HANSER

Leseprobe

zu

Visual C# 2017
Grundlagen, Profiwissen und Rezepte

Walter Doberenz
Thomas Gewinnus
Jürgen Kotz
Walter Saumweber

ISBN (Buch): 978-3-446-45359-3
ISBN (E-Book): 978-3-446-45370-8

Weitere Informationen und Bestellungen unter
http://www.hanser-fachbuch.de/
sowie im Buchhandel

© Carl Hanser Verlag, München

Inhalt

Vorwort ... **XXI**

TEIL I: Grundlagen ... **1**

1 Einstieg in Visual Studio 2017 ... **3**

1.1 Die Installation von Visual Studio 2017 ... 3
1.1.1 Überblick über die Produktpalette ... 3
1.1.2 Anforderungen an Hard- und Software ... 4
1.2 Unser allererstes C#-Programm ... 5
1.2.1 Vorbereitungen ... 5
1.2.2 Quellcode schreiben ... 7
1.2.3 Programm kompilieren und testen ... 7
1.2.4 Einige Erläuterungen zum Quellcode ... 8
1.2.5 Konsolenanwendungen sind out ... 9
1.3 Die Windows-Philosophie ... 10
1.3.1 Mensch-Rechner-Dialog ... 10
1.3.2 Objekt- und ereignisorientierte Programmierung ... 10
1.3.3 Programmieren mit Visual Studio 2017 ... 12
1.4 Die Entwicklungsumgebung Visual Studio 2017 ... 13
1.4.1 Neues Projekt ... 13
1.4.2 Die wichtigsten Fenster ... 14
1.5 Microsofts .NET-Technologie ... 18
1.5.1 Zur Geschichte von .NET ... 18
1.5.2 .NET-Features und Begriffe ... 20
1.6 Praxisbeispiele ... 28
1.6.1 Unsere erste Windows-Forms-Anwendung ... 28
1.6.2 Umrechnung Euro-Dollar ... 32

2 Grundlagen der Sprache C# ... **41**

2.1 Grundbegriffe ... 41
2.1.1 Anweisungen ... 41

2.1.2 Bezeichner 42
2.1.3 Schlüsselwörter 43
2.1.4 Kommentare 44

2.2 Datentypen, Variablen und Konstanten 45
2.2.1 Fundamentale Typen 45
2.2.2 Wertetypen versus Verweistypen 46
2.2.3 Benennung von Variablen 47
2.2.4 Deklaration von Variablen 47
2.2.5 Typsuffixe 49
2.2.6 Zeichen und Zeichenketten 49
2.2.7 object-Datentyp 52
2.2.8 Konstanten deklarieren 53
2.2.9 Nullable Types 53
2.2.10 Typinferenz 54
2.2.11 Gültigkeitsbereiche und Sichtbarkeit 55

2.3 Konvertieren von Datentypen 56
2.3.1 Implizite und explizite Konvertierung 56
2.3.2 Welcher Datentyp passt zu welchem? 58
2.3.3 Konvertieren von string 58
2.3.4 Die Convert-Klasse 61
2.3.5 Die Parse-Methode 61
2.3.6 Boxing und Unboxing 62

2.4 Operatoren 63
2.4.1 Arithmetische Operatoren 64
2.4.2 Zuweisungsoperatoren 66
2.4.3 Logische Operatoren 67
2.4.4 Rangfolge der Operatoren 69

2.5 Kontrollstrukturen 71
2.5.1 Verzweigungsbefehle 71
2.5.2 Schleifenanweisungen 74

2.6 Benutzerdefinierte Datentypen 77
2.6.1 Enumerationen 77
2.6.2 Strukturen 79

2.7 Nutzerdefinierte Methoden 81
2.7.1 Methoden mit Rückgabewert 82
2.7.2 Methoden ohne Rückgabewert 83
2.7.3 Parameterübergabe mit ref 84
2.7.4 Parameterübergabe mit out 85
2.7.5 Methodenüberladung 86
2.7.6 Optionale Parameter 87
2.7.7 Benannte Parameter 89

2.8 Praxisbeispiele 90
2.8.1 Vom PAP zur Konsolenanwendung 90
2.8.2 Ein Konsolen- in ein Windows-Programm verwandeln 92

2.8.3 Schleifenanweisungen verstehen ... 94
2.8.4 Benutzerdefinierte Methoden überladen ... 96
2.8.5 Anwendungen von Visual Basic nach C# portieren ... 99

3 OOP-Konzepte ... 107

3.1 Kleine Einführung in die OOP ... 107
3.1.1 Historische Entwicklung ... 108
3.1.2 Grundbegriffe der OOP ... 109
3.1.3 Sichtbarkeit von Klassen und ihren Mitgliedern ... 111
3.1.4 Allgemeiner Aufbau einer Klasse ... 112
3.1.5 Das Erzeugen eines Objekts ... 114
3.1.6 Einführungsbeispiel ... 117
3.2 Eigenschaften ... 121
3.2.1 Eigenschaften mit Zugriffsmethoden kapseln ... 121
3.2.2 Berechnete Eigenschaften ... 123
3.2.3 Lese-/Schreibschutz ... 125
3.2.4 Property-Accessoren ... 126
3.2.5 Statische Felder/Eigenschaften ... 126
3.2.6 Einfache Eigenschaften automatisch implementieren ... 129
3.3 Methoden ... 130
3.3.1 Öffentliche und private Methoden ... 130
3.3.2 Überladene Methoden ... 131
3.3.3 Statische Methoden ... 132
3.4 Ereignisse ... 134
3.4.1 Ereignis hinzufügen ... 134
3.4.2 Ereignis verwenden ... 137
3.5 Arbeiten mit Konstruktor und Destruktor ... 140
3.5.1 Konstruktor und Objektinitialisierer ... 141
3.5.2 Destruktor und Garbage Collector ... 144
3.5.3 Mit using den Lebenszyklus des Objekts kapseln ... 146
3.5.4 Verzögerte Initialisierung ... 147
3.6 Vererbung und Polymorphie ... 148
3.6.1 Klassendiagramm ... 148
3.6.2 Method-Overriding ... 149
3.6.3 Klassen implementieren ... 149
3.6.4 Implementieren der Objekte ... 153
3.6.5 Ausblenden von Mitgliedern durch Vererbung ... 154
3.6.6 Allgemeine Hinweise und Regeln zur Vererbung ... 156
3.6.7 Polymorphes Verhalten ... 157
3.6.8 Die Rolle von System.Object ... 160
3.7 Spezielle Klassen ... 161
3.7.1 Abstrakte Klassen ... 161
3.7.2 Versiegelte Klassen ... 163

3.7.3 Partielle Klassen ... 163
3.7.4 Statische Klassen ... 165
3.8 Schnittstellen (Interfaces) ... 165
3.8.1 Definition einer Schnittstelle ... 166
3.8.2 Implementieren einer Schnittstelle ... 166
3.8.3 Abfragen, ob Schnittstelle vorhanden ist ... 167
3.8.4 Mehrere Schnittstellen implementieren ... 168
3.8.5 Schnittstellenprogrammierung ist ein weites Feld 168
3.9 Praxisbeispiele ... 169
3.9.1 Eigenschaften sinnvoll kapseln ... 169
3.9.2 Eine statische Klasse anwenden ... 172
3.9.3 Vom fetten zum schlanken Client ... 174
3.9.4 Schnittstellenvererbung verstehen ... 184
3.9.5 Rechner für komplexe Zahlen ... 189
3.9.6 Sortieren mit IComparable/IComparer ... 198
3.9.7 Einen Objektbaum in generischen Listen abspeichern ... 202
3.9.8 OOP beim Kartenspiel erlernen ... 208
3.9.9 Eine Klasse zur Matrizenrechnung entwickeln ... 213

4 Arrays, Strings, Funktionen ... 219

4.1 Datenfelder (Arrays) ... 219
4.1.1 Array deklarieren ... 220
4.1.2 Array instanziieren ... 220
4.1.3 Array initialisieren ... 221
4.1.4 Zugriff auf Array-Elemente ... 222
4.1.5 Zugriff mittels Schleife ... 223
4.1.6 Mehrdimensionale Arrays ... 224
4.1.7 Zuweisen von Arrays ... 226
4.1.8 Arrays aus Strukturvariablen ... 227
4.1.9 Löschen und Umdimensionieren von Arrays ... 228
4.1.10 Eigenschaften und Methoden von Arrays ... 229
4.1.11 Übergabe von Arrays ... 231
4.2 Verarbeiten von Zeichenketten ... 232
4.2.1 Zuweisen von Strings ... 232
4.2.2 Eigenschaften und Methoden von String-Variablen ... 233
4.2.3 Wichtige Methoden der String-Klasse ... 236
4.2.4 Die StringBuilder-Klasse ... 238
4.3 Reguläre Ausdrücke ... 241
4.3.1 Wozu werden reguläre Ausdrücke verwendet? ... 241
4.3.2 Eine kleine Einführung ... 241
4.3.3 Wichtige Methoden/Eigenschaften der Klasse Regex ... 242
4.3.4 Kompilierte reguläre Ausdrücke ... 244
4.3.5 RegexOptions-Enumeration ... 245
4.3.6 Metazeichen (Escape-Zeichen) ... 246

4.3.7 Zeichenmengen (Character Sets) 247
4.3.8 Quantifizierer 249
4.3.9 Zero-Width Assertions 250
4.3.10 Gruppen 254
4.3.11 Text ersetzen 254
4.3.12 Text splitten 255

4.4 Datums- und Zeitberechnungen 256
4.4.1 Die DateTime-Struktur 256
4.4.2 Wichtige Eigenschaften von DateTime-Variablen 258
4.4.3 Wichtige Methoden von DateTime-Variablen 258
4.4.4 Wichtige Mitglieder der DateTime-Struktur 259
4.4.5 Konvertieren von Datumstrings in DateTime-Werte 260
4.4.6 Die TimeSpan-Struktur 261

4.5 Mathematische Funktionen 263
4.5.1 Überblick 263
4.5.2 Zahlen runden 263
4.5.3 Winkel umrechnen 264
4.5.4 Potenz- und Wurzeloperationen 264
4.5.5 Logarithmus und Exponentialfunktionen 264
4.5.6 Zufallszahlen erzeugen 265

4.6 Zahlen- und Datumsformatierungen 266
4.6.1 Anwenden der ToString-Methode 266
4.6.2 Anwenden der Format-Methode 268
4.6.3 Stringinterpolation 269

4.7 Praxisbeispiele 270
4.7.1 Zeichenketten verarbeiten 270
4.7.2 Zeichenketten mit StringBuilder addieren 273
4.7.3 Reguläre Ausdrücke testen 277
4.7.4 Methodenaufrufe mit Array-Parametern 278

5 Weitere Sprachfeatures 283

5.1 Namespaces (Namensräume) 283
5.1.1 Ein kleiner Überblick 283
5.1.2 Einen eigenen Namespace einrichten 284
5.1.3 Die using-Anweisung 285
5.1.4 Namespace Alias 286

5.2 Operatorenüberladung 287
5.2.1 Syntaxregeln 287
5.2.2 Praktische Anwendung 287

5.3 Collections (Auflistungen) 288
5.3.1 Die Schnittstelle IEnumerable 289
5.3.2 ArrayList 291
5.3.3 Hashtable 293
5.3.4 Indexer 293

5.4 Generics 296
5.4.1 Klassische Vorgehensweise 296
5.4.2 Generics bieten Typsicherheit 298
5.4.3 Generische Methoden 299
5.4.4 Iteratoren 300
5.5 Generische Collections 301
5.5.1 List-Collection statt ArrayList 301
5.5.2 Vorteile generischer Collections 302
5.5.3 Constraints 302
5.6 Das Prinzip der Delegates 303
5.6.1 Delegates sind Methodenzeiger 303
5.6.2 Einen Delegate-Typ deklarieren 303
5.6.3 Ein Delegate-Objekt erzeugen 304
5.6.4 Delegates vereinfacht instanziieren 306
5.6.5 Anonyme Methoden 306
5.6.6 Lambda-Ausdrücke 308
5.6.7 Lambda-Ausdrücke in der Task Parallel Library 310
5.7 Dynamische Programmierung 312
5.7.1 Wozu dynamische Programmierung? 312
5.7.2 Das Prinzip der dynamischen Programmierung 312
5.7.3 Optionale Parameter sind hilfreich 315
5.7.4 Kovarianz und Kontravarianz 316
5.8 Weitere Datentypen 317
5.8.1 BigInteger 317
5.8.2 Complex 319
5.8.3 Tuple<> 320
5.8.4 SortedSet<> 321
5.9 Praxisbeispiele 322
5.9.1 ArrayList versus generische List 322
5.9.2 Generische IEnumerable-Interfaces implementieren 325
5.9.3 Delegates, anonyme Methoden, Lambda Expressions 329
5.9.4 Dynamischer Zugriff auf COM Interop 333

6 Einführung in LINQ 337
6.1 LINQ-Grundlagen 337
6.1.1 Die LINQ-Architektur 337
6.1.2 Anonyme Typen 339
6.1.3 Erweiterungsmethoden 340
6.2 Abfragen mit LINQ to Objects 341
6.2.1 Grundlegendes zur LINQ-Syntax 342
6.2.2 Zwei alternative Schreibweisen von LINQ-Abfragen 343
6.2.3 Übersicht der wichtigsten Abfrageoperatoren 344
6.3 LINQ-Abfragen im Detail 345

6.3.1 Die Projektionsoperatoren Select und SelectMany ... 346
6.3.2 Der Restriktionsoperator Where ... 348
6.3.3 Die Sortierungsoperatoren OrderBy und ThenBy ... 348
6.3.4 Der Gruppierungsoperator GroupBy ... 350
6.3.5 Verknüpfen mit Join ... 352
6.3.6 Aggregat-Operatoren ... 353
6.3.7 Verzögertes Ausführen von LINQ-Abfragen ... 355
6.3.8 Konvertierungsmethoden ... 356
6.3.9 Abfragen mit PLINQ ... 357

6.4 Praxisbeispiele ... 360
6.4.1 Die Syntax von LINQ-Abfragen verstehen ... 360
6.4.2 Aggregat-Abfragen mit LINQ ... 363
6.4.3 LINQ im Schnelldurchgang erlernen ... 365
6.4.4 Strings mit LINQ abfragen und filtern ... 368
6.4.5 Duplikate aus einer Liste oder einem Array entfernen ... 369
6.4.6 Arrays mit LINQ initialisieren ... 372
6.4.7 Arrays per LINQ mit Zufallszahlen füllen ... 374
6.4.8 Einen String mit Wiederholmuster erzeugen ... 376
6.4.9 Mit LINQ Zahlen und Strings sortieren ... 377
6.4.10 Mit LINQ Collections von Objekten sortieren ... 378
6.4.11 Ergebnisse von LINQ-Abfragen in ein Array kopieren ... 381

7 C#-Sprachneuerungen im Überblick ... 383

7.1 C# 4.0 – Visual Studio 2010 ... 383
7.1.1 Datentyp dynamic ... 383
7.1.2 Benannte und optionale Parameter ... 384
7.1.3 Covarianz und Contravarianz ... 386

7.2 C# 5.0 – Visual Studio 2012 ... 386
7.2.1 Async und Await ... 386
7.2.2 CallerInfo ... 386

7.3 Visual Studio 2013 ... 387

7.4 C# 6.0 – Visual Studio 2015 ... 387
7.4.1 String Interpolation ... 387
7.4.2 Schreibgeschützte AutoProperties ... 387
7.4.3 Initialisierer für AutoProperties ... 388
7.4.4 Expression Body Funktionsmember ... 388
7.4.5 using static ... 388
7.4.6 Bedingter Nulloperator ... 389
7.4.7 Ausnahmenfilter ... 389
7.4.8 nameof-Ausdrücke ... 390
7.4.9 await in catch und finally ... 390
7.4.10 Indexinitialisierer ... 390

7.5 C# 7.0 – Visual Studio 2017 ... 391
7.5.1 out-Variablen ... 391

7.5.2 Tupel . . . 391
7.5.3 Mustervergleich . . . 392
7.5.4 Discards . . . 394
7.5.5 Lokale ref-Variablen und Rückgabetypen . . . 394
7.5.6 Lokale Funktionen . . . 394
7.5.7 Mehr Expression-Bodied Member . . . 394
7.5.8 throw-Ausdrücke . . . 395
7.5.9 Verbesserung der numerischen literalen Syntax . . . 395

TEIL II: Technologien . . . 397

8 Zugriff auf das Dateisystem . . . 399

8.1 Grundlagen . . . 399
8.1.1 Klassen für den Zugriff auf das Dateisystem . . . 400
8.1.2 Statische versus Instanzen-Klasse . . . 400
8.2 Übersichten . . . 401
8.2.1 Methoden der Directory-Klasse . . . 402
8.2.2 Methoden eines DirectoryInfo-Objekts . . . 402
8.2.3 Eigenschaften eines DirectoryInfo-Objekts . . . 402
8.2.4 Methoden der File-Klasse . . . 403
8.2.5 Methoden eines FileInfo-Objekts . . . 404
8.2.6 Eigenschaften eines FileInfo-Objekts . . . 404
8.3 Operationen auf Verzeichnisebene . . . 405
8.3.1 Existenz eines Verzeichnisses/einer Datei feststellen . . . 405
8.3.2 Verzeichnisse erzeugen und löschen . . . 405
8.3.3 Verzeichnisse verschieben und umbenennen . . . 406
8.3.4 Aktuelles Verzeichnis bestimmen . . . 406
8.3.5 Unterverzeichnisse ermitteln . . . 407
8.3.6 Alle Laufwerke ermitteln . . . 407
8.3.7 Dateien kopieren und verschieben . . . 408
8.3.8 Dateien umbenennen . . . 409
8.3.9 Dateiattribute feststellen . . . 409
8.3.10 Verzeichnis einer Datei ermitteln . . . 411
8.3.11 Alle im Verzeichnis enthaltenen Dateien ermitteln . . . 411
8.3.12 Dateien und Unterverzeichnisse ermitteln . . . 412
8.4 Zugriffsberechtigungen . . . 413
8.4.1 ACL und ACE . . . 413
8.4.2 SetAccessControl-Methode . . . 413
8.4.3 Zugriffsrechte anzeigen . . . 414
8.5 Weitere wichtige Klassen . . . 415
8.5.1 Die Path-Klasse . . . 415
8.5.2 Die Klasse FileSystemWatcher . . . 416
8.5.3 Die Klasse ZipArchive . . . 417
8.6 Datei- und Verzeichnisdialoge . . . 419

8.6.1 OpenFileDialog und SaveFileDialog ... 419
8.6.2 FolderBrowserDialog ... 421
8.7 Praxisbeispiele ... 422
8.7.1 Infos über Verzeichnisse und Dateien gewinnen ... 422
8.7.2 Eine Verzeichnisstruktur in die TreeView einlesen ... 425
8.7.3 Mit LINQ und RegEx Verzeichnisbäume durchsuchen ... 428

9 Dateien lesen und schreiben ... 433

9.1 Grundprinzip der Datenpersistenz ... 433
9.1.1 Dateien und Streams ... 433
9.1.2 Die wichtigsten Klassen ... 434
9.1.3 Erzeugen eines Streams ... 435
9.2 Dateiparameter ... 435
9.2.1 FileAccess ... 435
9.2.2 FileMode ... 436
9.2.3 FileShare ... 436
9.3 Textdateien ... 437
9.3.1 Eine Textdatei beschreiben bzw. neu anlegen ... 437
9.3.2 Eine Textdatei lesen ... 438
9.4 Binärdateien ... 440
9.4.1 Lese-/Schreibzugriff ... 440
9.4.2 Die Methoden ReadAllBytes und WriteAllBytes ... 441
9.4.3 Erzeugen von BinaryReader/BinaryWriter ... 441
9.5 Sequenzielle Dateien ... 442
9.5.1 Lesen und Schreiben von strukturierten Daten ... 442
9.5.2 Serialisieren von Objekten ... 443
9.6 Dateien verschlüsseln und komprimieren ... 444
9.6.1 Das Methodenpärchen Encrypt/Decrypt ... 444
9.6.2 Verschlüsseln unter Windows Vista/7/8/10 ... 445
9.6.3 Verschlüsseln mit der CryptoStream-Klasse ... 446
9.6.4 Dateien komprimieren ... 447
9.7 Memory Mapped Files ... 448
9.7.1 Grundprinzip ... 448
9.7.2 Erzeugen eines MMF ... 449
9.7.3 Erstellen eines Map View ... 449
9.8 Praxisbeispiele ... 450
9.8.1 Auf eine Textdatei zugreifen ... 450
9.8.2 Einen Objektbaum persistent speichern ... 454
9.8.3 Ein Memory Mapped File (MMF) verwenden ... 461
9.8.4 Hex-Dezimal-Bytes-Konverter ... 463
9.8.5 Eine Datei verschlüsseln ... 467
9.8.6 Eine Datei komprimieren ... 470
9.8.7 PDFs erstellen/exportieren ... 472

9.8.8 Eine CSV-Datei erstellen ... 475
9.8.9 Eine CSV-Datei mit LINQ lesen und auswerten ... 478
9.8.10 Einen korrekten Dateinamen erzeugen ... 480

10 Asynchrone Programmierung ... 481

10.1 Übersicht ... 481
10.1.1 Multitasking versus Multithreading ... 482
10.1.2 Deadlocks ... 483
10.1.3 Racing ... 484
10.2 Programmieren mit Threads ... 485
10.2.1 Einführungsbeispiel ... 486
10.2.2 Wichtige Thread-Methoden ... 487
10.2.3 Wichtige Thread-Eigenschaften ... 489
10.2.4 Einsatz der ThreadPool-Klasse ... 490
10.3 Sperrmechanismen ... 492
10.3.1 Threading ohne lock ... 492
10.3.2 Threading mit lock ... 494
10.3.3 Die Monitor-Klasse ... 496
10.3.4 Mutex ... 500
10.3.5 Methoden für die parallele Ausführung sperren ... 501
10.3.6 Semaphore ... 502
10.4 Interaktion mit der Programmoberfläche ... 503
10.4.1 Die Werkzeuge ... 504
10.4.2 Einzelne Steuerelemente mit Invoke aktualisieren ... 504
10.4.3 Mehrere Steuerelemente aktualisieren ... 506
10.4.4 Ist ein Invoke-Aufruf nötig? ... 506
10.4.5 Und was ist mit WPF? ... 507
10.5 Timer-Threads ... 509
10.6 Die BackgroundWorker-Komponente ... 510
10.7 Asynchrone Programmierentwurfsmuster ... 512
10.7.1 Kurzübersicht ... 513
10.7.2 Polling ... 514
10.7.3 Callback verwenden ... 516
10.7.4 Callback mit Parameterübergabe verwenden ... 516
10.7.5 Callback mit Zugriff auf die Programmoberfläche ... 518
10.8 Asynchroner Aufruf beliebiger Methoden ... 519
10.8.1 Die Beispielklasse ... 519
10.8.2 Asynchroner Aufruf ohne Callback ... 520
10.8.3 Asynchroner Aufruf mit Callback und Anzeigefunktion ... 521
10.8.4 Aufruf mit Rückgabewerten (per Eigenschaft) ... 522
10.8.5 Aufruf mit Rückgabewerten (per EndInvoke) ... 523
10.9 Es geht auch einfacher – async und await ... 524
10.9.1 Der Weg von synchron zu asynchron ... 524

