Galileo Computing

Visual C# 2012

Das umfassende Handbuch: Spracheinführung, Objektorientierung, Programmiertechniken

Bearbeitet von
Andreas Kühnel

erweitert 2012. Buch. ca. 1402 S.
ISBN 978 3 8362 1997 6
Format (B x L): 19 x 24 cm

Weitere Fachgebiete > EDV, Informatik > Programmiersprachen: Methoden > Microsoft Programmierung

schnell und portofrei erhältlich bei

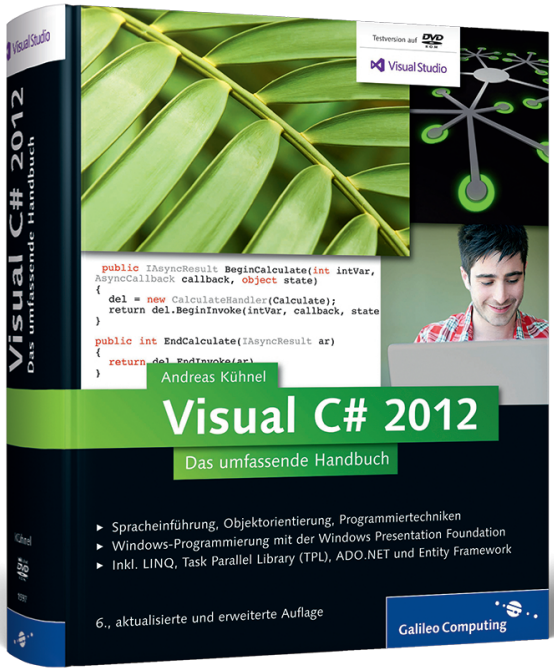
beck-shop.de
DIE FACHBUCHHANDLUNG

Die Online-Fachbuchhandlung beck-shop.de ist spezialisiert auf Fachbücher, insbesondere Recht, Steuern und Wirtschaft. Im Sortiment finden Sie alle Medien (Bücher, Zeitschriften, CDs, eBooks, etc.) aller Verlage. Ergänzt wird das Programm durch Services wie Neuerscheinungsdienst oder Zusammenstellungen von Büchern zu Sonderpreisen. Der Shop führt mehr als 8 Millionen Produkte.

Andreas Kühnel

Visual C# 2012

Das umfassende Handbuch



Galileo Press

Auf einen Blick

1 Allgemeine Einführung in .NET 33
2 Grundlagen der Sprache C# 57
3 Das Klassendesign 133
4 Vererbung, Polymorphie und Interfaces 205
5 Delegates und Ereignisse 259
6 Strukturen und Enumerationen 289
7 Fehlerbehandlung und Debugging 299
8 Auflistungsklassen (Collections) 333
9 Generics – Generische Datentypen 359
10 Weitere C#-Sprachfeatures 385
11 LINQ 435
12 Arbeiten mit Dateien und Streams 467
13 Binäre Serialisierung 511
14 XML 519
15 Multithreading und die Task Parallel Library (TPL) 613
16 Einige wichtige .NET-Klassen 665
17 Projektmanagement und Visual Studio 2012 705
18 Einführung in die WPF und XAML 759
19 WPF-Layout-Container 789
20 Fenster in der WPF 809
21 WPF-Steuerelemente 833
22 Elementbindungen 921
23 Konzepte von WPF 935
24 Datenbindung 975
25 Weitere Möglichkeiten der Datenbindung 1005
26 Dependency Properties 1041
27 Ereignisse in der WPF 1057
28 WPF-Commands 1077
29 Benutzerdefinierte Controls 1097
30 2D-Grafik 1107
31 ADO.NET – Verbindungsorientierte Objekte 1127
32 ADO.NET – Das Command-Objekt 1149
33 ADO.NET – Der SqlDataAdapter 1177
34 ADO.NET – Daten im lokalen Speicher 1191
35 ADO.NET – Aktualisieren der Datenbank 1231
36 Stark typisierte DataSets 1251
37 Einführung in das ADO.NET Entity Framework 1277
38 Datenabfragen des Entity Data Models (EDM) 1301
39 Entitätsaktualisierung und Zustandsverwaltung 1335
40 Konflikte behandeln 1363
41 Plain Old CLR Objects (POCOs) 1371

Inhalt

Vorwort zur 6. Auflage 31

1 Allgemeine Einführung in .NET 33

1.1 Warum .NET? 33
1.1.1 Ein paar Worte zu diesem Buch 35
1.1.2 Die Beispielprogramme 37
1.2 .NET unter die Lupe genommen 38
1.2.1 Das Entwicklerdilemma 38
1.2.2 .NET – Ein paar allgemeine Eigenschaften 39
1.2.3 Das Sprachenkonzept 40
1.2.4 Die »Common Language Specification« (CLS) 42
1.2.5 Das »Common Type System« (CTS) 43
1.2.6 Das .NET Framework 44
1.2.7 Die »Common Language Runtime« (CLR) 45
1.2.8 Die .NET-Klassenbibliothek 45
1.2.9 Das Konzept der Namespaces 47
1.3 Assemblys 48
1.3.1 Die Metadaten 49
1.3.2 Das Manifest 50
1.4 Die Entwicklungsumgebung 50
1.4.1 Editionen von Visual Studio 2012 50
1.4.2 Hard- und Softwareanforderungen 51
1.4.3 Die Installation 51
1.4.4 Die Entwicklungsumgebung von Visual Studio 2012 52

2 Grundlagen der Sprache C# 57

2.1 Konsolenanwendungen 57
2.1.1 Allgemeine Anmerkungen 57
2.1.2 Ein erstes Konsolenprogramm 57
2.2 Grundlagen der C#-Syntax 60
2.2.1 Kennzeichnen, dass eine Anweisung abgeschlossen ist 60
2.2.2 Anweisungs- und Gliederungsblöcke 61
2.2.3 Kommentare 62
2.2.4 Die Groß- und Kleinschreibung 63
2.2.5 Die Struktur einer Konsolenanwendung 64

2.3 Variablen und Datentypen ... 66
2.3.1 Variablendeklaration ... 66
2.3.2 Der Variablenbezeichner ... 67
2.3.3 Der Zugriff auf eine Variable ... 68
2.3.4 Ein- und Ausgabemethoden der Klasse »Console« ... 68
2.3.5 Die einfachen Datentypen ... 74
2.3.6 Typkonvertierung ... 80
2.4 Operatoren ... 87
2.4.1 Arithmetische Operatoren ... 88
2.4.2 Vergleichsoperatoren ... 91
2.4.3 Logische Operatoren ... 91
2.4.4 Bitweise Operatoren ... 95
2.4.5 Zuweisungsoperatoren ... 98
2.4.6 Stringverkettung ... 98
2.4.7 Sonstige Operatoren ... 99
2.4.8 Operator-Vorrangregeln ... 99
2.5 Datenfelder (Arrays) ... 100
2.5.1 Die Deklaration und Initialisierung eines Arrays ... 100
2.5.2 Der Zugriff auf die Array-Elemente ... 102
2.5.3 Mehrdimensionale Arrays ... 103
2.5.4 Festlegen der Array-Größe zur Laufzeit ... 104
2.5.5 Bestimmung der Array-Obergrenze ... 105
2.5.6 Die Gesamtanzahl der Array-Elemente ... 106
2.5.7 Verzweigte Arrays ... 106
2.6 Kontrollstrukturen ... 108
2.6.1 Die »if«-Anweisung ... 108
2.6.2 Das »switch«-Statement ... 113
2.7 Programmschleifen ... 117
2.7.1 Die »for«-Schleife ... 118
2.7.2 Die »foreach«-Schleife ... 128
2.7.3 Die »do«- und die »while«-Schleife ... 129

3 Das Klassendesign ... 133

3.1 Einführung in die Objektorientierung ... 133
3.2 Die Klassendefinition ... 136
3.2.1 Klassen im Visual Studio anlegen ... 136
3.2.2 Das Projekt »GeometricObjectsSolution« ... 136
3.2.3 Die Deklaration von Objektvariablen ... 139
3.2.4 Zugriffsmodifizierer einer Klasse ... 140

3.2.5 Splitten einer Klassendefinition mit »partial« ... 140
3.2.6 Arbeiten mit Objektreferenzen ... 141
3.3 Referenz- und Wertetypen ... 143
3.3.1 Werte- und Referenztypen nutzen ... 144
3.4 Die Eigenschaften eines Objekts ... 145
3.4.1 Öffentliche Felder ... 145
3.4.2 Datenkapselung mit Eigenschaftsmethoden sicherstellen ... 147
3.4.3 Die Ergänzung der Klasse »Circle« ... 149
3.4.4 Lese- und schreibgeschützte Eigenschaften ... 149
3.4.5 Sichtbarkeit der Accessoren »get« und »set« ... 150
3.4.6 Unterstützung von Visual Studio 2012 ... 151
3.4.7 Automatisch implementierte Eigenschaften ... 152
3.5 Methoden eines Objekts ... 153
3.5.1 Methoden mit Rückgabewert ... 153
3.5.2 Methoden ohne Rückgabewert ... 157
3.5.3 Methoden mit Parameterliste ... 157
3.5.4 Methodenüberladung ... 159
3.5.5 Variablen innerhalb einer Methode (lokale Variablen) ... 162
3.5.6 Referenz- und Wertparameter ... 163
3.5.7 Besondere Aspekte einer Parameterliste ... 168
3.5.8 Zugriff auf private Daten ... 173
3.5.9 Die Trennung von Daten und Code ... 174
3.5.10 Namenskonflikte mit »this« lösen ... 175
3.5.11 Methode oder Eigenschaft? ... 176
3.5.12 Umbenennen von Methoden und Eigenschaften ... 177
3.6 Konstruktoren ... 178
3.6.1 Konstruktoren bereitstellen ... 179
3.6.2 Die Konstruktoraufrufe ... 180
3.6.3 Definition von Konstruktoren ... 180
3.6.4 »public«- und »internal«-Konstruktoren ... 181
3.6.5 »private«-Konstruktoren ... 181
3.6.6 Konstruktorenaufrufe umleiten ... 182
3.6.7 Vereinfachte Objektinitialisierung ... 183
3.7 Der Destruktor ... 184
3.8 Konstanten in einer Klasse ... 185
3.8.1 Konstanten mit dem Schlüsselwort »const« ... 185
3.8.2 Schreibgeschützte Felder mit »readonly« ... 186
3.9 Statische Klassenkomponenten ... 186
3.9.1 Statische Eigenschaften ... 186
3.9.2 Statische Methoden ... 189

3.9.3 Statische Klasseninitialisierer ... 190
3.9.4 Statische Klassen ... 191
3.9.5 Statische Klasse oder Singleton-Pattern? ... 192
3.10 Namensräume (Namespaces) ... 193
3.10.1 Zugriff auf Namespaces ... 194
3.10.2 Die »using«-Direktive ... 196
3.10.3 Globaler Namespace ... 197
3.10.4 Vermeiden von Mehrdeutigkeiten ... 197
3.10.5 Namespaces festlegen ... 198
3.10.6 Der »::«-Operator ... 200
3.10.7 Unterstützung von Visual Studio 2012 bei den Namespaces ... 201
3.11 Stand der Klasse »Circle« ... 203

4 Vererbung, Polymorphie und Interfaces 205

4.1 Die Vererbung ... 205
4.1.1 Basisklassen und abgeleitete Klassen ... 205
4.1.2 Die Ableitung einer Klasse ... 206
4.1.3 Klassen, die nicht abgeleitet werden können ... 208
4.1.4 Konstruktoren in abgeleiteten Klassen ... 208
4.1.5 Der Zugriffsmodifizierer »protected« ... 209
4.1.6 Die Konstruktorverkettung in der Vererbung ... 210
4.2 Der Problemfall geerbter Methoden ... 214
4.2.1 Geerbte Methoden mit »new« verdecken ... 216
4.2.2 Abstrakte Methoden ... 217
4.2.3 Virtuelle Methoden ... 219
4.3 Typumwandlung und Typuntersuchung von Objektvariablen ... 220
4.3.1 Die implizite Typumwandlung von Objektreferenzen ... 220
4.3.2 Die explizite Typumwandlung von Objektreferenzen ... 222
4.3.3 Typuntersuchung mit dem »is«-Operator ... 223
4.3.4 Typumwandlung mit dem »as«-Operator ... 224
4.4 Polymorphie ... 224
4.4.1 Die »klassische« Methodenimplementierung ... 225
4.4.2 Abstrakte Methoden ... 226
4.4.3 Virtuelle Methoden ... 227
4.5 Weitere Gesichtspunkte der Vererbung ... 230
4.5.1 Versiegelte Methoden ... 230
4.5.2 Überladen einer Basisklassenmethode ... 231
4.5.3 Statische Member und Vererbung ... 232
4.5.4 Geerbte Methoden ausblenden? ... 232

4.6 Das Projekt »GeometricObjectsSolution« ergänzen ... 233
4.6.1 Die Klasse »GeometricObject« ... 233
4.7 Eingebettete Klassen (Nested Classes) ... 237
4.8 Interfaces (Schnittstellen) ... 237
4.8.1 Einführung in die Schnittstellen ... 237
4.8.2 Die Schnittstellendefinition ... 238
4.8.3 Die Schnittstellenimplementierung ... 239
4.8.4 Die Interpretation der Schnittstellen ... 244
4.8.5 Änderungen am Projekt »GeometricObjects« ... 249
4.9 Das Zerstören von Objekten – der »Garbage Collector« ... 251
4.9.1 Die Arbeitsweise des Garbage Collectors ... 251
4.9.2 Expliziter Aufruf des Garbage Collectors ... 252
4.9.3 Der Destruktor ... 253
4.9.4 Die »IDisposable«-Schnittstelle ... 254
4.9.5 Die Ergänzungen in den Klassen »Circle« und »Rectangle« ... 257

5 Delegates und Ereignisse 259

5.1 Delegates ... 259
5.1.1 Einführung in das Prinzip der Delegates ... 259
5.1.2 Verwendung von Delegates ... 263
5.1.3 Vereinfachter Delegatenaufruf ... 263
5.1.4 Multicast-Delegates ... 264
5.1.5 Anonyme Methoden ... 266
5.1.6 Kovarianz und Kontravarianz mit Delegaten ... 268
5.2 Ereignisse eines Objekts ... 270
5.2.1 Ereignisse bereitstellen ... 271
5.2.2 Die Reaktion auf ein ausgelöstes Ereignis ... 273
5.2.3 Allgemeine Betrachtungen der Ereignishandler-Registrierung ... 275
5.2.4 Wenn der Ereignisempfänger ein Ereignis nicht behandelt ... 276
5.2.5 Ereignisse mit Übergabeparameter ... 277
5.2.6 Ereignisse in der Vererbung ... 281
5.2.7 Hinter die Kulissen des Schlüsselworts »event« geblickt ... 282
5.2.8 Die Schnittstelle »INotifyPropertyChanged« ... 284
5.3 Änderungen im Projekt »GeometricObjects« ... 285
5.3.1 Überarbeitung des Events »InvalidMeasure« ... 285
5.3.2 Weitere Ereignisse ... 286

6 Strukturen und Enumerationen 289

6.1 Strukturen – eine Sonderform der Klassen ... 289
6.1.1 Die Definition einer Struktur ... 289
6.1.2 Initialisieren einer Strukturvariablen ... 290
6.1.3 Konstruktoren in Strukturen ... 291
6.1.4 Änderung im Projekt »GeometricObjects« ... 292
6.2 Enumerationen (Aufzählungen) ... 295
6.2.1 Wertzuweisung an enum-Mitglieder ... 296
6.2.2 Alle Mitglieder einer Aufzählung durchlaufen ... 297
6.3 Boxing und Unboxing ... 298

7 Fehlerbehandlung und Debugging 299

7.1 Laufzeitfehler behandeln ... 299
7.1.1 Laufzeitfehler erkennen ... 300
7.1.2 Die »try...catch«-Anweisung ... 302
7.1.3 Behandlung mehrerer Exceptions ... 304
7.1.4 Die Reihenfolge der »catch«-Zweige ... 306
7.1.5 Ausnahmen in einer Methodenaufrufkette ... 307
7.1.6 Ausnahmen werfen oder weiterleiten ... 307
7.1.7 Die »finally«-Anweisung ... 308
7.1.8 Die Klasse »Exception« ... 309
7.1.9 Benutzerdefinierte Ausnahmen ... 314
7.2 Debuggen mit Programmcode ... 319
7.2.1 Einführung ... 319
7.2.2 Die Klasse »Debug« ... 320
7.2.3 Die Klasse »Trace« ... 324
7.2.4 Bedingte Kompilierung ... 324
7.3 Fehlersuche mit Visual Studio 2012 ... 327
7.3.1 Debuggen im Haltemodus ... 327
7.3.2 Das »Direktfenster« ... 330
7.3.3 Weitere Alternativen, um Variableninhalte zu prüfen ... 331

8 Auflistungsklassen (Collections) 333

8.1 Grundlagen ... 333
8.2 Collections im Namespace »System.Collections« ... 333
8.2.1 Die elementaren Schnittstellen der Auflistungsklassen ... 335

8.3 Die Klasse »ArrayList« ... 337
8.3.1 Einträge hinzufügen ... 337
8.3.2 Datenaustausch zwischen einem Array und einer »ArrayList« ... 340
8.3.3 Die Elemente einer »ArrayList« sortieren ... 341
8.3.4 Sortieren von Arrays mit »ArrayList.Adapter« ... 346
8.4 Die Klasse »Hashtable« ... 348
8.4.1 Methoden und Eigenschaften der Schnittstelle »IDictionary« ... 348
8.4.2 Beispielprogramm zur Klasse »Hashtable« ... 349
8.5 Die Klassen »Queue« und »Stack« ... 353
8.5.1 Die Klasse »Stack« ... 354
8.5.2 Die Klasse »Queue« ... 355
8.6 Eigene Auflistungen mit »yield« durchlaufen ... 356

9 Generics – Generische Datentypen 359

9.1 Problembeschreibung ... 359
9.2 Bereitstellen einer generischen Klasse ... 360
9.2.1 Mehrere generische Typparameter ... 362
9.2.2 Vorteile der Generics ... 363
9.3 Bedingungen (Constraints) festlegen ... 363
9.3.1 Constraints mit der »where«-Klausel ... 363
9.3.2 Typparameter auf Klassen oder Strukturen beschränken ... 365
9.3.3 Mehrere Constraints definieren ... 365
9.3.4 Der Konstruktor-Constraint »new()« ... 366
9.3.5 Das Schlüsselwort »default« ... 366
9.4 Generische Methoden ... 367
9.4.1 Methoden und Constraints ... 368
9.5 Generics und Vererbung ... 368
9.5.1 Virtuelle generische Methoden ... 369
9.6 Konvertierung von Generics ... 370
9.7 Generische Delegates ... 371
9.7.1 Generische Delegates und Constraints ... 372
9.7.2 Anpassung des Beispiels »GeometricObjects« ... 372
9.8 Nullable-Typen ... 373
9.8.1 Konvertierungen mit Nullable-Typen ... 374
9.9 Generische Collections ... 374
9.9.1 Die Interfaces der generischen Auflistungsklassen ... 375
9.9.2 Die generische Auflistungsklasse »List<T>« ... 375
9.9.3 Vergleiche mit Hilfe des Delegaten »Comparison<T>« ... 378

9.10 Kovarianz und Kontravarianz generischer Typen ... 379
9.10.1 Kovarianz mit Interfaces ... 379
9.10.2 Kontravarianz mit Interfaces ... 381
9.10.3 Zusammenfassung ... 382
9.10.4 Generische Delegaten mit varianten Typparametern ... 383

10 Weitere C#-Sprachfeatures 385

10.1 Implizit typisierte Variablen ... 385
10.2 Anonyme Typen ... 386
10.3 Lambda-Ausdrücke ... 387
10.3.1 Projektion und Prädikat ... 389
10.4 Erweiterungsmethoden ... 389
10.5 Partielle Methoden ... 393
10.5.1 Wo partielle Methoden eingesetzt werden ... 394
10.6 Operatorüberladung ... 396
10.6.1 Einführung ... 396
10.6.2 Die Syntax der Operatorüberladung ... 396
10.6.3 Die Operatorüberladungen im Projekt »GeometricObjectsSolution« ... 397
10.6.4 Die Operatoren »true« und »false« überladen ... 402
10.6.5 Benutzerdefinierte Konvertierungen ... 403
10.7 Indexer ... 407
10.7.1 Überladen von Indexern ... 409
10.7.2 Parameterbehaftete Eigenschaften ... 411
10.8 Attribute ... 414
10.8.1 Das »Flags«-Attribut ... 415
10.8.2 Benutzerdefinierte Attribute ... 418
10.8.3 Attribute auswerten ... 422
10.8.4 Festlegen der Assembly-Eigenschaften in »Assembly-Info.cs« ... 424
10.9 Dynamisches Binden ... 426
10.9.1 Eine kurze Analyse ... 427
10.9.2 Dynamische Objekte ... 427
10.10 Unsicherer (unsafe) Programmcode – Zeigertechnik in C# ... 429
10.10.1 Einführung ... 429
10.10.2 Das Schlüsselwort »unsafe« ... 429
10.10.3 Die Deklaration von Zeigern ... 430
10.10.4 Die »fixed«-Anweisung ... 431
10.10.5 Zeigerarithmetik ... 432
10.10.6 Der Operator »->« ... 433

11 LINQ 435

11.1 Was ist LINQ? 435
11.1.1 Verzögerte Ausführung 436
11.1.2 LINQ-Erweiterungsmethoden an einem Beispiel 437
11.2 LINQ to Objects 440
11.2.1 Musterdaten 440
11.2.2 Die allgemeine LINQ-Syntax 442
11.3 Die Abfrageoperatoren 444
11.3.1 Übersicht der Abfrageoperatoren 444
11.3.2 Die »from«-Klausel 445
11.3.3 Mit »where« filtern 446
11.3.4 Die Projektionsoperatoren 449
11.3.5 Die Sortieroperatoren 450
11.3.6 Gruppieren mit »GroupBy« 451
11.3.7 Verknüpfungen mit »Join« 453
11.3.8 Die Set-Operatoren-Familie 456
11.3.9 Die Familie der Aggregatoperatoren 457
11.3.10 Quantifizierungsoperatoren 460
11.3.11 Aufteilungsoperatoren 461
11.3.12 Die Elementoperatoren 463
11.3.13 Die Konvertierungsoperatoren 466

12 Arbeiten mit Dateien und Streams 467

12.1 Einführung 467
12.2 Namespaces der Ein- bzw. Ausgabe 468
12.2.1 Das Behandeln von Ausnahmen bei E/A-Operationen 469
12.3 Laufwerke, Verzeichnisse und Dateien 469
12.3.1 Die Klasse »File« 469
12.3.2 Die Klasse »FileInfo« 475
12.3.3 Die Klassen »Directory« und »DirectoryInfo« 478
12.3.4 Die Klasse »Path« 482
12.3.5 Die Klasse »DriveInfo« 484
12.4 Die »Stream«-Klassen 485
12.4.1 Die abstrakte Klasse »Stream« 486
12.4.2 Die von »Stream« abgeleiteten Klassen im Überblick 488
12.4.3 Die Klasse »FileStream« 489
12.5 Die Klassen »TextReader« und »TextWriter« 496

12.5.1 Die Klasse »StreamWriter« ... 496
12.5.2 Die Klasse »StreamReader« ... 500
12.6 Die Klassen »BinaryReader« und »BinaryWriter« ... 502
12.6.1 Komplexe binäre Dateien ... 504

13 Binäre Serialisierung 511

13.1 Einführung in die Serialisierung ... 511
13.1.1 Serialisierungsverfahren ... 512
13.2 Serialisierung mit »BinaryFormatter« ... 513
13.2.1 Die Deserialisierung ... 515
13.2.2 Serialisierung mehrerer Objekte ... 516

14 XML 519

14.1 Grundlagen ... 519
14.2 XML-Dokumente ... 519
14.2.1 Wohlgeformte und gültige XML-Dokumente ... 520
14.2.2 Die Regeln eines wohlgeformten XML-Codes ... 522
14.2.3 Kommentare ... 525
14.2.4 Verarbeitungsanweisungen ... 525
14.2.5 Reservierte Zeichen in XML ... 526
14.2.6 CDATA-Abschnitte ... 526
14.2.7 Namensräume (Namespaces) ... 527
14.3 Die Gültigkeit eines XML-Dokuments ... 534
14.3.1 XML Schema Definition (XSD) ... 535
14.3.2 Ein XML-Dokument mit einem XML-Schema verknüpfen ... 536
14.3.3 Die Struktur eines XML-Schemas ... 539
14.4 Die Klasse »XmlReader« ... 545
14.4.1 XML-Dokumente mit einem »XmlReader«-Objekt lesen ... 545
14.4.2 Validieren eines XML-Dokuments ... 551
14.5 Eigenschaften und Methoden der Klasse »XmlReader« ... 554
14.6 Die Klasse »XmlWriter« ... 557
14.6.1 Die Methoden der Klasse »XmlWriter« ... 561
14.7 Navigation durch XML (XPath) ... 562
14.7.1 Die Klasse »XPathNavigator« ... 562
14.7.2 XPath-Ausdrücke ... 566
14.7.3 Der Kontextknoten ... 567
14.7.4 Beispiele mit XPath-Ausdrücken ... 569

14.7.5 Knotenmengen mit der »Select«-Methode ... 571
14.7.6 Auswerten von XPath-Ausdrücken ... 575
14.8 Das Document Object Model (DOM) ... 579
14.8.1 Allgemeines ... 579
14.8.2 Arbeiten mit »XmlDocument« ... 581
14.8.3 »XmlDocument« und »XPathNavigator« ... 582
14.8.4 Die Klasse »XmlNode« (Operationen mit Knoten) ... 582
14.8.5 Manipulieren einer XML-Struktur ... 590
14.8.6 Ändern eines Knotens ... 592
14.8.7 Löschen in einem XML-Dokument ... 594
14.9 Serialisierung mit »XmlSerializer« ... 596
14.9.1 XML-Serialisierung mit Attributen steuern ... 598
14.10 LINQ to XML ... 601
14.10.1 Allgemeines ... 601
14.10.2 Die Klassenhierarchie von LINQ to XML ... 601
14.10.3 Die Klasse »XElement« ... 602
14.10.4 Die Klasse »XDocument« ... 605
14.10.5 Navigation im XML-Dokument ... 605
14.10.6 Änderungen am XML-Dokument vornehmen ... 611

15 Multithreading und die Task Parallel Library (TPL) 613

15.1 Überblick ... 613
15.2 Multithreading mit der Klasse »Thread« ... 614
15.2.1 Einführung in das Multithreading ... 614
15.2.2 Threadzustände und Prioritäten ... 614
15.2.3 Zusammenspiel mehrerer Threads ... 616
15.2.4 Die Entwicklung einer einfachen Multithreading-Anwendung ... 616
15.2.5 Die Klasse »Thread« ... 619
15.2.6 Threadpools nutzen ... 627
15.2.7 Die Synchronisation von Threads ... 629
15.2.8 Der »Monitor« zur Synchronisation ... 631
15.2.9 Das Attribut »MethodImpl« ... 637
15.2.10 Das Synchronisationsobjekt »Mutex« ... 638
15.2.11 Grundlagen asynchroner Methodenaufrufe ... 639
15.2.12 Asynchroner Methodenaufruf ... 640
15.2.13 Asynchroner Aufruf mit Rückgabewerten ... 644
15.2.14 Eine Klasse mit asynchronen Methodenaufrufen ... 647
15.3 Die TPL (Task Parallel Library) ... 650
15.3.1 Allgemeines zur Parallelisierung mit der TPL ... 651

15.3.2 Die Klasse »Parallel« ... 651
15.3.3 Die Klasse »Task« ... 657
15.4 Asynchrone Programmierung mit »async« und »await« ... 661

16 Einige wichtige .NET-Klassen 665

16.1 Die Klasse »Object« ... 665
16.1.1 Referenzvergleiche mit »Equals« und »ReferenceEquals« ... 666
16.1.2 »ToString« und »GetType« ... 666
16.1.3 Die Methode »MemberwiseClone« und das Problem des Klonens ... 667
16.2 Die Klasse »String« ... 670
16.2.1 Das Erzeugen eines Strings ... 671
16.2.2 Die Eigenschaften von »String« ... 672
16.2.3 Die Methoden der Klasse »String« ... 672
16.2.4 Zusammenfassung der Klasse »String« ... 683
16.3 Die Klasse »StringBuilder« ... 684
16.3.1 Allgemeines ... 684
16.3.2 Die Kapazität eines »StringBuilder«-Objekts ... 685
16.3.3 Die Konstruktoren der Klasse »StringBuilder« ... 686
16.3.4 Die Eigenschaften der Klasse »StringBuilder« ... 686
16.3.5 Die Methoden der Klasse »StringBuilder« ... 687
16.3.6 Allgemeine Anmerkungen ... 689
16.4 Der Typ »DateTime« ... 690
16.4.1 Die Zeitspanne »Tick« ... 690
16.4.2 Die Konstruktoren von »DateTime« ... 691
16.4.3 Die Eigenschaften von »DateTime« ... 692
16.4.4 Die Methoden der Klasse »DateTime« ... 693
16.5 Die Klasse »TimeSpan« ... 694
16.6 Ausgabeformatierung ... 697
16.6.1 Formatierung mit der Methode »String.Format« ... 697
16.6.2 Formatierung mit der Methode »ToString« ... 701
16.6.3 Benutzerdefinierte Formatierung ... 701

17 Projektmanagement und Visual Studio 2012 705

17.1 Der Projekttyp »Klassenbibliothek« ... 705
17.1.1 Mehrere Projekte in einer Projektmappe verwalten ... 706
17.1.2 Die Zugriffsmodifizierer »public« und »internal« ... 707
17.1.3 Friend Assemblys ... 707
17.1.4 Einbinden einer Klassenbibliothek ... 708

17.2 Assemblys ... 709
17.2.1 Ein Überblick über das Konzept der Assemblys ... 709
17.2.2 Allgemeine Beschreibung privater und globaler Assemblys ... 710
17.2.3 Die Struktur einer Assembly ... 711
17.2.4 Globale Assemblys ... 716
17.3 Konfigurationsdateien ... 721
17.3.1 Die verschiedenen Konfigurationsdateien ... 721
17.3.2 Die Struktur einer Anwendungskonfigurationsdatei ... 723
17.3.3 Eine Anwendungskonfigurationsdatei mit Visual Studio 2012 bereitstellen ... 726
17.3.4 Einträge der Anwendungskonfigurationsdatei auswerten ... 727
17.3.5 Editierbare, anwendungsbezogene Einträge mit <appSettings> ... 732
17.4 Versionsumleitung in einer Konfigurationsdatei ... 734
17.4.1 Die Herausgeberrichtliniendatei ... 735
17.5 XML-Dokumentation ... 736
17.5.1 Das Prinzip der XML-Dokumentation ... 737
17.5.2 Die XML-Kommentartags ... 739
17.5.3 Generieren der XML-Dokumentationsdatei ... 740
17.6 Der Klassendesigner (Class Designer) ... 742
17.6.1 Ein typisches Klassendiagramm ... 742
17.6.2 Hinzufügen und Ansicht von Klassendiagrammen ... 743
17.6.3 Die Toolbox des Klassendesigners ... 744
17.6.4 Das Fenster »Klassendetails« ... 745
17.6.5 Klassendiagramme als Bilder exportieren ... 747
17.7 Refactoring ... 747
17.7.1 Methode extrahieren ... 748
17.7.2 Bezeichner umbenennen ... 749
17.7.3 Felder einkapseln ... 750
17.8 Code-Snippets (Codeausschnitte) ... 750
17.8.1 Codeausschnitte einfügen ... 751
17.8.2 Die Anatomie eines Codeausschnitts ... 752
17.9 »ClickOnce«-Verteilung ... 753
17.9.1 Allgemeine Beschreibung ... 753
17.9.2 Erstellen einer ClickOnce-Anwendung ... 754
17.9.3 Die Installation einer ClickOnce-Anwendung ... 757

18 Einführung in die WPF und XAML ... 759

18.1 Die Merkmale einer WPF-Anwendung ... 759
18.1.1 Anwendungstypen ... 761

18.1.2 Eine WPF-Anwendung und deren Dateien ... 762
18.1.3 Ein erstes WPF-Beispiel ... 765
18.1.4 Wichtige WPF-Features ... 768
18.1.5 Der logische und der visuelle Elementbaum ... 770
18.2 XAML (Extended Application Markup Language) ... 773
18.2.1 Die Struktur einer XAML-Datei ... 774
18.2.2 Eigenschaften eines XAML-Elements in Attributschreibweise festlegen ... 776
18.2.3 Eigenschaften im Eigenschaftsfenster festlegen ... 776
18.2.4 Die Eigenschaft-Element-Syntax ... 777
18.2.5 Inhaltseigenschaften ... 778
18.2.6 Typkonvertierung ... 781
18.2.7 Markup-Erweiterungen (Markup Extensions) ... 782
18.2.8 XML-Namespaces ... 785
18.2.9 XAML-Spracherweiterungen ... 787

