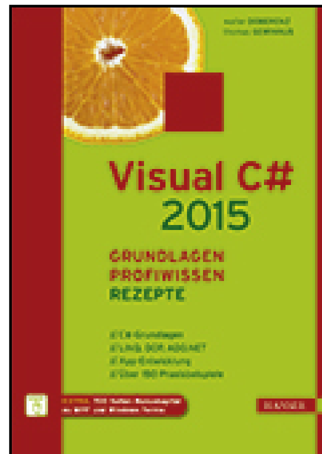
HANSER



Leseprobe

Walter Doberenz, Thomas Gewinnus

Visual C# 2015 – Grundlagen, Profiwissen und Rezepte

ISBN (Buch): 978-3-446-44381-5

ISBN (E-Book): 978-3-446-44606-9

Weitere Informationen oder Bestellungen unter

http://www.hanser-fachbuch.de/978-3-446-44381-5

sowie im Buchhandel.

© Carl Hanser Verlag, München

Inhaltsverzeichnis

Vorwort **45**

Teil I: Grundlagen

1 Einstieg in Visual Studio 2015 .... **51**

1.1 Die Installation von Visual Studio 2015 51
1.1.1 Überblick über die Produktpalette 51
1.1.2 Anforderungen an Hard- und Software 52
1.2 Unser allererstes C#-Programm 53
1.2.1 Vorbereitungen 53
1.2.2 Quellcode schreiben 55
1.2.3 Programm kompilieren und testen 55
1.2.4 Einige Erläuterungen zum Quellcode 56
1.2.5 Konsolenanwendungen sind out 57
1.3 Die Windows-Philosophie 57
1.3.1 Mensch-Rechner-Dialog 57
1.3.2 Objekt- und ereignisorientierte Programmierung 58
1.3.3 Programmieren mit Visual Studio 2015 59
1.4 Die Entwicklungsumgebung Visual Studio 2015 60
1.4.1 Neues Projekt 61
1.4.2 Die wichtigsten Fenster 62
1.5 Microsofts .NET-Technologie 65
1.5.1 Zur Geschichte von .NET 65
1.5.2 .NET-Features und Begriffe 67
1.6 Wichtige Neuigkeiten in Visual Studio 2015 74
1.6.1 Entwicklungsumgebung 74
1.6.2 Neue C#-Sprachfeatures 74
1.6.3 Code-Editor 75
1.6.4 NET Framework 4.6 75

1.7 Praxisbeispiele ... 76
1.7.1 Unsere erste Windows Forms-Anwendung ... 76
1.7.2 Umrechnung Euro-Dollar ... 80

2 Grundlagen der Sprache C# ... 89

2.1 Grundbegriffe ... 89
2.1.1 Anweisungen ... 89
2.1.2 Bezeichner ... 90
2.1.3 Schlüsselwörter ... 91
2.1.4 Kommentare ... 91
2.2 Datentypen, Variablen und Konstanten ... 92
2.2.1 Fundamentale Typen ... 92
2.2.2 Wertetypen versus Verweistypen ... 93
2.2.3 Benennung von Variablen ... 94
2.2.4 Deklaration von Variablen ... 94
2.2.5 Typsuffixe ... 96
2.2.6 Zeichen und Zeichenketten ... 97
2.2.7 object-Datentyp ... 99
2.2.8 Konstanten deklarieren ... 99
2.2.9 Nullable Types ... 100
2.2.10 Typinferenz ... 101
2.2.11 Gültigkeitsbereiche und Sichtbarkeit ... 102
2.3 Konvertieren von Datentypen ... 102
2.3.1 Implizite und explizite Konvertierung ... 102
2.3.2 Welcher Datentyp passt zu welchem? ... 104
2.3.3 Konvertieren von string ... 105
2.3.4 Die Convert-Klasse ... 107
2.3.5 Die Parse-Methode ... 107
2.3.6 Boxing und Unboxing ... 108
2.4 Operatoren ... 109
2.4.1 Arithmetische Operatoren ... 110
2.4.2 Zuweisungsoperatoren ... 111
2.4.3 Logische Operatoren ... 112
2.4.4 Rangfolge der Operatoren ... 115
2.5 Kontrollstrukturen ... 116
2.5.1 Verzweigungsbefehle ... 116
2.5.2 Schleifenanweisungen ... 119

2.6 Benutzerdefinierte Datentypen ... 122
2.6.1 Enumerationen ... 122
2.6.2 Strukturen ... 123
2.7 Nutzerdefinierte Methoden ... 125
2.7.1 Methoden mit Rückgabewert ... 126
2.7.2 Methoden ohne Rückgabewert ... 127
2.7.3 Parameterübergabe mit ref ... 128
2.7.4 Parameterübergabe mit out ... 129
2.7.5 Methodenüberladung ... 130
2.7.6 Optionale Parameter ... 131
2.7.7 Benannte Parameter ... 132
2.8 Praxisbeispiele ... 133
2.8.1 Vom PAP zur Konsolenanwendung ... 133
2.8.2 Ein Konsolen- in ein Windows-Programm verwandeln ... 135
2.8.3 Schleifenanweisungen verstehen ... 137
2.8.4 Benutzerdefinierte Methoden überladen ... 139
2.8.5 Anwendungen von Visual Basic nach C# portieren ... 142