10.9.2 Achtung: Fehlerquellen! 526
10.9.3 Eigene asynchrone Methoden entwickeln 528
10.10 Praxisbeispiele 531
10.10.1 Spieltrieb & Multithreading erleben 531
10.10.2 Prozess- und Thread-Informationen gewinnen 544
10.10.3 Ein externes Programm starten 549

11 Die Task Parallel Library 553

11.1 Überblick 553
11.1.1 Parallel-Programmierung 553
11.1.2 Möglichkeiten der TPL 556
11.1.3 Der CLR-Threadpool 556
11.2 Parallele Verarbeitung mit Parallel.Invoke 557
11.2.1 Aufrufvarianten 558
11.2.2 Einschränkungen 559
11.3 Verwendung von Parallel.For 559
11.3.1 Abbrechen der Verarbeitung 561
11.3.2 Auswerten des Verarbeitungsstatus 562
11.3.3 Und was ist mit anderen Iterator-Schrittweiten? 563
11.4 Collections mit Parallel.ForEach verarbeiten 564
11.5 Die Task-Klasse 565
11.5.1 Einen Task erzeugen 565
11.5.2 Den Task starten 566
11.5.3 Datenübergabe an den Task 568
11.5.4 Wie warte ich auf das Ende des Task? 569
11.5.5 Tasks mit Rückgabewerten 571
11.5.6 Die Verarbeitung abbrechen 574
11.5.7 Fehlerbehandlung 578
11.5.8 Weitere Eigenschaften 579
11.6 Zugriff auf das User-Interface 580
11.6.1 Task-Ende und Zugriff auf die Oberfläche 580
11.6.2 Zugriff auf das UI aus dem Task heraus 582
11.7 Weitere Datenstrukturen im Überblick 584
11.7.1 Threadsichere Collections 584
11.7.2 Primitive für die Threadsynchronisation 585
11.8 Parallel LINQ (PLINQ) 585
11.9 Praxisbeispiel: Spieltrieb – Version 2 585
11.9.1 Aufgabenstellung 586
11.9.2 Global-Klasse 586
11.9.3 Controller-Klasse 587
11.9.4 LKW-Klasse 589
11.9.5 Schiff-Klasse 590
11.9.6 Oberfläche 593

12 Fehlersuche und Behandlung ... 595
12.1 Der Debugger ... 595
12.1.1 Allgemeine Beschreibung ... 595
12.1.2 Die wichtigsten Fenster ... 596
12.1.3 Debugging-Optionen ... 599
12.1.4 Praktisches Debugging am Beispiel ... 602
12.2 Arbeiten mit Debug und Trace ... 606
12.2.1 Wichtige Methoden von Debug und Trace ... 606
12.2.2 Besonderheiten der Trace-Klasse ... 610
12.2.3 TraceListener-Objekte ... 611
12.3 Caller Information ... 613
12.3.1 Attribute ... 613
12.3.2 Anwendung ... 614
12.4 Fehlerbehandlung ... 615
12.4.1 Anweisungen zur Fehlerbehandlung ... 615
12.4.2 try-catch ... 615
12.4.3 try-finally ... 620
12.4.4 Das Standardverhalten bei Ausnahmen festlegen ... 623
12.4.5 Die Exception-Klasse ... 624
12.4.6 Fehler/Ausnahmen auslösen ... 625
12.4.7 Eigene Fehlerklassen ... 625
12.4.8 Exceptionhandling zur Entwurfszeit ... 627
12.4.9 Code Contracts ... 628

13 XML in Theorie und Praxis ... 629
13.1 XML – etwas Theorie ... 629
13.1.1 Übersicht ... 629
13.1.2 Der XML-Grundaufbau ... 632
13.1.3 Wohlgeformte Dokumente ... 633
13.1.4 Processing Instructions (PI) ... 635
13.1.5 Elemente und Attribute ... 636
13.1.6 Verwendbare Zeichensätze ... 637
13.2 XSD-Schemas ... 640
13.2.1 XSD-Schemas und ADO.NET ... 640
13.2.2 XML-Schemas in Visual Studio analysieren ... 642
13.2.3 XML-Datei mit XSD-Schema erzeugen ... 646
13.2.4 XSD-Schema aus einer XML-Datei erzeugen ... 647
13.3 Verwendung des DOM unter .NET ... 647
13.3.1 Übersicht ... 647
13.3.2 DOM-Integration in C# ... 649
13.3.3 Laden von Dokumenten ... 649
13.3.4 Erzeugen von XML-Dokumenten ... 650
13.3.5 Auslesen von XML-Dateien ... 652

13.3.6 Direktzugriff auf einzelne Elemente 653
13.3.7 Einfügen von Informationen 654
13.3.8 Suchen in den Baumzweigen 657
13.4 XML-Verarbeitung mit LINQ to XML 660
13.4.1 Die LINQ to XML-API 660
13.4.2 Neue XML-Dokumente erzeugen 662
13.4.3 Laden und Sichern von XML-Dokumenten 664
13.4.4 Navigieren in XML-Daten 665
13.4.5 Auswählen und Filtern 667
13.4.6 Manipulieren der XML-Daten 668
13.4.7 XML-Dokumente transformieren 669
13.5 Weitere Möglichkeiten der XML-Verarbeitung 672
13.5.1 XML-Daten aus Objektstrukturen erzeugen 672
13.5.2 Schnelles Suchen in XML-Daten mit XPathNavigator 676
13.5.3 Schnelles Auslesen von XML-Daten mit XmlReader 678
13.5.4 Erzeugen von XML-Daten mit XmlWriter 680
13.5.5 XML transformieren mit XSLT 682
13.6 JSON – JavaScriptObjectNotation 684
13.6.1 Grundlagen 684
13.6.2 De-/Serialisierung mit JSON 684
13.7 Praxisbeispiele 687
13.7.1 Mit dem DOM in XML-Dokumenten navigieren 687
13.7.2 XML-Daten in eine TreeView einlesen 691
13.7.3 In Dokumenten mit dem XPathNavigator navigieren 695

14 Einführung in ADO.NET und Entity Framework 701

14.1 Eine kleine Übersicht 701
14.1.1 Die ADO.NET-Klassenhierarchie 701
14.1.2 Die Klassen der Datenprovider 703
14.1.3 Das Zusammenspiel der ADO.NET-Klassen 705
14.2 Das Connection-Objekt 706
14.2.1 Allgemeiner Aufbau 706
14.2.2 SqlConnection 706
14.2.3 Schließen einer Verbindung 707
14.2.4 Eigenschaften des Connection-Objekts 708
14.2.5 Methoden des Connection-Objekts 710
14.2.6 Der ConnectionStringBuilder 711
14.3 Das Command-Objekt 711
14.3.1 Erzeugen und Anwenden eines Command-Objekts 712
14.3.2 Erzeugen mittels CreateCommand-Methode 713
14.3.3 Eigenschaften des Command-Objekts 713
14.3.4 Methoden des Command-Objekts 715
14.3.5 Freigabe von Connection- und Command-Objekten 717

14.4 Parameter-Objekte 718
14.4.1 Erzeugen und Anwenden eines Parameter-Objekts 718
14.4.2 Eigenschaften des Parameter-Objekts 719
14.5 Das CommandBuilder-Objekt 720
14.5.1 Erzeugen 720
14.5.2 Anwenden 720
14.6 Das DataReader-Objekt 721
14.6.1 DataReader erzeugen 721
14.6.2 Daten lesen 722
14.6.3 Eigenschaften des DataReaders 723
14.6.4 Methoden des DataReaders 723
14.7 Das DataAdapter-Objekt 724
14.7.1 DataAdapter erzeugen 724
14.7.2 Command-Eigenschaften 725
14.7.3 Fill-Methode 726
14.7.4 Update-Methode 727
14.8 Entity Framework 728
14.8.1 Überblick 728
14.8.2 DatabaseFirst 730
14.8.3 CodeFirst 738
14.9 Praxisbeispiele 743
14.9.1 Wichtige ADO.NET-Objekte im Einsatz 743
14.9.2 Eine Aktionsabfrage ausführen 745
14.9.3 Eine StoredProcedure aufrufen 747
14.9.4 Die Datenbank aktualisieren 750

15 Das DataSet 755

15.1 Grundlegende Features des DataSets 755
15.1.1 Die Objekthierarchie 756
15.1.2 Die wichtigsten Klassen 756
15.1.3 Erzeugen eines DataSets 757
15.2 Das DataTable-Objekt 759
15.2.1 DataTable erzeugen 759
15.2.2 Spalten hinzufügen 759
15.2.3 Zeilen zur DataTable hinzufügen 760
15.2.4 Auf den Inhalt einer DataTable zugreifen 761
15.3 Die DataView 763
15.3.1 Erzeugen einer DataView 764
15.3.2 Sortieren und Filtern von Datensätzen 764
15.3.3 Suchen von Datensätzen 765
15.4 Typisierte DataSets 765
15.4.1 Ein typisiertes DataSet erzeugen 766
15.4.2 Das Konzept der Datenquellen 767
15.4.3 Typisierte DataSets und TableAdapter 768

15.5 Die Qual der Wahl ... 769
15.5.1 DataReader – der schnelle Lesezugriff ... 770
15.5.2 DataSet – die Datenbank im Hauptspeicher ... 770
15.5.3 Objektrelationales Mapping – die Zukunft? ... 771
15.6 Praxisbeispiele ... 772
15.6.1 In der DataView sortieren und filtern ... 772
15.6.2 Suche nach Datensätzen ... 774
15.6.3 Ein DataSet in einen XML-String serialisieren ... 776
15.6.4 Untypisiertes DataSet in ein typisiertes konvertieren ... 780

16 Verteilen von Anwendungen ... 787
16.1 ClickOnce-Deployment ... 788
16.1.1 Übersicht/Einschränkungen ... 788
16.1.2 Die Vorgehensweise ... 789
16.1.3 Ort der Veröffentlichung ... 789
16.1.4 Anwendungsdateien ... 790
16.1.5 Erforderliche Komponenten ... 791
16.1.6 Aktualisierungen ... 792
16.1.7 Veröffentlichungsoptionen ... 793
16.1.8 Veröffentlichen ... 794
16.1.9 Verzeichnisstruktur ... 794
16.1.10 Der Webpublishing-Assistent ... 796
16.1.11 Neue Versionen erstellen ... 797
16.2 Setup-Projekte ... 797
16.2.1 Installation ... 797
16.2.2 Ein neues Setup-Projekt ... 799
16.2.3 Dateisystem-Editor ... 804
16.2.4 Registrierungs-Editor ... 805
16.2.5 Dateityp-Editor ... 806
16.2.6 Benutzeroberflächen-Editor ... 806
16.2.7 Editor für benutzerdefinierte Aktionen ... 807
16.2.8 Editor für Startbedingungen ... 808

17 Weitere Techniken ... 809
17.1 Zugriff auf die Zwischenablage ... 809
17.1.1 Das Clipboard-Objekt ... 809
17.1.2 Zwischenablage-Funktionen für Textboxen ... 811
17.2 Arbeiten mit der Registry ... 812
17.2.1 Allgemeines ... 812
17.2.2 Registry-Unterstützung in .NET ... 814
17.3 .NET-Reflection ... 815
17.3.1 Übersicht ... 815
17.3.2 Assembly laden ... 816
17.3.3 Mittels GetType und Type Informationen sammeln ... 816
17.3.4 Dynamisches Laden von Assemblies ... 819

17.4 Praxisbeispiele ... 822
17.4.1 Zugriff auf die Registry ... 822
17.4.2 Dateiverknüpfungen erzeugen ... 824
17.4.3 Betrachter für Manifestressourcen ... 826
17.4.4 Die Zwischenablage überwachen und anzeigen ... 829
17.4.5 Die WIA-Library kennenlernen ... 832
17.4.6 Auf eine Webcam zugreifen ... 843
17.4.7 Auf den Scanner zugreifen ... 845
17.4.8 OpenOffice.org Writer per OLE steuern ... 851
17.4.9 Nutzer und Gruppen des aktuellen Systems ermitteln ... 858
17.4.10 Testen, ob Nutzer in einer Gruppe enthalten ist ... 860
17.4.11 Testen, ob der Nutzer ein Administrator ist ... 862
17.4.12 Die IP-Adressen des Computers bestimmen ... 863
17.4.13 Die IP-Adresse über den Hostnamen bestimmen ... 864
17.4.14 Diverse Systeminformationen ermitteln ... 865
17.4.15 Alles über den Bildschirm erfahren ... 873
17.4.16 Sound per MCI aufnehmen ... 875
17.4.17 Mikrofonpegel anzeigen ... 878
17.4.18 Pegeldiagramm aufzeichnen ... 880
17.4.19 Sound-und Videodateien per MCI abspielen ... 883

18 Konsolenanwendungen ... 893

18.1 Grundaufbau/Konzepte ... 893
18.1.1 Unser Hauptprogramm – Program.cs ... 894
18.1.2 Rückgabe eines Fehlerstatus ... 895
18.1.3 Parameterübergabe ... 896
18.1.4 Zugriff auf die Umgebungsvariablen ... 898
18.2 Die Kommandozentrale: System.Console ... 899
18.2.1 Eigenschaften ... 899
18.2.2 Methoden/Ereignisse ... 900
18.2.3 Textausgaben ... 900
18.2.4 Farbangaben ... 901
18.2.5 Tastaturabfragen ... 903
18.2.6 Arbeiten mit Streamdaten ... 904
18.3 Praxisbeispiel ... 906
18.3.1 Farbige Konsolenanwendung ... 906
18.3.2 Weitere Hinweise und Beispiele ... 908

Anhang A: Glossar ... 909

Anhang B: Wichtige Dateiextensions ... 913

Index ... 915

Vorwort

C# ist die momentan von Microsoft propagierte Sprache, sie bietet die Möglichkeiten und Flexibilität von C++ und erlaubt trotzdem eine schnelle und unkomplizierte Programmierpraxis wie einst bei Visual Basic. C# ist (fast) genauso mächtig wie C++, wurde aber komplett neu auf objektorientierter Basis geschrieben.

Damit ist C# das ideale Werkzeug zum Programmieren beliebiger Komponenten für das Microsoft .NET Framework, beginnend bei Windows Forms- über WPF-, ASP.NET-, und mobilen-Anwendungen (mittlerweile auch für Android und iOS) bis hin zu systemnahen Applikationen.

Das vorliegende Buch ist ein Angebot für künftige wie auch für fortgeschrittene C#-Programmierer. Seine Philosophie knüpft an die vielen anderen Titel an, die in dieser Reihe in den vergangenen zwanzig Jahren zu verschiedenen Programmiersprachen erschienen sind:

- Programmieren lernt man nicht durch lineares Durcharbeiten eines Lehrbuchs, sondern nur durch unermüdliches Ausprobieren von Beispielen, verbunden mit ständigem Nachschlagen in der Referenz.
- Der Umfang einer modernen Sprache wie C# in Verbindung mit Visual Studio ist so gewaltig, dass ein seriöses Programmierbuch das Prinzip der Vollständigkeit aufgeben muss und nach dem Prinzip „so viel wie nötig" sich lediglich eine „Initialisierungsfunktion" auf die Fahnen schreiben kann.

Gegenüber anderen Büchern zur gleichen oder ähnlichen Thematik nimmt unser Titel für sich in Anspruch, gleichzeitig Lehr- und Übungsbuch zu sein.

Zum Buchinhalt

Wie Sie bereits dem Buchtitel entnehmen können, wagt das vorliegende Werk den Spagat zwischen einem Grundlagen- und einem Profibuch. Sinn eines solchen Buches kann es nicht sein, eine umfassende Schritt-für-Schritt-Einführung in Visual C# 2017 zu liefern oder all die Informationen noch einmal zur Verfügung zu stellen, die Sie in der Produktdokumentation (MSDN) ohnehin schon finden und von denen Sie in der Regel nur ein Mausklick oder die F1-Taste trennt.

- Für den *Einsteiger* wollen wir den einzig vernünftigen und gangbaren Weg beschreiten, nämlich nach dem Prinzip „so viel wie nötig" eine schmale Schneise durch den Urwald der .NET-Programmierung mit Visual C# 2017 schlagen, bis er eine Lichtung erreicht hat, die ihm erste Erfolgserlebnisse vermittelt.

- Für den *Profi* wollen wir in diesem Buch eine Vielzahl von Informationen und Know-how bereitstellen, wonach er bisher in den mitgelieferten Dokumentationen, im Internet bzw. in anderen Büchern vergeblich gesucht hat.

Die Kapitel des Buchs und die Bonuskapitel haben wir in vier Themenkomplexen gruppiert:

1. Grundlagen der Programmierung mit C# (Buch)
2. Technologien (Buch)
3. Windows Forms-Anwendungen (online)
4. WPF-Anwendungen (online)

Die Kapitel innerhalb eines Teils bilden einerseits eine logische Aufeinanderfolge, können andererseits aber auch quergelesen werden. Im Praxisteil eines jeden Kapitels werden anhand realer Problemstellungen die behandelten Programmiertechniken im Zusammenhang demonstriert.

Zu den Codebeispielen

Um den Einstieg in C# so einfach wie möglich zu gestalten, haben wir uns entschlossen die Beispiele mit WindowsForms zu erstellen.

Alle Beispieldaten dieses Buchs und die Bonuskapitel können Sie sich unter der folgenden Adresse herunterladen:

downloads.hanser.de

Beim Nachvollziehen der Buchbeispiele beachten Sie bitte Folgendes:

- Kopieren Sie die Buchbeispiele auf die Festplatte. Wenn Sie auf die Projektmappendatei (**.sln*) klicken, wird Visual Studio in der Regel automatisch geöffnet und das jeweilige Beispiel wird in die Entwicklungsumgebung geladen, wo Sie es z. B. mittels F5-Taste kompilieren und starten können.
- Einige wenige Datenbankprojekte verwenden absolute Pfadnamen, die Sie vor dem Kompilieren des Beispiels erst noch anpassen müssen.
- Für einige Beispiele sind ein installierter Microsoft SQL Server Express LocalDB erforderlich.
- Beachten Sie die zu einigen Beispielen beigefügten *Liesmich.txt*-Dateien, die Sie auf besondere Probleme hinweisen.

Nobody is perfect

Sie werden in diesem Buch nicht alles finden, was Visual C# 2017 bzw. das .NET Framework 4.7 zu bieten haben. Manches ist sicher in einem anderen Spezialtitel noch besser oder ausführlicher beschrieben. Aber Sie halten mit unserem Buch einen überschaubaren Breitband-Mix in den Händen, der sowohl vertikal vom Einsteiger bis zum Profi als auch horizontal von den einfachen Sprachelementen bis hin zu komplexen Anwendungen jedem etwas bietet, ohne dabei den Blick auf das Wesentliche im .NET-Dschungel zu verlieren.

Wenn Sie Vorschläge oder Fragen zum Buch haben, können Sie uns gern unter

juergen.kotz@primetime-software.de

kontaktieren.

Wir hoffen, dass wir Ihnen mit diesem Buch einen nützlichen Begleiter bei der .NET-Programmierung zur Seite gestellt haben, der es verdient, seinen Platz nicht im Regal, sondern griffbereit neben dem Computer einzunehmen.

Jürgen Kotz, Walter Saumweber, Walter Doberenz und Thomas Gewinnus

München, im Dezember 2017

14 Einführung in ADO.NET und Entity Framework

Nach wie vor ist ADO.NET die Basis-Datenzugriffstechnologie des .NET-Frameworks. Auch wenn in vielen neuen Projekten mittlerweile das *Entity Framework* zum Einsatz kommt, darf nicht vergessen werden, dass die Basis des Entity Frameworks auch weiterhin ADO.NET ist. Der Inhalt des vorliegenden Kapitels beschränkt sich auf eine knappe Einführung in die ADO.NET-Technologie sowie des Entity Frameworks, wobei der Schwerpunkt auf dem ersten Teil des Objektmodells, den Datenprovidern, liegt.

Erst das Kapitel 14 widmet sich ausführlich einem der Kernobjekte von ADO.NET, dem *DataSet*. Auf die speziellen Probleme der Datenbindung, d. h., auf die Interaktion der Benutzerschnittstelle mit den ADO.NET-Objekten, gehen wir in den speziellen Kapiteln 26 und 33 in Bezug auf den jeweiligen Anwendungstyp näher ein.

HINWEIS: Ausgeklammert werden musste in diesem Buch die weiterführende Technologie LINQ to Entities, einen kleinen Vorgeschmack bietet Ihnen die Kurzeinführung zu Entity Framework.

14.1 Eine kleine Übersicht

Die umfangreichen Klassenbibliotheken von ADO.NET verlangen vom Einsteiger eine ziemlich steile „Lernkurve“. Er tut gut daran, sich zunächst einen Gesamtüberblick zu verschaffen.

14.1.1 Die ADO.NET-Klassenhierarchie

ADO.NET setzt sich aus einer ziemlich komplexen Hierarchie vieler Klassen zusammen. Die daraus erzeugten Objekte lassen sich zunächst in zwei Gruppen aufteilen:

- Datenprovider
- Datenkonsument

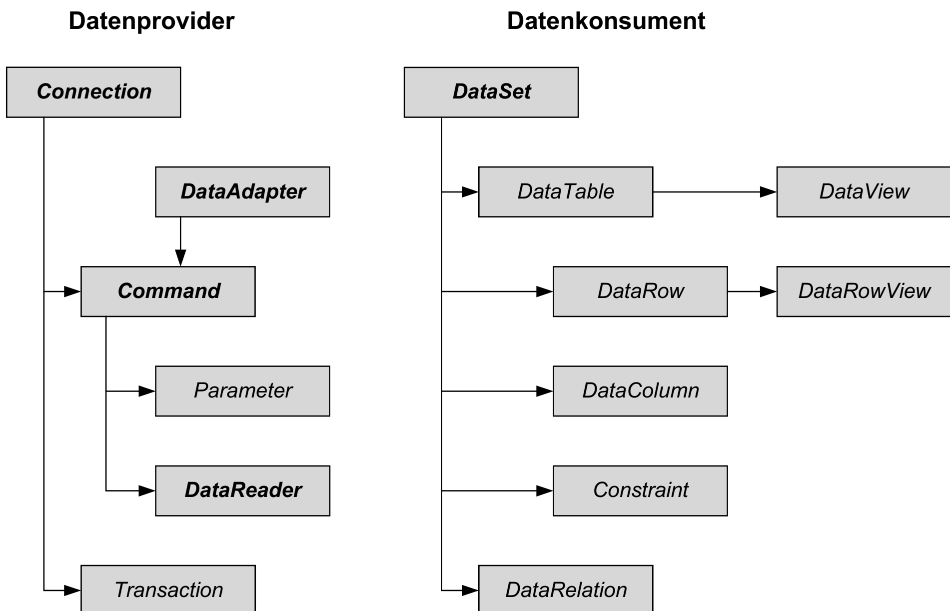
Während der *Datenprovider* die Daten zur Verfügung stellt, ist der *Datenkonsument* der Teil einer Applikation, welcher die Dienste eines Datenproviders nutzt, um auf beliebige Daten zuzugreifen, sie zu lesen, zu speichern und zu ändern.

