // anzahl paschs

   // der konstruktor der spieler-klasse
   public Spieler(Wuerfler w)
   {
      // sichern der ereignisklassenreferenz
      fWuerfler = w;
   }

}
```

Listing 20.10: Das Grundgerüst der Client-Klasse

20.2.6 Callback-Funktion des Abonnenten

Jetzt fehlt für die `Spieler`-Klasse noch die Methode, welche das aufgetretene Ereignis verarbeitet (letztlich also der Grund für den ganzen Code). Ich gebe in diesem Beispiel lediglich eine Meldung auf der Konsole aus und erhöhe einen Zähler für die Anzahl der geworfenen Paschs.

```
class Spieler
{
   Wuerfler fWuerfler;  // die wuerfler-klasse
   int fPaschCount;     // anzahl paschs

   // der konstruktor der spieler-klasse
   public Spieler(Wuerfler w)
   {
      // sichern der ereignisklassenreferenz
      fWuerfler = w;
      fPaschCount = 0;
   }

   // die parameter-deklaration folgt dem delegate
   public void PaschGewuerfelt(object sender,
                               PaschInfo pi)
   {
      Console.WriteLine("Pasch mit {0}!",
                        pi.PaschZahl);
      fPaschCount++;
   }

}
```

Listing 20.11: Jetzt wird auf das Ereignis reagiert

20.2.7 Die Callback-Methode anmelden

Das fehlende Glied ist nun nur noch die Verbindung der Methode `PaschGewuerfelt` mit dem Ereignis-Objekt `OnPasch`. Dies geschieht über eine neue Instanz des Delegate (mehr dazu gleich nach dem Listing).

```
class Spieler
{
   Wuerfler fWuerfler;      // die wuerfler-klasse
   public int fPaschCount;  // anzahl paschs
```

```
   // der konstruktor der spieler-klasse
   public Spieler(Wuerfler w)
   {
      // sichern der ereignisklassenreferenz
      fWuerfler = w;
      fPaschCount = 0;

      // wir wollen das ereignis mitbekommen
      // also callback registrieren
      fWuerfler.OnPasch +=
          new Wuerfler.Pasch(PaschGewuerfelt);

   }

   // die parameter-deklaration folgt dem delegate
   public void PaschGewuerfelt(object sender,
                               PaschInfo pi)
   {
      Console.WriteLine("Pasch mit {0}!",
                        pi.PaschZahl);
      fPaschCount++;
   }

}
```

Listing 20.12: Jetzt wird das Ereignis auch gemeldet

Vielleicht haben Sie sich über die etwas merkwürdige Notation bei der Registrierung der Callback-Methode gewundert. Dies hat seinen Grund darin, dass für die Delegates die Operatoren + und – (und damit auch += und -=) überladen wurden. Wird nun eine neue Instanz eines Delegate zu diesem 'addiert', so werden anschließend beide aufgerufen – eine sehr schöne Möglichkeit, diese Verkettung der Aufrufe auszudrücken (ja, Sie haben richtig vermutet: mit einem -= lässt sich die Callback-Methode bei Bedarf wieder vom Ereignis abmelden).

20.2.8 Abschlussarbeiten

Ich hatte Ihnen versprochen, dass die `Wuerfler`-Klasse auch etwas Sinnvolles leistet, nämlich eine Methode für eine bestimmte Anzahl von Würfen der beiden Würfel anzubieten. Dies möchte ich vor dem Erstellen der Testklasse noch nachholen.

```
public class Wuerfler
{
   public delegate void Pasch(object sender,
                              PaschInfo info);

   public event Pasch OnPasch; // das event-objekt

   // ein pasch fuer die zahl "wert" ist aufgetreten
   public void PaschNotify(int wert)
   {
      // ist der aufruf überhaupt notwendig?
      if (OnPasch != null)
      {
         // ereignisinformation erstellen
         PaschInfo pi = new PaschInfo(wert);
         // Ereignis auslösen
         OnPasch(this, pi);
      }
   }

   // wurfelt nCount mal und meldet dabei paschs
   public void Wuerfle(int nCount)
   {
      int z1, z2; // die beiden werte

      Random w1 = new Random(12345); // würfel 1
      Random w2 = new Random(67890); // würfel 2

      for(int i = 0; i < nCount; i++)
      {
         z1 = w1.Next(1,6);
         z2 = w2.Next(1,6);

         // haben wir einen pasch?
         if (z1 == z2)
            PaschNotify(z1);
      }
   }

}
```

Listing 20.13: Jetzt kann die Wuerfler-Klasse auch tatsächlich würfeln

20.2.9 Die Testklasse bauen

```
public class Test
{
   public static void Main()
   {
      // instanz der wuerfler-klasse erzeugen
      Wuerfler w = new Wuerfler();

      // einen spieler erzeugen, der sich
      // fuer das ereignis anmeldet
      Spieler s = new Spieler(w);

      // wuerfeln lassen
      w.Wuerfle(100);

      // etwas statistik zum schluss
      Console.WriteLine("Anzahl Paschs: {0}",
                        s.fPaschCount);

   }
}
```

Listing 20.14: Die Testklasse für die Ereignisverarbeitung

Damit habe ich eine funktionsfähige Klasse und Sie eine Schablone für eigene .NET-Komponenten, die in der Lage sind, Ereignisse zu erzeugen oder zu verarbeiten.

20.2.10 Das große Ganze

Hier noch einmal die jeweils letzten Versionen der einzelnen Methoden und die komplette Testklasse, so dass Sie in der Lage sind, den Zusammenhang zwischen den einzelnen Methoden und Aufrufen leichter zu verfolgen.

```
/*
    Goto C# -- EventDemo
    Autor: Armin Hanisch
*/

namespace EventDemo
{
```

```
// die Wuerfler-Klasse mit der Ereignisauslösung
public class Wuerfler
{
   public delegate void Pasch(object sender,
                              PaschInfo info);

   public event Pasch OnPasch; // das event-objekt

   // ein pasch fuer die zahl "wert" ist aufgetreten
   public void PaschNotify(int wert)
   {
      // ist der aufruf überhaupt notwendig?
      if (OnPasch != null)
      {
         // ereignisinformation erstellen
         PaschInfo pi = new PaschInfo(wert);
         // Ereignis auslösen
         OnPasch(this, pi);
      }
   }

   // wurfelt nCount mal und meldet dabei paschs
   public void Wuerfle(int nCount)
   {
      int z1, z2; // die beiden werte

      Random w1 = new Random(12345); // würfel 1
      Random w2 = new Random(67890); // würfel 2

      for(int i = 0; i < nCount; i++)
      {
         z1 = w1.Next(1,6);
         z2 = w2.Next(1,6);

         // haben wir einen pasch?
         if (z1 == z2)
            PaschNotify(z1);
      }
   }

}


// Die Spieler-Klasse mit der Callback-Methode
class Spieler
```

```
{
   Wuerfler fWuerfler;      // die wuerfler-klasse
   public int fPaschCount;  // anzahl paschs

   // der konstruktor der spieler-klasse
   public Spieler(Wuerfler w)
   {
      // sichern der ereignisklassenreferenz
      fWuerfler = w;
      fPaschCount = 0;

      // wir wollen das ereignis mitbekommen
      // also callback registrieren
      fWuerfler.OnPasch +=
          new Wuerfler.Pasch(PaschGewuerfelt);

   }

   // die parameter-deklaration folgt dem delegate
   public void PaschGewuerfelt(object sender,
                               PaschInfo pi)
   {
      Console.WriteLine("Pasch mit {0}!",
                        pi.PaschZahl);
      fPaschCount++;
   }

}


// Die Testklasse zum Ausprobieren der Klassen
// wuerfler und spieler
public class Test
{
   public static void Main()
   {
      // instanz der wuerfler-klasse erzeugen
      Wuerfler w = new Wuerfler();

      // einen spieler erzeugen, der sich
      // fuer das ereignis anmeldet
      Spieler s = new Spieler(w);

      // wuerfeln lassen
      w.Wuerfle(100);
```

```
      // etwas statistik zum schluss
      Console.WriteLine("Anzahl Paschs: {0}",
                        s.fPaschCount);

   }
}
```

Listing 20.15: Das komplette Listing für die Ereignisbehandlung

20.3 Zusammenfassung

Der Datentyp `delegate` bietet die gleiche Funktionalität wie ein typsicherer Funktionszeiger in C++, aber durch die Möglichkeit der Aufrufverkettung noch darüber hinaus reichende Möglichkeiten. Sie können einen `delegate` als reinen Funktionszeiger zum Aufrufen wechselnder Methoden benutzen.

Ereignisse in C# werden über einen `delegate` realisiert. Dieser wird mit dem Schlüsselwort `event` als `public`-Referenz veröffentlicht. Damit können Clients eine Instanz dieses `delegate` erzeugen und mit einer Referenz auf eine eigene Callback-Methode mit der gleichen Signatur ausstatten.

Durch das Überladen der Operatoren + und – für den Typ `delegate` bietet sich die Möglichkeit, eine Methode in diese Kette einfach per Addition zum `delegate` einzuhängen. Beachten Sie aber, dass Sie von der .NET-Laufzeitumgebung keine Garantie für eine bestimmte Reihenfolge des Aufrufs bekommen. Sie sollten also keinesfalls davon ausgehen, dass der als Letzter mit += in die Kette eingehängte `delegate` auch als Letzter aufgerufen wird.



**Reference –
Teil III**

21 Garbage Collection

Eines der Hauptprobleme bei der Erstellung von stabilen Anwendungen sind Speicherlecks (*memory leaks*). Wie bei einem Schiff, das im Laufe seiner Fahrt Wasser aufnimmt und durch minimale Lecks immer einen gewissen Anteil des sog. Bilgenwassers mitschleppt, haben viele Anwendungen ebenfalls kleine Lecks, durch die der Speicher verloren geht. Wie alle Vergleiche, hinkt auch dieser, vor allem hinsichtlich zweier Punkte: Es dringt kein Wasser ein, sondern das Betriebssystem verliert Speicher, da die Anwendung RAM für Objekte und Ressourcen belegt, diesen aber nicht mehr ordnungsgemäß freigibt. Zweitens fehlte den meisten Windows-Anwendungen bisher ein so nützliches Instrument wie die Bilgenpumpe im Schiff, die das Wasser abpumpt und dem Meer zurückgibt.

21.1 Grundlagen

Zur Freude für alle Seebären und auch Landratten unter den Entwicklern besitzt die .NET-Laufzeitumgebung genau diese Bilgenpumpe für Speicher! Nur dass diese hier Garbage Collection (auf deutsch 'Abfallsammlung' oder 'Müllabfuhr') heißt. Auch dies ist ein grundlegendes Konzept objektorientierter Programmiersprachen und in Sprachen wie Smalltalk, Eiffel oder Lisp längst Standard. Als Entwickler belegen Sie Speicher für Ressourcen und Objekte, kümmern sich aber nicht mehr um die Freigabe, dies erledigt das System automatisch im Hintergrund. Alle Objekte, auf die keine Referenz mehr verweist (und auf die somit auch nicht mehr zugegriffen werden kann), werden aus dem Speicher entfernt. Den Zeitpunkt, zu dem eine solche 'Müllsammlung' durchgeführt wird, legt aber die .NET-Laufzeitumgebung selbst fest.

In vielen OOP-Systemen war die Garbage Collection ein Performanceproblem. Die Auflösung nicht mehr benötigter Referenzen kann komplexer sein als im einfachen Modell von COM und der Hintergrundprozess lief immer im unpassendsten Augenblick. Stellen Sie sich vor, Sie steuern gerade in Echtzeit eine Rakete und der Garbage Collector hält Ihren Thread für einige Millisekunden an!

21.1.1 Die alte Welt – Reference Counting

Das alte Referenzzählungsmodell von COM basierte auf folgender Annahme: Jede Referenz auf ein Objekt erhöht dessen internen Zähler um eins. Greifen also zum Beispiel drei Objekte oder Zeiger auf einen String zu, enthält dessen Referenzzähler den Wert drei. Nun wird das erste Objekt freigegeben, damit erniedrigt sich der Wert des Referenzzählers auf zwei. Fällt dieser Wert auf Null, kann das Objekt aus dem Speicher entfernt werden. So weit, so einfach.

Jetzt habe ich in meiner Anwendung zwei Objekte, welche von anderen Zeigern aus referenziert werden. Jedes dieser Objekte verweist zusätzlich auf das andere. Nun gebe ich die beiden Referenzen frei, indem sich meine Zeiger auf einen Nullwert setze. Was passiert? Die beiden Referenzzähler besitzen immer noch einen von Null verschiedenen Wert, da beide Objekte auf das jeweils andere zeigen. Ich habe keine Möglichkeit mehr, auf die beiden Objekte zuzugreifen, aber die Objekte werden nie wieder freigegeben – ein wunderschönes Speicherleck!

21.1.2 Die neue Welt – Garbage Collection in .NET

Die .NET-Laufzeitumgebung verfährt bei der Garbage Collection anders. Auch hier läuft der Prozess der Garbage Collection extra, was sich daran zeigt, dass der Destruktor einer Klasse in einem anderen Thread läuft als die restlichen Methoden der Klasse. Sie können dies in einem Beispiel leicht überprüfen, indem Sie die Hashcodes für beide Threads ausgeben lassen und die Werte vergleichen.

Die Garbage Collection ist allerdings in der Lage, auch solche gegenseitigen Referenzen wie im COM-Beispiel aufzulösen und dadurch Speicherlecks zu verhindern. Erst bei dieser Zerstörung des Objekts wird aber auch der Destruktor einer Klasse aufgerufen. Sie können also nicht sicher sein, wann der Destruktor der von Ihnen programmierten Klasse aufgerufen wird bzw. ob er während der Laufzeit der Anwendung überhaupt aufgerufen wird. Wie ich im Abschnitt ***Die Garbage Collection steuern*** noch zeigen werde, existieren aber einige Möglichkeiten, in diesen Prozess einzugreifen.

21.2 Finalize, Dispose, Close oder was?

Das Konzept der nicht deterministischen Finalisierung verwirrt vor allem gestandene C++-Entwickler. Schließlich sind viele Klassen so entworfen worden, dass die belegten Ressourcen bei einer Zerstörung des Objekts wieder freigegeben werden. Genau dazu wurde auch der Destruktor einer Klasse benutzt. In der .NET-Welt soll nun alles anders sein? Nicht ganz, Sie sollten nur ein paar Regeln beachten.

21.2.1 Konzepte für die Freigabe von Objekten

Da Sie nicht wissen können, wann der Finalizer der Kasse aufgerufen wird, müssen die Zerstörung des Objekts und die Freigabe belegter Ressourcen einfach voneinander getrennt werden. Die Entwickler von C# empfehlen sowieso, auf die Erstellung eines Finalizers zu verzichten, wenn dies nicht unbedingt notwendig ist. Falls Sie diese Methode implementieren wollen, tun Sie dies wie in C++ durch das 'Gegenteil' eines Konstruktors, nämlich mit einem Destruktor. Diese Methode besitzt den gleichen Namen wie die Klasse, allerdings im Gegensatz zu einem Konstruktor mit einem vorangestellten ~. Weitere Details zur Syntax erfahren Sie im Abschnitt 7.5 ***Konstruktoren und Destruktoren*** im Kapitel 7 über die ***Klassen in C#***.

Wie lautet nun der empfohlene Ansatz zur Trennung von Ressourcenfreigabe und Objektabbau? Erstellen Sie dazu eine Methode, deren Aufruf für die Freigabe der Ressourcen sorgt und rufen Sie diese anstelle des Destruktors auf. Microsoft verwendet innerhalb der .NET-Laufzeitumgebung in den Klassen zwei Methoden, entweder `Close()` oder `Dispose()`. Hierbei handelt es sich um eine reine Konvention: `Close()` legt nahe, dass darauf auch wieder ein `Open()` folgen könnte. Das Objekt gibt die Ressource also nicht dauerhaft frei. Für die endgültige Freigabe von belegten Ressourcen wie Dateien oder Speicher sollte eine Methode `Dispose()` verwendet werden.



Hier zeigt sich in C# eine wunderliche Mischung aus Konvention und Unterstützung durch die Programmiersprache. Dies ist einer der wenigen dunklen Flecken auf der Landkarte von C#, den Microsoft baldmöglichst auflösen sollte. Hier lauert die Gelegenheit zu schwer entdeckbaren Fehlern. Für `Dispose()` nämlich bietet die .NET-Laufzeitumgebung eine Unterstützung zum deterministischen Aufruf.

Implementiert Ihre Klasse die Schnittstelle `IDisposable` (die nur die eine Methode `public void Dispose()` enthält), so wird diese Methode sofort und automatisch aufgerufen, sobald die Programmausführung den Gültigkeitsbereich des Objekts verlässt.

21.2.2 Sichere Speicherfreigabe mit using

Dazu benutzen Sie das Schlüsselwort `using`, allerdings nicht zum Import von Klassen und Namensräumen, sondern in einer völlig anderen Bedeutung. Hier bedeutet `using` die 'Benutzung' einer Ressource, die automatisch über `Dispose()` beim Verlassen der Gültigkeit der Referenz wieder freigegeben werden soll. Auch hier ist ein kleines Beispiel einfacher zu verstehen als lange Erklärungen:

```
// Goto C# -- Armin Hanisch
// freigabe von ressourcen mit using und dispose

namespace GCDemo
{

public class TheBigOne : IDisposable;
{
   // belegt 50 mb ram und
   // öffnet 30 dateien ....

   // implementierung von dispose()
   public void Dispose()
   {
      // alles wieder freigeben
   }
}

public class Testklasse
{
   public static void Main()
   {
      // hier nutzung von using
      using( TheBigOne klotz = new TheBigOne() )
      {
         // arbeiten mit der instanz
         klotz.Machwas();
      }

      // ab hier ist klotz nicht mehr gueltig,
```

```
      // also wird dispose() aufgerufen.
   }
}

}
```

Listing 21.1: Sichere Freigabe von Ressourcen mit Dispose() und using

Wo liegt der Vorteil dieses Ansatzes gegenüber dem manuellen Aufruf von `Dispose()`? Hier stellt die .NET-Laufzeitumgebung und nicht Sie als Entwickler sicher, dass die Methode `Dispose()` aufgerufen wird. Also eine Stelle weniger, an der Sie etwas vergessen können. Zum anderen wird der Code dadurch auch lesbarer. Immerhin sind die meisten Anwendungen doch etwas komplexer als die kleinen Beispiele hier. Daher kann es durchaus vorkommen, dass ein Objekt an zwei oder drei Stellen (geplant, in einer Exception-Routine und für einen Sonderfall) aus dem Gültigkeitsbereich kommt und Sie an all diesen Stellen explizit `Dispose()` aufrufen müssten. Die Klammerung in einem `using`-Block spart diese Arbeit.

Die Nachteile? Eigentlich keine, nur dass dieser Support wie gesagt nur für die Methode `Dispose()` und nicht für `Close()` gilt. Vielleicht ändert sich dies ja noch …

Darüber hinaus funktioniert dieser Ansatz nur mit Klassen, die auch die Schnittstelle `IDisposable` implementieren, Sie können also keine riesigen Arrays aus Werttypen damit freigeben, es sei denn, Sie kapseln diese in einer eigenen Klasse (was sowieso meist der sinnvollere Ansatz ist). Folgende Klasse implementieren diese Schnittstelle und können also `Dispose()` verwenden:

BinaryReader	MessageQueueEnumerator
BinaryWriter	MessageQueueTransaction
Brush	OleDbDataReader
Component	OleDbTransaction
ComponentDesigner	PaintEventArgs
Container	Pen

Tabelle 21.1: Klassen, die IDisposable implementieren

Control	Region
Cursor	RegistryKey
CustomLineCap	ResourceReader
DesignerTransaction	ResourceSet
EncoderParameter	ResourceWriter
EncoderParameters	ResXResourceReader
EventHandlerList	ResXResourceWriter
Font	SearchResultCollection
FontCollection	ServicedComponent
FontFamily	Socket
Graphics	SqlDataReader
GraphicsPath	SqlTransaction
GraphicsPathIterator	Stream
HttpApplication	StringFormat
Icon	TcpClient
Image	TempFileCollection
ImageAttributes	TemplateEditingService
InheritanceService	TemplateEditingVerb
IsolatedStorageFile	TextReader
License	TextWriter
LocalizationExternderProvider	Timer
ManagementObjectCollection	TraceListener
MarshalByValueComponent	UdpClient
Matrix	WaitHandle
MessageEnumerator	WebResource

Tabelle 21.1: Klassen, die IDisposable implementieren (Forts.)

21.3 GC-Generationen

Wird für ein Objekt Speicher belegt, verwaltet die .NET-Laufzeitumgebung diesen in so genannten Generationen. Gleich nach der Belegung befindet sich der Speicherblock in der Generation Null. Erfolgt nun ein Durchlauf des Garbage Collectors und existieren für diesen Speicher noch Referenzen, wird dieser natürlich nicht freigegeben. Dafür wird er sozusagen 'befördert'. Die Generation wird nämlich nach jedem solchen Durchlauf um eins erhöht, bis der von der .NET-Laufzeitumgebung maximal unterstützte Wert erreicht ist. Diesen können Sie über die Klasse `GC` des Namensraums `System` und die Nur-Lesen-Eigenschaft `MaxGeneration` abfragen.

Die .NET-Laufzeitumgebung geht davon aus, dass Speicher, der erst kürzlich belegt wurde, auch bald wieder freigegeben wird. Speicher, der schon mehrere Durchläufe des Garbage Collectors 'überlebt' hat, weil er noch referenziert wird, wird sehr wahrscheinlich noch länger belegt bleiben. In welcher Generation sich Ihr Objekt gerade befindet, lässt sich ebenfalls über eine Methode der Klasse `GC` feststellen:

```
Console.WriteLine( "Generation: {0}",
                   GC.GetGeneration(obj) );
```

Listing 21.2: Feststellen der Speichergeneration eines Objekts

21.4 Die Garbage Collection steuern

Sie können sich die .NET-Müllabfuhr bestellen, natürlich nicht per Telefon, sondern per Methodenaufruf. Rufen Sie dazu die Methode `Collect()` der Klasse `GC` auf. Diese Methode besitzt eine überladene Version, welche als Parameter die Generation akzeptiert, bis zu der (ausgehend von Generation 0) eine Garbage Collection durchgeführt werden soll. Damit können Sie (bei entsprechender Planung Ihrer Anwendung) die Leistung erhöhen, indem Sie die Möglichkeiten einer generationsbezogenen Garbage Collection nutzen.

Nachdem die Methoden Ihrer Klasse und ein eventueller Destruktor in verschiedenen Threads laufen, kann es sein, dass nach einem Aufruf des Garbage Collectors mit `Collect()` eine ganze Reihe von solchen Aufrufen angestoßen wird. Für eine zeitkritische Anwendung sollte daher

eine Möglichkeit der Synchronisierung oder zumindest des Abwartens dieser Destruktoren bestehen. Genau diese Möglichkeit bietet ein Aufruf der Methode `GC.WaitForPendingFinalizers()`. Vor einem zeitkritischen Abschnitt können Sie also mit einem Aufruf der Methode die Ausführung der Destruktoren abwarten.

```
// Beispiel zur Garbage Collection in C#
using System;

class TestObj
{
   private string name;

   // der konstruktor
   public TestObject(string name)
   {
      this.name = name;
      Console.WriteLine("Konstruktor für {0}!",
                        this.name );
   }

   // der destruktor
   public ~TestObject()
   {
      Console.WriteLine("Destruktor für {0}!",
                        this.name );
   }

   // ausgeben der akt. Generation
   public void ShowGen()
   {
      Console.WriteLine("Generation = {0}",
                      GC.GetGeneration(this) );
   }
}

class TestKlasse
{

public static void Main()
{
   Console.WriteLine("GC-Demo");
   Console.WriteLine("Max. Generationen: {0}",
                     GC.MaxGeneration );
```

```
    TestObj to = new TestObj("Objekt 1");
    // hier sollte "0" ausgegeben werden
    to.GetGen();

    // garbage collection starten
    GC.Collect();
    // jetzt sind wir in generation "1"
    to.GetGen();

    // noch einengarbage collection starten
    // objekt "to" ist danach in generation "2"
    GC.Collect();

    // jetzt eine garbage collection bis zur
    // generation 1 ausführen
    GC.Collect(1);

    // jetzt unser objet freigeben
    to = null;

    // diesmal komplette garbage collection
    GC.Collect();

    // jetzt sollte der destruktor aufgerufen werden
    GC.WaitForPendingFinalizers();

    Console.WriteLine("Demo-Ende GC");

}
}
```

Listing 21.3: Garbage Collection am praktischen Beispiel

21.5 Weitere Methoden der Klasse GC

21.5.1 Finalisierungen unterdrücken

Sie können die Ausführung des Destruktors auch unterdrücken, wenn Sie die Aufräumarbeiten bereits in einer eigenen Methoden durchgeführt haben. Dazu schreiben Sie sich diese Methode (ich habe hier den Namen `Dispose()` gewählt) und rufen im eigentlichen Destruktor nur noch diese Methode auf.

Wenn Sie nun in der Anwendung bereits `Dispose()` aufgerufen haben, ist die Ausführung des Destruktors nicht mehr nötig (wir wollen schließlich Ressourcen nicht mehr als einmal freigeben und damit wieder Fehler erzeugen) und sollte daher unterdrückt werden. Dies geschieht mit Hilfe der Methode `SuppressFinalize()` der Klasse `GC`.

```
public class GCTest
{
   public void Dispose()
   {
      // ressourcen freigeben
      ... code

      // finalisierung nicht mehr noetig
      GC.SuppressFinalize(this);
   }

   public ~GCTest()
   {
      // die "aufraeummethode" aufrufen
      Dispose();
   }
}
```

Listing 21.4: Unterdrücken der Finalisierung durch SuppressFinalize()

21.5.2 Destruktoren doch wieder ausführen

Nehmen wir an, Sie haben für ein Objekt mit der Methode `SuppressFinalize()` die Ausführung eines Destruktors abgeschaltet. Im Programmablauf ergibt sich aber nun die Notwendigkeit, diese doch wieder ausführen zu lassen (obwohl dies nicht gerade für eine professionelle Ablaufplanung spricht). Auch hierfür bietet die Klasse GC die entsprechende Methode: `ReRegisterForFinalize()`(es gibt Momente, da gefällt mir die gemischte Schreibweise gar nicht mehr!).



Die Aufrufe von `SuppressFinalize()` und `ReRegisterForFinalize()` werden von der .NET-Laufzeitumgebung nicht synchronisiert! Sie können also `ReRegisterForFinalize()` versehentlich mehr als einmal aufrufen, woraufhin der Destruktor Ihrer Klasse ebenfalls mehr als einmal aufgerufen wird. Wenn Sie dann beim zweiten Mal versuchen, auf bereits freigegebene Ressourcen für die vermeintlich anstehende Frei-

gabe noch einmal zuzugreifen, handeln Sie sich einen Laufzeitfehler ein. Die Dokumentation zur .NET-Laufzeitumgebung sagt zwar aus, dass der Thread für die Finalizer mit nicht behandelten Exceptions klarkommt, aber man sollte nicht mit dem Feuer spielen!

21.5.3 Objekte am Leben erhalten

Wollen Sie, dass eine Instanz einer Klasse von der Garbage Collection nicht freigegeben wird, auch wenn keine Referenz mehr auf diese Instanz existiert, dann verwenden Sie die Methode `KeepAlive()`. Diese bekommt die Instanz der Klasse als Parameter und verhindert eine Garbage Collection für dieses Objekt bis zum Aufruf der Methode. Wie bitte? Bis zum Aufruf von `KeepAlive()`, nicht danach?

Als ich dies zum ersten Mal in der .NET-Dokumentation gelesen habe, habe ich auch gestutzt, beim zweiten Blick ist dieser Sachverhalt aber vollkommen logisch, wenn man sich die Arbeitsweise von `KeepAlive()` vorstellt. Dieser Aufruf erzeugt eine Referenz auf das Objekt, damit wird es bei einer Garbage Collection auch nicht freigegeben. Vom Beginn der Methode bis zum Aufruf dieser Methode also wirkt dies wie ein 'Schutzschild'.

Der Einsatzzweck liegt im Aufruf von nicht von der .NET-Laufzeitumgebung verwaltetem Code (COM, Win32-API, alte DLLs usw.) aus verwaltetem Code heraus. Hier kann es durchaus nötig sein, ein Objekt oder eine Ressource zu behalten, auch wenn gerade keine Referenz auf dieses Objekt zeigt.

21.6 Zusammenfassung

Microsoft hat sich bei der ewigen Wahl zwischen vollständiger Freiheit des Entwicklers und Konsistenz der Speicheranforderungen für den Weg einer automatischen Speicherverwaltung entschieden, wenn Sie sicheren (also durch die CLR verifizierbaren) Code erstellen.

Die periodisch arbeitende Garbage Collection läuft in einem eigenen Thread und kümmert sich um die Freigabe nicht mehr benutzbarer Speicherobjekte. Im Gegensatz zum bei COM verwendeten System

der Referenzzählung ist die Speicherverwaltung der .NET-Laufzeitumgebung in der Lage, auch zirkuläre Referenzen aufzulösen.

Die Verwaltung des Speichers erfolgt in Generationen und kann innerhalb bestimmter Grenzen vom Entwickler gesteuert werden. Über die Implementierung der Schnittstelle `IDisposable` sind Sie in der Lage, bei Verlassen des Gültigkeitsbereichs einer Instanz sofort einen Finalizer aufrufen zu lassen.

22 Reflection in C#

Reflection bedeutet auf Deutsch nicht nur Spiegelung, sondern auch die Selbstbetrachtung und das Widerspiegeln einer Idee in ihrer Umsetzung. Nein, das wird kein Kapitel über philosophisches Programmieren. Allerdings ist der Begriff Selbstbetrachtung eine gute Zusammenfassung, was mit Reflection in C# (und natürlich allen anderen Sprachen der .NET-Umgebung) machbar ist.

RTTI und noch viel weiter

Auch andere OOP-Sprachen wie C++ oder Delphi bieten bereits die Möglichkeit, mit Hilfe von RTTI (*run-time type information*) die Klasse eines Objektes und diverse Informationen über Objekte und Datenstrukturen zur Laufzeit eines Programms festzustellen. C# dagegen erlaubt den kompletten Zugriff auf Informationen einer Klasse bis hin zur dynamischen Erzeugung von Methoden zur Laufzeit!

Mit Hilfe der Reflection-Klassen ist auch der Zugriff auf Metadaten von Assemblies möglich. Sie können also nicht nur herausfinden, wie die Objekte und Klassen aussehen oder Methoden aufrufen, deren Namen Sie erst zur Laufzeit erfahren, sondern eben auch Daten über Daten. Wie ich im Kapitel über Attribute zeigen werde, kann der Code dadurch mit Informationen versehen werden, die dem Programm zur Laufzeit zur Verfügung stehen, die nicht unbedingt etwas mit der Ausführung des Programms zu tun haben.

Da der Namensraum `System.Reflection` weit über vierzig Klassen beinhaltet, bitte ich Sie um Verständnis, dass ich im Rahmen dieses Buches nur einen Einblick in die Thematik geben kann, der aber für eigene Gehversuche sicher ausreichen wird.

22.1 Type, GetType() und typeof()

Der Ausgangspunkt für Experimente mit Metadaten- und Typinformationen ist die Klasse `Type` der .NET-Laufzeitumgebung. Ein solches `Type`-Objekt bekommen Sie entweder über die Methode `GetType()` oder den Operator `typeof()`.

Bevor ich Ihnen das erste Beispiel zeige, möchte ich noch kurz auf die Unterschiede zwischen den beiden Möglichkeiten eingehen. `GetType()` ist eine Methode von `object` und liefert daher ein Type-Objekt für eine bestehende Instanz einer Klasse. Der Operator `typeof()` kommt ohne Instanz aus und kann direkt mit einer Klasse aufgerufen werden. Daher wird in vielen Fällen `typeof()` verwendet, wenn Sie den Namen der Klasse kennen, da dann der Aufwand für die Erstellung eines Objekts nicht notwendig ist.

22.1.1 Die Beispielklasse

Die folgende Klasse werde ich als 'Experimentierfeld' für die nachfolgenden Beispiele zum Thema reflection verwenden.

```
namespace TypInfo {

interface IMachwas {
   void SagHallo(string name);
   }

public class Beispiel : IMachwas
{
   private int fInt;
   private string fString = "geheim";
   private double fSpeed;

   public double speed
   {
      get { return fSpeed; }
      set { fSpeed = value; }
   }
   public Beispiel(int x)
   {
      fInt = x;
   }

   public void SagHallo(string name) {
      Console.WriteLine("Hallo {0}!", name);
   }
```

```
   public bool Schaltjahr(uint jahr) {
      return (jahr % 400 == 0) ||
             ( (jahr % 100 != 0) && (jahr % 4 == 0));
   }
}

}
```

Listing 22.1: Die Beispielklasse zum Thema reflection

22.1.2 Her mit dem Typ: das Type-Objekt

Bevor wir anfangen können, mit den reflection classes zu arbeiten, benötigen wir ein Type-Objekt. Wie weiter oben bei ***Type, GetType() und typeof()*** geschildert, können Sie dies über die Methode `GetType()` eines Objektes oder den Operator `typeof()` erledigen, dem Sie den Klassennamen übergeben. Ich beschränke mich hier auf die Verwendung von `GetType()`.

```
namespace TypInfo {
using System;
public class Test {
   public static void Main() {
      Beispiel bsp = New Beispiel(100);
      Type t = bsp.GetType();
      Console.WriteLine("Datentyp = {0}", t.ToString());
   }
}
}
```

Listing 22.2: Ausgabe des Datentyps mit GetType()

Als Ausgabe erhalten Sie den Namen des Datentyps der Klasseninstanz, in diesem Fall `TypInfo.Beispiel`. Nun grabe ich etwas tiefer und lasse alle Mitglieder der Klasse auflisten.

22.1.3 Members only – Klassenmitglieder auflisten

Dazu verwende ich eine der vielen Methoden der Klasse `Type`, nämlich `GetMembers()`. Alle möglichen Methoden und Eigenschaften zu erläutern, würde beinahe ein eigenes Buch erfordern, da die Klasse `Type` weit über 100 Eigenschaften und Methoden besitzt, was Ihnen eine Vorstellung geben soll, wie mächtig die Möglichkeiten von reflection sind. Sie finden

darin natürlich auch Methoden für die einzelnen Typen von Mitgliedern, z.B. `GetMethods()`, `GetFields()`, `GetInterfaces()` und so weiter.

Die zu ergänzenden Zeilen im Code habe ich fett markiert.

```
namespace TypInfo {
using System;
public class Test {
   public static void Main() {
      Beispiel bsp = new Beispiel(100);
      Type t = bsp.GetType();
      // ausgeben des datentyps
      Console.WriteLine("Datentyp {0}", t.ToString());
      // alle felder, methoden, etc. auflisten
      MemberInfo[] mi = t.GetMembers();
      foreach(MemberInfo m in mi)
         Console.WriteLine( m.ToString() );
   }
}
}
```

Listing 22.3: Jetzt werden die Mitglieder auch ausgegeben

Dieser Code erzeugt die folgende Ausgabe:

```
Datentype = TypInfo.Beispiel
Void SagHallo(System.String)
Int32 GetHashcode()
Boolean Equals(System.Object)
System.String ToString()
Double get_Speed()
Void set_speed(Double)
Boolean Schaltjahr(UInt32)
System.Type GetType()
Void .ctor(Int32)
Double speed
```

Wie Sie erkennen, lässt sich mit sehr wenig Code bereits eine Menge an Informationen über die Mitglieder unserer Beispielklasse erfahren. Die Ausgabe umfasst wirklich alle `public`-Mitglieder unserer Klasse, auch die, welche von `object` geerbt wurden, wie z.B. `ToString()`. Bei der Ausgabe verwendet die `ToString()`-Methode der `MemberInfo-Klasse` allerdings nicht die C#-Aliase für die Datentypen, sondern die allgemeinen Namen der .NET-Umgebung. Verständlich, schließlich ist eines Zeile von .NET ja die sprachübergreifende Programmierung.

Nicht erkannt haben Sie vielleicht das Mitglied mit der Signatur `Void .ctor(Int32)`. Hier handelt es sich um den Konstruktor unserer Klasse! Hätte ich anstelle der Methode `GetMembers()` die Methode `GetMethods()` benutzt, wäre der Konstruktor nicht mit aufgelistet worden, da es sich eben nicht um eine normale Methode handelt. Dafür bietet die Klasse `Type` aber auch eine Methode `GetConstructors()`. Sie sehen also, es ist für alles gesorgt.

Sicher ist sicher – ReflectionSecurity

Was Sie allerdings sicher erkannt haben, ist die Tatsache, dass nur alle als `public` deklarierten Mitglieder der Klasse aufgelistet wurden, obwohl wir explizit `GetMembers()` aufgerufen haben. Dies ist eine Sicherheitseinstellung der .NET-Laufzeitumgebung. Solange eine Klasse nicht die entsprechenden `ReflectionSecurity`-Rechte besitzt, verhindert die Laufzeitumgebung den Zugriff auf alles, was nicht als `public` deklariert ist. Da mit reflection problemlos der Zugriff auf fremdes geistiges Eigentum möglich wäre (Sie möchten schließlich nicht jedem die Interna Ihrer wertvollen Additionsklasse freilegen – Sie erinnern sich noch an das Kapitel zum Überladen von Methoden?), sollten Sie die Rechte für Ihre eigenen Klassen entsprechend einstellen. Nähere Informationen zu diesem Thema finden Sie in der Dokumentation zum .NET Framework SDK.

22.1.4 Parameter fischen

So weit, so gut. Ich (und natürlich auch Sie, wenn Sie die Beispielcodes von der CD kopiert haben) kann jetzt den Namen einer Methode herausfinden, es fehlen aber noch Informationen über die für den Aufruf notwendigen Parameter. Den Code dafür möchte ich jetzt nachreichen. Dafür baue ich den Aufruf von `GetMembers()` in ein `GetMethods()` (damit ändert sich auch die Klasse von `MemberInfo` in `MethodInfo`, Vorsicht!) um und gebe dann in der Schleife auch die Parameterinformationen aus:

```
using System;
namespace TypInfo {

public class Test {
   public static void Main()
   {
      Beispiel bsp = new Beispiel(100);
      Type t = bsp.GetType();
```

```
      // ausgeben des datentyps
      Console.WriteLine("Datentyp {0}", t.ToString());
      // alle felder, methoden, etc. auflisten
      MethodInfo[] mi = t.GetMethods();
      foreach(MethodInfo m in mi)
      {
         Console.WriteLine("-------------------------");
         Console.WriteLine("Meth {0}: ", m.ToString());
         foreach(ParameterInfo p in m.GetParameters())
            Console.WriteLine("   Parameter: {0} {1}",
                              p.ParameterType, p.Name);
      }
      Console.ReadLine();  // warten nach Ende ...
   }
}
}
}
```

Listing 22.4: Ausgabe aller Parameter jeder Methode

Nach dem Start liefert das Programm eine Ausgabe mit den gewünschten Typinformationen der Beispielklasse:

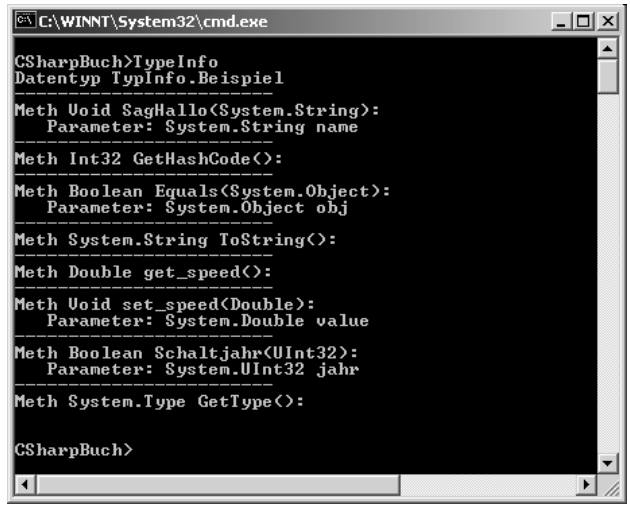


Abbildung 22.1: Reflection erlaubt das Untersuchen von Klassen

22.1.5 Aktivieren – Dynamischer Methodenaufruf

Damit sind eigentlich alle Informationen verfügbar, um mit Hilfe von reflection dynamisch zur Laufzeit eine Methode mit den entsprechenden Parametern zu versorgen und diese aufzurufen. Ich gebe hier nur den Code für die Methode `Main()` der Testklasse wieder, um Platz zu sparen.