19 WPF-Layout-Container 789

19.1 Die Container-Steuerelemente ... 789
19.1.1 Gemeinsame Eigenschaften der Layout-Container ... 790
19.1.2 Das »Canvas« ... 791
19.1.3 Das »StackPanel« ... 792
19.1.4 Das »WrapPanel« ... 795
19.1.5 Das »DockPanel« ... 796
19.1.6 Das »Grid«-Steuerelement ... 798
19.1.7 Das »UniformGrid« ... 804
19.2 Verschachteln der Layout-Container ... 805

20 Fenster in der WPF 809

20.1 Hosts der WPF ... 809
20.2 Fenster vom Typ »Window« ... 810
20.2.1 Mehrere Fenster in einer Anwendung ... 812
20.3 Fenster vom Typ »NavigationWindow« ... 814
20.3.1 Das »Page«-Element ... 816
20.4 Hosts vom Typ »Frame« ... 817
20.5 Navigation zwischen den Seiten ... 818
20.5.1 Navigation mit »HyperLink« ... 819
20.5.2 Der Verlauf der Navigation – das Journal ... 820
20.5.3 Navigation mit »NavigationService« ... 822
20.5.4 Navigation im Internet ... 824

20.5.5 Navigieren mit dem Ereignis »RequestNavigate« des »HyperLink«-Elements ... 825
20.6 Datenübergabe zwischen den Seiten ... 825
20.6.1 Datenübergabe mit der Methode »Navigate« ... 826
20.7 Nachrichtenfenster mit »MessageBox« ... 828
20.7.1 Die Methode »MessageBox.Show« ... 829

21 WPF-Steuerelemente 833

21.1 Die Hierarchie der WPF-Komponenten ... 833
21.2 Allgemeine Eigenschaften der WPF-Steuerelemente ... 835
21.2.1 Den Außenrand mit der Eigenschaft »Margin« festlegen ... 835
21.2.2 Den Innenrand mit der Eigenschaft »Padding« festlegen ... 835
21.2.3 Die Eigenschaft »Content« ... 836
21.2.4 Die Größe einer Komponente ... 838
21.2.5 Die Ausrichtung einer Komponente ... 839
21.2.6 Die Sichtbarkeit eines Steuerelements ... 840
21.2.7 Die Farbeinstellungen ... 841
21.2.8 Die Schriften ... 842
21.3 Die unterschiedlichen Schaltflächen ... 842
21.3.1 Die Basisklasse »ButtonBase« ... 843
21.3.2 Das Steuerelement »Button« ... 843
21.3.3 Das Steuerelement »ToggleButton« ... 844
21.3.4 Das Steuerelement »RepeatButton« ... 845
21.3.5 Das Steuerelement »Checkbox« ... 847
21.3.6 Das Steuerelement »RadioButton« ... 847
21.4 Einfache Eingabesteuerelemente ... 848
21.4.1 Das Steuerelement »Label« ... 848
21.4.2 Das Steuerelement »TextBox« ... 849
21.4.3 Das Steuerelement »PasswordBox« ... 852
21.4.4 Das Steuerelement »TextBlock« ... 853
21.5 WPF-Listenelemente ... 856
21.5.1 Das Steuerelement »ListBox« ... 857
21.5.2 Die »ComboBox« ... 860
21.5.3 Das Steuerelement »ListView« ... 861
21.5.4 Das Steuerelement »TreeView« ... 863
21.5.5 Das Steuerelement »TabControl« ... 869
21.5.6 Die Menüleiste ... 870
21.5.7 Das Kontextmenü ... 873

21.5.8 Symbolleisten ... 875
21.5.9 Die Statusleiste ... 878
21.6 Weitere Steuerelemente ... 879
21.6.1 Das Steuerelement »ToolTip« ... 879
21.6.2 Die »Progressbar« ... 881
21.6.3 Das Steuerelement »Slider« ... 881
21.6.4 Das »GroupBox«-Steuerelement ... 882
21.6.5 Das Steuerelement »ScrollViewer« ... 883
21.6.6 Das Steuerelement »Expander« ... 885
21.6.7 Das Steuerelement »Border« ... 886
21.6.8 Die »Image«-Komponente ... 887
21.6.9 »Calendar« und »DatePicker« zur Datumsangabe ... 889
21.6.10 Das Steuerelement »InkCanvas« ... 890
21.7 Das »Ribbon«-Steuerelement ... 893
21.7.1 Voraussetzungen für den Zugriff auf das »Ribbon«-Control ... 893
21.7.2 Ein kurzer Überblick ... 893
21.7.3 Der XAML-Code ... 894
21.8 FlowDocuments ... 899
21.8.1 Allgemeine Beschreibung eines FlowDocuments ... 899
21.8.2 Eigenschaften eines »FlowDocuments« ... 900
21.8.3 Die Blöcke eines »FlowDocuments« ... 900
21.8.4 Inline-Elemente ... 905
21.8.5 »FlowDocuments« mit Code erzeugen ... 907
21.8.6 Speichern und Laden eines »FlowDocuments« ... 910
21.9 Das Element »FlowDocumentViewer« ... 911
21.9.1 Das Anzeigeelement »FlowDocumentScrollViewer« ... 911
21.9.2 Das Anzeigeelement »FlowDocumentPageViewer« ... 912
21.9.3 Das Anzeigeelement »FlowDocumentReader« ... 912
21.10 XPS-Dokumente mit »DocumentViewer« ... 913
21.10.1 Allgemeines zum XPS-Format ... 913
21.10.2 Beispielprogramm ... 914
21.11 Das Steuerelement »RichTextBox« ... 915

22 Elementbindungen 921

22.1 Einführung in die Bindungstechnik ... 921
22.1.1 Ein einfaches Bindungsbeispiel ... 921
22.2 Die Klasse »Binding« ... 924
22.2.1 Die Bindungsrichtung festlegen ... 925
22.2.2 Aktualisierung der Bindung ... 928

22.2.3 Die Ereignisse »SourceUpdated« und »TargetUpdated« ... 930
22.2.4 Beenden einer Bindung ... 931
22.3 Bindungsalternativen ... 931
22.3.1 Die Eigenschaft »Source« ... 931
22.3.2 Anbindung an relative Datenquellen ... 932
22.3.3 Die Bindung an »DataContext« ... 934

23 Konzepte von WPF 935

23.1 Anwendungsspezifische Ressourcen ... 935
23.2 Anwendungsübergreifende Ressourcen ... 937
23.2.1 Mehrere Ressourcenwörterbücher ... 939
23.2.2 Die Suche nach einer Ressource ... 940
23.3 Logische Ressourcen ... 940
23.3.1 Statische Ressourcen ... 941
23.3.2 Dynamische Ressourcen ... 944
23.3.3 Ressourcen mit C#-Code bearbeiten ... 945
23.3.4 Abrufen von Systemressourcen ... 946
23.4 Styles ... 948
23.4.1 Einfache Styles ... 948
23.4.2 Typisierte Styles ... 952
23.4.3 Erweitern von Styles ... 953
23.4.4 EventSetter ... 954
23.5 Trigger ... 956
23.5.1 Eigenschaftstrigger ... 957
23.5.2 Datentrigger ... 960
23.5.3 Ereignistrigger ... 961
23.6 Templates ... 962
23.6.1 Allgemeines zu »ControlTemplates« ... 963
23.6.2 Definition innerhalb eines Styles ... 968
23.7 Ermitteln des visuellen Elementbaums ... 969
23.7.1 Das Tool »Expression Blend« ... 969
23.7.2 Standard-Template mit Code abfragen ... 971

24 Datenbindung 975

24.1 Bindung benutzerdefinierter Objekte ... 975
24.1.1 Ein Objekt mit XAML-Code erzeugen und binden ... 976
24.1.2 Ein Objekt mit C#-Code erzeugen und binden ... 977
24.1.3 Aktualisieren benutzerdefinierter Objekte ... 979

24.2 Auflistungen binden ... 981
24.2.1 Allgemeine Gesichtspunkte ... 981
24.2.2 Anbindung an eine »ListBox« ... 982
24.2.3 Änderungen der Collection an die bindenden Elemente weiterleiten ... 984
24.3 Validieren von Bindungen ... 987
24.3.1 Die Validierung im Datenobjekt ... 988
24.3.2 Eine benutzerdefinierte »ValidationRule« ... 990
24.3.3 Validierung mit der Schnittstelle »IDataErrorInfo« ... 991
24.3.4 Fehlerhinweise individuell anzeigen ... 993
24.3.5 Ereignisauslösung bei einem Validierungsfehler ... 995
24.4 Daten konvertieren ... 995
24.4.1 Mehrfachbindungen und Konverterklassen ... 999
24.5 Datenbindung an ADO.NET- und LINQ-Datenquellen ... 1000
24.5.1 Das Binden an ADO.NET-Objekte ... 1001
24.5.2 Das Binden an LINQ-Ausdrücke ... 1002

25 Weitere Möglichkeiten der Datenbindung 1005

25.1 »ItemsControl«-Steuerelemente anpassen ... 1005
25.1.1 Den Style eines »ListBoxItem«-Elements ändern ... 1006
25.1.2 DataTemplates festlegen ... 1008
25.1.3 »DataTemplates« mit Trigger ... 1010
25.2 Alternative Datenbindungen ... 1014
25.2.1 Die Klasse »ObjectDataProvider« ... 1014
25.3 Navigieren, Filtern, Sortieren und Gruppieren ... 1016
25.3.1 Navigieren ... 1018
25.3.2 Sortieren ... 1021
25.3.3 Filtern ... 1022
25.3.4 Gruppieren ... 1026
25.4 Das Steuerelement »DataGrid« ... 1030
25.4.1 Elementare Eigenschaften des »DataGrid« ... 1032
25.4.2 Spalten definieren ... 1033
25.4.3 Details einer Zeile anzeigen ... 1039

26 Dependency Properties 1041

26.1 Die Charakteristik von Abhängigkeitseigenschaften ... 1041
26.2 Den Wert einer Abhängigkeitseigenschaft bilden ... 1042
26.3 Definition einer Dependency Property ... 1043
26.3.1 Registrieren einer Abhängigkeitseigenschaft ... 1044

26.3.2 Der Eigenschaftswrapper ... 1045
26.3.3 Die Eigenschaftsmetadaten ... 1046
26.3.4 Freigabe des spezifischen Eigenschaftswertes ... 1050
26.3.5 Vererbung von Abhängigkeitseigenschaften ... 1050
26.4 Validieren einer Abhängigkeitseigenschaft ... 1051
26.4.1 Validieren mit »ValidateValueCallback« ... 1051
26.4.2 Validieren mit »CoerceValueCallback« ... 1052
26.5 Angehängte Eigenschaften (Attached Property) ... 1053
26.5.1 Angehängte Eigenschaften zur Laufzeit ändern ... 1055

27 Ereignisse in der WPF 1057

27.1 Ereignishandler bereitstellen ... 1057
27.2 Routing-Strategien ... 1058
27.2.1 Der durchlaufene Elementbaum ... 1060
27.2.2 Beispielanwendung ... 1060
27.2.3 Sonderfall der Mausereignisse ... 1062
27.3 Der Ereignishandler ... 1063
27.3.1 Die Klasse »RoutedEventArgs« ... 1063
27.3.2 Die Quelle des Routing-Prozesses ... 1064
27.3.3 Die Eigenschaft »Handled« ... 1065
27.3.4 Registrieren und Deregistrieren eines Ereignishandlers mit Code ... 1066
27.4 Definition eines Routed Events ... 1066
27.4.1 Ereignisauslösung ... 1068
27.4.2 Das Ereignis als Attached Event verwenden ... 1069
27.4.3 Unterdrückte Ereignisse ... 1070
27.5 Mausereignisse in der WPF ... 1071
27.5.1 Ziehen der Maus ... 1071
27.5.2 Auswerten der Mausklicks ... 1072
27.5.3 Capturing ... 1073

28 WPF-Commands 1077

28.1 Allgemeine Beschreibung ... 1077
28.1.1 Ein einführendes Beispiel ... 1077
28.2 Vordefinierte WPF-Commands ... 1079
28.3 Commands verwenden ... 1080
28.3.1 Command-Bindungen einrichten ... 1081
28.3.2 Lokalität der Befehlsbindung ... 1083
28.3.3 Befehlsbindung mit Programmcode ... 1083

28.3.4 Das Befehlsziel mit »CommandTarget« angeben ... 1084
28.3.5 Zusätzliche Daten bereitstellen ... 1085
28.3.6 Befehle mit Maus oder Tastatur aufrufen ... 1086
28.4 Die Anatomie eines »Command«-Objekts ... 1087
28.4.1 Das Interface »ICommand« ... 1088
28.4.2 Die Klassen »RoutedCommand« und »RoutedUICommand« ... 1088
28.4.3 Das Interface »ICommandSource« ... 1090
28.5 Das MVVM-Pattern ... 1091
28.5.1 Ein simples Beispielprogramm ... 1093

29 Benutzerdefinierte Controls 1097

29.1 Erstellen eines benutzerdefinierten Steuerelements ... 1097
29.2 Der XAML-Code ... 1099
29.3 Die Programmlogik des Steuerelements ... 1100
29.3.1 Die Eigenschaften ... 1100
29.3.2 Ein Ereignis bereitstellen ... 1102
29.3.3 Das Steuerelement um einen »Command« ergänzen ... 1103
29.4 Testanwendung ... 1104

30 2D-Grafik 1107

30.1 Shapes ... 1107
30.1.1 Allgemeine Beschreibung ... 1107
30.1.2 Line-Elemente ... 1108
30.1.3 Ellipse- und Rectangle-Elemente ... 1109
30.1.4 Polygon- und Polyline-Elemente ... 1109
30.1.5 Darstellung der Linien ... 1109
30.2 Path-Elemente ... 1111
30.2.1 GeometryGroup ... 1112
30.2.2 CombinedGeometry ... 1113
30.2.3 PathGeometry ... 1114
30.3 Brush-Objekte ... 1115
30.3.1 SolidColorBrush ... 1116
30.3.2 LinearGradientBrush ... 1117
30.3.3 RadialGradientBrush ... 1119
30.3.4 TileBrush ... 1120
30.3.5 ImageBrush ... 1122
30.3.6 VisualBrush ... 1123
30.3.7 DrawingBrush ... 1125

31 ADO.NET – Verbindungsorientierte Objekte 1127

31.1 Allgemeines ... 1127
31.2 Die Datenprovider ... 1128
31.3 Die Verbindung zu einer Datenbank herstellen ... 1129
31.3.1 Das Connection-Objekt ... 1129
31.3.2 Die Verbindungszeichenfolge ... 1130
31.3.3 Die Verbindung mit einer SQL Server-Instanz aufbauen ... 1131
31.3.4 Öffnen und Schließen einer Verbindung ... 1134
31.3.5 Das Verbindungspooling ... 1138
31.3.6 Die Ereignisse eines »Connection«-Objekts ... 1142
31.3.7 Verbindungszeichenfolgen aus einer Konfigurationsdatei abrufen ... 1144
31.3.8 Verbindungen mit dem OleDb-Datenprovider ... 1146

32 ADO.NET – Das Command-Objekt 1149

32.1 Die Datenbankabfrage ... 1149
32.2 Das SqlCommand-Objekt ... 1149
32.2.1 Erzeugen eines SqlCommand-Objekts ... 1150
32.2.2 Die Methode »CreateCommand« des Connection-Objekts ... 1151
32.2.3 Ausführen des SqlCommand-Objekts ... 1151
32.2.4 Die Eigenschaft »CommandTimeout« des SqlCommand-Objekts ... 1152
32.3 Aktionsabfragen absetzen ... 1152
32.3.1 Datensätze hinzufügen ... 1152
32.3.2 Datensätze löschen ... 1153
32.3.3 Datensätze ändern ... 1154
32.3.4 Abfragen, die genau ein Ergebnis liefern ... 1154
32.4 Das SqlDataReader-Objekt ... 1154
32.4.1 Datensätze einlesen ... 1155
32.4.2 Schließen des SqlDataReader-Objekts ... 1157
32.4.3 MARS (Multiple Active Resultsets) ... 1158
32.4.4 Batchabfragen mit »NextResult« durchlaufen ... 1159
32.4.5 Das Schema eines SqlDataReader-Objekts untersuchen ... 1160
32.5 Parametrisierte Abfragen ... 1162
32.5.1 Parametrisierte Abfragen mit dem SqlClient-Datenprovider ... 1162
32.5.2 Die Klasse »SqlParameter« ... 1165
32.5.3 Asynchrone Abfragen ... 1165
32.5.4 Gespeicherte Prozeduren (Stored Procedures) ... 1169

33 ADO.NET – Der SqlDataAdapter 1177

33.1 Was ist ein DataAdapter? 1177
33.2 Die Konstruktoren der Klasse DataAdapter 1179
33.3 Arbeiten mit dem SqlDataAdapter 1179
33.3.1 Die Eigenschaft »SelectCommand« 1179
33.3.2 Den lokalen Datenspeicher mit »Fill« füllen 1180
33.3.3 Öffnen und Schließen von Verbindungen 1181
33.3.4 Doppelter Aufruf der Fill-Methode 1182
33.3.5 Mehrere DataAdapter-Objekte aufrufen 1182
33.3.6 Die Spalten- und der Tabellenbezeichner einer DataTable 1183
33.3.7 Paging mit der Fill-Methode 1183
33.4 Tabellenzuordnung mit der Klasse »TableMappings« 1184
33.4.1 Spaltenzuordnungen in einem DataSet 1186
33.4.2 Spaltenzuordnungen einer DataTable 1187
33.4.3 Die Eigenschaft »MissingMappingAction« des DataAdapters 1188
33.5 Das Ereignis »FillError« des SqlDataAdapters 1188

34 ADO.NET – Daten im lokalen Speicher 1191

34.1 Allgemeines 1191
34.2 Verwenden des DataSet-Objekts 1192
34.2.1 Ein DataSet-Objekt erzeugen 1192
34.2.2 Die Anatomie einer DataTable 1192
34.2.3 Der Zugriff auf eine Tabelle im DataSet 1193
34.2.4 Der Zugriff auf die Ergebnisliste 1194
34.2.5 Dateninformationen in eine XML-Datei schreiben 1195
34.3 Gültigkeitsprüfung im DataSet 1196
34.3.1 Dem DataSet Schemainformationen übergeben 1196
34.3.2 Eigenschaften einer DataColumn, die der Gültigkeitsprüfung dienen 1198
34.3.3 Die Constraints-Klassen einer »DataTable« 1199
34.3.4 Das Schema mit Programmcode erzeugen 1200
34.3.5 Schemainformationen mit SqlDataAdapter abrufen 1201
34.4 Änderungen in einer DataTable vornehmen 1204
34.4.1 Editieren einer DataRow 1204
34.4.2 Löschen einer Datenzeile 1206
34.4.3 Eine neue Datenzeile hinzufügen 1206
34.4.4 Der Sonderfall: Autoinkrementspalten 1207
34.4.5 Was bei einer Änderung einer Datenzeile passiert 1209
34.4.6 Manuelles Steuern der Eigenschaft »DataRowState« 1213

34.5 Mit mehreren Tabellen arbeiten ... 1214
34.5.1 Der Weg über JOIN-Abfragen ... 1214
34.5.2 Mehrere Tabellen in einem DataSet ... 1216
34.5.3 Eine DataRelation erzeugen ... 1216
34.5.4 DataRelations und Einschränkungen ... 1217
34.5.5 In Beziehung stehende Daten suchen ... 1219
34.5.6 Ergänzung zum Speichern von Schemainformationen in einer XML-Schemadatei ... 1221
34.6 Filtern und suchen in einer DataTable ... 1222
34.6.1 Die Methode »Find« ... 1222
34.6.2 Die Methode »Select« ... 1223
34.7 Objekte vom Typ »DataView« ... 1224
34.7.1 Einen »DataView« erzeugen ... 1225
34.7.2 Auf die Datenzeilen in einem »DataView« zugreifen ... 1225
34.7.3 Die Eigenschaft »Sort« und die Methode »Find« ... 1226
34.7.4 Die Methode »FindRows« ... 1226
34.7.5 Die Eigenschaft »RowFilter« ... 1227
34.7.6 Die Eigenschaft »RowStateFilter« ... 1227
34.7.7 Änderungen an einem »DataView«-Objekt ... 1227
34.7.8 Aus einem »DataView« eine »DataTable« erzeugen ... 1229

35 ADO.NET – Aktualisieren der Datenbank 1231

35.1 Aktualisieren mit dem »CommandBuilder« ... 1231
35.1.1 Die von »SqlCommandBuilder« generierten Aktualisierungsstatements ... 1233
35.1.2 Konfliktsteuerung in einer Mehrbenutzerumgebung ... 1233
35.1.3 Die Eigenschaft »ConflictOption« des »SqlCommandBuilders« ... 1236
35.1.4 Die Eigenschaft »SetAllValues« ... 1237
35.2 Manuell gesteuerte Aktualisierung ... 1238
35.2.1 Eigene Aktualisierungslogik ... 1239
35.2.2 Das Beispielprogramm ... 1240
35.3 Konfliktanalyse ... 1242
35.3.1 Den Benutzer über fehlgeschlagene Aktualisierungen informieren ... 1243
35.3.2 Konfliktverursachende Datenzeilen bei der Datenbank abfragen ... 1244
35.4 Neue Autoinkrementwerte abrufen ... 1249

36 Stark typisierte DataSets 1251

36.1 Ein stark typisiertes DataSet erzeugen ... 1251
36.1.1 Typisierte DataSets mit dem Visual Studio Designer erstellen ... 1251

36.1.2 Das Kommandozeilentool XSD.exe ... 1254
36.2 Die Anatomie eines typisierten DataSets ... 1255
36.2.1 Die Datenzeilen einer Tabelle ausgeben ... 1255
36.2.2 Datenzeilen hinzufügen ... 1258
36.2.3 Datenzeilen bearbeiten ... 1259
36.2.4 Datenzeilen suchen ... 1259
36.2.5 NULL-Werte im typisierten DataSet ... 1260
36.2.6 Die Daten in einem hierarchischen DataSet ... 1260
36.3 Typisierte DataSets manuell im Designer erzeugen ... 1261
36.3.1 Eine »DataTable« manuell erzeugen ... 1261
36.3.2 Der »DataTable« Spalten hinzufügen ... 1262
36.3.3 Beziehungen zwischen den Tabellen erstellen ... 1262
36.4 Weiter gehende Betrachtungen ... 1264
36.5 Der »TableAdapter« ... 1264
36.5.1 Einen »TableAdapter« mit Visual Studio erzeugen ... 1264
36.5.2 Die Methode »Fill« des »TableAdapters« ... 1269
36.5.3 Die Methode »GetData« ... 1270
36.5.4 Die Methode »Update« ... 1270
36.5.5 Aktualisieren mit den DBDirect-Methoden ... 1270
36.5.6 TableAdapter mit mehreren Abfragen ... 1271
36.5.7 Änderungen an einem »TableAdapter« vornehmen ... 1274
36.6 Fazit: Typisierte oder nicht typisierte DataSets? ... 1275

37 Einführung in das ADO.NET Entity Framework 1277

37.1 Kritische Betrachtung von ADO.NET ... 1277
37.2 Ein erstes Entity Data Model (EDM) erstellen ... 1279
37.3 Das Entity Data Model im Designer ... 1283
37.3.1 Die übergeordneten Eigenschaften einer Entität ... 1283
37.3.2 Eigenschaften eines Entitätsobjekts ... 1284
37.3.3 Assoziationen im Entity Data Model ... 1287
37.3.4 Der Kontext der Entitäten ... 1288
37.4 Der Aufbau des Entity Data Models ... 1289
37.5 Die Klassen des Entity Data Models (EDM) ... 1292
37.5.1 Die Entitätsklassen ... 1293
37.5.2 Der ObjectContext ... 1296
37.6 Die Architektur des Entity Frameworks ... 1297
37.6.1 Object Services ... 1298
37.6.2 Die Schichten des Entity Frameworks ... 1298

38 Datenabfragen des Entity Data Models (EDM) 1301

38.1 Abfragen mit LINQ to Entities 1302
38.1.1 Allgemeine Begriffe in LINQ 1302
38.1.2 Einfache Abfragen 1302
38.1.3 Navigieren in Abfragen 1309
38.1.4 Aggregatmethoden 1314
38.1.5 Joins in LINQ definieren 1315
38.1.6 In Beziehung stehende Daten laden 1318
38.2 Abfragen mit Entity SQL 1324
38.2.1 Ein erstes Beispiel mit Entity SQL 1324
38.2.2 Die fundamentalen Regeln der Entity-SQL-Syntax 1325
38.2.3 Filtern mit Entity SQL 1326
38.2.4 Parametrisierte Abfragen 1328
38.3 Der EntityClient-Provider 1329
38.3.1 Verbindungen mit »EntityConnection« 1330
38.3.2 Die Klasse »EntityCommand« 1331
38.4 Abfrage-Generator-Methoden (QueryBuilder-Methoden) 1332
38.5 SQL-Direktabfragen 1333

39 Entitätsaktualisierung und Zustandsverwaltung 1335

39.1 Aktualisieren von Entitäten 1335
39.1.1 Entitäten ändern 1335
39.1.2 Hinzufügen neuer Entitäten 1337
39.1.3 Löschen einer Entität 1341
39.2 Der Lebenszyklus einer Entität im Objektkontext 1344
39.2.1 Der Zustand einer Entität 1344
39.2.2 Das Team der Objekte im Überblick 1344
39.2.3 Neue Entitäten im Objektkontext 1345
39.2.4 Die Zustände einer Entität 1347
39.2.5 Zusätzliche Entitäten in den Datencache laden 1349
39.2.6 Die Zustandsverfolgung mit »MergeOption« steuern 1349
39.3 Das »ObjectStateEntry«-Objekt 1352
39.3.1 Die Current- und Originalwerte abrufen 1354
39.3.2 Die Methode »TryGetObjectStateEntry« 1355
39.3.3 Abrufen bestimmter Gruppen 1355
39.3.4 Die Methode »GetModifiedProperties« 1356
39.4 Die Klasse »EntityKey« 1357
39.4.1 Die Methoden »GetObjectByKey« und »TryGetObjectByKey« 1357

39.5 Komplexere Szenarien 1358
39.5.1 Die Methode »ChangeState« 1359
39.5.2 Die Methoden »ApplyCurrentChanges« und »ApplyOriginalChanges« 1360

40 Konflikte behandeln 1363

40.1 Allgemeine Betrachtungen 1363
40.1.1 Das pessimistische Sperren 1364
40.1.2 Das optimistische Sperren 1364
40.2 Konkurrierende Zugriffe mit dem Entity Framework 1365
40.2.1 Das Standardverhalten des Entity Frameworks 1365
40.2.2 Das Aktualisierungsverhalten mit »Fixed« beeinflussen 1366
40.2.3 Auf die Ausnahme »OptimisticConcurrencyException« reagieren 1367
40.2.4 Das »ClientWins«-Szenario 1368
40.2.5 Das »StoreWins«-Szenario 1370

41 Plain Old CLR Objects (POCOs) 1371

41.1 Ein erstes Projekt mit POCO-Klassen 1371
41.1.1 Erstellen einfacher POCO-Klassen 1371
41.1.2 Erstellen des Objektkontextes 1373
41.2 Datenabfrage mit Hilfe der POCOs 1375
41.2.1 In Beziehung stehende Daten laden 1375
41.3 Änderungen verfolgen 1377
41.3.1 Die Methode »DetectChanges« 1377
41.3.2 In Beziehung stehende POCOs aktualisieren 1379

Index 1385

Kapitel 12
Arbeiten mit Dateien und Streams

12.1 Einführung

Das .NET Framework bietet eine Klassenbibliothek, die in Namespaces organisiert ist. Jeder Namespace beschreibt eine zusammenhängende oder zumindest doch verwandte Thematik. Mit Daten zu operieren, egal ob man Daten schreibt oder liest, steht im Zusammenhang mit Dateien. Daher ist es auch nicht erstaunlich, dass sich die wichtigsten Klassen, die mit Dateien und Datenoperationen zu tun haben, in einem Namespace wiederfinden: `System.IO`.

Wollte man ein kurzes, allgemein gehaltenes Inhaltsverzeichnis von `System.IO` angeben, müsste dieses drei Hauptabschnitte umfassen:

1. Klassen, die ihre Dienste auf der Basis von Dateien und Verzeichnissen anbieten
2. Klassen, die den Datentransport beschreiben
3. Ausnahmeklassen

Der Schwerpunkt liegt wohl eher auf den Klassen, die durch Punkt 2 beschrieben werden, und geht weit über die Operationen hinaus, die im direkten Zusammenhang mit Dateien stehen. Daraus resultiert letztendlich auch die Namensangabe des Namespaces `IO` für Input/Output-Operationen oder, wie es auch sehr häufig in der deutschen Übersetzung lautet, E/A-Operationen (für die Ein- und Ausgabe).

In diesem Kapitel geht es primär darum, Dateninformationen aus einer beliebigen Datenquelle zu holen und an ein beliebiges Ziel zu schicken. Meist sind sowohl die Quelle als auch das Ziel eines Datenstroms Dateien, aber es kann auch noch ganz andere Anfangs- und Endpunkte geben, beispielsweise:

- eine Benutzeroberfläche
- Netzwerkverbindungen
- Speicherblöcke
- Drucker
- andere Peripheriegeräte

In gehobenen Programmiersprachen wird ein Datenfluss als **Stream** bezeichnet. Ein Stream hat einen Anfangs- und einen Endpunkt: eine Quelle, an der der Datenstrom entspringt, und das Ziel, das den Datenstrom empfängt. Die Methoden `Console.WriteLine` und `Console.ReadLine`, mit denen wir quasi schon von der ersten Seite dieses Buches an arbeiten, erzeugen auch solche Datenströme.

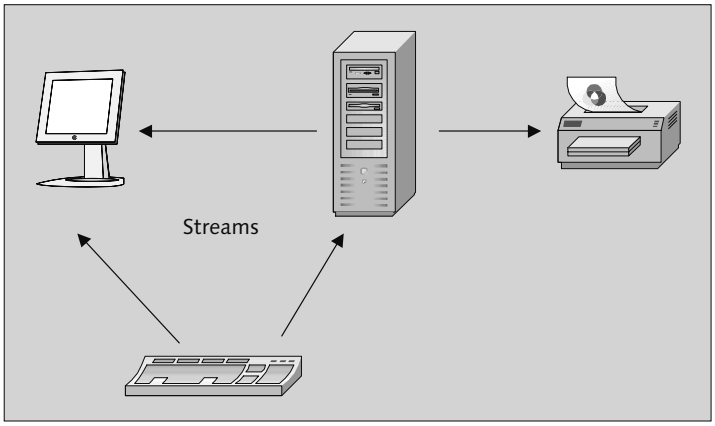


Abbildung 12.1 Datenströme einer lokalen Arbeitsstation

Streams haben individuelle Charakteristiken. Das ist auch der Grund, weshalb es nicht nur eine `Stream`-Klasse gibt, sondern mehrere. Jeder Stream dient ganz speziellen Anforderungen und kann diese mehr oder weniger gut erfüllen. Beispielsweise gibt es Streams, deren Daten direkt als Text interpretiert werden, während andere nur Bytesequenzen transportieren, die der Empfänger erst in das richtige Format bringen muss, um den Inhalt zu interpretieren.

Ein Stream ist nicht dauerhaft: Er wird geöffnet und liest oder schreibt Daten. Nach dem Schließen sind die Daten verloren, wenn sie nicht von einem Empfänger, beispielsweise einer Datei, dauerhaft gespeichert werden.

12.2 Namespaces der Ein- bzw. Ausgabe

Die elementarsten Klassen für die Ein- und Ausgabe sind im Namespace `System.IO` organisiert. Es sollte nicht unerwähnt bleiben, dass die .NET-Klassenbibliothek mit weiteren Namespaces aufwartet, die Klassen für besondere Aufgaben bereitstellen.

- Im Namespace `System.IO.Compression` werden mit `DeflateStream` und `GZipStream` zwei Klassen angeboten, die Methoden und Eigenschaften zur Datenkomprimierung und -dekomprimierung bereitstellen.
- Mit den Klassen des Namespaces `System.IO.IsolatedStorage` wird eine Art virtuelles Dateisystem beschrieben. Dieses ermöglicht die Speicherung von Einstellungen und temporären Daten, die mit der Anwendung eindeutig verknüpft sind. Typischerweise werden im isolierten Speicher Daten abgelegt, die ansonsten beispielsweise in der Registry gespeichert werden müssten. Das Besondere dabei ist, dass weniger vertrauenswürdiger Code auf die im isolierten Speicher befindlichen Daten nicht zugreifen kann.
- Streams müssen nicht zwangsläufig mit Dateien oder Verzeichnissen in direktem Zusammenhang stehen, sondern beschreiben Datenströme in allgemeiner Form. Wollen Sie die

serielle Schnittstelle programmieren, werden Sie daher auch auf die Methoden und Eigenschaften der Klassen im Namespace `System.IO.Ports` zurückgreifen müssen.