Die Objekte *Connection*, *Command*, *DataReader* und *DataAdapter* sind die Hauptelemente des .NET-Datenprovider-Modells.

Man bezeichnet die Datenprovider auch als *Verbundene Objekte*, da sie immer in Beziehung zu einer bestimmten Datenquelle stehen. Die Datenkonsumenten hingegen sind *Unverbundene Objekte*, weil sie – ganz im Sinne der ADO.NET-Philosophie – unabhängig von einer Datenquelle ihr völlig autarkes Dasein führen.

Der allen übergeordnete Datenkonsument ist das *DataSet*. Es ist gewissermaßen das Kernobjekt von ADO.NET und vergleichbar mit den vom alten ADO her bekannten *Recordset*-Objekten. Allerdings ist es weitaus komplexer, da es z. B. mehrere *DataTable*-Objekte und die Beziehungen (Relationen) zwischen ihnen kapseln kann. Ein *DataSet* kann (unter Verwendung eines *DataAdapters*) direkt von der Datenquelle geladen werden. Es kann aber auch – ähnlich einem Array – völlig unabhängig von einer Datenbank mit Werten gefüllt werden. Der *DataAdapter* ist quasi das Verbindungsglied zwischen Datenprovider (Datenbank) und Datenkonsument (unverbundene Objekte).

HINWEIS: Um ein erstes praktisches Feeling für die ADO.Net-Klassen zu entwickeln, sollte der Einsteiger bereits jetzt ein einfaches Beispiel ausprobieren, z. B. das Praxisbeispiel in Abschnitt 14.9.1, Wichtige ADO.NET-Objekte im Einsatz.



14.1.2 Die Klassen der Datenprovider

Im Einklang mit dem ADO.NET-Objektmodell sind Datenprovider stets in mehrfacher Ausfertigung vorhanden. Die Präfixe charakterisieren die Zugehörigkeit zu einem bestimmten *.NET-Datenprovider,* z. B.:

- *OleDb...*
  Diese Klassen (z. B. *OleDbConnection*) dienen dem OLE-Db-Zugriff auf unterschiedlichste Datenbanktypen, für die ein Treiber installiert ist.
- *Sql...*
  Diese Klassen (z. B. *SqlConnection*) dienen dem schnelleren Direktzugriff auf den Microsoft SQL Server.

Der *Datenprovider* im .NET Framework kapselt die Datenbank und ermöglicht den Zugriff über eine einheitliche Schnittstelle. Er fungiert quasi als Brücke zwischen einer Anwendung und einer Datenbank und wird zum Abrufen von Daten aus einer Datenbank und zum Abgleichen von Änderungen an diesen Daten mit der Datenbank verwendet.

Die Datenquelle selbst kann eine beliebige Struktur haben und sich an einem beliebigen Ort befinden, z. B. eine lokale Access-Datenbank, ein SQL-Server oder aber auch eine Oracle-Datenbank.

.NET-Datenprovider

In der folgenden Tabelle sind wichtige Klassen der *OleDb-* und *SqlServer*-Provider paarweise aufgelistet:

System.Data.OleDb	System.Data.SqlClient	Bedeutung
OleDbConnection	*SqlConnection*	Stellt die Verbindung zur Datenquelle her
OleDbCommand	*SqlCommand*	Führt eine SQL-Abfrage aus
OleDbDataReader	*SqlDataReader*	Ermöglicht einen sequenziellen Nur-Lese-Zugriff auf die Datenquelle
OleDbDataAdapter	*SqlDataAdapter*	Ermöglicht das Füllen eines *DataSet*s mit den Ergebnissen einer SQL-Abfrage
OleDbCommandBuilder	*SqlCommandBuilder*	Erstellt automatisch *Command*-Objekte für die Übernahme der in einem *DataSet* vorgenommenen Änderungen in die Datenbank
OleDbTransaction	*SqlTransaction*	Organisiert die Anwendung von Transaktionen

Weitere Datenprovider

Die Liste der .NET-Datenprovider ist keinesfalls nur auf die nach einer Standardinstallation vorhandenen Provider beschränkt. Neben *OleDb* und *SqlClient* sind in ADO.NET u. a. auch die folgenden Provider enthalten:

- *System.Data.Odbc*
- *System.Data.SqlServerCe*
- *System.Data.OracleClient*

Anzeige der installierten Datenprovider

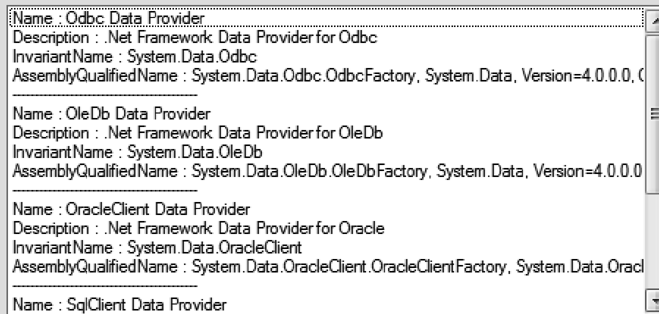
Einen Überblick über alle auf Ihrem System installierten ADO.NET-Datenprovider können Sie mit der Methode *GetFactoryClasses* der *DbProviderFactories*-Auflistung aus dem *System.Data.Common*-Namespace gewinnen.

Beispiel 14.1: Alle verfügbaren Datenprovider in einer *ListBox* anzeigen

C#

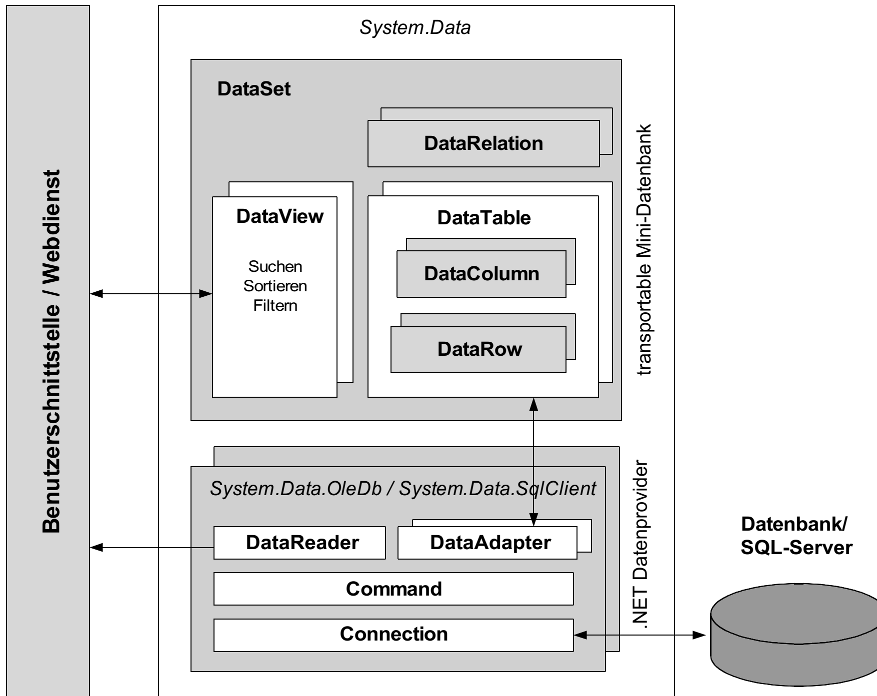
```
DataTable providers = System.Data.Common.DbProviderFactories.GetFactoryClasses();
foreach (DataRow provider in providers.Rows)
{
   foreach (DataColumn col in providers.Columns)
          listBox1.Items.Add(col.ColumnName + " : " + provider[col]);
   listBox1.Items.Add("------------------------------------------");
}
```

Ergebnis



14.1.3 Das Zusammenspiel der ADO.NET-Klassen

In der Abbildung wird versucht, den grundlegenden Zusammenhang zwischen den ADO. NET-Klassen in vereinfachter Form zu verdeutlichen:



Die in der obigen Abbildung angegebenen Namespaces (Namensräume) für die ADO.NET-Klassen sind:

- *System.Data*
- *System.Data.OleDb*
- *System.Data.SQLClient*

Das *DataSet* ist vollständig von der Datenbank entkoppelt, denn dazwischen hat sich ein *.NET-Datenprovider* geschoben, der im Bedarfsfall den Datentransport (über die OLE-DB- bzw. die direkte SQL-Server-Schnittstelle) übernimmt.

HINWEIS: Eine einfache und schnellere Möglichkeit für den Zugriff auf die Datenquelle ist der *DataReader*, da mit ihm auf direktem Weg – also ohne *DataAdapter* und *DataSet* – Daten in die Benutzeroberfläche eingelesen werden können. Das Entity Framework benutzt ausschließlich den DataReader, um die Objektstrukturen zu füllen.

14.2 Das Connection-Objekt

Um überhaupt auf eine Datenbank zugreifen zu können, muss als Erstes eine Verbindung zu ihr hergestellt werden. Dazu führt in der Regel kein Weg am *Connection*-Objekt vorbei.

14.2.1 Allgemeiner Aufbau

Der am häufigsten zum Erzeugen und Initialisieren eines *Connection*-Objekts benutzte Konstruktor nimmt einen *ConnectionString* als Parameter entgegen:

Syntax:

```
Connection conn = new Connection(string ConnectionString);
```

Der *ConnectionString* – die gleichzeitig auch wichtigste Eigenschaft des *Connection*-Objekts – kapselt alle erforderlichen Verbindungsparameter.

Durch Aufruf der (parameterlosen) *Open*-Methode erhält das *Connection*-Objekt eine offene Verbindung aus dem Verbindungspool, falls diese verfügbar ist. Andernfalls wird eine neue Verbindung mit der Datenquelle erstellt.

Nach einer Standardinstallation von Visual Studio stehen – je nach Auswahl des .NET-Providers – verschiedene *Connection*-Objekte zur Verfügung, z. B.:

- *OleDbConnection*-Objekt
  ... gewährleistet den Zugriff auf eine Vielzahl von Datenquellen, angefangen mit einfachen Textdateien über Tabellen bis hin zu kompletten Datenbanken.
- *SqlConnection*-Objekt
  ... ist speziell für die Verwendung mit dem SQL Server optimiert, indem die OLE-DB-Schicht umgangen wird.

14.2.2 SqlConnection

Parameter für Sql-Zugriff

Die Tabelle zeigt die wichtigsten Angaben für den Sql-Server-Zugriff:

Parameter	Bedeutung
Data Source	Name der Datenquelle (Name der SQL-Server-Instanz, z. B. *localhost\SqlExpress*)
Initial Catalog	Name der Datenbank
Integrated Security	Kennzeichen, ob über die Windows-Authentifizierung auf die Datenbank zugegriffen werden soll (alternativ *UserName* und *Password*)

Sql-Provider für Microsoft-SQL-Server-Datenbank

Beispiel 14.2: Öffnen einer SQL-Verbindung zur Northwind-Datenbank mittels Windows-Authentifizierung.

C#

```
using System.Data.SqlClient;
...
SqlConnection conn = new SqlConnection(
       "Data Source=.\\SqlExpress;Initial Catalog=Northwind;
                        Integrated Security=true");
conn.Open();
```

Im obigen Beispiel wurde der *ConnectionString* dem *new*-Konstruktor übergeben. Man kann ihn aber auch separat zuweisen, wie das folgende Beispiel zeigt.

Beispiel 14.3: Eine zum Vorgängerbeispiel äquivalente Variante

C#

```
using System.Data.SqlClient;
...
SqlConnection conn = new SqlConnection();
conn.ConnectionString = "Data Source=.\\SqlExpress;
        Initial Catalog=Northwind;Integrated Security=true");
conn.Open();
```

Wie Sie erkennen, besteht ein *ConnectionString* aus einer Zeichenfolge mit Attribut/Wert-Paaren für Informationen, die zum Anmelden an einen Datenbankserver und Zugriff auf eine bestimmte Datenbank erforderlich sind.

HINWEIS: Die Reihenfolge der Parameter im *ConnectionString* ist ohne Bedeutung!

14.2.3 Schließen einer Verbindung

Nachdem die Daten übertragen worden sind, sollte die Verbindung mithilfe der *Close*-Methode wieder geschlossen werden. Ansonsten bleibt die Connection weiter geöffnet, auch nachdem die Connection-Instanz selbst terminiert wurde!

Um die Netzbelastung gering zu halten, sollte man – ganz im Sinne der ADO.NET-Philosophie – das Öffnen und Schließen einer Verbindung möglichst innerhalb einer einzigen Befehlskette durchführen.

Beispiel 14.4: Es wird kurzzeitig eine Verbindung zur SqlServer-Datenbank *Northwind* geöffnet, um die Kundentabelle in ein DataSet zu übertragen. Danach wird die Verbindung sofort wieder geschlossen.

C#

```
using System.Data.SqlClientOleDb;
...

SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                     Initial Catalog=Northwind;Integrated Security=true");
```

Verbindung öffnen:

```
       conn.Open();
```

Daten übertragen:

```
       DataSet ds = new DataSet();
       SqlDataAdapter da = new SqlDataAdapter(
                             "SELECT * FROM Customers", conn);
       da.Fill(ds, "Kunden");
```

Verbindung schließen:

```
       conn.Close();
```

HINWEIS: Obiger Code ist in vielen Fällen nicht ganz unproblematisch, z. B. wenn die Datenbank nicht vorhanden ist. Die saubere Lösung wird im Abschnitt 14.3.5diskutiert. Vielleicht sollten Sie hier auch einen using-Block verwenden.

14.2.4 Eigenschaften des Connection-Objekts

Da wir im Verlauf dieses Abschnitts bereits viele Eigenschaften des *Connection*-Objekts en passant besprochen haben, soll diese knappe Zusammenfassung den Überblick erleichtern und gleichzeitig einige zusätzliche Informationen liefern.

ConnectionString-Eigenschaft

Diese zweifelsohne wichtigste Eigenschaft des *Connection*-Objekts kapselt sämtliche Verbindungsinformationen zur Datenbank. Außerdem ist es die einzige Eigenschaft, die nicht schreibgeschützt ist (wenn keine Verbindung zur Datenquelle besteht).

Database- und DataSource-Eigenschaft

Was ist der Unterschied zwischen beiden Eigenschaften? Die *DataSource*-Eigenschaft des *Connection*-Objekts entspricht dem *Data Source*-Attribut innerhalb des *ConnectionStrings* und enthält den Speicherort der Datenquelle.

Für eine serverbasierte Datenquelle (Microsoft SQL Server, Oracle) bedeutet der Speicherort den Namen des Computers, auf dem der Server installiert ist. Bei dateibasierten Datenbanken, wie z. B. Access, verweist diese Eigenschaft auf den Datenbankpfad (z. B. *c:\Beispiele\Nordwind.mdb*).

Die *Database*-Eigenschaft ist hingegen für Datenquellen, wie z. B. den SQL Server, gedacht, die mehrere Datenbanken unterstützen, und entspricht dem Attribut *Initial Catalog* im *ConnectionString*. Beim *SQL-Server-OleDb-Provider* können wir aber alternativ beide Attributbezeichner verwenden.

Beispiel 14.5: Zwei gleichwertige Möglichkeiten

C#

```
conn.ConnectionString = "Provider=SQLOLEDB.1; Data Source=.\\SQLEXPRESS; " +
                        "Initial Catalog=Northwind;Integrated Security=SSPI";
oder
conn.ConnectionString = "Provider=SQLOLEDB.1; Data Source=.\\SQLEXPRESS; " +
                        "Database=Northwind;Integrated Security=SSPI";
label1.Text = conn.DataSource;              // liefert ".\SQLEXPRESS"
label2.Text = conn.Database;                // liefert "Northwind"
```

Provider-Eigenschaft

Es klingt möglicherweise etwas verwirrend: Während wir unter dem Begriff *.NET-Datenprovider* eine Klassenbibliothek für den Datenzugriff verstehen, ist *Provider* auch eine Eigenschaft des *OleDbConnection*-Objekts.

Die *Provider*-Eigenschaft bezeichnet hier die OLE-DB-Schnittstelle, welche die Datenquelle des jeweiligen Herstellers kapselt. Die Tabelle erklärt einige häufig benutzte OLE-DB-Schnittstellen:

Datenquelle	Provider-Eigenschaft
Microsoft Access	Microsoft.Jet.OLEDB.4.0
Microsoft SQL Server	SQLOLEDB.1
Microsoft Indexing Service	MSIDXS.1
Oracle	MSDAORA.1

ServerVersion-Eigenschaft

Diese Eigenschaft liefert eine Zeichenfolge zurück, die die Version der Datenbank enthält. Durch Abprüfen von *ServerVersion* können Sie z. B. gewährleisten, dass keine Abfragen an den Server geschickt werden, die von diesem nicht unterstützt werden (z. B. Abfrageergebnisse als XML liefern).

ConnectionTimeout-Eigenschaft

Obwohl diese Eigenschaft schreibgeschützt ist, haben Sie trotzdem die Möglichkeit, innerhalb des *ConnectionString* anzugeben, wie viel Sekunden der OleDb-Provider versuchen soll, die Verbindung zur Datenbank herzustellen.

Beispiel 14.6: Zeit bis zum Timeout der Verbindungsaufnahme auf zehn Sekunden festlegen

C#

```
conn.ConnectionString = "Provider=SQLOLEDB.1; Data Source=.\\SQLEXPRESS;...;
                         Connect Timeout=10; ... " ;
```

State-Eigenschaft

Diese Eigenschaft liefert den aktuellen Verbindungsstatus. Die möglichen Werte sind Mitglieder der *ConnectionState*-Enumeration.

Konstante	Verbindungszustand
Open	Geöffnet
Closed	Geschlossen
Connecting	Verbindung wird aufgebaut.
Executing	Eine Abfrage wird ausgeführt.
Fetching	Daten werden abgerufen.
Broken	Unterbrochen

14.2.5 Methoden des Connection-Objekts

Open- und Close-Methode

Wenn Sie die *Open*-Methode auf einem bereits geöffneten *Connection*-Objekt ausführen, wird ein Fehler ausgelöst. Hingegen verursacht der Aufruf von *Close* über einer bereits geschlossenen Verbindung keinen Fehler.

HINWEIS: Da Sie standardmäßig mit Verbindungspooling arbeiten, wird die Verbindung nicht wirklich geschlossen, sondern nur zurück an den Pool gesendet.

Es ist keine Vergesslichkeit der Autoren, wenn in manchen Beispielen das *Connection*-Objekt weder mit *Open* geöffnet noch mit *Close* geschlossen wird. Gewissermaßen im Hintergrund können *Fill*- und *Update*-Methode eines *DataAdapter*-Objekts automatisch die Verbindung öffnen (wenn sie nicht schon geöffnet ist) und sie auch wieder schließen, wenn die Operation beendet ist.

ChangeDatabase-Methode

Viele Server, wie z.B. auch der SQL Server, unterstützen mehrere Datenbanken. Mit der *ChangeDatabase*-Methode können Sie die Datenbank zur Laufzeit wechseln, ohne den USE-Befehl verwenden zu müssen.

Beispiel 14.7: Zwei äquivalente Varianten zum Wechseln der Datenbank

C#

```
conn.ChangeDatabase("Northwind");
```

oder

```
OleDbCommand cmd = conn.CreateCommand();
cmd.CommandText = "USE Northwind";
cmd.ExecuteNonQuery();
```

CreateCommand-Methode

Mit dieser Methode können Sie ein neues *Command*-Objekt erzeugen und damit etwas Schreibarbeit einsparen (siehe obiges Beispiel).

14.2.6 Der ConnectionStringBuilder

Um das Zusammenbauen eines *ConnectionString*s etwas übersichtlicher zu gestalten, gibt es die providerspezifische *ConnectionStringBuilder*-Klasse.

Beispiel 14.8: Vergleich von zwei Möglichkeiten für das Erstellen einer Verbindungszeichenfolge zur *Northwind*-Datenbank des SQL Servers

C#

```
using System.Data.SqlClient;
```

Ohne *ConnectionStringBuilder*:

```
string connStr = "Data Source = .\\SQLEXPRESS; Initial Catalog=Northwind;
                  Integrated Security=true";
conn = new SqlConnection(connStr);
```

Mit *ConnectionStringBuilder*:

```
SqlConnectionStringBuilder csb = new SqlConnectionStringBuilder();
csb.DataSource = ".\\SQLEXPRESS";
csb.IntegratedSecurity = true;
csb.InitialCatalog = "Northwind";
```

14.3 Das Command-Objekt

An Abfragen aller Art (SQL-Queries, Stored Procedures) führt beim Programmieren von Datenbankanwendungen kein Weg vorbei. Unter ADO.NET werden für alle Datenbankabfragen *Command*-Objekte benutzt, die zentraler Bestandteil der jeweiligen .NET-Datenprovider sind.

14.3.1 Erzeugen und Anwenden eines Command-Objekts

Wie bei fast allen anderen ADO.NET-Objekten, bieten sich auch zum Erzeugen eines *Command*-Objekts verschiedene Konstruktoren an. Eine übliche Vorgehensweise ist es, die gewünschte Abfrage neben dem zuvor angelegten *Connection*-Objekt an den Konstruktor der Klasse zu übergeben:

Syntax:

```
Command cmd = new Command(string sqlCommand, Connection conn);
```

Das so erzeugte und initialisierte *Command*-Objekt kann dann z. B. an den Konstruktor der *DataAdapter*-Klasse weitergereicht werden, um letztendlich ein *DataSet* zu füllen.

Aber es geht auch ohne *DataAdapter* und *DataSet*, denn um SQL-Anweisungen direkt gegen die Datenquelle zu fahren, kann eine der *Execute*-Methoden (*ExecuteNonQuery*, *ExecuteReader*, *ExecuteScalar*) aufgerufen werden.