```
public static void Main() {
   Type t = typeof(Beispiel);
   // parameter für den konstruktor bauen
   object[] parms = new object[] {42};
   // eine instanz der klasse erzeugen
   object obj = Activator.CreateInstance(t, parms);
   // parameter für den Methodenaufruf bauen
   parms[0] = 2001;
   BindingFlags bflags = BindingFlags.Default |
                         BindingFlags.InvokeMethod;
   bool sj = (bool) t.InvokeMember(
                       "Schaltjahr",  // name
                       bflags,        // flags
                       null,          // binder
                       obj,           // instanz
                       parms          // parameter
                       );
   Console.WriteLine("2001 ist Schaltjahr: {0}", sj);
}
```

Listing 22.5: Die Testklasse für den dynamischen Methodenaufruf

Die Methode `InvokeMember` der `Type`-Klasse erlaubt es, eine Methode oder eine Eigenschaft für eine Instanz dynamisch zu aktivieren. Der erste Parameter ist der Name des Klassenmitglieds. Danach folgen die sog. `BindingFlags`, welche beispielsweise festlegen, ob eine Methode aufgerufen, eine Eigenschaft gesetzt oder gelesen wird. Der dritte Parameter ist ein `Binder`-Objekt, mit dem die Bindung des Objektes genauer gesteuert werden kann. Damit können Sie bei überladenen Methoden unter anderem festlegen, welche Version benutzt werden soll und andere Einstellungen treffen. Parameter Nummer vier ist eine `object`-Variable mit der von uns erzeugten Instanz der Klasse und der letzte Parameter schließlich ist das Array mit den einzelnen Parameterwerten für die Aktivierung. Sollte beim Aufruf einer Methode (`BindingFlags.InvokeMethod`) die entsprechende Methode nicht gefunden

werden, wird eine `MissingMethod`-Exception ausgelöst. Dementsprechend existieren Exceptions für alle anderen Fehlermöglichkeiten. Die Details hierzu finden Sie in der Dokumentation zum .NET-Framework SDK.

22.2 Reflection überzeugend demonstriert

Dieser Abschnitt bringt ein Beispiel für den Einsatz von Reflection, der Ihnen zeigt, dass es sich bei den Reflection-Klassen nicht um höhere Sphären der Programmierung handelt. Ganz im Gegenteil können Sie auch bei kleinen Tools mit dem Einsatz dieser Technik oftmals leidige Schreibarbeit vermeiden.

Die Idee zu diesem Beispiel stammt von Jayant Mukherjee, einem Autor bei *www.csharpindex.com*. Auch ihm scheinen ebenso wie mir die meisten Namen der Farben der Color-Struktur nichts zu sagen. Insofern ist dieses Beispiel auch gleich eine kleine Wiederholung zum Thema Windows Forms. Diese Anwendung macht nichts weiter, als die Namen aller Farben in einer Listbox auszugeben und die Zeile auch mit genau dieser Farbe einzufärben. Als kleiner Zusatz wird bei einem Wechsel der Auswahl in einem Label zusätzlich zum Farbnamen der entsprechende RGB-Wert ausgegeben.

22.2.1 Die Problemstellung

Wie ich Ihnen im Kapitel zu den Windows Forms und GDI+ bereits geschildert habe, befinden sich in der `Color`-Struktur als statische Mitglieder ungefähr 140 Farbeinträge. Für ein solches Farbauswahlprogramm können Sie nun entweder (wie der arme Jayant) noch einmal alle diese Farben als `Color[]` in einem Array sammeln oder Reflection benutzen. Vor .NET und C# blieb mit einer anderen Programmiersprache nur die erste Möglichkeit, in C# dagegen kann ich etwas mehr grübeln und dafür mit wesentlich weniger Tipparbeit auskommen.

Wenn alles fertig ist, soll die Anwendung so aussehen, wie auf der folgenden Abbildung. Wenn Sie das Programm Stück für Stück mit mir entwickeln möchten, brauchen Sie also nur ein Formular, eine Listbox und ein Label.

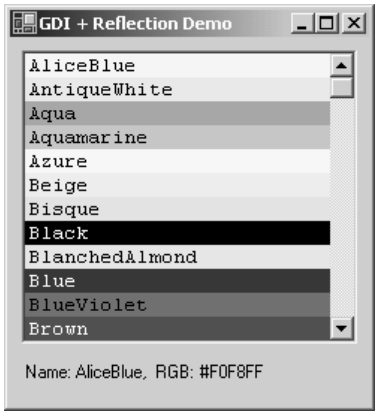


Abbildung 22.2: Die fertige Farbauswahl-Anwendung

Falls Sie sich über die merkwürdige Art der Ausgabe der Rot-, Grün- und Blauanteile der Farbe wundern: Dies ist die Art und Weise, wie Farben in HTML-Seiten angegeben werden, pro Farbkanal ein Byte in Hexadezimaldarstellung mit einem vorangestellten Nummernzeichen. Sollte Ihnen bei der Ausgabe also eine der Farben gefallen, können Sie diese Angabe für Ihren HTML-Editor verwenden.

22.2.2 Der Lösungsansatz

Die `Color`-Struktur ist ein Datentyp, also kann ich über die Reflection-Klassen auf die einzelnen Mitglieder des Typs zugreifen. Doch alles der Reihe nach, zuerst hole ich mir über `typeof` das Typobjekt für `Color`:

```
Type t = typeof(Color);
```

Der passende Zeitpunkt für das Auslesen der entsprechenden `struct`-Mitglieder ist für unser Beispiel das `Load`-Ereignis des Formulars. Der Name des Formulars lautet bei mir `frmMain`, die Listbox heißt `lbColors` und das Label `lblRGB`. Die komplette Methode für die Ereignisbehandlung sieht also wie folgt aus:

```
private void frmMainLoad(object sender,
                         System.EventArgs e)
{
   Type t = typeof(Color);
   PropertyInfo[] props;
```

```
   props = t.GetProperties(
                BindingFlags.Static
              | BindingFlags.Public
                        );

   foreach(PropertyInfo pi in props)
   {
      lbColors.Items.Add( pi );
   }
   lbColors.SelectedIndex = 0;
}
```

Listing 22.6: Die Ereignisbehandlungsroutine für frmMain_Load

Die Methode `GetProperties()` liefert für ein Typobjekt alle Eigenschaften (es gibt auch noch `GetInterfaces()`, `GetMethods()` usw.) in einem Array vom Datentyp `PropertyInfo`. Von diesen Methoden existiert jeweils auch eine überladene Version, bei der die Suchkriterien innerhalb des Typobjekts angegeben werden können (die Version ohne Parameter liefert nur alle `public`-Mitglieder, gleich ob statisch oder Instanzmitglied).

Durch die Angabe der `BindingFlags` `Static` und `public` wird die Suche auf statische, öffentliche Eigenschaften des Typs beschränkt, was genau die Auflistung der vordefinierten Farben liefert. Nachdem ich einige Informationen über die Eigenschaften später noch benötige, um die Werte auszulesen, füge ich das komplette `PropertyInfo`-Objekt der Listbox hinzu. Für diese Listbox habe ich die Eigenschaft `DrawMode` auf den Wert `OwnerDrawFixed` eingestellt, daher kann ich für die Ausgabe sowieso selbst bestimmen, was ich von diesen Informationen anzeigen lasse.

Dafür wird vom System für jeden Eintrag in der Listbox das Ereignis `DrawItem` ausgelöst, für das ich die folgende Ereignisbehandlungsroutine entworfen habe:

```
private void lbColorsDrawItem(object sender,
                              DrawItemEventArgs e)
{
   // wert der eigenschaft auslesen und in einer
   // variable vom typ Color speichern
   PropertyInfo pi = ((sender as ListBox).Items[e.Index]
                     as PropertyInfo);
   Color c = (Color) pi.GetValue(null, null);
```

```
    // die farbe fuer die beschriftung abhaengig von
    // der helligkeit des farbwertes einstellen
    Color c2 = ((byte)c.R+(byte)c.G+(byte)c.B <= 300) ?
                        Color.FromArgb(255, 255, 255) :
                        Color.FromArgb(0, 0, 0);

    // farbbalken fuer hintergrund zeichnen
    Brush brush = new SolidBrush(c2);
    e.Graphics.FillRectangle(new SolidBrush(c),
                             e.Bounds);

    // den Namen der Eigenschaft (Farbe) ausgeben
    e.Graphics.DrawString(pi.Name, e.Font,
                          brush, e.Bounds);
}
```

Listing 22.7: Die Routine für die Ausgabe eines Listbox-Eintrags

Hier findet der eigentliche trickreiche Teil statt. Mit dem `PropertyInfo`-Objekt kann ich über dessen Eigenschaft `Name` auf den Namen der Eigenschaft und damit den Farbnamen zugreifen, zusätzlich aber auch über einen Aufruf der Methode `GetValue()` auf den jeweiligen Wert dieser Eigenschaft (die Farbstruktur). Die beiden etwas seltsam aussehenden `null` als Parameter haben ihren Grund darin, dass es sich hier um statische, nicht indizierte Eigenschaften handelt. Der erste Parameter bekommt nämlich normalerweise die Instanz übergeben, für die der Wert der Eigenschaft bestimmt werden soll. Da bei einer statischen Eigenschaft nur ein Wert für den gesamten Typ gespeichert wird, kann hier ein `null` übergeben werden. Der zweite Paramater wäre ein entsprechendes Array mit Indices für indizierte Eigenschaften. Damit habe ich sowohl den Namen als auch den Wert der Eigenschaft zur Laufzeit bestimmt, ohne den Namen der Eigenschaft im voraus zu kennen!

Nun folgt der wahrscheinlich einzige Einsatz des Bedingungsoperators in diesem Buch (irgendwo musste ich das Ding einfach mal unterbringen). Ab einer bestimmten Helligkeit des Farbbalkens wird die Beschriftung in Schwarz angezeigt, darunter in Weiß, damit die Bezeichnungen der Farben auch zu erkennen sind. Der Rest ist aus dem Kapitel über die Windows Forms bereits bekannt und malt nur noch die Bezeichnung in das vorgegebene Rechteck.

Bleibt noch das Update des Labels mit dem RGB-Wert für die jeweils aktive Farbe. Nachdem es hier hauptsächlich um das Thema Reflection geht, habe ich auch keine großen Anstrengungen unternommen, die verschiedenen Zustände der Listbox (Focus, Auswahl, etc.) zu unterscheiden, das überlasse ich Ihnen als kleine Übung.

```
private void lbColorsUpdateLabel(object sender,
                                 System.EventArgs e)
{
   // Eigenschaftswert holen
   PropertyInfo pi = ((sender as ListBox).Items[
               (sender as ListBox).SelectedIndex
                ] as PropertyInfo);

   Color c = (Color) pi.GetValue(null, null);

   lblRGB.Text = "Name: " + pi.Name + ",  RGB: " +
   string.Format("#{0:X2}{1:X2}{2:X2}", c.R, c.G, c.B);
}
```

Listing 22.8: Aktualisierung des Labels nach einer neuen Auswahl

Denken Sie daran, im Quellcode (wenn Sie nicht den von der CD bevorzugen) den Namensraum `System.Reflection` auch zu importieren, andernfalls müssen Sie alle Bezeichner vollqualifiziert angeben.

22.3 Zusammenfassung

Die Reflection-Klassen bieten Ihnen weit über das bisher bekannte Konzept der Laufzeit-Typinformationen (run time type information) hinausgehende Möglichkeiten, Datentypen in Ihren Anwendungen zu untersuchen. Mit Hilfe der im Namensraum `System.Reflection.Emit` vorhandenen Klassen sind Sie sogar in der Lage, Datentypen und ganze Assemblies zur Laufzeit dynamisch zu erzeugen. Auf diese Weise lassen sich Programme erzeugen, die Programme erzeugen!

Damit besitzen Sie die Möglichkeit, zur Laufzeit Ihrer Anwendung andere Klassen nach Belieben zu untersuchen, Methoden aufzurufen oder Eigenschaften nach bestimmten Kriterien zu suchen sowie deren Wert zu ändern. In der Dokumentation zur .NET-Laufzeitumgebung bzw. zu C# finden Sie noch weitergehende Beispiele, unter anderem

das Durchsuchen eines Assemblys nach den darin enthaltenen Modulen und den in den Modulen liegenden Klassen.

Natürlich ist mit dem Erforschen von Klassen noch nicht Schluss mit dem Thema reflection. Eine sehr sinnvolle Anwendung werde ich Ihnen im nächsten Kapitel vorstellen, wenn es um das Thema Attribute und attribut-basierte Programmierung geht.

23 Attribute

23.1 Grundlagen

In jeder Anwendung sind zusätzlich zum Programmcode oft noch Informationen enthalten, die über die reinen Anweisungen der Programmausführung hinausgehen. Dies sind oft Informationen für die sprachübergreifende Programmierung, die Angabe von Speicherlayouts oder komplette externe Zusatzdateien, wie die IDL-Dateien bei der COM-Programmierung. Aber auch Compilerschalter für die bedingte Kompilierung oder die Angaben von Dateipfaden beim Import von DLLs fallen in diese Kategorien.

All diese deklarativen Informationen werden in C# mit einer Konstruktion gespeichert, die als Attribut bezeichnet wird. Diese Attribute sind eine der Voraussetzungen für das bereits mehrfach erwähnte 'One-Stop-Coding', bei dem die gesamten, für die Übersetzung nötigen Informationen in einer einzelnen Quellcodedatei vorliegen.

Prinzipiell ist ein Attribut nur eine Zusatzinformation, die zu einer Klasse oder einer Methode weitere Informationen liefert. Ich werde später gleich noch vorstellen, für welche Elemente von C# Sie Attribute vergeben können. Zuerst möchte ich aber anhand eines kleinen Beispiels die Angabe von Attributen vorstellen.

```
class Testklasse
{
   [Conditional("DEBUG")]
   public void EineMethode()
   {
      // code steht hier
   }
}
```

Listing 23.1: Der Einsatz des vordefinierten Attributs Conditional

Dieses, bereits in der .NET-Laufzeitumgebung vordefinerte Attribut `Conditional` entspricht in seiner Funktionsweise den Schaltern für die bedingte Kompilierung in anderen Programmiersprachen. Dies bedeutet,

dass die Methode `EineMethode` nur dann in die Klasse aufgenommen wird, wenn das Symbol `DEBUG` bei der Übersetzung angegeben wurde.

Diese Attribute und ihre Werte können über Reflection auch wieder während der Laufzeit ausgelesen werden. Darüber hinaus können Sie auch eigene, benutzerdefinierte Attribute erstellen, um so beispielsweise die Zuordnung einer Datenbankspalte zu einem Klassenattribut als Zusatzinformation in Ihrem Quellcode unterzubringen.

Aufgrund dieser universellen Natur der Attribute ist es unmöglich, hier alle Bedeutungen aufzulisten, die Sie in C#-Quelltexten sehen werden. Es gibt allerdings einige, bereits in der .NET-Laufzeitumgebung von Microsoft vordefinierte Attribute mit einer bestimmten Bedeutung.

23.2 Vordefinierte Attribute

Die folgenden Attribute werden vom C#-Compiler bereits erkannt und können von Ihnen ohne vorherige Deklaration im Code benutzt werden. Die genaue Bedeutung der einzelnen Attribute finden Sie in den einzelnen Abschnitten dieses Buches noch einmal im Zusammenhang aufgeführt. Sollten Sie also mit dem einen oder anderen Attribute noch nichts anfangen können, schlagen Sie einfach den Querverweis nach.

Denken Sie auch daran, dass der Name der Attributklasse in der .NET-Laufzeitumgebung mit einem »Attribute« endet. Die Klasse heißt also beispielsweise nicht `Conditional`, sondern `ConditionalAttribute`, falls Sie Details in der Onlinehilfe nachschlagen möchten.

Diese Liste enthält bei weitem nicht alle in der .NET-Laufzeitumgebung definierten Attribute, aber dennoch diejenigen, die Sie während 90% Ihrer Arbeit mit C# benötigen werden. Am Ende dieses Abschnitt finden Sie eine Liste der von der Klasse `Attribute` abgeleiteten Klassen der .NET-Laufzeitumgebung.

AttributeUsage

Gibt an, dass es sich bei der folgenden Klassendefinition um eine Attributklasse handelt. Details hierzu finden Sie im Abschnitt »***Das Attribut AttributeUsage***« weiter unten.

Conditional

Dieses Attribut steuert die bedingte Kompilierung. Als Parameter `ConditionString` wird ein String mit den Symbolen angegeben, die definiert sein müssen, um dieses Element in den Code aufzunehmen. Dieses Attribut kann pro Element nur einmal verwendet werden.

Beispiel:

```
[Conditional("DEBUG")]
```

DLLImport

Gibt an, dass das nachfolgend definierte Klassenmitglied als Routine aus einer nicht verwalteten DLL importiert wird. Neben dem Namen der entsprechenden Bibliothek können Sie, wie aus dem Beispiel ersichtlich, auch zusätzliche Optionen wie Zeichensatz oder Aufrufkonvention angeben.

Beispiel:

```
[DllImport("KERNEL32.DLL", EntryPoint="MoveFileW",
 SetLastError=true, CharSet=CharSet.Unicode,
 ExactSpelling=true,
 CallingConvention=CallingConvention.StdCall)]

public static extern
bool MoveFile(String src, String dst);
```

Flags

Legt fest, dass eine Aufzählung als Flags, dass heißt als Kombination aus Bits aufgefasst werden soll. Die einzelnen Elemente der Aufzählung lassen sich dann mit Hilfe des ODER-Operators (|) miteinander verknüpfen. Die .NET-Laufzeitumgebung ändert bei der Angabe dieses Attributs die Funktion von `ToString()` für eine Variable, die eine Aufzählung enthält. Weitere Details finden Sie im Abschnitt über die ***Aufzählungen***.

Beispiel:

```
[Flags]
```

NonSerialized

Gibt an, dass ein Feld einer Klasse nicht serialisiert werden soll, wenn eine entfernte Verarbeitung auftritt. Das markierte Mitglied wird bei der Serialisierung der Klasse übergangen.

Beispiel:

```
protected byte IchSchon;
[NonSerialized] protected int IchNicht;
```

Obsolete

Gibt an, dass die entsprechend markierte Konstruktion als veraltet anzusehen ist. Dieses Attribut besitzt neben der auszugebenden Meldung `message` noch einen optionalen Parameter `iserror`, der steuert, ob eine Warnung oder ein Kompilierfehler erzeugt wird. Dieses Attribut kann pro Element nur einmal verwendet werden.

Beispiel:

```
[Obsolete("Benutzen Sie in Zukunft Machwas2()", true)]
```

Serializable

Dieses Attribut gibt an, dass eine entsprechend markierte Klasse serialisiert werden kann. Alle Klassen, die nicht die Schnittstelle `ISerializable` implementieren, müssen mit diesem Attribut gekennzeichnet sein, um serialisiert werden zu können.

Beispiel:

```
[Serializable]
public class Test
{
}
```

Weitere Attribute

Innerhalb der .NET-Laufzeitumgebung sind eine ganze Reihe von zusätzlichen Attributen definiert und werden dort hauptsächlich für die Steuerung von Aufrufen und anderen Details eingesetzt. Im Anhang finden Sie eine Tabelle mit allen von der Klasse `System.Attribute` abgeleiteten Klassen.

23.3 Was passiert bei der Benutzung eines Attributes?

Wenn der C#-Compiler bei der Übersetzung des Quelltextes auf ein Attribut stößt, sucht er in den verfügbaren (d.h. mit `using` angegebenen) Namensräumen nach einer Klasse mit dem Namen des Attributes. Endet der Name des Attributes nicht auf 'Attribute', ergänzt der Compiler dies vor der Suche. Wird eine solche Klasse gefunden, stellt der Compiler fest, ob dieses Attribut an dieser Stelle im Quelltext zulässig ist (wie dies geschieht, erfahren Sie im nächsten Abschnitt). Ist die Verwendung nicht zulässig, wird ein Compilerfehler generiert. Daher sollten Sie bei der Erstellung von benutzerdefinierten Attributen sicherstellen, die richtige Auswahl für die erlaubte Platzierung des Attributes anzugeben.

23.4 Benutzerdefinierte Attribute

23.4.1 Definition der Attributklasse

Um selbst Attribute zu erstellen, benötigen Sie eine Klasse für das neue Attribut. Diese Klasse muss von der Klasse System.Attribute abgeleitet werden und der Name der Klasse muss dem Namen des Attributes, gefolgt von 'Attribute' entsprechen.

Um dem Compiler mitzuteilen, dass Sie hier keine normale Klasse, sondern ein Attribut erstellen, wird natürlich auch wieder ein Attribut benutzt. Dieses Attribut legt fest, wofür Sie das Attribut erstellen, ob dieses mehrfach verwendet werden kann und vererbt wird. Das folgende Beispiel definiert eine Klasse für ein Attribut `RevisionHistory`, mit dem Sie unter einer URL eine HTML-Seite mit Informationen zur Entwicklung einer Klasse bereitstellen können. Damit ist auch gleich festgelegt, dass Sie dieses Attribut nur einer Klasse zuweisen können und es auch nur einmal pro Klasse verwendet werden kann.

```
[AttributeUsage(AttributeTargets.Class,
 AllowMultiple=false, Inherited=false)]
public class RevisionHistoryAttribute : Attribute
{
   private string theURL;
   public RevisionHistoryAttribute(string URL)
```

```
    {
        this.theURL = URL;
    }

}
```

Listing 23.2: Erstellen eines benutzerdefinierten Attributes

Mit dieser Klasse können Sie anschließend das Attribut `RevisionHistory` in Ihrem Quelltext benutzen:

```
[RevisionHistory("http://myrcs/projektXYZ/class1.htm")]
class class1
{
    // mitglieder der klasse
}
```

Listing 23.3: Benutzen des Attributes RevisionHistoryAttribute

23.4.2 Das Attribut AttributeUsageAttribute

Wie ich bei der Definition der Attributklasse geschildert habe, ist dieses Attribut notwendig, um dem C#-Compiler mitzuteilen, dass es sich bei der folgenden Klasse um die Definition eines Attributes handelt. Dieses Attribut benötigt mindestens einen Parameter, nämlich die möglichen Ziele für dieses Attribut.

Im Aufzählungsdatentyp `System.AttributeTargets` finden Sie eine Aufzählung aller möglichen Ziele für ein benutzerdefiniertes Attribut. Die einzelnen Werte der Aufzählung können von Ihnen bei Bedarf mit Hilfe des logischen Oder-Operators (|) auch verknüft werden. Folgende Auswahlmöglichkeiten stehen Ihnen in der Aufzählung zur Verfügung:

Zielname	Beschreibung
All	Attribut kann allen Elementen zugewiesen werden.
Assembly	Attribut kann einem Assembly zugewiesen werden.
Class	Attribut kann einer Klasse zugewiesen werden.
Constructor	Attribut kann einem Konstruktor zugewiesen werden.

Tabelle 23.1: Mögliche Ziele für die Definition eines Attributes

Zielname	Beschreibung
Delegate	Attribut kann einem Delegate zugwiesen werden.
Enum	Attribut kann einer Aufzählung zugewiesen werden.
Event	Attribut kann einem Ereignis zugewiesen werden.
Field	Attribut kann einem Feld zugewiesen werden.
Interface	Attribut kann einer Schnittstelle zugewiesen werden.
Method	Attribut kann einer Methode zugewiesen werden.
Module	Attribut kann einem Modul zugewiesen werden.
Parameter	Attribut kann einem Parameter zugewiesen werden.
Property	Attribut kann einer Eigenschaft zugewiesen werden.
ReturnValue	Attribut kann einem Rückgabewert zugewiesen werden.
Struct	Attribut kann einem Werttyp zugewiesen werden.

Tabelle 23.1: Mögliche Ziele für die Definition eines Attributes (Forts.)

Die beiden folgenden Parameter des Attributes `AttributeUsageAttribute` sind optional und können daher auch weggelassen werden.

Der optionale Parameter `AllowMultiple` legt fest, ob dieses Attribut nur einmal oder mehrmals dem gleichen Ziel zugewiesen werden kann. In unserem Beispiel wäre eine mehrfache Verwendung eher verwirrend, daher habe ich diesen Parameter auf `false` gesetzt. Vorgabewert ist ebenfalls `false`, also hätte ich dies auch weglassen können. Allerdings hätte ich dann kein Beispiel mehr für einen benannten Parameter dieses Attributes gehabt …

Der Parameter `Inherited` schießlich bestimmt die Vererbung des Attributes. Jawohl, Sie können bei Bedarf ein Attribut an alle abgeleiteten Elemente des Ursprungselements vererben lassen. Bei einer Klasse sind dies alle abgeleiteten Klassen, bei einer Methode alle Methoden, die die Methode überschreiben usw. Auch hier ist der Standardwert wieder `false`.

23.4.3 Optionale Parameter in eigenen Attributen

Auch für die von Ihnen definierten Attribute können Sie natürlich optionale, benannte Parameter (mit Vorgabewerten) vergeben. Dazu müssen wir unsere Klasse nur etwas erweitern. Für das Beispiel ergänze ich einen optionalen Parameter mit einer eMail-Adresse eines armen Supporters, der für die Aktualisierung der Seiten zuständig ist.

```
[AttributeUsage(AttributeTargets.Class,
 AllowMultiple=false, Inherited=false)]
public class RevisionHistoryAttribute : Attribute
{
   private string theURL;

   public RevisionHistoryAttribute(string URL)
   {
      this.theURL = URL;
      this.theAddr = "";
   }

   // oeffentliche eigenschaft für den
   // optionalen parameter
   public string email
   {
      get
      {
         return theAddr;
      }
      set
      {
         theAddr = value;
      }
   }

}
```

Listing 23.4: Ergänzung des Attributes um einen optionalen Parameter

Es hat sich in der Klasse nicht viel geändert. Ich habe lediglich eine öffentliche Eigenschaft ergänzt. Das allein reicht aus, um einen optionalen Parameter mit dem Namen dieser Eigenschaft benutzen zu können:

```
[RevisionHistory("http://myrcs/projektXYZ/class1.htm",
  email="support@documentation.com")]
class class1
{
   // mitglieder der klasse
}
```

Listing 23.5: So kann der optionale Parameter benutzt werden

Was passiert hier? Der Compiler erkennt anhand der Angabe des benannten Parameters, dass dieser nicht dem Konstruktor zu übergeben ist. Stattdessen sucht er innerhalb der Klasse nach einem öffentlichen Feld oder einer öffentlichen Eigenschaft mit dem entsprechenden Namen.

Ich hätte also genauso gut anstelle der Eigenschaft einfach das Feld als `public` deklarieren können. Dies würde allerdings den Prinzipien der objektorientierten Programmierung widersprechen, nach denen von außen keine Felder einer Klasse geändert werden können. Ein Zugriff sollte nur über Methoden oder Eigenschaften erfolgen. Gewöhnen Sie sich dies am besten gleich zu Beginn der Arbeit mit C# an, später werden Sie froh über diese Angewohnheit sein, wenn alle Ihre Kollegen abends noch nach Fehlern suchen und Sie bereits den Feierabend genießen …

23.5 Abrufen von Attributinformationen

Für einen sinnvollen Einsatz von Attributen muss natürlich auch eine Möglichkeit vorhanden sein, diese Informationen zur Laufzeit der Anwendungen wieder abfragen zu können. Dies geschieht über die Abfrage der Attributinstanzen. Für den Compiler ist die Deklaration von Informationen mit Attributen vom Prinzip her nichts anderes als eine Klassendefinition. Daher sind Sie in der Lage, über Reflection diese Attribute auch abzufragen. Die wichtigste Methode dazu ist `Attribute.GetCustomAttributes` im Namensraum `System`. Bevor ich Ihnen den entsprechenden Code vorstelle, sehen wir uns zuerst an, was eigentlich intern beim Abfragen eines Attributes passiert.

23.5.1 Abfragen eines Attributes

Die Attributspezifikation

```
[RevisionHistory("http://myrcs/projektXYZ/class1.htm")]
```

entspricht intern der folgenden Konstruktion

```
RevisionHistoryAttribute revhist =
    new RvisionHistoryAttribute("<url>");
```

Es macht allerdings keinen Sinn, bereits beim Start der Anwendung für alle Attribute eine Instanz zu erzeugen. Schließlich kann der Fall eintreten, dass kein einziges Attribut abgefragt wird und die .NET-Laufzeitumgebung benötigt auch so schon genug Ressourcen Ihres Entwicklungssystems. Daher wird diese Instanziierung der Attributklasse erst dann ausgeführt, wenn Sie das Attribut abfragen.

Der Aufruf von `GetCustomAttributes` bewirkt eine Instanziierung aller Attributklassen für das entsprechende Objekt. Diese Instanzen werden dann von der Methode als Array von Attributen zurückgeliefert. Dieses Array können Sie anschließend in einer Schleife durchlaufen und die einzelnen Felder abfragen.

23.5.2 Abrufen der Informationen mit Reflection

Hier nun der Code für diese Zielsetzung. Zuerst hole ich mir durch den Aufruf von `GetCustomAttributes` ein Array mit allen für diesen Typ oder dieses Klassenmitglied definierten Attributen. Da die Methode `Main` eine statische Methode ist, kann hier natürlich kein `this` verwendet werden, sondern es ist der Operator `typeof` zu benutzen, um an den Typ für die Klasse `Class1` zu kommen. Anschließend werden in einer Schleife alle Attribute mit ihrem Klassennamen ausgegeben (nachdem wir nur ein Attribut haben und dies auch noch `AllowMultiple` auf `false` gesetzt hat, sollte die Schleife tunlichst nur eine Zeile als Ausgabe produzieren). Zusätzlich prüfe ich in der Schliefe noch mit dem Typoperator `is`, ob in der Liste der Attribute eine Instanz der Klasse `RevisionHistoryAttribute` vorkommt und gebe dann den Inhalt aus.

```
// Goto C# -- Armin Hanisch
// benutzerdefinierte attribute abfragen

[RevisionHistory("http://myrcs/projektXYZ/Class1.htm")]
class Class1
{
   static void Main(string[] args)
   {
      // auslesen des attributes über reflection
      Attribute[] attrs =
        Attribute.GetCustomAttributes(typeof(Class1));
      foreach(Attribute a in attrs)
      {
         Console.WriteLine(a.ToString());
         if(a is RevisionHistoryAttribute)
         {
            RevisionHistoryAttribute revhist =
                        (RevisionHistoryAttribute)a;
            Console.WriteLine(revhist.URL);
         }
         Console.ReadLine();
      }
   }
}
```

Listing 23.6: Die Abfrage der benutzerdefinierten Attribute

23.5.3 Feststellen, ob ein Attribut definiert ist

Es ist nicht die beste Praxis, auf gut Glück einfach Attribute abzufragen und zu hoffen, dass Ihre Kollegen auch brav alle notwendigen Metadaten angegeben haben. Daher definiert die .NET-Laufzeitumgebung auch Methoden, um zu prüfen, ob ein Attribut zugewiesen wurde. Dieses eine Attribut können Sie dann auch gezielt mit der Methode `GetCustomAttribute` (achten Sie auf das fehlende Plural-s!) abrufen.

```
// Goto C# -- Armin Hanisch
// abfrage, ob ein attribut definiert ist

// bitte beachten: diesmal ist kein attribut definiert
// [RevisionHistory("http://myrcs/default.htm")]
class Testclass
{
   static void Main(string[] args)
```

```
   {
      // auslesen des attributes über reflection
      if(Attribute.IsDefined(typeof(Testclass),
         typeof(RevisionHistoryAttribute)))
         {
            RevisionHistoryAttribute revhist =
                 Attribute.GetCustomAttribute(
                     typeof(Testclass),
                     typeof(RevisionHistoryAttribute))
                     as RevisionHistoryAttribute;
            Console.WriteLine(revhist.URL);
         }
         else
            Console.WriteLine("Das Attribut " +
               "RevisionHistoryAttribute ist " +
               "nicht deklariert.");
         Console.ReadLine();
   }
}
```

Listing 23.7: Abfragen eines bestimmten benutzerdefinierten Attributes

23.6 Zusammenfassung

Attribute dienen der Aufnahme deklarativer Informationen in den Quelltext und erlauben mit C# so den Verzicht auf externe Headerdateien, Deklarationen (wie IDL bei der COM-Programmierung) und Hilfsmittel anderer Programmiersprachen.

Dieses Attributsystem der .NET-Laufzeitumgebung ist flexibel durch den Entwickler erweiterbar, Sie können also jederzeit eigene Attribute ergänzen, beispielsweise für die Zuordnung von Datenbankspalten zu einzelnen Feldern einer Klasse oder URLs mit Zusatzinformationen zu einzelnen Methoden.

Abfragen lassen sich diese Attribute mit Hilfe von Reflection und der Klasse `Attribute` aus dem Namensraum `System`. Dabei lassen sich sowohl alle benutzerdefinierten Attribute einem Array zurückliefern als auch gezielt bestimmte Attribute abfragen.

24 Überladen von Operatoren

Nein, damit ist kein überarbeiteter Systemverwalter gemeint, sondern in C# lassen sich Klassen und Strukturen mit Methoden ausstatten, welche die Neudefinition von Operatoren erlauben. Damit können Sie diese Operatoren mit eigenen Datentypen benutzen und so Ihren Quelltext leichter erfassbar machen.

Das Überladen von Operatoren ist ein relativ fortgeschrittenes Konzept bei einer Programmiersprache und Sie werden 90 bis 95% Ihrer Arbeit auch ohne dieses Thema erledigen können. Wenn Sie also gerade keine Zeit haben, können Sie dieses Kapitel auch erst bei Bedarf durchlesen, da ich das Thema Typkonvertierungen und Überladen von Konvertierungsoperatoren in einem eigenen Kapitel behandle.

Das folgende kleine Codestück demonstriert die Vorteile von Operatoren für eigene Datentypen, der Code kann dadurch lesbarer werden. Natürlich hat die Sache auch Schattenseiten, stellen Sie sich vor, ein Entwickler überlädt + mit der Funktion von – und andere nette Dinge. Aber dazu komme ich am Ende des Kapitels.