3 OOP-Konzepte ... 149

3.1 Kleine Einführung in die OOP ... 149
3.1.1 Historische Entwicklung ... 149
3.1.2 Grundbegriffe der OOP ... 151
3.1.3 Sichtbarkeit von Klassen und ihren Mitgliedern ... 153
3.1.4 Allgemeiner Aufbau einer Klasse ... 154
3.1.5 Das Erzeugen eines Objekts ... 155
3.1.6 Einführungsbeispiel ... 158
3.2 Eigenschaften ... 163
3.2.1 Eigenschaften mit Zugriffsmethoden kapseln ... 163
3.2.2 Berechnete Eigenschaften ... 165
3.2.3 Lese-/Schreibschutz ... 167
3.2.4 Property-Accessoren ... 168
3.2.5 Statische Felder/Eigenschaften ... 168
3.2.6 Einfache Eigenschaften automatisch implementieren ... 171
3.3 Methoden ... 172
3.3.1 Öffentliche und private Methoden ... 172
3.3.2 Überladene Methoden ... 173
3.3.3 Statische Methoden ... 174

3.4 Ereignisse ... 176
3.4.1 Ereignis hinzufügen ... 176
3.4.2 Ereignis verwenden ... 179
3.5 Arbeiten mit Konstruktor und Destruktor ... 182
3.5.1 Konstruktor und Objektinitialisierer ... 182
3.5.2 Destruktor und Garbage Collector ... 185
3.5.3 Mit using den Lebenszyklus des Objekts kapseln ... 188
3.5.4 Verzögerte Initialisierung ... 188
3.6 Vererbung und Polymorphie ... 189
3.6.1 Klassendiagramm ... 189
3.6.2 Method-Overriding ... 190
3.6.3 Klassen implementieren ... 191
3.6.4 Implementieren der Objekte ... 194
3.6.5 Ausblenden von Mitgliedern durch Vererbung ... 196
3.6.6 Allgemeine Hinweise und Regeln zur Vererbung ... 197
3.6.7 Polymorphes Verhalten ... 199
3.6.8 Die Rolle von System.Object ... 201
3.7 Spezielle Klassen ... 202
3.7.1 Abstrakte Klassen ... 202
3.7.2 Versiegelte Klassen ... 204
3.7.3 Partielle Klassen ... 204
3.7.4 Statische Klassen ... 205
3.8 Schnittstellen (Interfaces) ... 206
3.8.1 Definition einer Schnittstelle ... 207
3.8.2 Implementieren einer Schnittstelle ... 207
3.8.3 Abfragen, ob Schnittstelle vorhanden ist ... 208
3.8.4 Mehrere Schnittstellen implementieren ... 209
3.8.5 Schnittstellenprogrammierung ist ein weites Feld ... 209
3.9 Praxisbeispiele ... 209
3.9.1 Eigenschaften sinnvoll kapseln ... 209
3.9.2 Eine statische Klasse anwenden ... 212
3.9.3 Vom fetten zum schlanken Client ... 214
3.9.4 Schnittstellenvererbung verstehen ... 226
3.9.5 Rechner für komplexe Zahlen ... 231
3.9.6 Formel-Rechner mit dem CodeDOM ... 240
3.9.7 Berechnungsergebnisse als Diagramm darstellen ... 248
3.9.8 Sortieren mit IComparable/IComparer ... 252
3.9.9 Einen Objektbaum in generischen Listen abspeichern ... 257