Beispiel 14.9: Erzeugen und Anwenden eines *Command*-Objekts

C#

Es werden zwei *SqlCommand*-Objekte erstellt. Mit dem ersten werden die Namen der Firmen aller Pariser Kunden aus der Northwind-Datenbank geändert, mit dem zweiten wird ein *SqlDataAdapter* erstellt, der zum Befüllen eines *DataSet*-Objekts mit den Kundendatensätzen dient.

```
SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                     Initial Catalog=Northwind;Integrated Security=true");
SqlCommand updCmd = new SqlCommand("UPDATE Custmers
          SET CompanyName = 'Pariser Kunde' WHERE City = 'Paris'", conn);
SqlCommand selCmd = new SqlCommand("SELECT CompanyName, ContaktName, City
                    FROM Customers WHERE City = 'Paris'", conn);
SqlDataAdapter da = new SqlDataAdapter(selCmd);
DataSet ds = new DataSet();
conn.Open();
```

UPDATE-Befehl wird gegen die Datenbank gefahren:

```
updCmd.ExecuteNonQuery();
```

DataSet erhält neue Tabelle („PariserKunden") mit Datensätzen gemäß SELECT-Befehl:

```
da.Fill(ds, "PariserKunden");
conn.Close();
```

HINWEIS: Den vollständigen Code finden Sie im Praxisbeispiel in Abschnitt 14.9.2, Eine Aktionsabfrage ausführen.

14.3.2 Erzeugen mittels CreateCommand-Methode

Auch mithilfe der *CreateCommand*-Methode eines *Connection*-Objekts können Sie ein *Command*-Objekt erzeugen. Damit ersparen Sie sich etwas Schreibarbeit.

Beispiel 14.10: Zwei äquivalente Varianten, wenn ein gültiges *Connection*-Objekt vorliegt

C#

Variante A:

```
SqlCommand cmd = new SqlCommand();
cmd.Connection = conn;
```

Variante B:

```
SqlCommand cmd = conn.CreateCommand();
```

14.3.3 Eigenschaften des Command-Objekts

Wir werden uns auch hier auf eine knappe Darstellung der wichtigsten Eigenschaften beschränken.

Connection- und CommandText-Eigenschaft

Beide Eigenschaften werden üblicherweise bereits im Konstruktor übergeben (siehe oben), man kann sie aber auch separat zuweisen.

Beispiel 14.11: Zwei Varianten zum Erzeugen und Initialisieren eines *SqlCommand*-Objekts

C#

```
SqlCommand cmd = new SqlCommand("UPDATE Customers
        SET CompanyName = 'Pariser Firma' WHERE City = 'Paris'", conn);
```

... ist äquivalent zu:

```
SqlCommand cmd = new SqlCommand();
cmd.Connection = conn;
cmd.CommandText = "UPDATE CompanyName SET City = 'Pariser Firma'
                            WHERE City = 'Paris'";
```

CommandTimeout-Eigenschaft

Um festzulegen, wie lange die Ausführung einer Abfrage maximal dauern darf, können Sie der *CommandTimeout*-Eigenschaft einen Wert in Sekunden zuweisen (Standardwert = 30 Sekunden).

Beispiel 14.12: Ein *DataSet* wird mit der Kundenliste der SQL-Datenbank *Northwind* gefüllt, wofür maximal 30 Sekunden zur Verfügung stehen.

C#

```
SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                     Initial Catalog=Northwind;Integrated Security=true");
SqlCommand cmd = new SqlCommand("SELECT CustomerID, CompanyName
                 FROM Customers", conn);
//Die Ausführung der Abfrage darf maximal 30 sek dauern:
cmd.CommandTimeout = 30;
SqlDataAdapter da = new SqlDataAdapter();
da.SelectCommand = cmd;
DataSet ds = new DataSet();
conn.Open();
da.Fill(ds, "Kunden");
conn.Close();
```

HINWEIS: Sollte eine Abfrage dennoch zu lange dauern, so können Sie diese innerhalb einer asynchronen Umgebung mithilfe der *Cancel*-Methode abbrechen.

Zum Ausführen einfacher Datenbankabfragen (wie im obigen Beispiel) können Sie auf das explizite Erzeugen eines *Command*-Objekts verzichten, denn man kann den SQL-String auch direkt dem *DataAdapter* als Parameter übergeben.

Beispiel 14.13: Eine gleichwertige Realisierung des Vorgängerbeispiels

C#

```
SqlDataAdapter da = new SqlDataAdapter("SELECT CustomerID, CompanyName
            FROM Customers", conn);
da.SelectCommand.CommandTimeout = 30;
```

CommandType-Eigenschaft

Mit der *CommandType*-Eigenschaft definieren Sie die auszuführende Operation. Mittels der gleichnamigen Enumeration stehen drei Möglichkeiten zur Verfügung:

- *Text* (Standardwert)
  Hier können Sie eine frei definierbare SQL-Abfrage übergeben.
- *StoredProcedure*
  Hier soll eine in der Datenbank gespeicherte Prozedur bzw. Auswahlabfrage aufgerufen werden.
- *TableDirect*
  Hier wird direkt der Name einer Tabelle angegeben (entspricht *SELECT * FROM <Tabellenname>*).

Beispiel 14.14: Aufruf der Stored Procedure „Sales by Years“ in der Datenbank *Northwind*

C#

```
SqlCommand cmd = new SqlCommand("Sales by Year", conn);
cmd.CommandType = CommandType.StoredProcedure;
SqlParameter parm1 = new SqlParameter("@Beginning_Date", SqlDbType.DateTime);
```

Definition als Input-Parameter:

```
parm1.Direction = ParameterDirection.Input;
```

Das *Beginn*-Datum wird der ersten *TextBox* entnommen:

```
parm1.Value = Convert.ToDateTime(textBox1.Text);
```

Parameter hinzufügen:

```
cmd.Parameters.Add(parm1);
```

14.3.4 Methoden des Command-Objekts

ExecuteNonQuery-Methode

Diese Methode setzen Sie vor allem ein, um Aktionsbefehle auf Basis von UPDATE, INSERT oder DELETE direkt gegen die Datenbank auszuführen (also ohne Verwendung von *DataAdapter* und *DataSet*). Rückgabewert ist hier die Anzahl der betroffenen Datensätze (sonst –1).

Beispiel 14.15: *ExecuteNonQuery*-Methode

C#

Ein *SqlCommand*-Objekt wird erzeugt und eine UPDATE-Anweisung gegen die Datenbank ausgeführt. Die Anzahl betroffener Datensätze wird angezeigt (ein gültiges *SqlConnection*-Objekt *conn* wird vorausgesetzt).

```
string updStr =
   "UPDATE Customer SET CompanyName = 'Pariser Firma' WHERE City = 'Paris'";
SqlCommand updCmd = new SqlCommand(updStr, conn);
conn.Open();
//SQL-Anweisung ausführen und Anzahl betroffener Datensätze anzeigen:
label1.Text = cmd.ExecuteNonQuery.ToString();
```

HINWEIS: Das ausführliche Beispiel befindet sich im Praxisbeispiel in Abschnitt 14.9.2, Eine Aktionsabfrage ausführen.

Weitere Möglichkeiten für Aktionsbefehle sind die Abfrage der Struktur einer Datenbank oder das Erstellen von Datenbankobjekten, wie z. B. Tabellen.

ExecuteReader-Methode

Auf Basis eines SELECT-Befehls erzeugt diese Methode ein *DataReader*-Objekt. Ein Instanziieren des *DataReader*s mittels *new*-Konstruktor entfällt deshalb.

Beispiel 14.16: Auf Basis eines *SqlConnection*-Objekts und eines SELECT-Befehls wird ein *SqlCommand*-Objekt erstellt und zum Erzeugen eines *SqlDataReader*-Objekts verwendet.

C#

```
string selStr = "SELECT CompanyName, ContactName, City
                FROM Customers WHERE City = 'Paris'";
SqlCommand selCmd = new SqlCommand(selStr, conn);
conn.Open();
SqlDataReader dr = selCmd.ExecuteReader(CommandBehavior.CloseConnection);
```

ExecuteScalar-Methode

Rückgabewert dieser Methode ist das Objekt der ersten Spalte der ersten Zeile aus der Menge der zurückgegebenen Datensätze. Generell eignet sich die *ExecuteScalar*-Methode des *Command*-Objekts für alle Abfragen, bei denen man nur an der Rückgabe eines einzigen Werts interessiert ist.

Beispiel 14.17: Abfrage des Namens der Firma eines bestimmten Kunden

C#

```
SqlCommand cmd = new SqlCommand("SELECT Companyname
                 FROM Customers WHERE CustomerId = 'ALFKI'", conn);
conn.Open();
label1.Text = Convert.ToString(cmd.ExecuteScalar);
```

Besonders vorteilhaft kann man *ExecuteScalar* zur Ausführung von Aggregatfunktionen verwenden, was weniger Aufwand erfordert als die Anwendung der *ExecuteReader*-Methode.

Beispiel 14.18: Aus der Datenbank *Northwind* wird die Anzahl der in Paris wohnhaften Kunden abgefragt und angezeigt.

C#

```
SqlCommand cmd = new SqlCommand();
cmd.Connection = conn;
cmd.CommandText = "SELECT COUNT(*) AS Anzahl FROM Customers
                   WHERE City = 'Paris'";
cmd.Connection.Open();
// oder auch: conn.Open();
label1.Text = cmd.ExecuteScalar().ToString();
cmd.Connection.Close()
```

14.3.5 Freigabe von Connection- und Command-Objekten

In einfachen Codebeispielen stellt man häufig fest, dass der Aufruf von *Dispose()* auf *SqlConnection*- und *SqlCommand*-Objekten fehlt.[1] Auch wurde der Datenzugriffscode nicht in *try-finally*-Blöcke eingerahmt.

Beispiel 14.19: Die (leider nicht ganz saubere) Programmierung einer Datenbankverbindung

C#

```
SqlConnection conn = new SqlConnection(connString);
SqlCommand cmd = new SqlCommand(cmdString, conn);
conn.Open();
cmd.ExecuteNonQuery();
conn.Close();
```

Das Problem ist, dass *SqlConnection* und *SqlCommand* die Schnittstelle *IDisposable* implementieren, d. h., es können auch Ressourcen aus nicht verwaltetem (unmanaged) Code zu bereinigen sein. Als Programmierer müssen Sie dann unter allen Umständen absichern, dass *Dispose()* auf diesen Objekten aufgerufen wird, nachdem die Arbeit mit ihnen beendet ist. Weil bei Nichtverfügbarkeit der Datenbank immer ein Fehler auftreten kann, sollten Sie den Aufruf von *Dispose()* auch für diesen Fall gewährleisten.

Das Problem lässt sich elegant mittels des *using*-Schlüsselworts lösen, welches Ihnen lästige Schreibarbeiten abnimmt, denn intern wird automatisch ein *try-finally*-Block um das entsprechende Objekt generiert und beim Beenden wird für das Objekt *Dispose()* aufgerufen.

Beispiel 14.20: Saubere Programmierung des Vorgängerbeispiels

C#

```
using (SqlConnection conn = new SqlConnection(connString))
{
    using (SqlCommand cmd = new SqlCommand(cmdString, conn))
    {
        conn.Open();
        cmd.ExecuteNonQuery();
    }
}
```

Der intern generierte Code für obige Zeilen dürfte etwa folgendermaßen aussehen:

```
SqlConnection conn =  null;
SqlCommand cmd = null;
try
{
    conn = new SqlConnection(connString);
    cmd = new SqlCommand(cmdString, conn);
    conn.Open();
    cmd.ExecuteNonQuery();
}
finally
```

[1] Auch die Codebeispiele dieses Buchs bilden da (aus Platzgründen!) keine Ausnahme.

```
{
    if (null != cmd) cmd.Dispose();
    if (null != conn) conn.Dispose();
}
```

HINWEIS: Falls Sie, wie im obigen Beispiel, den Aufruf von *Close()* für die *SqlConnection* vermissen, seien Sie trotzdem unbesorgt: Intern überprüft *Dispose()* den Status der Verbindung und schließt diese für Sie.

14.4 Parameter-Objekte

In vielen Fällen enthält ein *Command*-Objekt Parameter bzw. Parameter-Auflistungen, mit denen parametrisierte Abfragen durchführbar sind.

14.4.1 Erzeugen und Anwenden eines Parameter-Objekts

Einer der möglichen Konstruktoren:

Syntax:

```
Parameter prm = new Parameter(string pName, DbType pType);
```

Nach dem Zuweisen weiterer Eigenschaften erfolgt das Hinzufügen zur *Parameters*-Auflistung des *Command*-Objekts:

```
cmd.Parameters.Add(Parameter prm );
```

Beispiel 14.21: Ein *SqlParameter*-Objekt *p1* wird zur *Parameters*-Collection eines vorhandenen *SqlCommand*-Objekts hinzugefügt.

C#

Im Konstruktor Namen und Datentyp übergeben:

```
SqlParameter p1 = new SqlParameter("@Geburtstag", SqlDbType.DateTime);
```

Datumswert aus einer *TextBox* zuweisen ...

```
p1.Value = Convert.ToDateTime(textBox1.Text);
```

... und zum *SqlCommand*-Objekt hinzufügen:

```
cmd.Parameters.Add(p1);
```

Die so erzeugten Parameter werden zur Laufzeit in die *CommandText*-Eigenschaft des *Command*-Objekts „eingebaut“.

Beispiel 14.22: Der definierte Parameter *@Geburtstag* wird in einer SQL-Abfrage eingefügt.

C#

```
cmd.CommandText = "SELECT * FROM Employees WHERE (BirthDate > @Geburtstag)";
```

14.4.2 Eigenschaften des Parameter-Objekts

ParameterName- und Value-Eigenschaft

Beide Eigenschaften dürften selbsterklärend sein.

Beispiel 14.23: Eine alternative Zuweisung für das obige erste Beispiel wäre:

C#

```
p1.ParameterName = "@Geburtstag";
p1.Value = Convert.ToDateTime(textBox1.Text);
```

DbType, OleDbType und SqlDbType-Eigenschaft

Durch das Spezifizieren des Datentyps wird der Wert des Parameters dem Datentyp des .NET-Datenproviders angepasst, bevor er an die Datenquelle weitergereicht wird. Fehlt die Typangabe, so leitet ihn ADO.NET von der *Value*-Eigenschaft des *Parameter*-Objekts ab.

Alternativ zur *OleDbType*- bzw. *SqlDbType*-Eigenschaft kann der Datentyp eines Parameters auch allgemein (generisch) aus *System.Data.DbType* abgeleitet werden.

Beispiel 14.24: Ein *Byte*-Parameter wird erzeugt, initialisiert und zur *Parameters*-Collection eines *SqlCommand*-Objekts hinzugefügt.

C#

```
SqlParameter prm = cmd.Parameters.Add("@p2", SqlDbType.TinyInt);
```

Direction-Eigenschaft

Die Eigenschaft bestimmt die Richtung des Parameters relativ zum *DataSet*. Die *ParameterDirection*-Enumeration enthält die in der Tabelle aufgeführten Werte:

ParameterDirection	Beschreibung
Input	Es handelt sich um einen Eingabeparameter (Standard).
InputOutput	Der Parameter unterstützt sowohl Ein- als auch Ausgabe.
Output	Es handelt sich um einen Ausgabeparameter.
ReturnValue	Der Parameter ist ein Rückgabewert aus einer Operation.

Beispiel 14.25: Ein *OleDbParameter* wird erstellt und seine *Direction*-Eigenschaft festgelegt.

C#

```
public void CreateOleDbParameter()
{
    OleDbParameter p1 = new OleDbParameter("Description", OleDbType.VarChar, 50);
    p1.IsNullable = true;
    p1.Direction = ParameterDirection.Output;
}
```

14.5 Das CommandBuilder-Objekt

Das manuelle Zuweisen der *Insert-*, *Update-* und *DeleteCommand*-Eigenschaften des *DataAdapter*s ist mitunter eine ziemlich aufwendige Angelegenheit. Falls Ihr *DataAdapter* nur auf einer einzigen Datenbanktabelle aufsetzt, können Sie vorteilhaft den *CommandBuilder* zum automatischen Generieren der *Command*-Objekte verwenden.

14.5.1 Erzeugen

Voraussetzung für den Einsatz eines *CommandBuilder*-Objekts ist, dass dem *DataAdapter* vorher die *SelectCommand*-Eigenschaft zugewiesen wurde. Eine einzige Anweisung reicht dann aus, um einen *CommandBuilder* mit einem *DataAdapter* zu verkoppeln:

Syntax:

```
CommandBuilder cmdBuilder = new CommandBuilder(DataAdapter da);
```

Der *CommandBuilder* verfolgt nun argwöhnisch alle am *DataSet* vorgenommenen Änderungen und generiert die erforderlichen Queries bzw. *Command*-Objekte selbstständig im Hintergrund.

14.5.2 Anwenden

Die *Update*-Methode des *DataAdapter*s würde im folgenden Beispiel ohne *OleDbCommandBuilder* fehlschlagen.

Beispiel 14.26: Aktualisieren der Kunden-Tabelle aus der *Northwind*-Datenbank

C#

Beim Instanziieren erhält der *DataAdapter* automatisch auch seine *SelectCommand*-Eigenschaft, sodass diese nicht explizit zugewiesen werden muss:

```
SqlDataAdapter da = new SqlDataAdapter("SELECT * FROM Customers", conn);
```

Ein *SqlCommandBuilder* wird mit dem *SqlDataAdapter* verbunden:

```
SqlCommandBuilder cmdB = new SqlCommandBuilder(da);
```

Ein *DataSet* wird mit den Daten gefüllt:

```
DataSet ds = new DataSet();
conn.Open();
da.Fill(ds, "Kunden");
...
```

Nachdem die Daten geändert wurden, werden die Änderungen in die Datenbank zurückgeschrieben:

```
da.Update(ds, "Kunden");
```

Beim Aufruf von *Dispose* wird die Zuordnung von *CommandBuilder* zu *DataAdapter* aufgehoben und die generierten Befehle werden nicht mehr verwendet.

HINWEIS: Ein Beispiel für den Einsatz des *CommandBuilder*s finden Sie im Praxisbeispiel in Abschnitt 14.9.4, Die Datenbank aktualisieren.

14.6 Das DataReader-Objekt

Häufig genügt ein Lesezugriff auf die Datensätze. Dabei müssen im Frontend meist nur einige für die Listendarstellung benötigte Komponenten gefüllt bzw. aktualisiert werden (*ListBox*, *ComboBox*, *ListView*, *TreeView*, *DataGridView* usw.).

Im .NET-Framework gibt es für diesen Zweck den *DataReader*. Diese Klasse ist für einen einmaligen ReadOnly-Hochgeschwindigkeitszugriff auf eine Datensatzgruppe optimiert und ähnelt anderen *Reader*-Objekten wie *TextReader*, *StreamReader* und *XmlReader*. In Abhängigkeit vom verwendeten .NET-Datenprovider gibt es auch hier unterschiedliche Typen (*SqlDataReader*, *OleDbDataReader*).

14.6.1 DataReader erzeugen

Einen *DataReader* erzeugt man in der Regel nicht mit dem *new*-Konstruktor, sondern mit der *ExecuteReader*-Methode des zugrunde liegenden *Command*-Objekts:

Syntax:

```
DataReader dr = cmd.ExecuteReader();
```

Mitunter wird auch dem *Execute*-Konstruktor als Argument der Wert *CloseConnection* (aus der *CommandBehavior*-Enumeration) übergeben. Damit ist gewährleistet, dass die Verbindung automatisch nach dem Durchlauf des *DataReader*s geschlossen wird.

Beispiel 14.27: Ein *DataReader* wird instanziiert.

C#

```
DataReader dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);
```

14.6.2 Daten lesen

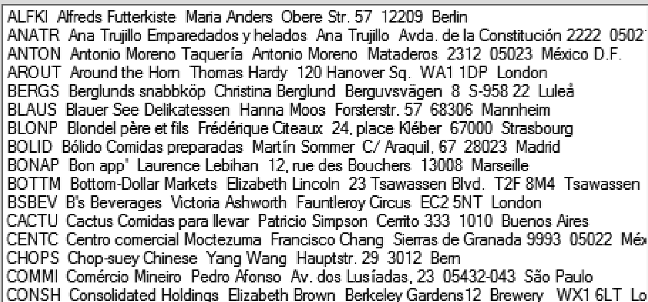
Das Auslesen der Informationen innerhalb einer Schleife ist typisch für die Arbeit mit dem *DataReader*.

Beispiel 14.28: Die Kundentabelle aus der *Northwind*-Datenbank wird zeilenweise ausgelesen.

C#

```
const string SQL = "SELECT * FROM Customers ORDER BY CustomerId";
SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                     Initial Catalog=Northwind;Integrated Security=true");
SqlCommand cmd = new SqlCommand(SQL, conn);
SqlDataReader dr;
conn.Open();
dr = cmd.ExecuteReader();
string str;
string tab = "  ";
while (dr.Read())
{
  str = dr["CustomerId"] + tab;
  str += dr["CompanyName"] + tab;
  str += dr["ContaktName"] + tab;
  str += dr["Street"] + tab;
  str += dr["PostalCode"] + tab;
  str += dr["City"];
  listBox1.Items.Add(str);
}
dr.Close();
conn.Close();
```

Ergebnis



HINWEIS: Es ist wichtig, dass Sie den *DataReader* so schnell wie möglich nach dem Auslesen der Daten wieder schließen, da sonst das *Connection*-Objekt blockiert ist!

14.6.3 Eigenschaften des DataReaders

Item-Eigenschaft

Diese Eigenschaft ermöglicht den Zugriff auf die aktuelle Spalte, der Rückgabewert ist vom *Object*-Datentyp (ähnlich der *Item*-Eigenschaft des *DataRow*-Objekts). Falls der Datentyp vorher bekannt ist, sollte man eine der *Get*-Methoden (siehe unten) für den Zugriff verwenden.

FieldCount-Eigenschaft

Diese Eigenschaft liefert die Gesamtanzahl der Datensätze.

IsClosed-Eigenschaft

Der Wert ist *true*, falls der *DataReader* geschlossen ist.

14.6.4 Methoden des DataReaders

Read-Methode

Damit wird das automatische Weiterbewegen zum nächsten Datensatz innerhalb der *while*-Schleife ermöglicht (Rückgabewert *true/false*).

GetValue- und GetValues-Methode

Während *GetValue* – ähnlich der *Item*-Eigenschaft – den Wert einer Spalte (basierend auf dem Spaltenindex) zurückgibt, nimmt *GetValues* ein Array entgegen, in welchem der *DataReader* den Inhalt der aktuellen Zeile ablegt. Mit *GetValues* wird beste Performance erreicht.

GetOrdinal- und ähnliche Methoden

Eine Vielzahl von *Get*...-Methoden ermöglichen ein Konvertieren der gelesenen Werte in fast jeden Datentyp.

Beispiel 14.29: Ein Datumswert aus der *Employee*-Tabelle wird ausgelesen.