```
MeinTyp xyz;
MeinTyp abc;

xyz = new MeinTyp(345);
abc = new MeinTyp();
// abc = abc + xyz
// fuktioniert ohne überladung nicht
// daher umständlich und weniger intuitiv:
abc.Add(xyz);
```

Listing 24.1: Ohne überladene Operatoren müssen Methoden benutzt werden

24.1 Wie funktioniert Operatorüberladung?

Dafür existiert in C# das Schlüsselwort `operator`, mit dem Sie eine statische Methode in Ihrer Klasse oder Struktur deklarieren, welche die entsprechende Arbeit des Operators erledigt. Je nach Anzahl der Operanden muss die Methode einen oder zwei Parameter akzeptieren und verarbeiten.

Die allgemeine Form der Überladung sieht so aus:

```
public static <datentyp>
operator <operator> ( <parameterliste> )
```

In den nächsten Abschnitten werden Sie gleich einige praktische Beispiele dazu sehen.

24.2 Was kann überladen werden?

Im Gegensatz zu C++ ist die Auswahl der überladbaren Operatoren eingeschränkt. Nicht, weil C# diese Funktionalität fehlen würde, sondern um den Quellcode lesbar zu halten und die schlimmsten Auswüchse kreativer 'Überladungswut' von vorneherein zu verhindern. In vielen Fällen, bei denen in anderen Sprachen Operatoren überladen wurden, ist es bei C# nicht mehr nötig. So wurden in C++ beispielsweise die eckigen Klammern oft überladen, in C# existieren für diesen Zweck die Indexer.

24.2.1 Unäre Operatoren

Überladen werden können die Operatoren `+`, `–`, `!`, `~`, `++`, `--`, `true`, `false`. Die ersten Operatoren sind wahrscheinlich einsichtig, daher möchte ich zuerst die beiden 'Operatoren' `true` und `false` behandeln.

Der Grund für eine mögliche Überladung ist ein Konzept, das 'dreiwertige Logik' genannt wird. Damit ist nicht gemeint, dass Computer entweder funktionieren, nicht funktionieren oder eben nur manchmal funktionieren, sondern dass es neben `true` und `false` noch einen Zustand 'nicht vorhanden' gibt. Wenn Sie sich schon einmal mit SQL-Datenbanken beschäftigt haben, ist Ihnen das sicher bekannt. In einem logischen Datenfeld kann entweder wahr oder falsch stehen oder das Feld ist noch gar nicht belegt, also weder das eine noch das andere (in SQL heißt dieser Zustand `NULL`). Damit ist die normale zweiwertige Aussage »wahr ist **nicht falsch**« eben nicht mehr möglich.

Langer Rede kurzer Sinn: Für eigene logische Typen müssen Sie die beiden Operatoren `true` und `false` überladen, wobei jede Methode einen `bool`-Wert zurückliefern muss. C# wertet diese dreiwertigen Ausdrücke dann nämlich über den Rückgriff auf `true` und `false` aus.

C++



An dieser Stelle auch gleich noch ein Hinweis an die C++-Entwickler. Aus Gründen der Konsistenz unterscheidet C# bei den Inkrement- und Dekrement-Operatoren nicht zwischen der Präfix- und der Postfix-Version. Beide sind nicht unterscheidbar, da Operatorüberladungen als statische Methoden definiert werden.

24.2.2 Binäre Operatoren

An dieser Stelle behandle ich die arithmetischen und logischen Operatoren, die relationalen Operatoren folgen gleich im nächsten Abschnitt.

Sie können die folgenden Operatoren überladen: `+`, `–`, `*`, `/`, `^`, `%`, `&`, `|`, `<<`, `>>`. Alle Methoden zum Überladen binärer Operatoren müssen zwei Parameter akzeptieren (wobei zumindest einer vom Typ sein muss, für den der Operator überladen wird). Der Rückgabewert muss nicht notwendigerweise der Typ sein, für den Sie den Operator überladen, aber in 99% aller Fälle wird das der Fall sein (*Ich versuche immer noch, ein schönes Beispiel zu finden, wo ein anderer Rückgabetyp Sinn macht. Falls Sie eines wissen, schicken Sie mir eine E-Mail*).

24.2.3 Relationale Operatoren

Überladen werden können die Operatoren `==`, `!=`, `<=`, `>=`, `<` und `>`. Die beiden am häufigsten überladenen Operatoren werden wahrscheinlich auch in Ihren Anwendungen `==` und `!=` sein, damit eine Prüfung auf Übereinstimmung möglich wird. Das Standardverhalten dieser beiden Operatoren hängt davon ab, ob Sie Werttypen oder Referenztypen miteinander vergleichen. Bei einem Werttyp wird der Inhalt verglichen, bei den Referenztypen wird verglichen, ob die Referenz übereinstimmt, also beide auf die gleiche Adresse zeigen.

Als einziger Referenztyp bietet `string` hier bereits einen Service für den Entwickler an: die Operatoren sind bereits 'passend' überladen, so dass der Inhalt eines Strings und nicht seine Referenz getestet wird. Weitere Informationen dazu finden Sie in Kapitel 11 ***Vererbung***.

Auch bei der Überladung von Operatoren zeigt sich wieder, dass es dem Entwickler in C# nicht so einfach gemacht wird, sich selbst in den Fuß zu schießen. Im Gegensatz zu C++ ist für `==` und `!=` nur das 'Kom-

plettpaket' zu haben: Entweder Sie überladen beide Operatoren und sorgen so für eine konsistente Prüfung auf Übereinstimmung oder Nichtübereinstimmung oder Sie überladen keinen der beiden Operatoren.

24.2.4 Ein Beispiel zur Operatorüberladung

Die folgende Struktur redefiniert zwei Operatoren zur Benutzung mit diesem Datentyp. Die Struktur `Farbe` speichert Farbangaben im RGB-Farbraum mit einem Bytewert pro Farbkanal (HTML-Designern wird das bekannt vorkommen). Bei der Addition zweier Farben werden die Werte für die einzelnen Farbkanäle addiert, aber nur bis zum Maximalwert von 255 pro Kanal (weißer als weiß geht nicht bei der Lichtmischung). Bei der Subtraktion werden die Werte für die Farbkanäle analog dazu bis minimal 0 subtrahiert. Eine kleine Testklasse verdeutlicht das Ergebnis.

```
/*
   Goto C# -- Armin Hanisch
   beispiel zur überladung von operatoren
*/

using System;

namespace OpOver
{

   struct Farbe
   {
      byte fRot;
      byte fGruen;
      byte fBlau;

      public Farbe(byte rot, byte gruen, byte blau)
      {
         fRot = rot;
         fGruen = gruen;
         fBlau = blau;
      }


      public override string ToString()
```

```
{
   return String.Format("#{0:X2}{1:X2}{2:X2}",
                        fRot, fGruen, fBlau);
}


public static Farbe operator + (
                     Farbe links, Farbe rechts)
{
   Farbe ergebnis;
   int temp;

   temp = (links.fRot + rechts.fRot);
   ergebnis.fRot =
     temp > 255 ? (byte)255 : (byte)temp;

   temp = (links.fGruen + rechts.fGruen);
   ergebnis.fGruen =
     temp > 255 ? (byte)255 : (byte)temp;

   temp = (links.fBlau  + rechts.fBlau);
   ergebnis.fBlau  =
     temp > 255 ? (byte)255 : (byte)temp;

   return ergebnis;
}


public static Farbe operator - (
                     Farbe links, Farbe rechts)
{
   Farbe ergebnis;

   temp = (links.fRot - rechts.fRot);
   ergebnis.fRot =
     temp < 0 ? (byte)0 : (byte)temp;

   temp = (links.fGruen - rechts.fGruen);
   ergebnis.fGruen =
     temp < 0 ? (byte)0 : (byte)temp;

   temp = (links.fBlau - rechts.fBlau);
   ergebnis.fBlau  =
     temp < 0 ? (byte)0 : (byte)temp;
```

```
            return ergebnis;
        }
    }

    // die testklasse zum ausprobieren
    class Testklasse
    {
        static void Main(string[] args)
        {
            Farbe f1, f2;
            f1 = new Farbe(051, 153, 153);
            f2 = new Farbe(051, 102, 153);

            Console.WriteLine("Farbe 1 = {0}", f1 );
            Console.WriteLine("Farbe 2 = {0}", f2 );
            Console.WriteLine("Farbe 1 + Farbe 2 = {0}",
                              f1+f2 );
            Console.WriteLine("Farbe 1 - Farbe 2 = {0}",
                              f1-f2 );
        }
    }
}
```

Listing 24.2: Die Beispielklasse für das Überladen von Operatoren

24.2.5 Überladen oder nicht überladen?

Für einige Operatoren stellt sich diese Frage nicht, da sie in C# nicht überladen werden können. Die Menge der überladbaren Operatoren wurde vom C#-Team bei Microsoft nach den Gesichtspunkten der Lesbarkeit und Konsistenz von Quelltexten und der praktischen Erfahrung ausgewählt. So können auch die Abkürzungsoperatoren wie += und -= nicht überladen werden, da das einfach eine kürzere Schreibweise für die Kombination aus Operator und Zuweisung darstellt und die binären Operatoren wie + oder – können Sie ja überladen.

Für alle überladbaren Operatoren sollten Sie überlegen, ob diese Möglichkeit von C# einen Nutzen für den Benutzer ihrer Klassen und Strukturen bringt. Wird es einfacher, eine Berechnung auszudrücken oder nicht? Bei Datentypen wie Polarkoordinaten, komplexen Zahlen oder Längen- und Breitengraden kann es durchaus sinnvoll sein, einige Operatoren zu überladen. In vielen Fällen müssen Sie für Vergleiche die beiden relationalen Operatoren == und != überladen.

Keinen Sinn macht ein Überladen, wenn dadurch der Operator eine andere oder gar die gegenteilige Bedeutung bekommt. So würde bei der Struktur `Farbe` der unäre Operator ++ keinen Sinn ergeben, denn was ist 'die nächste Farbe'? Obwohl hier noch mit einem 'Heller' oder 'Dunkler' argumentiert werden kann, ist für mich bereits die Grenze der Selbstdokumentation überschritten. Zeigen Sie einfach einem Kollegen oder einer Kollegin ein Codestück, in dem die überladenen Operatoren benutzt werden. Wird der Code ohne Erklärung verstanden, gut. Müssen Sie erst lange erklären, was der Operator macht, benutzen Sie lieber Methoden für diese Funktionalität.

24.3 Zusammenfassung

In C# können Sie eine Vielzahl von Operatoren überladen und damit eigene Datentypen mit diesen Operatoren benutzen. Die Menge der überladbaren Operatoren ist aus Gründen der Lesbarkeit des Quellcodes eingeschränkt, reicht aber für den praktischen Einsatz vollkommen aus.

Sie definieren eine Überladung für einen Operator mit einer öffentlichen statischen Methode und dem Schlüsselwort `operator`, gefolgt vom Operatorsymbol. Je nach Anzahl der Operanden muss diese Methode einen oder zwei Parameter akzeptieren.

Bestimmte Operatoren sind nur zusammen überladbar, um Inkonsistenzen zu vermeiden.

25 Serialisierung

Vor der objektorientierten Programmierung wurde die Serialisierung, also das Überdauern eines Progammendes durch die Daten, dadurch gewährleistet, dass aktuelle Daten und Variableninhalte in eine Datei geschrieben wurden. Der Code lag in der ausführbaren Datei vor und nach dem erneuten Start wurde die Datendatei geöffnet und der Entwickler hatte die Aufgabe, alle Daten wieder in die richtigen Strukturen zu laden.

In der objektorientierten Programmierung bilden Daten und Code logische Einheiten, eben die einzelnen Objekte. Der Grundgedanke der Serialisierung ist der, dass ein Objekt selbst dafür verantwortlich und in der Lage ist, seinen aktuellen Status auf einen persistenten Speicher zu schreiben. Später kann das Objekt anstelle einer kompletten Neuerzeugung aus diesem Speicher gelesen und mit den Daten wiederhergestellt werden. Dieser Prozess wird als Deserialiserung bezeichnet.

Allerdings ist Serialisierung nicht nur für das Speichern von Objektinstanzen bis zum nächsten Start der Anwendung notwendig. Sobald eine Anwendung auf mehr als einem Rechner bzw. über mehr als einen Prozess hinweg arbeitet, muss eine Möglichkeit vorhanden sein, das Objekt zu übertragen (remoting). Daher muss ein Objekt hierfür ebenfalls seine Daten 'transportsicher' verpacken und innerhalb des Zielprozesses wieder auspacken.

25.1 Attribute oder ISerializable?

Damit eine Instanz einer Klasse serialisiert werden kann (besser, sich selbst serialisieren kann), haben Sie als Entwickler zwei Möglichkeiten, jeweils mit den in der EDV üblichen Vor- und Nachteilen. Sie können mehr oder weniger alles der .NET-Laufzeitumgebung überlassen, haben wenig Arbeit, allerdings auch keinen Einfluss auf die Serialisierung. Alternativ behalten Sie die größtmögliche Kontrolle, müssen dafür aber einen erhöhten Aufwand einplanen. In dieser Beziehung hat sich also in der EDV seit einem halben Jahrhundert nichts geändert …

Die einfache Methode nutzt entsprechende Attribute, ich werde diesen Ansatz gleich in einem Beispiel zeigen. Die zweite Methode erfordert die Implementierung der Schnittstelle `ISerializable` und wird etwas später behandelt. Zuerst jedoch benötigen wir für die Beispiele eine Testklasse.

25.2 Die Beispielklasse

Für die folgenden Beispiele benutze ich eine kleine Klasse mit minimalem Inhalt: ein Konstruktor, zwei private Felder und eine Methode, mit der die beiden privaten Felder ausgegeben werden. Hier der Code der Klasse, noch ohne jede Vorbereitung für die Serialisierung:

```
class DieKlasse
{
   private int[] zahlen;
   private double geheim;

   public DieKlasse(int number)
   {
      zahlen = new int[number];
      for(int i = 0; i < number; i++)
      {
         zahlen[i] = i*i;
      }
      Console.Write("double-Wert eingeben : ");
      geheim = Convert.ToDouble(Console.ReadLine());
   }

   public void Ausgabe()
   {
      for(int i = 0; i < zahlen.Length; i++)
      {
         Console.WriteLine("Position {0} = {1}",
                           i, zahlen[i]);
      }

      Console.WriteLine("double-wert = {0,6:0.00}",
                        geheim);
   }

}
```

Listing 25.1: Die Beispielklasse für die Serialisierung

25.3 Attributgesteuerte Serialisierung

25.3.1 Serializable und NonSerialized

In der Standardeinstellung der .NET-Laufzeitumgebung kann eine Klasse nicht serialisiert werden. Um diese Funktionalität zu erhalten, muss die Klasse mit dem Attribut `Serializable` markiert sein. Näheres zum Thema Attribute erfahren Sie in Kapitel 23 ***Attribute***. Nachdem eine Klasse mit diesem Attribut markiert wurde, gelten alle Felder und Mitglieder dieser Klasse als serialisierbar. Dies ist allerdings nicht in allen Fällen sinnvoll. Beispielsweise können einzelne Felder der Klasse Daten enthalten, die nur für die laufende Session (IDs, Schlüsselpaare, Cookies, etc.) von Interesse sind und nicht serialisiert werden sollten.

Diese Felder, die während der Serialisierung ignoriert werden sollen, müssen mit einem zweiten Attribut, `NonSerialized`, markiert werden. Für unser Beispiel soll das Array mit den Integerwerten serialisiert werden, der Doublewert `geheim` aber nicht. Nach der Markierung mit den Attributen sieht die Klasse wie folgt aus:

```
// Goto C# -- Armin Hanisch
// klasse mit attributen für die serialisierung

[Serializable]
class DieKlasse
{
   private int[] zahlen;
   [NonSerialized] private double geheim;

   public DieKlasse(int number)
   {
      zahlen = new int[number];
      for(int i = 0; i < number; i++)
      {
         zahlen[i] = i*i;
      }
      Console.Write("double-Wert eingeben : ");
      geheim = Convert.ToDouble(Console.ReadLine());
   }

   public void Ausgabe()
   {
```

```
      for(int i = 0; i < zahlen.Length; i++)
      {
         Console.WriteLine("Position {0} = {1}",
                           i, zahlen[i]);
      }

      Console.WriteLine("double-wert = {0,6:0.00}",
                        geheim);
   }

}
```

Listing 25.2: Die Beispielklasse für die Serialisierung mit Attributen

25.3.2 Formatter-Klassen

Für die eigentliche Serialisierung wird nun noch ein Transportmechanismus benötigt, der eine entsprechende Formatierung der Daten (binär, textuell oder XML) vornimmt. Für die ersten Beispiele werde ich die Daten in eine Binärdatei schreiben und dazu eine Instanz der Klasse `BinaryFormatter` verwenden. Diese Klasse sowie die anderen benötigen Datentypen finden Sie in den folgenden Namensräumen:

- System.Runtime.Serialization;
- System.Runtime.Serialization.Formatters.SOAP;
- System.Runtime.Serialization.Formatters.Binary;

Im folgenden Beispiel erfolgt die Serialisierung in eine Binärdatei auf der Platte. Für andere Anwendungen kann aber, wie bereits erläutert, der Transport in einen anderen Prozess oder auf eine andere Maschine nötig sein. Daher existiert neben dem parameterlosen Konstruktor noch eine zweite Überladung. Diese akzeptiert neben einem `ISurrogateSelector`, den wir hier nicht behandeln, einen so genannten `StreamingContext`.

Dieser Wert gibt an, wie die Serialisierung vonstatten geht, so dass Sie beispielsweise zur Laufzeit entscheiden können, ob Sie ein Fensterhandle serialisieren möchten oder nicht. Handelt es sich um ein Streaming von Prozess zu Prozess auf der gleichen Maschine, behält das Handle durchaus seine Gültigkeit, beim Transport auf eine andere Maschine würden Sie dieses Handle aber mit Sicherheit nicht verwen-

den können. Dieser `StreamingContext` ist eine Aufzählung und besitzt die folgenden Mitglieder:

Aufzählungs-element	Beschreibung
All	Die serialisierten Daten können zu jedem beliebigen Kontext aus der Aufzählung transportiert werden oder von dort stammen.
Clone	Das Objekt soll gecloned werden und wird anschließend im gleichen Prozessraum weiterlaufen, daher können z.B. Handles oder ähnliche Zugriffskennungen weiter benutzt werden.
CrossAppDomain	Der Kontext für Quelle oder Ziel sind in verschiedenen Application Domains (mehr dazu bei Threading und AppDomains).
CrossMachine	Quell- oder Zielkontext befinden sich auf unterschiedlichen Computern.
CrossProcess	Quell- und Zielkontext befinden sich in zwei verschiedenen Prozessräumen auf dem gleiche Computer.
File	Quell- oder Zielkontext ist eine Datei. Es wird empfohlen davon auszugehen, dass die Lebensdauer der Datei länger ist als die des erzeugenden Prozesses, so dass keine Informationen in die Datei serialisiert werden sollten, welche bei der Deserialisierung noch auf den Prozess zugreifen müssen.
Other	Unbekannter Kontext für die Serialisierung.
Persistence	Die allgemeine Form von `File`, wobei das gleiche gilt wie unter `File`.
Remoting	Gibt an, dass der Zielkontext eine unbekannte entfernte Prozessumgebung ist. Es sollten keine Annahmen darüber getroffen werden, ob es sich bei Quell- oder Zielkontext um den gleichen Computer handelt.

Tabelle 25.1: Die verschiedenen Möglichkeiten für den StreamingContext

25.3.3 Der Prozess der Serialisierung

Bevor wir eine Instanz der Klasse serialisieren, wird erst eine Instanz dieser Klasse benötigt. Daher beginnt unsere Beispielmethode `Serialize` mit folgendem Code:

```
// eine instanz der klasse erzeugen und ausgeben
DieKlasse cls = new DieKlasse(5);
cls.Ausgabe();
```

Damit wird das private Feldarray `zahlen` mit dem Quadrat der Zahlen von 0 bis 4 gefüllt und vom Benutzer eine Eingabe für das `double`-Feld `geheim` erwartet. Jetzt muss die entsprechende Formatter-Instanz erzeugt werden. Im Beispiel wird diese an einen `FileStream` angehängt:

```
// ein formatierer fuer die serialisierung
BinaryFormatter bf = new BinaryFormatter();

// die datei, in die serialisiert wird
FileStream fs = new FileStream("klasse.dat",
                FileMode.OpenOrCreate,FileAccess.Write);
```

Das wars auch schon an Vorbereitung. Alles was noch fehlt, ist der Aufruf der Methode `Serialize` des Formatierers, danach muss der Dateistrom nur noch geschlossen werden und die Binärdatei enthält die serialisierte Instanz der Klasse.

```
// klasse in den dateistrom serialisieren
bf.Serialize(fs, cls);
fs.Flush();
fs.Close();
```



Abbildung 25.1: Der Inhalt der Datei klasse.dat

Auf der Platte können wir uns die Datei KLASSE.DAT nun mit einem Hex-Editor ansehen, um einmal zu sehen, was sich aus dieser Datei erkennen lässt (*auch in Zeiten hochsprachlicher OOP-Entwicklung habe ich mir meine binäre Neugier nie abgewöhnen können …*)

Sie sehen, in der Datei finden Sie den kompletten Namen des Assemblies, den Namensraum und natürlich die Namen und die Werte der serialisierten Felder (in diesem Fall `zahlen`). Das Feld `geheim` dagegen ist in dieser Datei nicht enthalten, da wir das Attribut `[NonSerialized]` dafür verwendet hatten.

25.3.4 Deserialisierung

Aus dieser bestehenden Datei lässt sich die Instanz der Klasse auch wieder laden. Prinzipiell muss der Prozess dazu nur umgekehrt werden:

- `FileStream` und `BinaryFormatter` erzeugen
- Die Methode `Deserialize` aufrufen
- Methoden der Klasse aufrufen

Hier der Code für die Methode `Deserialize`, allerdings ohne jede Fehlerprüfung, ob die Datei KLASSE.DAT auch existiert (in Produktionscode würden Sie natürlich nie auf Fehlerprüfung verzichten, nicht wahr?).

```
static void Deserialize()
{
   // instanz der klasse erzeugen und ausgeben
   DieKlasse cls;

   Console.WriteLine("Lesen ...");

   // der formatierer fuer die deserialisierung
   BinaryFormatter bf = new BinaryFormatter();

   // die datei, aus der deserialisiert wird
   FileStream fs = new FileStream("klasse.dat",
                   FileMode.Open,FileAccess.Read);

   // klasse in den dateistrom serialisieren
   // und per konvertierung in typ DieKlasse wandeln
```

```
    cls = (DieKlasse)bf.Deserialize(fs);
    fs.Close();

    // die wieder gelesenen Werte ausgeben
    cls.Ausgabe();

    // fertig, bescheid geben
    Console.WriteLine("Fertig. [ENTER] drücken.");
    Console.ReadLine();
}
```

Listing 25.3: Die Methode für die Deserialisierung einer Klasse

Da die Methode Deserialize des Formatters nicht wissen kann, welcher Typ da aus der Datei gelesen wird, liefert die Methode einen Typ `Object`. Aus diesem Grund muss bei der Zuweisung an die Klassenvariable eine Typkonvertierung in den passenden Datentyp erfolgen.

Was passiert nun aber mit dem Feld `geheim`? Dessen Inhalt wurde ja nicht serialisiert. Daher vergibt die .NET-Laufzeitumgebung den Standardwert für den jeweiligen Datentyp, bei einem `double` den Wert `0`. Daher auch die Ausgabe von `0` auf der nachfolgenden Abbildung.

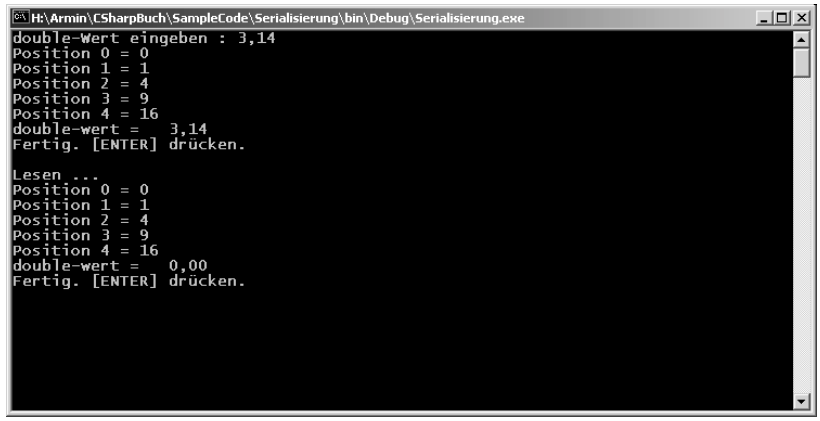


Abbildung 25.2: Einmal serialisieren und deserialisieren, bitte

Hier noch einmal der komplette Code für das Beispielprogramm zur attributgesteuerten Serialisierung und Deserialisierung.

```
// Goto C# -- Armin Hanisch
// Serialisierung und Deserialisierung
// von Klassen über Attribute

using System;
using System.IO;
using System.Collections;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace Serialisierung
{
   // diese klasse implementiert ISerializable
   [Serializable]
   public class DieKlasse : ISerializable
   {
      // die felder der klasse
      protected int[] zahlen;
      protected double geheim;

      // der standard-konstruktor
      public DieKlasse(int number)
      {
         zahlen = new int[number];
         for(int i = 0; i < number; i++)
         {
            zahlen[i] = i*i;
         }
         Console.Write("double-Wert eingeben : ");
         geheim = Convert.ToDouble(Console.ReadLine());
      }

      // ausgabe der felder
      public void Ausgabe()
      {
         for(int i = 0; i < zahlen.Length; i++)
         {
            Console.WriteLine("Position {0} = {1}",
                              i, zahlen[i]);
         }

         Console.WriteLine("double-wert = {0,6:0.00}",
                           geheim);
      }
```

```
   // deserialisierungs-konstruktor
   public DieKlasse(SerializationInfo info,
                    StreamingContext context)
   {
      // das feld zahlen deserialisieren
      zahlen = (int[])info.GetValue(
               "DieKlasse-zahlen", typeof(int[]));
      }

   // serialisierung des objekts starten,
   // benötigte daten liefern
   public void GetObjectData(SerializationInfo info,
                             StreamingContext context)
   {
      // wir serialisieren nur "zahlen"
      info.AddValue("DieKlasse-zahlen", zahlen);

      /*
         hier könnte man noch den StreamingContext
         prüfen, beispielsweise für das retten von
         handles, die auch im zielprozess gelten

         if(context.State ==
            StreamingContextStates.CrossProcess)
         {
         }
      */
   }

}

//
// die testklasse zur serialisierung
//
class Testklasse
{
   static void Serialize()
   {
      // eine instanz der klasse erzeugen
      // und felder ausgeben
      DieKlasse cls = new DieKlasse(5);
      cls.Ausgabe();

      // wir brauchen einen formatierer
      BinaryFormatter bf = new BinaryFormatter();
```

```
   // die datei, in die serialisiert wird
   FileStream fs = new FileStream("klasse.dat",
      FileMode.OpenOrCreate,FileAccess.Write);

   // klasse in den dateistrom serialisieren
   bf.Serialize(fs, cls);
   fs.Flush();
   fs.Close();

   // fertig, bescheid geben
   Console.WriteLine("Fertig. [ENTER] drücken.");
   Console.ReadLine();
}

static void Deserialize()
{
   // eine instanz der klasse erzeugen
   // und felder ausgeben
   DieKlasse cls;

   Console.WriteLine("Lesen ...");

   // der formatierer fuer die deserialisierung
   BinaryFormatter bf = new BinaryFormatter();

   // die datei, aus der deserialisiert wird
   FileStream fs = new FileStream("klasse.dat",
                   FileMode.Open,FileAccess.Read);

   // klasse in den dateistrom serialisieren
   cls = (DieKlasse)bf.Deserialize(fs);
   fs.Close();

   // die wieder gelesenen Werte ausgeben
   cls.Ausgabe();

   // fertig, bescheid geben
   Console.WriteLine("Fertig. [ENTER] drücken.");
   Console.ReadLine();
}

static void Main(string[] args)
{
   Serialize();
   Deserialize();
```

```
            Console.WriteLine("Fertig. [ENTER] drücken.");
            Console.ReadLine();
        }

    }
}
```

Listing 25.4: Das Beispielprogramm für die Serialisierung über Attribute

25.4 ISerializable implementieren

Wollen Sie den Prozess der Serialisierung genauer steuern, um beispielsweise zur Laufzeit darüber zu entscheiden, ob ein Feld der Klasse serialisiert werden soll oder nicht, dann implementieren Sie die Schnittstelle `ISerializable` in Ihrer Klasse.

> Mag sein, dass ich die .NET-Dokumentation falsch gelesen habe, aber dort wird erwähnt, dass eine Klasse für die Serialisierung entweder das Attribut `[Serializable]` **oder** die Schnittstelle `ISerializable` benutzen muss. Tatsächlich muss zumindest in der Beta2 auch für Klassen, welche die Schnittstelle `ISerializable` benutzen, das Attribut `[Serializable]` benutzt werden.

Die Schnittstelle `ISerializable` erfordert zwei Methoden, einmal `GetObjectData` mit der Signatur (`SerializationInfo`, `StreamingContext`) und zum zweiten einen Konstruktor für die Deserialisierung mit der gleichen Signatur. Über die Methode `GetObjectData`, die der Formatter zu Beginn der Serialisierung aufruft, können Sie über den Aufruf der Methode `AddValue` der `SerializationInfo`-Instanz bestimmen, welche Felder wie serialisiert werden. Zusammen mit der Auswertung des `StreamingContext` besitzen Sie eine sehr flexible Möglichkeit, genau zu bestimmen, was alles aus der Klasse in die Serialisierung wandert.

Hier der Code für die Beispielklasse, vorbereitet für die Unterstützung der Schnittstelle `ISerializable`.

```
[Serializable]
public class DieKlasse2 : ISerializable
{
   private int[] zahlen;
   private double geheim;
```

```
public DieKlasse2(int number)
{
   zahlen = new int[number];
   for(int i = 0; i < number; i++)
   {
      zahlen[i] = i*i;
   }
   Console.Write("double-Wert eingeben : ");
   geheim = Convert.ToDouble(Console.ReadLine());
}

public void Ausgabe()
{
   for(int i = 0; i < zahlen.Length; i++)
   {
      Console.WriteLine("Position {0} = {1}",
                        i, zahlen[i]);
   }

   Console.WriteLine("double-wert = {0,6:0.00}",
                     geheim);
}

// Deserialization constructor
public DieKlasse2(SerializationInfo info,
                  StreamingContext context)
{
   //deserialize value
   zahlen = (int[])info.GetValue("DieKlasse-zahlen",
                                 typeof(int[]));
}


// anforderung der zu serialisierenden daten
public void GetObjectData(SerializationInfo info,
                          StreamingContext context)
{
   // serialisierung von zahlen
   info.AddValue("DieKlasse-zahlen", zahlen);

   // beispiel für einbeziehung des
   // StreamingContext
   if(context.State ==
      StreamingContextStates.CrossProcess)
   {
```

```
            // hier code, falls nur CrossProcess
        }
    }

}
```

Listing 25.5: Serialisierung über ISerializable

Die Deserialisierung funktioniert exakt so, wie bei der attributgesteuerten Deserialisierung, wobei auch hier wieder eine entsprechende Typkonvertierung vorgenommen werden muss. Im Code des Beispiels finden Sie diese Typkonvertierung auch beim Deserialisierungskonstruktor, da die Methode `GetValue` ebenfalls nur den Typ `Object` liefert.

Damit sind die wichtigsten Dinge zum Thema Serialisierung in einen persistenten Speicher mit Hilfe von Binärdateien erwähnt. Analog dazu verläuft die Serialisierung über einen SOAP-Formatter, der allerdings dazu ein XML-Format und keine Binärdatei verwendet. Auf das Thema Serialisierung und Remoting werde ich hier nicht eingehen, Beispiele dazu finden Sie in der Online-Dokumentation der .NET-Laufzeitumgebung. Stattdessen möchte ich an dieser Stelle das Thema XML noch einmal aufgreifen und die Serialisierung von und nach XML-Dateien über einen `XmlSerializer` behandeln, da Sie hier eine im Gegensatz zu den Binärdateien mit anderen Anwendungen leicht bearbeitbare Serialisierung Ihrer Klassen erhalten.

25.5 XML-Serialisierung

Die XML-Serialisierungsfähigkeiten der .NET-Laufzeitumgebung erlauben Ihnen die Nutzung der Flexibilität von XML und gleichzeitig die Beibehaltung der umfangreichen Typinformationen von C# in einem leicht zu lesenden Dateiformat. Der XmlSerializer kümmert sich bei komplexen Feldern oder Eigenschaften (aggregierten Objekten, Arrays, etc.) automatisch um eine Darstellung als Unterelement und die entsprechende Auswahl der Elemente. So liefert eine XML-Serialisierung der folgenden Klasse `MyClass`

```
public class MyObject
{
   public string ObjectName;
}
```

```
public class MyClass
{
   public MyObject MyObjectProperty;
}
```

die folgende XML-Darstellung:

```
<MyClass>
   <MyObjectProperty>
      <Objectname>My String</ObjectName>
   </MyObjectProperty>
</MyClass>
```

Falls Sie in der XML-Struktur nur den Status des Objektes ohne besondere Anforderung festhalten wollen, müssen Sie die Klasse in keinster Weise verändern, keine Attribute hinzufügen oder anderes. Wollen Sie allerdings die XML-Ausgabe steuern oder muss die Klasse ein bestimmtes Schema erfüllen, haben Sie natürlich die entsprechenden Einflussmöglichkeiten.

25.5.1 Die Beispielklasse

Bevor ich den Code für die Serialisierung bespreche, benötige ich, erraten, wieder eine Beispielklasse. Damit die Sache nicht langweilig wird, hier eine neue Klasse mit ein paar Feldern mehr.

```
// beispielklasse für die xml-serialisierung

public class Kunde
{
   public string Name;
   public int Punkte;
   public DateTime Datum;
   public int[] Zahlen;

   public Kunde()
   {
      Zahlen = new int[6];
      Zahlen[0] = 12;
      Zahlen[1] = 22;
      Zahlen[2] = 32;
      Zahlen[3] = 42;
      Zahlen[4] = 52;
```

```
        Zahlen[5] = 62;
    }
}
```

Listing 25.6: Die Beispielklasse für die XML-Serialisierung

25.5.2 Serialisierung in eine XML-Datei

Für die Serialisierung in eine XML-Datei wird eine Instanz eines `XmlSerializer`s benötigt. Hier die benötigten Namensräume:

```
using System;                    // basis-sachen
using System.IO;                 // für den TextWriter
using System.Xml;                // für das XML
using System.Xml.Serialization;  // für XmlSerializer
```

Listing 25.7: Die benötigten Namensräume für die XML-Serialisierung

Die Serialisierung in eine XML-Datei ist mit der .NET-Laufzeitumgebung sehr einfach und mühelos durchzuführen. Der Code besteht nur aus einer Handvoll Zeilen, wie Sie gleich sehen werden.

Der `XmlSerializer` benötigt im Konstruktor Informationen über den zu serialisierenden Datentyp

```
XmlSerializer ser = new XmlSerializer(typeof(Kunde));
```

Anschließend benötigen wir noch eine Ausgabemöglichkeit, hier eine Instanz eines StreamWriters, die in die Datei KUNDE.XML schreibt.

```
StreamWriter writer = new StreamWriter("kunde.xml");
```

Alles, was nun noch bleibt, ist der Aufruf der Methode `Serialize` der `XmlSerializer`-Instanz, die als Parameter den Ausgabestrom und die Instanz der zu serialisierenden Klasse übergeben bekommt. Danach wird der Strom geschlossen.