3.9.10 OOP beim Kartenspiel erlernen 263
3.9.11 Eine Klasse zur Matrizenrechnung entwickeln 267

4 Arrays, Strings, Funktionen 273

4.1 Datenfelder (Arrays) 273
4.1.1 Array deklarieren 273
4.1.2 Array instanziieren 274
4.1.3 Array initialisieren 274
4.1.4 Zugriff auf Array-Elemente 275
4.1.5 Zugriff mittels Schleife 276
4.1.6 Mehrdimensionale Arrays 277
4.1.7 Zuweisen von Arrays 279
4.1.8 Arrays aus Strukturvariablen 280
4.1.9 Löschen und Umdimensionieren von Arrays 281
4.1.10 Eigenschaften und Methoden von Arrays 282
4.1.11 Übergabe von Arrays 284
4.2 Verarbeiten von Zeichenketten 285
4.2.1 Zuweisen von Strings 285
4.2.2 Eigenschaften und Methoden von String-Variablen 286
4.2.3 Wichtige Methoden der String-Klasse 288
4.2.4 Die StringBuilder-Klasse 290
4.3 Reguläre Ausdrücke 292
4.3.1 Wozu werden reguläre Ausdrücke verwendet? 293
4.3.2 Eine kleine Einführung 293
4.3.3 Wichtige Methoden/Eigenschaften der Klasse Regex 294
4.3.4 Kompilierte reguläre Ausdrücke 296
4.3.5 RegexOptions-Enumeration 297
4.3.6 Metazeichen (Escape-Zeichen) 297
4.3.7 Zeichenmengen (Character Sets) 298
4.3.8 Quantifizierer 300
4.3.9 Zero-Width Assertions 301
4.3.10 Gruppen 304
4.3.11 Text ersetzen 305
4.3.12 Text splitten 306
4.4 Datums- und Zeitberechnungen 307
4.4.1 Die DateTime-Struktur 307
4.4.2 Wichtige Eigenschaften von DateTime-Variablen 308
4.4.3 Wichtige Methoden von DateTime-Variablen 309

4.4.4 Wichtige Mitglieder der DateTime-Struktur 309
4.4.5 Konvertieren von Datumstrings in DateTime-Werte 310
4.4.6 Die TimeSpan-Struktur 311
4.5 Mathematische Funktionen 312
4.5.1 Überblick 312
4.5.2 Zahlen runden 313
4.5.3 Winkel umrechnen 313
4.5.4 Potenz- und Wurzeloperationen 313
4.5.5 Logarithmus und Exponentialfunktionen 314
4.5.6 Zufallszahlen erzeugen 314
4.6 Zahlen- und Datumsformatierungen 315
4.6.1 Anwenden der ToString-Methode 315
4.6.2 Anwenden der Format-Methode 317
4.6.3 Stringinterpolation 318
4.7 Praxisbeispiele 319
4.7.1 Zeichenketten verarbeiten 319
4.7.2 Zeichenketten mit StringBuilder addieren 322
4.7.3 Reguläre Ausdrücke testen 325
4.7.4 Methodenaufrufe mit Array-Parametern 327

5 Weitere Sprachfeatures 331

5.1 Namespaces (Namensräume) 331
5.1.1 Ein kleiner Überblick 331
5.1.2 Einen eigenen Namespace einrichten 332
5.1.3 Die using-Anweisung 333
5.1.4 Namespace Alias 334
5.1.5 Namespace Alias Qualifizierer 334
5.2 Operatorenüberladung 335
5.2.1 Syntaxregeln 335
5.2.2 Praktische Anwendung 336
5.3 Collections (Auflistungen) 337
5.3.1 Die Schnittstelle IEnumerable 337
5.3.2 ArrayList 339
5.3.3 Hashtable 341
5.3.4 Indexer 341
5.4 Generics 343
5.4.1 Klassische Vorgehensweise 344
5.4.2 Generics bieten Typsicherheit 345

5.4.3 Generische Methoden 346
5.4.4 Iteratoren 347
5.5 Generische Collections 348
5.5.1 List-Collection statt ArrayList 348
5.5.2 Vorteile generischer Collections 349
5.5.3 Constraints 349
5.6 Das Prinzip der Delegates 349
5.6.1 Delegates sind Methodenzeiger 350
5.6.2 Einen Delegate-Typ deklarieren 350
5.6.3 Ein Delegate-Objekt erzeugen 350
5.6.4 Delegates vereinfacht instanziieren 352
5.6.5 Anonyme Methoden 353
5.6.6 Lambda-Ausdrücke 354
5.6.7 Lambda-Ausdrücke in der Task Parallel Library 356
5.7 Dynamische Programmierung 358
5.7.1 Wozu dynamische Programmierung? 358
5.7.2 Das Prinzip der dynamischen Programmierung 358
5.7.3 Optionale Parameter sind hilfreich 361
5.7.4 Kovarianz und Kontravarianz 362
5.8 Weitere Datentypen 362
5.8.1 BigInteger 362
5.8.2 Complex 365
5.8.3 Tuple<> 365
5.8.4 SortedSet<> 366
5.9 Praxisbeispiele 367
5.9.1 ArrayList versus generische List 367
5.9.2 Generische IEnumerable-Interfaces implementieren 371
5.9.3 Delegates, anonyme Methoden, Lambda Expressions 374
5.9.4 Dynamischer Zugriff auf COM Interop 378

6 Einführung in LINQ 381

6.1 LINQ-Grundlagen 381
6.1.1 Die LINQ-Architektur 381
6.1.2 Anonyme Typen 382
6.1.3 Erweiterungsmethoden 384
6.2 Abfragen mit LINQ to Objects 385
6.2.1 Grundlegendes zur LINQ-Syntax 385
6.2.2 Zwei alternative Schreibweisen von LINQ Abfragen 386