C#

```
DateTime aDate;
aDate = dr.GetDateTime(dr.GetOrdinal("BirthDate"));
```

Bemerkungen

Nach dem Laden der Daten mit dem *DataReader* kopiert man in der Regel die Datensätze zeilenweise in Objekte um (Entity Framework erledigt das automatisch für Sie).

Beispiel 14.30: Einlesen von Kundendaten

C#

```
Kunde kd = new Kunde();
kd.ID = Convert.ToInt64(dr["CustomerId"]);
k.Ort = Convert.ToString(dr["City"]);
```

Das ist insbesondere bei vielen Tabellenspalten sehr arbeitsaufwendig. Außerdem liegt die meiste Arbeit noch vor Ihnen, denn in der Regel wollen Sie die Daten nicht nur lesen, sondern Sie wollen auch Änderungen speichern.

14.7 Das DataAdapter-Objekt

Datenadapter werden in einer Art „Brückenfunktion" dazu genutzt, Daten mittels SQL-Anweisungen aus Datenquellen in *DataSet*s zu transportieren bzw. um Datenquellen mit den geänderten Inhalten von *DataSet*s zu aktualisieren. Das *DataAdapter*-Objekt verwendet das *Connection*-Objekt des jeweiligen .NET-Datenproviders, um eine Verbindung zu einer Datenquelle herzustellen, und ist außerdem auf verschiedene *Command*-Objekte angewiesen.

Hin- und Rücktransport der Daten zwischen Datenquelle und *DataSet* werden mit der *Fill*- und *Update*-Methode des *DataAdapter*s realisiert. Beide lösen die entsprechenden SQL-Anweisungen aufgrund der dem *DataAdapter* übergebenen *Command*-Objekte aus.

14.7.1 DataAdapter erzeugen

Mehrere überladene Konstruktoren stellen den Newcomer vor die Qual der Wahl.

Konstruktor mit SELECT-String und Connection-Objekt

Im einfachsten Fall kommt man sogar ohne *Command*-Objekt aus, es genügt, dem Konstruktor des *DataAdapter*-Objekts eine SELECT-Anweisung und das *Connection*-Objekt als Parameter zu übergeben:

Syntax:

```
DataAdapter da = new DataAdapter(string selectStr, Connection conn);
```

Beispiel 14.31: Ein *DataAdapter* füllt ein *DataSet* mit Datensätzen aus der *Northwind*-Datenbank.

C#

```
using System.Data.SqlClient;
...
SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                     Initial Catalog=Northwind;Integrated Security=true");
SqlDataAdapter da = new SqlDataAdapter("SELECT * FROM Customers
                    WHERE City = 'Paris'", conn);
DataSet ds = new DataSet();
conn.Open();
da.Fill(ds, "PariserKunden");
conn.Close();
```

Konstruktor mit SelectCommand-Objekt

Eine weitere Möglichkeit ist die Verwendung eines Konstruktors, dem ein *Command*-Objekt (SELECT-Befehl) zu übergeben ist:

Syntax:

```
DataAdapter da = new DataAdapter(Command selectCommand);
```

Beispiel 14.32: Das Vorgängerbeispiel wird mit einem *Command*-Objekt realisiert.

C#

```
using System.Data.SqlClient;
...
SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                     Initial Catalog=Northwind;Integrated Security=true");
SqlCommand cmd = new SqlCommand("SELECT CompanyName FROM Customers
                                 WHERE City = 'Paris'");
cmd.Connection = conn;
SqlDataAdapter da = new SqlDataAdapter(cmd);
DataSet ds = new DataSet();
conn.Open();
da.Fill(ds, "PariserKunden");
conn.Close();
```

14.7.2 Command-Eigenschaften

Ein *DataAdapter* benötigt für die komplette Zusammenarbeit mit der Datenquelle vier verschiedene *Command*-Objekte, die als Eigenschaften zugewiesen werden:

- *SelectCommand* zur Abfrage
- *UpdateCommand* zur Aktualisierung
- *InsertCommand* zum Einfügen
- *DeleteCommand* zum Löschen

Beispiel 14.33: Realisierung der Vorgängerbeispiele mittels *SelectCommand*-Eigenschaft

C#

```
using System.Data.SqlClient;
...
SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                     Initial Catalog=Northwind;Integrated Security=true");
SqlCommand cmd = new SqlCommand("SELECT CompanyName FROM Customers
                                 WHERE City = 'Paris'");
cmd.Connection = conn;
SqlDataAdapter da = new SqlDataAdapter();
da.SelectCommand = cmd;
DataSet ds = new DataSet();
conn.Open();
da.Fill(ds, "PariserKunden");
conn.Close();
```

Die *SelectCommand*-Eigenschaft muss gesetzt werden, **bevor** die *Fill*-Methode des *DataAdapters* aufgerufen wird.

14.7.3 Fill-Methode

Die relativ unkomplizierte *Fill*-Methode des *DataAdapters* hatten Sie bereits in zahlreichen Beispielen kennengelernt. Hier noch einmal die am häufigsten benutzte Aufrufvariante:

Syntax:

```
DataAdapter da.Fill(DataSet ds, string tblName)
```

Beispiel 14.34: Ein *DataSet* wird mit der Kundentabelle aus der *Northwind*-Datenbank gefüllt. Im *DataSet* sollen die Namen aller Firmen aus Paris geändert werden in „Pariser Firma“.

C#

```
SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                     Initial Catalog=Northwind;Integrated Security=true");
SqlDataAdapter da = new SqlDataAdapter("SELECT * FROM Customers", conn);
DataSet ds = new DataSet();
da.Fill(ds, "Kunden");
```

Das Arbeiten mit den Daten im *DataSet*:

```
DataTable dt = ds.Tables["Kunden"];
foreach (DataRow cRow in dt.Rows)
// alle Zeilen der DataTable durchlaufen
{
    if (cRow["City"].ToString() == "Paris")
    {
       cRow["City"] = "Pariser Firma";
    }
}
...
```

Das Beispiel wird im folgenden Abschnitt fortgesetzt!

Begrenzen der Datenmenge

Geht es nur um die Übertragung kleinerer Datenmengen, so ist die bislang praktizierte Vorgehensweise problemlos, nicht aber wenn es sich um Hunderte von Datensätzen handelt.

Abhilfe schafft eine (überladene) Version der *Fill*-Methode, die die Anzahl der zu transportierenden Datensätze begrenzt:

Syntax:

```
int z  = DataAdapter da.Fill(DataSet ds, int start, int anzahl, string tblName);
```

start = Nummer der Startzeile

anzahl = Anzahl der abzurufenden Datensätze

z = Anzahl der tatsächlich zurückgegebenen Datensätze

Beispiel 14.35: Ab Zeile 100 werden 50 Zeilen aus der Datenbank abgerufen und in die „Kunden"-Tabelle gefüllt.

C#

```
int z = da.Fill(ds, 100, 50, "Kunden");
```

14.7.4 Update-Methode

Irgendwann einmal müssen die im *DataSet* vorgenommenen Änderungen in die Datenquelle zurückgeschrieben werden. Zu diesem Zweck wird (kurzzeitig) eine Verbindung zur Datenbank aufgebaut. Genauso wie beim Füllen spielt auch hier ein *DataAdapter*-Objekt die Vermittlerrolle, wobei dessen *Update*-Methode gewissermaßen das Pendant zur *Fill*-Methode ist und zum Zurückschreiben der im *DataSet* vorgenommenen Änderungen in die Datenquelle dient.

Genauso wie die *Fill*-Methode benötigt die *Update*-Methode als Parameter die Instanz eines *DataSet*s und (optional) den Namen der *DataTable*.

Syntax:

```
DataAdapter da.Update(DataSet ds, string tblName);
```

Bei der *Update*-Methode läuft es nicht ganz so einfach ab wie bei der *Fill*-Methode, da ein *DataSet* völlig autark existiert und nur gelegentlich mit der Datenbank verbunden wird.

Beispiel 14.36: (*Fortsetzung*)

C#

Ziel ist das Zurückschreiben der in der Spalte „CompanyName" (und nur dort!) vorgenommenen Änderungen in die Datenquelle. Grundlage ist eine UPDATE-Anweisung mit zwei Parametern (die @ sind die Platzhalter):

```
...
SqlCommand cmd = new SqlCommand("UPDATE Customers
        SET CompanyName = @name WHERE CustomerId = @id", conn);
```

Der *Add*-Methode werden Parametername, Datentyp und Spaltenbreite übergeben:

```
cmd.Parameters.Add("@name", SqlDbType.NVarChar, 30);
```

Für die Schlüsselspalte:

```
SqlParameter prm = cmd.SqlParameters.Add("@id", SqlDbType.NVarChar);
prm.SourceColumn = "CustomerId";
```

Der ursprüngliche Wert (beim Füllen des *DataSets*) ist maßgebend:

```
da.UpdateCommand = cmd;
da.Update(ds, "Kunden");
```

HINWEIS: Der Kern der Aktualisierungslogik liegt in der WHERE-Bedingung der UPDATE-Anweisung. Der Datensatz wird nur dann aktualisiert, wenn der Wert der Schlüsselspalte, mit dem er geladen wurde, noch vorhanden ist.

- Durch Einsatz eines *CommandBuilder*-Objekts kann das manuelle Erstellen der *UpdateCommand*-, *InsertCommand*- und *DeleteCommand*-Eigenschaften automatisiert werden.
- Ein komplettes Beispiel finden Sie im Praxisbeispiel in Abschnitt 14.9.4, Die Datenbank aktualisieren.

14.8 Entity Framework

14.8.1 Überblick

Das Entity Framework ist Microsofts ORM (Objektrelationaler Mapper), der den Zugriff auf die Datenbank über ein Objektmodell gewährleistet. Für jede Tabelle der Datenbank, auf die der Zugriff gewährt werden soll, wird im Objektmodell eine Klasse definiert (eine sogenannte Entität). Jede Spalte wird dabei durch eine Property dargestellt. Methoden wird man in den Entitätsklassen nicht finden.

Da wir heutzutage zumeist mit relationalen Datenbanken arbeiten, ist auch noch interessant, wie Relationen im Objektmodell dargestellt werden, da es ja keine Fremdschlüsselbeziehungen bei Objekten gibt.

Dies wird in EF dadurch realisiert, dass das übergeordnete Objekt eine zusätzliche Property als Liste des untergeordneten Objekts enthält, und auch das untergeordnete Objekt enthält als zusätzliche Property einen Verweis auf das übergeordnete Objekt. Diese Properties werden auch als *NavigationProperties* bezeichnet.

Diese NavigationProperties werden jedoch erst geladen, wenn ein Zugriff auf diese Eigenschaft erfolgt. Dieser Vorgang nennt sich *LazyLoading*. Würden die abhängigen Objekte auch sofort geladen, würde sehr schnell die Gefahr bestehen, dass man beim Lesen eines Datensatzes sich die halbe Datenbank in den Speicher zieht.

HINWEIS: Das Entity Framework (aktuell in der Version 6) ist nicht Bestandteil des .NET Frameworks. Das Entity Framework muss über den Nuget-Paketmanager zu Ihrem Projekt hinzugefügt werden.

Der gesamte Datenzugriff wird dann über eine Klasse, die von *DbContext* erbt, erledigt. In Ihrem abgeleiteten Context definieren Sie lediglich, welche Objekte bearbeitet werden und auf welche Tabellen somit zugegriffen wird. Die gesamte Basisfunktionalität zum lesenden und schreibenden Zugriff steckt bereits in der Basisklasse. Sie müssen somit keine SQL-Befehle mehr in Ihrer Anwendung definieren und laden und speichern Ihre Daten nur mit bereits bestehenden Methoden der von *DbContext* abgeleiteten Klasse.

Zum Erzeugen des Objektmodells gibt es prinzipiell zwei unterschiedliche Technologien, die wir uns in diesem Einführungskapitel nun ansehen wollen.

Zum einen können Sie von einer bestehenden Datenbank das Objektmodell automatisch generieren lassen (*DatabaseFirst*). Bei Änderungen am Datenbankmodell können Sie dann Ihr Modell immer wieder aktualisieren. Sollten Tabellenspalten gelöscht oder umbenannt oder ganze Tabellen gelöscht worden sein, dann erkennen Sie den jetzt fehlerhaften Zugriff+ sofort durch Compilerfehler. Ohne einen Objektrelationalen Mapper ist das ein nicht zu unterschätzender Aufwand, da ja sämtliche SQL-Anweisungen in Zeichenketten stehen, die der Compiler natürlich nicht analysieren kann.

DatabaseFirst gab es seit den ersten Versionen von Entity Framework, jedoch wird momentan die zweite Zugriffstechnologie *CodeFirst* empfohlen. Zum einen ist *CodeFirst* wesentlich wartbarer und zum anderen wird *DatabaseFirst* in zukünftigen Versionen des Entity Framework Cores nicht mehr angeboten.

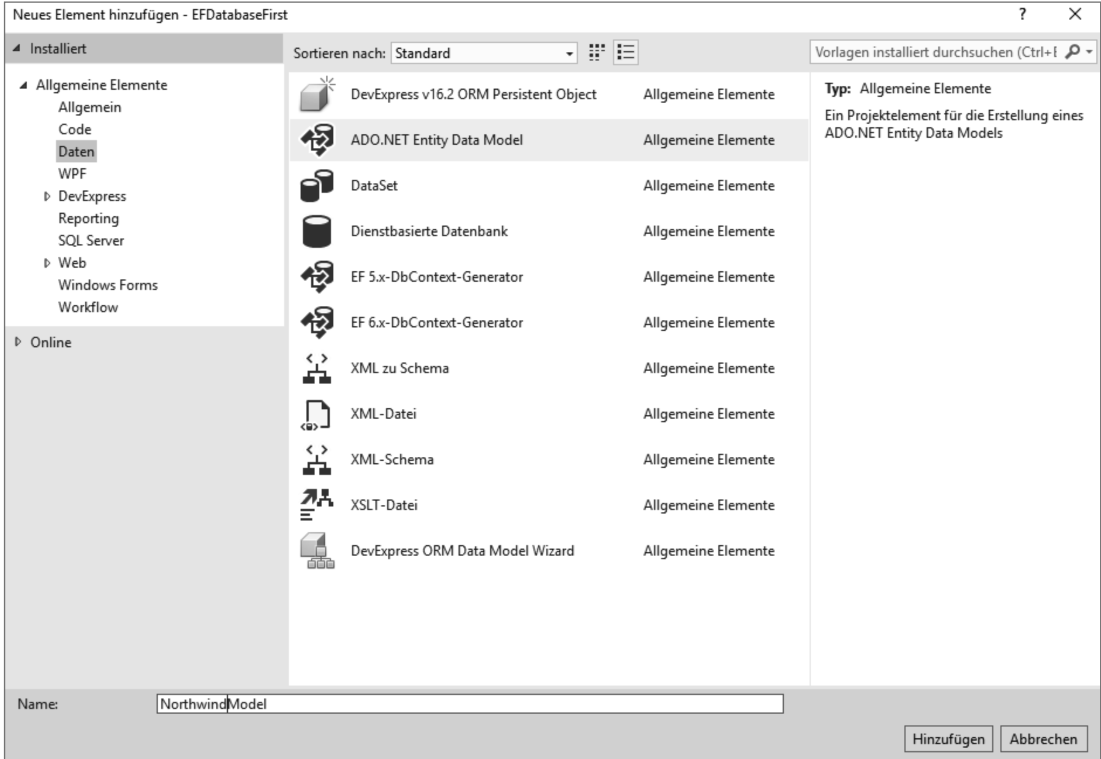
Bei *CodeFirst* definieren Sie das Objektmodell und aus diesen Klassendefinitionen können Sie dann automatisch die Datenbank erzeugen. Durch Migrationen können Sie außerdem definieren, dass Änderungen an Ihrem Objektmodell automatisch zu einer Anpassung der Datenstruktur führen.

HINWEIS: Es gibt auch die Möglichkeit, aus einer bestehenden Datenbank ein CodeFirst-Modell zu erzeugen. Sie erhalten dann eine 1:1-Abbildung des Modells. Sämtliche weiteren Änderungen der Struktur erfolgen dann aber über Ihre Entitätsklassen und durch Migrationen wird die Datenbankstruktur dann aktualisiert.

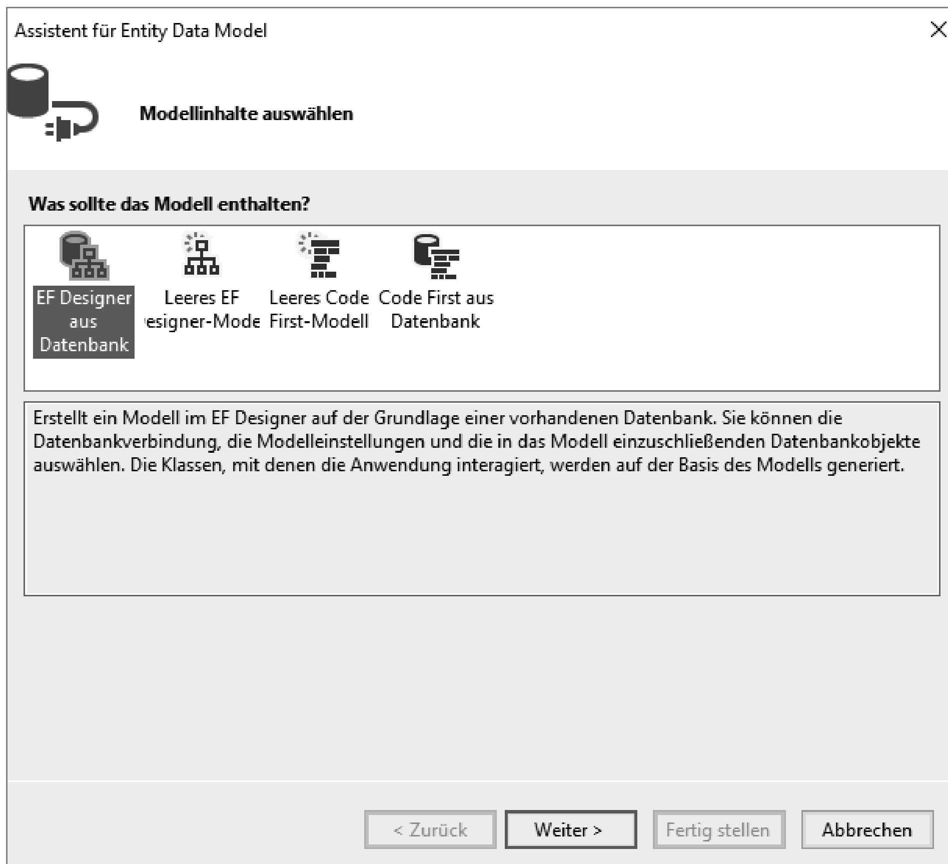
So, nun aber genug der Theorie. Schauen wir uns doch beide Sachen einfach mal praktisch an. Als Beispiel dient wieder unsere bereits bekannte *Northwind*-Datenbank.

14.8.2 DatabaseFirst

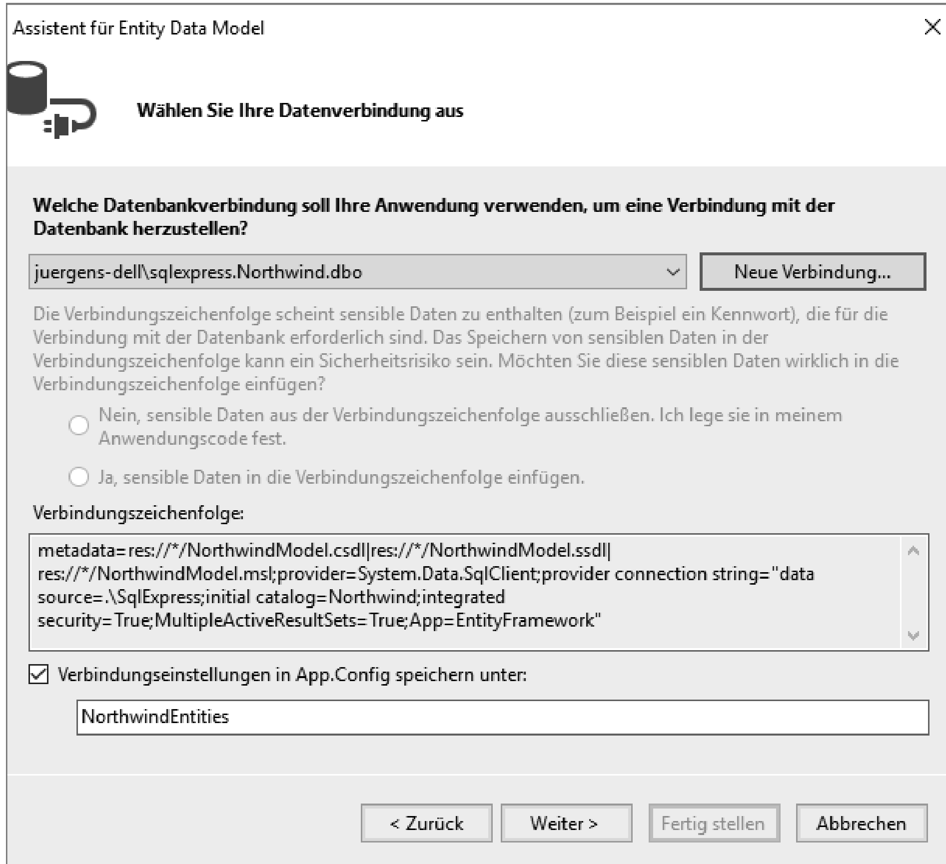
Zuerst müssen wir einem beliebigen Projekttyp (ich werde eine WindowsFormsApp als Beispiel definieren) ein neues Element vom Typ *ADO.NET Entity Data Model* zum Projekt hinzufügen und geben diesem den Namen *NorthwindModel*.