12.2.1 Das Behandeln von Ausnahmen bei E/A-Operationen

Bei fast allen Dateioperationen kann es zur Laufzeit eines Programms aus den verschiedensten Gründen sehr schnell zu Ausnahmen kommen. Beispielsweise wird eine zu kopierende Datei im angegebenen Pfad nicht gefunden, das Zielverzeichnis existiert nicht, als Quelle oder Ziel wird ein Leerstring übergeben usw. Daher sollten Sie unbedingt darauf achten, eine Fehlerbehandlung zu implementieren. Die Dokumentation unterstützt Sie, wenn es darum geht, auf mögliche Fehler zu reagieren, denn dort werden alle Ausnahmen aufgeführt, die beim Aufruf einer Methode auftreten könnten.

Alle Ausnahmen im Zusammenhang mit E/A-Operationen werden auf eine gemeinsame Basis zurückgeführt: `IOException`. Sie sollten auch diesen allgemeinen Fehler immer behandeln, damit der Anwender nicht Gefahr läuft, durch eine unberücksichtigte Ausnahme die Laufzeitumgebung des Programms unfreiwillig zu beenden.

12.3 Laufwerke, Verzeichnisse und Dateien

Die Klassenbibliothek des .NET Frameworks unterstützt den Entwickler mit mehreren Klassen, die Laufwerke, Verzeichnisse und Dateien beschreiben. Diese wollen wir als Erstes ansehen. Dabei handelt es sich um:

- `File`
- `FileInfo`
- `Directory`
- `DirectoryInfo`
- `Path`
- `DriveInfo`

`File` bzw. `FileInfo` und `Directory` bzw. `DirectoryInfo` liefern sehr ähnliche Daten zurück. Bei `File`/`FileInfo` handelt es sich um Daten, die eine Datei betreffen, bei `Directory`/`DirectoryInfo` um Daten von Verzeichnissen. Möchten Sie Pfadangaben ermitteln, hilft Ihnen `Path` weiter. Erwähnenswert ist zudem `DriveInfo`, um Laufwerksinformationen abzufragen.

12.3.1 Die Klasse »File«

Methoden der Klasse »File«

Die Klasse `File` ist statisch definiert und stellt daher nur statische Methoden zur Verfügung. Die Methoden der sehr ähnlichen Klasse `FileInfo` hingegen sind Instanzmethoden. Funktionell sind sich beide Klassen ähnlich und unterscheiden sich nicht gravierend.

Mit den Klassenmethoden von `File` lässt sich eine Datei erstellen, kopieren, löschen usw. Sie können auch die Attribute einer Datei lesen oder setzen, und – was auch sehr wichtig ist – Sie können eine Datei öffnen. In Tabelle 12.1 sind die wichtigsten Methoden samt Rückgabetyp aufgeführt.

Methode	Rückgabetyp	Beschreibung
`AppendAllText`	`void`	Öffnet eine Datei und fügt die angegebene Zeichenfolge an die Datei an.
`AppendText`	`StreamWriter`	Hängt Text an eine existierende Datei an.
`Copy`	`void`	Kopiert eine bestehende Datei an einen anderen Speicherort.
`Create`	`FileStream`	Erzeugt eine Datei in einem angegebenen Pfad.
`CreateText`	`StreamWriter`	Erstellt oder öffnet eine Textdatei.
`Delete`	`void`	Löscht eine Datei.
`Exists`	`Boolean`	Gibt einen booleschen Wert zurück, der `false` ist, wenn die angegebene Datei nicht existiert.
`GetAttributes`	`FileAttributes`	Liefert das Bitfeld der Dateiattribute.
`GetCreationTime`	`DateTime`	Liefert das Erstellungsdatum und die Uhrzeit einer Datei.
`GetLastAccessTime`	`DateTime`	Liefert Datum und Uhrzeit des letzten Zugriffs.
`GetLastWriteTime`	`DateTime`	Liefert Datum und Uhrzeit des letzten Schreibzugriffs.
`Move`	`void`	Verschiebt eine Datei in einen anderen Ordner oder benennt sie um.
`Open`	`FileStream`	Öffnet eine Datei.
`OpenRead`	`FileStream`	Öffnet eine Datei zum Lesen.
`OpenText`	`StreamReader`	Öffnet eine Textdatei zum Lesen.
`OpenWrite`	`FileStream`	Öffnet eine Datei zum Schreiben.
`ReadAllBytes`	`byte[]`	Öffnet eine Binärdatei und liest den Inhalt der Datei in ein Byte-Array ein.
`ReadAllLines`	`string[]`	Öffnet eine Textdatei und liest alle Zeilen der Datei in ein Zeichenfolgen-Array ein.

Tabelle 12.1 Methoden der Klasse »File«

Methode	Rückgabetyp	Beschreibung
ReadAllText	string	Öffnet eine Textdatei, liest alle Zeilen der Datei in eine Zeichenfolge ein und schließt dann die Datei.
SetAttributes	void	Setzt Dateiattribute.
SetCreationTime	void	Setzt Erstellungsdatum und -uhrzeit.
SetLastAccessTime	void	Setzt Datum und Uhrzeit des letzten Zugriffs.
SetLastWriteTime	void	Setzt Datum und Uhrzeit des letzten Schreibzugriffs.
WriteAllBytes	void	Erstellt eine neue Datei und schreibt das angegebene Byte-Array in die Datei.
WriteAllLines	void	Erstellt eine neue Datei und schreibt das angegebene Zeichenfolgen-Array in die Datei.
WriteAllText	void	Erstellt eine neue Datei und schreibt das angegebene Zeichenfolgen-Array in die Datei.

Tabelle 12.1 Methoden der Klasse »File« (Forts.)

Wie Sie sehen können, geben viele Methoden ein `Stream`-Objekt zurück. Das hat seinen Grund, denn mit einer geöffneten Datei will man arbeiten, sei es, um den Inhalt zu lesen, oder sei es, um etwas in die Datei zu schreiben. Diese Operationen setzen aber einen Stream voraus, was das .NET Framework durch die Definition verschiedener `Stream`-Klassen abdeckt. Ein `FileStream` beschreibt dabei einfache Bytesequenzen, ein `StreamReader` arbeitet mit ASCII-basierten Dateien.

Kopieren einer Datei

Zum Kopieren einer Datei dient die Methode `Copy`, beispielsweise:

```
File.Copy("C:\\Test.txt", "D:\\Test.txt");
```

Das erste Argument erwartet die Angabe des Dateinamens der zu kopierenden Datei. Befindet sich die zu kopierende Datei in keinem bekannten Suchpfad, muss der gesamte Zugriffspfad beschrieben werden. Im zweiten Argument müssen Sie das Zielverzeichnis und den Namen der Dateikopie angeben. Den Namen der kopierten Datei dürfen Sie gemäß den systemspezifischen Richtlinien festlegen, er muss nicht mit dem Ursprungsnamen der Datei übereinstimmen. Versucht man, ein Ziel anzugeben, in dem bereits eine gleichnamige Datei existiert, wird die Ausnahme `DirectoryNotFoundException` ausgelöst.

Wenn Sie die Pfadangabe in einer Zeichenfolge beschreiben, müssen Sie beachten, dass das einfache Backslash-Zeichen als Escape-Zeichen interpretiert wird. Um diese Interpretation

aufzuheben, müssen Sie entweder zwei Backslashes hintereinander angeben oder alternativ der Zeichenfolge ein @-Zeichen voranstellen.

```
File.Copy(@"C:\Test.txt", @"D:\Test.txt");
```

Sie können die Methode `Copy` auch einsetzen, wenn im Zielverzeichnis bereits eine Datei existiert, die denselben Namen hat wie die Datei, die Sie im zweiten Argument angeben. Rufen Sie dann die Überladung von `Copy` auf, die im dritten Parameter einen booleschen Wert erwartet. Übergeben Sie `true`, wird keine Ausnahme ausgelöst und die bereits vorhandene Datei durch den Inhalt der im ersten Argument übergebenen Datei überschrieben.

Löschen einer Datei

Zum Löschen einer Datei dient die statische Methode `Delete`. Dieser wird der komplette Pfad der zu löschenden Datei übergeben, z.B.:

```
File.Delete(@"C:\MyDocuments\MyDoc.txt");
```

Die Pfadangabe kann absolut oder relativ sein.

Verschieben einer Datei

Mit `Move` lassen sich Dateien aus einem Quellverzeichnis in ein anderes Zielverzeichnis verschieben:

```
File.Move(@"C:\MyDocuments\MyDoc.txt", @"C:\Allgemein\MyDoc.Doc");
```

Diese Methode kann auch zum Umbenennen von Dateinamen benutzt werden.

Prüfen, ob eine Datei existiert

Beabsichtigen Sie, eine bestimmte Datei zu öffnen, stellt sich zunächst die Frage, ob eine Datei dieses Namens in dem angegebenen Pfad tatsächlich existiert. Die Klasse `File` veröffentlicht zur Beantwortung die Methode `Exists`, die den booleschen Wert `false` zurückliefert, wenn die Datei nicht gefunden wird.

```
[...]
string path = @"C:\MyFile.txt";
if (File.Exists(path)) {
  // Datei existiert im angegebenen Pfad
}
[...]
```

Listing 12.1 Prüfen, ob eine bestimmte Datei bereits existiert

Eine ähnliche Codesequenz ist in jedem Programm sinnvoll, in dem eine Operation die Existenz einer Datei zwingend voraussetzt. Das erspart die Codierung einer Ausnahmebehandlung.

Öffnen einer Datei

Zum Öffnen einer Datei benutzen Sie eine der Methoden `OpenRead`, `OpenText`, `OpenWrite` oder `Open` (siehe auch Tabelle 12.1). Sehen wir uns exemplarisch die komplexeste Überladung der letztgenannten Methode an.

```
public static FileStream Open(string path, FileMode mode,
                              FileAccess access, FileShare share);
```

Dem Parameter *path* wird beim Aufruf die Pfadangabe als Zeichenfolge mitgeteilt. Diese besteht aus dem Pfad und dem Dateinamen.

Für das Öffnen einer Datei ist das Betriebssystem zuständig, das wissen muss, wie es die Datei öffnen soll. Der *mode*-Parameter vom Typ `FileMode` steuert dieses Verhalten. Dabei handelt es sich um eine im Namespace `System.IO` definierte Enumeration, die insgesamt sechs Konstanten definiert (siehe Tabelle 12.2).

FileMode-Konstante	Beschreibung
`Append`	Öffnet eine bestehende Datei und setzt den Dateizeiger an das Dateiende. Damit wird das Anhängen von Dateninformationen an die Datei ermöglicht. Existiert die Datei noch nicht, wird sie erzeugt.
`Create`	Erzeugt eine neue Datei. Existiert bereits eine gleichnamige Datei, wird diese überschrieben.
`CreateNew`	Erzeugt in jedem Fall eine neue Datei. Existiert im angegebenen Pfad bereits eine gleichnamige Datei, wird die Ausnahme `IOException` ausgelöst.
`Open`	Öffnet eine bestehende Datei. Wird diese unter der Pfadangabe nicht gefunden, kommt es zur Ausnahme `FileNotFoundException`.
`OpenOrCreate`	Öffnet eine bestehende Datei. Sollte diese im angegebenen Pfad nicht existieren, wird eine neue erzeugt.
`Truncate`	Öffnet eine Datei und löscht deren Inhalt. Nachfolgende Leseoperationen führen dazu, dass eine Ausnahme ausgelöst wird.

Tabelle 12.2 Die Konstanten der Enumeration »FileMode«

Der *mode*-Parameter beschreibt das Verhalten des Betriebssystems beim Öffnen einer Datei, gibt jedoch nicht an, was mit dem Inhalt der Datei geschehen soll. Soll er nur gelesen werden, oder möchte der Anwender in die Datei schreiben? Vielleicht sind auch beide Operationen gleichzeitig gewünscht. Diese Festlegung wird im Parameter *access* getroffen, der ebenfalls auf einer Aufzählung basiert – `FileAccess`. Diese hat nur drei Mitglieder: `Read`, `Write` und `ReadWrite`.

FileAccess-Konstante	Beschreibung
`Read`	Datei wird für den Lesezugriff geöffnet.
`Write`	Datei wird für den Schreibzugriff geöffnet.
`ReadWrite`	Datei wird für den Lese- und Schreibzugriff geöffnet.

Tabelle 12.3 Die Konstanten der Enumeration »FileAccess«

Eine Datei, die mit `FileAccess.Read` geöffnet wird, ist schreibgeschützt. Eine lesegeschützte Datei, deren Inhalt verändert werden soll, wird mit `FileAccess.Write` geöffnet. Die dritte Konstante `FileAccess.ReadWrite` beschreibt sowohl einen lesenden als auch einen schreibenden Zugriff.

Kommen wir nun zum letzten Parameter der `Open`-Methode – *share*. Dieser beschreibt das Verhalten der Datei, wenn nach dem ersten Öffnen weitere Zugriffe auf die Datei erfolgen. Wie schon von den beiden vorher besprochenen Parametern bekannt ist, wird auch dieser durch Konstanten beschrieben, die einer Enumeration des Namespaces `System.IO` zugerechnet werden – `FileShare`. Die Mitglieder ähneln denen der Enumeration `FileAccess`, werden aber um ein weiteres Mitglied ergänzt (genau genommen sind es zwei, aber das zweite spielt für uns keine Rolle).

FileShare-Konstante	Beschreibung
`None`	Alle weiteren Versuche, diese Datei zu öffnen, werden konsequent abgelehnt.
`Read`	Diese Datei darf von anderen Anwendungen oder Threads nur zum Lesen geöffnet werden.
`Write`	Diese Datei darf von anderen Anwendungen oder Threads nur zum Editieren geöffnet werden.
`ReadWrite`	Diese Datei darf von anderen Anwendungen oder Threads sowohl zum Lesen als auch zum Editieren geöffnet werden.

Tabelle 12.4 Die Konstanten der Enumeration »FileShare«

Damit haben wir die Parameterliste der Methode `Open` abgehandelt. Ihnen stehen alle Hilfsmittel zur Verfügung, eine beliebige Datei unter bestimmten Voraussetzungen und Begleitumständen zu öffnen. Führen Sie sich aber vor Augen, dass eine geöffnete Datei nicht automatisch ihre Dateninformationen liefert oder sich manipulieren lässt. Diese Operationen haben mit dem Vorgang des Öffnens noch nichts zu tun, setzen ihn aber voraus. Beachten Sie in diesem Zusammenhang, dass Operationen, die den Inhalt einer Datei beeinflussen, auf dem zurückgegebenen Objekt vom Typ `FileStream` ausgeführt werden.

In der folgenden Codezeile wird exemplarisch eine Datei mit `Open` unter Angabe verschiedener Optionen geöffnet:

```
FileStream stream = File.Open(@"C:\MyTestfile.txt",
    FileMode.OpenOrCreate, FileAccess.ReadWrite, FileShare.None);
```

Die Parameter besagen, dass die Datei *MyTestfile.txt* im Laufwerk *C:* geöffnet werden soll – falls es dort eine solche Datei gibt. Wenn nicht, wird sie neu erzeugt. Der Inhalt der Datei lässt sich nach dem Öffnen sowohl lesen als auch ändern. Gleichzeitig werden weitere Zugriffe auf die Datei strikt unterbunden.

Der einfachste Weg, in eine Datei zu schreiben und daraus zu lesen

Der Weg, um in eine Datei zu schreiben oder eine Datei zu lesen, erforderte bisher immer mehrere Codezeilen. Eigentlich viel zu viel Aufwand, um eben schnell eine einfache Dateioperation auszuführen. Das .NET Framework wartet mit einer kaum vollständig zu beschreibenden Vielzahl an Möglichkeiten auf, die uns das Leben als Entwickler vereinfachen sollen. Besonders erwähnt werden sollen an dieser Stelle exemplarisch die Methoden `ReadAllBytes`, `ReadAllLines` und `ReadAlltext` zum Lesen und `WriteAllBytes`, `WriteAllLines` und `WriteAllText` zum Schreiben. Diese gehören zur Klasse `File`. Mit der simplen Anweisung

```
File.WriteAllText(@"C:\MyTextFile.txt", text);
```

können Sie bereits den Inhalt der Variablen *text* in die angegebene Datei schreiben. Existiert die Datei schon, wird sie einfach überschrieben. Genauso einfach ist auch die inhaltliche Auswertung der Datei:

```
Console.WriteLine(File.ReadAllText(@"C:\MyTextFile.txt"));
```

12.3.2 Die Klasse »FileInfo«

Die ebenfalls nicht ableitbare Klasse `FileInfo` ist das Pendant zu der im vorigen Abschnitt beschriebenen Klasse `File`. Während `File` nur statische Methoden veröffentlicht, beziehen sich die Methoden der Klasse `FileInfo` auf eine konkrete Instanz. Um diese zu erhalten, steht nur ein Konstruktor zur Verfügung, dem Sie als Argument die Pfadangabe zu der Datei übergeben müssen, z. B.:

```
FileInfo myFile = new FileInfo(@"C:\MyDocuments\MyFile.txt");
```

Der Konstruktor prüft nicht, ob die Datei tatsächlich existiert. Bevor Sie Operationen auf das Objekt ausführen, sollten Sie deshalb in jedem Fall vorher mit `Exists` sicherstellen, dass die Datei existiert.

```
if (myFile.Exists) {
  // Datei existiert
}
```

Während `Exists` in der Klasse `File` als Methode implementiert ist, der die Pfadangabe beim Aufruf übergeben werden muss, handelt es sich in der Klasse `FileInfo` um eine schreibgeschützte Eigenschaft des `FileInfo`-Objekts.

Die Eigenschaften eines »FileInfo«-Objekts

`FileInfo` veröffentlicht eine Reihe von Eigenschaften, denen der Zustand der Datei entnommen werden kann. So können Sie beispielsweise die Länge der Datei abfragen oder sich ein Objekt vom Typ `Directory` zurückgeben lassen (ein `Directory`-Objekt beschreibt ein Verzeichnis als Objekt, ähnlich wie `FileInfo` eine Datei beschreibt).

Eigenschaft	Beschreibung
`Attributes`	Ermöglicht das Setzen oder Auswerten der Dateiattribute (*Hidden*, *Archive*, *ReadOnly* usw.).
`CreationTime`	Liefert oder setzt das Erstellungsdatum der Datei.
`Directory`	Liefert eine Instanz des Verzeichnisses.
`DirectoryName`	Liefert eine Zeichenfolge mit der vollständigen Pfadangabe, jedoch ohne den Dateinamen.
`Extension`	Liefert die Dateierweiterung einschließlich des vorangestellten Punktes.
`FullName`	Gibt einen String mit der vollständigen Pfadangabe einschließlich des Dateinamens zurück.
`LastAccessTime`	Liefert oder setzt die Zeit des letzten Zugriffs auf die Datei.
`LastWriteTime`	Liefert oder setzt die Zeit des letzten schreibenden Zugriffs auf die Datei.
`Length`	Gibt die Länge der Datei zurück.
`Name`	Gibt den vollständigen Namen der Datei zurück.

Tabelle 12.5 Die Eigenschaften eines Objekts vom Typ »FileInfo«

Dateiattribute setzen und auswerten

Wir wollen uns aus Tabelle 12.5 die Eigenschaft `Attributes` genauer ansehen, die ihrerseits vom Typ `FileAttributes` ist. `Attributes` beschreibt ein Bit-Feld bestimmter Größe. Jedes Attribut einer Datei wird durch Setzen eines bestimmten Bits in diesem Bit-Feld beschrieben. Um festzustellen, ob ein Dateiattribut gesetzt ist, muss das alle Attribute beschreibende Bit-Feld mit dem gesuchten Dateiattribut bitweise »&«-verknüpft werden. Weicht das Ergebnis von der Zahl 0 ab, ist das Bit gesetzt.

Dazu ein Zahlenbeispiel: Nehmen wir an, das Attribut XYZ würde durch die Bitkombination 0000 1000 (= 8) beschrieben und das Bit-Feld würde aktuell 0010 1001 (= 41) enthalten. Um zu prüfen, ob das Attribut XYZ durch das Bit-Feld beschrieben wird, gilt:

```
    0000 1000
&   0010 1001
--------------
=   0000 1000
```

Das Ergebnis ist nicht »0« und daher so zu interpretieren, dass das Attribut im Bit-Feld gesetzt ist.

Um aus einer Datei ein bestimmtes Attribut herauszufiltern, beispielsweise `Hidden`, müssten wir daher wie folgt vorgehen:

```
FileInfo f = new FileInfo(@"C:\Testfile.txt");
if (0 != (f.Attributes & FileAttributes.Hidden))
{
  // Datei ist versteckt (hidden)
  [...]
}
```

Listing 12.2 Prüfen des Hidden-Attributs

Soll das Attribut gesetzt werden, bietet sich der ^-Operator an:

```
f.Attributes = f.Attributes ^ FileAttributes.Hidden;
```

In gleicher Weise können Sie mit den Methoden `GetAttributes` und `SetAttributes` der Klasse `File` arbeiten.

Die Methoden eines »FileInfo«-Objekts

Die Klassen `File` und `FileInfo` sind sich in den Funktionalitäten, die dem Entwickler angeboten werden, sehr ähnlich: Es lassen sich Dateien löschen, verschieben, umbenennen, kopieren, öffnen usw. Viele geben ein `Stream`-Objekt für weiter gehende Operationen zurück.

Methode	Rückgabetyp	Beschreibung
`AppendText`	`StreamWriter`	Hängt Text an eine existierende Datei an.
`CopyTo`	`FileInfo`	Kopiert die Datei an einen anderen Speicherort.
`Create`	`FileStream`	Erzeugt eine Datei.
`CreateText`	`StreamWriter`	Erzeugt eine neue Textdatei.
`Delete`		Löscht die Datei.
`Exists`	`Boolean`	Gibt einen booleschen Wert zurück, der false ist, wenn die angegebene Datei nicht existiert.

Tabelle 12.6 Die Methoden eines Objekts vom Typ »FileInfo«

Methode	Rückgabetyp	Beschreibung
MoveTo		Verschiebt die Datei in einen anderen Ordner oder benennt sie um.
Open	FileStream	Öffnet eine Datei.
OpenRead	FileStream	Öffnet eine Datei zum Lesen.
OpenText	StreamReader	Öffnet eine Textdatei zum Lesen.
OpenWrite	FileStream	Öffnet eine Datei zum Schreiben.

Tabelle 12.6 Die Methoden eines Objekts vom Typ »FileInfo« (Forts.)

12.3.3 Die Klassen »Directory« und »DirectoryInfo«

Die Klasse `File` veröffentlicht statische Methoden, um mit Dateien zu operieren, und die Klasse `FileInfo` beschreibt die Referenz auf ein konkretes Dateiobjekt. Ähnlich gestaltet sind die Klassen `Directory` und `DirectoryInfo`: `Directory` hat nur statische Methoden, und `DirectoryInfo` basiert auf einer konkreten Instanz. Es stellt sich natürlich sofort die Frage, warum die Architekten der Klassenbibliothek jeweils zwei Klassen mit nahezu gleichen Fähigkeiten vorgesehen haben.

Der entscheidende Unterschied liegt in der Art und Weise, wie die Klassen im Hintergrund arbeiten. Zugriffe auf das Dateisystem setzen immer operative Berechtigungen voraus. Verfügt der Anwender nicht über die entsprechenden Rechte, wird die angeforderte Aktion abgelehnt. Die beiden Klassen `File` und `Directory` prüfen das bei jedem Zugriff erneut und belasten so das System unnötig, während die Überprüfung von den Klassen `DirectoryInfo` und `FileInfo` nur einmal ausgeführt wird.

Mit `Directory` kann man Ordner anlegen, löschen oder verschieben, die in einem Verzeichnis physikalisch gespeicherte Dateinamen abrufen. Man kann mit `Directory` außerdem verzeichnisspezifische Eigenschaften sowie das Erstellungsdatum oder das Datum des letzten Zugriffs ermitteln. Tabelle 12.7 liefert einen Überblick über die Methoden von `Directory`.

Methode	Beschreibung
CreateDirectory	Erzeugt ein Verzeichnis oder Unterverzeichnis.
Delete	Löscht ein Verzeichnis.
Exists	Überprüft, ob das angegebene Verzeichnis existiert.
GetCreationTime	Liefert das Erstellungsdatum samt Uhrzeit.

Tabelle 12.7 Methoden der Klasse »Directory«

Methode	Beschreibung
GetDirectories	Liefert die Namen aller Unterverzeichnisse eines spezifizierten Ordners.
GetFiles	Liefert alle Dateinamen eines spezifizierten Ordners zurück.
GetFileSystemEntries	Liefert die Namen aller Unterverzeichnisse und Dateien eines spezifizierten Ordners.
GetParent	Liefert den Namen des übergeordneten Verzeichnisses.
Move	Verschiebt ein Verzeichnis samt Dateien an einen neuen Speicherort.
SetCreationTime	Legt Datum und Uhrzeit eines Verzeichnisses fest.

Tabelle 12.7 Methoden der Klasse »Directory« (Forts.)

Die Fähigkeiten der Klasse `DirectoryInfo` ähneln denen von `Directory`, setzen jedoch ein konkretes Objekt für den Zugriff auf die Elementfunktionen voraus.

Beispielprogramm

Im folgenden Beispielprogramm werden einige Methoden und Eigenschaften der Klassen `File`, `FileInfo` und `Directory` benutzt. Das Programm fordert den Anwender dazu auf, an der Konsole ein beliebiges Verzeichnis anzugeben, dessen Unterverzeichnisse und Dateien ermittelt und unter Angabe der Dateigröße und der Dateiattribute an der Konsole ausgegeben werden.

```
// Beispiel: ..\Kapitel 12\FileDirectorySample
class Program {
  public static void Main() {
    Program dirTest = new Program();
    FileInfo myFile;
    // Benutzereingabe anfordern
    string path = dirTest.PathInput();
    int len = path.Length;
    // alle Ordner und Dateien holen
    string[] str = Directory.GetFileSystemEntries(path);
    Console.WriteLine();
    Console.WriteLine("Ordner und Dateien im Verzeichnis {0}", path);
    Console.WriteLine(new string('-', 80));
    for (int i = 0; i <= str.GetUpperBound(0); i++) {
      // prüfen, ob der Eintrag ein Verzeichnis oder eine Datei ist
      if(0 == (File.GetAttributes(str[i]) & FileAttributes.Directory)) {
        // str(i) ist kein Verzeichnis
        myFile = new FileInfo(str[i]);
```

```
        string fileAttr = dirTest.GetFileAttributes(myFile);
        Console.WriteLine("{0,-30}{1,25} kB {2,-10} ",
                                    str[i].Substring(len - 1),
                                    myFile.Length / 1024,
                                    fileAttr);
      }
      else
        Console.WriteLine("{0,-30}{1,-15}", str[i].Substring(len), "Dateiordner");
    }
    Console.ReadLine();
  }
  // Benutzer zur Eingabe eines Pfads auffordern
  string PathInput() {
    Console.Write("Geben Sie den zu durchsuchenden ");
    Console.Write("Ordner an: ");
    string searchPath = Console.ReadLine();
    // wenn die Benutzereingabe als letztes Zeichen kein'\'
    // enthält, muss dieses angehängt werden
    if(searchPath.Substring(searchPath.Length - 1) != "\\")
      searchPath += "\\";
    return searchPath;
  }
  // Feststellung, welche Dateiattribute gesetzt sind, und
  // Rückgabe eines Strings, der die gesetzten Attribute enthält
  string GetFileAttributes(FileInfo strFile) {
    string strAttr;
    // prüfen, ob das Archive-Attribut gesetzt ist
    if(0 != (strFile.Attributes & FileAttributes.Archive))
      strAttr = "A ";
    else
      strAttr = "  ";
    // prüfen, ob das Hidden-Attribut gesetzt ist
    if(0 != (strFile.Attributes & FileAttributes.Hidden))
      strAttr += "H ";
    else
      strAttr += "  ";
    // prüfen, ob das ReadOnly-Attribut gesetzt ist
    if(0 != (strFile.Attributes & FileAttributes.ReadOnly))
      strAttr += "R ";
    else
      strAttr += "  ";
    // prüfen, ob das System-Attribut gesetzt ist
    if(0 != (strFile.Attributes & FileAttributes.System))
      strAttr += "S ";
    else
      strAttr += "  ";
```

```
    return strAttr;
  }
}
```

Listing 12.3 Analyse eines Verzeichnisses

Starten Sie die Anwendung, könnte die Ausgabe an der Konsole ungefähr so wie in Abbildung 12.2 dargestellt aussehen.

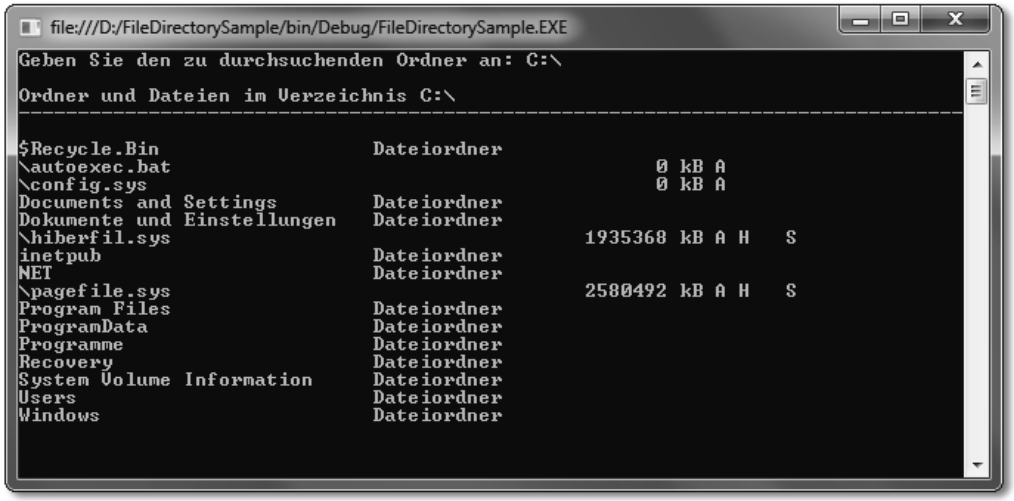


Abbildung 12.2 Ausgabe des Beispiels »FileDirectorySample«

Die komplette Anwendung ist in der Klasse `Program` realisiert. Die Klasse enthält neben der statischen Methode `Main` die Methoden `PathInput` und `GetAttributes`. Die Methode `PathInput` liefert eine Zeichenfolge zurück, die den Pfad des Verzeichnisses enthält, dessen Inhalt abgefragt werden soll. Wichtig ist im Kontext des Beispiels, die Rückgabezeichenfolge mit einem Backslash abzuschließen, da ansonsten die spätere Ausgabe an der Konsole nicht immer gleich aussieht.

`GetFileSystemEntries` liefert als Ergebnis des Aufrufs ein `String`-Array, das dem Feld *str* zugewiesen wird.

```
string str = Directory.GetFileSystemEntries(path);
```

Jedes Element des Arrays kann sowohl eine Datei- als auch eine Verzeichnisangabe enthalten. Daher wird in einer `for`-Schleife das Array vom ersten bis zum letzten Element durchlaufen, um festzustellen, ob das Element eine Datei oder ein Verzeichnis beschreibt. Handelt es sich um ein Verzeichnis, ist das Attribut `Directory` gesetzt. Mit

```
if(0 == (File.GetAttributes(str[i]) & FileAttributes.Directory))
```

wird das geprüft. Die Bedingung liefert `true`, wenn eine Datei vorliegt.

Nun folgt ein ganz entscheidender Punkt. Da das Programm die Größe der Datei ausgeben soll, können wir nicht mit `File` arbeiten, da in dieser Klasse keine Methode vorgesehen ist, die uns die Länge der Datei liefert. Dies ist nur über eine Instanz der Klasse `FileInfo` mit der Auswertung der schreibgeschützten Eigenschaft `Length` möglich, die bei jedem Schleifendurchlauf auf eine andere Datei verweist.

Die benutzerdefinierte Methode `GetAttributes` dient dazu, das übergebene `FileInfo`-Objekt auf die Attribute `Hidden`, `ReadOnly`, `Archive` und `System` hin zu untersuchen. Aus dem Ergebnis wird eine Zeichenfolge zusammengesetzt, die den Anforderungen der Anwendung entspricht. Zum Schluss erfolgt noch die formatierte Ausgabe an der Konsole. Für den Verzeichnis- bzw. Dateinamen ist eine maximale Breite von 30 Zeichen vorgesehen. Ist dieser Wert größer, ist das Ergebnis eine zwar etwas unansehnliche Ausgabe, aber unseren Ansprüchen soll sie genügen.

12.3.4 Die Klasse »Path«

Eine Pfadangabe beschreibt den Speicherort einer Datei oder eines Verzeichnisses. Die Schreibweise der Pfadangabe wird vom Betriebssystem vorgegeben und ist nicht auf allen Plattformen zwangsläufig identisch. Bei manchen Systemen muss die Pfadangabe mit dem Laufwerksbuchstaben beginnen, bei anderen Systemen ist das nicht unbedingt vorgeschrieben. Pfadangaben können sich auch auf Dateien beziehen. Es gibt Systeme, die ermöglichen als Dateierweiterung zur Beschreibung des Dateityps maximal drei Buchstaben, während andere durchaus mehr zulassen.