6.2.3 Übersicht der wichtigsten Abfrage-Operatoren ... 387
6.3 LINQ-Abfragen im Detail ... 388
6.3.1 Die Projektionsoperatoren Select und SelectMany ... 389
6.3.2 Der Restriktionsoperator Where ... 390
6.3.3 Die Sortierungsoperatoren OrderBy und ThenBy ... 391
6.3.4 Der Gruppierungsoperator GroupBy ... 392
6.3.5 Verknüpfen mit Join ... 395
6.3.6 Aggregat-Operatoren ... 395
6.3.7 Verzögertes Ausführen von LINQ-Abfragen ... 397
6.3.8 Konvertierungsmethoden ... 398
6.3.9 Abfragen mit PLINQ ... 398
6.4 Praxisbeispiele ... 401
6.4.1 Die Syntax von LINQ-Abfragen verstehen ... 401
6.4.2 Aggregat-Abfragen mit LINQ ... 404
6.4.3 LINQ im Schnelldurchgang erlernen ... 407
6.4.4 Strings mit LINQ abfragen und filtern ... 409
6.4.5 Duplikate aus einer Liste oder einem Array entfernen ... 410
6.4.6 Arrays mit LINQ initialisieren ... 413
6.4.7 Arrays per LINQ mit Zufallszahlen füllen ... 415
6.4.8 Einen String mit Wiederholmuster erzeugen ... 417
6.4.9 Mit LINQ Zahlen und Strings sortieren ... 418
6.4.10 Mit LINQ Collections von Objekten sortieren ... 419
6.4.11 Ergebnisse von LINQ-Abfragen in ein Array kopieren ... 422

Teil II: Technologien

7 Zugriff auf das Dateisystem ... 425
7.1 Grundlagen ... 425
7.1.1 Klassen für den Zugriff auf das Dateisystem ... 426
7.1.2 Statische versus Instanzen-Klasse ... 426
7.2 Übersichten ... 427
7.2.1 Methoden der Directory-Klasse ... 427
7.2.2 Methoden eines DirectoryInfo-Objekts ... 428
7.2.3 Eigenschaften eines DirectoryInfo-Objekts ... 428
7.2.4 Methoden der File-Klasse ... 428
7.2.5 Methoden eines FileInfo-Objekts ... 429
7.2.6 Eigenschaften eines FileInfo-Objekts ... 430

7.3 Operationen auf Verzeichnisebene 430
7.3.1 Existenz eines Verzeichnisses/einer Datei feststellen 430
7.3.2 Verzeichnisse erzeugen und löschen 431
7.3.3 Verzeichnisse verschieben und umbenennen 431
7.3.4 Aktuelles Verzeichnis bestimmen 432
7.3.5 Unterverzeichnisse ermitteln 432
7.3.6 Alle Laufwerke ermitteln 432
7.3.7 Dateien kopieren und verschieben 433
7.3.8 Dateien umbenennen 434
7.3.9 Dateiattribute feststellen 434
7.3.10 Verzeichnis einer Datei ermitteln 436
7.3.11 Alle im Verzeichnis enthaltenen Dateien ermitteln 436
7.3.12 Dateien und Unterverzeichnisse ermitteln 436
7.4 Zugriffsberechtigungen 437
7.4.1 ACL und ACE 437
7.4.2 SetAccessControl-Methode 438
7.4.3 Zugriffsrechte anzeigen 438
7.5 Weitere wichtige Klassen 439
7.5.1 Die Path-Klasse 439
7.5.2 Die Klasse FileSystemWatcher 440
7.6 Datei- und Verzeichnisdialoge 441
7.6.1 OpenFileDialog und SaveFileDialog 442
7.6.2 FolderBrowserDialog 443
7.7 Praxisbeispiele 444
7.7.1 Infos über Verzeichnisse und Dateien gewinnen 444
7.7.2 Eine Verzeichnisstruktur in die TreeView einlesen 448
7.7.3 Mit LINQ und RegEx Verzeichnisbäume durchsuchen 450

8 Dateien lesen und schreiben 455

8.1 Grundprinzip der Datenpersistenz 455
8.1.1 Dateien und Streams 455
8.1.2 Die wichtigsten Klassen 456
8.1.3 Erzeugen eines Streams 457
8.2 Dateiparameter 457
8.2.1 FileAccess 457
8.2.2 FileMode 457
8.2.3 FileShare 458