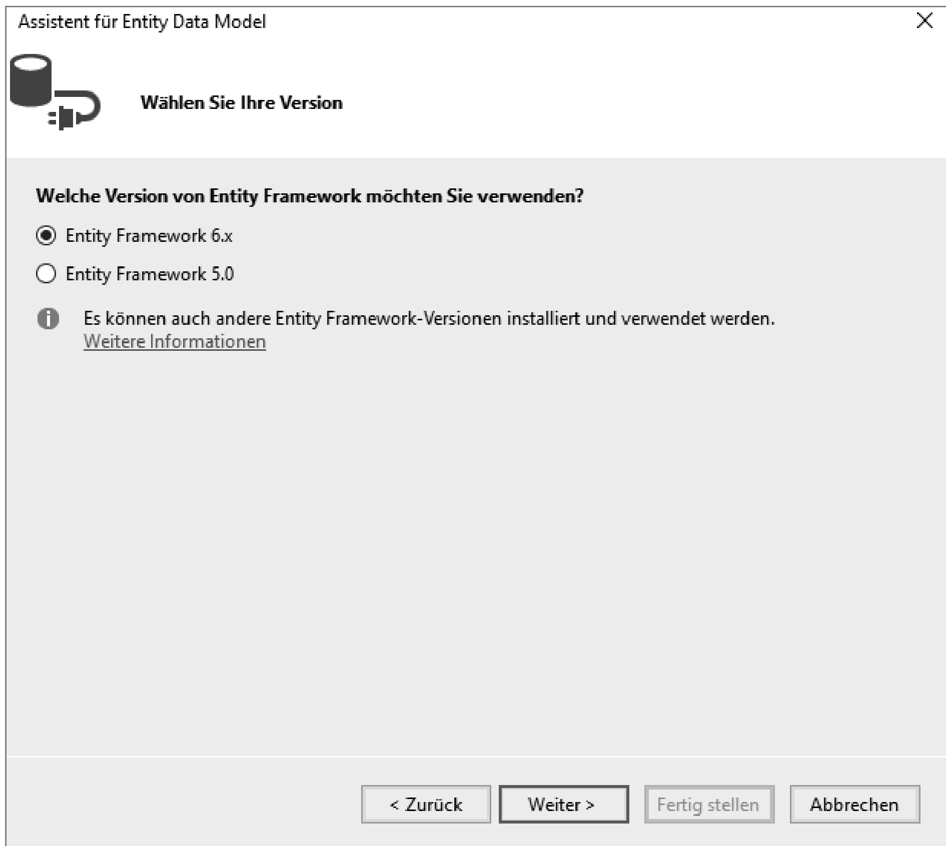
Im nächsten Schritt wählen Sie aus, ob Sie *DatabaseFirst* oder *CodeFirst* verwenden wollen:



Im folgenden Dialog wird dann die Datenbank ausgewählt, aus deren Struktur das Modell erzeugt wird:



Sollten Sie das Entity Framework noch nicht per Nuget-Paket zu Ihrem Projekt hinzugefügt haben, erledigt das der Assistent für Sie. Wählen Sie einfach aus, welche Version von Entity Framework verwendet werden soll (ich würde immer die aktuellste empfehlen):



Im abschließenden Schritt definieren Sie nur noch, welche Datenbankobjekte in Ihr Objektmodell integriert werden sollen. Der Übersichlichkeit halber wähle ich jetzt nur die beiden Tabellen *Customers* und *Orders* aus:

Assistent für Entity Data Model

Wählen Sie Ihre Datenbankobjekte und Einstellungen.

Welche Datenbankobjekte möchten Sie in Ihr Modell einschließen?

- [x] dbo
  - [ ] __MigrationHistory
  - [ ] Categories
  - [ ] CustomerCustomerDemo
  - [ ] CustomerDemographics
  - [x] Customers
  - [ ] Employees
  - [ ] EmployeeTerritories
  - [ ] Order Details
  - [x] Orders
  - [ ] Products
  - [ ] Region

- [ ] Generierte Objektnamen in den Singular oder Plural setzen
- [x] Fremdschlüsselspalten in das Modell einbeziehen
- [x] Ausgewählte gespeicherte Prozeduren und Funktionen in das Entitätsmodell importieren

Modellnamespace:

NorthwindModel

< Zurück | Weiter > | Fertig stellen | Abbrechen

Durch einen Klick auf den Button *Fertig stellen* wird das Modell nun komplett erzeugt. Da jetzt im Hintergrund Code erzeugt wird, können Sie die aufpoppende Sicherheitswarnung mit *OK* bestätigen und auch die weitere Anzeige der Meldung unterdrücken:

Sicherheitswarnung

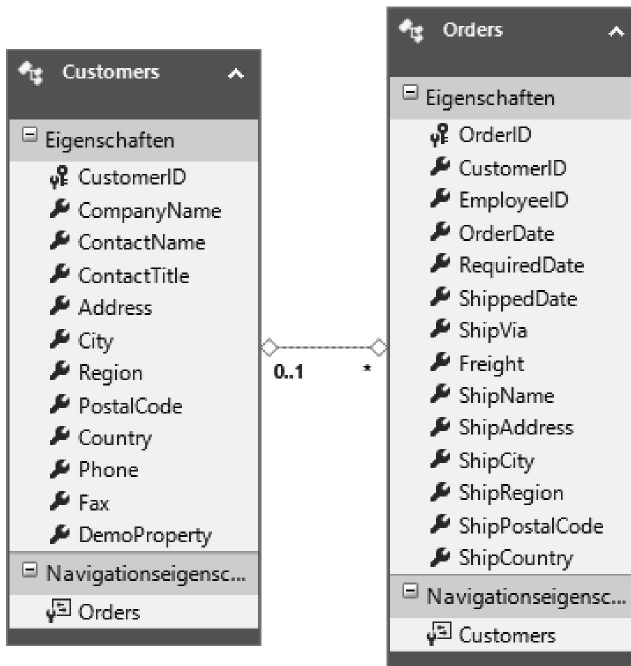
Das Ausführen dieser Textvorlage stellt eine potenzielle Gefahr für Ihren Computer dar. Führen Sie sie nicht aus, wenn sie aus einer nicht vertrauenswürdigen Quelle stammt.

Klicken Sie auf "OK" , um die Vorlage auszuführen.
Klicken Sie auf "Abbrechen" , um den Prozess zu beenden.

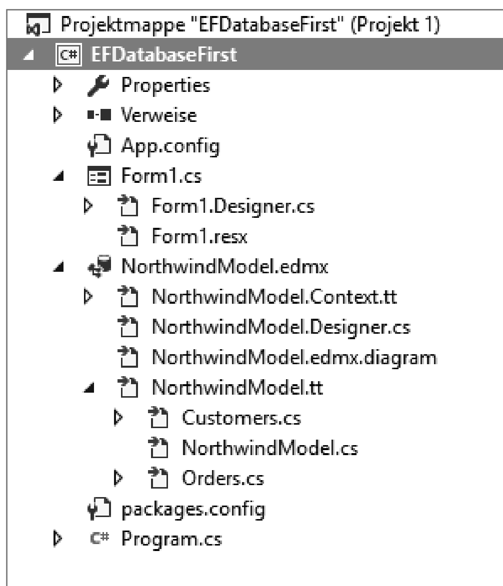
- [x] Diese Meldung nicht mehr anzeigen

OK | Abbrechen

Das Ergebnis wird dann in einer relationalen Ansicht dargestellt:



Tatsächlich wurde jedoch eine sogenannte *Northwind.edmx*-Datei erzeugt und wenn man diese im Projektmappenexplorer aufklappt, kann man auch die zugehörigen erzeugten Klassen sehen:



Schauen wir uns das Ergebnis mal etwas genauer an.

Die übergeordneten Elemente, die Sie sehen, sind tt-Dateien. tt (in der Literatur liest man auch öfter T4) sind sogenannte TextTransformationToolkitTemplate-Dateien. Darin ist die Definition enthalten, wie die Codedateien zu generieren sind. Sie können diese Dateien, natürlich mit ein bisschen Hintergrundwissen über T4) auch editieren, wenn Sie ein paar Sachen anders machen möchten.

Ein direktes Manipulieren der Codedateien ist leider nicht möglich (möglich schon, aber nicht sinnvoll), da beim nächsten Generieren alle Ihre Änderungen überschrieben werden.

So betrachten wir uns nun die erzeugten Codedateien.

Beispiel 14.37: Generierte Codedatei NorthwindModel.Context.cs

C#

```
using System.Data.Entity;
using System.Data.Entity.Infrastructure;

public partial class NorthwindEntities : DbContext
{
    public NorthwindEntities()
        : base("name=NorthwindEntities")
    {
    }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        throw new UnintentionalCodeFirstException();
    }

    public virtual DbSet<Customers> Customers { get; set; }
    public virtual DbSet<Orders> Orders { get; set; }
}
```

Die Klasse *NorthwindEntities* erbt dabei von der Klasse *DbContext*. Über den Konstruktor wird der *ConnectionString* aus der Konfigurationsdatei ausgelesen (der Assistent hat für uns einen ConnectionString mit dem Namen *NorthwindEntities* in die *app.config* eingetragen).

Zu guter Letzt wurden noch die zwei Properties *Customers* und *Orders* als generische *DbSet*s definiert, über die wir auf die Tabellen zugreifen können.

Beispiel 14.38: Generierte Codedatei Customers.cs.

C#

```
using System;
using System.Collections.Generic;

    public partial class Customers
    {
        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
                "CA2214:DoNotCallOverridableMethodsInConstructors")]
        public Customers()
        {
            this.Orders = new HashSet<Orders>();
```

```
        }

        public string CustomerID { get; set; }
        public string CompanyName { get; set; }
        public string ContactName { get; set; }
        public string ContactTitle { get; set; }
        public string Address { get; set; }
        public string City { get; set; }
        public string Region { get; set; }
        public string PostalCode { get; set; }
        public string Country { get; set; }
        public string Phone { get; set; }
        public string Fax { get; set; }

        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
                             "CA2227:CollectionPropertiesShouldBeReadOnly")]
        public virtual ICollection<Orders> Orders { get; set; }
    }
```

In der Klasse *Customers* sind nur noch Properties definiert. Die Namen und Datentypen passen dabei zu den Spalten der Tabelle *Customers* in der *Northwind*-Datenbank.

Einzig die Property *Orders* ist als eine zusätzliche Liste definiert (und bildet somit die Fremdschlüsselbeziehung zwischen *Customers* und *Orders* ab). Durch die Definition als *virtual* wird übrigens für diese Liste *LazyLoading* aktiviert.

Jetzt stellt sich noch die Frage, wie man mit diesen Objekten arbeitet. Dazu wollen wir die Daten in einem *DataGrid* anzeigen und auch editieren lassen (auch wenn ich kein Freund davon bin, Daten in einem Grid zu editieren). Mit einem Speichern-Button sollen die Änderungen dann in die Datenbank zurückgeschrieben werden.

Beispiel 14.39: Lesen und Speichern der Daten

C#

```
public partial class Form1 : Form
{
    private NorthwindEntities context;
    public Form1()
    {
       InitializeComponent();
    }

    private void Form1_Load(object sender, EventArgs e)
    {
       context = new NorthwindEntities();
       dataGridView1.DataSource = context.Customers.ToList();
    }

    private void btnSpeichern_Click(object sender, EventArgs e)
    {
        context.SaveChanges();
    }
}
```

Innerhalb des Formulars definieren wir eine Variable für unseren Context. Dieser wird im Konstruktor instanziiert und mit der Anweisung *context.Customers.ToList()* werden die Daten aus der Datenbank geladen. Die geladenen Daten werden als Datenquelle der *DataGridView* zugewiesen.

Unter dem Click-Event des Speicherbuttons folgt nur eine Anweisung: *context.SaveChanges()*. Das Entity Framework protokolliert alle Ihre Änderungen und schickt die entsprechenden Anweisungen als SQL-Befehle innerhalb einer Transaktion an die Datenbank.

Sie müssen natürlich nicht alle Daten aus der Tabelle laden, mit einer *Linq*-Abfrage können Sie die Daten selbstverständlich auch filtern.

Wenn Sie zum Beispiel nur Kunden aus Deutschland haben wollen, dann führen Sie folgende Abfrage aus:

Beispiel 14.40: Laden von ausschließlich deutschen Kunden

C#

```
dataGridView1.DataSource =
    context.Customers.Where(c=>c.Country=="Germany").ToList();
```

14.8.3 CodeFirst

Bei *CodeFirst* ist der einzige Unterschied die Erstellung des Modells. Der Einsatz der Klassen und der Datenzugriff sind danach anlog wie bei *DatabaseFirst*.

Bei *CodeFirst* definieren Sie einfach Ihre Entitätsklassen und die Contextklasse, fertig. Ein *edmx*-Modell gibt es bei *CodeFirst* nicht, was die Wartbarkeit erheblich erhöht.

Zuerst müssen wir aber unserem Projekt das Entity Framework NuGet-Paket hinzufügen.

Im zweiten Schritt definieren wir (der Einfachheit halber) nur eine Entitätsklasse mit einigen Properties.

Beispiel 14.41: Entitätsklasse Person.cs

C#

```
[Table("Personen")]
public class Person
{
    public int Id { get; set; }
    [Required]
    public string Name { get; set; }
    public DateTime Geburtstag { get; set; }
    [StringLength(20)]
    public string Telefon { get; set; }
    public int? Groesse { get; set; }
}
```

In dieser Klasse wird eine Property *Id* vom Datentyp *int* definiert. Es ist eine Konvention, dass eine Property *Id* automatisch der Primärschlüssel dieser Tabelle wird. Da es sich beim Datentyp um einen Integer handelt, bedeutet dies auch, dass es eine *Identity*-Spalte ist.

Die Property *Name* wird mit dem Attribut *Required* dekoriert. Dies bedeutet, dass es sich um eine *NotNull*-Spalte handelt. Über Attribute wird einiges gesteuert (mehr dazu in der nachfolgenden Tabelle). Die Property *Telefon* erhält das Attribut *StringLength*, um die maximale Länge des Strings in der Datenbank zu definieren. Die Eigenschaft *Groesse* letztendlich ist als *int?* definiert, somit wird diese Spalte in der Datenbank als *null*-Spalte angelegt.

Attribute aus System.ComponentModel.DataAnnotations

In der folgenden Tabelle sind die wichtigsten Attribute zur Definition bei CodeFirst aufgeführt:

Attribut	Bedeutung
Key	Spalte wird als Primärschlüssel definiert
Table	Name der Tabelle, wenn nicht angegeben identisch mit Klassennamen
Column	Name der Spalte, wenn nicht angegeben identisch mit Propertyname; damit auch Angabe von alternativen Datentypen möglich
Index	Definition eines Index, auch mehrspaltig möglich
Required	Bei Strings *NotNull*-Spalte
DatabaseGenerated	Angabe wie der Primärschlüssel definiert wird (*Identity*, *Computed* oder *None*)
NotMapped	Property wird nicht der Datenbanktabelle hinzugefügt

In dieser Art und Weise können Sie jetzt beliebige weitere Entitätsklassen zu Ihrem Projekt hinzufügen.

Dann fehlt uns nur noch die Definition unserer Contextklasse.

Beispiel 14.42: Contextklasse PersonContext.cs

C#

```
public class PersonContext : DbContext
{
    public PersonContext():base("name=PersonContext")
    {
    }
    public virtual DbSet<Person> Personen { get; set; }
}
```

In dieser Klasse definieren wir, welche Klassen dem Context hinzugefügt werden (wir haben ja nur eine). *Personen* wird dabei als generisches *DbSet* vom Typ *Person* definiert.

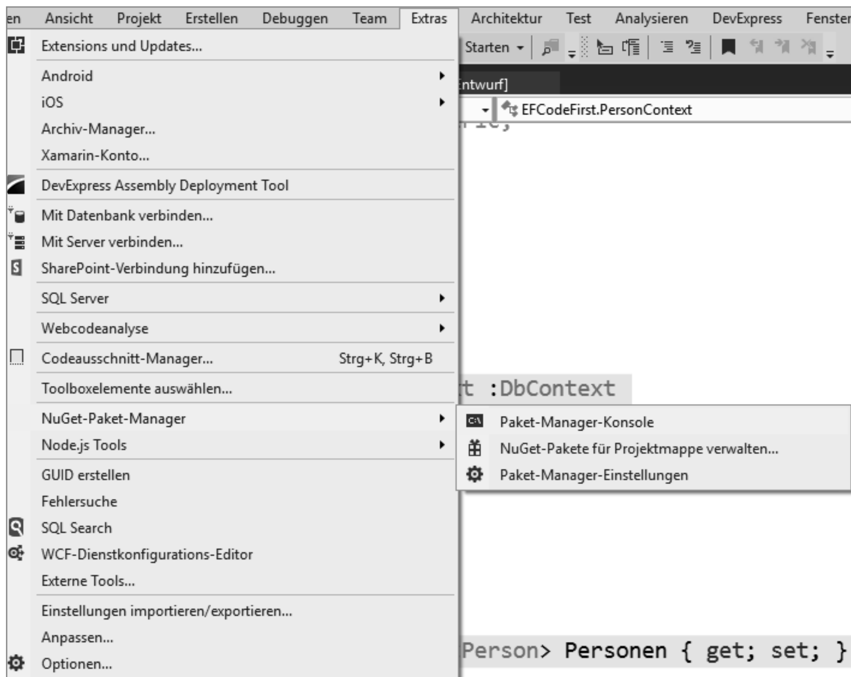
Über den Konstruktor definieren wir, dass der *Connectionstring* in der Konfigurationsdatei steht und *PersonContext* heißt.

Fügen Sie also der Datei *app.config* folgenden Abschnitt hinzu:

```
<connectionStrings>
    <add name="PersonContext" connectionString="data source=.\SqlExpress;  initial
catalog=PersonenDb;integrated security=True; MultipleActiveResultSets=True;
App=Entity Framework" providerName="System.Data.SqlClient" />
  </connectionStrings>
```

Somit ist unser Modell fertig. Nun müssen wir nur noch mittels Migrationen die Datenbank anlegen und später auch aktualisieren.

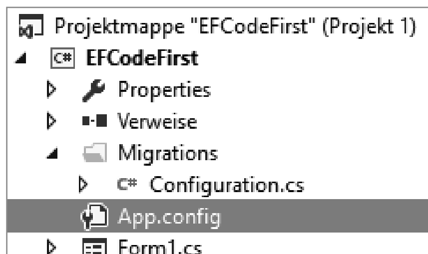
Öffnen Sie dafür die Konsole des Paketmanagers mittels dem Menübefehl *Extras/NuGet-Paket-Manager/Paket-Manager-Konsole*.



Geben Sie anschließend in der Konsole den Befehl *enable-migrations* ein.

```
Paket-Manager-Konsole
Paketquelle: Alle    Standardprojekt: EFCodeFirst
PM> enable-migrations
Checking if the context targets an existing database...
Code First Migrations enabled for project EFCodeFirst.
PM>
```

Durch die Ausführung dieses Befehls wird im Projekt ein Ordner *Migrations* mit einer Klasse *Configuration.cs* angelegt. Innerhalb dieser Klasse gibt es eine *Seed*-Methode, in der Sie zum Beispiel Testdaten erzeugen können, die nach jeder Migration in die Datenbank geschrieben werden.



Um nun die erste Migration anzulegen, geben Sie in der Konsole den Befehl *add-migration InitialMigration* ein.

```
PM> add-migration InitialMigration
Scaffolding migration 'InitialMigration'.
The Designer Code for this migration file includes a snapshot of your current Code First model. This snapshot is used to calculate the changes
 to your model when you scaffold the next migration. If you make additional changes to your model that you want to include in this migration,
then you can re-scaffold it by running 'Add-Migration InitialMigration' again.
PM> |
```

Dadurch wird automatisch eine neue Migrationsklasse angelegt. Der Name der Klasse hat noch einen führenden Zeitstempel.

Beispiel 14.43: Migrationsklasse InitialMigration.cs.

C#

```
using System.Data.Entity.Migrations;

public partial class InitialMigration : DbMigration
{
    public override void Up()
    {
        CreateTable(
            "dbo.Personen",
            c => new
                {
                    Id = c.Int(nullable: false, identity: true),
                    Name = c.String(nullable: false),
```

```
                    Geburtstag = c.DateTime(nullable: false),
                    Telefon = c.String(maxLength: 20),
                    Groesse = c.Int(),
                })
            .PrimaryKey(t => t.Id);
    }
    public override void Down()
    {
        DropTable("dbo.Personen");
    }
}
```

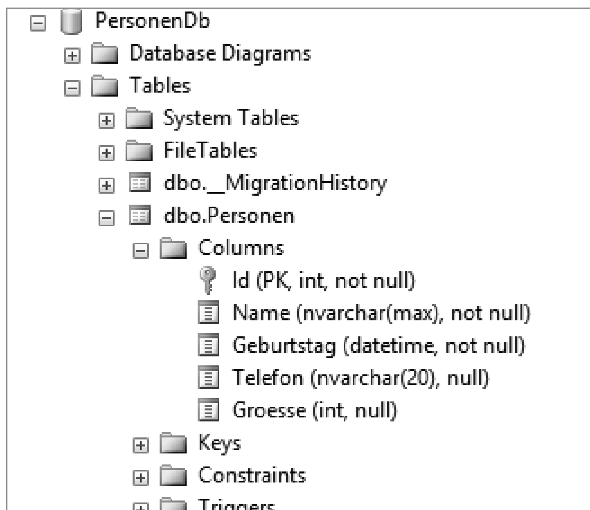
Die Routine *Up()* enthält den Code, der auf der Datenbank ausgeführt werden soll. *Down()* wird für einen Downgrade benötigt.

Um die Migration jetzt durchzuführen, geben Sie in der Konsole den Befehl *update-database* an.

```
PM> update-database
Specify the '-Verbose' flag to view the SQL statements being applied to the target database.
Applying explicit migrations: [201708152118030_InitialMigration].
Applying explicit migration: 201708152118030_InitialMigration.
Running Seed method.
```

Sie können dies natürlich auch per Programmcode machen, sodass zum Beispiel bei jedem Programmstart die Datenbank aktualisiert wird. Sehr vorteilhaft bei lokalen Datenbanken, bei Serveranwendungen vielleicht nicht so optimal.

Nachdem der Befehl erfolgreich ausgeführt wurde, haben Sie eine neue Datenbank angelegt.



Die Tabelle *Personen* ist genauso angelegt worden, wie wir es definiert haben. Zusätzlich gibt es noch eine Tabelle *__MigrationHistory*. Über diese Tabelle weiß die Datenbank, welche Migrationen schon ausgeführt wurden, und führt diese somit nicht doppelt aus.

Alle weiteren Änderungen am Objektmodell werden analog mit zusätzlichen Migrationen erstellt.

Auch der Zugriff funktioniert analog, wie im DatabaseFirst-Beispiel gezeigt.

Dieses Kapitel soll nur einen ersten Überblick über das Entity Framework geben. Für detaillierte Informationen empfehlen wir die Lektüre von Büchern über das Entity Framework.

14.9 Praxisbeispiele

14.9.1 Wichtige ADO.NET-Objekte im Einsatz

Wer sich nicht nur blind auf die Hilfe von Assistenten verlassen möchte, sollte sich in der ADO.NET-Objekthierarchie ein wenig auskennen, damit er die Objekte bei Bedarf selbst per Code programmieren kann.

Die *Columns*- und *Rows*-Auflistungen zählen zu den wichtigsten Eigenschaften der *DataTable*-Klasse, weil sie den Zugriff auf sämtliche Spalten und Zeilen der Tabelle ermöglichen. Das vorliegende Beispiel soll das Zugriffsprinzip verdeutlichen, indem es uns den Inhalt der *Products*-Tabelle der *Northwind*-Datenbank anzeigt.