Das sind nicht die einzigen Unterscheidungsmerkmale, die plattformspezifisch sind. Denken Sie nur an die Separatoren, mit denen zwei Verzeichnisse oder ein Verzeichnis von einer Datei getrennt werden. Windows-basierte Plattformen benutzen dazu das Zeichen Backslash (\), andere Systeme schreiben einen einfachen Slash (/) vor.

Methoden der Klasse »Path«

Alle Mitglieder der Klasse `Path` sind statisch und haben die Aufgabe, eine Pfadangabe in einer bestimmten Weise zu filtern. Sie benötigen daher keine Instanz der Klasse `Path`, um auf ein Feld oder eine Methode dieser Klasse zugreifen zu können.

Methode	Beschreibung
`GetDirectoryName`	Liefert aus einer gegebenen Pfadangabe das Verzeichnis zurück.
`GetExtension`	Liefert aus einer gegebenen Pfadangabe die Dateierweiterung einschließlich des führenden Punktes zurück.
`GetFileName`	Liefert den vollständigen Dateinamen zurück.

Tabelle 12.8 Methoden der Klasse »Path«

Methode	Beschreibung
GetFileNameWithoutExtension	Liefert den Dateinamen ohne Dateierweiterung zurück.
GetFullPath	Liefert die komplette Pfadangabe zurück.
GetPathRoot	Liefert das Stammverzeichnis.

Tabelle 12.8 Methoden der Klasse »Path« (Forts.)

Beachten Sie dabei, dass keine dieser Methoden testet, ob die Datei oder das Verzeichnis tatsächlich existiert. Es werden lediglich die Zeichenkette und die Vorschriften der spezifischen Plattform zur Bestimmung des Ergebnisses herangezogen.

Mit

```
string strPath = @"C:\Windows\system32\kernel32.dll"
```

liefern die Methoden die folgenden Rückgaben:

```
// liefert C:\
Console.WriteLine(Path.GetPathRoot(strPath));
// liefert C:\Windows\system32
Console.WriteLine(Path.GetDirectoryName(strPath));
// liefert kernel32
Console.WriteLine(Path.GetFileNameWithoutExtension(strPath));
// liefert kernel32.dll
Console.WriteLine(Path.GetFileName(strPath));
// liefert C:\windows\system32\kernel32.dll
Console.WriteLine(Path.GetFullPath(strPath));
// liefert .dll
Console.WriteLine(Path.GetExtension(strPath));
```

Temporäre Verzeichnisse

Sehr viele Anwendungen arbeiten mit Verzeichnissen, in die Dateien temporär, also nicht dauerhaft geschrieben werden. Die Klasse `Path` bietet mit `GetTempPath` eine Methode an, die das temporäre Verzeichnis des aktuell angemeldeten Benutzers liefert.

```
public static string GetTempPath()
```

Unter Windows 7 ist dieses Verzeichnis standardmäßig unter dem Namen *Temp* in

```
C:\Users\<Username>\AppData\Local\Temp\
```

zu finden.

Mit `GetTempFileName` wird eine leere Datei im temporären Verzeichnis angelegt, der Rückgabewert ist die komplette Pfadangabe:

```
public static string GetTempFileName()
```

Eine temporäre Datei kann von den anderen Methoden dazu benutzt werden, Zwischenergebnisse zu speichern, Informationen kurzfristig zu sichern und Abläufe zu protokollieren. Allerdings sollten Sie nicht vergessen, temporäre Dateien auch wieder zu löschen, wenn sie nicht mehr benötigt werden.

12.3.5 Die Klasse »DriveInfo«

Mit `DriveInfo` können Sie herausfinden, welche Laufwerke verfügbar sind und um welchen Typ von Laufwerk es sich dabei handelt. Zudem können Sie mit Hilfe einer Abfrage die Kapazität und den verfügbaren freien Speicherplatz auf dem Laufwerk ermitteln.

Eigenschaft	Rückgabetyp	Beschreibung
`AvailableFreeSpace`	`long`	Gibt die Menge an verfügbarem freiem Speicherplatz auf einem Laufwerk an.
`DriveFormat`	`string`	Ruft den Namen des Dateisystems ab.
`DriveType`	`DriveType`	Ruft den Laufwerkstyp ab.
`IsReady`	`bool`	Der Rückgabewert gibt an, ob das Laufwerk bereit ist.
`Name`	`string`	Liefert den Namen des Laufwerks.
`RootDirectory`	`DirectoryInfo`	Liefert das Stammverzeichnis des Laufwerks.
`TotalFreeSpace`	`long`	Liefert den verfügbaren Speicherplatz.
`TotalSize`	`long`	Ruft die Gesamtgröße des Speicherplatzes auf einem Laufwerk ab.
`VolumeLabel`	`string`	Ruft die Datenträgerbezeichnung eines Laufwerks ab.

Tabelle 12.9 Eigenschaften der Klasse »DriveInfo«

Die Eigenschaft `DriveType` sollten wir uns noch etwas genauer ansehen. Sie liefert als Ergebnis des Aufrufs eine Konstante der gleichnamigen Enumeration ab. Diese hat insgesamt sieben Mitglieder, die Sie Tabelle 12.10 entnehmen können.

Member	Beschreibung
`CDRom`	Optischer Datenträger (z. B. CD oder DVD)
`Fixed`	Festplatte
`Network`	Netzlaufwerk
`NoRootDirectory`	Das Laufwerk hat kein Stammverzeichnis.

Tabelle 12.10 Mitglieder der Enumeration »DriveType«

Member	Beschreibung
`Ram`	RAM-Datenträger
`Removable`	Wechseldatenträger
`Unknown`	Unbekannter Laufwerkstyp

Tabelle 12.10 Mitglieder der Enumeration »DriveType« (Forts.)

12.4 Die »Stream«-Klassen

Ein Stream ist die abstrahierte Darstellung eines Datenflusses aus einer geordneten Abfolge von Bytes. Welcher Natur dieser Datenstrom ist – ob er aus einer Datei stammt, ob er die Eingabe eines Benutzers an der Tastatur enthält oder ob er möglicherweise aus einer Netzwerkverbindung bezogen wird – bleibt zunächst einmal offen. Die Beschaffenheit des Datenflusses hängt nicht nur von Sender und Empfänger ab, sondern auch ganz entscheidend vom Betriebssystem.

Ein Entwickler soll seine Aufgabe unabhängig von diesen spezifischen Details lösen. E/A-Streams werden deshalb von Klassen beschrieben, die Allgemeingültigkeit garantieren. Das spielt insbesondere bei der Entwicklung von .NET-Anwendungen eine wesentliche Rolle, um die Plattformunabhängigkeit des Codes zu gewährleisten.

Streams dienen generell dazu, drei elementare Operationen ausführen zu können:

- Dateninformationen müssen in einen Stream geschrieben werden. Nach welchem Muster das geschieht, wird durch den Typ des Streams vorgegeben.
- Aus dem Datenstrom muss gelesen werden, ansonsten könnte man die Daten nicht weiterverarbeiten. Das Ziel kann unterschiedlich sein: Die Bytes können Variablen oder Arrays zugewiesen werden, sie könnten aber auch in einer Datenbank landen und zur Ausgabe an einem Peripheriegerät wie dem Drucker oder dem Monitor dienen.
- Nicht immer ist es erforderlich, den Datenstrom vom ersten bis zum letzten Byte auszuwerten. Manchmal reicht es aus, erst ab einer bestimmten Position zu lesen. Man spricht dann vom wahlfreien Zugriff.

Nicht alle Datenströme können diese drei Punkte gleichzeitig erfüllen. Beispielsweise unterstützen Datenströme im Netzwerk nicht den wahlfreien Zugriff.

Bei den Streams werden grundsätzlich zwei Typen unterschieden:

- **Base-Streams**, die direkt aus einem Strom Daten lesen oder in diesen hineinschreiben. Diese Vorgänge können z.B. in Dateien, im Hauptspeicher oder in einer Netzwerkverbindung enden.
- **Pass-Through-Streams** ergänzen einen Base-Stream um spezielle Funktionalitäten. So können manche Streams verschlüsselt oder im Hauptspeicher gepuffert werden. Pass-

Through-Streams lassen sich hintereinander in Reihe schalten, um so die Fähigkeiten eines Base-Streams zu erweitern. Auf diese Weise lassen sich sogar individuelle Streams konstruieren.

12.4.1 Die abstrakte Klasse »Stream«

Die Klasse `Stream` ist die abstrakte Basisklasse aller anderen `Stream`-Klassen. Sie stellt alle fundamentalen Eigenschaften und Methoden bereit, die von den abgeleiteten Klassen geerbt werden und letztendlich deren Funktionalität ausmachen.

Die von der Klasse `Stream` abgeleiteten Klassen unterstützen mit ihren Methoden nur Operationen auf Bytesequenzen. Da allein durch eine Bytesequenz noch keine Aussage darüber getroffen ist, welcher Datentyp sich hinter mehreren aufeinanderfolgenden Bytes verbirgt, muss der Inhalt eines solchen Stroms noch interpretiert werden.

Die Eigenschaften der Klasse »Stream«

Streams stellen Schreib-, Lese- und Suchoperationen bereit. Allerdings unterstützt nicht jeder Stream gleichzeitig alle Operationen. Um in einem gegebenen Stream dessen Verhaltensweisen festzustellen, können Sie die Eigenschaften `CanRead`, `CanWrite` und `CanSeek` abfragen, die einen booleschen Wert zurückliefern und damit Auskunft über die Charakteristik dieses `Stream`-Objekts liefern. Die Eigenschaft `Length` liefert die Länge des Streams und `Position` die aktuelle Position innerhalb des Streams. Letztere wird allerdings nur von den Streams bereitgestellt, die auch die Positionierung mit der `Seek`-Methode unterstützen.

Eigenschaft	Beschreibung
`CanRead`	Ruft in einer abgeleiteten Klasse einen Wert ab, der angibt, ob der aktuelle Stream Lesevorgänge unterstützt.
`CanWrite`	Ruft in einer abgeleiteten Klasse einen Wert ab, der angibt, ob der aktuelle Stream Schreibvorgänge unterstützt.
`CanSeek`	Ruft in einer abgeleiteten Klasse einen Wert ab, der angibt, ob der aktuelle Stream Suchvorgänge unterstützt.
`Length`	Ruft in einer abgeleiteten Klasse die Länge des Streams in Byte ab.
`Position`	Ruft in einer abgeleiteten Klasse die Position im aktuellen Stream ab oder legt diese fest.

Tabelle 12.11 Eigenschaften der abstrakten Klasse »Stream«

Die Methoden der Klasse »Stream«

Die wichtigsten Methoden aller `Stream`-Klassen dürften `Read`, `Write` und `Seek` sein. Sehen wir uns zunächst die Definitionen der beiden Methoden `Read` und `Write` an, die von jeder abgeleiteten Klasse überschrieben werden müssen.

```
public abstract int Read(in byte[] buffer,int offset,int count);
public abstract void Write(byte[] buffer, int offset, int count);
```

Einem schreibenden Stream müssen Sie die Daten übergeben, die in den Datenstrom geschrieben werden sollen. Die `Write`-Methode benutzt dazu den ersten Parameter, liest die Elemente byteweise ein und schreibt sie in den Strom. Der Empfänger des Datenstroms kann die Bytes mit `Read` im ersten Parameter entnehmen. Der zweite Parameter *offset* bestimmt die Position im Array, ab der der Lese- bzw. Schreibvorgang beginnen soll. Meistens wird hier die Zahl 0 eingetragen, d.h., die Operation greift auf das erste Array-Element zu – entweder lesend oder schreibend. Im dritten und letzten Parameter wird angegeben, wie viele Bytes gelesen oder geschrieben werden sollen.

Beachten Sie auch, dass `Write` als Methode ohne Rückgabewert implementiert ist, während `Read` einen `int` liefert, dem Sie die Anzahl der gelesenen Bytes entnehmen können, die in den Puffer – also das Array – geschrieben worden sind. Der Rückgabewert ist 0, wenn das Ende des Streams erreicht ist. Er kann aber auch kleiner sein als im dritten Parameter angegeben, wenn weniger Bytes im Stream eingelesen werden.

Die abstrakte Klasse `Stream` definiert zwei weitere, ähnliche Methoden, die jedoch jeweils nur immer ein Byte aus dem Datenstrom lesen oder in diesen hineinschreiben: `ReadByte` und `WriteByte`. Beide Methoden sind parameterlos und setzen den Positionszeiger innerhalb des Streams um eine (Byte-)Position weiter.

```
public virtual int ReadByte();
public virtual void WriteByte(byte value);
```

Der Rückgabewert der `ReadByte`-Methode ist –1, wenn das Ende des Datenstroms erreicht ist.

Um in einem Datenstrom ab einer vorgegebenen Position zu lesen oder zu schreiben, bietet sich die `Seek`-Methode an:

```
public abstract long Seek(long offset, SeekOrigin origin);
```

Mit den beiden Parametern *offset* und *origin* wird der Startpunkt für den Positionszeiger im Stream festgelegt, ab dem weitere E/A-Operationen aktiv werden. *offset* beschreibt die Verschiebung in Bytes ab der unter *origin* festgelegten Ursprungsposition. *origin* ist vom Typ der Aufzählung *SeekOrigin*, in der die drei Konstanten aus Tabelle 12.12 definiert sind.

Member	Beschreibung
`Begin`	Gibt den Anfang eines Streams an.
`Current`	Gibt die aktuelle Position innerhalb eines Streams an.
`End`	Gibt das Ende eines Streams an.

Tabelle 12.12 Konstantenliste der Aufzählung »SeekOrigin«

Mit `SeekOrigin.Begin` wird der Positionszeiger auf das erste Byte des Datenstroms gesetzt, mit `SeekOrigin.Current` behält er seine augenblickliche Position bei, und mit `SeekOrigin.End` wird er auf das Byte gesetzt, das als Erstes den Bytes des vollständigen Streams folgt. Ausgehend von *origin* wird durch Addition von *offset* die gewünschte Startposition ermittelt.

Ein Stream, der einmal geöffnet worden ist und Daten in den Puffer geschrieben hat, sollte ordnungsgemäß mit `Close` geschlossen werden.

Sie haben jetzt so viele Methoden im Schnelldurchlauf kennengelernt, dass alle erwähnten noch einmal in übersichtlicher tabellarischer Form zusammengefasst werden sollen (siehe Tabelle 12.13).

Methode	Beschreibung
`Close`	Schließt den aktuellen Stream und gibt alle dem aktuellen Stream zugeordneten Ressourcen frei.
`Read`	Liest eine Folge von Bytes aus dem aktuellen Stream und setzt den Datenzeiger im Stream um die Anzahl der gelesenen Bytes weiter.
`ReadByte`	Liest ein Byte aus dem Stream und erhöht die Position im Stream um ein Byte. Der Rückgabewert ist –1, wenn das Ende des Streams erreicht ist.
`Seek`	Legt die Position im aktuellen Stream fest.
`Write`	Schreibt eine Folge von Bytes in den aktuellen Stream und erhöht den Datenzeiger im Stream um die Anzahl der geschriebenen Bytes.
`WriteByte`	Schreibt ein Byte an die aktuelle Position im Stream und setzt den Datenzeiger um eine Position im Stream weiter.

Tabelle 12.13 Methoden der abstrakten Klasse »Stream«

12.4.2 Die von »Stream« abgeleiteten Klassen im Überblick

Sie haben in den vorherigen Ausführungen nur die wichtigsten Methoden und Eigenschaften der Klasse `Stream` kennengelernt. Die bisherigen Aussagen sollten genügen, um eine Vorstellung davon zu erhalten, welche wesentlichen Verhaltensweisen an die abgeleiteten Klassen weitervererbt werden.

Den in Tabelle 12.14 aufgeführten Klassen dient `Stream` als Basisklasse. Dabei ist die Tabelle nicht vollständig, sondern beinhaltet nur die wichtigsten Typen. Beachten Sie bitte, dass die verschiedenen ableitenden Klassen nicht alle demselben Namespace angehören.

Stream-Typ	Beschreibung
`BufferedStream`	Die Klasse `BufferedStream` wird benutzt, um Daten eines anderen E/A-Datenstroms zu puffern. Ein Puffer ist ein Block von Bytes im Arbeitsspeicher des Rechners, der dazu benutzt wird, den Datenstrom zu cachen, um damit die Anzahl der Aufrufe an das Betriebssystem zu verringern. Dadurch lässt sich insgesamt die Effizienz verbessern. Diese Klasse wird immer im Zusammenhang mit anderen Klassen eingesetzt.
`CryptoStream`	Daten, die nicht in ihrem Originalzustand in einen Strom geschrieben werden sollen, lassen sich mit der Klasse `CryptoStream` verschlüsseln. `CryptoStream` wird immer zusammen mit einem anderen Stream kombiniert.
`FileStream`	Diese Klasse wird dazu benutzt, um Daten in Dateien des Dateisystems zu schreiben. Eine Netzwerkverbindung kann ebenfalls das Ziel dieses Datenstroms sein.
`GZipStream`	Mit den Methoden dieser Klasse können Sie Byteströme komprimieren und dekomprimieren.
`MemoryStream`	Meistens sind Dateien oder Netzwerkverbindungen das Ziel der Datenströme. Es kann jedoch auch sinnvoll sein, Daten bewusst temporär in den Hauptspeicher zu schreiben und sie später von dort wieder zu lesen. Viele Anwendungen arbeiten nach dem Prinzip, Daten in eine temporäre Datei zu speichern. Ein `MemoryStream` kann temporäre Dateien ersetzen und trägt damit zur Steigerung der Leistungsfähigkeit einer Anwendung bei, da das Schreiben und Lesen in den Hauptspeicher um ein Vielfaches schneller ist als das Schreiben auf die Festplatte.
`NetworkStream`	Ein Datenfluss, der auf der Klasse `NetworkStream` basiert, sendet die Daten basierend auf Sockets. Das Besondere an diesem Datenstrom ist, dass er nur die Fähigkeit hat, Daten vollständig in den Strom zu schreiben oder aus diesem zu lesen. Der Zugriff auf beliebige Daten innerhalb des Stroms ist nicht möglich.

Tabelle 12.14 Die von »Stream« abgeleiteten Klassen

12.4.3 Die Klasse »FileStream«

Die Klasse `FileStream` ist die universellste Klasse und erscheint damit in vielen Anwendungsfällen am geeignetsten. Sie hat die Fähigkeit, sowohl byteweise aus einer Datei zu lesen als auch byteweise in eine Datei zu schreiben. Außerdem kann ein Positionszeiger auf eine beliebige Position innerhalb des Streams gesetzt werden. Ein `FileStream` puffert die Daten, um die Ausführungsgeschwindigkeit zu erhöhen. Die Größe des Puffers beträgt standardmäßig 8 KByte.

Die FileStream-Klasse bietet eine Reihe von Konstruktoren an, um dem Objekt bestimmte Verhaltensweisen und Eigenschaften mit auf den Lebensweg zu geben:

```
public FileStream(string, FileMode);
public FileStream(string, FileMode, FileAccess);
public FileStream(string, FileMode, FileAccess, FileShare);
public FileStream(string, FileMode, FileAccess, FileShare, int);
public FileStream(string, FileMode, FileAccess, FileShare, int, bool);
```

Sie können ein FileStream-Objekt erzeugen, indem Sie im ersten Parameter eine Pfadangabe als Zeichenfolge übergeben. Der Parameter FileMode beschreibt, wie das Betriebssystem die Datei öffnen soll (FileMode.Append, FileMode.Create, FileMode.CreateNew ...). FileAccess hingegen gibt an, wie auf die Datei zugegriffen werden darf (FileAccess.Read, FileAccess.Write oder FileAccess.ReadWrite). Sie haben diese Typen bereits im Abschnitt zur Klasse File kennengelernt (siehe auch die Tabellen 12.2 und 12.3). Der Parameter vom Typ FileShare gibt an, ob ein gemeinsamer Zugriff auf die Datei möglich ist oder nicht (siehe auch Tabelle 12.4).

Der Puffer, in den ein FileStream die Daten zur Steigerung der Leistungsfähigkeit schreibt, ist standardmäßig 8 KByte groß. Mit dem Parameter vom Typ int können Sie die Größe des Puffers bei der Instanziierung beeinflussen. Mit dem letzten Parameter vom Typ bool kann noch angegeben werden, ob das Objekt asynchrone Zugriffe unterstützen soll.

Das Schreiben in einen »FileStream«

Das folgende Listing demonstriert, wie mit einem FileStream-Objekt Daten in eine Datei geschrieben werden:

```
static void Main(string[] args) {
  byte[] arr = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
  string path = @"D:\Testfile.txt";
  FileStream fs = new FileStream(path, FileMode.Create);
  fs.Write(arr, 0, arr.Length);
  fs.Close();
}
```

Listing 12.4 Schreiben in eine Datei mit einem »FileStream«-Objekt

Zunächst wird ein Byte-Array deklariert und mit insgesamt zehn Zahlen initialisiert. In der zweiten Anweisung wird der Name der Datei festgelegt, in die das Array geschrieben werden soll.

Bei der Instanziierung des FileStream-Objekts werden dem Konstruktor im ersten Argument der Pfad und die Datei bekannt gegeben, auf der der Stream operieren soll. Die Fähigkeiten dieses Streams beschreibt das zweite Argument: Die Konstante FileMode.Create teilt dem Konstruktor mit, dass das FileStream-Objekt eine neue Datei erzeugen kann oder, falls im angegebenen Pfad bereits eine gleichnamige Datei existiert, diese überschreiben soll. Mit

```
fs.Write(arr, 0, arr.Length);
```

wird der Inhalt des Arrays *arr* dem `Stream`-Objekt übergeben. Die Syntax der Methode `Write` der Klasse `FileStream` lautet wie folgt:

```
public void Write(byte[] array, int offset, int count);
```

Dabei haben die drei Parameter die folgende Bedeutung:

Parameter	Beschreibung
array	Ein Byte-Array, in das die übergebenen Daten gelesen werden
offset	Die Indexposition im Array, an der die Leseoperation beginnen soll
count	Die Anzahl der zu lesenden Bytes

Tabelle 12.15 Die Parameter der Methode »Write« eines »FileStream«-Objekts

Der Schreibvorgang des Beispiels startet mit dem ersten Array-Element. Das sagt der zweite Parameter der `Write`-Methode aus. Die Anzahl der zu schreibenden Bytes bestimmt der dritte Parameter – in unserem Beispiel werden alle Array-Elemente dem Datenstrom zugeführt. Zum Schluss wird der `FileStream` mit der Methode `Close` geschlossen.

Das Lesen aus einem »FileStream«

Wir wollen uns nun auch vom Erfolg unserer Bemühungen überzeugen und die Datei auswerten. Dazu wird der Programmcode des Listings 12.4 wie folgt ergänzt:

```
static void Main(string[] args) {
  [...]
  byte[] arrRead = new byte[10];
  fs.Read(arrRead, 0, 10);
  for (int i = 0; i < arr.Length; i++)
    Console.WriteLine(arrRead[i]);
  fs.Close();
}
```

Listing 12.5 Lesen einer Datei mit einem »FileStream«-Objekt

Wir deklarieren ein weiteres Array (*arrRead*), in das wir das Ergebnis der Leseoperation hineinschreiben. Da uns bekannt ist, wie viele Byte-Elemente sich in unserer Datei befinden (wie unfair), können wir die Array-Grenze schon im Voraus festlegen.

Nun kommt es zum Aufruf der `Read`-Methode. Zuerst wollen wir uns wieder die Syntax dieser Methode anschauen:

```
public override int Read(in byte[] array, int offset, int count);
```

Die Parameter sind denen der `Write`-Methode sehr ähnlich. Das `FileStream`-Objekt, auf dem die `Read`-Methode aufgerufen wird, repräsentiert eine bestimmte Datei. Diese wurde bereits

über den Konstruktor bekannt gegeben. Aus der Datei werden die Daten in das Array eingelesen, das durch den Parameter *array* beschrieben wird. Der erste Array-Index, der der Schreiboperation zur Verfügung steht, wird im Parameter *offset* angegeben, die Anzahl der aus dem `FileStream` zu lesenden Bytes im dritten Parameter *count*.

Wir wollen den ersten Byte-Wert aus dem Datenstrom in das mit 0 indizierte Element des Arrays *arrRead* schreiben und geben das im zweiten Parameter bekannt. Die Gesamtanzahl der zu lesenden Bytes teilen wir dem dritten Parameter mit. In einer Schleife wird danach das eingelesene Array durchlaufen und an der Konsole ausgegeben.

Starten Sie nun die Laufzeit des Programms, wird die Enttäuschung groß sein und Sie an den eigenen Programmierfähigkeiten zweifeln lassen! Denn bedauerlicherweise werden nicht die Zahlenwerte ausgegeben, die wir in die Datei geschrieben haben, sondern nur Nullen. Haben wir etwas falsch gemacht, und wenn ja, wie ist das zu erklären?

Der Positionszeiger

Die Schreib- bzw. Leseposition in einem Datenstrom wird durch einen Positionszeiger beschrieben. Schließlich muss der Strom wissen, auf welchem Byte die folgende Operation ausgeführt werden soll. Bei der Instanziierung eines `Stream`-Objekts verweist der Zeiger zunächst auf das erste Byte im Stream. Mit dem Aufruf der `Write`-Methode wird ein Wert daher genau an diese Position geschrieben. Anschließend wird der Positionszeiger auf die folgende Byte-Position verschoben.

Dieser Vorgang wiederholt sich bei jedem Schreibvorgang, von denen es in unserem Beispiel zehn gibt, nämlich für jedes zu schreibende Array-Element einen. Am Ende, wenn wir unsere zehn Bytes in den Strom geschrieben haben, verweist der Positionszeiger auf das folgende, nun elfte Byte im Stream (siehe Abbildung 12.3).

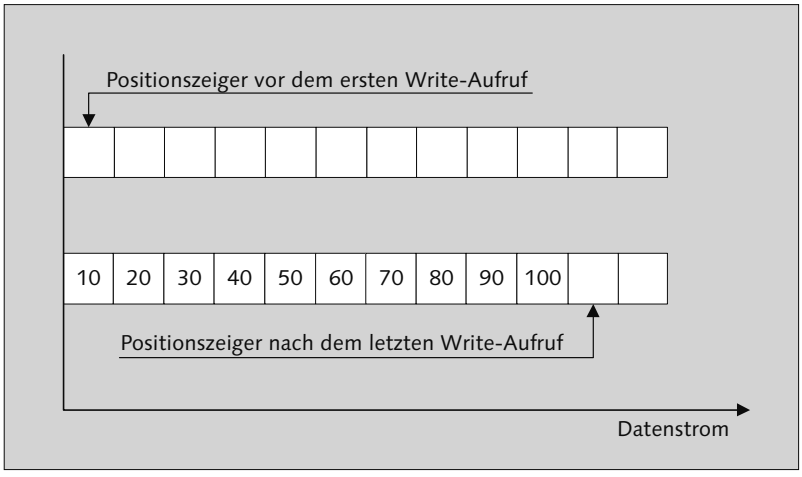


Abbildung 12.3 Der Positionszeiger in einem »Stream«-Objekt

Genau das verursacht nun ein Problem. Wir rufen auf dem `FileStream`-Objekt die `Read`-Methode auf und lesen ab der Position, die aktuell durch den Datenzeiger beschrieben wird. Das ist aber die elfte Stelle im Datenstrom – und nicht die erste, wie wir es eigentlich erwartet haben bzw. wie es hätte sein sollen.

Kommen wir nun zur Lösung. `FileStream` beerbt die Klasse `Stream` und hat daher auch eine `Seek`-Methode, mit der wir den Positionszeiger beliebig einstellen können:

```
public override long Seek(long offset, SeekOrigin origin);
```

Wir überlegen uns, wohin wir den Ursprung des Positionszeigers verlegen wollen – natürlich an den Anfang des Datenstroms. Also muss der zweite Parameter der `Seek`-Methode auf

```
origin = SeekOrigin.Begin
```

festgelegt werden (siehe dazu auch Tabelle 12.13). Nun geben wir im ersten Argument den tatsächlichen und endgültigen Startpunkt des Positionszeigers bezogen auf den im zweiten Parameter definierten Ursprung an. Er lautet 0, denn schließlich wollen wir den Zeiger auf die erste Position des Datenstroms setzen.

```
[...]
fs.Write(arr, 0, arr.Length);
[...]
fs.Seek(0, SeekOrigin.Begin);
fs.Read(arrRead, 0, 10);
[...]
```

Natürlich wäre es auch möglich, zunächst das `FileStream`-Objekt zu schließen und es danach neu zu instanziieren. Damit hätten wir wieder einen Positionszeiger, der auf das erste Byte im Stream zeigt.

Wenn wir nach der Ergänzung mit `Seek` das Programm noch einmal starten, wird das Ergebnis wie erwartet ausgegeben.

Zum Schluss wollen wir noch einmal den gesamten Code zusammenfassen.

```
// Beispiel: ..\Kapitel 12\FileStreamSample
class Program {
  static void Main(string[] args) {
    byte[] arr = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
    string path = "D:\\Testfile.txt";
    // Stream öffnen
    FileStream fs = new FileStream(path, FileMode.Create);
    // in den Stream schreiben
    fs.Write(arr, 0, arr.Length);
    byte[] arrRead = new byte[10];
    // Positionszeiger auf den Anfang des Streams setzen
    fs.Seek(0, SeekOrigin.Begin);
    // Stream lesen
```

```
    fs.Read(arrRead, 0, 10);
    for (int i = 0; i < arr.Length; i++)
      Console.WriteLine(arrRead[i]);
    Console.ReadLine();
    // FileStream schließen
    fs.Close();
  }
}
```

Listing 12.6 Das komplette Beispiel des Einsatzes der Klasse »FileStream«

Wir müssen nicht zwangsläufig das komplette Byte-Array vom ersten bis zum letzten Element in die Datei schreiben. Mit

```
fs.Write(arr, 2, arr.Length - 2);
```

ist das erste zu schreibende Element dasjenige, das die Zahl 30 enthält (entsprechend dem Index 2). Wenn wir allerdings den dritten Parameter, der die Anzahl der zu lesenden Bytes angibt, nicht entsprechend anpassen, wird über das Ende des Arrays hinaus gelesen, was zu der Exception `ArgumentException` führt.

Eine Textdatei mit »FileStream« lesen

Obwohl es spezialisierte Klassen zum Lesen und Schreiben in eine Textdatei gibt, lässt sich das auch mit einem `FileStream`-Objekt realisieren. Wir wollen uns das im Folgenden ansehen:

```
// Beispiel: ..\Kapitel 12\TextdateiMitFileStream
class Program {
  static void Main(string[] args) {
    // Benutzereingabe anfordern
    Console.Write("Geben Sie die zu öffnende Datei an: ");;
    string strFile = Console.ReadLine();
    // prüfen, ob die angegebene Datei existiert
    if (! File.Exists(strFile)) {
      Console.WriteLine("Die Datei {0} existiert nicht!", strFile);
      Console.ReadLine();
      return;
    }
    // Datei öffnen
    FileStream fs = File.Open(strFile, FileMode.Open);
    // Byte-Array, in das die Daten aus dem Datenstrom eingelesen werden
    byte[] puffer = new Byte[fs.Length];
    // die Zeichen aus der Datei lesen und in das Array
    // schreiben, der Lesevorgang beginnt mit dem ersten Zeichen
    fs.Read(puffer, 0, (int)fs.Length);
    // das Byte-Array elementweise einlesen und jedes Array-Element
```

```
    // in Char konvertieren
    for (int i = 0; i < fs.Length; i++)
      Console.Write(Convert.ToChar(puffer[i]));
    Console.ReadLine();
  }
}
```

Listing 12.7 Textdatei mit einem »FileStream«-Objekt einlesen

Nach dem Start der Laufzeitumgebung wird der Benutzer dazu aufgefordert, den Pfad zu einer Textdatei anzugeben. Diesmal beschreiten wir allerdings einen anderen Weg und rufen den Konstruktor nicht direkt auf, sondern die Methode `Open` der Klasse `File`:

```
FileStream fs = File.Open(strFile, FileMode.Open);
```

Diese Anweisung funktioniert tadellos, weil der Rückgabewert der `File.Open`-Methode die Referenz auf eine `FileStream`-Instanz liefert. Gegen den Weg über einen `FileStream`-Konstruktor, der diese Möglichkeit auch bietet, ist grundsätzlich auch nichts einzuwenden. Im folgenden Schritt wird das Byte-Array *puffer* deklariert und mit einer Kapazität initialisiert, die der Größe der Datei entspricht:

```
byte[] puffer = new Byte[fs.Length];
```

Die Größe der Datei besorgen wir uns mit der Eigenschaft `Length` der Klasse `FileStream`, die uns die Größe des Datenstroms liefert. Daran schließt sich die Leseoperation mit `Read` an, die den Inhalt der Textdatei byteweise liest und in das Array *puffer* schreibt:

```
fs.Read(puffer, 0, (int)fs.Length);
```

Weil der dritte Parameter der `Read`-Methode ein Datum vom Typ `int` erwartet und die Eigenschaft `Length` einen `long` liefert, müssen wir noch in den richtigen Datentyp konvertieren.