8.3 Textdateien ... 458
8.3.1 Eine Textdatei beschreiben bzw. neu anlegen ... 458
8.3.2 Eine Textdatei lesen ... 460
8.4 Binärdateien ... 461
8.4.1 Lese-/Schreibzugriff ... 461
8.4.2 Die Methoden ReadAllBytes und WriteAllBytes ... 462
8.4.3 Erzeugen von BinaryReader/BinaryWriter ... 462
8.5 Sequenzielle Dateien ... 463
8.5.1 Lesen und schreiben von strukturierten Daten ... 463
8.5.2 Serialisieren von Objekten ... 464
8.6 Dateien verschlüsseln und komprimieren ... 465
8.6.1 Das Methodenpärchen Encrypt/Decrypt ... 465
8.6.2 Verschlüsseln unter Windows Vista/7/8/10 ... 465
8.6.3 Verschlüsseln mit der CryptoStream-Klasse ... 466
8.6.4 Dateien komprimieren ... 467
8.7 Memory Mapped Files ... 468
8.7.1 Grundprinzip ... 468
8.7.2 Erzeugen eines MMF ... 469
8.7.3 Erstellen eines Map View ... 470
8.8 Praxisbeispiele ... 471
8.8.1 Auf eine Textdatei zugreifen ... 471
8.8.2 Einen Objektbaum persistent speichern ... 474
8.8.3 Ein Memory Mapped File (MMF) verwenden ... 481
8.8.4 Hex-Dezimal-Bytes-Konverter ... 483
8.8.5 Eine Datei verschlüsseln ... 487
8.8.6 Eine Datei komprimieren ... 490
8.8.7 Echte ZIP-Dateien erstellen ... 492
8.8.8 PDFs erstellen/exportieren ... 494
8.8.9 Eine CSV-Datei erstellen ... 497
8.8.10 Eine CSV-Datei mit LINQ lesen und auswerten ... 500
8.8.11 Einen korrekten Dateinamen erzeugen ... 503

9 Asynchrone Programmierung ... 505

9.1 Übersicht ... 505
9.1.1 Multitasking versus Multithreading ... 506
9.1.2 Deadlocks ... 507
9.1.3 Racing ... 507

9.2 Programmieren mit Threads ... 509
9.2.1 Einführungsbeispiel ... 509
9.2.2 Wichtige Thread-Methoden ... 510
9.2.3 Wichtige Thread-Eigenschaften ... 512
9.2.4 Einsatz der ThreadPool-Klasse ... 513
9.3 Sperrmechanismen ... 515
9.3.1 Threading ohne lock ... 515
9.3.2 Threading mit lock ... 517
9.3.3 Die Monitor-Klasse ... 519
9.3.4 Mutex ... 522
9.3.5 Methoden für die parallele Ausführung sperren ... 524
9.3.6 Semaphore ... 524
9.4 Interaktion mit der Programmoberfläche ... 526
9.4.1 Die Werkzeuge ... 526
9.4.2 Einzelne Steuerelemente mit Invoke aktualisieren ... 526
9.4.3 Mehrere Steuerelemente aktualisieren ... 528
9.4.4 Ist ein Invoke-Aufruf nötig? ... 528
9.4.5 Und was ist mit WPF? ... 529
9.5 Timer-Threads ... 530
9.6 Die BackgroundWorker-Komponente ... 531
9.7 Asynchrone Programmier-Entwurfsmuster ... 534
9.7.1 Kurzübersicht ... 534
9.7.2 Polling ... 535
9.7.3 Callback verwenden ... 537
9.7.4 Callback mit Parameterübergabe verwenden ... 538
9.7.5 Callback mit Zugriff auf die Programm-Oberfläche ... 539
9.8 Asynchroner Aufruf beliebiger Methoden ... 540
9.8.1 Die Beispielklasse ... 540
9.8.2 Asynchroner Aufruf ohne Callback ... 542
9.8.3 Asynchroner Aufruf mit Callback und Anzeigefunktion ... 543
9.8.4 Aufruf mit Rückgabewerten (per Eigenschaft) ... 544
9.8.5 Aufruf mit Rückgabewerten (per EndInvoke) ... 544
9.9 Es geht auch einfacher – async und await ... 545
9.9.1 Der Weg von Synchron zu Asynchron ... 546
9.9.2 Achtung: Fehlerquellen! ... 548
9.9.3 Eigene asynchrone Methoden entwickeln ... 550

9.10 Praxisbeispiele ... 552
9.10.1 Spieltrieb & Multithreading erleben ... 552
9.10.2 Prozess- und Thread-Informationen gewinnen ... 565
9.10.3 Ein externes Programm starten ... 570