Oberfläche

Sie brauchen lediglich eine *ListBox* und einen *Button* zum Beenden (siehe Laufzeitansicht).

Quellcode

```
using System.Data.SqlClient;
...
```

Alles beginnt mit der Festlegung der Verbindungszeichenfolge (*ConnectionString*) zur *Northwind*-Datenbank.

```
    private void Form1_Load(object sender, System.EventArgs e)
    {
        SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                     Initial Catalog=Northwind;Integrated Security=true");
        string cmdStr =
           "SELECT ProductId  As ArtikelNr, ProductName as Artikelname,
            QuantityPerUnit As Liefereinheit, UnitPrice As Einzelpreis,
            ReorderLevel As Mindestbestand FROM Products";
        SqlCommand cmd = new SqlCommand(cmdStr, conn);
```

HINWEIS: Im obigen Select (sowie auch in allen folgenden) wurden Alias-Namen für die Spalten vergeben, um bei den Beschriftungen deutschsprachige Spaltennamen auszugeben.

Nun geht es um das Füllen des *DataSets* mithilfe des *DataAdapter*:

```
SqlDataAdapter da = new SqlDataAdapter(cmd);
DataSet ds = new DataSet();
conn.Open();
da.Fill(ds, "ArtikelListe");
conn.Close();
```

Die Datenbankverbindung ist ab jetzt wieder getrennt und der Benutzer arbeitet mit dem abgekoppelten *DataSet* quasi wie mit einer Minidatenbank:

```
DataTable dt = ds.Tables[«ArtikelListe»];
```

Nachdem je eine Zeilen- und Spaltenvariable definiert sind, sorgen zwei ineinander verschachtelte *foreach*-Schleifen für den Durchlauf der Auflistungen:

```
        foreach (DataRow cRow in dt.Rows)
        {
            foreach (DataColumn cCol in dt.Columns)
                listBox1.Items.Add($"{cCol.ColumnName} =  {cRow[cCol.Ordinal]}");
           listBox1.Items.Add("-----------------------------------------------");
        }
    }
```

Test

Sofort nach Programmstart erscheint der Inhalt der *Artikel*-Tabelle in der *ListBox*.

Bemerkungen

- Am Quellcode können Sie den typischen Ablauf eines Datenbankzugriffs studieren: Verbindung öffnen, Daten übertragen, Verbindung schließen.
- Beim Durchlaufen der Datensätze werden Sie die vom altvertrauten ADO-*Recordset*-Objekt her bekannten Methoden wie *MoveFirst*, *MoveNext* etc. vergeblich suchen. Dafür besteht unter ADO.NET keinerlei Notwendigkeit mehr, da alle Datensätze im *DataSet* quasi wie in einem Array gespeichert sind und ein sofortiger (indizierter) Zugriff möglich ist, ohne dass man sich erst mühsam „hinbewegen“ muss.

14.9.2 Eine Aktionsabfrage ausführen

Wir wollen an die *Northwind*-Beispieldatenbank folgendes SQL-Statement absetzen:

```
UPDATE Customers SET CompanyName = 'Londoner Firma' WHERE City = 'London'
```

Das vorliegende Beispiel zeigt, wie Sie dazu die *ExecuteNonQuery*-Methode des *Command*-Objekts verwenden können.

Oberfläche

Sie brauchen ein *DataGridView*, zwei *TextBox*en, zwei *Buttons* und einige *Labels* (siehe Laufzeitansicht). Beide *TextBox*en sollen dazu dienen, dass Sie die Einträge für den Firmennamen und den Ort zur Laufzeit verändern können.

Quellcode

Für das Ausführen des Beispiels wären eigentlich ein *Connection*- und ein *Command*-Objekt völlig ausreichend. Da wir uns aber auch von der Wirkung des UPDATE-Befehls überzeugen wollen, müssen wir einigen zusätzlichen Aufwand für die Anzeige betreiben: Das *DataGridView* benötigt ein *DataSet* als Datenquelle, welches wiederum von einem *DataAdapter* gefüllt wird.

```
using System.Data.SqlClient;
public partial class Form1 : Form
{
    SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                     Initial Catalog=Northwind;Integrated Security=true");
    DataSet ds = new DataSet();
    SqlCommand cmd = new SqlCommand();
```

Aktionsabfrage starten:

```
    private void button1_Click(object sender, System.EventArgs e)
    {
        SqlDataAdapter da = new SqlDataAdapter("SELECT CompanyName As Firma,
                        ContactName As Kontaktperson, City As Ort
                        FROM Customers ORDER BY CompanyName", conn);
        ds.Clear();
        cmd.Connection = conn;
```

Das Zusammenbasteln des UPDATE-Strings verlangt etwas Fingerspitzengefühl, darf man doch auch die Apostrophe ('), die die Feldbezeichner einschließen, nicht vergessen:

```
        cmd.CommandText = $"UPDATE Customers SET CompanyName = '{textBox1.Text}'
                            WHERE City = '{textBox2.Text}'";
```

Sicherheitshalber haben wir diesmal den kritischen Programmteil in eine Fehlerbehandlungsroutine eingebaut:

```
        try
        {
            conn.Open();
```

Die folgende Anweisung führt den UPDATE-Befehl aus und zeigt gleichzeitig die Anzahl der in der Datenbank geänderten Datensätze an:

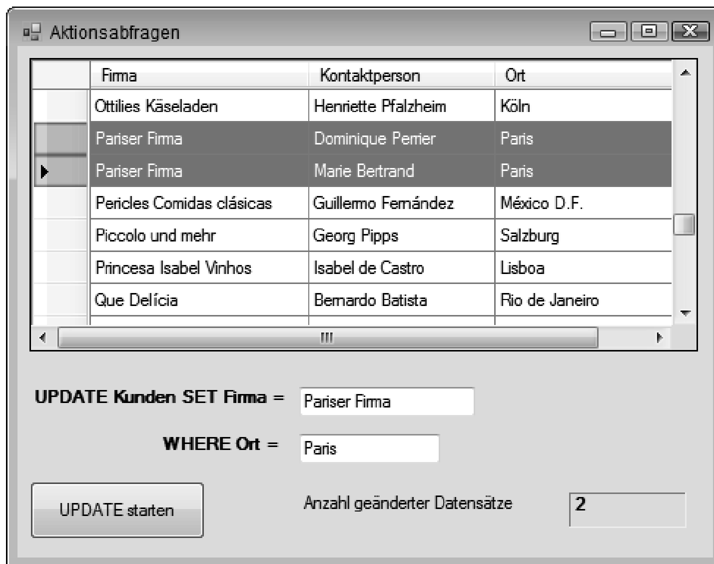
```
            label1.Text = cmd.ExecuteNonQuery().ToString();
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
        da.Fill(ds, "Kunden");
        conn.Close();
```

Das *DataGridView* an das *DataSet* anklemmen:

```
        dataGridView1.DataSource = ds;
        dataGridView1.DataMember = "Kunden";
    }
}
```

Test

Stimmt die Verbindungszeichenfolge des *Connection*-Objekts, dürfte es keine Probleme beim Ausprobieren unterschiedlicher Updates geben.

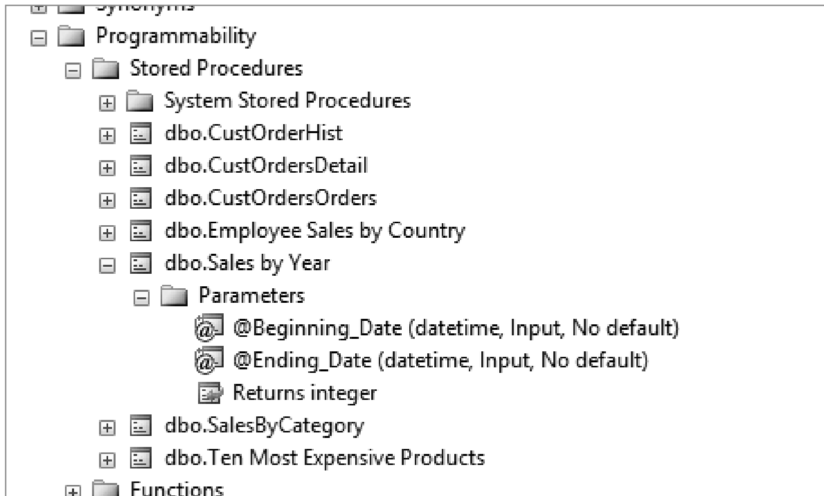


Bemerkungen

- Bei SQL-Aktionsabfragen werden keine Datensätze gelesen bzw. zurückgeliefert, sondern es geht lediglich um direkte Änderungen in der Datenquelle per SQL-Befehl (UPDATE, INSERT, DELETE). Ein *DataSet* ist dabei nicht beteiligt!
- Wie Sie die Änderungen zuerst in einer *DataTable* vornehmen und erst danach in die Datenbank zurückschreiben, erfahren Sie im Praxisbeispiel in Abschnitt 14.9.4, Die Datenbank aktualisieren.

14.9.3 Eine StoredProcedure aufrufen

Öffnen Sie im SqlServer Managment Studio innerhalb der *Northwind*-Datenbank den Knoten *Programmability* und Sie sehen alle vordefinierten StoredProcedures der Datenbank, die Sie natürlich auch selbst um weitere ergänzen können:



Hinter jeder Auswahlabfrage verbirgt sich in der Regel eine parametrisierte SQL-SELECT-Anweisung, die Sie sich im Management-Studio anzeigen lassen können. Dabei finden Sie auch die zu übergebenden Parameter und deren Datentypen leicht heraus:

Oberfläche

Ein *DataGridView*, zwei *TextBox*en und ein *Button* sollen für unseren Test genügen (siehe Laufzeitansicht am Schluss).

Quellcode

```
using System.Data.SqlClient;
public partial class Form1 : Form
{
...
```

```
private void button1_Click(object sender, System.EventArgs e)
{
    SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                  Initial Catalog=Northwind;Integrated Security=true");

    SqlCommand cmd = new SqlCommand("[Sales By Year]", conn);
    cmd.CommandType = CommandType.StoredProcedure;
```

Die Definition der beiden Parameter und das Hinzufügen zur *Parameters*-Auflistung des *Command*-Objekts:

```
    SqlParameter parm1 = new SqlParameter("@Beginning_Date", SqlDbType.Date);
    parm1.Direction = ParameterDirection.Input;
    parm1.Value = Convert.ToDateTime(textBox1.Text);
    cmd.Parameters.Add(parm1);

    SqlParameter parm2 = new SqlParameter("@Ending_Date", SqlDbType.Date);
    parm2.Direction = ParameterDirection.Input;
    parm2.Value = Convert.ToDateTime(textBox2.Text);
    cmd.Parameters.Add(parm2);
```

Das *Command*-Objekt wird dem Konstruktor des *DataAdapter*s übergeben. Nach dem Öffnen der *Connection* wird die Abfrage ausgeführt. Die zurückgegebenen Datensätze werden in einer im *DataSet* neu angelegten Tabelle mit einem von uns frei bestimmten Namen *Jahresumsätze* gespeichert:

```
    OleDbDataAdapter da = new OleDbDataAdapter(cmd);
    DataSet ds = new DataSet();
    try
    {
        conn.Open();
        da.Fill(ds, "Jahresumsätze");
        conn.Close();
    }
    catch(Exception ex)
    {
        MessageBox.Show(ex.ToString());
    }
```

Die Anzeige:

```
    dataGridView1.DataSource = ds;
    dataGridView1.DataMember = "Jahresumsätze";
```

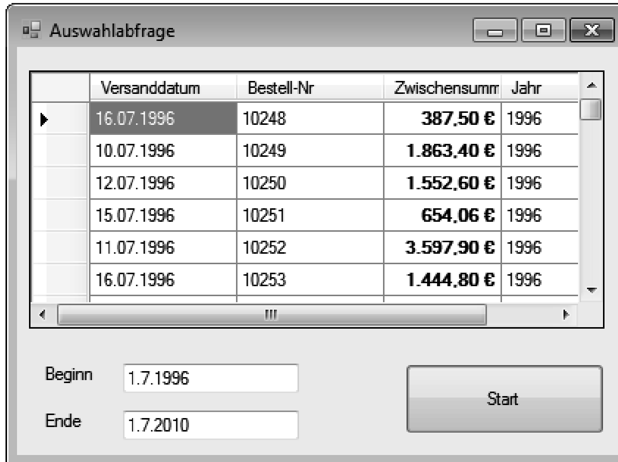
Wenigstens die Währungsspalte sollte eine ordentliche Formatierung erhalten (bei den übrigen Spalten belassen wir es bei den Standardeinstellungen):

```
    dataGridView1.Columns.Remove("Subtotal");
    DataGridViewTextBoxColumn tbc = new DataGridViewTextBoxColumn();
    tbc.DataPropertyName = "Subtotal";
    tbc.HeaderText = "Zwischensumme";
    tbc.Width = 80;
    tbc.DefaultCellStyle.Format = "c";
    tbc.DefaultCellStyle.Alignment = DataGridViewContentAlignment.MiddleRight;
    tbc.DefaultCellStyle.Font = new Font(dataGridView1.Font, FontStyle.Bold);
    tbc.DisplayIndex = 2;
```

```
        dataGridView1.Columns.Add(tbc);
    }
}
```

Test

Nach Eingabe sinnvoller Datumswerte dürfte sich Ihnen der folgende Anblick bieten:



14.9.4 Die Datenbank aktualisieren

Aktualisieren (UPDATE), Hinzufügen (INSERT) und Löschen (DELETE) von Datensätzen zählen zu den kritischen Datenbankoperationen, die auch unter ADO.NET weitaus mehr Aufmerksamkeit erfordern als eine einfache SELECT-Abfrage.

Ganz im Einklang mit der ADO.NET-Philosophie müssen wir dabei in drei Etappen vorgehen:

- Das *DataSet* mit der Datenbank verbinden, um bestimmte Datensätze von dort abzuholen (hierzu wird das *SelectCommand*-Objekt des *DataAdapter*s eingesetzt).
- Bei abgekoppelter Datenbank die Änderungen direkt im *DataSet* vornehmen (hierzu ist eine SQL-Anweisung leider untauglich, da das *DataSet* kein SQL kennt).
- Das *DataSet* irgendwann wieder mit der Datenbank verbinden, um die Inhalte zu aktualisieren (hierzu werden *UpdateCommand*-, *InsertCommand*- und *DeleteCommand*-Objekt des *DataAdapter*s gebraucht).

Wir wollen das am Beispiel der *Products*-Tabelle aus der Datenbank *Northwind.mdb* demonstrieren.

Oberfläche

Neben zwei *Button*s zum Anzeigen und Aktualisieren brauchen wir noch eine *DataGridView*-Komponente (siehe Laufzeitabbildung).

Quellcode (Command-Objekte selbst programmiert)

```
using System.Data.SqlClient;

public partial class Form1 : Form
{
    //Die wichtigsten Objekte sollten global verfügbar sein:
    private SqlConnection conn = new SqlConnection("Data Source=.\\SqlExpress;
                     Initial Catalog=Northwind;Integrated Security=true");
    private SqlDataAdapter da = null;
    private DataSet ds = null;
    ...
```

Die folgende Methode *getArtikel* liefert ein gefülltes *DataSet* zurück:

```
    public DataSet getArtikel()
    {
```

SelectCommand-Objekt für *DataAdapter* erstellen (geschieht automatisch beim Instanziieren):

```
        string selStr = "SELECT ProductId As ArtikelNr, ProductName As Artikelname,
                         UnitPrice As Einzelpreis, ReorderLevel As Mindestbestand
                         FROM Products ORDER BY Productname";
        da = new SqlDataAdapter(selStr, conn);
```

Die folgende Anweisung sorgt dafür, dass neu hinzugefügte Datensätze sofort einen Primärschlüssel erhalten.

```
        da.MissingSchemaAction = MissingSchemaAction.AddWithKey;
        conn.Open();
        DataSet ds = new DataSet();
        da.Fill(ds, "Artikel");
        conn.Close();
        return ds;
    }
```

Der Methode *setArtikel* wird ein gefülltes *DataSet* per Referenz übergeben. Mithilfe eines *CommandBuilder*s werden im Hintergrund für den *DataAdapter* die *UpdateCommand*-, *InsertCommand*- und *DeleteCommand*-Objekte erstellt, die für das Zurückschreiben der im *DataSet* vorgenommenen Änderungen in die Datenbank verantwortlich zeichnen.

```
    public void setArtikel(ref DataSet ds)
    {
        SqlCommandBuilder cb = new SqlCommandBuilder(da);

        conn.Open();
        da.Update(ds, "Artikel");
        conn.Close();
    }
```

Anzeigen:

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

```
        dataGridView1.DataSource = null;
        ds = getArtikel();
        dataGridView1.DataSource = ds;
        // DataGridView mit DataSet verbinden
        dataGridView1.DataMember = "Artikel";
        formatDataGridView(dataGridView1);
    }
```

Aktualisieren:

```
    private void button2_Click(object sender, EventArgs e)
    {
```

Nur die Änderungen in die Datenbank zurückschreiben:

```
        DataSet ds1 = ds.GetChanges();
        if (ds1 != null)
        {
            try
            {
                setArtikel(ref ds1);
```

Die per Referenz zurückgegebenen Datensätze werden mit dem Original-*DataSet* zusammengeführt:

```
                ds.Merge(ds1);
                ds.AcceptChanges();
                MessageBox.Show("Datenbank wurde aktualisiert!", "Erfolg");
            }
            catch (Exception ex)
            {
                ds.RejectChanges();
                MessageBox.Show(ex.Message, "Fehler");
            }
        }
    }
}
```

Test

Klicken Sie auf die „Artikel anzeigen“-Schaltfläche, um das *DataSet* anzuzeigen. Nehmen Sie dann einige Änderungen direkt im *DataGridView* vor, fügen Sie Datensätze hinzu (dazu scrollen Sie an das Ende des *DataGridViews*) oder löschen Sie Datensätze (mit der *Entf*-Taste, nachdem Sie die komplette Zeile markiert haben). Klicken Sie auf „Artikel aktualisieren“, um die Änderungen in die Datenbank zu übertragen. Lassen Sie sich danach erneut die Artikel anzeigen, um sich davon zu überzeugen, dass alle Änderungen tatsächlich in der Datenbank angekommen sind.

Bemerkungen

- Der Code zur Formatierung der *Einzelpreis*-Spalte des *DataGridView*s wurde hier nicht mit abgedruckt (siehe Beispieldaten).
- Es ist auch möglich, mehrere Datensätze hintereinander zu ändern, hinzuzufügen bzw. zu löschen, bevor der Abgleich mit der Datenbank erfolgt.
- In der Regel werden Sie nur die von Ihnen selbst hinzugefügten Datensätze löschen können, da die originalen Datensätze in Relationen zu anderen Tabellen eingebunden sind.

Index

Symbole

.NET-Framework 20
??-Operator 54

A

Abbruchbedingung 604
Abort 488
Abs 263
abstract 161 f.
Abstraktion 18
Access Control Entries 413
Accessor 126
ACE 413
ACL 387, 413
Activator 819, 821
Add 668
AddAccessRule 413
AddAfterSelf 668
AddBeforeSelf 668
AddDays 258
AddExtension 452
AddFirst 668
AddHours 258
AddMinutes 258
AddMonths 258
AddRange 292
AddressList 864
AddressWidth 870
AddYears 258
Administrator 861 f.
ADO.NET-Klassen 705
ADO.NET-Objektmodell 703
Aktionsabfrage 712, 715, 745
Ancestors 665
Anfangswerte 48
Anonyme Methoden 306, 329
Anonyme Typen 339
App.config 611
Append 436
AppendChild 650 f., 655 f.
AppendLine 477
AppendText 437
Application 865
ApplicationServices 872
args 896
Arithmetische Operatoren 64
Array 103, 219, 231, 372, 377
ArrayList 291, 296, 301
as-Operator 57
Assemblierung 9, 24, 32
Assembly
– dynamisch laden 815, 819
– GetExecutingAssembly 816
– Laden 816
– LoadFrom 816
AssemblyInfo 872
AssemblyInfo.cs 894
async 524
Asynchrone Programmierentwurfsmuster 512
Atn 263
Attribute 26, 632, 636, 648
Attributes 410
Auflistung 288
Ausgabefenster 598
Ausnahmen 623
Ausschneiden 811
Aussteuerungsbalken 878
Auswahlabfrage 747
Auto-Property 130
AutoResetEvent 588
AvailablePhysicalMemory 870

AvailableVirtualMemory 870
await 524

B

BackgroundColor 899
BackgroundWorker 510
Barrier 585
base 151
Basisklassen 150
BatteryChargeStatus 871
Beep 486, 900
Befehlsfenster 597
BeginInvoke 505, 520
BigInteger 317
Bild drucken 840
Bild einlesen 837
Bildschirm 868 f., 873
Binärdatei 440
BinaryFormatter 202, 204, 434, 443
BinaryReader 434, 440 f.
BinarySearch 230
BinaryWriter 434, 440 f.
BindingNavigator 454
BindingSource 203, 454
BitsPerPixel 874
BlockingCollection 584
bool 46
Boolesche Operatoren 68
BootMode 866
Bounds 874
Boxing 62
break 76, 102, 440
Breakpoints 601
byte 45
Byte 464
Byte-Array 776

C

Callback 516
CallbackTimer 531, 535
Caller Information 613
CallerFilePath 614
CallerLineNumber 614
CallerMemberName 614
CameraDeviceType 841
CancellationPending 511
CancellationToken 576, 586
CancellationTokenSource 576
case 71, 102
C#-Compiler 5
ChangeClipboardChain 830
ChangeDatabase 710
char 46
CheckAccess 507
CheckFileExists 420, 452
CheckPathExits 420
ChildNodes 653
class 9, 104, 113
ClassesRoot 814
ClassLoader 23
Clear 230
ClickOnce-Deployment 788
Clipboard 809
– ContainsText 810
– GetImage 811
– SetText 810
Clone 230, 239 f., 758
Close 146, 710
CLR 21, 23
CLR-Threadpool 556
CLS 21
Code Contracts 628
Code Manager 23
Codefenster 17
Collection 288, 378
CollectionBase 672
ColumnName 762
Columns 759, 761
COM-Komponenten 25
Command 711, 725
CommandBuilder 720
CommandLine 872
CommandText 713
CommandTimeout 713
CommandType 714
COM-Marshaller 24
Common Language Runtime 18, 23
Common Language Specification 21
Common Type System 21 f.
CompanyName 872
CompareDocumentOrder 665
Complex 319
ComputerName 869
ConcurrentBag 584
ConcurrentDictionary 584
ConcurrentQueue 584
ConcurrentStack 584
Connection 706, 713
ConnectionString 708
ConnectionStringBuilder 711