Ein `FileStream`-Objekt arbeitet grundsätzlich nur auf der Basis von Bytes, es weiß nichts von dem tatsächlichen Typ, der sich im Datenstrom verbirgt. Eine Textdatei enthält aber Zeichen, die, als ANSI-Zeichen interpretiert, erst den wirklichen Informationsgehalt liefern. Deshalb müssen wir jedes einzelne Byte des Streams in einen `char`-Typ konvertieren. Das geschieht in einer Schleife, die alle Bytes des Arrays abgreift, konvertiert und danach an der Konsole ausgibt.

```
for (int i = 0; i < fs.Length; i++)
  Console.Write(Convert.ToChar(puffer[i]));
```

Würde den Daten aus der Datei ein anderer Typ zugrunde liegen, beispielsweise `int` oder `float`, müsste dieser Zieldatentyp angegeben werden. Wissen Sie nicht, welcher Typ in der Datei gespeichert ist, können Sie mit dem Inhalt praktisch nichts anfangen.

12.5 Die Klassen »TextReader« und »TextWriter«

Wie Sie in den vorhergehenden Abschnitten gesehen haben, stellt die Klasse `Stream` Operationen bereit, mit denen Sie unformatierte Daten byteweise lesen und schreiben können. `Stream`-Objekte bieten sich daher insbesondere für allgemeine Operationen an, beispielsweise für das Kopieren von Dateien. Die Klasse `Stream` beziehungsweise die daraus abgeleiteten Klassen sind aber weniger gut für textuelle Ein- und Ausgabeoperationen geeignet.

Um den üblichen Anforderungen von Textoperationen zu entsprechen, stellt die .NET-Klassenbibliothek die beiden abstrakten Klassen `TextReader` und `TextWriter` bereit. Objekte, die aus der Klasse `Stream` abgeleitet werden, unterstützen den vollständigen Satz an E/A-Operationen, also sowohl das Lesen als auch das Schreiben. Nun wird die Bearbeitung auf zwei Klassen aufgeteilt, die entweder nur lesen oder nur schreiben können.

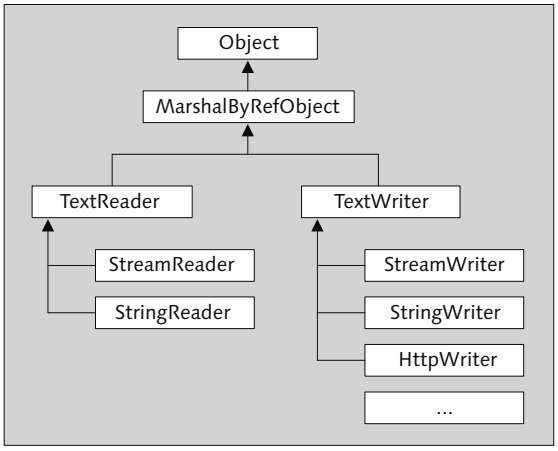


Abbildung 12.4 Objekthierarchie der »Reader«- und »Writer«-Klassen

`TextReader` und `TextWriter` sind abstrakt definiert und müssen daher abgeleitet werden. Das .NET Framework bietet solche Ableitungen mit `StreamReader` und `-Writer` sowie `StringReader` und `-Writer` an. Von `TextWriter` gibt es auch noch weitere, spezialisierte Ableitungen. Im Folgenden werden wir uns mit den Klassen `StreamReader` und `StreamWriter` beschäftigen.

12.5.1 Die Klasse »StreamWriter«

Die Konstruktoren der Klasse »StreamWriter«

Wir werden uns daher zunächst einigen Konstruktoren der Klasse `StreamWriter` zuwenden, um zu sehen, auf welcher Basis sich ein Objekt dieses Typs erzeugen lässt.

```
public StreamWriter(Stream);
public StreamWriter(string);
public StreamWriter(Stream, Encoding);
```

```
public StreamWriter(string, bool);
public StreamWriter(Stream, Encoding, int);
public StreamWriter(string, bool, Encoding);
public StreamWriter(string, bool, Encoding, int);
```

Es fällt zunächst auf, dass wir jedem Konstruktor entweder eine Zeichenfolge oder ein Objekt vom Typ `Stream` übergeben müssen. Entscheiden wir uns für eine Zeichenfolge, enthält diese die Pfadangabe zu einer Datei.

Da die Klasse `Stream` abstrakt ist, können wir natürlich keine Referenz auf ein konkretes `Stream`-Objekt übergeben. Aber die Klasse `Stream` wird abgeleitet, beispielsweise von `FileStream`. Die Referenz auf ein Objekt einer aus `Stream` abgeleiteten Klasse gilt aber nach den Paradigmen der Objektorientierung gleichzeitig als ein Objekt vom Typ der Basisklasse. Also kann dem Parameter im Konstruktor, der den Typ `Stream` erwartet, ein Objekt vom Typ einer aus `Stream` abgeleiteten Klasse übergeben werden.

Nun sehen wir uns natürlich sofort mit der Frage konfrontiert, welchen Sinn es hat, ein `Stream`-Objekt als Argument an den Konstruktor zu übergeben. Wie Sie sich vielleicht noch erinnern, werden die `Stream`-Objekte generell in zwei Typen klassifiziert: in Base-Streams und Pass-Through-Streams. Ein Base-Stream endet zum Beispiel direkt in einer Datei oder in einer Netzwerkverbindung, ein Pass-Through-Stream ist ein »Durchlaufobjekt«, das die Fähigkeiten eines Base-Streams erweitert.

Betrachten wir zunächst den Konstruktor der Klasse `StreamWriter`, der in einem String eine Pfadangabe entgegennimmt:

```
public StreamWriter(string);
```

Ein Objekt, das basierend auf dieser Erstellungsroutine instanziiert wird, weiß, wohin die Daten geschrieben werden – nämlich in die Datei, die durch das `String`-Argument beschrieben wird, z.B.:

```
StreamWriter myStreamWriter = new StreamWriter(@"D:\MyText.txt");
```

Wir erzeugen mit dieser Anweisung einen Base-Stream, der die Daten – genauer gesagt eine Zeichenfolge – in eine Datei schreiben kann. Nun wollen wir ein anderes `StreamWriter`-Objekt erzeugen, diesmal allerdings auf Basis der Übergabe eines `FileStream`-Objekts.

```
FileStream fs = new FileStream(@"D:\Test.txt", FileMode.CreateNew);
StreamWriter myStreamWriter = new StreamWriter(fs);
```

In der ersten Anweisung wird ein Objekt vom Typ `FileStream` erstellt, das eine neue Datei namens *Test.txt* in der Root *D:* erzeugt. Dieses Objekt wird seinerseits als Argument an den Konstruktor der Klasse `StreamWriter` übergeben. Als Resultat liegt eine Hintereinanderschaltung von zwei `Stream`-Objekten vor, woraus sich Nutzen ziehen lässt. Wie Sie wissen, schreiben und lesen Objekte, die auf der `Stream`-Klasse basieren, nur elementare Bytes. Demgegenüber schreiben `StreamWriter`-Objekte Zeichen mit einer speziellen Verschlüsse-

lung (Encoding) in den Datenstrom. Sie arbeiten im Endeffekt mit einem Datenstrom, der die Charakteristika beider Datenflüsse kombiniert. In ähnlicher Weise könnten Sie natürlich auch einen `MemoryStream` oder `NetworkStream` als Argument übergeben.

Standardmäßig verschlüsselt `StreamWriter` nach UTF-8, eine Abweichung davon wird durch die Wahl eines Konstruktors erreicht, der einen Parameter vom Typ `Encoding` aus dem Namespace `System.Text` entgegennimmt. Sie können hier beispielsweise ein Objekt vom Typ `UTF7Encoding` oder `UnicodeEncoding` (entspricht der UTF-16-Kodierung) übergeben.

Anmerkung

Schreiben wir Zeichen in einen Stream, müssen die Bytes in bestimmter Weise interpretierbar sein. Standardmäßig wird in Mitteleuropa zur Kodierung der ANSI-Zeichensatz (Codeseite 1252) benutzt, der Zeichencodes zwischen 0 und 255 zulässt und unter anderem auch Sonderzeichen wie »ä«, »ö« und »ü« beschreibt. Damit unterscheidet sich der ANSI-Zeichensatz vom ASCII-Zeichensatz, der nur die Codes von 0 bis 127 festlegt. Um einen Text korrekt zu übertragen und anzuzeigen, dürfte streng genommen nur der ASCII-Zeichensatz verwendet werden, weil nur die Codes 0–127 unter ANSI und ASCII identisch sind.

Um Probleme dieser Art zu vermeiden, wurde mit Unicode ein neuer Zeichensatz geschaffen. Allerdings hat auch Unicode unterschiedliche Formate, denn es wird zwischen UTF-7, UTF-8, UTF-16 und UTF-32 unterschieden. Der UTF-8-Zeichensatz ist wohl der wichtigste, denn er ist der Standard unter .NET. In diesem Zeichensatz werden Unicode-Zeichen in einer unterschiedlichen Anzahl Bytes verschlüsselt. Die ASCII-Zeichen werden in einem Byte gespeichert, alle anderen Zeichen in weiteren zwei bis vier Byte. Das hat den Vorteil, dass Systeme, die nur ASCII- oder ANSI-Zeichen verarbeiten, mit der UTF-8-Kodierung klarkommen.

Einige Konstruktoren erwarten zusätzlich einen booleschen Wert. Dieser kommt nur im Zusammenhang mit den Konstruktoren vor, die in einer Zeichenfolge die Pfadangabe zu der Datei erhalten, in die der Datenstrom geschrieben werden soll. Mit `true` werden die zu schreibenden Daten an das Ende der Datei gehängt – vorausgesetzt, es existiert bereits eine Datei gleichen Namens in dem Verzeichnis. Mit der Übergabe von `false` wird eine existierende Datei überschrieben.

Der letzte Parameter, der Ihnen in zwei Konstruktoren zur Verfügung steht, empfängt einen Wert vom Typ `int`, mit dem Sie die Größe des Puffers beeinflussen können.

Das Schreiben in den Datenstrom

Schauen wir uns zunächst ein Codefragment an, mit dem wir eine Datei erzeugen, in die wir den obligatorischen Text »Visual C# macht Spaß« schreiben:

```
StreamWriter sw = new StreamWriter(@"D:\NewFile.txt");
sw.WriteLine("Visual C#");
```

```
sw.WriteLine("macht Spaß!");
sw.Close();
```

Listing 12.8 Mit »StreamWriter« in eine Textdatei schreiben

Einfacher geht es nicht mehr! Zunächst wird ein Konstruktor aufgerufen und diesem zur Initialisierung des `StreamWriter`-Objekts eine Zeichenkette als Pfadangabe übergeben. Daraufhin wird entweder die Datei erzeugt oder eine existierende gleichnamige Datei im angegebenen Verzeichnis überschrieben. Mit jedem Aufruf der von `TextWriter` geerbten Methode `WriteLine` wird eine Zeile in die Datei geschrieben und ihr am Ende ein Zeilenumbruch angehängt. Mit unserem Codefragment erzeugen wir also eine zweizeilige Textdatei.

Es liegt die Vermutung nahe, dass `StreamWriter` eine zweite Methode zum Schreiben in den Datenstrom bereitstellt, die ohne den automatisch angehängten Zeilenumbruch in den Strom schreibt. Ein Blick in die Klassenbibliothek bestätigt die Vermutung: Es gibt eine Methode `Write`. Diese Methode ist genauso überladen wie die Methode `WriteLine`. `Write` und `WriteLine` bilden den Kern der Klasse `StreamWriter`. Viel mehr Methoden hat die Klasse auch nicht anzubieten, denn alle anderen sind bereits gute Bekannte: `Close`, um einen auf dieser Klasse basierenden Strom zu schließen, und `Flush`, um die im Puffer befindlichen Daten in den Strom zu schreiben und den Puffer zu leeren. Tabelle 12.16 gibt die wichtigsten Methoden eines `StreamWriter`-Objekts wieder.

Methode	Beschreibung
`Close`	Schließt das aktuelle Objekt sowie alle eingebetteten Streams.
`Flush`	Schreibt die gepufferten Daten in den Stream und löscht danach den Inhalt des Puffers.
`Write`	Schreibt in den Stream, ohne einen Zeilenumbruch anzuhängen.
`WriteLine`	Schreibt in den Stream und schließt mit einem Zeilenumbruch ab.

Tabelle 12.16 Methoden eines »StreamWriter«-Objekts

Die Eigenschaften der Klasse »StreamWriter«

Mit `AutoFlush` veranlassen Sie, dass Daten aus dem Puffer in den Datenstrom geschrieben werden, sobald eine der `Write`/`WriteLine`-Methoden aufgerufen wird und diese Eigenschaft auf `true` gesetzt ist. Wollen Sie das aktuelle Textformat erfahren, können Sie die Eigenschaft `Encoding` auswerten:

```
StreamWriter sw = new StreamWriter(@"C:\NewFile.txt", false, Encoding.Unicode);
Console.WriteLine("Format: {0}", sw.Encoding.ToString());
```

Als dritte und letzte Eigenschaft steht Ihnen noch `BaseStream` zur Verfügung, die das Objekt des Base-Streams liefert, auf dem das `StreamWriter`-Objekt basiert.

Eigenschaften	Beschreibung
AutoFlush	Löscht den Puffer nach jedem Aufruf von Write oder WriteLine.
BaseStream	Liefert eine Referenz auf den Base-Stream zurück.
Encoding	Liefert das aktuelle Encoding-Schema zurück.

Tabelle 12.17 Die Eigenschaften der Klasse »StreamWriter«

12.5.2 Die Klasse »StreamReader«

Die aus der Klasse TextReader abgeleitete Klasse StreamReader ist das Gegenstück zur Klasse StreamWriter. Betrachtet man ihre Möglichkeiten, sind die Klassen praktisch identisch – abgesehen von der Tatsache, dass das charakteristische Merkmal dieser Klasse in der Fähigkeit zu finden ist, Daten einer bestimmten Kodierung aus einem Strom zu lesen.

Die Konstruktoren ähneln denen der Klasse StreamWriter. Sie nehmen im einfachsten Fall die Referenz auf einen Stream oder eine Pfadangabe als String entgegen. Sie gestatten aber auch, die eingelesenen Zeichen nach einem durch Encoding beschriebenen Schema zu interpretieren oder die Puffergröße zu variieren. Tabelle 12.18 enthält die wichtigsten Methoden eines StreamReaders.

Methode	Beschreibung
Peek	Liest ein Zeichen aus dem Strom und liefert den int-Wert zurück, der das Zeichen repräsentiert, verarbeitet das Zeichen aber nicht. Der Zeiger wird nicht auf die Position des folgenden Zeichens gesetzt, wenn Peek aufgerufen wird, sondern verbleibt in seiner Stellung. Verweist der Zeiger hinter den Datenstrom, ist der Rückgabewert –1.
Read	Liest ein oder mehrere Zeichen aus dem Strom und liefert den int-Wert zurück, der das Zeichen repräsentiert. Ist kein Zeichen mehr verfügbar, ist der Rückgabewert –1. Der Positionszeiger verweist auf das nächste zu lesende Zeichen. Eine zweite Variante dieser überladenen Methode liefert die Anzahl der eingelesenen Zeichen.
ReadLine	Liest eine Zeile aus dem Datenstrom – entweder bis zum Zeilenumbruch oder bis zum Ende des Stroms. Der Rückgabewert ist vom Typ string.
ReadToEnd	Liest von der aktuellen Position des Positionszeigers bis zum Ende des Stroms alle Zeichen ein.

Tabelle 12.18 Methoden der Klasse »StreamReader«

Wir wollen nun an einem Codebeispiel das Lesen aus einem Strom testen.

```
// Beispiel: ..\Kapitel 12\StreamReaderSample
class Program {
  static void Main(string[] args) {
    // Datei erzeugen und mit Text füllen
    StreamWriter sw = new StreamWriter(@"D:\MyTest.kkl");
    sw.WriteLine("Visual C#");
    sw.WriteLine("macht viel Spass.");
    sw.Write("Richtig??");
    sw.Close();
    // die Datei an der Konsole einlesen
    StreamReader sr = new StreamReader(@"D:\MyTest.kkl");
    while(sr.Peek() != -1)
      Console.WriteLine(sr.ReadLine());
    sr.Close();
    Console.ReadLine();
  }
}
```

Listing 12.9 Textdatei mit »StreamReader« lesen

Zunächst wird mit einem `StreamWriter`-Objekt eine Datei mit dem Namen *MyTest.kkl* erzeugt. Die Dateierweiterung ist frei gewählt, sie muss nicht zwangsläufig *.txt* zur Kennzeichnung als Textdatei lauten. Wichtig ist nur, die Daten der Datei beim späteren Lesevorgang richtig zu interpretieren. Solange wir wissen, dass wir es mit einer Textdatei zu tun haben, bereitet uns eine individuelle Dateierweiterung keine Probleme.

In den Datenstrom *sw* vom Typ `StreamWriter` werden drei Textzeilen geschrieben. Danach darf man nicht vergessen, den Strom wieder zu schließen, denn ansonsten wird man mit einer Fehlermeldung konfrontiert, wenn nachfolgend der Versuch unternommen wird, die Datei zum Lesen zu öffnen.

Um den Dateiinhalt zu lesen, nutzen wir ein Objekt vom Typ `StreamReader`, dessen Konstruktor wir den Pfad zu der Datei übergeben. Mit der `ReadLine`-Methode wird Zeile für Zeile aus dem Strom gelesen. Um den Lesevorgang zum richtigen Zeitpunkt wieder zu beenden, müssen wir das Ende der Datei feststellen. Hierbei ist die Methode `Peek` behilflich, deren Rückgabewert –1 ist, wenn der Zeiger auf die Position hinter dem Ende des Stroms verweist. Dieses Verhalten machen wir uns zunutze, indem wir daraus die Abbruchbedingung der Schleife formulieren. In der `while`-Schleife werden so lange mit der `ReadLine`-Methode des `StreamReader`-Objekts Zeilen aus dem Datenstrom geholt (und dabei wird automatisch der Zeiger auf das nächste einzulesende Zeichen gesetzt), bis die Abbruchbedingung erfüllt wird, d.h. `Peek` –1 zurückliefert.

Die Ausgabe an der Konsole wird wie folgt lauten:

```
Visual C#
macht Spass.
Richtig??
```

Da wir die komplette Textdatei auslesen wollen, könnten wir auch einen einfacheren Weg gehen und die komplette `while`-Schleife gegen die folgende Programmcodezeile austauschen:

```
Console.WriteLine(sr.ReadToEnd());
```

12.6 Die Klassen »BinaryReader« und »BinaryWriter«

Daten werden in einer Datei byteweise gespeichert. Dieses Grundprinzip macht sich die Klasse `FileStream` zunutze, indem sie Daten Byte für Byte in den Datenstrom schreibt oder aus einem solchen liest. Dazu werden Methoden angeboten, die entweder nur ein einzelnes Byte behandeln oder auf Basis eines Byte-Arrays operieren. Eine spezialisiertere Form der einfachen, byteweisen Vorgänge bieten uns die Klassen `BinaryReader` bzw. `BinaryWriter`. Mit `BinaryReader` lesen Sie aus dem Datenstrom, mit `BinaryWriter` schreiben Sie in einen solchen hinein. Das Besondere an den beiden Klassen ist die Behandlung der übergebenen oder ausgewerteten Daten.

Die Methoden der Klassen »BinaryReader« und »BinaryWriter«

Fast schon erwartungsgemäß veröffentlicht die Klasse `BinaryWriter` eine `Write`-Methode, die vielfach überladen ist. Sie können dieser Methode einen beliebigen primitiven Typ als Argument übergeben, der mit der ihm eigenen Anzahl von Bytes in der Datei gespeichert wird. Ein `int` schreibt sich demnach mit vier Bytes in eine Datei, ein `long` mit acht.

Ähnliches gilt auch für die Methode `Read`, der noch der Typ als Suffix angehängt wird, der gelesen wird, z.B. `ReadByte`, `ReadInt32`, `ReadSingle` usw.

Die Konstruktoren der Klassen »BinaryReader« und »BinaryWriter«

In den beiden Klassen `BinaryReader` und `BinaryWriter` stehen Ihnen nur jeweils zwei Konstruktoren zur Verfügung. Dem ersten können Sie die Referenz auf ein Objekt vom Typ `Stream` übergeben, dem zweiten zusätzlich noch eine `Encoding`-Referenz.

Binäre Datenströme auswerten

Aus einem Strom die Bytes auszulesen, ist kein Problem. Halten Sie aber nur die rohen Bytes in den Händen, werden diese in den meisten Fällen nur von geringem Nutzen sein. Das Problem ist, eine bestimmte Sequenz von Bytes in richtiger Weise zu interpretieren. Kennen Sie den Typ der Dateninformationen nicht, sind die Bytes praktisch wertlos. Betrachten Sie dazu das folgende Beispiel:

```
// Beispiel: ..\Kapitel 12\BinaryReaderSample_1
class Program {
  static void Main(string[] args) {
    // eine Datei erzeugen und einen Integer-Wert in die Datei schreiben
    FileStream fileStr = new FileStream(@"D:\Binfile.mic", FileMode.Create);
    BinaryWriter binWriter = new BinaryWriter(fileStr);
    int intArr = 500;
    binWriter.Write(intArr);
    binWriter.Close();
    // Datei öffnen und den Inhalt byteweise auslesen
    FileInfo fi = new FileInfo(@"D:\Binfile.mic");
    FileStream fs = new FileStream(@"D:\Binfile.mic", FileMode.Open);
    byte[] byteArr = new byte[fi.Length];
    // Datenstrom in ein Byte-Array einlesen
    fs.Read(byteArr, 0, (int)fi.Length);
    // Konsolenausgabe
    Console.Write("Interpretation als Byte-Array: ");
    for (int i = 0; i < fi.Length; i++)
      Console.Write(byteArr[i] + " ");
    Console.Write("\n\n");
    fs.Close();
    // Dateiinhalt textuell auswerten
    StreamReader strReader = new StreamReader(@"D:\Binfile.mic");
    Console.Write("Interpretation als Text: ");
    Console.WriteLine(strReader.ReadToEnd());
    strReader.Close();
    Console.ReadLine();
  }
}
```

Listing 12.10 Auswerten binärer Datenströme

Zuerst wird ein Objekt vom Typ `FileStream` erzeugt, um eine neue Datei anzulegen bzw. eine gleichnamige Datei zu überschreiben. Die Objektreferenz wird einem Konstruktor der Klasse `BinaryWriter` übergeben. Die Methode `Write` schreibt anschließend einen Integer mit dem Inhalt 500 in die Datei. Anschließend wird die Datei ausgelesen. Wir stellen uns dabei dumm und tun so, als wüssten wir nicht, von welchem Datentyp die in der Datei *D:\Binfile.mic* gespeicherte Zahl ist. Also testen wir den Dateiinhalt, zuerst byteweise und danach auch noch zeichenorientiert, in der Hoffnung, ein sinnvolles Ergebnis zu erhalten.

Zum byteweisen Lesen greifen wir auf die Klasse `FileStream` zurück, lesen den Datenstrom aus der Datei in das Byte-Array *byteArr* ein und geben dann die Elemente des Arrays an der Konsole aus:

```
Interpretation als Byte-Array: 244 1 0 0
```

Ein Unbedarfter wird vielleicht wegen der fehlerfreien Ausgabe in Verzückung geraten, wir wissen aber, dass es nicht das ist, was wir ursprünglich in die Datei geschrieben haben. Wie aber ist die Ausgabe zu interpretieren, die mit Sicherheit auf jedem Rechner genauso lauten wird?

Die vier Zahlen repräsentieren die vier Bytes aus der Datei. Dabei erfolgt die Anzeige vom Lower-Byte bis zum Higher-Byte. In die »richtige«, besser gesagt, gewohnte Reihenfolge gebracht müssten wir demnach

```
0 0 1 244
```

schreiben. Wir wissen, dass diese Bytes einen Integer beschreiben – und sie tun es auch, wenn wir uns nur die Bitfolge ansehen:

```
0000 0000 0000 0000 0000 0001 1111 0100
```

Die Kombination aller Bits ergibt tatsächlich die Dezimalzahl 500. Ein Benutzer, der nicht weiß, wie die vier Bytes zu interpretieren sind, hat die Qual der Wahl: Handelt es sich um vier einzelne Bytes oder um zwei Integer oder vielleicht um eine Zeichenfolge? Letzteres testet unser Code ebenfalls, das Ergebnis der Ausgabe ist ernüchternd: Uns grinst ein Smiley mit ausgestreckter Zunge an.

Ändern wir nun den Lesevorgang der Daten so ab, dass wir den Dateiinhalt tatsächlich als `int` auswerten:

```
FileStream fs = new FileStream(@"D:\Binfile.mic", FileMode.Open);
BinaryReader br = new BinaryReader(fs);
Console.WriteLine(br.ReadInt32());
```

Das Ergebnis wird diesmal mit der korrekten Ausgabe an der Konsole enden.

12.6.1 Komplexe binäre Dateien

Der Informationsgehalt binärer Dateien kann nur dann korrekt ausgewertet werden, wenn der Typ, den die Daten repräsentieren, bekannt ist. Im vorhergehenden Abschnitt haben Sie dazu ein kleines Beispiel gesehen. Binäre Dateien können aber mehr als nur einen einzigen Typ speichern, es können durchaus unterschiedliche Typen in beliebiger Reihenfolge sein. Um zu einem späteren Zeitpunkt auf die Daten zugreifen zu können, muss nur der strukturelle Aufbau der Datei – das sogenannte Dateiformat – der gespeicherten Informationen bekannt sein, ansonsten ist die Datei praktisch wertlos.

Dateien unterscheiden sich im Dateiformat: Eine Bitmap-Datei wird die Informationen zu den einzelnen Pixeln anders speichern, als Word den Inhalt eines Dokuments speichert; eine JPEG-Datei unterscheidet sich wiederum von einer MPEG-Datei. Die Dateierweiterung ist als Kennzeichnung einer bestimmten Spezifikation anzusehen, nämlich als Spezifikation der Datenstruktur in der Datei. Praktisch alle Binärdateien werden sich in ihrem Dateiformat unterscheiden.

Wir wollen uns nun in einem etwas aufwendigeren Beispiel dem Thema komplexer Binärdateien nähern, um das Arbeiten mit solchen Dateien zu verstehen, ohne zugleich in zu viel Programmcode die Übersicht zu verlieren. Sie können das Konzept, das sich hinter diesem Beispiel verbirgt, in ähnlicher Weise auch auf andere bekannte Dateiformate anwenden.

Dazu geben wir uns zunächst eine Struktur vor, die ein `Point`-Objekt beschreibt:

```
public struct Point {
   public int XPos {get; set;}
   public int YPos {get; set;}
   public long Color {get; set;}
}
```

Listing 12.11 Definition der Struktur »Point« für das folgende Beispiel

Der Typ `Point` veröffentlicht drei Daten-Member: `XPos` und `YPos` jeweils vom Typ `int` sowie `Color` vom Typ `long`. Nun wollen wir eine Klasse entwickeln, die in der Lage ist, die Daten vieler `Point`-Objekte in einer Datei zu speichern und später auch wieder auszulesen. Außerdem soll eine Möglichkeit geschaffen werden, um auf die Daten eines beliebigen `Point`-Objekts in der Datei zugreifen zu können.

Die erste Überlegung ist, wie das Format einer Datei aussehen muss, um den gestellten Anforderungen zu entsprechen. Die Daten mehrerer `Point`-Objekte hintereinander zu speichern, ist kein Problem. Stellen Sie sich aber nun vor, Sie würden versuchen, die Informationen des zehnten Punkts aus einer Datei zu lesen, in der nur die Daten für fünf Punkte enthalten sind. Das kann zu keinem erfolgreichen Ergebnis führen.

Wir wollen daher eine Information in die Datei schreiben, der wir die gespeicherte `Point`-Anzahl entnehmen können. Der Typ dieser Information muss klar definiert sein, damit jedes Byte in der Datei eine klare Zuordnung erhält. Im Folgenden wird diese Information in einem `int` gespeichert, und zwar am Anfang der Datei.

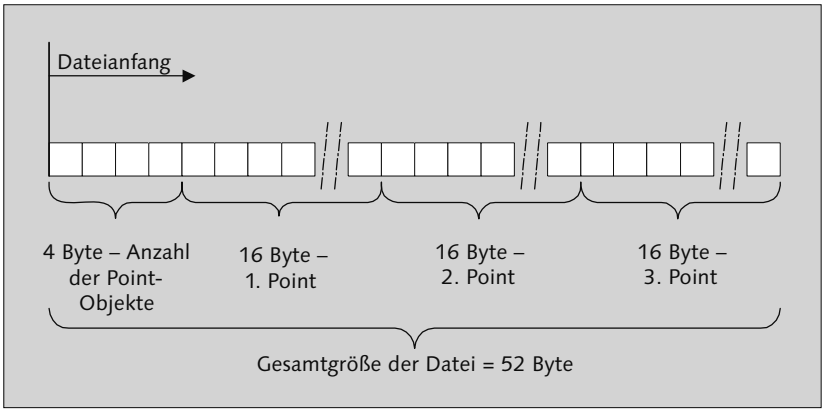


Abbildung 12.5 Datei mit drei gespeicherten Point-Objekten

Damit haben wir die Spezifikation der binären Datei festgelegt. Die Auswertung der ersten vier Bytes liefert die Anzahl der gespeicherten `Point`-Objekte, und die folgenden insgesamt 16 Byte großen Blöcke beschreiben jeweils einen Punkt. Wir könnten jetzt noch festlegen, dass Dateien dieses Typs beispielsweise die Dateierweiterung *.pot* erhalten, aber eine solche Festlegung wird der Code des folgenden Beispiels nicht berücksichtigen.

Da wir uns nun auf ein Dateiformat geeinigt haben, wollen wir uns das weitere Vorgehen überlegen. Wir könnten die gesamte Programmlogik in `Main` implementieren mit dem Nachteil, dass etwaige spätere Änderungen zu Komplikationen führen könnten. Besser ist es, sich das objektorientierte Konzept der Modularisierung in Erinnerung zu rufen. Deshalb wird eine Klasse definiert, deren Methoden die Dienste zur Initialisierung der *Point*-Objekte, zum Speichern in einer Datei, zum Lesen der Datei und zur Ausgabe der Daten eines beliebigen `Point`-Objekts zur Verfügung stellen. Der Name der Klasse sei `PointReader`, die Bezeichner der Methoden lauten `WriteToFile`, `GetFromFile` *und* `GetPoint`.

Grundsätzlich können Methoden als Instanz- oder Klassenmethoden veröffentlicht werden. Instanzmethoden würden voraussetzen, dass die Klasse `PointReader` instanziiert wird. Das Objekt wäre dann an eine bestimmte Datei gebunden, die `Point`-Objekte enthält. Statische Methoden sind flexibler einsetzbar, verlangen allerdings auch bei jedem Aufruf die Pfadangabe zu der Datei. In diesem Beispiel sollen die Methoden statisch sein.

Widmen wir uns der Methode `WriteToFile`. Sie hat die Aufgabe, eine Datei zu generieren, die die Anforderungen unserer Spezifikation zur Speicherung von *Point*-Objekten erfüllt. Die Pfadangabe muss der Methode als Argument übergeben werden.

Wie wird der Code in dieser Methode arbeiten? Zunächst muss eine `int`-Zahl in die Datei geschrieben werden, die der Anzahl der `Point`-Objektdaten entspricht. Danach werden `Point` für `Point` alle Objektdaten übergeben, bis das Array durchlaufen ist.

```
public static void WriteToFile(string path, Point[] array) {
  FileStream fileStr = new FileStream(path, FileMode.Create);
  BinaryWriter binWriter = new BinaryWriter(fileStr);
  // Anzahl der Punkte in die Datei schreiben
  binWriter.Write(array.Length);
  // die Point-Daten in die Datei schreiben
  for(int i = 0; i < array.Length; i++) {
    binWriter.Write(array[i].XPos);
    binWriter.Write(array[i].YPos);
    binWriter.Write(array[i].Color);
  }
  binWriter.Close();
}
```

Listing 12.12 »Point«-Daten in eine Datei schreiben

Die Daten der `Point`-Objekte sollen mit einer Instanz der Klasse `BinaryWriter` in die Datei geschrieben werden. Dazu benötigen wir auch ein Objekt vom Typ `FileStream`. Da alle Daten

hintereinander in eine neue Datei geschrieben werden sollen, müssen wir `FileStream` im Modus `Create` öffnen.