10 Die Task Parallel Library ... 573

10.1 Überblick ... 573
10.1.1 Parallel-Programmierung ... 573
10.1.2 Möglichkeiten der TPL ... 576
10.1.3 Der CLR-Threadpool ... 576
10.2 Parallele Verarbeitung mit Parallel.Invoke ... 577
10.2.1 Aufrufvarianten ... 578
10.2.2 Einschränkungen ... 579
10.3 Verwendung von Parallel.For ... 579
10.3.1 Abbrechen der Verarbeitung ... 581
10.3.2 Auswerten des Verarbeitungsstatus ... 582
10.3.3 Und was ist mit anderen Iterator-Schrittweiten? ... 583
10.4 Collections mit Parallel.ForEach verarbeiten ... 583
10.5 Die Task-Klasse ... 584
10.5.1 Einen Task erzeugen ... 584
10.5.2 Den Task starten ... 585
10.5.3 Datenübergabe an den Task ... 587
10.5.4 Wie warte ich auf das Ende des Task? ... 588
10.5.5 Tasks mit Rückgabewerten ... 590
10.5.6 Die Verarbeitung abbrechen ... 593
10.5.7 Fehlerbehandlung ... 596
10.5.8 Weitere Eigenschaften ... 597
10.6 Zugriff auf das User-Interface ... 598
10.6.1 Task-Ende und Zugriff auf die Oberfläche ... 599
10.6.2 Zugriff auf das UI aus dem Task heraus ... 600
10.7 Weitere Datenstrukturen im Überblick ... 602
10.7.1 Threadsichere Collections ... 602
10.7.2 Primitive für die Threadsynchronisation ... 603
10.8 Parallel LINQ (PLINQ) ... 603
10.9 Praxisbeispiel: Spieltrieb – Version 2 ... 604
10.9.1 Aufgabenstellung ... 604
10.9.2 Global-Klasse ... 604
10.9.3 Controller-Klasse ... 606

10.9.4 LKW-Klasse ... 607
10.9.5 Schiff-Klasse ... 609
10.9.6 Oberfläche ... 611

11 Fehlersuche und Behandlung ... 613

11.1 Der Debugger ... 613
11.1.1 Allgemeine Beschreibung ... 613
11.1.2 Die wichtigsten Fenster ... 614
11.1.3 Debugging-Optionen ... 617
11.1.4 Praktisches Debugging am Beispiel ... 619
11.2 Arbeiten mit Debug und Trace ... 623
11.2.1 Wichtige Methoden von Debug und Trace ... 623
11.2.2 Besonderheiten der Trace-Klasse ... 627
11.2.3 TraceListener-Objekte ... 627
11.3 Caller Information ... 630
11.3.1 Attribute ... 630
11.3.2 Anwendung ... 630
11.4 Fehlerbehandlung ... 631
11.4.1 Anweisungen zur Fehlerbehandlung ... 631
11.4.2 try-catch ... 632
11.4.3 try-finally ... 637
11.4.4 Das Standardverhalten bei Ausnahmen festlegen ... 639
11.4.5 Die Exception-Klasse ... 640
11.4.6 Fehler/Ausnahmen auslösen ... 641
11.4.7 Eigene Fehlerklassen ... 641
11.4.8 Exceptionhandling zur Entwurfszeit ... 643
11.4.9 Code Contracts ... 644

12 XML in Theorie und Praxis ... 645

12.1 XML – etwas Theorie ... 645
12.1.1 Übersicht ... 645
12.1.2 Der XML-Grundaufbau ... 648
12.1.3 Wohlgeformte Dokumente ... 649
12.1.4 Processing Instructions (PI) ... 652
12.1.5 Elemente und Attribute ... 652
12.1.6 Verwendbare Zeichensätze ... 654

12.2 XSD-Schemas ... 656
12.2.1 XSD-Schemas und ADO.NET ... 656
12.2.2 XML-Schemas in Visual Studio analysieren ... 658
12.2.3 XML-Datei mit XSD-Schema erzeugen ... 661
12.2.4 XSD-Schema aus einer XML-Datei erzeugen ... 662
12.3 Verwendung des DOM unter .NET ... 663
12.3.1 Übersicht ... 663
12.3.2 DOM-Integration in C# ... 664
12.3.3 Laden von Dokumenten ... 664
12.3.4 Erzeugen von XML-Dokumenten ... 665
12.3.5 Auslesen von XML-Dateien ... 667
12.3.6 Direktzugriff auf einzelne Elemente ... 669
12.3.7 Einfügen von Informationen ... 669
12.3.8 Suchen in den Baumzweigen ... 672
12.4 XML-Verarbeitung mit LINQ to XML ... 675
12.4.1 Die LINQ to XML-API ... 675
12.4.2 Neue XML-Dokumente erzeugen ... 677
12.4.3 Laden und Sichern von XML-Dokumenten ... 679
12.4.4 Navigieren in XML-Daten ... 680
12.4.5 Auswählen und Filtern ... 682
12.4.6 Manipulieren der XML-Daten ... 682
12.4.7 XML-Dokumente transformieren ... 684
12.5 Weitere Möglichkeiten der XML-Verarbeitung ... 687
12.5.1 Die relationale Sicht mit XmlDataDocument ... 687
12.5.2 XML-Daten aus Objektstrukturen erzeugen ... 690
12.5.3 Schnelles Suchen in XML-Daten mit XPathNavigator ... 693
12.5.4 Schnelles Auslesen von XML-Daten mit XmlReader ... 696
12.5.5 Erzeugen von XML-Daten mit XmlWriter ... 698
12.5.6 XML transformieren mit XSLT ... 700
12.6 Praxisbeispiele ... 702
12.6.1 Mit dem DOM in XML-Dokumenten navigieren ... 702
12.6.2 XML-Daten in eine TreeView einlesen ... 705
12.6.3 Ein DataSet in einen XML-String konvertieren ... 709
12.6.4 Ein DataSet in einer XML-Datei speichern ... 713
12.6.5 In Dokumenten mit dem XPathNavigator navigieren ... 716