ConnectionTimeout 709
ConsoleKeyInfo 903
ConsoleModifiers 903
const 53
Constraint 302
continue 74
Convert 60
ConvertStringToByteArray 468
Copy 409, 757
Copyright 872
CopyTo 230, 239 f., 409
Cos 263
CountdownEvent 585
Create 436
CreateCommand 711, 713
CreateDirectory 405
CreateElement 651
CreateEncryptor 446
CreateFromFile 449
CreateGraphics 881
CreateInstance 821
CreateNavigator 676, 697
CreateNew 449
CreateSubdirectory 406
CreateSubKey 815, 822
CreateText 437
CreateViewAccessor 450
CreationTime 410
CRUD 771
CryptoStreams 446, 467
C#-Source-Datei 7
CSV-Datei 475, 478
CTS 21
Current 290
CurrentClockSpeed 870
CurrentConfig 814
CurrentCulture 866
CurrentDirectory 872
CurrentUser 814
CursorLeft 899
CursorTop 899
CursorVisible 899

D

Data Encryption Standard 467
DataAdapter 724
Database 708
DataBindings 206
DataColumn 759
DataGridView 203
DataReader 721
DataSource 708
DataTable 759
DataTime 258
DataView 763, 772, 774
Datei
– komprimieren 470
– kopieren und verschieben 408
– umbenennen 409
– verschlüsseln 467
Dateiattribute 409
Dateiname 480
Dateiparameter 435
Dateiverknüpfungen 824
Datenkonsument 701
Datenprovider 701, 703
Datenquelle 767
Datenstrukturen 584
Datentypen 45, 101
Datenzugriff 80
DateTime 258
Datumsformatierung 267
Datumsfunktionen 256
Day 258
DayOfWeek 258
DayOfYear 258
DaysInMonth 259
DbProviderFactories 704
Deadlocks 483
Debug 606
– Write 607
– WriteIf 607
– WriteLineIf 607
Debugger 595
decimal 46
Decrypt 444
default 74
DefaultExt 451
Dekrement 65
delegate 105, 134
Delegate 303, 329
– instanziieren 306
Delete 406, 763
DeleteCommand 725
DeleteSubKey 815
DeleteSubKeyTree 815, 824
DeleteValue 815
Deployment 789
Depth 678
DereferenceLinks 420
DES 467

Descendants 665
Description 872
DESCryptoServiceProvider 446, 468
Deserialize 205
Designer 16
DesktopDirectory 421
Destruktor 140, 144
DeviceManager 834
DeviceName 874
Dezimalzahlen 463
Diagnostics 486
DialogResult 452
Dictionary 301, 898
Dimensionsgrenzen 225
Direction 719
Directory 400, 404
DirectoryInfo 400
DirectoryName 404
DirectorySecurity 413
Dispose 717
Distinct 369
do 74, 76
DOCTYPE 636
Document Object Model 647
Document Type 648
DOM 647
double 46
DriveInfo 400
Duplikate 369
Dynamische Programmierung 312
DynData 814

E

Eigenschaften 11, 122
Eigenschaften-Fenster 17
Eigenschaftsmethoden 213
Einfügen 811
Einzelschritt-Modus 605
Element 632, 636, 648, 695
Elements 666
else 102
else if 71
EnableRaisingEvents 416
Encrypt 444
EndElement 695
EndInvoke 505, 520
EndsWith 233
Enter 496
Entwicklungsumgebung 13
enum 77, 103
Enumerable 372, 376
Enumerationen 103
Environment 897
Ereignis 11, 105, 134
– auslösen 136
Erweiterungsmethoden 340, 360
event 105, 134
EventLog 613
EventLogTraceListener 612
Events 11
Exception 624
ExceptionManager 23
ExecutablePath 872
ExecuteNonQuery 712, 715
ExecuteReader 712, 716, 721
ExecuteScalar 712, 716
Exists 410
Exit 496
Exp 263
ExpandAll 692
Exponentialfunktion 264
ExtClock 870
Extension 404
Extension-Method-Syntax 343, 360

F

Fehlerbehandlung 615
Fehlerklassen 616, 625
FieldCount 723
File 400, 434
FileAccess 435
FileDropList 809
FileExtension 838
FileInfo 400, 434, 437
FileMode 436
FileName 420, 452
FileSecurity 413
FileShare 436
FileStream 204, 434
FileSystemAccessRule 413
FileSystemWatcher 400, 416
Fill 726
Filter 416, 452
FilterIndex 420
Filtern 764
Filters 420
Find 762
FirstChild 652, 688, 696
float 46
FolderBrowserDialog 421

FontFamily 867
FontSmoothingContrast 867
FontSmoothingType 867
Fonts 421
for 74 f., 102
for-each 683
foreach 103, 159, 223, 301
ForegroundColor 899
Form1.cs 16
Format 268
Formatters 204
Formulare 11
FromCurrentSynchronizationContext 580, 593
FullName 404
Funktionen 104

G

Garbage Collector 144
Generics 296, 298
Generische Schnittstelle 371
Geräteeigenschaften 836
get 104, 126
GetChanges 758
GetCommandLineArgs 897
GetCreationTime 410
GetCurrent 863
GetCurrentDirectory 406, 876
GetDataObject 809
GetDataPresent 810
GetDirectories 407
GetElementsByTagName 657
GetEnumerator 290, 300
GetEnvironmentVariables 869, 898
GetExecutingAssembly 872
GetFactoryClasses 704
GetFields 817
GetFiles 411
GetHostEntry 864 f.
GetHostName 864
GetLength 230, 240
GetManifestResourceNames 827
GetManifestResourceStream 828
GetMembers 817
GetMethod 821
GetMethods 817
GetProcessById 546
GetProcesses 545
GetProperties 817
GetShortPathName 875
GetSubKeyNames 815
Getter-only Auto-Property 130
GetTypes 816
GetValue 723, 823
GetValueNames 815
GetValues 723
Gigabyte 464
goto 74
Grafikbearbeitung 842
Gruppen 858
GZipStream 470

H

Haltepunkte 603
Hashtable 293
HasValue 54
Hauptprogramm 894
Hexadezimal 463
Hour 258
HTML 629

I

IAsyncResult 515 ff., 521
ICollection 290
IComparable 198
IComparer 198
ICryptoTransform 469
IDataObjekt 810
IDisposable 717
Idle-Prozesse 545
IEnumerable 289, 297, 372, 380
IEnumerator 290
IEqualityComparer 371
if 71, 102
IgnoreWhitespace 694
ImageFile 838, 842
ImageProcess 842
ImportRow 761
Indent 681
IndentChars 681
IndentLevel 608
IndentSize 608
Index 219
Indexer 213, 288, 295, 327
IndexOf 230, 233
Indexprüfung 222
InitialDirectory 452
Initialisierer 372
Initialisierung 115
Initialize 230, 240

Inkrement 65
InnerText 658
Insert 233
InsertCommand 725
InstallShield 797
Instanz 110
Instanziieren 114
int 45
Int16 45
Int32 45
Int64 46
Intellisense 119
internal 112
internal protected 112
InteropServices 884
Interrupt 487
Invoke 504, 518, 520, 819, 821
InvokeRequired 506
IPAddress 865
IP-Adresse 863
IsAbstract 817
IsAdmin 862
IsAfter 666
IsAlive 489
IsBackGround 490
IsBefore 666
IsClass 817
IsClosed 723
IsCOMObject 817
IsCompleted 515, 563
IsEnum 817
IsFontSmoothingEnabled 867
IsInterface 817
IsLeapYear 259
IsPublic 817
IsSealed 817
Item 723
Iterator 300, 563
iTextSharp 473

J

JIT-Compiler 19
Join 477, 488

K

Kapselung 18, 110
Kartenspiel 208
Kartesische Koordinaten 190
Kilobyte 464
Klasse 110
Klassendefinition 104
Kommentare 44, 632
Komplexe Zahlen 190
Komprimieren 447
Konsolenanwendung 90, 893
Konstante Felder 128
Konstanten 45, 53
Konstruktor 140, 544
– überladen 213
Kontextmenü 824
Kontravarianz 316
Konverter 463
Kopieren 811
Kovarianz 316
Kritische Abschnitte 531
Kurz-Operatoren 66
Kurzschlussauswertung 68

L

Lambda Expression 329
Lambda-Ausdruck 308, 360, 366
LastAccessTime 410
LastWriteTime 410
Laufwerke 407
Length 230, 233, 239 f.
LINQ 365, 368, 377 f., 381, 479
– Abfrageoperatoren 344
– Aggregat-Operatoren 353
– AsEnumerable 356
– Count 353
– GroupBy 350
– Grundlagen 337
– Gruppierungsoperator 350
– Join 352
– Konvertierungsmethoden 356
– OrderBy 348
– OrderByDescending 348
– Projektionsoperatoren 346
– Restriktionsoperator 348
– Reverse 350
– Select 346
– SelectMany 346
– Sortierungsoperatoren 348
– Sum 354
– ThenBy 348
– ToArray 356
– ToDictionary 356
– ToList 356
– ToLookup 356

– Where 348
LINQ to XML-API 660
LINQ-Abfrageoperatoren 342
LINQ-Architektur 337
LINQ-Provider 338
LINQ-Syntax 342
List 296, 301
List-Klasse 301
Load 664
LoadedAssemblies 872
LoadXml 649
LocalApplicationData 421
LocalMachine 814
lock 494
Log 263
Log10 263
Logarithmus 264
Logische Operatoren 67
Lokale Variablen 55
Lokal-Fenster 598
long 46
LongRunning 580
LowestBreakIteration 563

M

MachineName 869
Main 895
MainModule 545
ManagementObject 859
ManagementObjectSearcher 859
Manifestressourcen
– Betrachter 826
ManualResetEvent 591
ManualResetEventSlim 585
Manufacturer 870
Map View 449
Matrix 213, 218
Max 263
MCI 875, 877, 883
mciGetErrorString 875, 884
mciSendString 875, 884
Media 486
Megabyte 464
Memory Mapped File 448 ff., 461
MemoryStream 776, 781
MenuStrip 865
Messwertliste 363
Metadaten 25
Metasprache 630
Methoden 11, 81, 104, 130
– generische 299
– überladen 96, 213
Methodenzeiger 303
MethodImpl 501
Methods 11
Microsoft Intermediate Language Code 19
Microsoft.VisualBasic 866
Microsoft.VisualBasic.Devices 866
Mikrofon 877
Mikrofonpegel 878
Min 263, 461
Minute 258
MMF 448
Modifiers 903
Monitor 496
MonitorCount 868
MonitorsSameDisplayFormat 868
Month 258
Move 406, 409
MoveBufferArea 900
MoveNext 290
MoveTo 409
MoveToNext 676, 696
MoveToPrevious 676, 696
MoveToRoot 696
MSIL-Code 19, 32
MultiSelect 420
Multitasking 482
Multithreading 27, 482, 531
Mutex 500
MyComputer 421
MyDocuments 421

N

nameof 458
Namespace 24, 115, 817
NET-Reflection 815
Network 869
Netzwerk 869
new 105, 140, 220
NewRow 760
Next 374
NextSibling 653, 688
Nodes 666
NodeType 679
NotifyFilter 416
Now 259
NTFS 445

null 53, 116
Nullable Type 53
Nutzer 858

O

object 46, 52
Objekt 105, 110
Objektbaum 202, 454
Objektinitialisierer 143, 339 f.
ODER 69
OleDbConnection 706
OOP 208
Open 436, 710
OpenFileDialog 419, 451
OpenOffice.org 851
OpenOrCreate 436
OpenSubKey 823, 867
OpenText 438
Operatoren 63, 101
Operatorenüberladung 189
Optionale Parameter 315
orderby 378
OSFullName 866
OSVersion 866
out 85
override 151
OverwritePrompt 452

P

PadLeft 233
PadRight 234
PAP 90
Parallel LINQ 585
Parallel.For 559
Parallel.ForEach 564
Parallel.Invoke 557
ParallelLoopResult 562
Parallel-Programmierung 553
Parameter 718
ParameterName 719
Parameterübergabe 84 f., 896
ParentNode 688
Parse 60, 259
Parser 635
Path 391, 400, 415
PC-Name 869
PDF 472
PDFsharp 474
PeekChar 441
Pegeldiagramm 880
PerformanceData 814
Pi 263
PI 635
Platform 866
PlatformID.Win32Windows 867
PLINQ 357, 585
Polarkoordinaten 190
Polling 514
Polymorphes Verhalten 157
Polymorphie 19, 111, 148, 160
Portieren 99
Potenz 264
Pow 263
PowerLineStatus 871
PowerStatus 871
PreferFairness 580
PreviousSibling 688
Primary 874
PrimaryMonitorMaximizedWindowSize 868
PrimaryMonitorSize 868
Priority 489
private 112
Procedure-Step 601
Process 544, 549
Processing Instructions 632, 635, 648
ProcessName 545
ProcessorCount 870
Process.Start 550
ProcessThread 544
ProductName 872
ProductVersion 872
Program 894
Program.cs 893
Programm starten 549
ProgressBar 878
ProgressChanged 511
Projektmappen-Explorer 15
Projekttyp 14
Property 11, 169
Property-Accessoren 126
protected 112
Provider 709
Prozeduren 104
Prozedurschritt 605
Prozesse 544
public 112
Pulse 496 f.
PulseAll 496 f.

Q

Query-Expression-Syntax 343, 360
Queue 298, 301
QueueUserWorkItem 491

R

Racing 484
Random 210, 265, 374
Range 373
Rank 230, 239 f.
Read 435
ReadAllBytes 441
ReadAllLines 439
ReadAllText 439
ReadContentAsFloat 680
ReadKey 900, 903
ReadLine 9
ReadLines 439
ReadToEnd 438 f.
ReadWrite 435
ReadXml 776, 781
ref 84 f.
Referenzieren 114
Referenztyp 52, 232
Reflexion 25
Regedit.exe 812
Registrierungsdatenbank 813
Registrierungseditor 812
Registry 812, 814, 822, 867
RegistryKey 812, 814
ReleaseMutex 500
Remove 234, 669, 763
RemoveAccessRule 413
RemoveAll 669
RemoveAnnotations 669
RemoveAttributes 669
RemoveContent 669
Repeat 374
Replace 234
Reset 290
Resume 487
return 74, 105, 300
Round 263
RowFilter 764
Rows 761
Rückrufmethode 513
Rücksprung 605

S

SaveFileDialog 419, 451
Scanner 842, 845
Scanner-Assistent 841
ScannerDeviceType 841
Schaltjahr 259
Schleifen 102
Schleifenabbruch 561
Schleifenanweisungen 74
Schlüsselwörter 43
Schriftarten 867
Screen 873
ScreenOrientation 868
sealed 163
Second 258
Security Engine 23
select 378
Select 676
SELECT 748
SelectCommand 725
SelectNodes 657
SelectSingleNode 653 f., 656 f., 696
Semaphore 502
SemaphoreSlim 585
SendMessage 830
Sequenzielle Datei 442
Serialisieren 443
Serialisierung 26
Serializable 203, 443
Serializable-Attribut 455
Serialization 204
Serialize 204
ServerVersion 709
ServicePack 866
set 104, 122, 126
SetAccessControl 388, 413
SetAttributeValue 668
SetBufferSize 900
SetClipboardViewer 830
SetCurrentDirectory 406
SetCursorPosition 900
SetDataObject 809
SetElementValue 668
Setup-Projekt 797
SetValue 822
SetWindowPosition 900
SetWindowSize 900
Shared Methoden 213
short 45
ShowAcquireImage 850

ShowAcquisitionWizard 841
ShowDialog 452
Sign 263
Sin 263
Single-Step 600
Skip 667
SkipWhile 667
Sleep 488
SocketDesignation 870
Sort 198, 230, 764
SortedList 301
SortedSet 321
Sortieren 377, 764
Sound 875, 877, 883
SpecialFolder 421
Sperrmechanismen 492
SpinLock 585
SpinWait 585
Split 234
Sqr 263
Stack 298
StartInfo 549
StartsWidth 234
State 710
static 104, 165
Statische Klassen 165
Statische Methoden 132
Statischer Konstruktor 143
Steuerelemente 11
StoredProcedure 714
StreamReader 434, 438
StreamWriter 434, 437, 476
string 46
String 232
Stringaddition 273
StringBuilder 477, 875
StringReader 434, 777
StringWriter 434
struct 79, 103, 442
Strukturen 103
Strukturvariable 80
SubKeyCount 814
Subklassen 151 f.
SubString 234
Suchen 765
Suspend 487
switch 71, 102
System 45, 421
System.Collections.Concurrent 584
System.Console 899
System.Diagnostics 544
System.Environment 865
SystemInformation 865
System.IO.Compression 447
System.IO.FileStream 433
System.IO.Stream 433
System.Management 859
System.Net 863
System.Nullable 53
System.Object 160
System.Reflection 815
System.Security.AccessControl 413
System.Security.Cryptography 446
System.Security.Principal 862
System.Threading 485, 556
System.Threading.Tasks 556
System.TimeSpan 545
System.Xml 689
System.XML 649
System.Xml.Linq 660
System.Xml.Serialization 674
System.Xml.XPath 678
System.Xml.Xsl 682

T

TableAdapter 768
TableDirect 714
TableName 762
Take 667
TakeWhile 667
Tan 263
Task
– Canceled 579
– ContinueWith 571, 580
– Created 579
– Datenübergabe 568
– Faulted 579
– Fehlerbehandlung 578
– IsCanceled 579
– IsCompleted 579
– IsFaulted 579
– Klasse 565
– RanToCompletion 579
– Result 571
– return 574
– Rückgabewerte 571
– Running 579
– Start 567
– starten 566
– Status 579
– TaskCreationOptions 580

– Task-Ende 580
– Task-Id 579
– User-Interface 580
– Verarbeitung abbrechen 574
– Wait 569
– WaitAll 570
– WaitingForActivation 579
– WaitingForChildrenToComplete 579
– WaitingToRun 579
– Weitere Eigenschaften 579
TaskCreationOptions 587
Task.Factory.StartNew 565, 588
TaskScheduler 580, 586
Tastaturabfrage 903
TerminalServerSession 869
Textdatei 437, 450
TextWriterTraceListener 611
Thin Client 174
Thread 486 f., 531, 544
– initialisieren 531
– synchronisieren 531
Thread Service 24
Thread<> 528
ThreadInteruptedException 487
ThreadPool 490
Threadsicher 503
Threadsichere Collections 584
ThreadState 489
ThreadWaitReason 547
Throw 618, 625
ThrowIfCancellationRequested 576
Timer 878
Timer-Threads 509
TimeSpan 545
TimeSpan-Klasse 273
Title 420, 452, 872
ToArray 369, 378, 380
ToCharArray 234
Today 259
ToLongDateString 259
ToLongTimeString 259
ToLower 234
Toolbox 16
ToShortDateString 259
ToShortTimeString 259
ToString 59, 266
TotalPhysicalMemory 870
TotalProcessorTime 545
TotalVirtualMemory 870
ToUpper 234
Trace 606, 610
TraceListener 611
TrackBar 273
Trademark 872
Transform 682
Transformationsdatei 682
TreeView 425, 691
Trefferanzahl 604
Trim 234
Truncate 436
try 102
Try-Catch 615
TryEnter 496, 499
Try-Finally 620
Tuple 320
Type 817
Typecasting 297, 325
Typinferenz 54, 339, 360
Typisierte DataSets 765
Typsicherheit 297
Typsuffixe 49

U

Überladene Methoden 131
Überwachungsfenster 598
Uhr anzeigen 260
Umgebungsvariablen 898
UML 148
Unboxing 62
UND 69
Unicode 49
UnicodeEncoding 468, 776
Unified Modeling Language 148
UnIndent 608
UnspecifiedDeviceType 841
Unterverzeichnis 407
Update 720, 727, 792
UpdateCommand 725
UserDomainName 869
UserInteractive 869, 872
User-Name 869
UserName 869
UserProfile 421
Users 814
using 8, 115, 285, 461
using static 172
UTF-8 635
UTF-16 635

V

ValidateNames 452
Value 719
ValueCount 814
var 48, 54
Variablen 45
Variablentypen 45
VB 99
Verarbeitungsstatus 562
Vererbung 111
Vergleichsoperatoren 67
Veröffentlichen 794
Verschlüsseln 444
Version 873
VersionString 866
Verweistypen 46
Verzögerte Initialisierung 147
Verzweigungen 102
Video 877, 883
VideoDeviceType 841
VirtualScreen 868
Visual Studio 4
Visual Studio Enterprise 4
Visual Studio Professional 4
void 83

W

W3C 648
Wait 496 f.
WaitForExit 550
WaitOne 500, 502, 589, 592
WAV 875
Webcam 843
Webpublishing-Assistent 789, 796
Werkzeugkasten 16
Wertetypen 46
Where 302, 667
while 74 f., 102
WIA 832, 843
Wiederholmuster 376
Wiederverwendbarkeit 111
Windows Management Instrumentations 865
WindowsIdentity 860, 863
Winkel 264
winmm.dll 884
WMI 865
work stealing 557
WorkerReportsProgress 511
WorkingArea 868, 874
WorkingSet 872
WPF 507
Write 435, 900
WriteAllBytes 441
WriteAllText 477
WriteAttributeString 680, 685 f.
WriteEndDocument 681, 685 f.
WriteEndElement 681, 685 f.
WriteIf 606
WriteLine 9, 606, 900
Writer 851
WriteStartDocument 680, 685 f.
WriteStartElement 680, 685 f.
WriteXml 776, 781
Wurzel 264

X

XAttribute 662
XComment 662
XDeclaration 662
XDocument 661, 663
XDocumentType 662
XElement 661
Xelement.Load 664
XElement.Parse 664
XML 629
XML transformieren 669
XmlAttribute 674
XMLAttribute 648
XMLCDATASection 648
XMLCharacterData 648
XMLComment 648
XmlDocument 676, 687, 696
XmlDocumentType 648
XmlElement 648, 674
XmlEntity 648
XmlEnum 674
XmlIgnore 674
XmlImplementation 648
XMLNamedNodeMap 648
XmlNode 652
XMLNode 648
XMLNodeList 648
XmlNodeType 695
XmlProcessingInstruction 648
XmlReader 678, 694
XmlReaderSettings 678, 694
XmlRoot 674
XML-Schema 642
XmlSerializer 672

XmlText 648
XmlTextWriter 776
XmlWriter 680
XmlWriterSettings 681
XNode 662
XOR 69
XPathDocument 676
XPathNavigator 676, 695 f.
XPathNodeIterator 676
XProcessingInstruction 662
xsd.exe 647
XSD-Schema 640
XslCompiledTransform 682
XSLT 682

Y

Year 258
yield 300, 328

Z

Zahlenformatierung 267
Zeitfunktionen 256
Zeitmessung 275
Zufallszahlen 265, 374
Zugriffsberechtigung 387, 413
Zuweisungsoperatoren 66
Zwischenablage 829