```
ser.Serialize(writer, kd);
writer.Close();
```

Das war's! Ehrlich, mehr brauchen Sie für eine XML-Serialisierung mit .NET nicht zu programmieren. Sehen wir uns das Ergebnis (die XML-Datei) an:

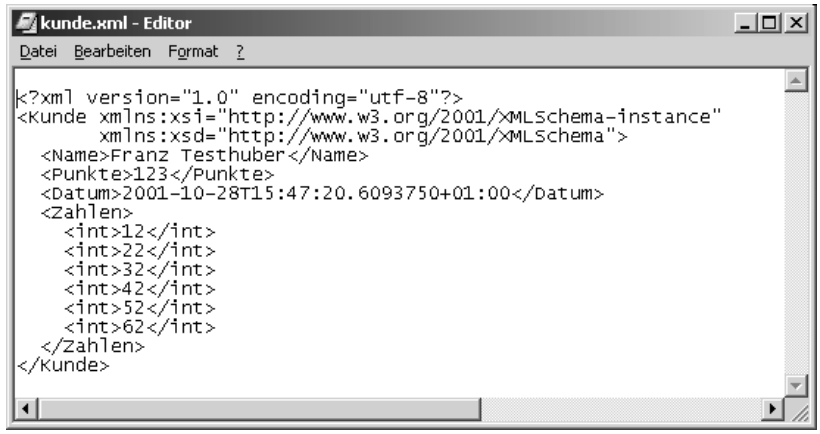


Abbildung 25.3: Die neu erstellte Datei mit den Kundendaten

25.5.3 Deserialisierung aus einer XML-Datei

Der umgekehrte Weg ist wie bei der binären Serialisierung ebenfalls wieder sehr einfach. Hier gleich der komplette Code für eine Methode `KundeFromXML` mit anschließender Ausgabe der Felder:

```
static void KundeFromXML(string datei)
{
   Kunde kd;

   XmlSerializer ser = new XmlSerializer(typeof(Kunde));
   TextReader reader = new StreamReader(datei);
   kd = (Kunde)ser.Deserialize(reader);
   reader.Close();

   Console.WriteLine("Name = " + kd.Name);
   Console.WriteLine("Punkte = " + kd.Punkte);
   Console.WriteLine("Datum = " + kd.Datum);
   foreach(int i in kd.Zahlen)
      Console.WriteLine("  Zahl = {0}", i);
}
```

Listing 25.8: Deserialisierung aus einer XML-Datei

25.5.4 Steuerung der XML-Serialisierung

Den Serialisierungsprozess, genauer die Erstellung der XML-Datei können Sie über den Einsatz verschiedener Attribute steuern. So möchten Sie vielleicht einige Felder der Klasse nicht als eigene Unterelemente, sondern als Attribute des Klassenelementes rendern lassen oder für einige Elemente den Datentyp ändern. Um beispielsweise das Wurzelelement der XML-Datei in `KundenListe` zu ändern, benutzen Sie vor der Klasse das Attribut `[XmlRoot("KundenListe")]`. Zusätzlich ändere ich in der Klassendefinition noch Folgendes ab:

- Das Datum soll als Attribut und nicht als Element gerendert werden, dazu wird das Feld in der Klasse mit dem Attribut `[XmlAttribute]` gekennzeichnet.
- Das Feld `punkte` soll als Element, aber mit dem Namen `umsatz` gerendert werden, dazu benutze ich das Attribut `[XmlElement("umsatz")]`.

Die geänderte Definition der Klasse `kunde` sieht nun so aus:

```
[XmlRoot("KundenListe")]
public class Kunde
{
   public string Name;
   [XmlElement("umsatz")]
   public int Punkte;
   [XmlAttribute] public DateTime Datum;

   public int[] Zahlen;

   public Kunde()
   {
      Zahlen = new int[6];
      Zahlen[0] = 12;
      Zahlen[1] = 22;
      Zahlen[2] = 32;
      Zahlen[3] = 42;
      Zahlen[4] = 52;
      Zahlen[5] = 62;
   }
}
```

Listing 25.9: Die Klasse kunde mit Steuerungsattributen

Wird nun die Serialisierung durchgeführt, bekommt die entstehende XML-Datei das gewünschte Aussehen, wie die nächste Abbildung zeigt.

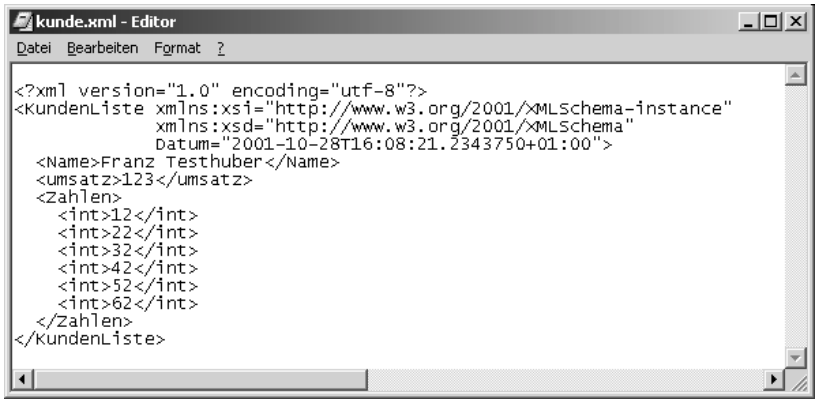


Abbildung 25.4: Die XML-Datei mit geändertem Rendering

Hier bietet die .NET-Laufzeitumgebung noch wesentlich weitergehende Möglichkeiten, die sich allerdings vom Thema des Buches zu sehr entfernen und in einer generellen .NET-Referenz besser aufgehoben sind. Daher möchte ich an dieser Stelle das Kapitel Serialisierung beenden und bin sicher, Sie haben bereits jetzt eine Menge Ideen für eigene Experimente.

25.6 Zusammenfassung

Serialisierung ist in C# auf einfache Art und Weise über die Angabe von Attributen möglich, Sie können aber auch jederzeit in diesen Prozess eingreifen und die Serialisierung der Objektinstanzen genauer steuern.

Eine Serialisierung ist nicht nur zum Speichern von Objektinstanzen über ein Beenden der Anwendung notwendig, sondern auch jedes Mal beim Überschreiten einer Sicherheitsgrenze (Application Domain, Prozess oder Maschine), wenn ein Remoting auftritt.

Mit einer Implementierung der Schnittstelle `ISerializable` sind Sie in der Lage, zur Laufzeit zu bestimmen, welche Felder serialisiert werden sollen und welche nicht.

Neben der Serialisierung in eine binäre Datei kann auch eine Serialisierung in eine XML-Datei vorgenommen werden, um so ein standardisiertes Format für die Übertragung von Objektdaten zur Verfügung zu haben. Die hierfür definierten Attribute der .NET-Laufzeitumgebung bieten eine ganze Reihe von Möglichkeiten, Formatierung und Inhalt der erzeugten XML-Datei zu steuern.

26 Assemblies

Sie finden dieses Kapitel ziemlich weit hinten im Buch, denn eigentlich ist es für das Verständnis der Programmiersprache C# nur bedingt notwendig. Wie ich in der Einführung aber bereits erläutert habe, gibt es Bereiche der .NET-Laufzeitumgebung, die essentiell für das Verständnis des Gesamtbildes sind und daher in diesem Buch zumindest überblicksartig behandelt werden sollten. Ich werde daher versuchen, das Thema Assemblies aus der Sicht des C#-Einsteigers zu schreiben, der sich nicht in die letzten Details vertiefen möchte, sondern einen fundierten Überblick benötigt, sozusagen den elektronischen Freischwimmer.

26.1 Die Probleme der Vergangenheit

Dieser Abschnitt lässt sich mit einem einzigen Ausdruck beschreiben: DLL Hell. Neue Versionen von COM-DLLs, die theoretisch abwärtskompatibel sein sollten, überschreiben die alte Version. Unglücklicherweise wusste der Entwickler, der mit der alten Version arbeitete von einigen Fehlern und programmierte einen Workaround. Jetzt fehlt der Fehler, was zu der paradoxen Situation führt, dass eine eigentlich fehlerbereinigte neue DLL erst Fehler verursacht. Oder ein Installationsprogramm prüft die Versionsnummer nicht korrekt und ruiniert ein normal laufendes System bis zur Unbenutzbarkeit. Fünf Applikationen installieren jeweils ihre eigene Version der Common Controls und und und …

Dazu kommt noch, dass es keine Möglichkeiten für ein durchgängiges Sicherheitskonzept in Windows gab. Code lief immer mit der Berechtigung des Benutzers, unabhängig vom Ursprungsort. Dass die Möglichkeit von Side-by-Side DLLs in Windows 2000 auch keine Lösung ist, zeigt folgendes einfaches Szenario: Sie benutzen eine In-Prozess COM-Komponente, die wiederum die Version 2 einer DLL benötigt. Gleichzeitig laden Sie selbst diese DLL, benötigen aber Version 3. Welche DLLs sollen nun in den Prozessraum Ihrer Anwendung geladen werden?

Über Themen wie Lokalisierung, Konfiguration durch externe Konfigurationsdateien ohne Registry, Installation in großen Firmennetzen reden wir lieber gar nicht …

26.2 Was sind Assemblies?

Ein Assembly ist die grundlegende Einheit für die Installation einer .NET-Anwendung. Eine Anwendung besteht aus mindestens einem, meist mehreren Assemblies, die in einem gemeinsamen Verzeichnis liegen. Ein Assembly ist eine selbstbeschreibende Einheit, die komplett ohne irgendeinen Eintrag in der Registry oder anderen Systemdateien auskommt – Sie können also schon mal anfangen, `rgsvr32.exe` zu vergessen.

Microsoft verwendet die deutsche Bezeichnung 'Baugruppe' für ein Assembly. Wenn die Übersetzung auch von der Wortbedeutung her korrekt sein mag, wird sie das gleiche Schicksal ereilen wie die Ausnahmebedingung, die jeder Entwickler einfach weiter als Exception bezeichnet. Ich hatte im ersten Entwurf des Buches so viele deutsche Begriffe wie möglich verwendet (schließlich ist dies ein deutsches Buch), aber nachdem ich die ersten Entwürfe von Kollegen mit entsprechenden Kommentaren wieder zurückbekam, habe ich mich den Sprachgewohnheiten der Branche angepasst. Lassen Sie mich wissen, was Sie zu diesem Thema denken.

Hier eine kleine Auflistung der wichtigsten Eigenschaften eines Assemblies. Im Verlauf dieses Kapitels werden Sie zu diesen Punkten genauere Informationen erhalten.

- Assemblies enthalten Versions- und Lokalisierungsinformationen, die eine Ausführung mehrerer Versionen des gleichen Assemblies ermöglichen. Diese Versionsinformationen können über Policies so genutzt werden, dass entweder automatisch oder nach Regeln definiert wird, welche Versionen noch miteinander kompatibel sind.
- Jedes Assembly ist eine selbstbeschreibende binäre Einheit, da für jeden im Assembly definierten Typ und alle seine Felder entsprechende Metadaten hinterlegt werden. Zur Nutzung sind keine weiteren Informationen wie Typbibliotheken oder die Registrierung nötig.

- Jedes Assembly enthält eine Metadaten-Beschreibung des Assemblies, der in ihm enthaltenen Module sowie der externen Referenzen. Diese Metadaten werden als 'Manifest' bezeichnet. Dort sind neben Name und Versionsnummer auch Informationen enthalten, welche Berechtigungen der Code benötigt sowie Hashcodes und kryptografische Informationen, um sicherzustellen, dass am Assembly keine Veränderungen vorgenommen wurden.
- Das Assemblyformat ähnelt zwar intern dem PE-Format einer ausführbaren Windows-Datei, allerdings ist der Code kein auf einen bestimmten Prozessor ausgerichteter Maschinencode, sondern die Opcodes der Intermediate Language (IL), einer leicht in CPU-spezifische Befehle umsetzbaren Version einer plattformunabhängigen Zwischensprache (mehr dazu gleich)
- Assemblies sind plattformunabhängig. Durch die Ablage des Codes in IL und die durchgehende Nutzung des Unicode-Standards für Ressourcen ist ein Assembly nicht notwendigerweise auf eine Windows-Umgebung angewiesen. Die sich daraus ergebenden Möglichkeiten und Konsequenzen werden seit Monaten im Internet heiß diskutiert und das bereits am Horizont auftauchende Mono-Projekt (*http://mononet.sourceforge.net*) lässt einen Blick in die Zukunft von .NET zu.
- Mehrere Assemblies lassen sich in eine Application Domain laden. Da alle Speicheranforderungen bei .NET unter der Kontrolle der .NET-Laufzeitumgebung geschehen, muss der relativ 'teure' Schutz eines eigenen Prozessraums mit allen nötigen Kontextumschaltungen nicht sein und die .NET-Laufzeitumgebung kann einzelne Anwendungen über sog. Application Domains trennen.
- Die Installation von Anwendungen und Assemblies ist mit minimalem Aufwand möglich. Die Assemblies werden auf dem Zielsystem in einen gemeinsamen Ordner kopiert und fertig. In der Realität wird dennoch meist der Windows Installer benutzt; dies allerdings nicht, weil Microsoft nicht die Wahrheit erzählt, sondern weil es auch bei .NET-Anwendungen meist nicht mit dem Transport der ausführbaren Dateien getan ist. Es müssen Verknüpfungen angelegt und Hinweise angezeigt, Ordner angelegt und natürlich auch Werbung für das Produkt gemacht werden. Bevor man all dies selbst neu erfindet, ist es natürlich einfacher, ein Installationsprogramm zu benutzen.

Hier der grundlegende, schematische Aufbau von Prozessraum und Assembly:

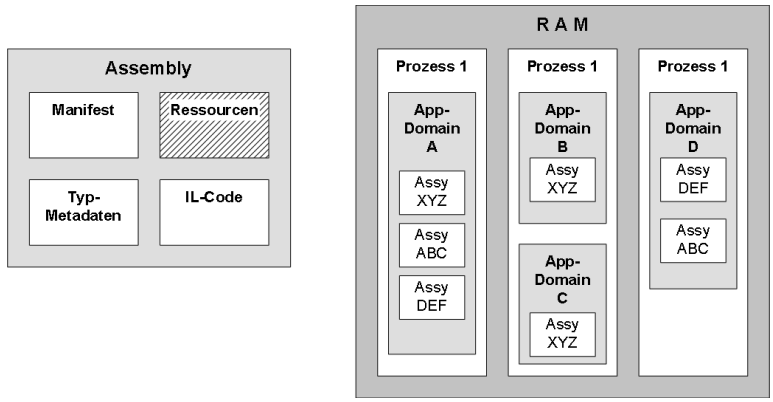


Abbildung 26.1: Aufbau von Assemblies und Nutzung des Arbeitsspeichers

Wie in der Liste oben beschrieben, ist ein Assembly die kleinste Installationseinheit einer Anwendung. In der Regel wird ein Assembly als EXE-Datei erstellt und andere Assemblies der Anwendung als Bibliotheken mit der Erweiterung DLL (wobei diese Assembly-Bibliotheken außer der Erweiterung nicht mehr viel mit den alten DLLs gemeinsam haben).

Wie Sie der rechten Abbildung entnehmen können, teilen sich unter Umständen mehrere Anwendungen einen Prozessraum und werden durch so genannte Anwendungsräume (application domains) voneinander abgeschottet. Hier optimiert die .NET-Laufzeitumgebung den Speicherverbrauch dadurch, dass Code pro Prozessraum nur einmal geladen wird, während die statischen und Instanzdaten pro Anwendungsraum geladen werden.

26.2.1 Private und öffentliche Assemblies

Die gute Nachricht zuerst: Um vieles, was im letzten Abschnitt aufgeführt wurde, müssen Sie sich gar nicht kümmern, wenn Sie nur Code für die eigene Anwendung schreiben. Die .NET-Laufzeitumgebung unterscheidet zwischen privaten und öffentlichen Assemblies (*private / shared*).

Wenn Sie nichts anderes festlegen, werden alle Assemblies, die der Compiler erstellt, als private Assemblies behandelt. Hier müssen Sie keine Rücksicht auf andere Anwendungen nehmen, keine Versionsrichtlinien und anderes prüfen. Die einzige Anwendung, die Probleme bekommen könnte, ist Ihre eigene. Keine andere Anwendung wird je die Assemblies Ihrer Anwendung überschreiben. Für zusätzliche Assemblies, Ressourcen und Lokalisierungen werden Unterverzeichnisse des Anwendungsverzeichnisses benutzt, also können auch hier keine Konflikte mit anderen Anwendungen entstehen.

Falls Sie dagegen eine Bibliothek zur Nutzung durch andere Anwendungen erstellen oder um Platz zu sparen ein Assembly systemweit verfügbar machen möchten, dann erstellen Sie ein öffentliches Assembly. Dies geschieht nicht über einen bestimmten Compilerschalter, sondern mit von Microsoft zur Verfügung gestellten Tools, da der Code unabhängig von seinem Einsatzgebiet ist. Die Infrastruktur und das Versionsverhalten des Codes müssen nun aber besser geplant werden, da dieses Assembly eben systemweit verfügbar ist. Eine der Anforderungen ist ein kryptografisch starker Name und eine systemweit eindeutige Bezeichnung (dafür existiert das Tool `sn.exe`). Alle öffentlichen Assemblies liegen in einem speziellen Bereich des Betriebssystems, dem globalen Assembly-Zwischenspeicher (global assembly cache), unter Windows der Ordner *Assembly* unterhalb des Windows-Ordners. Hier können Sie mit dem Werkzeug `gacutil.exe` sowohl den Inhalt auflisten als auch Assemblies eintragen oder wieder entfernen. Mehr dazu später.

Kryptografisch starker Name

Öffentliche Assemblies werden also durch einen 'starken Namen' geschützt. Neben der systemweit einheitlichen Kennung besteht dieser aus einem kryptografischen Schlüsselpaar aus einem privaten und einem öffentlichen Schlüssel. Die Online-Dokumentation der .NET-Laufzeitumgebung enthält eine gute Einführung in das Konzept der Verfahren mit öffentlichen Schlüsseln (*public key cryptography*), hier die Grundlagen zu erläutern würde zu weit führen. Kurz gesagt, wird bei diesem Verfahren der öffentliche Schlüssel verteilt und der private nicht herausgegeben. Werden Daten mit dem privaten Schlüssel verschlüsselt, können diese mit dem öffentlichen Schlüssel wieder entschlüsselt werden. Damit lässt sich die Herkunft verifizieren. Daten, die mit dem öffentlichen Schlüssel verschlüsselt werden, können nur mit dem privaten Schlüssel wieder in Klartext umgewandelt werden.

26.2.2 Assemblies angucken

Dafür existiert ein eigenes Werkzeug, `ildasm.exe`, das Sie im Verzeichnis des Framework SDK finden. Weitere Details zu diesem Werkzeug sowie eine Erklärung der einzelnen Symbole finden Sie im Referenzteil dieses Buches. Den Code unten habe ich übersetzen lassen und dann mit dem Kommando `ildasm AssemblyDemo.exe` angezeigt. Das Ergebnis dieses Aufrufs zeigt die Abbildung nach dem Listing.

```
using System;

namespace GotoCSharp.AssemblyDemo
{
   class MainClass
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Assemblydemo Main()");
         Console.ReadLine();
      }
   }
}
```

Listing 26.1: Der (minimale) Beispielcode

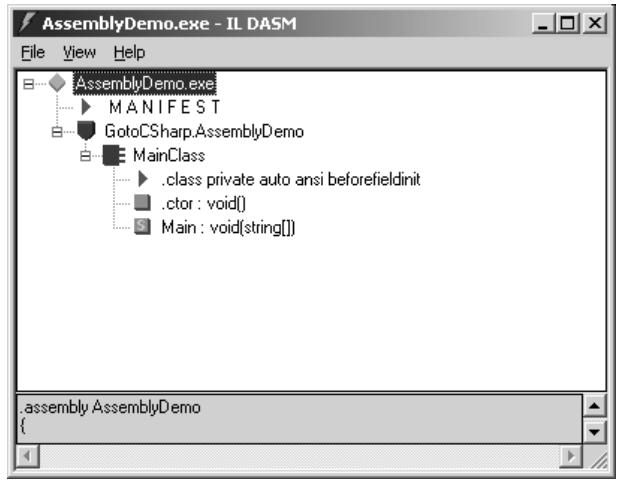


Abbildung 26.2: Das Assembly mit Manifest und IL-Code

Die Abbildung zeigt das Manifest und anschließend im Namensraum des Quellcodes die Klasse `MainClass` mit den Mitgliedern der Klasse, die in diesem Fall nur aus dem Default-Konstruktor und der statischen Methode `Main` bestehen.

Ein Doppelklick auf das Manifest zeigt dieses in einem Editorfenster an, auch den IL-Code für die Methode `Main` können Sie sich mit einem Doppelklick auf die Methode anzeigen lassen.

26.3 Öffentliche Assemblies erstellen

26.3.1 Der prinzipielle Ablauf

Nachdem ein privates Assembly erstellt (und getestet) wurde, wird für dieses Assembly ein Schlüsselpaar aus privatem und öffentlichem Schlüssel erstellt. Mit diesem kryptografischen Schlüssel wird das Assembly signiert und anschließend das signierte Assembly in den globalen Assembly-Cache eingetragen.

26.3.2 Beispiel-Bibliothek erstellen

Zur Erstellung eines öffentlichen Assemblies benötigen wir erst einmal den Code für die Bibliothek, in diesem Fall die gute alte Schaltjahresberechnung. Diese soll allerdings als Bibliothek und nicht als Anwendungsassembly erstellt werden. In Visual Studio wählen Sie dazu den Punkt KLASSENBIBLIOTHEK aus der Auswahl der möglichen C#-Projekte, auf Kommandozeilenebene benutzen Sie Compileroption `/target:library`, um eine Bibliothek zu erstellen.

Hier der Code für die Bibliothek (für eine Klasse eigentlich zu wenig, aber es geht hier ja darum, das Prinzip zu verdeutlichen und nicht um ein komplettes Produkt).

```
// Goto C# -- Armin Hanisch
// democode für ein shared assembly

using System;

namespace GotoCSharp.DemoLib
{
   public class Datum
```

```
   {
      private DateTime aDate;

      public Datum(DateTime dt)
      {
         aDate = dt;
      }

      public bool IstSchaltjahr()
      {
         return ( (aDate.Year % 4 == 0)
                  && (aDate.Year % 100 != 0) )
                || (aDate.Year % 400 == 0);
      }

   }
}
```

Listing 26.2: Der Quelltext für die DemoLib

Nach der Übersetzung erhalten Sie eine Datei `DemoLib.dll`, die nun nicht direkt aufgerufen werden kann, da keine Methode `Main` darin enthalten ist. Aus diesem privaten Assembly soll nun ein öffentliches Assembly werden

26.3.3 Schlüsselpaar erzeugen

Zur Erzeugung der kryptografischen Schlüssel dient das Tool `sn.exe` (strong name), das über eine ganze Reihe an Optionen und Funktionen verfügt. In unserem Fall soll ein Schlüsselpaar erzeugt werden, wofür die Option `-k` dient.

```
sn -k schluessel.snk
```

Dieses Kommando erzeugt das Schlüsselpaar in der Datei `schluessel.snk`. Die Erweiterung `.snk` ist eine Konvention von Microsoft und nicht zwingend vorgeschrieben.

26.3.4 Signieren des Assemblies

Dazu wird der Name der Schlüsseldatei dem Assembly über ein Attribut mitgeteilt. Wenn Sie mit Visual Studio.NET arbeiten, werden die entsprechenden Attribute vom Assistenten beim Anlegen eines neuen

Projektes automatisch in der Datei `AssemblyInfo.cs` angelegt, die folgenden Inhalt besitzt und auch Informationen über den Ablauf der Signierung sowie der Versionsverwaltung von Assemblies enthält. Arbeiten Sie ohne Visual Studio.NET, müssen Sie die Attribute selbst in den Quellcode integrieren.

```
using System.Reflection;
using System.Runtime.CompilerServices;

//
// Allgemeine Informationen über eine Assembly
// werden über folgende Attribute
// gesteuert. Ändern Sie diese Attributswerte,
// um die Informationen zu modifizieren,
// die mit einer Assembly verknüpft sind.
//
[assembly: AssemblyTitle("")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("")]
[assembly: AssemblyCopyright("")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

//
// Versionsinformationen für eine Assembly bestehen
// aus folgenden vier Werten:
//
//      Hauptversion
//      Nebenversion
//      Buildnummer
//      Revision
//
// Sie können alle Werte oder die standardmäßige
// Revision und Buildnummer
// mit '*' angeben:

[assembly: AssemblyVersion("1.0.*")]

//
// Um die Assembly zu signieren, müssen Sie einen
// Schlüssel angeben. Weitere Informationen
// über die Assemblysignierung finden Sie in der
```

```
// Microsoft .NET Framework-Dokumentation.
//
// Verwenden Sie folgende Attribute, um festzulegen
// welcher Schlüssel verwendet wird.
//
// Hinweis:
//   (*) Wenn kein Schlüssel angegeben ist, wird die
//       Assembly nicht signiert.
//   (*) KeyName verweist auf einen Schlüssel, der im
//       CSP (Crypto Service Provider) auf Ihrem
//       Computer installiert wurde. KeyFile verweist
//       auf eine Datei, die einen Schlüssel enthält.
//   (*) Wenn die Werte für KeyFile und KeyName
//       angegeben werden,
//       werden folgende Vorgänge ausgeführt:
//       (1) Wenn KeyName im CSP gefunden wird, wird
//           dieser Schlüssel verwendet.
//       (2) Wenn KeyName nicht vorhanden ist und
//           KeyFile vorhanden ist,
//           wird der Schlüssel in KeyFile im CSP
//           installiert und verwendet.
//   (*) Um eine KeyFile zu erstellen, können Sie das
//       Programm sn.exe (Strong Name) verwenden.
//       Wenn KeyFile angegeben wird, muss der Pfad von
//       KeyFile
//       relativ zum Projektausgabeverzeichnis sein:
//       %Project Directory%\obj\<configuration>.
//       Wenn sich KeyFile z.B. im Projektverzeichnis
//       befindet, geben Sie das AssemblyKeyFile-
//       Attribut  wie folgt an:
//     [assembly: AssemblyKeyFile("..\\..\\mykey.snk")]
//   (*) Das Verzögern der Signierung ist eine
//       erweiterte Option. Weitere Informationen
//       finden Sie in der
//       Microsoft .NET Framework-Dokumentation.
//
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile("")]
[assembly: AssemblyKeyName("")]
```

Listing 26.3: Die automatisch erzeugte Datei AssemblyInfo.cs

Wie in dieser Datei angegeben, wird jetzt bei dem Attribut `AssemblyKeyFile` der Name der Schlüsseldatei eingetragen.

```
[assembly: AssemblyKeyFile("..\\..\\schluessel.snk")]
```

Bei der nächsten Übersetzung wird das Assembly signiert und im Manifest die Informationen eingetragen. Hier eine Abbildung mit dem Inhalt des Manifests nach der Signierung (Doppelklick im ILDASM auf das Manifest). Unter dem Eintrag `.publickey` finden Sie nun den öffentlichen Schlüssel des Assemblies.

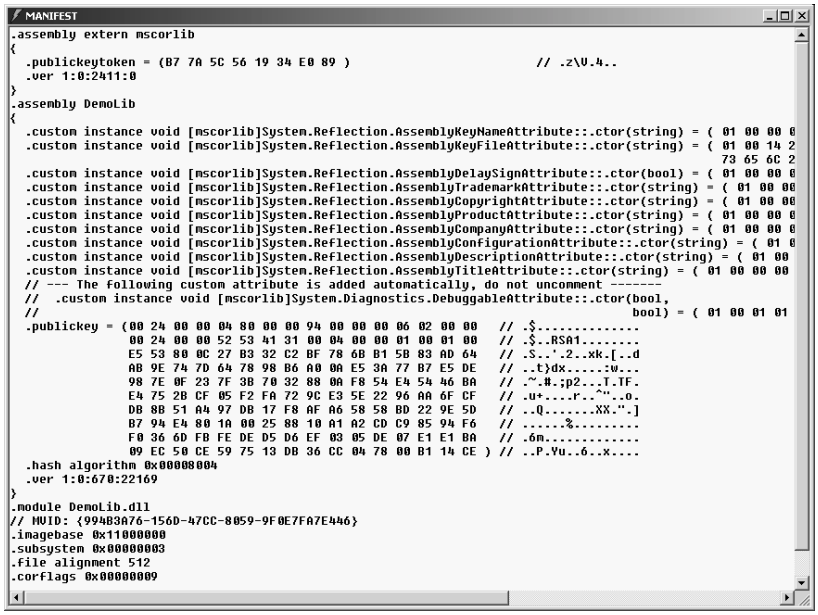


Abbildung 26.3: Der öffentliche Schlüssel ist im Manifest eingetragen

26.3.5 Der globale Assembly-Cache

Dieser Bereich speichert, wie der Name schon aussagt, globale (öffentliche) Assemblies. Für den weiteren Verlauf werde ich diesen Zwischenspeicher für die öffentlichen Assemblies einfach mit GAC (*global assembly cache*) abkürzen. Wenn Sie das Verzeichnis `%windowsdir%assembly` mit dem Explorer ansehen, erhalten Sie eine Übersicht über alle momentan installierten Assemblies. Die tatsächliche Struktur ab diesem Verzeichnis sieht anders aus und wird nicht angezeigt, da der Explorer die Inhalte des GAC über eine Shell-Erweiterung anzeigt.

Sie erkennen in der Auflistung neben dem Namen eine Spalte Typ (die angibt, ob zusätzlich zum Assembly eine vorübersetzte native-code Datei vorliegt), eine Angabe der Kultur (der regionalen Einstellungen) und die Signatur des öffentlichen Schlüssels. Wie wir später sehen werden, gehören alle diese Informationen zum Namen eines öffentlichen Assemblys, nicht nur der Name der Datei.

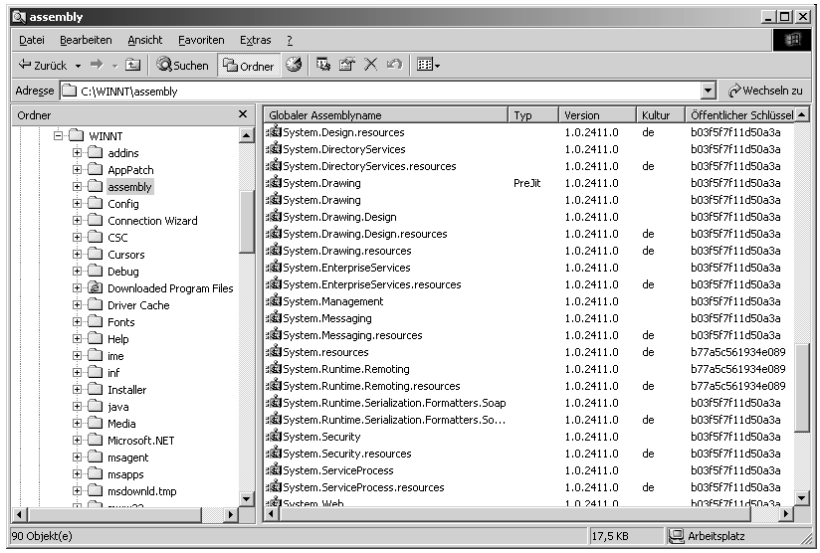


Abbildung 26.4: Der Explorer nutzt eine Shell-Erweiterung zur Anzeige

26.3.6 Assembly im GAC eintragen

In diesen GAC muss nun das Assembly eingetragen werden. Dies ist nicht mit einem einfachen Kopieren getan, da beispielsweise Ordner mit dem öffentlichen Schlüssel angelegt werden müssen. Microsoft stellt dafür das Programm `gacutil.exe` zur Verfügung, dessen Hilfsbildschirm hier die möglichen Optionen auflistet.

```
Microsoft (R) .NET Global Assembly Cache Utility.
Version 1.0.2914.16
Copyright (C) Microsoft Corp. 1998-2001.
All rights reserved.

Usage: Gacutil <option> [<parameters>]
```

```
Options:
 /i
   Installs an assembly to the global assembly cache.
   Include the name of the file containing the manifest
   as a parameter.
   Example:  /i myDll.dll

 /u[ngen]
   Uninstalls an assembly. Include the name of the
   assembly to remove as a parameter.  If ngen is
   specified, the assembly is removed from the cache
   of ngen'd files, otherwise the assembly
   is removed from the global assembly cache
   Examples:.
     /ungen myDll
     /u myDll,Version=1.1.0.0,Culture=en,
        PublicKeyToken=874e23ab874e23ab

 /l
   Lists the contents of the global assembly cache

 /cdl
   Deletes the contents of the download cache

 /ldl
   Lists the contents of the downloaded files cache

 /nologo
   Suppresses display of the logo banner

 /silent
   Suppresses display of all output

 /?
   Displays this help screen
```

Listing 26.4: Die möglichen Optionen von gacutil

Zum Eintragen unserer `DemoLib.dll` muss also die Option `/i` verwendet werden.

```
gacutil /i DemoLib.dll
Microsoft (R) .NET Global Assembly Cache Utility.
Version 1.0.2914.16
Copyright (C) Microsoft Corp. 1998-2001.
```

```
All rights reserved.

Assembly successfully added to the cache
```

Listing 26.5: Das Eintragen des Assemblies war erfolgreich

Fertig! Jetzt geht es daran, diese Bibliothek auszuprobieren und dafür erstellen wir ein neues Projekt.

26.4 Öffentliche Assemblies benutzen

Die Benutzer von Visual Studio sind hier vom Handling her im Vorteil. Sie fügen einfach einen neuen Projektverweis auf die Bibliothek hinzu. Benutzer der SDK-Version müssen beim Übersetzen die entsprechenden Verweise mit dem Schalter `/r` selbst angeben.



In der IDE setzen Sie eine neue Referenz, indem Sie bei den Verweisen des Projekts einen neuen Verweis hinzufügen. Die gesuchte Bibliothek haben wir zwar im GAC eingetragen, aber die Codebasis liegt im Ausgabeverzeichnis des Projektes, daher müssen Sie über die Schaltfläche DURCHSUCHEN den Pfad zu der DLL-Datei suchen.

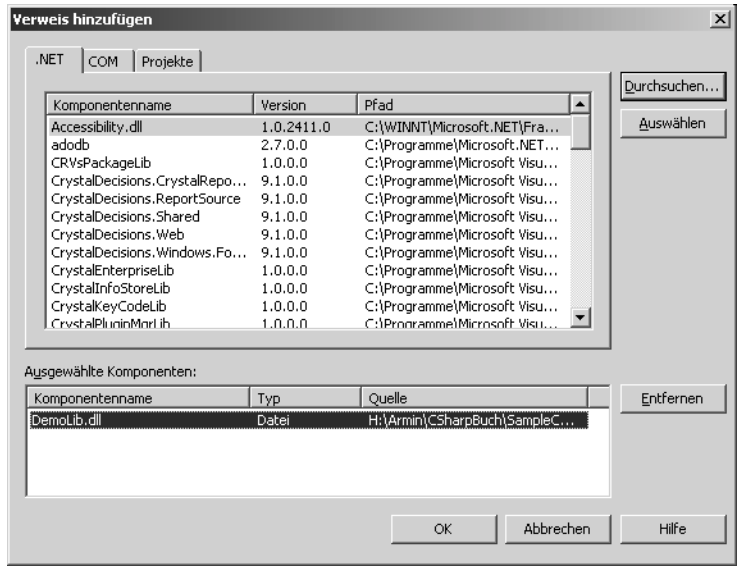


Abbildung 26.5: Verweise mit Visual Studio.NET hinzufügen

Hier ist der Programmcode für den Client, der ebenfalls wieder so einfach wie möglich gehalten ist. Diesmal nenne ich den Namensraum `GotoCSharp.DemoClient`, eine Klasse im gleichen Namensraum wäre genauso möglich gewesen.

```
// Goto C# -- Armin Hanisch
// client für die demolib

using System;
using GotoCSharp.DemoLib;

namespace GotoCSharp.DemoClient
{
   public class Testklasse
   {
      public static void Main()
      {
         string txt;
         Datum obj = new Datum(System.DateTime.Now);

         if(obj.IstSchaltjahr())
            txt = "ein";
         else
            txt = "kein";
         Console.WriteLine("Dieses Jahr ist " +
                           "{0} Schaltjahr",
                           txt);

      }
   }
}
```

Listing 26.6: Der Client für das DemoLib-Assembly

Betrachten wir mit ILDASM das Manifest für dieses Assembly, finden wir dort auch den Eintrag `.assembly extern` mit der Referenz auf das öffentliche Assembly. Dort steht auch das öffentliche Schlüsseltoken und die benötigte Versionsnummer.

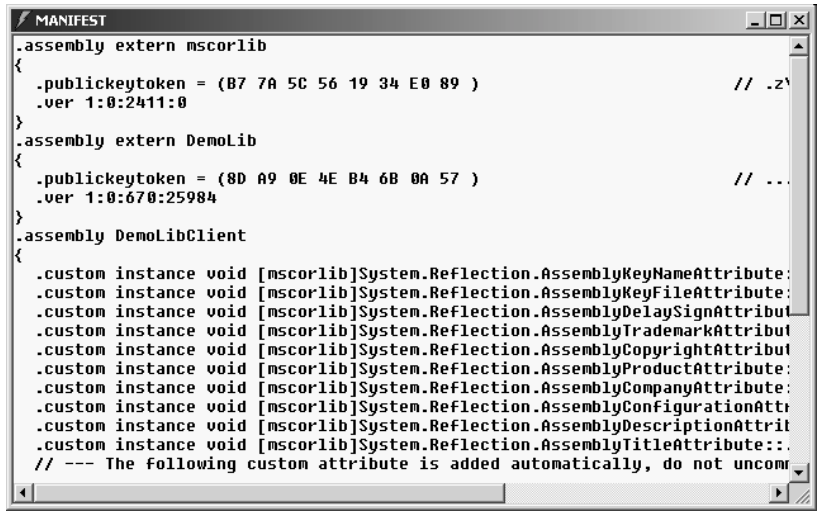


Abbildung 26.6: Das Manifest für den DemoLib-Client

26.5 Versionsverwaltung bei Assemblies

Falls Sie nur private Assemblies schreiben, ist das Thema Versionsverwaltung eher von untergeordneter Bedeutung. Für öffentliche Assemblies ist dies dagegen extrem wichtig, da beispielsweise zwei Referenzen aus einer Anwendung auf zwei verschiedene Versionen des öffentlichen Assemblys verweisen können.

26.5.1 Aufbau der Versionsnummern

Versionsinformationen für ein Assembly bestehen aus folgenden vier Werten:

- Hauptversion
- Unterversion
- Buildnummer
- Revision

Diese Daten werden ebenfalls über ein Attribut für das Assembly vergeben. Bei den Visual Studio-Projekten finden Sie diese zusammen mit den anderen Attributen in der Datei `AssemblyInfo.cs`. Arbeiten Sie mit

einer anderen Umgebung, können Sie diese Informationen genauso gut in die Datei mit aufnehmen.

```
[assembly: AssemblyVersion("1.0.*")]
```

Die Werte für Build und Revision können sie automatisch vergeben lassen, wenn Sie nach den beiden ersten Elementen einen Stern angeben. Die .NET-Laufzeitumgebung vergibt dann für den Build einfach die Anzahl der Tage seit dem 1. Januar 2000 (*Hurra! Wieder eine neue Zeitbasis …*) und für die Revision die Anzahl der Sekunden seit Mitternacht. Aus diesem Grund werden Sie in der Regel die Werte selbst vergeben.

Mit welcher Version ein Assembly noch arbeiten kann, wenn die geforderte Version nicht verfügbar ist, lässt sich über Versionsrichtlinien bestimmen. Das Standardverhalten des Systems sieht so aus: Alles, was sich nur durch die Revision unterscheidet, gilt als problemlos kompatibel. Ändert sich die Buildnummer, wird eine eingeschränkte Kompatibilität angenommen und ab einer Änderung der Unterversion sind die beiden Assemblies nicht mehr kompatibel. Falls Sie dieses Verhalten akzeptabel finden, müssen Sie nichts ändern, andernfalls tragen Sie eine neue Richtlinie in der Konfigurationsdatei der Anwendung ein.

26.6 Informationen im Programm abrufen

Im Namensraum `System.Reflection` finden Sie (unter anderem) die beiden Klassen `Assembly` und `AppDomain`, mit deren Methoden und Eigenschaften Sie Informationen auch zur Laufzeit Ihrer Anwendung erfragen können. Wer je versucht hat, mit dem Win32-API Versionsinformationen aus einer DLL auszulesen, wird allein für die Eigenschaft `Assembly.FullName` ein Dankschreiben nach Redmond senden. Hier ein kleines Beispielprogramm, das als Ausgangspunkt für eigene Erweiterungen und Experimente dienen kann.

```
// Goto C# -- Armin Hanisch
// informationen über assembly und appdomain

using System;
using System.Reflection;

namespace GotoCSharp.AssemblyDemo
```

```
{
   class MainClass
   {
      static void Main(string[] args)
      {
         // das laufende asembly holen
         Assembly asm = Assembly.GetExecutingAssembly();

         // die aktuelle application domain lesen
         AppDomain appdom = AppDomain.CurrentDomain;

         Console.WriteLine("Assemblyname = {0}",
                           asm.FullName);
         Console.WriteLine("Codebasis = {0}",
                           asm.CodeBase);

         Console.WriteLine("Basis akt. AppDomain = {0}",
                           appdom.BaseDirectory);
         Console.WriteLine("Name akt. AppDomain = {0}",
                           appdom.FriendlyName);

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

Listing 26.7: Wo bin ich und woher komme ich in der C#-Version

26.7 Konfigurationsdateien

Eines der Versprechen von .NET ist die Möglichkeit zu einem sog. *xcopy deployment*, also das einfache Kopieren aller Assemblies in ein Verzeichnis auf der Zielmaschine und der sofortigen Startfähigkeit der Anwendung. Im letzten Abschnitt habe ich aber des Öfteren die Möglichkeit von Richtlinien und Konfigurationseinstellungen erwähnt, die Sie für eine Anwendung treffen können. Aus dem oben genannten Grund kann dies natürlich nicht in der Registry gespeichert werden, sondern muss ebenfalls als kopierbare Datei vorliegen. Microsoft hat hier ebenfalls eine elegante Lösung gefunden und legt Konfigurationseinstellungen für die Anwendung in einer XML-Datei ab, die einfach den Namen des Anwendungs-EXEs mit einem angehängten `.config` trägt, also z.B. `DemoLibClient.exe.config`.

Diese Konfigurationsdateien erfüllen noch eine ganze Reihe von anderen Aufgaben. So sind hier Einträge über beim Laden von Assemblies oder Modulen zu durchsuchende Verzeichnisse vorhanden, Sicherheitseinstellungen können hier getroffen werden und vieles mehr.