Nachdem wir die Referenz auf den `FileStream` an den Konstruktor der `BinaryWriter`-Klasse übergeben haben, wird die Anzahl der `Points` in die Datei geschrieben. In einer Schleife greifen wir danach jedes `Point`-Objekt im Array ab und schreiben die Daten nacheinander in die Datei. Zum Schluss muss der `Writer` ordnungsgemäß geschlossen werden.

Unsere Datei ist erzeugt, und nur mit dem Kenntnisstand der Spezifikation, wie die einzelnen Bytes zu interpretieren sind, liefern die Daten die richtigen Werte. Die Methode `GetFromFile` zum Auswerten des Dateiinhalts muss sich an unsere Festlegung halten. Daher wird auch zuerst der Integer aus der Datei gelesen und daran anschließend die Daten der `Point`-Objekte. Der Rückgabewert der Methode ist die Referenz auf ein intern erzeugtes `Point`-Array.

```
public static Point[] GetFromFile(string path) {
  FileStream fs = new FileStream(path, FileMode.Open);
  BinaryReader br = new BinaryReader(fs);
  // liest die ersten 4 Bytes aus der Datei, die die Anzahl der
  // Point-Objekte enthält
  int anzahl = br.ReadInt32();
  // Lesen der Daten aus der Datei
  Point[] arrPoint = new Point[anzahl];
  for (int i = 0; i < anzahl; i++) {
    arrPoint[i].XPos = br.ReadInt32();
    arrPoint[i].YPos = br.ReadInt32();
    arrPoint[i].Color = br.ReadInt64();
  }
  br.Close();
  return arrPoint;
}
```

Listing 12.13 Lesen der »Point«-Daten aus einer Datei

Da wir die Kontrolle über jedes einzelne gespeicherte Byte der Datei haben und dieses richtig zuordnen können, muss es auch möglich sein, die Daten eines beliebigen `Point`-Objekts einzulesen. Dazu dient die Methode `GetPoint`. Bei deren Aufruf wird zunächst die Pfadangabe übergeben und als zweites Argument die Position des `Point`-Objekts in der Datei. Der Rückgabewert ist die Referenz auf das gefundene Objekt.

```
public static Point GetPoint(string path, int pointNo) {
  FileStream fs = new FileStream(path, FileMode.Open);
  int pos = 4 + (pointNo - 1) * 16;
  BinaryReader br = new BinaryReader(fs);
  // Prüfen, ob der user eine gültige Position angegeben hat
```

```
  if (pointNo > br.ReadInt32() || pointNo == 0) {
    string message = "Unter der angegebenen Position ist";
    message += " kein \nPoint-Objekt gespeichert.";
    throw new PositionException(message);
  }
  // den Zeiger positionieren
  fs.Seek(pos, SeekOrigin.Begin);
  // Daten des gewünschten Points einlesen
  Point savedPoint = new Point();
  savedPoint.XPos = br.ReadInt32();
  savedPoint.YPos = br.ReadInt32();
  savedPoint.Color = br.ReadInt64();
  br.Close();
  return savedPoint;
}
```

Listing 12.14 Einlesen eines bestimmten »Point«-Objekts

Die wesentliche Funktionalität der Methode steckt in der richtigen Positionierung der Zeigers, die aus der Angabe des Benutzers berechnet wird. Dabei muss berücksichtigt werden, dass am Dateianfang vier Bytes die Gesamtanzahl der Objekte in der Datei beschreiben und dass die Länge eines einzelnen `Point`-Objekts 16 Byte beträgt.

```
int pos = 4 + (pointNo - 1) * 16;
```

Die so ermittelte Position wird der `Seek`-Methode des `BinaryReader`-Objekts übergeben. Die Positionsnummer des ersten Bytes in der Datei ist 0, daher verweist der Zeiger mit der Übergabe der Zahl 4 auf das fünfte Byte. Wir setzen in diesem Fall natürlich den Ursprung *origin* des Zeigers auf den Anfang des Datenstroms.

```
fs.Seek(pos, SeekOrigin.Begin)
```

Da damit gerechnet werden muss, dass der Anwender eine Position angibt, die keinem Objekt in der Datei entspricht, sollte eine Ausnahme ausgelöst werden. Diese ist benutzerdefiniert und heißt `PositionException`.

```
public class PositionException : Exception {
   public PositionException() {}
   public PositionException(string message) : base(message) {}
   public PositionException(string message, Exception inner):base(message, inner){}
}
```

Listing 12.15 Anwendungsspezifische Exception

Damit ist unsere Klassendefinition fertig, und wir können abschließend die Implementierung testen. Dazu schreiben wir entsprechenden Testcode in die Methode `Main`:

```
// Beispiel: ..\Kapitel 12\BinaryReaderSample_2
public class Program {
  static void Main(string[] args) {
    // Point-Array erzeugen
    Point[] pArr = new Point[2];
    pArr[0].XPos = 10;
    pArr[0].YPos = 20;
    pArr[0].Color = 310;
    pArr[1].XPos = 40;
    pArr[1].YPos = 50;
    pArr[1].Color = 110;
    // Point-Array speichern
    PointReader.WriteToFile(@"D:\Test.pot",pArr);
    // gespeicherte Informationen aus der Datei einlesen
    Point[] x = PointReader.GetFromFile(@"D:\Test.pot");
    // alle eingelesenen Point-Daten ausgeben
    for(int i = 0; i < 2; i++) {
      Console.WriteLine("Point-Objekt-Nr.{0}", i + 1);
      Console.WriteLine();
      Console.WriteLine("p[{0}].XPos = {1}", i, x[i].XPos);
      Console.WriteLine("p[{0}].YPos = {1}", i, x[i].YPos);
      Console.WriteLine("p[{0}].Color = {1}", i, x[i].Color);
      Console.WriteLine(new string('=',30));
    }
    // einen bestimmten Point einlesen
    Console.Write("\nWelchen Punkt möchten Sie einlesen? ");
    int position = Convert.ToInt32(Console.ReadLine());
    try {
      Point myPoint = PointReader.GetPoint(@"D:\Test.pot", position);
      Console.WriteLine("p.XPos = {0}", myPoint.XPos);
      Console.WriteLine("p.YPos = {0}", myPoint.YPos);
      Console.WriteLine("p.Color = {0}", myPoint.Color);
    }
    catch(PositionException e) {
      Console.WriteLine(e.Message);
    }
    Console.ReadLine();
  }
}
```

Listing 12.16 Komplettes Beispielprogramm

Weil die `Main`-Methode nur zum Testen der zuvor entwickelten Klasse dient, werden auch nur zwei `Point`-Objekte erzeugt, die uns als Testgrundlage für die weiteren Operationen die-

nen. Außerdem ist die Datei, in die gespeichert wird, immer dieselbe. Für unsere Zwecke ist das völlig ausreichend. Nach dem Speichern mit

```
PointReader.WriteToFile(@"D:\Test.pot", pArr);
```

wird die Datei sofort wieder eingelesen und die zurückgegebene Referenz einem neuen Array zugewiesen:

```
Point[] x = PointReader.GetFromFile(@"D:\Test.pot");
```

In einer Schleife werden danach alle eingelesenen Objektdaten an der Konsole ausgegeben.

Aufregender ist es hingegen, die Daten eines bestimmten Punktes zu erfahren. Dem Aufruf von `GetPoint` wird neben der Pfadangabe auch noch die Position des `Point`-Objekts in der Datei übergeben. Die Übergabe einer unzulässigen Position führt dazu, dass die spezifische Ausnahme `PositionException` mit einer entsprechenden Fehlermeldung ausgelöst wird, andernfalls werden die korrekten Werte angezeigt.

Kapitel 22
Elementbindungen

In den Kapiteln 19 bis 21 wurden viele Aspekte der Oberflächengestaltung diskutiert. Nun wird es Zeit, sich näher mit den speziellen WPF-Techniken zu beschäftigen. Dazu starten wir mit einem Thema, das sich durch die folgenden Kapitel wie ein roter Faden ziehen wird: den Bindungen. Man unterscheidet dabei zwei grundlegende Techniken: die Element- und die Datenbindung. Bei der Elementbindung wird die Eigenschaft eines Elements an die Eigenschaft eines anderen Elements gebunden, bei der Datenbindung handelt es sich um die Bindung einer Elementeigenschaft an eine allgemeine Datenquelle. In diesem Kapitel wird schwerpunktmäßig auf die Elementbindung eingegangen. In Kapitel 24 sehen wir uns noch die Details der allgemeinen Datenbindung an.

22.1 Einführung in die Bindungstechnik

Bindungen werden zwischen zwei Elementen definiert. Auf der einen Seite befindet sich eine Datenquelle, auf der anderen das Datenziel. Im Datenziel wird eine Eigenschaft oder auch mehrere an Daten gebunden, die die Datenquelle zur Verfügung stellt. Datenquellen können ausgesprochen vielfältig sein:

- Es kann sich bei der Datenquelle um die Eigenschaft eines anderen Elements (z.B. ein Steuerelement) handeln.
- Die Daten können aus einer XML-Datei bezogen werden.
- Daten können aus einer Auflistung oder gar aus einer Datenbank stammen.

Eine Voraussetzung muss dabei seitens des Datenziels erfüllt sein: Die Datenbindung funktioniert nur, wenn die Eigenschaft des Datenziels, an die eine Datenquelle gebunden wird, als Abhängigkeitseigenschaft (Dependency Property) implementiert ist. Erfreulicherweise erfüllen die meisten Eigenschaften der WPF-spezifischen Klassen diese Bedingung.

22.1.1 Ein einfaches Bindungsbeispiel

Bevor wir uns mit den Details der Elementbindung beschäftigen, wollen wir uns ein einfaches Beispiel ansehen, um den Bindungseffekt zu verstehen.

Die einfachste Bindung in der WPF ist sicherlich die Elementbindung zwischen zwei WPF-Elementen. Auch hier wird zwischen der Datenquelle und dem Datenziel unterschieden. Die

Datenquelle stellt einen Wert zur Verfügung, das Datenziel bindet eine Eigenschaft an diesen Wert. Sollte sich der Wert in der Datenquelle ändern, wird auch die gebundene Eigenschaft des Datenziels geändert. Wie bereits erwähnt, muss die gebundene Eigenschaft des Datenziels dabei eine Bedingung erfüllen: Es muss eine Abhängigkeitseigenschaft (Dependency Property) sein.

Kommen wir zum angekündigten ersten Beispiel, und sehen Sie sich bitte Abbildung 22.1 an. Im Fenster ist oben eine `TextBox` und unten eine `ListBox` platziert. Die `ListBox` enthält ein paar Namen, von denen einer ausgewählt werden kann. Die Eigenschaft `Text` der `TextBox` ist an die `ListBox` in der Weise gebunden, dass der in der `ListBox` selektierte Name sofort in der `TextBox` angezeigt wird.

Abbildung 22.1 Einfache Datenbindung zwischen »TextBox« und »ListBox«

Sehen wir uns den XAML-Code an, auf dem die Abbildung basiert.

```
// Beispiel: ..\Kapitel 22\SimpleBinding
<Window ... >
  <StackPanel>
    <TextBox Height="25" Margin="10, 10, 10, 0" Name="textBox1"
             Text="{Binding ElementName=listBox1,
                            Path=SelectedItem.Content}"/>
    <ListBox Margin="10, 10, 10, 10" Name="listBox1">
      <ListBoxItem>Michael</ListBoxItem>
      <ListBoxItem>Gerald</ListBoxItem>
      <ListBoxItem>Franz</ListBoxItem>
      <ListBoxItem>Ralf</ListBoxItem>
      <ListBoxItem>Sebastian</ListBoxItem>
      <ListBoxItem>Thomas</ListBoxItem>
      <ListBoxItem>Friedhelm</ListBoxItem>
      <ListBoxItem>Karsten</ListBoxItem>
    </ListBox>
  </StackPanel>
</Window>
```

Listing 22.1 Der XAML-Code des Beispielprogramms »SimpleBinding«

Die Datenquelle ist in diesem Beispiel die `ListBox`. Für die Datenbindung sind bei diesem Element keine besonderen Maßnahmen notwendig. Die Bindung selbst ist in der `TextBox` definiert, dem Datenziel. Anstatt der Eigenschaft `Text` eine fixe Zeichenfolge zu übergeben, wird die Eigenschaft mit

```
Text="{Binding ElementName=listBox1, Path=SelectedItem.Content}"
```

an die `ListBox` gebunden. Datenbindungen werden mit der Markup-Erweiterung von XAML umgesetzt, d.h. innerhalb der geschweiften Klammern beschrieben. In der Markup-Erweiterung wird zuerst ein Objekt vom Typ `Binding` angegeben. Es müssen zwei Eigenschaften dieses Objekts festgelegt werden: `ElementName` und `Path`. `ElementName` beschreibt den Bezeichner des Elements, an das gebunden wird. `Path` gibt die Eigenschaft der Datenquelle an, an die gebunden werden soll. Die WPF verwendet den Ausdruck `Path` und nicht wie vielleicht erwartet `Property`, weil `Path` auf die Eigenschaft einer Eigenschaft zeigen kann – wie auch in unserem Beispiel `SelectedItem.Content`. In anderen Situationen kann es sich natürlich auch um die einfache Angabe einer Eigenschaft handeln (z.B. `Text`).

Wem die Bindung innerhalb einer Markup Extension zu unübersichtlich ist, kann die längere, besser strukturierte Schreibweise über XML-Knoten wählen.

```
<TextBox Height="25" Margin="10, 10, 10, 0" Name="textBox1" >
  <TextBox.Text>
    <Binding ElementName="listBox1" Path="SelectedItem.Content" />
  </TextBox.Text>
</TextBox>
```

Listing 22.2 Lange Schreibweise einer Datenbindung

Um die Darstellung unseres ersten Beispiels zu vervollständigen, darf nicht unerwähnt bleiben, dass seine Datenbindung natürlich auch mit Programmcode möglich ist. Bezogen auf unser Beispielprogramm könnte die Bindung im Konstruktor des `Window` erfolgen, nach dem Aufruf der Methode `InitializeComponent`:

```
public MainWindow() {
  InitializeComponent();
  Binding binding = new Binding("SelectedItem.Content");
  binding.ElementName = "listBox1";
  textBox1.SetBinding(TextBox.TextProperty, binding);
}
```

Listing 22.3 Datenbindung mit C#-Code

Bemerkenswert ist die Anbindung der Eigenschaft `Text` an die durch das `Binding`-Objekt beschriebene Datenquelle mit der Methode `SetBinding`, der zwei Argumente übergeben werden. Dem ersten wird die Abhängigkeitseigenschaft des Datenziels genannt. Wahrscheinlich haben Sie hier `textBox1.Text` erwartet. Stattdessen wird aber `TextBox.TextProperty` angegeben. Dieser Schreibweise begegnen wir hier zum ersten Mal. Sie ist typisch für Methoden, die einen Parameter vom Typ `DependencyProperty` definieren.

22.2 Die Klasse »Binding«

Ein `Binding`-Objekt beschreibt die Bindung zwischen einer Datenquelle und der bindenden Komponente. Meistens wird die Bindung im XAML-Code beschrieben, aber selbstverständlich kann die Bindung wie in Listing 22.3 gezeigt auch im C#-Code erfolgen.

Die wichtigsten Eigenschaften des `Binding`-Objekts können Sie der folgenden Tabelle entnehmen. Viele der aufgeführten Eigenschaften werden Sie im Verlauf des Kapitels noch innerhalb der Beispielprogramme wiederfinden.

Eigenschaft	Beschreibung
`Converter`	Diese Eigenschaft gibt an, welches Objekt als Konverter zwischen zwei Datentypen dienen soll.
`ElementName`	Diese Eigenschaft gibt den Bezeichner des Steuerelements an, das als Datenquelle dient.
`FallbackValue`	Diese Eigenschaft definiert einen Standardwert, der verwendet wird, wenn das `Binding`-Objekt noch keinen Wert zurückliefert.
`IsAsync`	Wird diese Eigenschaft auf `true` gesetzt (Standard ist `false`), werden die Daten aus der Datenquelle asynchron geladen oder asynchron in die Datenquelle geschrieben.
`Mode`	Legt den Bindungsmodus fest. Damit lässt sich die Verhaltensweise einer eventuellen gegenseitigen Aktualisierung zwischen Datenquelle und Datenziel steuern.
`NotifyOnSourceUpdate/ NotifyOnTargetUpdate`	Mit diesen Eigenschaften werden Ereignisse ermöglicht, die beim Aktualisieren der Datenquelle oder des Datenziels auftreten.
`Path`	Mit `Path` wird die Eigenschaft festgelegt, an die Daten gebunden werden sollen.
`RelativeSource`	Diese Eigenschaft legt die Datenquelle fest, allerdings relativ zum Datenziel.
`Source`	Die Eigenschaft `Source` legt das Objekt fest, das die Quelle der Datenbindung sein soll.
`UpdateSourceTrigger`	Mit dieser Eigenschaft wird angegeben, wann die Datenquelle aktualisiert werden soll.

Tabelle 22.1 Die Eigenschaften eines »Binding«-Objekts

Der Eigenschaft `Path` können auf unterschiedlichste Weise Pfade angegeben werden. Im einfachsten Fall geben Sie nur die Eigenschaft an, z.B.:

```
Path=Text
```

Es kann sich natürlich um eine untergeordnete Eigenschaft handeln, wie das Beispielprogramm *SimpleBinding* gezeigt hat:

```
Path=SelectedItem.Content
```

Sie könnten auch ohne Probleme an einen bestimmten Listeneintrag der `ListBox` binden:

```
Path=Items[0].Content
```

Nicht unerwähnt bleiben sollte auch, dass mehrere verschachtelte Eigenschaften möglich sind.

Sie können auf die explizite Angabe von `Path` auch verzichten und direkt die Eigenschaft nennen. Allerdings muss das sofort nach der Angabe des `Binding`-Objekts in der Markup Extension erfolgen:

```
Text="{Binding Text, ElementName= ...}"
```

22.2.1 Die Bindungsrichtung festlegen

Das letzte Beispielprogramm wollen wir für einen weiteren Test benutzen. Wählen Sie dazu zur Laufzeit einen der in der `ListBox` eingetragenen Namen aus. Dieser wird wegen der definierten Bindung in der `TextBox` angezeigt. Ändern Sie nun den Namen in der `TextBox`, und fokussieren Sie durch Drücken der ⇆-Taste erneut die `ListBox`. Sie werden feststellen, dass der neue Name in der `TextBox` den ursprünglichen im Listenfeld ersetzt. Anscheinend ist sogar das Datenziel in der Lage, die Datenquelle zu aktualisieren. Wir ergänzen nun den Bindungscode der `TextBox` wie folgt um die Eigenschaft `Mode=OneWay`:

```
Text="{Binding ElementName=listBox1, Path=SelectedItem.Content, Mode=OneWay}"
```

Listing 22.4 Änderung der Datenbindung im Beispielprogramm »SimpleBinding«

Wenn Sie denselben Test nun erneut durchführen, werden Sie einen gravierenden Unterschied feststellen: Der Eintrag in der `ListBox` wird jetzt nicht mehr verändert.

Durch Angabe des Bindungsmodus mit der Eigenschaft `Mode` des `Binding`-Objekts lassen sich offensichtlich die wechselseitigen Änderungen zwischen der Datenquelle und dem Datenziel steuern. `Mode` kann alle Werte annehmen, die in der Enumeration `BindingMode` beschrieben werden. Diese können Sie Tabelle 22.2 entnehmen.

Bindungstyp	Beschreibung
Default	Hiermit wird der Standardmodus beschrieben, der für die Eigenschaft vordefiniert ist. Dabei handelt es sich meistens um den Modus TwoWay.
OneTime	Der Wert wird nur einmal von der Quelle zum Ziel übertragen. Danach findet keine Aktualisierung mehr statt.
OneWay	Der Wert wird nur von der Quelle zum Ziel übertragen.
OneWayToSource	Der Wert wird von der Quelle zum Ziel übertragen. Ändert sich der Wert im Datenziel, wird die Datenquelle ebenfalls aktualisiert.
TwoWay	Die Werte sowohl von der Quelle als auch vom Ziel werden in beide Richtungen übertragen.

Tabelle 22.2 Bindungsarten der Enumeration »BindingMode«

Am besten lassen sich die Einstellungen verstehen, wenn die entsprechenden Effekte in einem Beispiel geprüft werden können. Dazu dient das folgende Beispielprogramm *BindungsModi*, dessen Darstellung zur Laufzeit Abbildung 22.2 zeigt.

```
// Beispiel: ..\Kapitel 22\BindungsModi
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="120" /><ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="25"/><RowDefinition />
    <RowDefinition /><RowDefinition />
    <RowDefinition /><RowDefinition />
  </Grid.RowDefinitions>
  <Label Content="Bindungsart" />
  <Label Content="Quelle" />
  <Label Content="Ziel" />
  <Label Content="Default" />
  <TextBox Name="txtDefaultQuelle" Grid.Column="1" Grid.Row="1" />
  <TextBox Name="txtDefaultZiel" Grid.Column="2" Grid.Row="1"
           Text="{Binding ElementName=txtDefaultQuelle, Path=Text, Mode=Default}"/>
  <Label Content="OneWay" />
  <TextBox Name="txtOneWayQuelle" Grid.Column="1" Grid.Row="2"/>
  <TextBox Name="txtOneWayZiel" Grid.Column="2" Grid.Row="2"
           Text="{Binding ElementName=txtOneWayQuelle, Path=Text, Mode=OneWay}"/>
  <Label Content="TwoWay" />
  <TextBox Name="txtTwoWayQuelle" Grid.Column="1" Grid.Row="3" />
  <TextBox Name="txtTwoWayZiel" Grid.Column="2" Grid.Row="3"
```

```
          Text="{Binding ElementName=txtTwoWayQuelle, Path=Text, Mode=TwoWay}"/>
  <Label Content="OneTime" />
  <TextBox Name="txtOneTimeQuelle" Grid.Column="1" Grid.Row="4" />
  <TextBox Name="txtOneTimeZiel" Grid.Column="2" Grid.Row="4"
          Text="{Binding ElementName=txtTwoWayQuelle, Path=Text,
                 Mode=OneTime}" Focusable="False" />
  <Label Content="OneWayToSource" />
  <TextBox Name="txtOneWayToSourceQuelle" Grid.Column="1" Grid.Row="5" />
  <TextBox Name="txtOneWayToSourceZiel" Grid.Column="2" Grid.Row="5"
          Text="{Binding ElementName=txtOneWayToSourceQuelle, Path=Text,
                 Mode=OneWayToSource}" />
</Grid>
```

Listing 22.5 Der XAML-Code des Beispielprogramms (gekürzt)

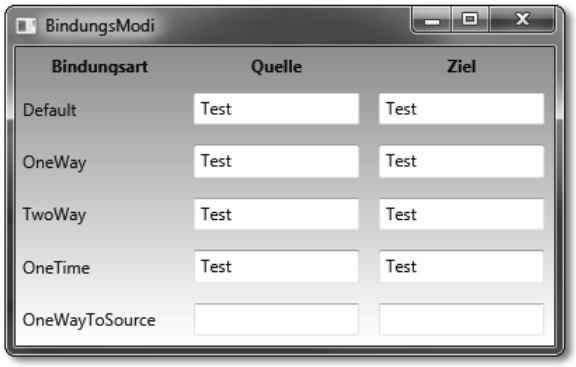


Abbildung 22.2 Ausgabe des Beispielprogramms »BindungsModi«

In der mittleren Spalte des `Grid`-Steuerelements sind die Eingabesteuerelemente zu finden, die als Datenquelle dienen, in der rechten Spalte sind die Zielobjekte angeordnet. Sie können in die `TextBox`en beider Spalten Änderungen eintragen und dabei beobachten, wie sich der entsprechende Gegenpart der Datenbindung verhält. Beachten Sie, dass Sie in Abhängigkeit vom Modus erst die `TextBox` verlassen müssen, in der Sie eine Änderung vorgenommen haben, damit die Änderung auf der anderen Seite der Bindung umgesetzt wird.

Auffällig bei den verschiedenen Modi in Tabelle 22.2 sind `OneWay` und `OneWayToSource`, die im Grunde genommen beide eine Bindung in der Art einer Einbahnstraße festschreiben. Der Unterschied ist, dass Sie `OneWayToSource` auch auf der Seite des Datenziels angeben können und es auf diese Weise schaffen, die angebundene Eigenschaft einer Datenquelle auch dann zu aktualisieren, wenn sie nicht als Abhängigkeitseigenschaft implementiert ist, sondern nur als einfache CLR-Eigenschaft.

Der Modus `OneWay` bietet sich an, wenn dem Benutzer schreibgeschützte Daten angezeigt werden sollen. Die `TwoWay`-Bindung ist empfehlenswert, wenn der Benutzer in der Lage sein

soll, die Daten im Steuerelement zu ändern, und wenn diese Änderung in die Datenquelle zurückgeschrieben werden soll.

22.2.2 Aktualisierung der Bindung

Sehen wir uns noch einmal das Beispielprogramm *SimpleBinding* an, und hier besonders die Modi `TwoWay` und `OneWay`. Ändert sich die Quelle, wird das Ziel der Bindung sofort aktualisiert. Ändert sich hingegen das Datenziel, wird die Quelle erst dann geändert, wenn das Zielelement den Fokus verliert. Das ist nicht immer wünschenswert und kann mit der Eigenschaft `UpdateSourceTrigger` des `Binding`-Objekts gesteuert werden. Darüber lässt sich festlegen, dass bei einer Änderung des Zielelements die Quelle erst durch den Aufruf einer Methode aktualisiert wird.

Die möglichen Einstellungen der Eigenschaft `UpdateSourceTrigger` sind in der gleichnamigen Enumeration `UpdateSourceTrigger` festgelegt.

Wert	Beschreibung
`Default`	Die Standardeinstellung variiert in Abhängigkeit vom verwendeten Element. Bei einer `TextBox` ist zum Beispiel die Standardeinstellung `LostFocus`. In den meisten Fällen handelt es sich aber um `PropertyChanged`.
`Explicit`	Die Änderung muss explizit durch Aufruf der `UpdateSource`-Methode des `BindingExpression`-Objekts erfolgen.
`LostFocus`	Die Quelle wird aktualisiert, wenn die Zielkomponente den Fokus verliert.
`PropertyChanged`	Die Aktualisierung erfolgt bei jeder Änderung. Allerdings beansprucht diese Einstellung die Ressourcen sehr intensiv.

Tabelle 22.3 Einstellungen der Eigenschaft »UpdateSourceTrigger«

Denken Sie immer daran, dass die in Tabelle 22.3 angegebenen Werte keinen Einfluss darauf haben, wie das Datenziel aktualisiert wird. Mit `UpdateSourceTrigger` legen Sie ausschließlich fest, wie die Quelle aktualisiert wird.

Mit diesem Wissen ausgestattet können Sie nun eine Datenbindung auf der Seite des Datenziels definieren, die bei jeder Änderung die Quelle über die Änderung informiert.

```
<TextBox Text="{Binding ElementName=textBox1, Path=Text, Mode=TwoWay,
        UpdateSourceTrigger=PropertyChanged}"/>
```

Listing 22.6 Änderung der Bindungsdaten bei jeder Änderung

Sie sollten insbesondere bei dem Element `TextBox` berücksichtigen, dass jede Änderung mit der einhergehenden Aktualisierung zu Lasten der Performance geht. Das ist auch der Grund für die Vorgabeeinstellung `LostFocus`.

Beispiel der Einstellung »UpdateSourceTrigger.Explicit«

Betrachten wir an dieser Stelle die Einstellung `Explicit` der Eigenschaft `UpdateSourceTrigger`, da sie sich von den anderen Modi unterscheidet. Dabei wird die Aktualisierung der Quelle durch Aufruf der Methode `UpdateSource` der Klasse `BindingExpression` von außen angestoßen. Das könnte beispielsweise durch das Drücken der ↵-Taste der Fall sein oder auch innerhalb eines Ereignishandlers programmiert sein, wie das folgende Beispiel zeigt (siehe auch Abbildung 22.3).

Abbildung 22.3 Fenster des Beispiels »ExplicitUpdate«

```
// Beispiel: ..\Kapitel 22\ExplicitUpdate
<Window ... >
  <StackPanel>
    <TextBox Name="txtSource" Margin="10"></TextBox>
    <TextBox Name="txtTarget" Margin="10"
             Text="{Binding ElementName=txtSource, Path=Text,
             UpdateSourceTrigger=Explicit}"/>
    <Button Name="btnUpdate" Margin="10" Width="120"
            Click="btnUpdate_Click">Aktualisieren</Button>
  </StackPanel>
</Window>
```

Dazu gehört der Ereignishandler der Schaltfläche in der Code-Behind-Datei:

```
private void btnUpdate_Click(object sender, RoutedEventArgs e) {
  BindingExpression binding = txtTarget.GetBindingExpression(TextBox.TextProperty);
  binding.UpdateSource();
}
```

Listing 22.7 C#-Code des Beispiels »ExplicitUpdate«

Hier betritt mit `BindingExpression` eine weitere Klasse die Bühne. Während ein Objekt vom Typ `Binding` lediglich die Eigenschaften einer Bindung beschreibt und sich somit auch für mehrere Bindungen eignet, repräsentiert das `BindingExpression`-Objekt die eigentliche Datenbindung zwischen der Quelle und dem Ziel. Das `BindingExpression`-Objekt erhalten Sie übrigens auch als Rückgabewert der Methode `SetBinding` (siehe Listing 22.3).

Neben der im Beispielprogramm eingesetzten Methode `UpdateSource` besitzt das `BindingExpression`-Objekt auch eine Methode `UpdateTarget`, mit der die Eigenschaft im Datenziel aktualisiert werden kann. Diese Methode lässt sich sinnvoll einsetzen, wenn die Datenquelle nur eine gewöhnliche CLR-Eigenschaft ist und nicht das Interface `INotifyPropertyChanged` implementiert (mehr dazu weiter unten).

22.2.3 Die Ereignisse »SourceUpdated« und »TargetUpdated«

Ändert sich das Datenziel oder die Datenquelle, werden, je nach Situation, die Ereignisse `SourceUpdated` und `TargetUpdated` ausgelöst. Das ermöglicht uns, im Kontext einer Aktualisierung weitere Operationen auszuführen. Für Irritationen sorgt am Anfang meistens die Namensgebung. `SourceUpdated` wird nämlich ausgelöst, wenn sich die Datenquelle aufgrund einer Änderung des Datenziels geändert hat. Analog wird das Ereignis `TargetUpdated` ausgelöst, wenn sich das Datenziel wegen einer Änderung der Datenquelle anpasst.

Ereignis	Beschreibung
`SourceUpdated`	Dieses Ereignis wird ausgelöst, wenn sich die Datenquelle wegen einer Änderung des Datenziels aktualisiert.
`TargetUpdated`	Dieses Ereignis wird ausgelöst, wenn sich das Datenziel wegen einer Änderung der Datenquelle aktualisiert.

Tabelle 22.4 Ereignisse bei der Aktualisierung datengebundener Komponenten

Die beiden Ereignisse werden nicht standardmäßig ausgelöst, sie müssen vorher aktiviert werden. Dazu dienen die beiden Eigenschaften `NotifyOnSourceUpdated` und `NotifyOnTargetUpdated`, die beide auf `true` festgelegt werden müssen. Darüber hinaus sind natürlich bei Bedarf die Ereignishandler der beiden Ereignisse zu registrieren, die als *Attached Events* in der Klasse `Binding` definiert sind.

Sehen wir uns das im folgenden Beispiel an. Dabei wird auf das Fenster aus dem Beispiel *ExplicitUpdate* des vorhergehenden Abschnitts zurückgegriffen, und es wird um das Ereignis `SourceUpdated` ergänzt.

```
<Window ...>
  [...]
  <TextBox Grid.Column="1" Name="txtSource" Margin="10"></TextBox>
  <TextBox Grid.Column="1" Grid.Row="1" Name="txtTarget" Margin="10"
           Binding.SourceUpdated="txtTarget_SourceUpdated"
           Text="{Binding ElementName=txtSource, Path=Text,
                          UpdateSourceTrigger=Explicit,
                          NotifyOnSourceUpdated=True}"/>
  <Button Name="btnUpdate" Grid.Column="1" Grid.Row="2" Margin="10"
```

```
          Click="btnUpdate_Click">Aktualisieren</Button>
  [...]
</Window>
```

Hier noch der C#-Code des Ereignishandlers:

```
private void txtTarget_SourceUpdated(object sender,
                                     DataTransferEventArgs e) {
  MessageBox.Show("Aktualisierung durchgeführt.");
}
```

Listing 22.8 Ergänzungen im Beispielprogramm »ExplicitUpdate«

Ändern Sie Daten in der als Datenziel definierten unteren `TextBox`, wird die obere `TextBox`, die eigentliche Datenquelle erst in dem Moment aktualisiert, wenn auf die Schaltfläche geklickt wird. Gleichzeitig wird das Ereignis `SourceUpdated` ausgelöst und signalisiert mit einem Nachrichtenfenster die durchgeführte Aktualisierung.