13 Einführung in ADO.NET ... 721

13.1 Eine kleine Übersicht ... 721
13.1.1 Die ADO.NET-Klassenhierarchie ... 721
13.1.2 Die Klassen der Datenprovider ... 722
13.1.3 Das Zusammenspiel der ADO.NET-Klassen ... 725
13.2 Das Connection-Objekt ... 726
13.2.1 Allgemeiner Aufbau ... 726
13.2.2 OleDbConnection ... 726
13.2.3 Schließen einer Verbindung ... 728
13.2.4 Eigenschaften des Connection-Objekts ... 728
13.2.5 Methoden des Connection-Objekts ... 730
13.2.6 Der ConnectionStringBuilder ... 731
13.3 Das Command-Objekt ... 731
13.3.1 Erzeugen und Anwenden eines Command-Objekts ... 732
13.3.2 Erzeugen mittels CreateCommand-Methode ... 732
13.3.3 Eigenschaften des Command-Objekts ... 733
13.3.4 Methoden des Command-Objekts ... 735
13.3.5 Freigabe von Connection- und Command-Objekten ... 736
13.4 Parameter-Objekte ... 737
13.4.1 Erzeugen und Anwenden eines Parameter-Objekts ... 737
13.4.2 Eigenschaften des Parameter-Objekts ... 738
13.5 Das CommandBuilder-Objekt ... 739
13.5.1 Erzeugen ... 739
13.5.2 Anwenden ... 739
13.6 Das DataReader-Objekt ... 740
13.6.1 DataReader erzeugen ... 740
13.6.2 Daten lesen ... 741
13.6.3 Eigenschaften des DataReaders ... 742
13.6.4 Methoden des DataReaders ... 742
13.7 Das DataAdapter-Objekt ... 743
13.7.1 DataAdapter erzeugen ... 743
13.7.2 Command-Eigenschaften ... 744
13.7.3 Fill-Methode ... 745
13.7.4 Update-Methode ... 746
13.8 Praxisbeispiele ... 747
13.8.1 Wichtige ADO.NET-Objekte im Einsatz ... 747
13.8.2 Eine Aktionsabfrage ausführen ... 748

13.8.3 Eine Auswahlabfrage aufrufen 751
13.8.4 Die Datenbank aktualisieren 753

14 Das DataSet 757

14.1 Grundlegende Features des DataSets 757
14.1.1 Die Objekthierarchie 758
14.1.2 Die wichtigsten Klassen 758
14.1.3 Erzeugen eines DataSets 759
14.2 Das DataTable-Objekt 760
14.2.1 DataTable erzeugen 761
14.2.2 Spalten hinzufügen 761
14.2.3 Zeilen zur DataTable hinzufügen 762
14.2.4 Auf den Inhalt einer DataTable zugreifen 763
14.3 Die DataView 765
14.3.1 Erzeugen einer DataView 765
14.3.2 Sortieren und Filtern von Datensätzen 765
14.3.3 Suchen von Datensätzen 766
14.4 Typisierte DataSets 766
14.4.1 Ein typisiertes DataSet erzeugen 767
14.4.2 Das Konzept der Datenquellen 768
14.4.3 Typisierte DataSets und TableAdapter 769
14.5 Die Qual der Wahl 770
14.5.1 DataReader – der schnelle Lesezugriff 771
14.5.2 DataSet – die Datenbank im Hauptspeicher 771
14.5.3 Objektrelationales Mapping – die Zukunft? 772
14.6 Praxisbeispiele 773
14.6.1 In der DataView sortieren und filtern 773
14.6.2 Suche nach Datensätzen 775
14.6.3 Ein DataSet in einen XML-String serialisieren 776
14.6.4 Untypisiertes in ein typisiertes DataSet konvertieren 781
14.6.5 Eine LINQ to SQL-Abfrage ausführen 786

15 Verteilen von Anwendungen 791

15.1 ClickOnce-Deployment 792
15.1.1 Übersicht/Einschränkungen 792
15.1.2 Die Vorgehensweise 793
15.1.3 Ort der Veröffentlichung 793
15.1.4 Anwendungsdateien 794

15.1.5 Erforderliche Komponenten 794
15.1.6 Aktualisierungen 795
15.1.7 Veröffentlichungsoptionen 796
15.1.8 Veröffentlichen 797
15.1.9 Verzeichnisstruktur 797
15.1.10 Der Webpublishing-Assistent 799
15.1.11 Neue Versionen erstellen 800
15.2 InstallShield 800
15.2.1 Installation 800
15.2.2 Aktivieren 801
15.2.3 Ein neues Setup-Projekt 801
15.2.4 Finaler Test 809
15.3 Hilfedateien programmieren 809
15.3.1 Der HTML Help Workshop 810
15.3.2 Bedienung am Beispiel 811
15.3.3 Hilfedateien in die Visual C#-Anwendung einbinden 813
15.3.4 Eine alternative Hilfe-IDE verwenden 817