Mögliche Richtlinien für die Auflösung von Versionskonflikten können hier auch getroffen werden. Dies möchte ich am Beispiel unseres Clients zeigen. Angenommen, nach vielen Verbesserungen ist die DemoLib mittlerweile bei der Versionsnummer 2.0.1.42 angelangt, würde aber immer noch mit dem Client funktionieren. Aufgrund der Standardrichtlinie für die Versionsverwaltung wird diese Bibliothek aber nicht mehr als kompatibel erkannt und beim Start des Clients erhalten Sie eine entsprechende Fehlermeldung, dass die Datei mit dem öffentlichen Assembly nicht gefunden werden konnte:

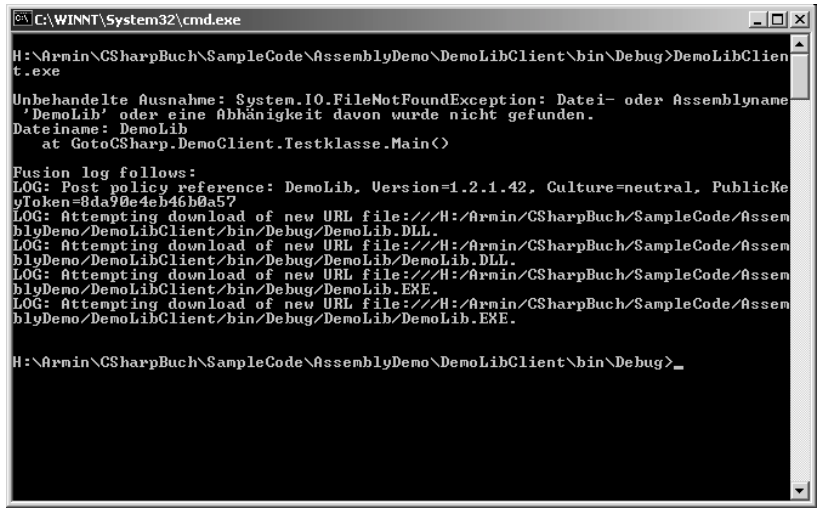


Abbildung 26.7: Eine versionskompatible Datei konnte nicht gefunden werden

Dafür lässt sich nun mit einer Konfigurationsdatei eine Art 'Umleitung' für die benutzten Versionen einstellen. Dies kann entweder manuell geschehen oder – jawohl es geht auch mit 'Fensterln' – über die Microsoft Management Konsole.

```
mmc <pfad>\mscorcfg.msc
```

Die Konfigurationsdatei für die MMC finden Sie in der aktuellen Version der .NET-Laufzeitumgebung, ersetzen Sie in der obigen Zeile `<pfad>` durch diesen Dateipfad oder kopieren Sie sich die Datei an eine besser geeignete Stelle.

Was Sie dann vor sich sehen, ist das .NET Admin Tool, mit dem Sie sowohl den GAC als auch einzelne Anwendungen und andere Einstellungen der .NET-Laufzeitumgebung konfigurieren können:

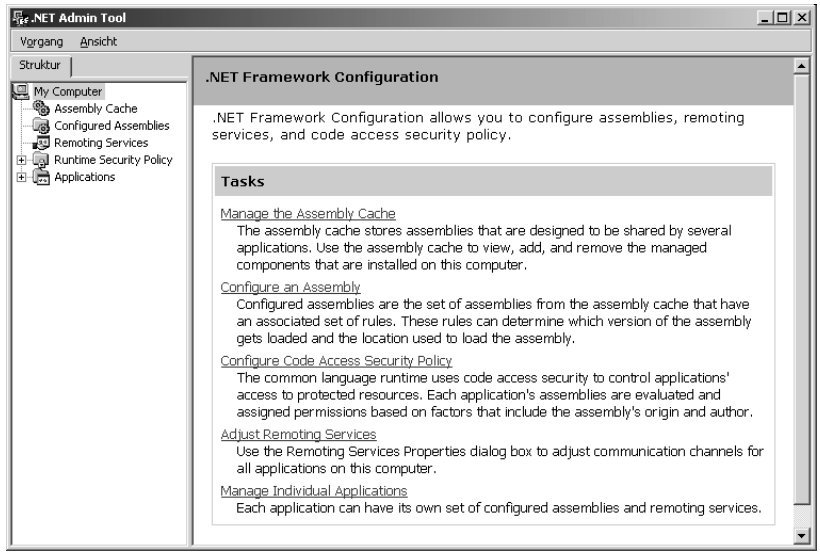


Abbildung 26.8: Das .NET-Konfigurationsprogramm in der MMC

Eine komplette Behandlung dieses Werkzeugs würde zu weit führen. Wir werden daher an dieser Stelle nur die Versionsrichtlinie für unsere Client-Anwendung ändern. Dazu wählen Sie aus der Baumansicht den Punkt APPLICATIONS (ANWENDUNGEN) und wählen dann aus dem Menü ACTION (VORGANG) den Punkt ADD … (HINZUFÜGEN …) aus. Suchen Sie dann den Code für die Clientanwendung (hier `DemoLibClient.exe`) und wählen Sie dann aus den Einträgen für diese Anwendung den Punkt CONFIGURED ASSEMBLIES (KONFIGURIERTE BAUGRUPPEN), um für unsere DemoLib die Einstellungen zu ändern.

Dazu haben Sie drei Möglichkeiten:

- Eine Auswahl eines Assemblies aus dem GAC
- Eine Auswahl aus den externen Assembly-Referenzen der Anwendung
- Die manuelle Angabe der Daten zum Assembly

Für unser Beispiel ist es am einfachsten, den zweiten Punkt zu wählen und dann die `DemoLib.dll` auszuwählen und den Dialog fertig stellen zu lassen.

Nun sollten Sie folgende Anzeige auf dem Bildschirm haben:

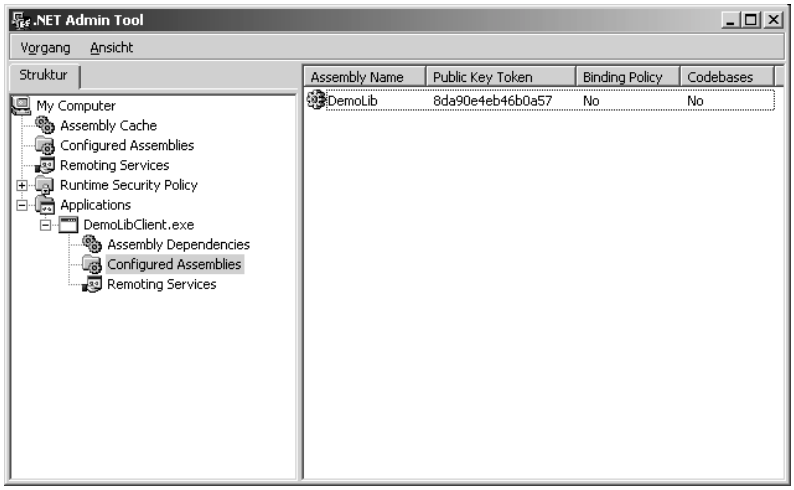


Abbildung 26.9: DemoLib.dll kann jetzt konfiguriert werden

Mit einem Klick der rechten Maustaste auf den Eintrag für die Bibliothek rufen Sie den Eigenschaftsdialog auf und wählen dort den Reiter BINDING POLICY aus.

Hier können Sie nun die gewünschten Eintragungen vornehmen, um die Bibliothek wieder für unseren Client als kompatibel einzustellen:

```
Requested: 1.0.1.1-1.9.9.9
New Version: 2.0.1.42
```

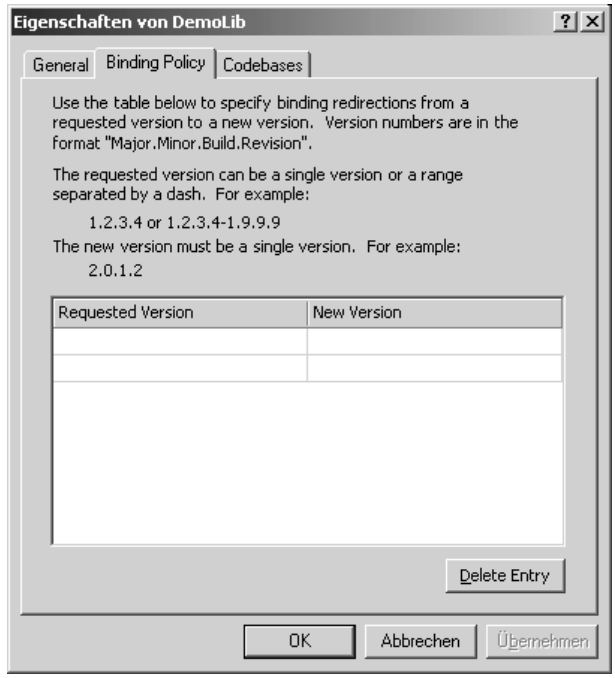


Abbildung 26.10: Die Einstellungen für die Versionsnummern

Nach dem Schließen der Anwendung können wir den Client neu starten und schon ist alles wieder in bester Ordnung!

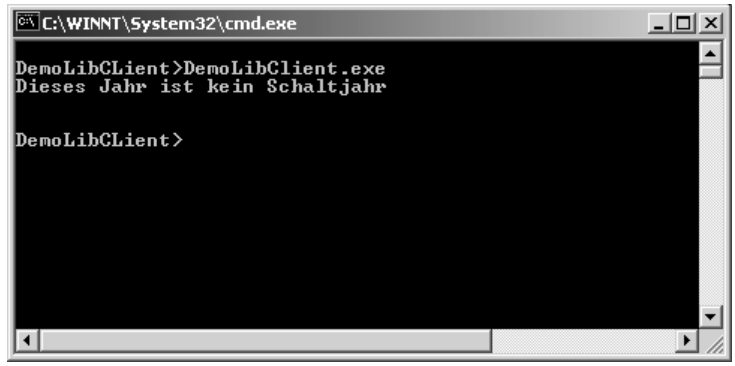


Abbildung 26.11: Nach der Konfiguration läuft der Client wieder

Hier der vom Konfigurationsprogramm erstellte Inhalt der Konfigurationsdatei. Natürlich hätten Sie diese auch von Hand erstellen können, aber als Freund der Kommandozeile bin sogar ich vom Komfort des Admin-Tools überzeugt.

Aus Gründen der leichteren Lesbarkeit habe ich den Zeilenumbruch der Datei etwas verändert, was aber der Gültigkeit der XML-Datei keinen Abbruch tut.

```
<?xml version="1.0"?>
<configuration>
  <runtime>
    <assemblyBinding
      xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="DemoLib"
            publicKeyToken="8da90e4eb46b0a57" />
        <bindingRedirect
             oldVersion="1.0.1.1-1.9.9.9"
             newVersion="2.0.1.42" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Listing 26.8: Inhalt der Datei DemoLibClient.exe.config

Das alles passiert im Verzeichnis der Applikation, ohne Auswirkungen auf andere Anwendungen, ohne Probleme mit einer evtl. im GAC installierten zusätzlichen Version – ein echter Fortschritt!

Ich hoffe, Sie haben damit einen ersten kleinen Einblick in das Thema Assemblies erhalten. Nachdem das Thema noch relativ neu ist, würde mich interessieren, was Sie als Leser für eine eventuelle zweite Auflage des Buches gerne an zusätzlichen Informationen erhalten würden.

26.8 Zusammenfassung

Assemblies sind die Installationseinheiten für eine .NET-Anwendung und enthalten Metadaten zur Beschreibung der im Assembly enthaltenen Datentypen sowie des Assemblys selbst.

Innerhalb der .NET-Laufzeitumgebung wird zwischen privaten und öffentlichen Assemblies unterschieden. Während bei privaten Assemblies die Konfiguration und die Versionsverwaltung unproblematisch ist, da Änderungen hier keine Auswirkung auf andere Anwendungen haben, stehen öffentliche Assemblies systemweit zur Verfügung.

Öffentliche Assemblies verfügen daher über einen starken kryptografischen Namen und werden im zentralen Speicher für öffentliche Assemblies (dem *global assembly cache*) eingetragen. Für das Signieren und die Verwaltung des GAC stellt Microsoft eigene Werkzeuge bereit.

Über XML-basierte Konfigurationsdateien lässt sich das Verhalten einer Anwendung in weiten Grenzen steuern. Sicherheitseinstellungen, Versionsrichtlinien und auch Suchpfade für Assemblies der Anwendung können hier konfiguriert werden. Falls diese Dateien nicht manuell angelegt werden sollten (der Dateiname lautet immer wie das Startassembly der Anwendung plus '.config'), lassen sich die Einstellungen auch über das .NET Admin-Programm in der MMC einstellen.

27 Unsicherer Code

Unsicherer Code? Was hat der in einem ernsthaften Buch zu suchen? Keine Panik, hier handelt es sich nur um einen etwas dramatischen Ausdruck aus dem .NET-Jargon. Unsafe code, wie es im Original heißt, läuft nicht unter dem Management der .NET-Laufzeitumgebung. Dies hat natürlich einige Auswirkungen auf Ihren Code: Da die .NET-Laufzeitumgebung den Code nicht mehr auf Sicherheit von Datentypen und Speicherzugriffen testen kann, benötigt das Assembly, welches den Code enthält, volle Zugriffsrechte. Damit kann unsicherer Code nicht als Download aus dem Web geholt und einfach ausgeführt werden, außerdem wird die Programmierung aufgrund des Wegfalls der Hilfestellungen der .NET-Laufzeitumgebung aufwändiger. Deshalb also die Bezeichnung unsicherer Code.

27.1 Zeiger in C#

Die kurze Zusammenfassung für alle Veteranen der prozeduralen Programmierung: Ja, C# bietet Zeiger und Zeigerarithmetik und ja, Sie können damit Ihr System genau so schön in den Boden rammen wie mit C++ oder C. Lediglich die ganz üblen Fälle des kreativen Einsatzes von Zeigern werden verhindert. Die Wahl des Schlüsselwortes für Blöcke mit Zeigerbenutzung sollte Ihnen Hinweis genug sein.

Eigentlich schreibe ich dieses Kapitel in der Hoffnung, dass Sie die Informationen, die darin enthalten sind, nie brauchen. Unsicherer Code und Zeiger bieten auch in C# den Eintritt in die Geisterbahn der EDV: unauffindbare Fehler, Stacküberlauf, abstürzende Prozesse, unsichere Typkonventierungen und noch viele weitere Überraschungen. Allerdings ist Goto C# ein Buch für Umsteiger von einer anderen Programmiersprache, die bereits über Programmiererfahrung verfügen. Ich werde Sie daher also nicht damit langweilen, dass der Einsatz von Zeigern mit entsprechender Vorsicht erfolgen sollte. Passen Sie einfach beim Programmieren auf, ich drücke Ihnen die Daumen …

Welche Gründe gibt es dann überhaupt für die Verwendung von unsicherem Code? Zuallererst Kompatibilität mit alten Anwendungen. In vielen Fällen wird es in der Übergangszeit zu .NET nötig sein, direkt

Speicher anzusprechen oder Typumwandlung vorzunehmen, die so in C# nicht möglich sind. Der zweite Grund ist Performance. Trotz aller Optimierungen besitzt die Speicherverwaltung mit der Garbage Collection einen gewissen Verwaltungsaufwand, der in bestimmten Fällen die entscheidenen Millisekunden zuviel an Rechenzeit benötigt. Um nicht völlig auf die Vorteile von C# verzichten zu müssen, kann dieser Teil als unsicher markiert werden.

27.2 Das Schlüsselwort unsafe

Welche Bereiche von Code lassen sich als unsicher markieren? Sie können eine komplette Klasse oder Struktur als `unsafe` markieren oder nur einzelne Methoden, unabhängig davon, welche Eigenschaften diese Methoden sonst noch besitzen (`static`, `virtual`, etc.). Wird eine Klasse als unsicher gekennzeichnet, gelten alle Felder und Methoden als unsicher. Ebenso wie Methoden lassen sich auch Felder einer Klasse als unsicher markieren, einzelne lokale Variablen in einer Methode jedoch nicht. Sie können allerdings innerhalb einer Methode einen `unsafe`-Block deklarieren und darin lokale Variablen mit Zeigern benutzen (falls Sie der Meinung sind, Sie brauchen das).

Folgender Code deklariert beispielsweise die Methode `Main()` in unserer Klasse als unsicher:

```
public class Test
{
   public static unsafe void Main()
   {
      // hier koennen pointer verwendet werden
   }
}
```

Listing 27.1: Eine unsichere Methode in einer Klasse

Anscheinend hatten die Entwickler bei Microsoft bereits die eine oder andere schlechte Erfahrung mit Zeigern. Abgesehen vom etwas dramatisch gewählten Schlüsselwort `unsafe` benötigen Sie noch einen zusätzlichen Compilerschalter bei der Übersetzung, bevor C# die Zeiger akzeptiert: die Option `/unsafe`. In Visual Studio.NET setzen Sie diesen Compilerschalter in den Eigenschaften des Projekts bei den KONFIGURATIONSEIGENSCHAFTEN unter dem Punkt ERSTELLEN.

27.3 Nutzung von Zeigern

27.3.1 Deklaration von Zeigern

Um einen Zeiger zu deklarieren, markieren Sie einen Teil Ihres Quellcodes als unsicher und benutzen dann die folgende Syntax:

```
<datentyp> * <variable>;

// beispiel:
int * pInt;
```

Listing 27.2: Deklaration einer Zeigervariablen

C++



Im Gegenatz zu C oder C++ ist C# eine typsichere Sprache und die Entwickler haben versucht, die Betonung des Datentyps auch bei unsicherem Code zu verdeutlichen. Der Stern, der die Deklaration eines Zeigertyps angibt, bezieht sich in C# auf den vorangestellten Typ und nicht auf die einzelne Variable. Dies macht Umsteigern oft das Lesen schwer. Ein Beispiel soll dies verdeutlichen:

Zwei Zeichenzeiger (*char pointer*) deklarieren Sie in C++ wie folgt:

```
char *pChar1, *pChar2;
```

In C# sieht die Deklaration folgendermaßen aus:

```
char* pChar1, pChar2;
```

Der Parser ist in der Beziehung gutmütig: Deklarieren Sie eine Variable, können Sie diese Deklararation entweder als `char* pChar1`, `char *pChar1` oder `char * pChar1` schreiben, je nach persönlicher Vorliebe.

27.3.2 Zeiger-Operatoren

Für die Benutzung stehen drei Operatoren zur Verfügung:

- `*` ist der Dereferenzierungsoperator und erlaubt den Zugriff auf den Inhalt der Speicherstelle, auf die der Zeiger verweist ('hol mir den Inhalt von').

- & ist der Adressoperator und liefert die Adresse einer Variablen oder Referenz.
- -> bietet eine Möglichkeit, Dereferenzierung und Feldzugriff zu verbinden (mehr dazu weiter unten).

27.3.3 Beispiel für Zeigervariablen

Einige einfache Beispiele für die Verwendung von Zeigern in C# zeigt die folgende Klasse:

```
// Goto C# -- Armin Hanisch
// verwendung von zeigern

using System;

namespace UnsafeCode
{
   class TestClass
   {
      static unsafe void Main(string[] args)
      {
         int x = 170;
         int y = 85;
         int* p1, p2;

         p1 = &x;
         p2 = p1;
         Console.WriteLine( *p2 );

         p2 = &y;
         *p1 = *p2;
         Console.WriteLine( *p1 );

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

Listing 27.3: Zeiger in C# – es geht wirklich!

Sehen wir uns der Reihe nach an, was passiert. Zuerst deklariere ich zwei Integervariablen `x` und `y` und weise diesen einen Startwert zu. Danach folgt die Deklaration zweier Zeiger auf einen Integer (C-Pro-

grammierer bitte noch einmal den Hinweis weiter oben lesen). Nun setze ich den ersten Zeiger mit Hilfe des Adress-Operators `&` auf die Adresse der Variablen `x`. Die Zuweisung `p2 = p1` sorgt dafür, dass beide Zeiger den gleichen Wert bekommen, also auf die gleiche Speicherstelle zeigen. Daher führt die Ausgabe des Inhalts der Adresse, auf die der Zeiger `p2` zeigt, ebenfalls zur Ausgabe von `170`.

Anschließend setze ich `p2` auf die Adresse der Variablen `y`. Die folgende Anweisung `*p1 = *p2` unterscheidet sich von der ersten durch die Angabe des Dereferenzierungsoperators `*`, also werden nicht die Zeiger zugewiesen, sondern die Inhalte der Speicherstellen, auf welche die Zeiger verweisen. Ergebnis: die Variable `x` erhält ebenfalls den Inhalt `85`.

Und da es sich bei einer Integervariablen um einen Werttyp handelt, spielt sich das Ganze auf dem Stack und nicht auf dem Heap ab.

27.3.4 Typumwandlungen von Zeigern

Der Inhalt eines Zeigers ist, wie Sie gerade erfahren haben, einfach eine Speicheradresse. Daher können Sie Zeigerwerte auch ausgeben. Die Methode `WriteLine` der Klasse `Console` besitzt allerdings keine Überladung für Zeiger, Sie müssen die Zeiger daher vorher in einen Integertyp umwandeln.

```
Console.WriteLine("Zeiger zeigt auf {0}", (uint)p1 );
```

Listing 27.4: Umwandlung eines Zeigers in einen Integerwert



Beachten Sie bei diesen Umwandlungen, dass ein Zeiger eine höchst systemspezifische Angelegenheit darstellt. Heutige Windows-Versionen benutzen noch Adressen mit einer Breite von 32 Bit, allerdings ist die erste 64Bit-Version von Windows XP bald erhältlich und dann bekommen Sie eine Menge Probleme, wenn Sie versuchen, 64 Bit in einen `uint` zu quetschen …

Achten Sie bei der Umwandlung also immer auf entsprechend breite Datentypen. Innerhalb eines `unsafe`-Blocks erfolgt bei einem Überlauf kein Fehler, auch nicht innerhalb eines `checked`-Blocks! Die .NET-Laufzeitumgebung geht davon aus, dass Sie wissen, was Sie da machen.

Natürlich können Sie einen Zeigertyp in einen anderen Zeigertyp umwandeln. Dazu werden wir aus dem Integerzeiger einen Zeiger auf ein Zeichen machen:

```
// Goto C# -- Armin Hanisch
// verwendung von zeigern, teil 2

using System;

namespace UnsafeCode
{
   class TestClass
   {
      static unsafe void Main(string[] args)
      {
         int x = 170;
         int y = 85;
         int* p1, p2;
         char* cPtr;

         p1 = &x;
         p2 = &y;

         cPtr = (char*)p2;
         Console.WriteLine("p1 als Zeichen: {0}",
                           *cPtr);

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

Listing 27.5: Umwandlung von Zeigertypen

Als Ausgabe erhalten Sie dann folgende Zeile:

```
p1 als Zeichen: U
```

da das Zeichen »`U`« den ASCII-Wert 85 besitzt.

Die Anweisung `cPtr = (char*)p2` ist also völlig in Ordnung, allerdings sollten Sie bei solchen Sachen genau wissen, was Sie tun! Hier arbeiten Sie völlig außerhalb der Überprüfung der .NET-Laufzeitumgebung und können nach Herzenlust ein `byte` als `double` auffassen.

void-Zeiger

Es gibt auch in C# den völlig typlosen 'Einfach nur so ein Zeiger'-Zeiger. Dieser wird als `void*` deklariert, bietet aber bei weitem nicht die Funktionalität wie z.B. in C++ oder C, da mit diesem Zeiger keine Zeigerarthmetik (s.u.) oder eine Dereferenzierung über * möglich ist. Letzlich bleibt nur der Aufruf von API-Funktionen mit reinen Zeigerparametern.

27.3.5 sizeof und Zeigerarithmetik

Für alle Werttypen können Sie mit Hilfe des Operators `sizeof` feststellen, wie viele Bytes dieser Datentyp physikalisch belegt. Da Strukturen ebenfalls Werttypen sind, ist die Verwendung von `sizeof` auch dafür zugelassen. Klassen als Referenztypen können nicht verwendet werden.

Diese Größen von Datentypen werden auch bei der Zeigerarithmetik benötigt. Sie können, wie in anderen Programmiersprachen auch, Zeiger in C# addieren, subtrahiern oder mit Werten verändern. Dazu bedient sich der Compiler der Funktion des `sizeof`-Operators und ändert den tatsächlichen Adresswert in Abhängigkeit vom Datentyp. Zählen Sie zu einem `int*` beispielsweise 1 hinzu, so liegt die neue Adresse, auf welche der Zeiger verweist, um 4 Byte höher im Speicher, da ein `int` eine Größe von 4 Byte besitzt.

Für die Zeigerberechnungen können Sie alle Additions- und Subtraktionsoperatoren verwenden. Für `void`-Zeiger ist diese Arithmetik allerdings nicht zugelassen, da hier keine Informationen über den zugrunde liegenden Datentyp vorhanden sind!

Auch hier wieder ein kleines Beispiel mit einer abgeänderten Version der Klasse:

```
// Goto C# -- Armin Hanisch
// zeigerarithmetik mit c#

using System;

namespace UnsafeCode
{
   class TestClass
   {
```

```
        static unsafe void Main(string[] args)
        {
            int x = 170;
            int y = 85;
            char* cPtr;
            int* p1, p2;

            p1 = &x;
            p2 = p1;

            cPtr = (char*)p1;
            Console.WriteLine("Wert von cPtr = {0}",
                              (uint)cPtr);

            Console.WriteLine("p1-0 als Zeichen ist '{0}'",
                              *cPtr );
            cPtr--;
            Console.WriteLine("p1-2 als Zeichen ist '{0}'",
                              *cPtr );
            cPtr--;
            Console.WriteLine("p1-4 als Zeichen ist '{0}'",
                              *cPtr );
            cPtr--;
            Console.WriteLine("p1-6 als Zeichen ist '{0}'",
                              *cPtr );

            Console.WriteLine("Wert von cPtr = {0}",
                              (uint)cPtr);

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

Listing 27.6: Zeigerarithmetik mit C#: Wanderung am Stack entlang

Hier wird zuerst die Adresse der Variablen x bestimmt und dann über eine Konvertierung dieses Integer-Zeigers in einen `char*` diese Speicherstelle als Zeichen aufgefasst. Anschließend wird das 'Zeichen' an dieser Stelle ausgegeben und einige Male die Adresse verändert, so dass wir sehen, wie sich die Daten ändern.

Falls Sie sich jetzt wundern, wieso sich für ein Zeichen (`cPtr--`) die Adresse um zwei Byte ändert: Denken Sie daran, dass die .NET-Laufzeitumgebung komplett Unicode-fähig ist, ein Zeichen also immer zwei Byte belegt. Dann kann ich auch gleich das zweite eventuell bestehende Rätsel auflösen: Wieso verringere ich die Speicheradressen, um durch den Speicher zu wandern? Ganz einfach: Die Variablen sind Werttypen und liegen auf dem Stack. Ein Stack im Rechner wird per Konvention immer von oben nach unten belegt, d.h. die Variable `y` liegt um einen `int` (4 Byte) tiefer als `x`. Deshalb finden Sie 4 Byte 'unter' der ersten Ausgabe auch unsere Variable `y` mit dem Wert `85` (dem »U«).

Dieses Programm liefert die folgende Ausgabe, anhand derer Sie die Funktion noch einmal nachvollziehen können:

```
Wert von cPtr = 1243468
p1-0 als Zeichen ist 'ª'
p1-2 als Zeichen ist ' '
p1-4 als Zeichen ist 'U'
p1-6 als Zeichen ist ' '
Wert von cPtr = 1243462
```

27.4 Der Operator ->

Auch Strukturen können als Werttypen auf dem Stack über Zeiger angesprochen werden. Die sich daraus ergebende Standardsyntax sieht allerdings nicht sonderlich elegant aus und widerspricht somit den Designprinzipien von C#. Daher ist an dieser Stelle auch der Zugriff über einen besonderen Operator möglich. Aber zuerst eine kleine Struktur zur Verdeutlichung:

```
struct Koordinaten
{
   double x;
   double y;
}
```

Nun können wir eine Struktur mit diesem Typ und einen Zeiger definieren:

```
Koordinaten coords = new Koordinaten();
Koordinaten* ptr = &coords;
```

Für den Zugriff auf die Felder sieht der Ausdruck allerdings so aus:

```
double wert = (*ptr).x;
```

Nicht besonders schön, oder? Daher exisitiert in C# der Kombinationsoperator für Dereferenzierung und Feldzugriff, `->`. Mit dessen Hilfe lässt sich der obige Ausdruck intuitiver formulieren:

```
double wert = ptr->x;
```

27.5 Zusammenfassung

Falls Sie für eine Anwendung Zugriff auch nicht verwalteten Speicher benötigen oder Code aus Zeiten vor .NET in die Anwendung einbinden müssen, dann können Sie dies mit unsicheren (mit dem Schlüsselwort `unsafe` markierten) Code erreichen.

Dieser Code ist von der .NET-Laufzeitumgebung nicht verifizierbar und kann daher nur mit entsprechend erweiterten Zugriffsrechten ausgeführt werden. Dann allerdings sind Operationen wie Zeigerarithmetik oder die ungeprüfte Umwandlung von Zeigertypen möglich.

Zusammen mit den Möglichkeiten, Routinen in DLL-Bibliotheken über das Attribut `DLLImport` anzusprechen, haben Sie damit eine Möglichkeit, alten Code weiter zu verwenden.

28 XML-Dokumentation

Die hinter diesem Prinzip stehende Idee ist einfach – die beste Zeit, den Code zu dokumentieren ist dann, wenn Sie ihn schreiben. Die Dokumentation wird als Teil des Quellcodes gespeichert und bei Bedarf mit einem geeigneten Werkzeug extrahiert.

Der Erfinder dieser Idee ist Donald Knuth, dessen Web-Dokumentationssystem mit den Tools `tangle` und `weave` das Vorbild für viele ähnliche Ansätze war. Auch das in der Java-Gemeinde bekannte `javadoc` basiert auf dieser Idee. C# ist allerdings die erste Programmiersprache, bei der die Unterstützung einer solchen integrierten Dokumentation Bestandteil der Sprache und des Compilers ist. Ein weiterer Fortschritt gegenüber anderen Ansätzen mit einem eigenen Format ist die Festlegung auf ein wohlgeformtes XML-Dokument für die extrahierte Dokumentation.

Natürlich kann ich mir auch hier nicht verkneifen, die eine oder andere Idee vorzustellen (vielleicht dringt sie ja bis zum C#-Team vor). Es wäre schön, wenn man wie bei Knuths System eine Möglichkeit hätte, die Dokumentation in der XML-Datei zu verändern und anschließend über eindeutige Bezeichner der einzelnen Elemente wieder den Quellcode zurück zu packen. Damit wäre wirklich ein Zwei-Wege-Werkzeuge geschaffen, mit dem man auch außerhalb der Entwicklungsumgebung die Dokumentation bearbeiten könnte. Aber das wird wohl ein Traum bleiben …

28.1 XML-Kommentare

Vielleicht haben Sie sich schon gewundert, warum Visual Studio bei vielen Kommentaren drei Schrägstriche macht, wenn doch laut Sprachspezifikation für einen Kommentar in C# zwei ausreichen (oder warum für mehrere Zeilen Kommentar nicht die Kombination aus `/*` und `*/` benutzt wird).

Der Grund ist der, dass C# mit den drei Schrägstrichen XML-Kommentare kennzeichnet, in denen die oben erwähnte integrierte Dokumentation enthalten ist. Diese Elemente werden dann vom Compiler

extrahiert und in eine XML-Datei geschrieben. Mit Hilfe von anderen Anwendungen oder einem XSL-Stylesheet können Sie die Dokumentation dann mehr oder weniger beliebig umformen.

28.1.1 Eingabe von XML-Kommentaren

Erstellen Sie im Quellcode einfach einen Kommentar, der mit drei anstatt zwei aufeinanderfolgenden Schrägstrichen beginnt und fügen Sie dahinter die entsprechenden (weiter unten aufgeführten) Dokumentations-Elemente ein.

In Visual Studio.NET geben Sie einfach nur die drei Schrägstrichen ein und die IDE liest aus dem vorhandenen Quellcode sogar die Parameterliste und andere Informationen aus. Die folgende Abbildung zeigt das automatisch generierte Gerüst, sobald Sie in der Zeile über der Methode den dritten »/« eingegeben haben.

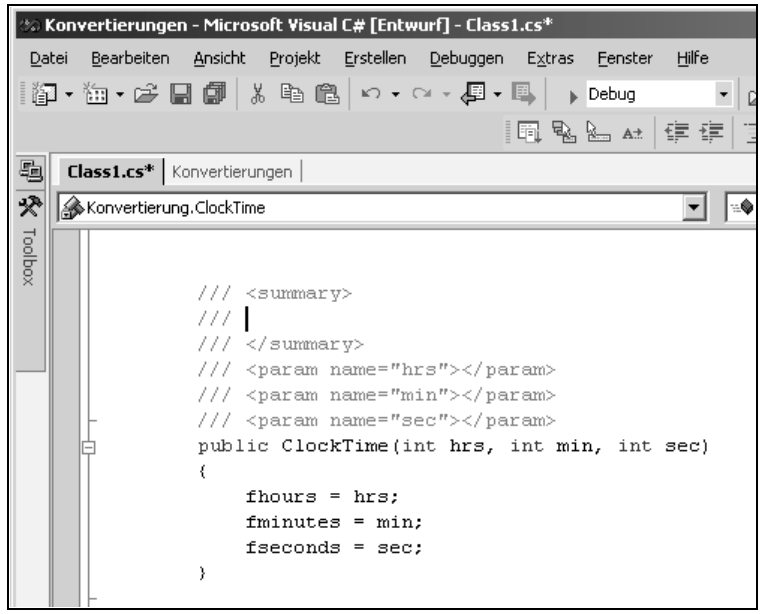


Abbildung 28.1: Automatisch generierte XML-Dokumentationselemente

28.1.2 Unterstützte XML-Elemente

Folgende XML-Tags werden vom Compiler erkannt. Bei den mit einem hochgestellten (SC) gekennzeichneten Tags überprüft der Compiler auch die Syntax des Elementinhalts (mehr dazu unter ***Syntaxprüfung der Elementinhalte***).

<c>	<list>	<returns>
<code>	<param> (SC)	<see> (SC)
<example>	<paramref> (SC)	<seealso> (SC)
<exception> (SC)	<permission> (SC)	<summary>
<include> (SC)	<remarks>	<value>

Tabelle 28.1: Liste der unterstützten XML-Dokumentations-Tags

Aus dem Kapitel über benutzerdefinierte Konvertierungen habe ich eine Methode mit XML-Kommentaren versehen, die für die nächsten Beispiele verwendet wird.

```
/// <summary>
/// Operator für die explizite Typwandlung von
/// Integer nach ClockTime.
/// Explizit, da der Integer-Wertebereich grösser ist
/// als der mögliche Wertebereich einer Variablen vom
/// Typ ClockTime.
/// </summary>
/// <exception
        cref="System.ArgumentOutOfRangeException">
/// Falls die Konvertierung nicht durchgeführt werden
/// kann, löst dies eine ArgumentOutOfRangeException
/// aus.
/// </exception>
/// <param name="seconds">
/// Sekunden seit Mitternacht
/// </param>
public static explicit operator ClockTime(int seconds)
{
   // code to convert from int to ClockTime
   if((seconds > 0) && (seconds < 86400))
      return new ClockTime(seconds);
   else
```

```
        throw new ArgumentOutOfRangeException("seconds",
                  "0 <= seconds < 86400");
}
```

Listing 28.1: Mit XML-Kommentaren versehene Methode

28.1.3 Generieren der Dokumentationdatei

Starten Sie den C# Compiler mit dem Schalter `/doc:<datei>` vor der Angabe der Quelldatei und es wird eine XML-Datei mit dem angegebenen Namen erzeugt. Aus dieser können sie dann (z.B. über XSL) eine HTML-Seite erstellen.

```
csc /doc:konv.xml konv.cs
```

Falls Sie Visual Studio verwenden, ist die Sache sogar noch einfacher: Sie wählen aus dem Menü EXTRAS einfach den Punkt ERSTELLEN VON KOMMENTARWEBSEITEN … und die IDE erledigt den Rest, erstellt sogar komplett verlinkte HTML-Seiten, die aus der IDE heraus betrachtet werden können.

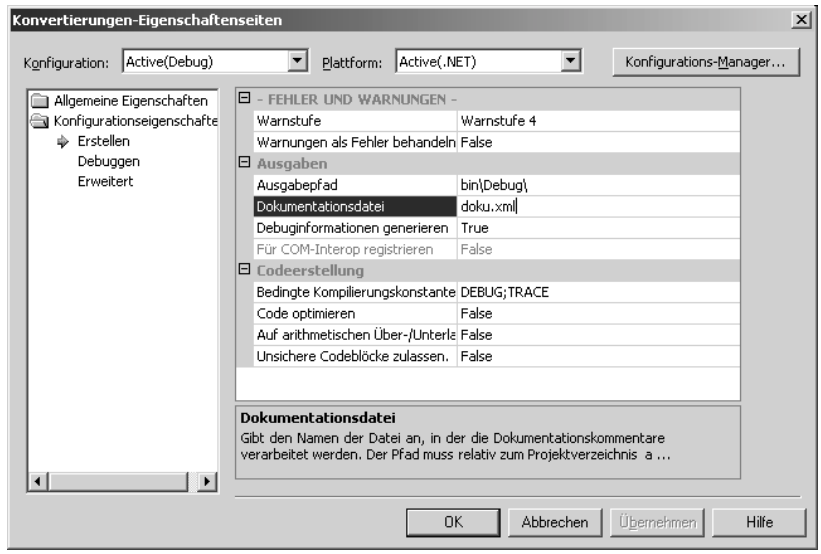


Abbildung 28.2: Die Einstellung des Namens der XML-Datei in Visual Studio

Sie können auch bei den Projekteigenschaften unter den Erstellungsoptionen bei Visual Studio einstellen, ob eine XML-Datei zusätzlich erstellt werden soll und wie diese heißen soll.

Das dann für die Methode erstellte XML-Fragment besitzt diesen Inhalt (die komplette Datei können Sie sich leicht selbst für ein eigenes Projekt generieren, diese wird in der folgenden Abbildung im Browser dargestellt).

```
<member name="M:Konvertierung.ClockTime.op_Explicit(
              System.Int32)~Konvertierung.ClockTime">
   <summary>
   Operator für die explizite Typwandlung von Integer
   nach ClockTime. Explizit, da der Integer-Wertebereich
   grösser ist als der mögliche Wertebereich einer
   Variablen vom Typ ClockTime.
   </summary>
   <exception
       cref="T:System.ArgumentOutOfRangeException">
    Diese Exception wird ausgelöst, wenn die
    Konvertierung nicht durchgeführt werden kann.
    </exception>
    <param name="seconds">
    Sekunden seit Mitternacht
    </param>
</member>
```

Listing 28.2: Die vom Compiler generierte XML-Datei

Wird die komplette Datei mit einem XML-fähigen Browser betrachtet, können Sie in der Struktur auch in gewissen Grenzen navigieren.

Die fertigen Kommentarwebseiten in Visual Studio sehen natürlich leichter lesbar aus und können direkt aus der IDE heraus erstellt und betrachtet werden. Wenn die in den Beta-Versionen von Visual Studio noch enthaltenen Probleme in der Release-Version behoben sind, wird dies ein sehr nützliches Tool werden.