22.2.4 Beenden einer Bindung

Um eine Bindung zu lösen, wird die statische Methode `ClearBinding` der Klasse `BindingOperations` aufgerufen und dabei das Zielobjekt und die abhängige Eigenschaft angegeben:

```
BindingOperations.ClearBinding(textBox1, TextBox.TextProperty);
```

Hier wird beispielsweise die Eigenschaft `Text` eines Elements mit dem Namen *textBox1* gelöst. Sind mehrere Eigenschaften der Zielkomponente datengebunden, können Sie auch mit einem einzigen Methodenaufruf alle Bindungen mit einer Anweisung lösen. Dazu dient die Methode `ClearAllBindings`:

```
BindingOperations.ClearAllBindings(textBox1);
```


22.3 Bindungsalternativen

Bisher haben wir die Datenbindung nur zwischen zwei Elementen mit der Eigenschaft `ElementName` kennengelernt. Möchten Sie eine Datenbindung mit einem nichtvisuellen Objekt, müssen Sie anstelle von `ElementName` eine der drei folgenden Eigenschaften einsetzen:

- `Source`
- `RelativeSource`
- `DataContext`

22.3.1 Die Eigenschaft »Source«

Die Eigenschaft `Source` ist die allgemeinste von allen. Sie können damit praktisch jedes CLR-Objekt zur Datenquelle erklären. Nehmen wir an, Sie hätten eine Klasse `Person` implemen-

tiert und würden im XAML-Code in einem Ressourcenabschnitt ein Objekt davon erstellen. Die Bindung des Objekts an eine `TextBox` würde mit der Eigenschaft `Source` erfolgen, z. B.:

```
<Window.Resources>
  <local:Person x:Key="pers" Name="Hans im Glück" />
</Window.Resources>
<Grid>
  <TextBox Text="{Binding Source={StaticResource pers}, Path=Name}" />
</Grid>
```

Listing 22.9 Bindung an ein CLR-Objekt

Auch ohne dass wir uns bisher über die Möglichkeiten des `Resources`-Abschnitts unterhalten haben, dürfte das Listing verständlich sein. In Kapitel 23 werden wir die Möglichkeiten dieses Abschnitts noch genauer untersuchen.

22.3.2 Anbindung an relative Datenquellen

Bindungen müssen sich nicht zwangsläufig auf ein anderes Steuerelement beziehen. Eine Bindung kann mit `RelativeSource` auch auf relative Quellen verweisen.

Der folgende XAML-Code beschreibt einen `Slider`, der vom Minimalwert 0 bis zum Maximalwert 10 seinen jeweils aktuellen Wert in einem `ToolTip`-Element anzeigt.

```
<Slider ToolTip="{Binding RelativeSource={RelativeSource Self},
                          Path=Value}"></Slider>
```

Listing 22.10 »RelativeSource« mit Verweis auf sich selbst

Dieser Relativverweis kann auch ohne Markup Extension definiert werden:

```
<Slider>
  <Slider.ToolTip>
     <Binding Path="Value">
      <Binding.RelativeSource>
        <RelativeSource Mode="Self"></RelativeSource>
      </Binding.RelativeSource>
    </Binding>
  </Slider.ToolTip>
</Slider>
```

Listing 22.11 Relativer Verweis ohne Markup Extension

Neben dem `Self`-Modus gibt es noch drei weitere Modi zur Angabe relativer Bezüge.

Modus	Beschreibung
`Self`	Verweis auf sich selbst.
`TemplatedParent`	Dieser Modus ist nur innerhalb eines Templates sinnvoll und verweist auf sich selbst in einem Template (`Self` würde `ControlTemplate` liefern und nicht das Element selbst).
`PreviousData`	In einer Auflistung kann über diesen Modus der vorherige Datensatz abgerufen werden.
`FindAncestor`	Relative Navigation durch den Elementbaum, mit Hilfe des gewünschten Typs und des Levels.

Tabelle 22.5 Die Modi von »RelativeSource«

Sehen wir uns noch ein Beispiel zu dem Modus `FindAncestor` an. Er dient dazu, die Datenquelle auf ein im Elementbaum höher liegendes Element eines bestimmten Typs zu setzen. Dieser Modus verlangt, dass in der `AncestorType`-Eigenschaft des `RelativeSource`-Objekts der gesuchte Typ angegeben wird.

In unserem Codefragment wollen wir die Eigenschaft `Text` einer `TextBox` an die Beschriftung der Titelleiste des `Window`-Elements binden (Eigenschaft `Title`).

```
<TextBox Text="{Binding
         RelativeSource={RelativeSource FindAncestor,
         AncestorType={x:Type Window}},
         Path=Title}" />
```

Listing 22.12 Bindung an ein im Elementbaum höher liegendes Element

Natürlich gibt es andere Fälle, in denen der angegebene Typ mehrfach im Elementbaum auftritt. Sie haben dann die Möglichkeit, mit der Eigenschaft `AncestorLevel` den n-ten gefundenen Typ anzugeben. Dabei entspricht der Wert 1 dem ersten gefundenen Element dieses Typs. In Listing 22.13 wird der Modus `FindAncestor` verwendet, um die `Content`-Eigenschaft eines `Button`-Objekts an die Eigenschaft `Text` der `TextBox` zu binden. Es wird dabei nach der zweiten Schaltfläche gesucht.

```
<TextBox Text="{Binding
         RelativeSource={RelativeSource FindAncestor,
         AncestorType={x:Type Button}, AncestorLevel=1},
         Path=Content}" />
```

Listing 22.13 Die Bindung an die zweite übergeordnete Schaltfläche im Elementbaum

22.3.3 Die Bindung an »DataContext«

`DataContext` ist eine Eigenschaft, die von allen Klassen bereitgestellt wird, die von `FrameworkElement` oder `FrameworkContentElement` abgeleitet sind. Die Eigenschaft `DataContext` beschreibt eine Datenquelle. Dabei kann es sich um ein einfaches benutzerdefiniertes Objekt handeln, aber auch um eine Tabelle, die nach einer Datenbankabfrage im lokalen Cache vorliegt.

`DataContext` hat eine besondere Charakteristik: Man kann ihr im Elementbaum weit oben eine Datenquelle angeben, beispielsweise im `Window`, z. B.:

```
this.DataContext = [Datenquelle];
```

Listing 22.14 Angabe der Datenquelle im »Window«

Allen im `Window` untergeordneten Elementen stehen nun die Daten zur Verfügung.

Die Bindung der Elemente an die durch `DataContext` beschriebene Datenquelle erfolgt auch in diesem Fall mit einem `Binding`-Objekt. Allerdings ist es völlig ausreichend, nur die `Path`-Eigenschaft festzulegen:

```
<TextBox Text="{Binding Path=Name}" />
```

Listing 22.15 Bindung an die in einem übergeordneten Element beschriebene Datenquelle

Sollte in einem Element die explizite Angabe der Datenquelle wie in Listing 22.15 fehlen, wird die WPF zuerst die `DataContext`-Eigenschaft des betreffenden Elements untersuchen. Sollte diese `null` sein, sucht die WPF den Elementbaum so lange ab, bis sie zum ersten Mal auf eine `DataContext`-Eigenschaft stößt, die nicht `null` ist. Diese wird dann als Datenquelle genutzt. Sollten alle `DataContext`-Eigenschaften `null` aufweisen, kann dem Element auch kein Wert zugewiesen werden.

Natürlich kann ein Steuerelement im Bedarfsfall trotzdem auf eine andere Datenquelle verweisen, wenn das gewünscht ist. Dazu ist nur `ElementName`, `Source` oder `RelativeSource` dem `Binding`-Objekt anzugeben.

Anmerkung

Auf ein Beispiel soll an dieser Stelle verzichtet werden. In Kapitel 24 werden Sie noch genügend Beispiele finden, die über `DataContext` eine Datenquelle angeben.

Index

-- -Operator ... 89
!=-Operator ... 91
#define ... 325
#elif ... 325
#elif-Direktive ... 325
#else-Direktive ... 325
#endif ... 325
#endif-Direktive ... 325
#if ... 325
#if-Direktive ... 325
#undef ... 325
%-Operator ... 88
()-Operator ... 99
*-Operator ... 88
++-Operator ... 88
+-Operator ... 88
.-Operator ... 99
/-Operator ... 88
<=-Operator ... 91
<applicationSettings> ... 725
<appSettings> ... 726
<-Operator ... 91
<param> ... 737
<returns> ... 737
<summary> ... 737
<userSettings> ... 726
= = -Operator ... 91
>=-Operator ... 91
->-Operator ... 433
>-Operator ... 91
?:-Operator ... 99, 112

A

Abhängigkeitseigenschaft ... 1044
Abort (Thread) ... 620
abstract ... 218
Abstrakte Klasse ... 218
Abstrakte Methode ... 218
AcceptChanges ... 1213, 1219
AcceptRejectRule ... 1219
AcceptRejectRule (Eigenschaft) ... 1219
AcceptsReturn (WPF) ... 850
AcceptsTab (WPF) ... 850
AccessText (WPF) ... 849
Add ... 348
AddDays ... 694
AddHandler (Routed Event) ... 1070
AddHours ... 694
AddMonths ... 694
AddSeconds ... 694
AddSeconds (DateTime) ... 694
AddWithValue (Methode) ... 1163
AdornedElementPlaceholder ... 994
Aggregatmethoden ... 1314
All ... 461
AllKeys ... 734
AllowDBNull (Eigenschaft) ... 1200
AllowMultiple ... 419, 420
AllowMultiple (Attribut) ... 419
AlternationCount ... 1006
Angehängte Eigenschaft ... 1053
Anonyme Methoden ... 266
Anonyme Typen ... 386
Anweisungsblöcke ... 61
Anweisungstrennzeichen ... 60
Anwendungskonfigurationsdatei ... 721
Any ... 460
App.config ... 727
App.xaml ... 764
App.xaml.cs ... 765
Append ... 687
Append (StringBuilder) ... 688
AppendAllText ... 470
AppendFormat ... 687
AppendLine ... 687
AppendText ... 470, 477
AppendText (File) ... 470
AppendText (FileInfo) ... 477
ApplicationCommands ... 1079
ApplicationException ... 309
ApplicationMenu ... 895
ApplicationSettingsBase ... 728
ApplyCurrentChanges ... 1360
ApplyOriginalChanges ... 1360
AppSettings ... 730
ArgumentException ... 304, 1201
ArgumentNullException ... 304
Arithmetische Operatoren ... 88
Array ... 100, 122
  Deklaration ... 100
  Elementanzahl ... 106
  Elementzugriff ... 102
  Größe ... 104
  mehrdimensional ... 103
  Obergrenze ... 105
ArrayList ... 337
  sortieren ... 341
ArrayList (Klasse) ... 337
Array-Obergrenze ... 105
Assembly ... 48, 705, 709
Assembly signieren ... 719
Assert ... 323

async ... 661
AsyncCallback ... 640, 642
AsyncCallback (Delegat) ... 640
Asynchrone Abfrage ... 1165
Asynchroner Aufruf ... 639
Asynchroner Methodenaufruf ... 639, 640
Asynchronous Processing ... 1166
AsyncState ... 643
Attached Event ... 1069
Attached Property ... 1053
Attribut ... 414
  Auswertung ... 422
  benutzerdefiniertes ... 418
Attribute (Klasse) ... 418
  GetCustomAttribute ... 423
Attributes ... 476
Attributes (FileInfo) ... 476
AttributeTargets ... 419
AttributeTargets (Attribut) ... 419
AttributeTargets (Aufzählung) ... 419
AttributeUsageAttribute ... 418
Aufzählungen ... 295
Ausdruck ... 87
Ausgabeformatierung ... 697
Ausnahme → Exception
Auto (Debuggen) ... 331
Auto (Debugmodus) ... 331
Auto (WPF) ... 796
Auto-Fenster ... 331
AutoFlush ... 500
AutoFlush (StreamWriter) ... 500
AutoGenerateColumns ... 1031
AutoIncrement (Eigenschaft) ... 1207
AutoIncrementSeed (Eigenschaft) ... 1207
AutoIncrementStep (Eigenschaft) ... 1208
Autoinkrementspalten ... 1207
AutoResetEvent ... 629
AvailableFreeSpace ... 484
Average ... 460
await ... 661

B

Background (WPF) ... 790, 841
Band (WPF) ... 877
BandIndex (WPF) ... 877, 878
base ... 213, 214
Base-Stream ... 485, 497
BaseStream ... 500
BaseStream (StreamWriter) ... 500
Bedingte Haltepunkte ... 328
Bedingte Kompilierung ... 324
Bedingungsausdruck ... 119
Befehlsfenster ... 330
BeginChange (WPF) ... 852
BeginEdit (Methode) ... 1205
BeginInit (WPF) ... 888
BeginInvoke ... 640, 641, 648
BeginRead (FileStream) ... 640, 647
BeginStoryBoard ... 962
BeginWrite (FileStream) ... 640, 647
Benannte Parameter ... 421
Benutzerdefinierte Konvertierung ... 403
Benutzerdefinierte WPF-Steuerelement-bibliothek ... 762
Bereiter Thread ... 614
Binäre Datenströme ... 502
BinaryFormatter ... 512, 513
BinaryFormatter (Klasse) ... 512
BinaryReader ... 502
BinaryReader (Klasse) ... 502
BinaryReader (Methoden) ... 502
BinaryWriter ... 502
BinaryWriter (Klasse) ... 502
BinaryWriter (Methoden) ... 502
Binding ... 924
Binding (WPF) ... 783
BindingExpression ... 929
BindingListCollectionView ... 1017
BindingOperations ... 931
BitArray ... 334
BitArray (Klasse) ... 334
BitmapImage ... 888
Bitweise Operatoren ... 95
BlackoutDates (WPF) ... 889
bool ... 75
Boolean ... 75
Boolesche Operatoren ... 91
Border ... 886
BorderBrush (WPF) ... 886
BorderThickness (WPF) ... 886
Boxing ... 298
break ... 116, 126, 130
break (switch-Anweisung) ... 116
Brush ... 1115
Bubbling-Event ... 1059
BufferedStream ... 489
BufferedStream (Klasse) ... 489
Button ... 843
ButtonBase ... 843
ButtonState ... 1073
Byte ... 74
byte ... 74

C

Calendar ... 889
Call by Reference ... 165

Call by Value ... 164
CancelEdit (Methode) ... 1205
CancelEventArgs ... 730
CanGoBack (WPF) ... 823
CanGoForward (WPF) ... 823
CanRead ... 486
CanRead (Stream) ... 486
CanRedo (WPF) ... 852
CanSeek ... 486
CanSeek (Stream) ... 486
CanUndo (WPF) ... 852
Canvas ... 791
CanWrite ... 486
CanWrite (Stream) ... 486
Capacity ... 686
Capacity (StringBuilder) ... 686
Capturing ... 1073
case ... 115
catch ... 302
CellEditingTemplate ... 1035
CellTemplate ... 1035
ChangedButton ... 1073
ChangePassword (Methode) ... 1133
ChangeState ... 1359
Char ... 75
char ... 75
Chars ... 672, 687
Chars (StringBuilder) ... 686
Checkbox ... 847
checked ... 84
Children (WPF) ... 790
Class Designer ... 742
class-Schlüsselwort ... 138
ClearAllBindings ... 931
ClearBinding ... 931
ClearValue ... 1050
ClickCount ... 1073
ClickMode ... 843
ClickMode (WPF) ... 843
ClickOnce ... 753
ClientWins ... 1368
Close (StreamWriter) ... 499
Close (WPF) ... 813
Closed (Methode) ... 1135
Closed (WPF) ... 813
Closing (WPF) ... 813
CLR ... 45
Code Snippet ... 750
Codeausschnitt ... 750
Code-Editor ... 53
CoerceValueCallback ... 1043, 1052
Collapsed (WPF) ... 840, 885
Collect (Klasse GC) ... 252
CollectionsUtil ... 334
CollectionsUtil (Klasse) ... 334
CollectionView ... 1017
ColumnDefinition (WPF) ... 799
ColumnDefinitions ... 799
ColumnDefinitions (WPF) ... 799
Columns ... 805
ColumnSpan (WPF) ... 802
CombinedGeometry ... 1113
ComboBox ... 860
ComboBoxItem ... 860
CommandBuilder ... 1231
CommandText ... 1151
CommandTimeout (Eigenschaft) ... 1152
Common Language Runtime ... 45
Common Language Specification (CLS) ... 42
Common Type System ... 43
Compare ... 344, 673, 674
CompareOrdinal ... 674
CompareOrdinal (String) ... 672
CompareTo ... 341, 673, 674
ComponentCommands ... 1079
Concat (String) ... 681
Conceptual Schema Definition ... 1289
ConditionalAttribute ... 326
Configuration ... 730
ConfigurationManager ... 730
ConfigurationSaveMode ... 734
ConflictOption ... 1236
Connect Timeout ... 1137
ConnectionState ... 1135
ConnectionString-Eigenschaft ... 1133
ConnectionStrings ... 730
Console ... 68
const ... 185
const (Schlüsselwort) ... 185
ConstraintCollection ... 1199
Constraints (Generic) ... 363
ContainsValue ... 352
Content (WPF) ... 836
Content (WPF-Eigenschaft) ... 779
ContentControl ... 778, 780
ContentElement ... 834, 1060
ContentPropertyAttribute ... 778
ContentRendered ... 773
ContentTemplate ... 1008
ContextMenu ... 873
ContextMenu (WPF) ... 873
ContextMenuService ... 874
continue ... 127, 130
continue-Anweisung ... 127
ContinueUpdateOnError ... 1242
Control ... 834
Convert ... 82, 996
  Methoden ... 82
Convert (Klasse) ... 82
ConvertBack ... 996
Converter ... 924, 998
ConverterParameter ... 998

Copy ... 470
Copy (File) ... 470, 471
Copy (WPF) ... 851
CopyTo ... 477, 687
CopyTo (FileInfo) ... 477
CornerRadius (WPF) ... 886
Count ... 457
Create ... 470, 477
Create (File) ... 470
Create (FileInfo) ... 477
CreateCommand (Methode) ... 1151
CreateDirectory ... 478
CreateDirectory (Directory) ... 478
CreateText ... 470, 477
CreateText (File) ... 470
CreateText (FileInfo) ... 477
CreationTime ... 476
CreationTime (FileInfo) ... 476
CryptoStream ... 489
CSDL ... 1289
CTS ... 43
CultureInfo ... 698
Cursor (WPF) ... 791
Cut (WPF) ... 851

D

Data (Exception) ... 310, 311
DataColumn ... 1192
DataColumnCollection ... 1192
DataColumnMappingCollection ... 1188
DataContext ... 931, 934, 977
DataErrorValidationRule ... 992
DataGrid ... 1030
DataGridCheckBoxColumn ... 1033
DataGridComboBox ... 1033
DataGridHyperLinkColumn ... 1033
DataGridTemplateColumn ... 1033
DataGridTextColumn ... 1033
DataMemberAttribute ... 1294
DataRelation ... 1216
DataRow ... 1193
DataRowCollection ... 1193
DataRowState ... 1210
DataRowVersion ... 1211
DataSet ... 1186, 1191
DataTable ... 1183, 1192
DataTemplate ... 1006, 1008
DataTrigger ... 960, 1010
DataView ... 1224
Datei
  kopieren ... 471
  löschen ... 472
  öffnen ... 473
Daten binär speichern ... 502
Datenfeld ... 100
Datenfeld (Array) ... 100
Datenfelder ... 100
Datenkapselung ... 147
Datenprovider ... 1129
Datentypen ... 74
DatePicker ... 889
DateTime ... 690
  AddSeconds ... 694
  Day ... 692
  DayOfWeek ... 692
  Eigenschaften ... 692
  Hour ... 692
  Methoden ... 693
  MilliSecond ... 692
  Minute ... 692
  Month ... 692
  Second ... 692
  Ticks ... 690
  Today ... 693
  ToFileTime ... 693
  ToLongDateString ... 693
  ToLongTimeString ... 693
  ToShortDateString ... 693
  ToShortTimeString ... 693
  UtcNow ... 692
  Year ... 692
DateTimeFormatInfo ... 698, 700
Day ... 692
Day (DateTime) ... 692
DayOfWeek ... 692
DayOfWeek (DateTime) ... 692
DBConcurrencyException ... 1242, 1247
Deadlock ... 634
Debug ... 320, 321
  Assert ... 323
  IndentLevel ... 322
  Write ... 322
  WriteIf ... 322
  WriteLine ... 321
  WriteLineIf ... 322
Debug (Klasse) ... 320
  Methoden ... 322
Debug (Schalter) ... 324
Debugging ... 319
Debug-Konfiguration ... 324
Decimal ... 75
decimal ... 75
default ... 115
default (Generic) ... 366
default (switch-Anweisung) ... 115
DefaultIfEmpty ... 466
Deferred Loading ... 1320
Delay (WPF) ... 845
delegate ... 262
delegate (Schlüsselwort) ... 261

Delete ... 470, 477, 478
Delete (Directory) ... 478
Delete (File) ... 470
Delete (FileInfo) ... 477
Delete (Methode) ... 1206
DeleteCommand ... 1232, 1239
DeleteObject-Methode ... 1341
Dependency Properties ... 768
Dependency Property ... 1041
DependencyObject ... 834, 1045
DependencyProperty ... 923, 1044, 1100
DependencyPropertyChangedEventArgs ... 1049
Dequeue ... 355
Deserialize ... 512
Deserialize (Methode) ... 512
Destruktor ... 184
DetectChanges ... 1377
Dezimalzahlen ... 76
Dialogfenster ... 812
DictionaryEntry ... 351
DiffHours ... 695
DiffSeconds ... 695
Directory ... 476, 478
CreateDirectory ... 478
Delete ... 478
Exists ... 478
GetCreationTime ... 478
GetDirectories ... 479
GetFiles ... 479
GetFileSystemEntries ... 479
GetParent ... 479
Move ... 479
SetCreationTime ... 479
Directory (FileInfo) ... 476
Directory (Klasse) ... 478
DirectoryInfo ... 478
DirectoryInfo (Klasse) ... 478
DirectoryName ... 476
DirectoryName (FileInfo) ... 476
DirectoryNotFoundException ... 304, 471
Direkter Event ... 1059
Direktfenster ... 330
DispatcherObject ... 834
DisplayDate (WPF) ... 889
DisplayDateStart (WPF) ... 889
DisplayMemberBinding (WPF) ... 862
DisplayMemberPath ... 982, 1006
DisplayMode (WPF) ... 889, 890
Distinct ... 456
DockPanel ... 796
DockPanel (WPF) ... 871
DocumentViewer ... 913
do-Schleife ... 129, 130
Double ... 74
double ... 74
DrawingBrush ... 841, 1125
DriveFormat ... 484
DriveInfo ... 484
DriveType ... 484
DropDownClosed (WPF) ... 861
DropDownOpened (WPF) ... 861
dynamic ... 426
Dynamisches Binden ... 426

E

Eager Loading ... 1323
EditingCommands ... 1079
EditingMode (WPF) ... 891
EDM ... 1279
EdmScalarProperty ... 1294
Eigenschaft ... 134
Eigenschaft-Element-Syntax ... 777
Eigenschaften
lesegeschützt ... 149
readonly ... 186
schreibgeschützt ... 149
set-Accessor ... 150
Eigenschaftsfenster ... 55
Einfache Datentypen ... 74
Einzelschritt ... 328
Elementare Datentypen ... 74
ElementAt ... 465
Elementbaum ... 770
ElementName ... 923
ElementName (WPF) ... 783
ElementOrDefault ... 465
EllipseGeometry ... 1111
else ... 109, 111
Encoding ... 500
Encoding (StreamWriter) ... 500
EndChange (WPF) ... 852
EndEdit (Methode) ... 1205
EndInit (WPF) ... 888
EndInvoke ... 640, 644, 648
EndRead ... 640
EndRead (FileStream) ... 640, 647
EndsWith ... 675
EndsWith (String) ... 675
EndWrite ... 640
EndWrite (FileStream) ... 640, 647
Enqueue ... 355
EnsureCapacity ... 687
Enter ... 631
Enter (Monitor) ... 631
Entitätscontainername ... 1288
Entitätsklasse ... 1293
Entity Data Model ... 1279
Entity SQL ... 1324
EntityClient-Provider ... 1329
EntityCollection ... 1311

EntityConnection ... 1330
EntityDataReader ... 1329
EntityKey ... 1345, 1357
EntityObject ... 1371
EntityState ... 1347
Enum
GetValues ... 297
enum ... 295, 296
Enumerationen ... 295
Equals ... 665, 666
Ereignis ... 270
Ereignis (allgemein) ... 270
Ereignisempfänger ... 270
Ereignishandler ... 274
Ereignisquelle ... 270
ErrorsTemplate ... 993
Erweiterungsmethode ... 389
Escape-Zeichen ... 72
event ... 282
event (Schlüsselwort) ... 272
EventManager ... 1102
Except ... 457
Exception ... 300
HelpLink ... 310
InnerException ... 310
Source ... 310
StackTrace ... 310
TargetSite ... 310
Exception (allgemein) ... 300
benutzerdefiniert ... 315
weiterleiten ... 307
Exception (Klasse)
Eigenschaften ... 310
ExceptionValidationRule ... 989
ExecuteNonQuery ... 1152
ExecuteNonQuery (Methode) ... 1152
ExecuteReader ... 1155
ExecuteReader (Methode) ... 1152
ExecuteStoreCommand-Methode ... 1333
ExecuteStoreQuery-Methode ... 1334
ExecuteXmlReader ... 1152
Exists ... 470, 472, 477, 478
Exists (Directory) ... 478
Exists (File) ... 470, 472
Exists (FileInfo) ... 475, 477
Exit ... 631
Exit (Monitor) ... 631
Expanded (WPF) ... 885
ExpandedDirection (WPF) ... 885
Expander ... 885
explicit ... 405
Explizite Implementierung (interface) ... 242
Explizite Konvertierung ... 81
Explizite Typumwandlung ... 222
Extension ... 476
Extension (FileInfo) ... 476
ExtraData (WPF) ... 828

F

Factory-Methode ... 1338
FallbackValue ... 924
Feld ... 134
Feld (Eigenschaft) ... 145
File ... 469
AppendText ... 470
Copy ... 470
Create ... 470
CreateText ... 470
Delete ... 470
Exists ... 470
GetAttributes ... 470
GetCreationTime ... 470
GetLastWriteTime ... 470
Move ... 470
Open ... 470
OpenRead ... 470, 473
OpenText ... 470, 473
OpenWrite ... 470, 473
SetAttributes ... 471
SetCreationTime ... 471
SetLastAccessTime ... 471
SetLastWriteTime ... 471
File (Methoden) ... 470
FileAccess ... 474
FileAccess (Aufzählung) ... 474
FileInfo ... 475
AppendText ... 477
Attributes ... 476
CopyTo ... 477
Create ... 477
CreateText ... 477
CreationTime ... 476
Delete ... 477
Directory ... 476
DirectoryName ... 476
Exists ... 475, 477
Extension ... 476
FullName ... 476
LastAccessTime ... 476
LastWriteTime ... 476
Length ... 476
MoveTo ... 478
Name ... 476
Open ... 478
OpenRead ... 478
OpenText ... 478
OpenWrite ... 478
FileInfo (Klasse) ... 475
FileLoadException ... 735

FileMode ... 473
FileMode (Aufzählung) ... 473
FileNotFoundException ... 304
FileShare ... 474
FileShare (Aufzählung) ... 474
FileStream ... 489, 639, 647
BeginRead ... 640
BeginWrite ... 640
EndRead ... 640
EndWrite ... 640
Lesen aus ... 491
Lesen von Textdatei mit ... 494
Schreiben in ... 490
FileStream (Klasse) ... 490
Fill (Methode) ... 1180
FillError (Ereignis) ... 1188
FillSchema (Methode) ... 1198, 1201
Finalize ... 665
finally ... 308
Find
DataView ... 1226
FindRows
DataView ... 1226
First ... 463
FirstDayOfWeek (WPF) ... 890
First-Methode ... 1306
FirstOrDefault ... 464
FirstOrDefault-Methode ... 1306
fixed ... 431
Flags (Attribut) ... 415
FlagsAttribute ... 415
float ... 74
FlowDirection ... 794
FlowDirection (WPF) ... 794, 805
FlowDocumentPageViewer ... 912
FlowDocumentReader ... 912
FlowDocuments ... 899
FlowDocumentScrollViewer ... 911
FlowDocumentViewer ... 911
Flush (StreamWriter) ... 499
Focusable (WPF) ... 791
FontFamily (WPF) ... 842
FontSize (WPF) ... 842
FontStretch (WPF) ... 842
FontStyle (WPF) ... 842
FontWeight (WPF) ... 842
foreach-Schleife ... 128, 170
Foreground (WPF) ... 841
ForeignKeyConstraint ... 1199
Format (String) ... 697, 699
Formatangaben ... 71
Formatausdruck ... 69
Formatausdruck (Konsolenausgabe) ... 70
Formatausgabe (Tabelle) ... 71
Formatierung ... 701
Formatierung (Datum und Zeit) ... 703
for-Schleife ... 118
Frame ... 817
FrameworkContentElement ... 834
FrameworkElement ... 834
FrameworkPropertyMetadata ... 1047, 1050, 1101
FrameworkPropertyMetadataOptions ... 1050
Freezable ... 834
Friend Assembly ... 707
from ... 443, 445
FullName ... 476
FullName (FileInfo) ... 476
Func ... 438

G

GAC ... 711, 716
gacutil.exe ... 720
Ganzzahlige Datentypen ... 75
Garbage Collector (Thread) ... 616
GC
Collect ... 252
Gemeinsame Assembly ... 710
Generic
Constraints ... 363
default ... 366
generische Methoden ... 367
Generische Delegates ... 371
Generischer Typparameter ... 361
Geometry ... 1111
GeometryGroup ... 1112
get-Accessor ... 150
GetAttributes ... 470
GetAttributes (File) ... 470
GetChildrenCount ... 772
GetChildRows ... 1219
GetCreationTime ... 470, 478
GetCreationTime (Directory) ... 478
GetCreationTime (File) ... 470
GetCustomAttribute ... 423
GetDirectories ... 479
GetDirectories (Directory) ... 479
GetDirectoryName ... 482
GetDirectoryName (Path) ... 482
GetEnumerator ... 356
GetExtension ... 482
GetExtension (Path) ... 482
GetFileName ... 482
GetFileName (Path) ... 482
GetFileNameWithoutExtension ... 483
GetFileNameWithoutExtension (Path) ... 483
GetFiles ... 479
GetFiles (Directory) ... 479
GetFileSystemEntries ... 479
GetFileSystemEntries (Directory) ... 479
GetFixedDocumentSequence (WPF) ... 914

GetFullPath ... 483
GetFullPath (Path) ... 483
GetHashCode ... 665
GetLastAccessTime ... 470
GetLastAccessTime (File) ... 470
GetLastWriteTime ... 470
GetLastWriteTime (File) ... 470
GetLength ... 105
GetLogicalTree ... 772
GetMaxThreads ... 628
GetModifiedProperties ... 1356
GetName (Methode) ... 1162
GetObjectByKey ... 1357
GetOrdinal (Methode) ... 1160
GetParent ... 479
GetParent (Directory) ... 479
GetParentRow ... 1219
GetPathRoot ... 483
GetPathRoot (Path) ... 483
GetSchemaTable (Methode) ... 1160
GetSection ... 730
GetTempFileName ... 483
GetTempFileName (Path) ... 483
GetTempPath ... 483
GetTempPath (Path) ... 483
GetType ... 665, 666
GetValue ... 1045
GetValues (Klasse Enum) ... 297
GetVisualTree ... 772
Gliederungsblöcke ... 61
Global Assembly Cache ... 711, 716
Global Assembly Cache → GAC
Globale Assembly ... 710, 716, 720
GoBack (WPF) ... 823
GoForward (WPF) ... 823
goto ... 117
goto-Anweisung ... 116
Grid ... 798
Grid.Column ... 802
Grid.Row ... 802
GridSplitter ... 802
GridView ... 861
GridViewColumn (WPF) ... 861
Groß- und Kleinschreibung ... 63
GroupBox ... 882
GroupBy ... 451
GroupJoin ... 454
GroupName (WPF) ... 847
GroupStyle ... 1006
GZipStream ... 489

H

Haltemodus ... 327, 328
Haltepunkt ... 328
Handled (Routed Event) ... 1063
HasDropShadow (WPF) ... 874
HasErrors (Eigenschaft) ... 1243
Hashtable ... 349
Hashtable (Klasse) ... 349
Header (WPF) ... 857
HeaderedContentControl ... 780
HeaderedItemControls ... 857
HeaderedItemsControl ... 780
Heap ... 144
Height (WPF) ... 791
HelpLink ... 310
HelpLink (Exception) ... 313
HelpPaneContent ... 895
Herausgeberrichtliniendatei ... 721, 735
Hidden (WPF) ... 840
Hide (WPF) ... 813
Hintergrundthread ... 627
HorizontalAlignment (WPF) ... 791, 839
HorizontalContentAlignment (WPF) ... 840
HorizontalScrollBarVisibility (WPF) ... 850
Hour ... 692
HybridDictionary ... 334
HybridDictionary (Klasse) ... 334
HyperLink (Eigenschaft) ... 819