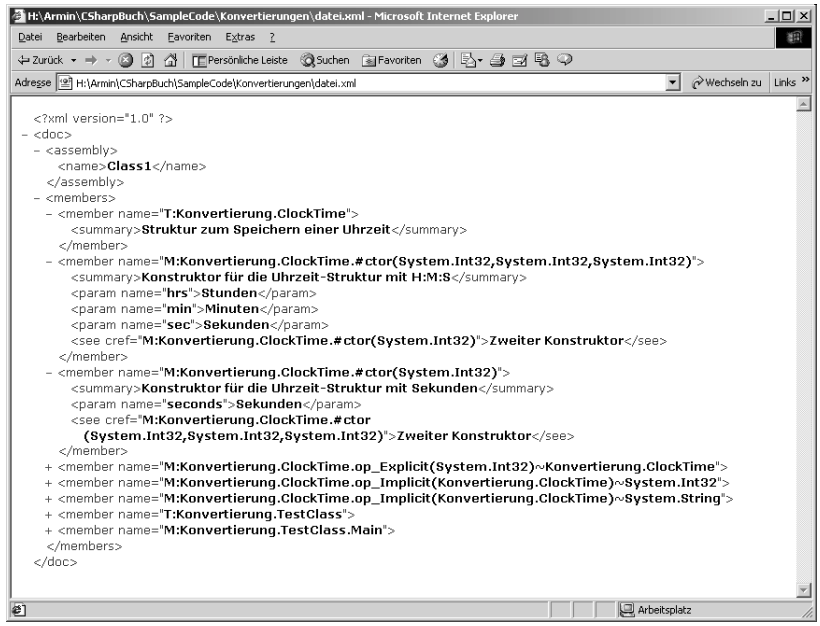


Abbildung 28.3: Die XML-Datei im Internet Explorer

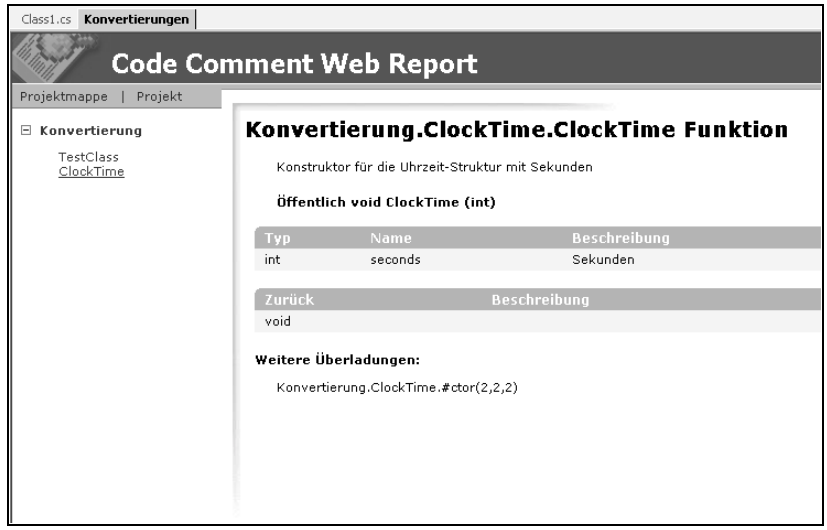


Abbildung 28.4: Die Kommentarwebseiten in Visual Studio

28.1.4 Syntaxprüfung des Elementinhaltes

Der Compiler führt bei den in der Tabelle 28.1 mit (SC) gekennzeichneten Tags eine Syntaxprüfung durch. Wird das Element gefunden, wird es zu einem vollqualifizierten Namen erweitert und eingetragen. Aus einem `operator ClockTime(int)`wird folgender Eintrag (alles in einer Zeile!):

```
M:Konvertierung
.ClockTime
.op_Explicit(System.Int32)
~Konvertierung.ClockTime
```

Dies kennzeichnet einen Member (»M:«) eines Datentyps, aus dem Namensraum Konvertierung, dort der Klasse oder Struktur ClockTime. Es handelt sich um einen expliziten Operator, der einen Int32 als Parameter erwartet und den Datentyp ClockTime aus dem Namensraum Konvertierung zurückgibt.

Ist der Name eines angegebenen Typs oder eines Parameters nicht vorhanden, bekommen Sie eine Warnung. Sie erhalten ebenso eine Warnung, wenn Sie vergessen, einen Parameter zu kommentieren.

```
warning CS1572: Der XML-Kommentar bei
'Konvertierung.ClockTime.explicit operator
Konvertierung.ClockTime(int)' hat einen Parametertag für
'sekonds', aber bei diesem Namen steht kein Parameter.
warning CS1573: Der Parameter 'seconds' hat keinen passenden
Parametertag im XML-Kommentar  (aber andere Parameter haben
einen).
```

Listing 28.3: Mögliche Warnungen des Compilers bei Dokumentations-Tags

28.2 Zusammenfassung

Die Möglichkeit, durch XML-Kommentare den Quellcode mit einer ausführlicheren Dokumentation zu versehen, die automatisch extrahiert werden kann, ist eine interessante Ergänzung von C#.

Obwohl Visual Studio den Prozess wesentlich erleichtert, ist die entsprechende Funktionalität bereits in der Kommandozeilenversion des Compilers über den Schalter `/doc:<datei>` enthalten. Sie benötigen lediglich einen XML-Browser oder ein XSL-Stylesheet zur Umwandlung in HTML.

Für viele Tags führt der Compiler eine Syntaxprüfung und eine Erweiterung auf einen kompletten Typnamen durch, so dass alle notwendigen Typinformationen ebenfalls in die XML-Datei mit aufgenommen werden.

29 Lösungen zu den Übungen

29.1 Kapitel 4

29.1.1 Die Fragen

1. Wie können Sie zwei Methoden mit gleichem Namen in Ihrer Anwendung benutzen, von denen jede in einem eigenen Namensraum vorhanden ist?
2. Suchen Sie in der Online-Dokumentation, in welchem Namensraum die Klasse BinaryFormatter vorhanden ist.
3. Welche Funktion hat die folgende Quellcodezeile?

```
using regexp = System.Text.RegularExpressions;
```

29.1.2 Die Antworten

Antwort 1

Dazu importieren Sie die beiden Namensräume mit je einem eigenen Alias und setzen den Aliasnamen als Präfix vor den Namen der Klasse, deren Methode Sie aufrufen möchten.

```
// beide namensräume bieten eine Klasse "regexp"

using abc = FirmaA.Textparsing.RegExp;
using xyz = FirmaB.Text.RegularExp;

// nutzen beider klassen und ihrer methoden
abc.regexp.Find(string, ausdruck);
xyz.regexp.Find(string, ausdruck);
```

Theoretisch würde es auch reichen, nur einen Alias zu verwenden oder immer beide Klassen vollqualifiziert anzugeben, diese Ansätze sollten aber aus Gründen der Lesbarkeit des Codes vermieden werden. Aus dem gleichen Grund sollten Sie auch sinnvollere Bezeichner für den Präfix wählen als `abc` …

Antwort 2

`System.Runtime.Serialization.Formatters.Binary`

Antwort 3

Diese Codezeile importiert die Klassen des angegebenen Namensraumes und stellt diese unter dem Aliasnamen `regexp` zur Verfügung. Damit können Konflikte durch die Verwendung gleichnamiger Klassen aus verschiedenen Namensräumen in einer Anwendung vermieden werden.

29.2 Kapitel 5

29.2.1 Die Fragen

1. Welche Vorteile bietet das sog. Boxing?
2. Welche Bedeutung besitzt die Angabe von 'M' hinter einer Zahlenkonstanten?
3. Warum sollten Sie vermeiden, mehrere Strings beispielsweise in einer Schleife mit dem '+'-Operator zu verketten? Worin liegt der Nachteil dieser Methode?
4. Sie möchten gerne sicherstellen, dass ein Wert während der Programmlaufzeit nicht unabsichtlich verändert werden kann. Eine Deklaration als Konstante scheidet aber aus, da Sie den Initialwert erst zur Laufzeit des Programms bestimmen können. Welche Möglichkeiten haben Sie, um das gewünschte Ergebnis zu erreichen?
5. Sie schreiben eine Methode, die zwei Werte liefert. Welche Möglichkeiten stehen Ihnen bei C# zur Realisierung zur Verfügung?
6. Warum müssen Sie bei einer Variablen, die als `out`-Parameter übergeben wird, nicht vorher eine Initialisierung durchführen?

29.2.2 Die Antworten

Antwort 1

Damit erlaubt C# die Realisierung des Konzeptes 'Alles ist ein Objekt'. Wird auf einen der einfachen Werttypen als Objekt zugegriffen, erzeugt die .NET-Laufzeitumgebung eine leere 'Objekthülle' und plat-

ziert dort eine Kopie des Werttyps. Dies ermöglicht es, auch beispielsweise einen Integer wie ein Objekt zu behandeln, vermeidet aber bei der normalen Berechnung von Werttypen ohne das Boxing den Performancenachteil anderer, reiner OOP-Sprachen.

Antwort 2

Die Angabe von »M« hinter einer Zahlenkonstante kennzeichnet diese als Währungstyp (`currency`). Ein »F« legt den Typ auf `float` fest und ohne jede Angabe wird automatisch der Datentyp `double` benutzt.

Antwort 3

Strings sind in der .NET-Laufzeitumgebung nicht veränderliche Zeichenkonstanten, die in einem eigenen Speicherpool gehalten werden. Wird eine neue Stringvariable mit dem gleichen Inhalt erstellt, wie ein bereits bestehender String, liefert die .NET-Laufzeitumgebung einfach die Adresse des bestehenden Strings zurück und belegt so den Speicherplatz nur einmal. Aus diesem Grund muss auch verhindert werden, dass sich der Inhalt eines String ändern kann.

Weisen Sie einem String die Verkettung zweier Ausgangsstrings zu, so legt die .NET-Laufzeitumgebung einen neuen String an, kopiert dort die beiden Ausgangsstrings hinein und gibt diese anschließend für die Garbage Collection frei. Daher ist die Verkettung in Schleifen ein relativ aufwändiger Vorgang. Benutzen Sie für die Veränderung von Zeichenketten daher eine Instanz der Klasse StringBuilder, diese kann bis zu 100 mal schneller arbeiten.

Antwort 4

Benutzen Sie dafür eine Readonly-Variable. Diese mit dem Schlüsselwort `readonly` vor dem Typ definierte Variable erlaubt ein einziges Mal eine Zuweisung eines Initialwertes und kann anschließend nur noch gelesen werden.

Antwort 5

Der weniger gute Ansatz wäre die Speicherung des zweiten Resultats in einem öffentlichen Feld der Klasse. Dies widerspricht aber dem Prinzip der Kapselung in der objektorientierten Programmierung, da Sie die Methode ohne das Feld nicht mehr benutzen könnten.

Verwenden Sie stattdessen für diese Zwecke `out`-Parameter. Ein mit diesem Modifikator gekennzeichneter Parameter kann innerhalb der Methode gesetzt werden und darf nur eine Variable sein, deren Inhalt nach dem Aufruf der Methode gelesen werden kann.

Antwort 6

Da ein `out`-Parameter nur dazu dient, innerhalb einer Methode gesetzt zu werden, was der Compiler auch prüft (weisen Sie einem `out`-Parameter in der Methode keinen Wert zu, erhalten Sie eine Fehlermeldung bei der Übersetzung), ist es unerheblich, welchen Anfangswert dieser Parameter hat und daher kann die zwangsweise Initialisierung entfallen.

29.3 Kapitel 6

29.3.1 Die Fragen

1. Erläutern Sie kurz den Unterschied zwischen einer Klasse und einem Objekt.
2. Welchen Vorteil bieten Eigenschaften gegenüber öffentlichen Feldern?
3. Sollte die Kopplung oder die Kohäsion einer Klasse maximiert werden?
4. Welche Designregeln sollten Sie für die Entscheidung zwischen Eigenschaften und Methoden zur Änderung von Daten der Klasse beachten?

29.3.2 Die Antworten

Antwort 1

Eine Klasse beinhaltet die Definition eines Objektes als textuelle Beschreibung seiner Mitglieder. Sie stellt sozusagen den Bauplan für ein reales Objekt dar. Ein Objekt ist die Instanz einer Klasse, also ein nach diesem Bauplan im Speicher erstelltes Gebilde. Erst von dieser Instanz können Sie Methoden und Felder nutzen. Eine Ausnahme stellen statische Klassen bzw. die statischen Methoden einer Klasse dar. Diese (in anderen Programmiersprachen als 'shared methods' oder 'class func-

tions' bezeichneten) Mitglieder einer Klasse werden für alle Instanzen einer Klasse nur einmal erzeugt und sind mit der Definition einer Klasse verfügbar. Aufgerufen werden statische Klassenmitglieder über den Namen der Klasse, während normale Instanzmethoden über den Namen der Objektvariablen aufgerufen werden.

Antwort 2

In der Regel dienen Eigenschaften als definierte Schnittstelle, so dass interne Details der Datenspeicherung vor Benutzern der Klasse und deren Objekten verborgen bleiben können. Stellt eine Klasse beispielsweise eine Eigenschaft 'Nachrichtenanzahl' zur Verfügung, ist es für die Benutzer des Objektes unerheblich, ob diese Anzahl intern in einem Attribut gespeichert, über eine Methode zur Zeit der Abfrage erst berechnet oder extern von einem anderen Server geholt wird. Weitere Informationen finden Sie im Abschnitt ***OOP-Vorteile von C#-Eigenschaften*** des Kapitels 7 ***Klassen in C#***.

Antwort 3

Kopplung beschreibt die gegenseitige Abhängigkeit funktionaler Module einer Anwendung oder Klasse. Als Kohäsion bezeichnet man den Grad der logischen Zusammengehörigkeit und Abgeschlossenheit einer Methode oder Klasse. Daher sollte die Kopplung zwischen Bestandteilen einer Anwendung minimiert und die Kohäsion der Module maximiert werden, unabhängig davon, ob dies auf Methoden- und Klassenebene oder auf Ebene von Anwendungsmodulen geschieht. Mehr Informationen dazu erhalten Sie in Kapitel 6 ***OOP-Grundlagen***.

Antwort 4

Eine Eigenschaft sollte dem logischen Modell einer tatsächlichen Eigenschaft eines Dings entsprechen und daher sowohl lesbar als auch schreibbar sein (ebenso, wie meine Katze mal satt, mal hungrig sein kann). Informationen, die Sie nur schreiben oder nur lesen können, sollten idealerweise mit einer Methode und einem entsprechenden Namen erstellt werden (das Geburtsdatum der Katze ist daher kein guter Kandidat für eine Eigenschaft, da sich diese Information nicht mehr ändert). Feststehende Informationen einer Instanz lassen sich auch über Konstanten abbilden, bei einer über alle Instanzen einer

Klasse feststehenden Information auch über statische Methoden (Konstanten in einer Klasse sind generell immer statisch).

29.4 Kapitel 7

29.4.1 Die Fragen

1. Welcher der beiden Modifikatoren ist restriktiver: `protected` oder `private`?
2. Warum sind Klassenmitglieder ohne Modifikator per default `private`?
3. Welche Einschränkung hat C# bei Methoden mit variabler Parameterzahl?
4. Wozu dient der Bezeichner `value` und wo wird dieser definiert?
5. Erstellen Sie eine Klasse mit überladenen Konstruktoren zur Speicherung einer Strecke in einem zweidimensionalen Koordinatensystem.

29.4.2 Die Antworten

Antwort 1

`private` ist nur innerhalb der Klasse selbst sichtbar und der Standardgültigkeitsbereich eines Klassenmitglieds (aus Sicherheitsgründen), `protected` ist in der Klasse selbst und in allen davon abgeleiteten Klassen sichtbar. Daher ist `private` restriktiver.

Antwort 2

Die standardmäßige Beschränkung auf `private` ist eine Sicherheitseinstellung. Alle so erstellten Klassenmitglieder sind nur innerhalb dieser Klasse selbst sichtbar und können daher das nach außen sichtbare Verhalten der Klasse nicht verändern.

Antwort 3

Die Erstellung eines `params`-Arrays für die Übergabe schränkt alle darin übergebenen Parameter auf den gleichen Datentyp ein, da alle Mitglieder eines Arrays den gleichen Datentyp besitzen müssen. Eine Alterna-

tive wäre ein `params`-Array vom Typ `Object`, da in C# alle Datentypen davon abstammen und so alle Datentypen übergeben werden können.

Antwort 4

`value` ist der Wert, den eine Eigenschaft beim Setzen über die `set`-Methode erhält. Dieser Bezeichner ist vordefiniert und auch nur in der Methode zum Setzen des Wertes einer Eigenschaft verfügbar. Der Datentyp stimmt immer mit dem Datentyp der Eigenschaft überein.

Antwort 5

```
public class Strecke
{
   public Strecke(int x1, int y1,
                  int x2, int y2)
   {
   }

   public Strecke(int x1, int y1,
                  double winkel, double laenge)
   {
   }

   public Strecke(double winkel1, double laenge1,
                  double winkel2, double laenge2)
   {
   }
}
```

Sie können nun eine Instanz der Klasse über alle drei Möglichkeiten der Angabe einer Strecke im 2D-System erstellen.

29.5 Kapitel 8

29.5.1 Die Fragen

1. Warum liefert der Code `if(x = 3) { Console.Write("ungerade"); }` eine Fehlermeldung bei der Übersetzung?
2. Sie wollen bei der Erfüllung einer bestimmten Bedingung in einer `while`-Schleife die Schleifenabarbeitung abbrechen und die Schleife komplett verlassen. Benutzen Sie dazu `break` oder `continue`?

3. Wie lässt sich vor einer Typumwandlung feststellen, ob eine Objektvariable auch von einem bestimmten Datentyp ist?
4. Sie wollen in einer `foreach`-Schleife feststellen, wie oft diese bereits durchlaufen wurde. Wie würden Sie den Code formulieren?
5. Was ist das Ergebnis von `8 ^ 3` in C#?

29.5.2 Die Antworten

Antwort 1

Das ist bei C/C++-Programmierern ein klassischer Bug: Ein einzelnes Gleichheitszeichen ist kein Vergleich, sondern eine Zuweisung! Da der Typ dieses Ausdrucks allerdings keinen Datentyp `bool` ergibt, der laut Syntax an dieser Stelle vorgeschrieben ist, kann der C#-Compiler diese Konstruktion erkennen und den Fehler so aufzeigen.

Antwort 2

Das richtige Statement ist `break`.

Antwort 3

Sie besitzen dazu zwei Möglichkeiten. Zum einen können Sie mit Hilfe des Typoperators `is` feststellen, ob es sich um einen bestimmten Typ handelt und dann sicher eine Typumwandlung durchführen. Die zweite (und in C# empfohlene) Möglichkeit ist die Verwendung des Operators `as` und ein anschließender Test auf eine `null`-Referenz.

```
// möglichkeit 1
Object objvar;
clsKunde mykunde;

if (objvar is clsKunde)
   mykunde = (clsKunde)objvar;

// möglichkeit 2
Object objvar;
clsKunde mykunde;

mykunde = objvar as clsKunde;
if (mykunde != null)
   // hier arbeiten mit mykunde
```

Antwort 4

Fangfrage! Es gibt im Gegensatz zu `for` bei `foreach` keine offizielle Möglichkeit, an den Schleifenzähler zu gelangen. Daher bleibt nur die Nutzung einer zusätzlichen Variable, die vor der Schleife auf 0 gesetzt wird und dann zu Beginn jedes Durchlaufs um 1 erhöht wird. Als Alternative wäre der Einsatz einer `for`-Schleife anstelle von `foreach` denkbar, wenn Sie Zugriff auf den Schleifenzähler benötigen.

Antwort 5

Lassen Sie sich nicht vom optischen Eindruck des Operators täuschen, wenn Sie nicht von C oder C++ kommen! Dieser Operator führt ein bitweises XOR durch und liefert damit das Ergebnis 11 (`1000 xor 0011 => 1011` binär).

29.6 Kapitel 9

29.6.1 Die Fragen

1. Können Sie bei der Erstellung einer Aufzählung Sprünge in den zugewiesenen nummerischen Werten haben?
2. Wie lässt sich die Bezeichnung eines Aufzählungsmitglieds ermitteln?
3. Sie möchten gerne 8 verschiedene Werte in einer Aufzählung speichern und wollen nur den minimal erforderlichen Platz für die Aufzählung verbrauchen. Wie würden Sie diese Aufzählung erstellen?
4. Können Sie der Variablen eines Aufzählungstyps anstelle eines Mitglieds der Aufzählung auch einfach den nummerischen Wert zuweisen?

29.6.2 Die Antworten

Antwort 1

Das ist möglich, ohne eine Zuweisung von Werten weist der Compiler jedem neuen Element den um eins erhöhten Ordinalwert des Vorgängers zu, Sie können diese Folge aber jederzeit ändern.

Antwort 2

Über die Methode `GetNames` des Aufzählungstyps für alle Elemente oder `GetName` für ein einzelnes Element. Die Syntax dazu lautet `GetName(<typ>, <wert_oder_element>)`. Mehr dazu finden Sie im Kapitel über die Aufzählungstypen.

Antwort 3

Mit einem Basisdatentyp `byte` als z.B. `public enum Minimal : byte`.

Antwort 4

Dies wird aus Gründen der Fehlervermeidung vom C#-Compiler verhindert. In C# sind mit `enum` definierte Aufzählungen typsicher und damit ein eigener Datentyp. Eine Zuweisung des Wertes ist nur über eine explizite Typumwandlung (einen *type cast*) möglich.

29.7 Kapitel 10

29.7.1 Die Fragen

1. Was ist ein `char[,] [,,] eingabe` für eine Struktur?
2. Erstellen Sie ein kleines Programm, mit dem Sie ein Array mit einer benutzerdefinierten Anzahl von `int`-Werten einlesen und daraus die jeweils höchste und niedrigste Zahl ermitteln. Ziehen Sie auch die Online-Hilfe für die Klasse `System.Math` zu Rate.
3. Wie können Sie einer Struktur bei der Deklaration gleich für jedes Feld einen Initialwert zuweisen?

29.7.2 Die Antworten

Antwort 1

Eine zweidimensionale Matrix, deren Einträge jeweils aus einem Würfel bestehen, der Zeichenwerte speichert.

Antwort 2

Das hier gezeigte Beispiel ist nur eine von mehreren Möglichkeiten (wobei auf eine Fehlerprüfung der Zahleneingabe verzichtet wurde, da Sie als Entwicker sicher nur gültige Integerwerte eingeben würden, nicht wahr? `;-)`).

```
// lösung für aufgabe 2 von kapitel 10
using System;
using System.IO;
using System.Reflection;

namespace TypInfo {

   public class Test {

      public static void Main()
      {
         int anzahl;
         int[] zahlen;

         Console.Write("Anzahl Werte eingeben : ");
         anzahl = int.Parse(Console.ReadLine());

         zahlen = new int[anzahl];

         for(int i = 0; i < anzahl; i++)
         {
            Console.Write("Wert {0} eingeben : ", i);
            zahlen[i] = int.Parse(Console.ReadLine());
         }

         Array.Sort(zahlen);
         Console.WriteLine("Maximum = {0}, " +
                 "Minimum = {1}",
                 zahlen[zahlen.GetUpperBound(0)],
                 zahlen[0]);

         Console.ReadLine();  // warten nach Ende ...
      }
   }
}
```

Antwort 3

Dies ist über einen Konstruktor für eine Struktur möglich, mit der die dem Konstruktor übergebenen Werte direkt in den entsprechenden Feldern der Struktur abgelegt werden. Weitere Details dazu finden Sie unter ***Konstruktoren bei structs*** in Kapitel 10 ***Structs und Arrays***.

```
Geld guthaben = new Geld(1500.00, new String("DEM") );
```

29.8 Kapitel 11

29.8.1 Die Fragen

1. Worauf müssen Sie achten, wenn Sie die Methode `Equals` in einer Ihrer Klassen überschreiben?
2. Wie erhalten Sie Zugriff auf die Klassenmitglieder der Basisklasse? Erstellen Sie eine Klasse, deren Konstruktor einen `int` als Parameter erwartet und diesen an den Konstruktor der Basisklasse weitergibt.
3. Was sind abstrakte Klassen und wo liegt der Unterschied zu mit `sealed` gekennzeichneten Klassen?

29.8.2 Die Antworten

Antwort 1

Die Methode `Equals` wird für den Test auf Gleichheit durch die .NET-Laufzeitumgebung benutzt. Diese geht davon aus, dass zwei Objekte mit gleichen Inhalten auch für die Methode `GetHashCode` denselben Wert liefern. Wenn Sie `Equals` überladen, sollen Sie dies also auch für `GetHashCode` tun.

Antwort 2

Mit Hilfe des Schlüsselwortes `base` als Präfix können Sie auf Mitglieder der Basisklasse zugreifen, sofern diese mit den Modifikatoren `public`, `protected` oder (falls sich die abgeleitete Klasse im gleichen Assembly befindet) `internal` deklariert wurden.

```
base.BasisMethode();
```

Der Konstruktor einer Basisklasse wird beim Konstruktor der Klasse aufgerufen, indem dieser nach einem Doppelpunkt hinter dem abgeleiteten Konstruktor aufgeführt wird.

```
class Neu : Vorfahr
{
   public Neu(int wert) : base(wert)
   {
      // konstruktorcode
   }
}
```

Listing 29.1: Aufruf des Konstruktors einer Basisklasse

Antwort 3

Abstrakte Klassen enthalten nur eine Auflistung der Methoden ohne implementierenden Code sowie die Eigenschaften der Klasse. Damit ist eine abstrakte Klasse praktisch eine Art Schablone für abgeleitete Klassen. Diese implementieren dann die Methoden der abstrakten Klasse. In Klassenbibliotheken werden abstrakte Klassen meist als Ausgangspunkt für Klassen mit gleicher Funktionalität, aber unterschiedlicher Implementierung benutzt (z.B. Ausgabe auf Speicherstream, Diskette oder Drucker). Von einer abstrakten Klasse lässt sich daher auch keine Instanz erzeugen, sondern es muss eine Klasse abgeleitet werden.

Das Gegenteil ist eine ‚versiegelte' Klasse, die mit dem Schlüsselwort `sealed` erzeugt wurde. Von diesen Klassen können keine weiteren Klassen mehr abgeleitet werden. Auf diese Art und Weise können Sie als Entwickler beispielsweise Ihr geistiges Eigentum schützen, wenn Sie Klassenbibliotheken erstellen.

29.9 Kapitel 12

29.9.1 Die Fragen

1. Erstellen Sie eine Methode, die prüft, ob eine als Parameter übergebene Datei auch vorhanden ist und bei Nichtvorhandensein eine entsprechende Exception auslöst (benutzen Sie dazu die Klasse `System.IO.File`).
2. Was passiert, wenn Sie in Ihrer Anwendung eine Exception auslösen, diese aber nicht behandeln?
3. Unter welchen Umständen kann es sinnvoll sein, 'stille Exceptions' zu haben, also bestimmte Exceptions einfach ohne Reaktion abzufangen und wie würden Sie dies programmieren?
4. Wie fangen Sie eine Divison durch 0 bei einer `double`-Variablen ab?

29.9.2 Die Antworten

Antwort 1

```
void ProcessFile(string filename)
{
   if(File.Exists(filename))
```

```
   {
      // hier code
   }
   else
      throw new
      FileNotFoundException("Datei nicht vorhanden!",
                            filename);
}
```

Antwort 2

Die Exception wird in der Aufrufhierarchie der Anwendung solange nach oben weitergereicht, bis sie behandelt wird. Findet diese Behandlung in keiner Methode statt, landet der Fehler irgendwann bei der Standardbehandlungsroutine der Anwendung, die dann dem Endanwender in einer Dialogbox den Fehler meldet. Aus diesem Grund sollten Sie selbst ausgelöste Exceptions und Laufzeitfehler immer in Ihrem Code behandeln, um den Benutzer mit sinnvolleren als den Standardmeldungen auf die Ursache des Problems hinzuweisen.

Antwort 3

Wenn bestimmte Bedingungen das Auftreten einer Exception ermöglichen, diese aber sicher ignoriert werden kann, da die Anwendung nur an einem Teil der möglichen Daten interessiert ist (ein Beispiel ist das Lesen von Elementen aus einer XML-Datei, bei der nur alle Inhalte eines <title>-Elementes ausgewertet werden sollen. Besitzt das Element einen anderen Namen, kann auch nicht der Inhalt gelesen werden und es wird eine Exception ausgelöst – bei dieser Aufgabenstellung aber kann diese ignoriert werden).

```
while (rdr.Read())
{
   if(rdr.MoveToContent() == XmlNodeType.Element)
   {
      try
      {
         Console.WriteLine("Buchtitel = {0}",
         rdr.ReadElementString("title"));
      }
      catch(XmlException)
      {
          // ignorieren ....
      }
```

```
    }
}
rdr.Close();
```

Antwort 4

Eine Division durch 0 löst bei Gleitkommazahlen in der .NET-Laufzeitumgebung keine Fehlermeldung aus. Stattdessen erhält das Ergebnis der Berechnung den in der Struktur des Datentyps (`double`, `single`) definierten Wert »Not a number« (NaN). Diesen können Sie nach der Division über die Eigenschaft `IsNaN` testen: if(dblvar.IsNaN) { // teilung durch 0 }. Weitere Informationen finden Sie in der Dokumentation zu den Strukuren `double` und `single`.

29.10 Kapitel 13

29.10.1 Die Fragen

1. Entwerfen Sie eine Schnittstelle für eine Gerätesteuerung mit den Methoden An, Aus, Vorwärmen (Parameter double), den zwei Eigenschaften Werkstücke und Brennzeit sowie dem öffentlichen `double`-Feld Abkuehlzeit.
2. Wie lassen sich zwei Methoden gleichen Namens aus zwei verschiedenen Schnittstellen in einer Klasse implementieren?
3. Wo liegt der Unterscheid zwischen einer abstrakten Klasse und einer Schnittstellendefinition?
4. Angenommen, eine Klasse `abc` implementiert eine Schnittstelle `xyz`. Von `abc` leiten Sie nun eine neue Klasse ab. Muss diese neue Klasse ebenfalls wieder die Methoden der Schnittstellen implementieren?

29.10.2 Die Antworten

Antwort 1

Falls Sie sich beim Durchlesen der Frage gewundert hatten, dann haben Sie Recht: Fangfrage! Eine Schnittstellendefinition kann keine Felder enthalten, da eine Schnittstelle nur die Funktionalität, nicht aber die Implementierung beschreibt. Aus diesem Grund sind auch nur Eigenschaften zulässig. Ob diese dann implementierungsabhängig über Fel-

der oder berechnete Eigenschaften gelöst werden, ist von der funktionalen Definition der Schnittstelle unabhängig.

```
interface Steuerung
{
   void An();
   void Aus();

   void Vorwaermen(double temperatur);

   int Werkstuecke
   {
      get;
      set;
   }

   DateTime Brennzeit
   {
      get;
      set;
   }
}
```

Antwort 2

Über eine explizite Implementierung, bei der Sie jeder Methode aus einer Schnittstelle den Namen der Schnittstelle voranstellen. Dies hat außerdem noch den Vorteil, dass diese Methoden und Eigenschaften über eine normale Instanz der Klasse nicht sichtbar sind und daher auch nicht versehentlich aufgerufen werden können.

Antwort 3

Eine abstrake Klasse ist auf der Ebene der Implementierungsvererbung etwas Ähnliches wie eine Schnittstelle. Sie beschreibt ebenfalls nur die Funktionalität, aber die tatsächliche Implementierung muss in einer abgeleiteten Klasse geschehen. Abstrakte Klassen dienen meist als Startpunkt einer Vererbungshierarchie. Eine Klasse kann in C# immer nur von einer Basisklasse abstammen, dagegen kann eine Klasse mehr als eine Schnittstelle implementieren. Weitere Details finden Sie in Kapitel 13 ***Schnittstellen***.

Antwort 4

Nein, denn die Basisklasse `abc` implementiert die Schnittstelle und die abgeleitete Klasse erbt die gesamte Funktionalität der Basisklasse und damit auch die Implementierung der Schnittstellenmethoden.

29.11 Kapitel 14

29.11.1 Die Fragen

1. Erstellen Sie ein Programm, mit dem Sie in einem Verzeichnis alle Dateien mit den Erweiterungen *.bak* und *.tmp.* löschen können, wenn diese älter sind als zwei Wochen.
2. Lesen Sie im Ordner System32 von Windows alle Dateien mit der Erweiterung *.exe* aus, für die eine gleichnamige *.dll* oder *.hlp*-Datei existiert.
3. Lesen Sie die Datei *books.xml* ein und setzen Sie bei den `price`-Elementen ein Attribut `currency` ein, dessen Inhalt »EUR« lautet.

29.11.2 Die Antworten

Antwort 1

Quellcode aufgrund der Länge auf der Buch-CD enthalten.

Antwort 2

Quellcode aufgrund der Länge auf der Buch-CD enthalten.

Antwort 3

Quellcode aufgrund der Länge auf der Buch-CD enthalten.

29.12 Kapitel 15

29.12.1 Die Fragen

1. Erstellen Sie eine Filteranwendung, bei der eine Textdatei eingelesen wird und über einen Stream als ASCII-Datei wieder geschrieben wird und ersetzen Sie darin alle Zeichen, die nicht im ASCII-Standard vorliegen, durch % und den Zeichencode in hexadezimaler Darstellung.

2. Wozu dienen Verbatim-Strings in C#?
3. Wie können Sie feststellen, ob ein bestimmter String bereits im Speicherpool der .NET-Laufzeitumgebung vorhanden ist?

29.12.2 Die Antworten

Antwort 1

Quellcode aufgrund der Länge auf der Buch-CD enthalten.

Antwort 2

Verbatim-Strings werden durch ein vorangestelltes »@«-Zeichen gekennzeichnet und erlauben die bequeme Angabe von Pfadnamen, da innerhalb dieser Strings keine Behandlung von Steuerzeichen vorkommt. Aus diesem Grund reicht beispielsweise für Pfade die Angabe eines einzelnen Rückwärtsschrägstrichs (»\«).

Antwort 3

Über einen Aufruf der Methode `IsInterned` der Klasse String.

29.13 Kapitel 16

29.13.1 Die Fragen

1. Erstellen Sie eine Struktur oder Klasse für die Speicherung von Bewertungspunkten von Kollegschülern von 0 bis 15 und Konvertierungsroutinen von und zu double-Werten. Dabei entspricht der double 6 – (punkte / 3).
2. Warum müssen Sie möglicherweise auch bei einer impliziten Typumwandlung den gewünschten Typ explizit angeben, da Sie andernfalls einen Compilerfehler erhalten?

29.13.2 Die Antworten

Antwort 1

```
struct Kolleg
{
   public int Punkte;
```

```
   public Kolleg(int punkte)
   {
      this.Punkte = punkte;
   }

   public static implicit operator double(Kolleg k)
   {
      return 6 - (k.Punkte / 3);
   }

   public static explicit operator Kolleg(double d)
   {
      int p = (int)Math.Round((6 - d) * 3);
      if ((p>=0) && (p <=15))
         return new Kolleg(p);
      else
         throw new Exception("Nur 0-15 Punkte");
   }
}
```

Antwort 2

Falls bei einer Zuweisung oder einem Methodenaufruf mehr als eine gleichberechtigte Möglichkeit für eine implizite Konvertierung besteht, kann der Compiler nicht entscheiden, welche benutzt werden soll.

Ein Beispiel: Sie erstellen eine Klasse mit einer impliziten Konvertierung vom Typ der Klasse zu einem `string` und einer weiteren impliziten Konvertierung zu einem `int`. Anschließend rufen Sie die Methode `WriteLine` der Klasse `Console` mit einer Instanz dieser Klasse auf. `WriteLine` besitzt eine Überladung für `string`, allerdings auch eine Überladung für `long`, wofür in der .NET-Laufzeitumgebung eine implizite Konvertierung von `int` existiert. Damit kann der Compiler nicht entscheiden, welche implizite Konvertierung aufgerufen werden soll und verlangt eine explizite Typumwandlung.

29.14 Kapitel 17

29.14.1 Die Fragen

1. Auch hier wieder ein kleines Projekt: Erstellen Sie eine Windows Forms Anwendung, mit der über Schieberegler für eine Farbfläche der Rot-, Grün- und Blauanteil eingestellt werden kann. Dies soll für jeden Kanal nur im Vielfachen von 51 möglich sein und die kombinierte RGB-Angabe soll als Hexwert in der Form #rrggbb (wie in HTML-Seiten) in einer Textbox nach jeder Änderung angezeigt werden.

29.14.2 Die Antworten

Antwort 1

Quellcode aufgrund der Länge auf der Buch-CD enthalten.

30 Die Werkzeuge

30.1 Der Compiler

Sicherlich neben dem Editor das wichtigste Werkzeug für die Programmierung mit C#. Die Erläuterungen in diesem Buch beziehen sich auf Leser, die nicht mit Visual Studio.NET arbeiten, sondern entweder (tapfer, tapfer) über die Kommandozeile mit Stapeldateien und Build-Tools oder mit einer alternativen Entwicklungsumgebung wie SharpDevelop. Zu Visual Studio komme ich in einem der nächsten Abschnitte.

30.1.1 Grundlagen

Der C#-Compiler erstellt wie alle Compiler der .NET-Laufzeitumgebung IL-Code, welcher dann von einem der JIT-Compiler in ausführbaren Maschinencode umgesetzt wird (mehr dazu finden Sie in Kapitel 2 ***Das .NET-Framework***). Im einfachsten Fall (Ihr Programm benutzt nur den System-Namensraum) reicht dem Compiler die Angabe der Quellcodedatei aus:

```
csc helloworld.cs
```

Anschließend erstellt der Compiler ein Assembly `helloworld.exe`, welches den IL-Code und das Manifest enthält. Wenn Sie andere Assemblies in Ihrer Anwendung referenzieren, müssen diese beim Aufruf des Compilers angegeben werden (siehe dazu die Tabelle ***Eingabedateien*** bei den Compiler-Optionen).

30.1.2 Compiler-Optionen

Ausgabedateien

Option	Bedeutung
/out:<datei>	Ausgabedateiname (abgeleitet von der ersten Quelldatei, falls nicht anders angegeben).

Tabelle 30.1: Die Optionen für die Ausgabedateien

Option	Bedeutung
/target:exe (oder /t:exe)	Erstellen eines ausführbaren Konsolenprogramms. Dies ist die Standardoption, wenn keine andere Option für den Zieltyp angegeben wird.
/target:winexe (oder /t:winexe)	Erstellen eines ausführbaren Windows-Programms.
/target:library (oder /t:library)	Option zum Erstellen einer Bibliothek.
/target:module (oder /t:module)	Erstellt ein Modul, das zu einem anderen Assembly hinzugefügt werden kann.
/define:<symbolliste>	Definiert die angegebenen Symbole für die bedingte Kompilierung.
/doc:<datei>	Zu erstellende XML-Dokumentationsdatei.

Tabelle 30.1: Die Optionen für die Ausgabedateien (Forts.)

Eingabedateien

/recurse:<suchmaske>	Dateien im aktuellen Verzeichnis und allen Unterverzeichnissen entsprechend ihrer Suchmaske einschließen.
/reference:<dateiliste> (oder /r:<dateiliste>)	Verweis auf das angegebene Assembly.
/addmodule: <dateiliste>	Die angegebenen Module mit diesem Assembly verknüpfen.

Tabelle 30.2: Die drei Optionen für die Behandlung von Eingabedateien

Ressourcen

/win32res:<datei>	Gibt Win32-Ressourcendateien (.res) an.
/win32icon:<datei>	Dieses Symbol für die Ausgabedatei verwenden.
/resource:<resinfo> (oder /res:<resinfo>)	Bettet die angegebene Ressource ein.
/linkresource:<resinfo> (oder /linkres: <resinfo>)	Verknüpft die angegebene Ressource mit diesem Assembly.

Tabelle 30.3: Optionen zur Einbindung von Ressource-Dateien

Codeerstellung

/debug[+|-]	Ausgabe von Debuginformationen.
/debug:{full|pdbonly}	Debugtyp angeben (‚full' ist Standard und erlaubt das Anhängen eines Debuggers an ein ausgeführtes Programm).
/optimize[+|-] (oder /o)	Optimierung aktivieren.
/incremental[+|-] (oder /incr)	Inkrementelle Kompilierung aktivieren.

Tabelle 30.4: Optionen für die Codegenerierung

Fehler und Warnungen

/warnaserror[+|-]	Warnungen als Fehler behandeln.
/warn:<n> (oder /w)	Warnungsebene (0-4) bestimmen.
/nowarn:<Warnungsliste>	Bestimmte Warnungsmeldungen deaktivieren.

Tabelle 30.5: Optionen zur Steuerung der Warnungseinstellungen

.NET-Sprachoptionen

/checked[+|-]	Überlaufprüfung aktivieren.
/unsafe[+|-]	Nicht gesicherten Code zulassen.

Tabelle 30.6: Die Optionen für die Laufzeitüberprüfungen

Verschiedene Optionen

@<Datei>	Antwortdatei für weitere Optionen.
/help (oder /?)	Listet die Compiler-Optionen auf.
/nologo	Compiler-Copyrightmeldung unterdrücken.
/noconfig	CSC.RSP-Datei nicht automatisch einbeziehen

Tabelle 30.7: Weitere Optionen des C#-Compilers

/baseaddress:<Adresse>	Basisadresse der Bibliothek, sinnvoll zusammen mit der Option /target:library.
/bugreport:<Datei>	'Bugreport'-Datei erstellen.
/codepage:<n>	Bestimmt die Codepage für das Öffnen der Quelldateien.
/utf8output	Compiler-Ausgabemeldungen in UTF-8-Codierung.
/main:<Typ> (oder /m:)	Bestimmt den Typ, der den Einstiegspunkt enthält (alle anderen möglichen Einstiegspunkte werden ignoriert).
/fullpaths	Der Compiler erstellt einen vollständig qualifizierten Pfad.
/filealign:<n>	Ausrichtung für die Ausgabedateiabschnitte.
/nostdlib[+|-]	*mscorlib.dll* nicht referenzieren.
/lib:<dateiliste>	Zusätzliche Verzeichnisse für die Verweissuche.

Tabelle 30.7: Weitere Optionen des C#-Compilers (Forts.)

30.2 ILDASM und ILASM

30.2.1 ILDASM

Der ILDASM ist ein Disassembler für die Intermediate Language und gleichzeitig auch ein gutes Werkzeug, um die .NET-Laufzeitumgebung besser kennen zu lernen. Mit dem ILDASM können Sie nachsehen, was aus Ihrem C# Code geworden ist, den Aufbau eines Assemblys studieren und noch viele Dinge mehr.

Sie können den IL-Disassembler entweder mit der grafischen Benutzeroberfläche oder als Kommandozeilenprogramm starten. Wenn Sie keinen oder nur den Dateinamen des Assemblys angeben, startet ILDASM im GUI-Modus. Ich habe ein kleines Beispielprogramm erstellt, welches wir nun untersuchen können.

```
ildasm HelloSolarSystem.exe
```

Dieser Aufruf start ILDASM im GUI-Modus und zeigt die Daten des Assemblies in einer Explorer-ähnlichen Baumansicht an. Für den Start im erweiterten Modus verwenden Sie den Kommandozeilenschalter `/advanced`.

30.2.2 Die GUI-Oberfläche

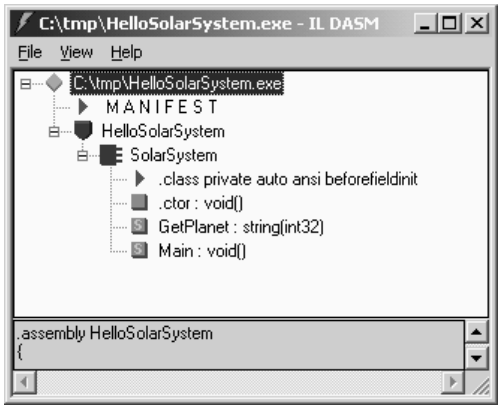


Abbildung 30.1: ILDASM in Aktion

Die Oberfläche besitzt zwei Anzeigeflächen: oben die Baumansicht mit dem Assemblydaten und unten eine kleine Textfläche mit Informationen über das Assembly. Im oberen Bereich können Sie mit der Maus oder der Tastatur die einzelnen Zweige öffnen und schließen und so untersuchen, aus welchen Typen mit welchen Methoden das Assembly aufgebaut ist. Sie erkennen im Beispiel eine Methode `GetPlanet`, welche einen `int32`-Parameter erwartet und einen `string` zurückliefert. Das kleine »S« im Kästchensymbol deutet darauf hin, dass es sich um eine statische Methode handelt. Weitere Informationen finden Sie in der Dokumentation zum .NET Framework-SDK und in der Online-Hilfe zu ILDASM.

Die Bedeutung der einzelnen Symbole können Sie der folgenden Abbildung entnehmen:

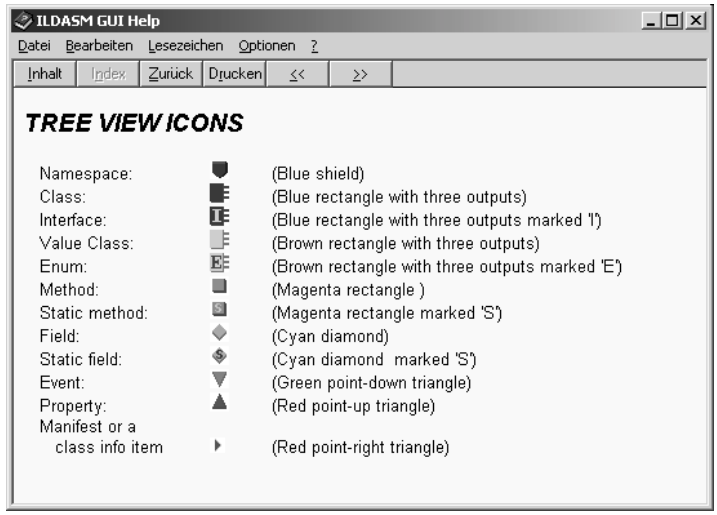


Abbildung 30.2: Die Symbole des ILDASM

30.2.3 Anzeige des IL-Codes

Wenn Sie auf eines der angezeigten Elemente doppelklicken, erhalten Sie eine Detailansicht. Beim Manifest sind es die Metadaten des Assemblys, für alle Methoden der entsprechende IL-Code. Die folgende Abbildung zeigt den IL-Code für die Methode `Main`:

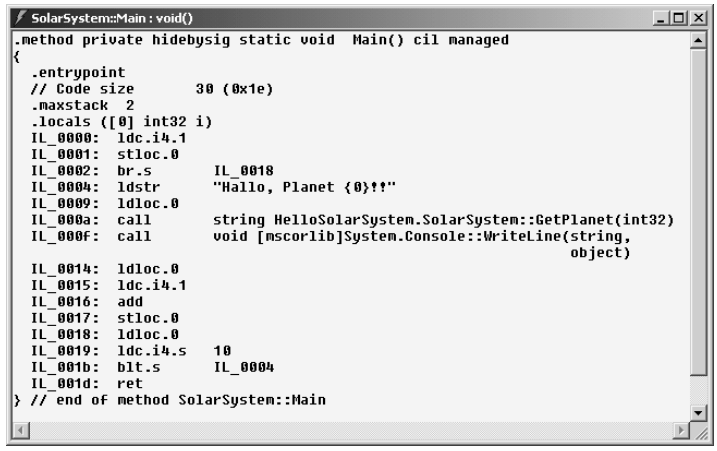


Abbildung 30.3: Der Code von Main in der Intermediate Language

30.3 ILASM

ILASM ist das Gegenstück und verwandelt Quelltexte in IL wieder in die entsprechenden binären Assemblies. Derzeit ist das Gespann aus diesen beiden Werkzeugen praktisch die einzige Möglichkeit, die Programmierung mit der IL zu erlernen, da die Dokumentation von Microsoft noch ziemlich spärlich ist. Um zu verstehen, was bei einem Programm überhaupt passiert, habe ich ein kleines Beispiel in C# geschrieben und mit dem ILDASM in IL-Code disassembliert. Anschließend habe ich damit herumgespielt und mit dem ILASM wieder übersetzen lassen und danach das Ergebnis analysiert (ich hätte nie gedacht, dass ich im dritten Jahrtausend noch einmal zu meinen Assembler-Wurzeln zurückkehre).

An dieser Stelle möchte ich auch gar nicht weiter auf die beiden Programme eingehen, da Sie weitergehende Kenntnisse dieser beiden Anwendungen nur bei sehr fortgeschrittenen Projekten mit C# oder einer direkten IL-Programmierung wirklich benötigen werden.

30.4 Debugger

Der .NET-Laufzeitdebugger dient der Fehlersuche und der Untersuchung von Programmen. Dieses Fehlersuchprogramm benutzt die in der .NET-Laufzeitumgebung enthaltene allgemeine Debug-Programmierschnittstelle für seine Funktionen.

Der komplette Quelltext des Debuggers wird sogar als Beispiel für die Nutzung dieser Schnittstelle mitgeliefert! Aber für uns C#-Entwickler hat die Sache einen Haken: Der Quellcode ist in managed C++ erstellt. Ich diskutiere daher nicht den Quellcode (was an dieser Stelle auch überhaupt nicht passen würde), sondern schildere nur in einer kurzen Übersicht den Aufruf und die entsprechenden Kommandos des Debuggers.

30.4.1 Aufruf des Debuggers

Sie starten den Debugger mit der folgenden Befehlszeile:

```
CORDBG [<program > [<args>]] [<CorDbg optional args>]
```

Listing 30.1: Kommandozeile für den Aufruf des Debuggers

Die optionalen Argumente sind Debuggerkommandos, welche mit einem Ausrufezeichen als Präfix versehen werden. Um beispielsweise die Anwendung `machwas.exe` zu debuggen und gleich beim Start des Debuggers einen Haltepunkt in der Methode `Main` zu setzen, würden Sie die folgende Kommandozeile benutzen:

```
cordbg machwas.exe !b Machwas.Testklasse::Main !g
```

30.4.2 Die Kommandos

Die folgenden Kommandos können Sie während einer Sitzung verwenden:

ap[pdomainenum]	Display appdomains/assemblies/modules in current process
a[ttach]	Attach to a running process
as[sociatesource]	Associate a source file with a breakpoint or stack frame
b[reak]	Set or display breakpoints
con[t]	Continue the current process
ca[tch]	Stop on exception, thread, and/or load events
dis[assemble]	Display native disassembled instructions
del[ete]	Remove one or more breakpoints
du[mp]	Dump the contents of memory
d[own]	Navigate down from the current stack frame pointer
de[tach]	Detach from the current process
ex[it]	Kill the current process and exit the debugger
f[unceval]	Function evaluation
g[o]	Continue the current process
h[elp]	Display debugger command descriptions
i[n]	Step into the next source line
ig[nore]	Ignore exception, thread, and/or load events
k[ill]	Kill the current process
l[ist]	Display loaded modules, classes, or global functions

Tabelle 30.8: Liste der Debugger-Kommandos

m[ode]	Display/modify various debugger modes
ns[ingle]	Step over the next native instruction
n[ext]	Step over the next source line
news[tr]	Create a new string via function evaluation
newobjnc	Create a new object via function evaluation, no constructor
newo[bj]	Create a new object via function evaluation
o[ut]	Step out of the current function
pro[cessenum]	Display all managed processes running on the system
p[rint]	Print variables (locals, args, statics, etc.)
pa[th]	Set or display the source file search path
q[uit]	Kill the current process and exit the debugger
regd[efault]	Change the JIT debugger
r[un]	Start a process for debugging
re[sume]	Resume a thread
rem[ove]	Remove one or more breakpoints
reg[isters]	Display CPU registers for current thread
ref[reshsource]	Reload a source file for display
<	Read and execute commands from a file
su[spend]	Suspend a thread
ss[ingle]	Step into the next native instruction
so	Step over the next source line
si, s[tep]	Step into the next source line
sh[ow]	Display source code lines
set	Modify the value of a variable (locals, statics, etc.)
setip	Set the next statement to a new line
Stop	Set or display breakpoints
t[hreads]	Set or display current threads
>	Write commands to a file
uw[here]	Display an unmanaged stack trace (Win32 mode only)

Tabelle 30.8: Liste der Debugger-Kommandos (Forts.)

ut[hreads]	Set or display unmanaged threads (Win32 mode only)
uc[lear]	Clear the current unmanaged exception (Win32 mode only)
u[p]	Navigate up from the current stack frame pointer
?	Display debugger command descriptions
wt	Track native instruction count and display call tree
wr[itememory]	Write memory to target process
w[here]	Display a stack trace for the current thread
x	Display symbols matching a given pattern

Tabelle 30.8: Liste der Debugger-Kommandos (Forts.)

30.5 WinCV

WinCV (Windows Class Viewer) ist ein Werkzeug zur Suche nach bestimmten Klassen in der .NET-Laufzeitumgebung. Nach dem Start können Sie nach dem Namensbestandteil einer Klasse suchen und WinCV liefert Ihnen eine Liste aller Klassen der .NET-Laufzeitumgebung, in denen der entsprechende Namensbestandteil vorkommt. In der Abbildung unten wurde beispielsweise nach »Listener« gesucht. Klicken Sie eine der Klassen im linken Feld an, werden Ihnen im rechten Feld die Mitglieder der Klasse angezeigt.

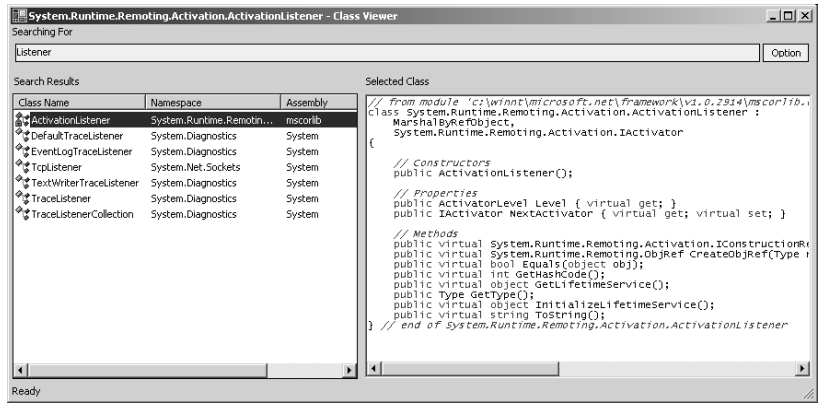


Abbildung 30.4: WinCV in Aktion

30.6 Visual Studio.NET

Sicherlich wird die .NET-Ausgabe von Visual Studio die meistbenutzte Umgebung für die Erstellung von Anwendungen für die .NET-Laufzeitumgebung werden. Die Behandlung aller Möglichkeiten der neuen Version von Visual Studio.NET erfordert einige Kapitel, wenn nicht ein eigenes Buch. Abhängig von Ihren Vorkenntnissen würde ich Sie mit den Ausführungen mehr oder weniger langweilen, wenn Sie eine andere Entwicklungsumgebung verwenden, daher werde ich nicht weiter auf VS.NET eingehen.

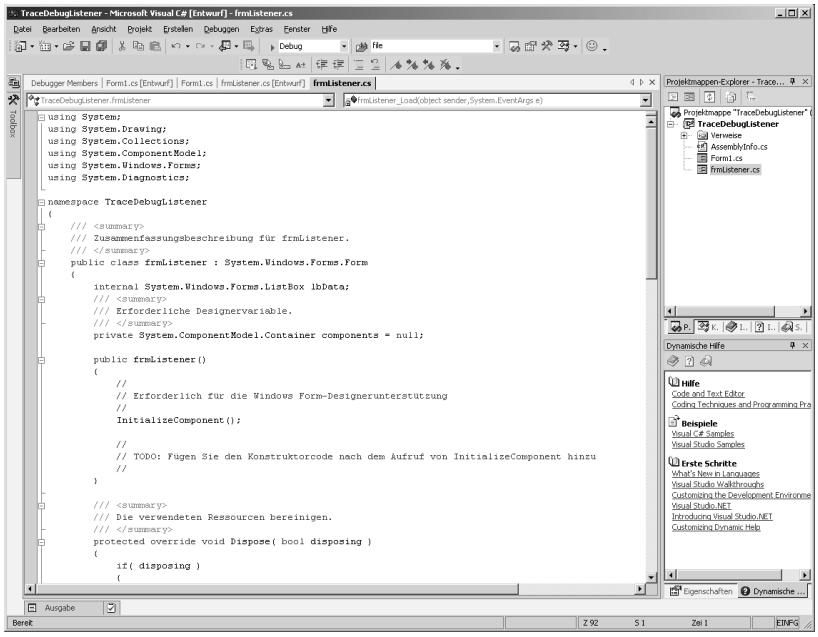


Abbildung 30.5: Visual Studio.NET

30.7 Zusammenfassung

Sie haben jetzt die wichtigsten Werkzeuge für die Programmierung mit C# kennen gelernt. Darüber hinaus existieren noch eine ganze Reihe mehr, vom Tool für die Verwaltung des globalen Assembly Caches bis hin zum Programm für die Verwaltung der Codezugriffssicherheit.

Werfen Sie einfach mal einen Blick in die Online-Dokumentation und den `bin`-Ordner des Framework SDK.

Noch einmal der Hinweis: Für dieses Buch benötigen Sie lediglich das .NET Framework SDK und einen Editor. Nur in den Kapiteln über die Erstellung von Windows Forms Anwendungen benutze ich Visual Studio.

31 Referenzteil

31.1 Datentypen

31.1.1 Werttypen

Datentyp	Wertebereich	Größe (phys. Größe)
bool	false | true	Logische Entscheidung (variable)
sbyte	-128 bis 127	8-bit integer mit Vorzeichen (1 Byte)
byte	0 bis 255	8-bit integer ohne Vorzeichen (1Byte)
char	U+0000 bis U+FFFF	Unicode 16-bit Zeichen (2 Byte)
short	-32,768 bis 32,767	16-bit integer mit Vorzeichen (2 Byte)
ushort	0 bis 65,535	16-bit integer ohne Vorzeichen (2 Byte)
int	-2,147,483,648 bis 2,147,483,647	32-bit integer mit Vorzeichen (4 Byte)
uint	0 bis 4,294,967,295	32-bit integer ohne Vorzeichen (4 Byte)

Tabelle 31.1: Die einfachen, ganzzahligen Werttypen in C#

Datentyp	Wertebereich	Größe (phys. Größe)
long	-9,223,372,036,854,775,808 bis 9,223,372,036,854,775,807	64-bit integer mit Vorzeichen (8 Byte)
ulong	0 bis 18,446,744,073,709,551,615	64-bit integer ohne Vorzeichen (8 Byte)

Tabelle 31.1: Die einfachen, ganzzahligen Werttypen in C# (Forts.)

Datentype	Wertebereich	Genauigkeit
float	±1.5 × 10-45 bis ±3.4 × 1038	7 Dezimalstellen
double	±5.0 × 10-324 bis ±1.7 × 10308	15-16 Dezimalstellen
decimal	1.0 × 10-28 bis 7.9 × 1028	28-29 Dezimalstellen

Tabelle 31.2: Die Gleitkomma-Werttypen in C#

Ebenfalls zu den Werttypen zählen alle Aufzählungstypen und Strukturen.

31.1.2 Implizite Typumwandlungen

Von	Nach
sbyte	short, int, long, float, double, or decimal
byte	short, ushort, int, uint, long, ulong, float, double, or decimal
short	int, long, float, double, or decimal
ushort	int, uint, long, ulong, float, double, or decimal

Tabelle 31.3: Liste der impliziten Typumwandlungen

Von	Nach
int	long, float, double, or decimal
uint	long, ulong, float, double, or decimal
long	float, double, or decimal
char	ushort, int, uint, long, ulong, float, double, or decimal
float	double
ulong	float, double, or decimal

Tabelle 31.3: Liste der impliziten Typumwandlungen (Forts.)

31.1.3 Referenztypen

class

```
[attribute]
[modifikator] class <bezeichner>
[:[basisklasse] [,interfaceliste] ]
{ <klassenrumpf> }[;]
```

- Attribute

  Zusätzliche deklarative Metadaten in Form von Attributen.

- Modifikator (optional)

  new, abstract, sealed

  public, private, protected, internal

- Basisklasse

  Name der Klasse, von welcher diese Klasse erbt. In C# gibt es nur eine einfache Vererbung, aber die mehrfache Implementierung von Schnittstellen.

- Interfaceliste

  Liste der von dieser Klasse implementierten Schnittstellen, jeweils getrennt durch Kommata.

- Klassenrumpf

  Deklaration der Mitglieder dieser Klasse

interface

```
[attribute]
[modifikator] interface <bezeichner>
[:interfaceliste]
{ <schnittstellendefinition> }[;]
```

- Attribute

  Zusätzliche deklarative Metadaten in Form von Attributen.

- Modifikator (optional)

  new

  public, private, protected, internal

- Interfaceliste

  Liste der Schnittstellen, welche diese Schnittstelle erbt

- Schnittstellendefinition

  Deklaration der Mitglieder dieser Schnittstelle. Eine Schnittstelle darf folgende Mitglieder enthalten: Methoden, Eigenschaften und Indexer

delegate

```
[attribute]
[modifikator] delegate datentyp <bezeichner>
( [formale-parameter] );
```

where:

- Attribute

  Zusätzliche deklarative Metadaten in Form von Attributen.

- Modifikator (optional)

  new

  public, private, protected, internal

- Datentyp

  Rückgabetyp, muss mit dem der Methode übereinstimmen, welche zugewiesen wird.

▼ Formale Parameter

Die Liste mit den Parametern für diesen Aufruf

object

Der Datentyp `object` entspricht dem .NET-Datentyp System.Object. Diesem Datentyp kann jeder andere Datentyp zugewiesen werden, da er die Basis aller Datentypen in C# darstellt.

string

Der C#-Datentyp `string` basiert auf dem .NET-datentyp `System.String` und enthält eine Reihe von Unicode-Zeichen. Alle Zeichen in einem `string` sind immer 16 Bit breit. Obwohl `string` ein Referenztyp ist, wurden die beiden Operatoren für den Vergleich auf Gleichheit (==) und Ungleichheit (!=) so überladen, dass hier die beiden **Inhalte** der Strings und nicht ihre Referenz verglichen wird. Damit können Sie für den Vergleich des Inhalts zweier Strings die gewohnten Operatoren verwenden.

31.2 Anweisungen

31.2.1 Auswahlanweisungen

if

```
if(<bedingung>)
   <anweisung1>
```

else

```
if(<bedingung>)
   <anweisung1>
else
   <anweisung2>
```

switch

```
switch(<ausdruck>)
{
   case <konstanenausdruck>:
      <anweisung>
      <verzweigungsanweisung>
 [ default:
      <anweisung>
      <verzweigungsanweisung> ]
}
```

31.2.2 Schleifenanweisungen

for

```
for(<initialisierung>; <bedingung>; <veränderung>)
   <schleifenkörper>
```

foreach

```
foreach(<elementtyp> <bezeichner> in <auflistung>)
   <schleifenkörper>
```

do

```
do
   <schleifenkörper>
while ( <bedingung> )
```

while

```
while ( <bedingung> )
   <schleifenkörper>
```

Innerhalb des Schleifenkörpers kann mit einer der Verzweigungsanweisungen der Schleifenkörper verlassen werden, wenn er mehr als eine Anweisung enthält.

31.2.3 Verzweigungsanweisungen

break

```
break;
```

continue

```
continue;
```

goto

```
goto <sprungmarke>;
goto case <konstantenausdruck>;
goto default;
```

Die beiden letzten Varianten sind nur innerhalb einer `switch`-Anweisung gültig.

return

```
return [ <ausdruck> ];
```

Beendet die Ausführung der Methode, in welcher diese Anweisung auftritt. Für alle Methoden vom Typ `void` wird der Wert von Ausdruck verworfen. Daher ist es für diese Methoden auch zulässig, die Anweisung `return` am Ende des Ausführungspfades wegzulassen.

31.2.4 Fehlerbehandlung

try / catch / finally

```
try
   <try-block>
catch <exception-declaration-1>
   <catch-block-1>
catch <exception-declaration-2>
   <catch-block-2>  . . .

try
   <try-block>
catch
   <catch-block>
```

throw

```
throw <ausdruck>;
throw;
```

Löst eine `Exception` aus. Die zweite Variante ist nur dann zulässig, wenn innerhalb eines `catch`-Blocks die Fehlerbedingung noch einmal ausgelöst werden soll (*re-throwing an exception*).

31.2.5 Codeausführung

checked / unchecked

```
checked <anweisungsblock> / unchecked <anweisungsblock>
checked ( <ausdruck> ) / unchecked <anweisungsblock>
```

Steuert die Überwachung des arithmetischen Überlaufs. Wird weder `checked` noch `unchecked` angegeben, gilt für Konstantenausdrücke immer `checked`, da diese zur Übersetzungszeit ausgewertet werden können. Für Ausdrücke, welche keinen konstanten Wert ergeben, hängt das Verhalten von der Laufzeitkonfiguration und den angegebenen Compiler-Optionen bei der Übersetzung des Quelltextes ab.

Geprüft werden alle Operationen mit Integer-Datentypen und den Operatoren für die vier Grundrechenarten, einschließlich der Prä- und Postinkrement-Operatoren.

lock

```
lock ( <ausdruck> ) <anweisung>
```

Legt für den Anweisungblock fest, dass nur jeweils ein Thread diesen Code ausführt (*critical section*). Der Wert für Ausdruck definiert das Objekt, für welches dieser Zugriff festgelegt werden soll. Dies ist entweder `this` als Referenz auf die aktuelle Instanz der Klasse zum Schutz von Instanzfeldern oder `typeof(<datentyp>)` zum Schutz der statischen Mitglieder des Datentyps.

31.2.6 Namensraumnutzung

using

```
using [ alias = ] namensraum_oder_klasse;
```

Legt fest, welcher Namensraum benutzt wird. Wird ein Alias festgelegt, ist auch die Angabe einer einzelnen Klasse möglich, für die dann der Alias festgelegt wird. Es können nur alle Typen aus einem Namensraum importiert werden, nicht einzelne Klassen.

namespace

```
namespace <name>
{
   <typdeklaration>
}
```

Definiert einen neuen Namensraum. Der Name darf auch Punkte enthalten. Die Grenze eines Namensraums ist nicht auf die Datei festgelegt, in welcher ein Namensraum definiert wird. Mehrere Dateien können Definitionen für diesen Namensraum enhalten, müssen die Definition des Namensraums aber in jeder Datei wiederholen. Innerhalb eines Namensraums sind folgende Deklarationen möglich:

- ein anderer Namensraum
- Klassen
- Schnittstellen

- Strukturen
- Aufzählungen
- Delegates

31.3 Präprozessorkommandos

31.3.1 #if, #else, elif, endif

Quellcode wird abhängig vom Wert des hinter `#if` angegebenen Ausdrucks in die Kompilierung einbezogen. Kann der Ausdruck hinter `#if` zu `true` evaluiert werden, wird der Teil im Quellcode bis zur nachfolgenden Anweisung in die Kompilierung einbezogen.

Mit den Anweisungen `#else` und `#elif` sind Verzweigungen möglich. Der Teil des Quellcodes zwischen einem `#else` und `#endif` wird in die Kompilierung einbezogen, wenn der bei `#if` vorhandene Ausdruck false ergibt.

Die Anweisung `#elif` erwartet selbst wieder einen Ausdruck, welcher dann geprüft wird, wenn der bei `#if` vorhandene Ausdruck false ergibt.

Sie können für den Ausdruck Klammern zur Festlegung der Priorität sowie die folgenden Operatoren benutzen:

- `==` (Gleichheit)
- `!=` (Ungleichheit)
- `&&` (logisch Und)
- `||` (logisch Oder)

Beispiel:

```
#define DEBUG
#define INTERNAL_BUILD
using System;
public class MyClass
{
   public static void Main()
   {
```

```
      #if (DEBUG && !INTERNAL_BUILD)
         Console.WriteLine("DEBUG is defined");
      #elif (!DEBUG && INTERNAL_BUILD)
         Console.WriteLine("INTERNAL_BUILD is defined");
      #elif (DEBUG && INTERNAL_BUILD)
         Console.WriteLine("DEBUG and INTERNAL_BUILD "
                           + "are defined");
      #else
         Console.WriteLine("DEBUG and INTERNAL_BUILD "
                           + "are not defined");
      #endif
   }
}
Ausgabe des Programms:
DEBUG and INTERNAL_BUILD are defined
```

31.3.2 #define, #undef

Mit `#define` definieren Sie innerhalb des Quelltextes ein Symbol für die bedingte Kompilierung. Die Anweisung `#undefine` löscht dieses Symbol wieder aus der Liste der definierten Symbole. Alternativ können Sie dies auch in Visual Studio.NET tun.

Beispiele:

```
#define DEBUG
#undefine INTERNAL_BUILD
```

31.3.3 #warning, #error

Diese Anweisungen geben bei der Kompilierung jeweils eine Warnung bzw. eine Fehlermeldung für die Position im Quelltext aus, an der sich die Anweisung befindet. Benutzen Sie `#warning` und `#error`, um beispielsweise sicherzustellen, dass Sie keine Kompilierung mit nicht kompatiblen Symbolen durchführen. Im Beispiel wird ein Fehler erzeugt, wenn das Symbol DEBUG noch definiert ist.

```
public class MyClass
{
   public static void Main()
   {
```

```
      #if DEBUG && RELEASE
      #error DEBUG und RELEASE gleichzeitig!!
      #endif
   }
}
```

31.3.4 #line

Benutzen Sie die Anweisung `#line`, um eine bestimmte Zeilennummer und einen Dateinamen auszugeben. Sinnvoll ist der Einsatz dieser Anweisung eigentlich nur bei der automatisierten Erstellung von Quellcode, so dass Sie bei der Fehlersuche die Zeilenummern des ursprünglichen Quelltextes verwenden können.

31.3.5 #region, #endregion

Diese Anweisungen dienen der Strukturierung des Quelltextes. Der hinter `#region` angegebene Name wird dann in Visual Sutdio.NET im Editor angezeigt, wenn dieser Bereich 'zugeklappt' wurde.

31.4 Namensräume

Nachfolgend finden Sie die Liste der derzeitigen Namensräume der .NET-Laufzeitumgebung.

31.4.1 MS-spezifische Namensräume

- Microsoft.CSharp
- Microsoft.JScript
- Microsoft.VisualBasic
- Microsoft.Win32

31.4.2 .NET-Namensräume

- System
- System.CodeDom
- System.CodeDom.Compiler

- System.Collections
- System.Collections.Specialized
- System.ComponentModel
- System.ComponentModel.Design
- System.ComponentModel.Design.Serialization
- System.Configuration
- System.Configuration.Assemblies
- System.Configuration.Install
- System.Data
- System.Data.Common
- System.Data.OleDb
- System.Data.SqlClient
- System.Data.SqlTypes
- System.Diagnostics
- System.Diagnostics.SymbolStore
- System.DirectoryServices
- System.Drawing
- System.Drawing.Design
- System.Drawing.Drawing2D
- System.Drawing.Imaging
- System.Drawing.Printing
- System.Drawing.Text
- System.EnterpriseServices
- System.EnterpriseServices.CompensatingResourceManager
- System.Globalization
- System.IO

- System.IO.IsolatedStorage
- System.Management
- System.Management.Instrumentation
- System.Messaging
- System.NET
- System.NET.Sockets
- System.Reflection
- System.Reflection.Emit
- System.Resources
- System.Runtime.CompilerServices
- System.Runtime.InteropServices
- System.Runtime.InteropServices.CustomMarshalers
- System.Runtime.InteropServices.Expando
- System.Runtime.Remoting
- System.Runtime.Remoting.Activation
- System.Runtime.Remoting.Channels
- System.Runtime.Remoting.Channels.Http
- System.Runtime.Remoting.Channels.Tcp
- System.Runtime.Remoting.Contexts
- System.Runtime.Remoting.Lifetime
- System.Runtime.Remoting.Messaging
- System.Runtime.Remoting.Metadata
- System.Runtime.Remoting.MetadataServices
- System.Runtime.Remoting.Proxies
- System.Runtime.Remoting.Services
- System.Runtime.Serialization

- System.Runtime.Serialization.Formatters
- System.Runtime.Serialization.Formatters.Binary
- System.Runtime.Serialization.Formatters.Soap
- System.Security
- System.Security.Cryptography
- System.Security.Cryptography.X509Certificates
- System.Security.Cryptography.Xml
- System.Security.Permissions
- System.Security.Policy
- System.Security.Principal
- System.ServiceProcess
- System.Text
- System.Text.RegularExpressions
- System.Threading
- System.Timers
- System.Web
- System.Web.Caching
- System.Web.Configuration
- System.Web.Security
- System.Web.Services
- System.Web.Services.Description
- System.Web.Services.Discovery
- System.Web.Services.Protocols
- System.Web.UI
- System.Web.UI.Design
- System.Web.UI.Design.WebControls

- System.Web.UI.HtmlControls
- System.Web.UI.WebControls
- System.Windows.Forms
- System.Windows.Forms.Design
- System.Xml
- System.Xml.Schema
- System.Xml.Serialization
- System.Xml.XPath
- System.Xml.Xsl

31.5 Vordefinierte Attribute

Die Attribute der folgenden Tabelle sind bereits innerhalb der .NET-Laufzeitumgebung definiert und können daher nicht mit eigener Bedeutung belegt werden.

AccessedThroughProperty	JustInTimeActivation
AmbientValue	LCIDConversion
ApplicationAccessControl	LicenseProvider
ApplicationActivation	ListBindable
ApplicationCrmEnabled	LoadBalancingSupported
ApplicationID	LoaderOptimization
ApplicationName	Localizable
ApplicationQueuing	ManagedName
AssemblyAlgorithmId	MarshalAs
AssemblyCompany	Match
AssemblyConfiguration	MergableProperty
AssemblyCopyright	MethodImpl
AssemblyCulture	MTAThread

Tabelle 31.4: Liste der vordefinierten Attribute

AssemblyDefaultAlias	MustRunInClientContext
AssemblyDelaySign	NeutralResourcesLanguage
AssemblyDescription	NonSerialized
AssemblyFileVersion	NotifyParentProperty
AssemblyFlags	ObjectPooling
AssemblyInformationalVersion	Obsolete
AssemblyKeyFile	OneWay
AssemblyKeyName	Optional
AssemblyProduct	Out
AssemblyTitle	ParamArray
AssemblyTrademark	ParenthesizePropertyName
AssemblyVersion	ParseChildren
AttributeUsage	PartialCaching
AutoComplete	PersistChildren
AutomationProxy	PersistenceMode
Bindable	PreserveSig
Browsable	PrimaryInteropAssembly
Category	PrivateComponent
ClassInterface	ProgId
CLSCompliant	PropertyTab
ComAliasName	ProvideProperty
ComConversionLoss	Proxy
ComImport	ReadOnly
CompilationRelaxations	RecommendedAsConfigurable
CompilerGlobalScope	RefreshProperties
ComponentAccessControl	RequiredAttribute
ComRegisterFunction	RootDesignerSerializer
ComSourceInterfaces	RunInstaller
COMTIIntrinsics	SatelliteContractVersion

Tabelle 31.4: Liste der vordefinierten Attribute (Forts.)

ComUnregisterFunction	SecureMethod
ComVisible	Security
Conditional	SecurityRole
ConstructionEnabled	Serializable
ConstructorNeedsTag	Soap
Context	SoapAttribute
ContextStatic	SoapDocumentMethod
ControlBuilder	SoapDocumentService
CustomConstant	SoapElement
DataBindingHandler	SoapEnum
Debuggable	SoapExtension
DebuggerHidden	SoapHeader
DebuggerStepThrough	SoapIgnore
DecimalConstant	SoapInclude
DefaultEvent	SoapRpcMethod
DefaultMember	SoapRpcService
DefaultProperty	SoapType
DefaultValue	STAThread
Description	StructLayout
	SuppressUnmanagedCodeSecurity
Designer	Synchronization
DesignerCategory	TagPrefix
DesignerSerializationVisibility	TemplateContainer
DesignerSerializer	ThreadStatic
DesignOnly	ToolboxBitmap
Discardable	ToolboxData
DispId	ToolboxItem
DllImport	ToolboxItemFilter
Editor	Transaction

Tabelle 31.4: Liste der vordefinierten Attribute (Forts.)

EditorBrowsable	TypeConverter
EventClass	TypeLibFunc
EventTrackingEnabled	TypeLibType
ExceptionClass	TypeLibVar
FieldOffset	UnverifiableCode
Flags	ValidationProperty
Guid	WebMethod
HttpMethod	WebService
IDispatchImpl	WebServiceBinding
IgnoreMember	XmlAnyAttribute
IISIntrinsics	XmlAnyElement
ImmutableObject	XmlArray
ImportedFromTypeLib	XmlArrayItem
In	XmlAttribute
IndexerName	XmlElement
Inheritance	XmlEnum
InstallerType	XmlIgnore
InstrumentationClass	XmlInclude
Instrumented	XmlRoot
InterfaceQueuing	XmlText
InterfaceType	XmlType

Tabelle 31.4: Liste der vordefinierten Attribute (Forts.)