I

IAsyncResult ... 640, 645
ICollection ... 335, 336
IComparer ... 343
Icon (WPF) ... 811
IDataErrorInfo ... 991
IDictionary ... 335, 348
Keys ... 349
Remove ... 348
Values ... 349
IDictionary (Eigenschaften und Methoden) ... 349
IEnumerable ... 335, 356
IEnumerable<T> ... 440
if ... 108
if-Anweisung ... 108
IFormatable ... 697
IFormatProvider ... 698
IFormatter
Serialize ... 512
ILDASM.EXE ... 713
ildasm.exe ... 713
IL-Disassembler ... 713
IList ... 336
Image ... 887
Image (WPF) ... 887
ImageBrush ... 841, 1122
implicit ... 404
Implizite Konvertierung ... 81

Implizite Typumwandlung ... 220
IMultiValueConverter ... 999
Include-Methode ... 1323
Indent ... 322
IndentLevel ... 322
Indexer ... 407
IndexOf ... 675
IndexOf (String) ... 675
InfoMessage (Ereignis) ... 1142
Inhaltseigenschaften ... 778
Inherited ... 419, 420
Inherited (Attribut) ... 419
Inherits ... 1050
Initialisierung ... 66, 141
Initialisierungsausdruck ... 119
InitializeComponent ... 775
InitialShowDelay (WPF) ... 881
InkCanvas ... 890
InkCanvasEditingMode (WPF) ... 891
Inline-Elemente ... 905
Inner Joins (EF) ... 1315
InnerException ... 310, 313
INotifyCollectionChanged ... 984
INotifyPropertyChanged ... 284, 980, 984
InputGestureText (WPF) ... 872
Insert ... 679, 687
Insert (String) ... 679, 680
Insert (StringBuilder) ... 688
InsertCommand ... 1232, 1239
Inserts (String) ... 679
Instanziierung ... 183
int ... 74
Int16 ... 74
Int32 ... 74
Int64 ... 74
Interface ... 237
interface ... 238
internal ... 140, 154
  Klasse ... 140
internal-Konstruktoren ... 181
InternalsVisibleTo ... 708
Intersect ... 457
Interval (WPF) ... 845
InvalidCastException ... 360
InvalidOperationException ... 1135
InvokeMember ... 426
IOException ... 304, 469
is (Operator) ... 223
IsAlive ... 621
IsAlive (Thread) ... 621
IsAsync ... 924
IsBackground ... 627
IsCancel (WPF) ... 814, 843
IsChecked (WPF) ... 845
IsCompleted ... 643
IsDBNull ... 1157
IsDefault (WPF) ... 814, 843
IsDefaulted (WPF) ... 844
IsDirectionReversed (WPF) ... 882
IsEditable (WPF) ... 860
IsHyphenationEnabled (WPF) ... 853, 900
IsIndeterminate ... 881
IsMoveToPointEnabled (WPF) ... 882
is-Operator ... 99, 223
IsOptimalParagraphEnabled (WPF) ... 900
IsPressed (WPF) ... 843
IsReady ... 484
IsSnapToTickEnabled (WPF) ... 881
Ist-ein(e)-Beziehung ... 208
IsThreeState (WPF) ... 845
ItemArray (Eigenschaft) ... 1205
ItemCollection ... 1017
ItemContainerStyle ... 1006, 1008
ItemControls (WPF) ... 857
ItemHeight (WPF) ... 796
Items (WPF) ... 857
ItemsControl ... 780
ItemSource (WPF) ... 862
ItemsSource ... 982, 1006
ItemTemplate ... 982, 1006
ItemWidth (WPF) ... 796
IValueConverter ... 996

J

JIT ... 41
Join ... 453, 622
Join (Thread) ... 622, 624
Journal (WPF) ... 820

K

KeepAlive (WPF) ... 817
Keys ... 349
Keys (IDictionary) ... 349
Klasse ... 133
  hinzufügen ... 136
  Zugriffsmodifizierer ... 140
Klassenbibliothek ... 705
Klassendefinition ... 136
Klassendesigner ... 742
Klassendetails ... 745
Kommentar ... 62
Komplexe Eigenschaft ... 1286
Konfigurationsdateien ... 721
Konsolenanwendung ... 57, 64
  Main-Prozedur ... 64
Konstanten ... 185
Konstruktor ... 178, 208

Konstruktorverkettung ... 182, 210
Kontravarianz ... 269, 379
Kontrollstrukturen ... 108
Konvertierung ... 80, 220
Konvertierung mit der Klasse Convert ... 82
Konvertierungsoperator ... 81
Kovarianz ... 268, 379

L

Label ... 848
Lambda-Ausdruck ... 387
Language (WPF) ... 852
Last ... 464
LastAccessTime ... 476
LastAccessTime (FileInfo) ... 476
LastChildFill ... 797
LastChildFill (WPF) ... 797
LastIndexOf ... 675
LastIndexOf (String) ... 675
LastOrDefault ... 464
LastWriteTime ... 476
LastWriteTime (FileInfo) ... 476
Laufender Thread ... 614
Laufzeitfehler ... 300
LayoutTransform (WPF) ... 869
Lazy Loading ... 1318
Left Outer Joins (EF) ... 1316
LeftToRight ... 794
LeftToRight (WPF) ... 794
Length ... 106
Length (FileInfo) ... 476
Length (Stream) ... 486
Length (String) ... 672
Lesegeschützte Eigenschaften ... 149
Lesen aus FileStream ... 491
Line ... 1108
LinearGradientBrush ... 841, 1117
LineBreak (WPF) ... 853
LineRight (WPF) ... 883
LINQ ... 435
ListBox ... 857
ListBoxItems ... 858
ListCollectionView ... 1017
ListDictionary ... 334
ListDictionary (Klasse) ... 334
ListView ... 861
Literale Initialisierung ... 101
Literale Initialisierung (Array) ... 101
LoadCompleted (WPF) ... 824
Load-Methode ... 1322
lock ... 632
lock (Thread) ... 632
lock-Anweisung ... 632
LogicalTreeHelper ... 772
Logische Operatoren ... 91, 92
Logischer Elementbaum ... 770
Lokal (Debuggen) ... 332
Lokal (Debugmodus) ... 332
Lokale Variable ... 162
Lokal-Fenster ... 332
long ... 74
LongCount ... 457

M

machine.config ... 721
Main (Klasse Console) ... 124
Main-Prozedur ... 64, 123
MainWindow.xaml ... 763
MainWindow.xaml.cs ... 764
Manifest ... 50, 710, 712
Mapping Specification ... 1289
Margin ... 835
Margin (WPF) ... 791, 835
MarkerOffset (WPF) ... 903
MarkerStyle (WPF) ... 903
Markup Extensions ... 782
Markup-Erweiterungen (WPF) ... 788
MARS (Multiple Active Resultsets) ... 1158
Maschinenkonfigurationsdatei ... 721
Max ... 460
MaxCapacity ... 686
MaxCapacity (StringBuilder) ... 686
MaxHeight (WPF) ... 791
MaxLength (Eigenschaft) ... 1200
MaxLines (WPF) ... 849
MaxWidth (WPF) ... 791
MediaCommands ... 1079
Mehrdimensionale Arrays ... 103
Mehrfachvererbung ... 206
MemberwiseClone ... 665, 667, 668
MemoryStream ... 489
MemoryStream (Klasse) ... 489
Menu ... 870
Menu (WPF) ... 870
MenuItem ... 870
MenuItem (WPF) ... 870
MergeOption ... 1349
Message ... 310
Message (Exception) ... 310
MessageBox ... 828
MessageBox (Klasse) ... 829
MessageBoxButton (WPF) ... 829
MessageBoxButtons ... 829, 830
MessageBoxButtons (Aufzählung) ... 829, 830
MessageBoxImage (WPF) ... 830
MessageBoxResult (WPF) ... 830
Metadaten ... 49

Methode 135, 153
Parameterliste 157
Referenzparameter 163
Wertparameter 163
Methoden mit Parameterliste 157
Methodenaufruf 156
MethodImplAttribute 637
Millisecond 692
Min 460
MinHeight (WPF) 791
MinLines (WPF) 849
Minute 692
MinWidth (WPF) 791
MissingMappingAction (Eigenschaft) 1188
MissingMappingAction (Enumeration) 1188
MissingSchemaAction (Eigenschaft) 1198
MissingSchemaAction (Enumeration) 1203
Modale Fenster 813
Mode 924
Monitor 631
Enter 631
Exit 631
Wait 633
Monitor (Klasse) 631
Monitor (Threadsynchronisation) 631
Month 692
MouseEnter 1071
MouseEventArgs 1063
MouseLeave 1071
MouseMove 1071
MouseRightButtonDown 1060
Move 470, 479
Move (Directory) 479
Move (File) 470, 472
MoveCurrentToFirst 1018
MoveCurrentToLast 1018
MoveCurrentToNext 1018
MoveCurrentToPosition 1018
MoveCurrentToPrevious 1018
MoveNext 356
MoveTo 478
MoveTo (FileInfo) 478
mscorlib.dll 714
MSIL-Code 41
MSL 1289
MultiBinding 999
MultiDataTrigger 1010
MultipleActiveResultSets 1158
Multithreading 614
MultiTrigger 959
Mutex 638

N

Name (FileInfo) 476
Namensbereiche 193
Namespace 47, 193
NameValueCollection 334, 734
Navigate (WPF) 822
Navigated (WPF) 824
Navigating (WPF) 823
NavigationCommands 1079
NavigationFailed (WPF) 824
NavigationProgress (WPF) 824
Navigationseigenschaft 1282, 1286, 1310
NavigationService 822
NavigationService (WPF) 819
NavigationStopped (WPF) 824
NavigationWindow 809, 814
NetworkStream 489
NetworkStream (Stream) 489
new 134
new (Generic) 366
new (Instanziierung) 139
new (Modifizierer) 216, 229
new-Modifikator 216
new-Operator 99
NewRow (Methode) 1206
NewValue (WPF) 865
newVersion (Attribut) 735
NextResult (Methode) 1159
NonSerialized (Attribut) 514
NonSerializedAttribut 514
NonSerializedAttribute 514
NotifyOnSourceUpdate 924
NotifyOnSourceUpdated 930
NotifyOnTargetUpdate 924
NotifyOnTargetUpdated 930
NotifyOnValidationError 995
Now 690, 692
Now (DateTime) 692
Nullable<T> 373
Nullable-Typen 373
NumberFormatInfo 698

O

Object 75, 206, 665
Equals 665
GetHashCode 665
GetType 665
Methoden 665
ReferenceEquals 665
ToString 230, 665
object 75
Object Services 1298
ObjectContext 1288, 1296, 1335
ObjectDataProvider 1014
ObjectQuery 1306
Object-relational Mapping 1279
ObjectSet 1303

ObjectSet<T> ... 1297
ObjectStateEntry ... 1345, 1352
ObjectStateManager ... 1345
Objekt ... 133
Objektmethoden ... 153
Objektorientierung ... 133
Objektvariable ... 139
ObservableCollection ... 984
ObservableCollection<T> ... 984
OldValue (WPF) ... 865
oldVersion (Attribut) ... 735
OleDbCommand ... 1149
OnContentRendered ... 773
Open ... 478
Open (File) ... 470
Open (FileInfo) ... 478
Open (Methode) ... 1135
OpenExeConfiguration ... 731
OpenMachineConfiguration ... 731
OpenRead ... 470, 478
OpenRead (File) ... 470, 473
OpenRead (FileInfo) ... 478
OpenText ... 470, 478
OpenText (File) ... 470, 473
OpenText (FileInfo) ... 478
OpenWrite ... 470, 478
OpenWrite (File) ... 470, 473
OpenWrite (FileInfo) ... 478
Operanden ... 88
--Operator ... 88
-Operator ... 99
operator ... 396
Operatoren ... 87
Operatorüberladung ... 396, 397
Operator-Vorrangregeln ... 99
OptimisticConcurrencyException ... 1367
OrderBy ... 450
OrderByDescending ... 450
Orientation ... 794
Orientation (WPF) ... 792
OriginalSource (Routed Event) ... 1063
out (Schlüsselwort) ... 165
OverflowMode (WPF) ... 876
override ... 229

P

Pad ... 677
Padding ... 835
Padding (WPF) ... 835
PadLeft ... 678
PadLeft (String) ... 678
PadRight ... 678
PadRight (String) ... 678
Page (WPF) ... 810, 815, 816
Paging (EF) ... 1307
Paragraph (WPF) ... 901
ParameterDirection (Enumeration) ... 1175
ParameterizedThreadStart ... 619
Parameterliste ... 157
Parameters-Auflistung ... 1163
Parametrisierte Abfrage ... 1162
params (Schlüsselwort) ... 170
params-Parameter ... 170
Parent (WPF) ... 864
partial ... 140
partial (Schlüsselwort) ... 140
Partielle Methoden ... 393
Pass-Through-Stream ... 485
Password (WPF) ... 853
PasswordBox ... 852
PasswordBox (WPF) ... 852
PasswordChanged (WPF) ... 853
PasswordChar (WPF) ... 853
Paste (WPF) ... 851
Path ... 482, 923, 1111
GetDirectoryName ... 482
GetExtension ... 482
GetFileName ... 482
GetFileNameWithoutExtension ... 483
GetFullPath ... 483
GetPathRoot ... 483
GetTempFileName ... 483
GetTempPath ... 483
Path (Klasse) ... 482
Path (WPF) ... 923
PathGeometry ... 1114
Placement (WPF) ... 874
PlacementRectangle (WPF) ... 881
PlacementTarget (WPF) ... 881
POCO ... 1371
Polling-Verfahren ... 1166
Polygon ... 1109
Polyline ... 1109
Pop ... 354
Position (Stream) ... 486
Positionale Parameter ... 421
Prädikat ... 389
Predicate<T> ... 1022
PreviewMouseMove ... 1071
PreviewMouseRightButtonDown ... 1060, 1062
PrimaryKey (Eigenschaft) ... 1200
Priority ... 625
Priority (Thread) ... 625
private ... 154, 210
Private Assembly ... 710
private-Konstruktoren ... 181
Programmschleifen ... 117
Progressbar ... 881
Projektion ... 389, 1305

Projektmappe ... 706
Verwalten mehrerer Projekte ... 706
Projektmappen-Explorer ... 54
PropertyChanged ... 284, 980
PropertyChangedCallback ... 1049
PropertyChangedEventArgs ... 980
PropertyGroupDescription ... 1026
PropertyMetadata ... 1047, 1052
protected ... 154, 209, 210
protected internal ... 154
Prozedurschritt ... 328
Prüfen auf Initialisierung ... 141
public ... 140, 154
Klasse ... 140
Pulse ... 633
Pulse (Monitor) ... 633
PulseAll (Monitor) ... 634
Punktnotation ... 69
Push ... 354

Q

Queue ... 353
Queue (Klasse) ... 353
QueueUserWorkItem ... 627
QuickAccessToolBar ... 895

R

RadialGradientBrush ... 841, 1119
RadioButton ... 847
Read ... 68
Console ... 68, 73
Read (Klasse Console) ... 69
Read (Stream) ... 487
Read (StreamReader) ... 500
ReadAllBytes ... 470
ReadAllLines ... 470
ReadAllText ... 471
ReadByte ... 487
ReadByte (Stream) ... 487, 488
ReadKey ... 58
ReadLine ... 59, 68, 73
ReadLine (Klasse Console) ... 69
readonly ... 186
ReadOnly (Eigenschaft) ... 1200
readonly (Schlüsselwort) ... 186
readonly-Modifikator ... 186
ReadToEnd (StreamReader) ... 500
RectangleGeometry ... 1111
Redo (WPF) ... 851
ref (Schlüsselwort) ... 165
Refactoring ... 747
ReferenceEquals ... 665, 666
Referenz ... 139
Referenzparameter ... 163
Referenztyp ... 143, 144
Refresh ... 1369
RefreshSection ... 731
Register ... 1044
RegisterAttach ... 1054
RegisterRoutedEvent ... 1102
Reinitialisierungsausdruck ... 119
RejectChanges ... 1213, 1219
RelativeSource ... 924, 931
Release (Schalter) ... 324
Release-Konfiguration ... 324
Remove ... 348, 679, 687
Remove (IDictionary) ... 348
Remove (String) ... 679
Remove (StringBuilder) ... 689
RepeatButton ... 845
Replace ... 679, 680, 687
Replace (String) ... 679, 680
Replace (StringBuilder) ... 689
ReportPropertyChanged ... 1295
ReportPropertyChanging ... 1295
RequestNavigate (WPF) ... 825
Reset ... 356
ResetAbort (Thread) ... 622
ResizeMode (WPF) ... 811
return ... 132, 155
Reverse ... 451
Ribbon ... 893
RibbonButton ... 896
RibbonGroup ... 898
RibbonTab ... 894
RichTextBox ... 915
Right Outer Join (EF) ... 1317
RightToLeft ... 794
RightToLeft (WPF) ... 794
RootDirectory ... 484
RotateTransform (WPF) ... 869
Routed Events ... 769, 1057
RoutedEventArgs ... 1063
Routing-Strategien ... 1058
RowChanged (Ereignis) ... 1205
RowChanging (Ereignis) ... 1205
RowDefinition (WPF) ... 799
RowDefinitions ... 799
RowDefinitions (WPF) ... 799
RowDetailsTemplate ... 1039
RowFilter
DataView ... 1227
Rows ... 805
RowSpan (WPF) ... 802
RowState (Eigenschaft) ... 1209
RowStateFilter ... 1227
RowUpdated ... 1244

RowUpdating ... 1244
Rule (Enumeration) ... 1219

S

SaveChanges-Methode ... 1337
SByte ... 74
sbyte ... 74
Schemainformationen ... 1196
SchemaType (Enumeration) ... 1202
Schleifen ... 117
Schlüsseldatei ... 718
Schreiben in FileStream ... 490
Schreibgeschützte Eigenschaften ... 149
ScrollToLeftEnd (WPF) ... 883
ScrollViewer ... 883
sealed ... 208, 231
Second ... 692
Seek ... 487, 493
Seek (Stream) ... 487
SeekOrigin ... 487
SeekOrigin (Aufzählung) ... 487
Select ... 449
select ... 443
SelectAll (WPF) ... 851
SelectCommand (Eigenschaft) ... 1179
SelectedDate (WPF) ... 890
SelectedDateFormat (WPF) ... 889
SelectedDates (WPF) ... 890
SelectedIndex (WPF) ... 858, 859
SelectedItem (WPF) ... 858, 859, 865
SelectedItemChanged (WPF) ... 865
SelectedItems (WPF) ... 858, 859
SelectedText ... 851
SelectedText (WPF) ... 851
SelectedValue (WPF) ... 865
SelectedValuePath (WPF) ... 865
SelectionLength (WPF) ... 851
SelectionMode (WPF) ... 857, 890
SelectionStart (WPF) ... 851
SelectMany ... 449
Separator (WPF) ... 870
Sequenz ... 1302
Serialisierung ... 511
Serialisierung im XML-Format ... 596
Serializable (Attribut) ... 513
SerializableAttribute ... 513
SerializationException ... 513
Serialize ... 512
Serialize (Methode) ... 512
Server-Explorer ... 56
set-Accessor ... 150
SetAdded (Methode) ... 1214
SetAllValues ... 1237
SetAttributes ... 471
SetAttributes (File) ... 471
SetBinding ... 923
SetCreationTime ... 471, 479
SetCreationTime (Directory) ... 479
SetCreationTime (File) ... 471
SetLastAccessTime ... 471
SetLastAccessTime (File) ... 471
SetLastWriteTime ... 471
SetLastWriteTime (File) ... 471
SetLoadCompletedHandler (WPF) ... 828
SetModified (Methode) ... 1214
Settings ... 728
SettingsChanging ... 729
SettingsSaving ... 729
SetValue ... 1045
Shape ... 1107
short ... 74
Show (Methode der MessageBox) ... 829
Show (WPF) ... 812
ShowDialog (WPF) ... 812
ShowDuration (WPF) ... 881
ShowInTaskBar (WPF) ... 811
ShowNavigationUI (WPF) ... 816
ShowPreview (WPF) ... 803
Single ... 74, 465
Single-Methode ... 1306
SingleOrDefault ... 465
SingleOrDefault-Methode ... 1306
SizeToContent (WPF) ... 811
Skalare Eigenschaft ... 1284
Skip ... 462
SkipWhile ... 462
Slider ... 881
SLN-Datei ... 706
sn.exe ... 719
SoapFormatter ... 512
SolidColorBrush ... 841, 1116
Sort ... 344
DataView ... 1226
SortDescription ... 1021
SortDescriptions ... 1021
SortedList ... 334
Source ... 310, 924, 931
Source (Exception) ... 310
Source (Routed Event) ... 1063
Source (WPF) ... 887
SourceColumn (Eigenschaft) ... 1240
SourceUpdated ... 930
SourceVersion (Eigenschaft) ... 1240
SpellCheck (WPF) ... 852
SpellingReform (WPF) ... 852
Split ... 679, 680
Split (String) ... 679, 680
SqlClient-Provider ... 1128
SqlCommand ... 1149

SqlConnection ... 1129
SqlConnectionStringBuilder ... 1134
SqlDataAdapter ... 1149, 1177
SqlDataReader ... 1149, 1154
SqlException ... 1164
SqlParameter ... 1163, 1165
SqlRowUpdatedEventArgs ... 1245
SSDL ... 1289
Stack ... 353
Stack (Klasse) ... 353
StackPanel ... 792
StackTrace ... 310
StackTrace (Exception) ... 310
Standardkonstruktor ... 179
Standard-Namespace (WPF) ... 785
Start (Thread) ... 618
Startprojekt festlegen ... 706
StartsWith ... 675
StartsWith (String) ... 675
StartupUri ... 764, 812
State (Eigenschaft) ... 1135
StateChange (Ereignis) ... 1143
static ... 187, 189
Statische Klasseninitialisierer ... 190
Statischer Konstruktor ... 190
StatusBar ... 878
StatusBar (WPF) ... 878
StatusBarItem ... 878
StopLoading (WPF) ... 823
StopStoryBoard ... 962
Storage Schema Definition ... 1289
StoreWins ... 1368
Stream ... 485
CanRead ... 486
CanSeek ... 486
CanWrite ... 486
Close ... 488
Length ... 486
Position ... 486
Read ... 487
ReadByte ... 487, 488
Seek ... 487
Write ... 487
WriteByte ... 487, 488
StreamReader ... 300, 500
Read ... 300, 500
ReadLine ... 300
ReadToEnd ... 300, 500
StreamWriter ... 496
AutoFlush ... 500
BaseStream ... 500
Close ... 499
Encoding ... 500
Flush ... 499
StreamWriter (Klasse) ... 496
Stretch ... 1107
Stretch (WPF) ... 838, 887
StretchDirection (WPF) ... 887
String ... 75, 670
Chars ... 672
CompareOrdinal ... 672
EndsWith ... 675
erzeugen ... 671
IndexOf ... 675
Insert ... 679, 680
LastIndexOf ... 675
Length ... 672
Methoden ... 683
PadLeft ... 678
PadRight ... 678
Replace ... 680
Split ... 680
StartsWith ... 675
Substring ... 676
ToCharArray ... 682
ToLower ... 679
ToUpper ... 679
Vergleichsmethoden ... 674
string ... 75
String.Format ... 697
StringBuilder ... 671, 684, 685
Append ... 688
Capacity ... 686
Chars ... 686
Eigenschaften ... 687
MaxCapacity ... 686
Methoden ... 687
Replace ... 689
StringCollection ... 334
StringCollection (Klasse) ... 334
StringDictionary ... 334
StringDictionary (Klasse) ... 334
String-Manipulationen ... 679
Stringvergleich ... 672
Stringverkettung ... 98
Stroke ... 1109
Strokes (WPF) ... 891
StrokeThickness ... 1109
struct ... 289, 290
Struktur ... 289
Struktur einer Konsolenanwendung ... 64
Style (WPF) ... 900
Styles ... 948
Substring ... 675
Substring (String) ... 676
Suchen in Zeichenketten ... 675
Sum ... 458
SUO-Datei ... 706
switch ... 113
switch-Statement ... 113

Synchronisation (Thread) 629
System
Enum 295
Read 73
ReadLine 73
Write 68
WriteLine 68
System.DateTime 690
System.Diagnostics 321
System.IO 467
System.IO.Directory 478
System.IO.DirectoryInfo 478
System.IO.File 469
System.IO.Path 482
System.IO.Stream 486
System.Object 665
System.Text 685
System.Threading 617
System.Threading.ThreadPriority 625
SystemColors 946
SystemFonts 946
SystemParameters 946

T

TabControl 869
TabItem 869
TableAdapter 1264
TableMappingCollection 1183
Tables 1193
TabStripPlacement (WPF) 869
Take 461
TakeWhile 462
Target (WPF) 848
TargetSite 310
TargetSite (Exception) 312
TargetUpdated 930
Teilstrings 679
Temporäre Verzeichnisse 483
TextBlock 853
TextBox 849
Textdatei mit FileStream lesen 494
TextDecoration (WPF) 851
TextEffects (WPF) 855
TextMarkerStyle (WPF) 903
TextReader 496
TextReader (Klasse) 496
TextTrimming (WPF) 853
TextWrapping (WPF) 850, 853
TextWriter 496
TextWriter (Klasse) 496
ThenBy 450
ThenByDescending 450
this 175, 212, 408
this (Operator) 408
this-Operator 175
Thread 614, 617, 619
Abort 620
IsAlive 621
Join 624
Priority 625
ResetAbort 622
Start 618
Thread (allgemein) 614
Thread (Klasse) 617
Thread (Synchronisation) 629
Thread starten 618
ThreadAbortException 620
Threadausführung anhalten 620
Threadpool 627
Threadpriorität 615, 625
Threadpriorität (allgemein) 614
ThreadPriority 625
ThreadPriority (Aufzählung) 625
ThreadStart 617
ThreadStart (Delegat) 617
Threadzustände 614
Tick 690
TickFrequency (WPF) 881
TickPlacement (WPF) 881
Ticks (DateTime) 690
TicksPerDay 695
TicksPerHour 695
TileBrush 1120
TimeSpan 694, 696
TicksPerDay 695
TicksPerHour 695
Title (WPF) 811
ToArray-Methode 1308
ToCharArray 682
ToCharArray (String) 682
Today (DateTime) 693
ToDictionary-Methode 1308
ToFileTime 693
ToFileTime (DateTime) 693
ToggleButton 844
ToList-Methode 1308
ToLongDateString 693
ToLongDateString (DateTime) 693
ToLongTimeString 693
ToLongTimeString (DateTime) 693
ToLookup-Methode 1308
ToLower 679
ToLower (String) 679
ToolBar 875
ToolBar (WPF) 876
ToolBarTray 877
ToolBarTray (WPF) 877
Toolbox 55
Tooltip 879
ToolTipService 881

Topmost (WPF) ... 811
ToShortDateString ... 693
ToShortDateString (DateTime) ... 693
ToShortTimeString ... 693
ToShortTimeString (DateTime) ... 693
ToString ... 230, 665, 666, 701
TotalFreeSpace ... 484
TotalSize ... 484
ToTraceString-Methode ... 1304
ToUpper ... 679
ToUpper (String) ... 679
Trace ... 324
Trace (Klasse) ... 324
Translate-Methode ... 1334
TreeView ... 863
TreeViewItem ... 864
Trigger ... 956
Trim ... 677
TrimEnd ... 677
TrimStart ... 677
try ... 302
TryGetObjectByKey ... 1357
TryGetObjectStateEntry ... 1355
Tunneling ... 1059
Tunneling-Event ... 1059
typeof-Operator ... 99, 297
Typfeststellung ... 223
Typinferenz ... 385
Typisierte DataSets ... 1251
Typkonvertierung ... 80
Typkonvertierung (WPF) ... 781
Typkonvertierungsoperator ... 81
Typmetadaten ... 711, 712
Typsuffix ... 77
Typumwandlung ... 220

U

Überwachen (Debuggen) ... 332
Überwachen (Debugmodus) ... 332
Überwachen-Fenster ... 332
UIElement ... 780, 834
UIElement3D ... 1060
UIElementCollection ... 780
uint ... 74
Uint16 ... 74
UInt32 ... 74
Uint64 ... 74
UIPropertyMetadata ... 1047
ulong ... 74
Unboxing ... 298
unchecked ... 84
Undo (WPF) ... 851
UndoLimit (WPF) ... 852
Unicode-Zeichen ... 77
UniFormFill (WPF) ... 887
UniformGrid ... 804
Unindent ... 322
Union ... 456
Unique (Eigenschaft) ... 1200
UniqueConstraint ... 1199
Unmanaged Code ... 429
unsafe ... 429
Unsicherer Programmcode ... 429
UpdateCommand ... 1232, 1239
UpdateException ... 1342
UpdateSourceTrigger ... 924, 928
UpdateSourceTrigger.Explicit ... 929
UpdateStatus (Enumeration) ... 1245
UpdateTarget ... 930
UriSource (WPF) ... 888
UserControl ... 1098
ushort ... 74
using (Anweisung) ... 256
using (Objektzerstörung) ... 256
using-Anweisung ... 256
using-Direktive ... 196
UtcNow ... 692
UtcNow (DateTime) ... 692
UTF-8-Zeichensatz ... 498

V

ValidatesOnExceptions ... 990
ValidateValueCallback ... 1043, 1051
ValidationErrorEventArgs ... 995
ValidationResult ... 990
ValidationRule ... 990
ValidationRules ... 989
ValueConversionAttribute ... 998
value-Parameter (Eigenschaft) ... 148
Values ... 349
Values (IDictionary) ... 349
var ... 385
Variablen ... 66
  initialisieren ... 67
Variablenbezeichner ... 67
Variablendeklaration ... 66
Verbindung
  öffnen ... 1134
  schließen ... 1136
Verbindungspooling ... 1138
Verbindungszeichenfolge ... 1129, 1130
Vererbung ... 205
Vergleichsoperatoren ... 91
Verkettung von Zeichenfolgen ... 98
Verschachtelte Schleifen ... 125
Versiegelte Methode ... 230
Versionierung ... 710, 717
Versionierung (Assemblys) ... 717

Versionsumleitung ... 734
VerticalAlignment (WPF) ... 791, 839
VerticalContentAlignment (WPF) ... 840
VerticalScrollBarVisibility (WPF) ... 850
Verweise
hinzufügen ... 708
Verzweigte Arrays ... 106
ViewBase ... 861
Virtuelle Methode ... 227
Visibility (WPF) ... 840
Visual ... 834
VisualBrush ... 841, 1123
VisualTreeHelper ... 772
Visueller Elementbaum ... 770
void ... 157
Vollqualifizierter Name ... 196
VolumeLabel ... 484
Vordergrundthread ... 627

W

Wahrheitstabellen ... 93
Wait ... 633
Wait (Monitor) ... 633
WaitCallback ... 627
WaitOne ... 629, 639
Wartender Thread ... 614
Warteschlange ... 614
Wertebereich elementarer Datentypen ... 74
Wertetyp ... 143, 144
Wertparameter ... 163
where ... 446
while-Schleife ... 129
Width (WPF) ... 791
Window ... 810
WindowStartupLocation (WPF) ... 811
WindowState (WPF) ... 812
WindowStyle (WPF) ... 812
WindowTitle (WPF) ... 816
WPF-Anwendung ... 762
WPF-Benutzersteuerelementbibliothek ... 762
WPF-Browseranwendung ... 762
WrapPanel ... 795
Write ... 68
Console ... 68
Debug ... 322
Write (Debug) ... 322
Write (Klasse Console) ... 68
Write (Stream) ... 487
WriteAllBytes ... 471
WriteAllLines ... 471
WriteAllText ... 471
WriteByte ... 487
WriteByte (Stream) ... 487, 488
WriteIf ... 322
WriteIf (Debug) ... 322
WriteLine ... 68, 321
WriteLine (Debug) ... 322
WriteLine (Klasse Console) ... 68
WriteLineIf ... 322
WriteLineIf (Debug) ... 322
WriteXmlSchema (Methode) ... 1203

X

x
Array ... 788
Class ... 787
Code ... 787
Key ... 788
Name ... 788
Null ... 788
Static ... 788
Type ... 788
XAML (Beschreibung) ... 773
XmlArray ... 598
XmlArrayAttribute ... 598
XmlArrayItem ... 598
XmlArrayItemAttribute ... 598
XmlAttribute ... 598
XML-Dokumentation ... 736
XmlElement ... 598
XmlElementAttribute ... 598
XmlIgnore ... 598
XmlIgnoreAttribute ... 598
XML-Namespaces (WPF) ... 785
XmlnsDefinitionAttribute ... 787
XmlRoot ... 598
XmlRootAttribute ... 598
XmlSerializer ... 512, 596

Y

Year ... 692
yield ... 356
yield return ... 358

Z

Zeichenfolge → String
Zeichenfolgenverkettung ... 98
Zeigerarithmetik ... 432
Zeigerdeklaration ... 430
Zugriffsmodifizierer ... 140
Klasse ... 140
Zuweisungsoperatoren ... 98