Stichwortverzeichnis

.NET
  Econo-JIT 30
  Konfigurationsprogramm 453f.
.NET-Framework, Anwendungssicht 35

A

Abstraktion 97
ADO 34, 40, 46
ADO.NET 34, 40
Adress-Operator 458f.
Adresswert 461
Aktivierung 61, 383
AllowMultiple 395, 397f., 400
Anführungszeichen 79f., 265, 267
Anwendungsverzeichnisse 435
API 28, 31, 38, 47, 49, 61, 180, 246, 375, 461
  Aufrufe 38, 180, 461
  Deklaration 61
AppDomains 415
Append 277
Application 36, 201, 293, 297, 299, 303ff., 311, 415, 433
Application Domain 36, 415, 433, 447f.
Arrays 58, 75, 86, 90ff., 96, 100, 120, 144, 156, 164f., 169ff., 181, 188f., 191, 206, 208, 225, 241, 268ff., 317ff., 325f., 329, 332ff., 349, 369, 383f., 386f., 400, 402, 413, 424
  ArrayList-Klasse 325f.
  Bytearray 246
  Dimension 58, 171ff.
  Elemente 86, 90ff., 173
  Mitglieder 90
  Variable 170
  Zugriff 333
ASCII 243ff., 279, 460
  Datei 243, 279
  Encoding 244
  Standard 244, 279
  Werte 460
ASP.NET 43, 53, 72, 292
Assembler 246, 255
Assemblies 30ff., 35ff., 42, 49, 72, 110, 112, 347, 377, 388f., 396, 417, 431ff., 441f., 444ff., 453ff.
  Anwendungen 437
  Assembly Cache 437, 441
  AssemblyInfo 439f., 446
  AssemblyKeyFile 440f.
  Assembly-Zwischenspeicher 435
  Bibliotheken 434
  Module 433
  Policies 432
  Sicherheit 433, 435, 437f., 454
  Verteilung 448
Asserts 64, 336ff.
asynchron 98, 241
Attribute 60f., 63, 95, 101, 120, 154, 163, 180, 243, 253, 255f., 260f., 264, 335, 377, 389, 391ff., 411ff., 417, 419, 422, 425, 428, 438ff., 446
  Attributklassen 392, 395f., 400
  Conditional 335, 391ff.
  Programmierstil 389
  vordefinierte 392, 395ff.
Auflistungen 40, 72, 83, 144, 235, 304, 307, 317, 324f., 329f., 333f., 337, 341f., 345, 386, 432, 442
  Auflistungsklassen 144, 325f., 328, 333
  Auflistungsstrukturen 325
Aufrufhierarchie 208, 211
Aufrufkonvention 393
Aufrufverkettung 362
Aufzählungen 58, 64, 72, 83, 139, 159ff., 180, 225ff., 237, 393, 396f., 415
  Elemente 165, 168, 415
  Namen 164

Werte 161
Wertefolge 162
Aufzählungsdatentyp 163, 396
Aufzählungsobjekt 269, 329
Aufzählungstyp 58, 148, 159ff., 166ff.
Ausdruck 40, 57, 78, 84, 92, 109, 137f., 141ff., 148ff., 153, 156, 160, 267, 283, 286, 347, 404, 431, 455, 464
ausführbare 30, 54, 235, 293, 335, 411, 433
Ausgabeformate 273
Ausgabeformatierung 63
Ausgabeklassen 72
Ausgabelog 346
Ausgabeparameter 86, 91
Ausgabepuffer 337
Ausgabestrom 426
Ausgabetyp 294
Ausgabezweck 277
Ausgangsbasis 110, 240
Ausgangspunkt 377, 447
Ausnahmefehler 200, 204, 432
Ausrichtung 59, 271
Auswahldialog 295
AutoFlush 337, 340

B

Basisdatentypen 162, 167
Basisklassen 49f., 71, 125, 183ff., 191f., 203, 213f., 218, 224f., 241, 243, 261, 293
bedingte Kompilierung 391, 393
Begrenzungszeichen 79
Benutzeroberfläche 53
Berechtigungen 237, 431, 433
Bereichstrenner 273
Beta2 422
Betaversion 335
Beta-Versionen 469
Betriebssystem 28, 31, 238, 365
Betriebssysteme 237
betriebssystemunabhängig 28
Betriebssystemverzeichnis 237
Bezeichner 33, 57f., 72, 81, 85, 109, 128, 130, 136f., 139, 148, 170, 225, 283, 306, 388, 465
Bibliothek 29, 31, 53, 393, 434f., 437, 444, 449, 451
BigEndianUnicode 244
Binärdarstellung 246
Binärdatei 240, 246f., 414, 416, 424
Binärdaten 246
Binärkompatibilität 98
BinaryFormatter 73, 414, 416f., 420
BinaryReader 245f., 369
BinarySearch 174, 326
BinaryWriter 246, 369
Binder-Objekt 383
BindingFlags 383, 386
Bindung 383
BitArray 325f.
Bitflags 163
Bit-Shift 147
Block 27, 56, 87, 89, 92, 138, 140ff., 144, 200ff., 205f., 208, 211, 456
Blockkommentar 56
books.xml 249f., 253, 257ff., 264
bool-Ausdruck 64, 155
Boolean 59, 78, 380
bool-Flag 129
Boxing 50, 64, 75f., 92f., 107
break 59, 89, 114f., 139, 144ff., 156, 160, 165, 322f., 346
Brush 369, 387
BufferedStream 241
Bytecode 30

C

C 27ff., 33, 36ff., 43
C#
Compiler 28, 54, 62, 115, 125, 140, 186, 191, 293, 335, 392, 395f.
Compilerschalter 335, 391, 435, 456
Datentypen 283
Eigenschaften 128
Implementierung 159

C#-Programme, grundlegender Aufbau 55
C++ 29, 33, 38f., 46ff., 54, 59, 62f., 65, 78, 80, 86, 109, 112f., 117, 121, 124ff., 128, 132, 137, 141, 145ff., 151, 155f., 159, 169, 173, 175f., 184, 192, 200, 203, 216, 222f., 235, 265, 271, 291, 297, 307, 347f., 362, 367, 377, 404f., 455, 457, 461
callback 304, 352, 354ff., 360, 362
CanRead 247
CanSeek 247
CanWrite 247
Capacity 276, 328
Cappuccino 325
Carghill 184
case 59, 114f., 139, 145f., 160, 165, 322f.
cast 155, 197, 223, 281, 283
catch 200ff., 206, 208, 211
ChangeExtension 236
Check 36
checked 48, 139, 146, 459
checked-Block 459
checked-Modus 48
Close 124, 233, 242ff., 246, 257f., 260f., 263, 337, 342, 367, 369, 416, 418, 421, 426f.
CLR 28, 34, 240f., 245
CLS 33ff., 64
CLS-Spezifikationen 34
Codebasis 444, 448
Codebibliotheken 72
Codeblock 146
Codebrowser 163
Code-Editor 296, 302, 309
collection 29, 37, 48, 105, 124, 373
Compare 239, 267
Compileroptionen 435, 437
const 79f., 82f., 132, 139, 149, 278
constructor 33, 104, 178, 396, 423
continue 139, 144, 156
Convert-Klasse 286, 288, 412f., 419, 423, 467
Cookies 42, 413
Copy 33, 239, 268, 326
CreateObject 98
CreateText 239
CrossAppDomain 415
CrossMachine 415
CrossProcess 415, 420, 423
CRW 61
C-Syntax 55
CTS 32f.
CTS-Typbaum 32

D

Dateiinformationen 238
Dateimodus 243
Dateiobjekt 241
Dateipfad 235, 450
Dateistrom 416
Dateisystem 66
Dateisystemobjekte 234
Dateizugriff 237
Datenfelder 58, 96, 101, 107, 111f., 122f., 125ff., 133f., 169, 176ff., 186, 216f., 221, 304, 399f., 404, 412f., 417, 422, 427f., 432, 456, 464
Datentypen 32f., 40, 42, 48, 50, 55ff., 64f., 67f., 70ff., 75ff., 85, 92, 95, 99, 101f., 106, 113, 118, 120, 126, 130, 132, 137, 139, 145, 152, 156, 160, 165, 167, 169f., 172, 174ff., 189, 195, 203, 219, 246, 265, 277, 279, 281f., 284ff., 289, 317, 321, 323, 335, 341, 362, 379f., 382, 385f., 388, 403, 406, 408f., 414, 418, 426, 428, 454f., 459, 461, 471
bool 59, 61, 64, 78, 111, 115f., 122f., 129, 131, 139, 141f., 147, 149, 152, 155, 197, 239, 261, 296, 302, 309, 331, 379, 383, 393, 404, 438
byte 58, 77, 79, 110, 122f., 139, 150f., 153, 162f., 176, 226, 237f., 246f., 281ff., 385, 387, 394, 406, 460f., 463

char 59, 78f., 139, 145, 153, 181, 267, 282, 457, 460, 462
decimal 59, 78, 81, 125ff., 130, 133, 139, 177ff., 282
double 59, 78, 80, 82, 85, 103, 105, 118f., 129, 139, 170ff., 199, 202f., 207, 209, 212, 217ff., 224, 226ff., 232, 282, 290, 339, 378, 380, 412f., 416, 418f., 422, 460, 463f.
Ganzzahltypen 162
Gleitkommazahlen 59, 78, 129, 203
int 32f., 50, 54, 56, 58, 75ff., 86, 89f., 101ff., 106, 110, 114ff., 118ff., 122f., 133, 139, 141, 143ff., 148, 153, 155f., 162, 166f., 170ff., 181, 184, 189, 193, 196, 198, 200f., 203, 207, 209, 242, 246, 266, 271, 278, 281ff., 318ff., 322, 324, 330, 340, 353ff., 358, 360, 378, 394, 407, 412f., 419, 422, 425, 427f., 457f., 460ff., 467, 469, 471
long 58, 77f., 139, 162, 282, 287, 289
nummerische 177, 281
sbyte 58, 77, 79, 139, 162, 282
uint 33, 58, 61, 77, 86, 139, 147, 149, 153, 162, 282, 323, 379, 459, 462
ulong 59, 78, 139, 162, 179f., 282
DateTime 121, 133, 195, 278, 283, 425, 428, 438, 445
Debug-Klassen 42, 64, 335ff., 339ff., 345f., 391ff.
Default-Konstruktor 122, 188, 437
DefaultTraceListener 342
Definition 50, 56, 60, 63, 66, 71, 89, 100, 102, 109, 111ff., 116, 120f., 130, 144f., 160, 163f., 170, 177, 203, 219, 221, 225, 228, 252, 297, 324, 336, 354, 395f., 428
Deklarationen 402
Deklarationsblock 89
delegate 139, 305f., 347ff., 351ff., 360, 362, 397
Designregeln 108
Destruktor 87, 105ff., 121, 123f., 133, 179, 198, 206, 366f., 371ff.
Dispose 106, 124, 296, 302, 309, 367ff., 373f.
DLL 45f., 49, 294, 375, 391, 393, 431, 434
do-Schleifen 142f.
DrawMode 386
dynamische Erzeugung 313
dynamischer Aufruf 32, 60, 312, 314, 325, 383

E

Eiffel 51, 365
Eigenschaften 29, 37, 50, 57, 59, 85, 97, 99ff., 107f., 111, 115, 125ff., 153, 156, 174f., 177, 186, 192f., 214, 217, 219, 221, 228, 232, 235ff., 239f., 247, 250, 253, 256, 259f., 266f., 276, 294, 300, 304, 307, 313, 317, 324, 326, 328, 330, 333f., 337, 341, 346, 349f., 371, 379, 383, 386ff., 397ff., 424, 432, 447, 456
Nur-Lese-Eigenschaften 130
PropertyInfo 385ff.
else 130, 138ff., 154, 156, 210, 221, 230, 251, 261, 266, 276, 286, 288, 313, 315, 318f., 331, 339, 402, 445, 467
Empty 235
Encoder 244
EncoderParameter 370
Encodings 243
end 138, 165, 307
Endlosschleife 143
Entscheidung 108, 166
Equals 195ff., 268f., 327, 380
Ereignisse 228, 299f., 304f., 312, 347, 351ff., 360, 386, 397
Ereignisbehandlungsroutine 251, 304f., 313, 352, 354, 386

Ereignisobjekt 352, 354ff., 358, 360
Ereignisverarbeitung 359
Event 139, 306, 313, 347, 354, 358, 360, 362, 397
EventArgs 251, 304f., 307, 311ff., 345, 353, 385, 388
Event-Handler 305f., 309, 313f.
Exception 60, 71, 76, 114, 116, 152, 165, 199ff., 222f., 243, 259, 276, 286, 320, 338f., 369, 384, 432, 467, 469
AmbiguousMatchException 204
AppDomainUnloadedException 204
ArgumentException 204, 319, 332
ArgumentOutOfRangeException 286, 288, 318ff., 331, 467, 469
ArithmeticException 202, 204, 207, 210f.
ArrayTypeMismatchException 204
BadImageFormatException 204
Bezeichnung 204
Block 208
CountZeroException 209f.
DivideByZeroException 202, 204
FileNotFoundException 339
FormatException 204
InvalidCastException 204
InvalidComObjectException 204
InvalidOleVariantTypeException 204
InvalidOperationException 204
InvalidProgramException 204
Klasse 200, 208f.
OutOfMemoryException 204
StackOverflowException 204
WarningException 204
Exists 239, 338
Exit 304f., 307, 311
Exklusiv-ODER 152
explicit 139, 180, 286, 288, 467, 469, 471
Explorer 441f.
extern 101, 139, 393, 445
external 61

F

Fail 337, 342
false 78, 115, 139, 142, 149f., 175, 230, 257, 303, 311, 325, 330f., 338, 395, 397f., 400, 404, 440
Fehlerbehandlung 60, 71, 114, 197, 199f., 202f., 205, 211
Fehlersuche 42, 64, 141, 166, 178, 246, 335
Feld 101
FieldOffset 180
File 212, 233f., 237ff., 241f., 245, 338, 415
FileStream 241ff., 245f., 416f., 421
Finalizer 107, 124f., 195, 198, 367, 375
finally 60, 125, 139, 205f.
FixedSize 327
float 59, 78, 80, 82, 85f., 88, 117, 139, 184f., 187, 189, 193, 197, 202f., 274, 282
Flush 246, 263, 337, 342, 416, 421
Font 298, 306, 370, 387
for 87, 89, 120, 134, 139, 142ff., 266, 278, 320, 340, 358, 360, 412f., 419, 423
foreach 59, 139, 144, 155f., 174, 188, 190, 207, 210, 238, 253f., 324f., 329f., 333f., 380, 382, 386, 401, 427
Format 249, 253, 268, 270, 272, 274f., 285, 289, 340, 388, 407, 424, 433, 465
Formatierungen 63, 270, 273
Format-Strings 270, 274f.
Formatter-Klassen 414, 416, 422
Framework 436, 440
Freigabe 105
friend-Klassen 112

Funktionen 37, 56f., 61, 63, 73, 76, 81, 88, 90ff., 96ff., 102, 112f., 149, 180, 230, 240, 270, 304, 325, 336, 338, 393, 403, 438, 461, 463

G

GDI+ 42, 384
Geltungsbereich 85
Generationen 371f.
get 102f., 105, 127, 129ff., 133, 184, 190, 193, 197, 219, 225ff., 229, 317ff., 322, 324, 327, 331, 350, 378, 380, 398
GetHashCode 195, 198, 269, 327
GetType 195, 269, 327, 377ff.
Global Assembly Cache 435, 441ff., 450f., 453f.
GUID 224f.

H

Hashcodes 433
Heap 57, 75, 169, 181, 284, 459

I

I/O-Operationen 240
IDisposable 368f.
IEnumerable 329f., 333f.
IEnumerator 329f., 333f.
if 64, 78, 89, 126f., 130f., 133, 138ff., 145, 148ff., 152ff., 196, 198, 209, 221f., 230, 251, 253, 258ff., 266f., 276, 286, 288, 296, 302, 307, 309, 313f., 318f., 331, 338, 355, 358, 360, 401f., 420, 423, 445, 467
Indexer 217, 266f., 277, 317ff., 329f., 333f., 341, 404
Inherited 395, 397f.
Initialisierung 86, 90, 92f., 121, 135, 143, 172f., 176, 178, 295, 345
Installation 432f.
Instanz 103
  Objekt 105
Instanzenzaehlung 133
Instanziierung 103, 400
Interface 59, 111, 153f., 185, 213, 216, 347, 397
Intermediate Language 433, 436f.
  ildasm 436, 441, 445
internal 110, 112, 139
InvokeMember 383
ISerializable 394, 411f., 419, 422, 424

J

Java 30, 33, 45ff., 51f., 54f., 57ff., 62, 65, 69, 75, 86, 106, 111, 113, 117, 124, 137, 142, 146f., 166, 170f., 187, 193f., 203, 216f., 223, 465
JIT 30, 36

K

Kapselung 97
Klassen 27, 29, 31f., 37, 39f., 42f., 49f., 53, 56f., 59f., 62f., 65f., 68, 71f., 75, 81, 87f., 95, 100, 102f., 105, 109ff., 114, 116, 124, 128, 136f., 153, 156, 169, 172, 175f., 181, 183, 186, 191, 193f., 197, 200, 203, 206, 213f., 216f., 220, 223ff., 231, 233f., 237, 239, 241, 243ff., 251, 254f., 268, 283, 290, 293, 306, 325f., 328, 333ff., 339, 341f., 347, 349, 352, 361, 367ff., 377, 381f., 384, 388f., 392, 394, 397, 399, 403, 408, 418f., 422, 424, 428, 447, 461
  abstrakte 33, 110, 128, 139, 193f., 216
  Mitglieder 59, 83, 102, 109ff., 131f., 161ff., 171, 173, 187, 194, 213, 221, 231, 337, 379ff., 384f., 413, 415
  Nachfahren 111
Kohäsion 98
Kommandozeile 54, 453
Kommentarwebseiten 468ff.
Konfigurations-
  dateien 432, 447ff., 453f.

Konfigurationsprogramm 450, 453
Konsolenanwendung 257, 294
Konstanten 78, 81ff., 93, 132, 137, 203, 270, 285
Konstruktor 85, 104f., 107, 120ff., 134ff., 172, 176, 178, 186, 188, 198, 234, 243, 276, 284, 286, 300, 304, 313, 320, 344, 367, 372, 381, 396, 399, 412, 414, 422, 426
Konvertierungen 173f., 281, 283, 285, 287, 289, 467
Kopplung 98
Kurzschlussauswertung 149f.

L

Labels 388
Laufzeitfehler 197, 199f., 205f., 208, 222, 287, 375
Laufzeitsystem 163, 245
Layout-Angabe 180
LIFO-Struktur 325
Listboxen 293
Listener 337f., 341ff.
Listener-Konzept 341
Literale 76f., 79f., 82f., 86, 137
Literalkennung 273
Load-Ereignis 385
Log 342, 346
Lokalisierung 432, 435
loop 138, 340

M

Main 54, 68ff., 77, 82, 90, 103, 121, 132, 134, 150, 155, 199, 201, 205, 207f., 210, 220ff., 230, 238, 242, 258f., 266, 273f., 278, 284, 289, 293f., 296ff., 303, 311, 320, 333, 340, 350f., 359, 361, 368, 372, 379ff., 383, 400f., 408, 421, 436ff., 445, 448, 456, 458, 460, 462
Manifest 35ff., 433, 435ff., 441f., 445f.
Maschinencode 30, 36, 433
Mehrfachoperationen 234
Mehrfachvererbung 184
MemberInfo 380f.
MemberInfo-Klasse 380
MemoryStream 241
MessageBox 251, 256, 306f., 311
Metadaten 31, 35, 47, 60, 163, 377, 401, 432f., 454
Methoden 33, 39, 76, 85, 89, 92f., 96, 100ff., 106, 111, 113, 115ff., 124ff., 132, 139, 159, 163ff., 174f., 186f., 189, 191ff., 206, 208f., 212, 214ff., 221, 223, 227f., 235ff., 243, 245f., 248, 252ff., 259, 261, 265, 267, 269f., 274, 276f., 285f., 294, 296f., 299, 304ff., 309, 312, 314, 326, 329f., 334f., 337f., 342, 346ff., 353f., 356f., 362, 367ff., 371ff., 377ff., 385ff., 391f., 397, 399ff., 403f., 409, 412, 416ff., 422, 424, 426f., 437f., 456, 459, 466ff.
MMC 450, 454
Modellierung 97
Modifikatoren 78, 110, 135f.
Mono-Projekt 433

N

Namenskollision 68
Namenskonflikte 65, 69
Namensräume 39ff., 49, 65ff., 122, 156, 248, 251, 293f., 306, 325, 335, 377, 388, 399, 402, 417, 426, 436f., 445, 447, 471
  Alias 70f., 78, 380
  System-Namespace 77
native-code 36, 442
NetworkStream 241, 247
Neudefinition 403
NonSerialized 394, 413, 417
null 83, 91, 139, 142, 154f., 172, 177, 222f., 242, 262f., 296, 301f., 308, 313f., 355, 358, 360, 373, 383, 386ff.
null-Referenz 205, 222, 235, 243, 268, 355
null-Wert 223, 273

O

Object 195, 233, 248, 268, 343, 380, 418, 424
Objekt 96, 105
Objektinstanziierung 153
objektorientiert 95
One-Stop-Coding 55, 297, 391
OOP-Grundlagen 32, 38f., 50f., 60, 83, 95ff., 100, 106, 109, 113, 117, 128, 153, 175, 183, 215, 219, 365, 377, 417
  Banane-Szenario 214
  base-konstruktor 185, 190
  Finalisierung 367, 374
Operator, Präfix 148
Operatoren 56, 59, 80, 84, 93, 103, 109, 120, 137, 146ff., 160, 163, 172, 183, 198, 220, 223, 231, 267, 284f., 289, 306, 357, 362, 377ff., 400, 403ff., 408f., 457, 463, 467, 469, 471
  as 220, 223
  Bedingungsoperator 84, 147, 155
  Bit-Komplement 123
  Dekrement 405
  Dereferenzierung 457ff., 461
  Division 147
  Gleichheit 196
  is 153
  logische 147f.
  modulo 147
  Postfix 148
  Überladung 403, 406
  Vergleichsoperatoren 151
  Zuweisungsoperatoren 152
Ordinalwert 58, 161
out 64, 90ff., 115f., 139, 325
out-Parameter 115f.
OutputDebugString-API 342

P

PaintEventArgs 298, 369
Parameter 56, 61, 76, 79, 88, 90f., 99, 113, 115f., 118ff., 123, 126, 153, 165, 173f., 178, 188, 197, 201, 211f., 214, 221, 225, 232, 235, 238, 240, 243, 270f., 297, 304, 312, 318, 324, 352, 354, 371, 375, 381ff., 386f., 393f., 396ff., 403, 405, 409, 426, 471
Parameterinformationen 381
Parameterliste 113, 117, 304, 466
Parametersignatur 119, 122
params 119f., 139, 383
Path 233ff., 240f.
PathSeparator 236
Peek 245, 327
Pointer 49, 170
Polymorphismus 100
printf-Notation 63
private 122, 313
Programmablauf 374
Programmierung, objektorientierte 95
Projekt 47, 49, 66, 69, 237, 294f., 300, 316, 444, 469
Projektverweise 444
protected 111f., 122, 125, 129, 133, 136, 139, 166f., 184, 186, 189, 193, 196, 213, 218, 296, 298, 302, 304f., 307, 309, 394, 419
Prozedurvariablen 347
Prozessraum 30, 98, 415, 431, 434
Prozesswechsel 31
public 54, 56, 62, 66ff., 77, 82ff., 88, 90, 100ff., 110, 112, 118ff., 125ff., 129ff., 135, 139, 150, 154f., 159, 161ff., 167, 176ff., 183ff., 193ff., 199, 201, 205, 207, 209f., 217f., 220ff., 224, 226ff., 252, 262, 273f., 284ff., 288, 293, 295, 298, 300f., 308, 317ff., 322, 331, 343f., 347ff., 353ff., 358ff., 362, 368, 372, 374, 378ff., 383, 386, 391, 393ff., 398f., 404, 406, 412f., 419, 422, 424f., 428, 435, 437, 445, 456, 467

R

Random 358, 360
Rank 329
RankException 204

readonly 84f., 121, 133f., 139, 195, 353
Referenzparameter 86, 91
Referenztypen 33, 57, 75f., 90ff., 107, 116f., 169f., 172, 175, 177, 196, 265, 405, 461
Referenzvariable 57, 171
Referenzzählungsmodell 366
Reflection-Klassen 32, 42, 60, 68, 72, 120, 154, 377ff., 381, 383ff., 388f., 392, 399ff., 439, 447
ref-Parameter 57, 75, 90ff., 116f., 139
Registrierung 432
Registry 432
Release-Version 469
Reset 330f., 333
Ressourcenschutz 206
Ressourcenverwaltung 37
Richtlinien 447ff.
RTTI 60, 377
Runtime 34, 42, 46, 69, 414, 419, 439

S

SAX 248, 257
Schleifen 87, 89, 142ff., 275, 279
Schlüsseldatei 438, 440
Schlüsselpaar 435, 437f.
Schlüsseltoken 445
Schlüsselwörter 55, 81, 102, 138f., 233
Schnittstellen 32, 50, 53, 64, 97f., 101, 128, 184, 186, 194, 213, 215ff., 231f., 329f., 333f., 368f., 394, 397, 412, 422
  Deklaration 228
  Implementierung 227f., 231
  Typ 220
Schutzblock 60
sealed 59, 110f., 139, 194f.
Seiteneffekte 97, 333
Serialisierung 42, 46, 73, 146, 166, 394, 411ff., 422, 424ff.
  Konstruktoren 420, 424
set 102, 126, 324
Shell-Erweiterung 442
Sicherheitseinstellungen 449, 454
Sicherheitskontext 36
Sicherheitsüberprüfung 37, 234, 245
Sicherheitszone 206
Sichtbarkeitsmodifikatoren 176
Side-by-Side 431
Signierung 439ff.
Simula 95, 97
sizeof 139, 147, 153, 461
Smalltalk 50f., 75, 106, 365
sn.exe 435, 438, 440
SOAP 27, 414, 424
SOAP-Formatter 424
Sort 174, 328, 349
Sortierroutinen 349
Speicherverwaltung 29, 31, 48, 104ff., 124f., 169, 368, 371, 456
Sprachspezifikation 50, 465
stackalloc 139
Stackmaschine 30
Stacks 39f., 72
Standardausgabe 241, 272, 341f.
Standard-Formatierung 271
Standard-Konstruktor 225
Standardwerte 87, 91, 161, 397, 418
Startpunkt 36, 132
static 54, 61, 68ff., 77, 82f., 87, 90, 103, 120f., 132f., 135, 139, 150, 155, 166f., 172, 199, 201, 205, 207, 210, 220ff., 230, 238, 242, 246, 258ff., 262, 266, 273f., 278, 284ff., 288, 293, 296, 298, 303, 311, 320, 333, 337ff., 348ff., 359, 361, 368, 372, 379ff., 383, 393, 401, 404, 407, 417, 420, 427, 436, 445, 448, 456, 458, 460, 462, 467
Streams 238, 240f., 243, 245, 247, 252, 279, 370
  Eigenschaften 247
  Klassen 240, 247
  Konzept 240
  Objekte 241
  StreamReader 72, 234, 242f., 245f., 427
  StreamWriter 243f., 246, 426

String 38, 79f., 142, 164, 166f., 177f., 235, 238, 252f., 265ff., 274ff., 285, 289, 317, 320ff., 340, 342, 347, 366, 380, 393, 407, 425
Konvertierungen 268
Literale 273
StringBuilder 42, 72, 239, 275ff., 348f.
String-Klasse 266f., 269, 277
String-Literale 80, 265
Strings 63, 80
struct 40, 43, 53, 56, 64, 66, 81, 83, 96, 112, 123, 129, 138f., 175ff., 186, 225, 283f., 288, 385, 397, 403, 406, 408, 411, 461, 463
StructLayout 180
Suchpfade 454
switch 87, 114, 138, 140, 145f.
Synchronisierung 45, 146, 372

T

Textdateien 245
TextReader 247, 252, 370, 427
TextWriter 370, 426
this 88, 91, 122f., 139, 167, 196, 227, 229, 293, 296, 298, 302, 305f., 309, 313f., 317ff., 322, 331, 355, 358, 360, 372, 374, 396, 398, 400, 443
Thread 146, 365f., 375
throw 139, 210f., 276, 286, 288, 318f., 331, 339, 468
ToArray 328
ToCharArray 270
ToString 77, 106, 121, 134, 163f., 195f., 224, 230, 239, 267, 270, 274f., 277f., 328, 333, 342, 348f., 379f., 382, 393, 401, 406
Trace-Klassen 42, 64
TraceListener 342ff., 370
true 78, 115f., 139, 142, 149, 155f., 175, 203, 222, 247, 325, 330f., 340, 393f., 404
try 60, 125, 139, 199ff., 205ff., 210, 259
try/catch 60, 199, 206
Typabfrage 155
Typauswertung 153
Type-Klasse 383
Type-Objekt 165, 269, 377ff.
typeof 139, 147, 153, 164ff., 377ff., 383, 385, 400ff., 420, 423, 426f.
Typinformationen 32, 60, 153, 377, 382, 388, 424, 472
Typkonventierungen 455
Typkonzept 48, 281
Typ-Operatoren 147, 152, 154, 220, 222
Typsicherheit 58, 84, 115, 132, 173, 197

U

Überladen 117f., 198, 362, 381, 403ff., 408f.
Überlaufprüfung 146
Umwandlungsmethoden 285
Unboxing 50, 75f., 92
unchecked 84, 139, 146
Unicode 59, 78f., 81, 463
unsafe 36, 139, 455f., 458ff., 462
Using 77
Anweisung 251
Block 369

V

Variablendeklarationen 85, 120, 140
Variableninitialisierung 64
Variablenkategorien 86
VB.NET 81, 117, 175, 267
Verbatim-Strings 80, 235, 265, 279
Vererbung 50, 59, 96, 99, 110f., 123, 128, 162, 181, 183, 187f., 194, 213f., 216, 218, 231, 283, 397, 405
Hierarchie 188f., 192
Verkettung 153, 279, 357
Versionierung 50, 59, 111
Versionseinstellungen 37
Versionsinformationen 432, 439, 446f.
Versionskonflikte 449
Versionsnummer 431, 433, 445, 449

Versionsrichtlinie 450
Verteilung 431
Verweise 294, 444
virtual 128, 139, 189ff., 252, 456
  Virtuelle Methoden 189
Visual Basic.NET 49, 113
Visual C++ 46
void 57, 68ff., 90, 103f., 112, 116f., 120ff., 126, 132, 134, 139, 150, 154f., 185ff., 190, 194, 197, 199, 201, 205, 207, 210, 217ff., 226f., 229f., 238, 242, 246, 251ff., 258ff., 262, 266, 273f., 278, 284, 289, 293, 296, 298, 302, 304f., 307, 309, 313f., 320, 331, 338ff., 343, 345, 351, 353f., 356ff., 368, 372, 374, 378ff., 383, 385f., 388, 391, 401, 408, 412f., 417, 419, 423, 427, 436, 445, 448, 456, 458, 460ff.
vollqualifiziert 71
vordefiniert 210, 235, 392

W

warning 471
Warnungen 471
Web-Anwendungen 292
WebForms 43, 72, 292
Wertparameter 86, 90, 117, 324
Werttypen 33, 57, 64, 75f., 90ff., 116f., 148, 169, 175, 177f., 181, 284, 369, 405, 461, 463
while 139, 142f., 156, 242, 257ff.
Win32 61, 180, 214, 291, 306, 342, 375, 447
Win32-API 447
Windows Forms 43, 52f., 72, 106, 250, 257, 291ff., 295, 297, 307, 316, 341, 384, 387
  Anchors 250
  Application.Run 293, 297, 299, 303, 311
  Color 166f., 224, 227f., 230, 298, 384ff., 388
  Color.FromArgb 387
  Farbe 105, 384f., 387f., 406, 409
  Farbkanal 385, 406
  Farbnamen 384, 387
  Farbstruktur 387
  Farbwert 159
  Fenstereigenschaften 344
  Fenstererstellung 291
  Fensterfunktion 299
  Fensterhandle 414
  FontCollection 370
  Formulare 291f.
  Formularklasse 304
  GUI-Anwendung 43, 53, 72, 291ff., 306, 334, 341
  GUI-Programmierung 291
  MainMenu 306, 308
  Menü 307, 311ff., 324, 344, 450, 468
  Menüpunkte 307, 313, 324
  MenuItem 306ff., 314
  Steuerelemente 43, 53, 293, 295, 297, 299, 302, 304, 312
wrapper 61

X

XML 27, 34, 43, 46, 56, 248, 250ff., 260ff., 414, 424ff., 448, 453f., 465ff.
  Ausgabe 425
  Dokumentation 465ff.
  Serialisierung 424, 426, 428
  XmlDocument 250ff.
  XmlReader 249, 252, 257
  XmlSerializer 424, 426f.
  XmlTextReader 249, 257ff.
  XmlTextWriter 261ff.
  XmlWriter 256, 261
  XSL-Stylesheet 472
XML-Schema 109, 113, 193f., 304, 312, 425

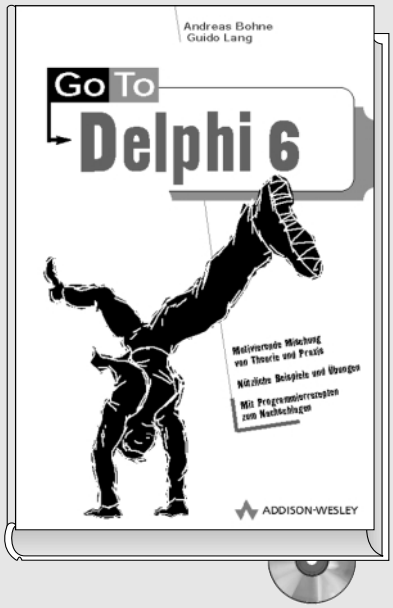
Z

Zeichenketten 79, 275, 277
Zeichenkonstante 267
Zeichenliterale 79
Zeigerarithmetik 455, 461f.
Zeigeroperationen 38

Zeiger-Operatoren 457
Zeit-Datentyp 283
Zugriffsmethoden 246
Zugriffsmodifikatoren 109, 111
Zugriffsrechte 455
Zugriffsschutz 176
Zwischencode 30
Zwischensprache 433

1856 fehlt!

THE SIGN OF EXCELLENCE



Go To Delphi 6

Andreas Bohne, Guido Lang

Anhand konkreter Beispiele und Übungen lernen Sie, wie Sie mit Delphi mit Grafik und Text umgehen, internetfähige Anwendungen programmieren, Datenbanken verwalten oder eigene Komponenten entwickeln. Themen wie Daten- und Speicherverwaltung, Fehlerbehandlung und die Verbindung von Delphi mit Windows werden dabei eingehend behandelt. Auch die neue Klassenbibliothek CLX, die Erstellung von Anwendungen für Linux sowie weitere Neuerungen in Delphi 6 werden berücksichtigt.

Go To

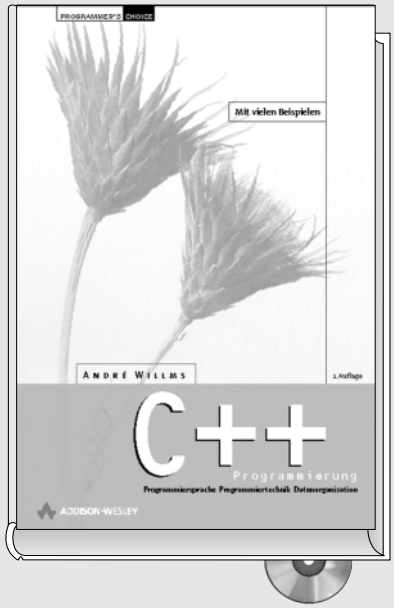
ca. 800 Seiten, 1 CD-ROM
€ 49,95 [D] / sFr 88,00
ISBN 3-8273-1774-6

www.addison-wesley.de

ADDISON-WESLEY

1386 fehlt!

THE SIGN OF EXCELLENCE



C++-Programmierung

2. Auflage

Programmiersprache, Programmiertechnik, Datenorganisation

André Willms

Die zweite, überarbeitete und erweiterte Auflage des Klassikers von André Willms führt Sie nach dem neuesten Standard in die C++-Programmierung ein. Die drei Themenbereiche des Buches – Sprache, Programmiertechnik und Datenorganisation – werden nicht in sich geschlossen vorgestellt, sondern Hand in Hand weiterentwickelt, sodass Sie am Ende nicht nur C++ beherrschen, sondern auch das nötige Wissen besitzen, um fast jedes Programmierproblem adäquat lösen zu können.

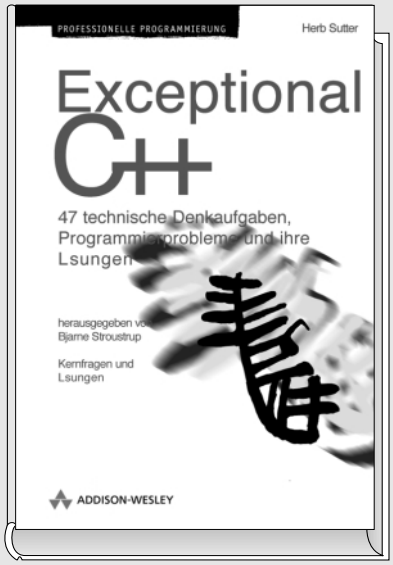
Programmer's Choice

ca. 450 Seiten, 1 CD-ROM
€ 34,95 [D] / sFr 63,00
ISBN 3-8273-1627-8

www.addison-wesley.de

ADDISON-WESLEY

THE SIGN OF EXCELLENCE



Exceptional C++

47 technische Denkaufgaben, Programmierprobleme und ihre Lösungen

Herb Sutter

Lösen Sie gerne anspruchsvolle C++-Probleme und begeistert Sie robuster und leicht erweiterbarer Code? Dann investieren Sie ein wenig Zeit und stellen Sie sich einigen kniffeligen C++-Entwurfs- und Programmierproblemen. Die Denkaufgaben in Exceptional C++ sind nicht nur unterhaltsam, sondern Sie werden Ihre C++-Fertigkeiten schärfen, um aus Ihnen einen noch besseren C++-Programmierer zu machen.

Professionelle Programmierung

272 Seiten
€ 34,95 [D] / sFr 63,00
ISBN 3-8273-1711-8

www.addison-wesley.de

ADDISON-WESLEY