16 Weitere Techniken 819

16.1 Zugriff auf die Zwischenablage 819
16.1.1 Das Clipboard-Objekt 819
16.1.2 Zwischenablage-Funktionen für Textboxen 821
16.2 Arbeiten mit der Registry 821
16.2.1 Allgemeines 822
16.2.2 Registry-Unterstützung in .NET 824
16.3 .NET-Reflection 825
16.3.1 Übersicht 825
16.3.2 Assembly laden 825
16.3.3 Mittels GetType und Type Informationen sammeln 826
16.3.4 Dynamisches Laden von Assemblies 828
16.4 Praxisbeispiele 831
16.4.1 Zugriff auf die Registry 831
16.4.2 Dateiverknüpfungen erzeugen 833
16.4.3 Betrachter für Manifestressourcen 835
16.4.4 Die Zwischenablage überwachen und anzeigen 838
16.4.5 Die WIA-Library kennenlernen 841
16.4.6 Auf eine Webcam zugreifen 853
16.4.7 Auf den Scanner zugreifen 855

16.4.8 OpenOffice.org Writer per OLE steuern 860
16.4.9 Nutzer und Gruppen des aktuellen Systems ermitteln 868
16.4.10 Testen, ob Nutzer in einer Gruppe enthalten ist 869
16.4.11 Testen, ob der Nutzer ein Administrator ist 871
16.4.12 Die IP-Adressen des Computers bestimmen 872
16.4.13 Die IP-Adresse über den Hostnamen bestimmen 873
16.4.14 Diverse Systeminformationen ermitteln 874
16.4.15 Environment Variablen auslesen 878
16.4.16 Alles über den Bildschirm erfahren 882
16.4.17 Sound per MCI aufnehmen 883
16.4.18 Mikrofonpegel anzeigen 887
16.4.19 Pegeldiagramm aufzeichnen 888
16.4.20 Sound-und Video-Dateien per MCI abspielen 892

17 Konsolenanwendungen 901

17.1 Grundaufbau/Konzepte 901
17.1.1 Unser Hauptprogramm – Program.cs 902
17.1.2 Rückgabe eines Fehlerstatus 903
17.1.3 Parameterübergabe 904
17.1.4 Zugriff auf die Umgebungsvariablen 905
17.2 Die Kommandozentrale: System.Console 906
17.2.1 Eigenschaften 907
17.2.2 Methoden/Ereignisse 907
17.2.3 Textausgaben 908
17.2.4 Farbangaben 909
17.2.5 Tastaturabfragen 910
17.2.6 Arbeiten mit Streamdaten 911
17.3 Praxisbeispiel 913
17.3.1 Farbige Konsolenanwendung 913
17.3.2 Weitere Hinweise und Beispiele 915

Teil III: Windows Apps

18 Erste Schritte 919

18.1 Grundkonzepte und Begriffe 919
18.1.1 Windows Runtime (WinRT) 919
18.1.2 Windows Apps 920

18.1.3 Fast and Fluid 921
18.1.4 Process Sandboxing und Contracts 922
18.1.5 .NET WinRT-Profil 924
18.1.6 Language Projection 924
18.1.7 Vollbildmodus – War da was? 926
18.1.8 Windows Store 926
18.1.9 Zielplattformen 927
18.2 Entwurfsumgebung 928
18.2.1 Betriebssystem 928
18.2.2 Windows-Simulator 928
18.2.3 Remote-Debugging 931
18.3 Ein (kleines) Einstiegsbeispiel 932
18.3.1 Aufgabenstellung 932
18.3.2 Quellcode 932
18.3.3 Oberflächenentwurf 935
18.3.4 Installation und Test 937
18.3.5 Touchscreen 939
18.3.6 Fazit 939
18.4 Weitere Details zu WinRT 941
18.4.1 Wo ist WinRT einzuordnen? 942
18.4.2 Die WinRT-API 943
18.4.3 Wichtige WinRT-Namespaces 945
18.4.4 Der Unterbau 946
18.5 Praxisbeispiel 948
18.5.1 WinRT in Desktop-Applikationen nutzen 948

19 App-Oberflächen entwerfen 953

19.1 Grundkonzepte 953
19.1.1 XAML (oder HTML 5) für die Oberfläche 954
19.1.2 Die Page, der Frame und das Window 955
19.1.3 Das Befehlsdesign 957
19.1.4 Die Navigationsdesigns 959
19.1.5 Achtung: Fingereingabe! 960
19.1.6 Verwendung von Schriftarten 960
19.2 Seitenauswahl und -navigation 961
19.2.1 Die Startseite festlegen 961
19.2.2 Navigation und Parameterübergabe 961
19.2.3 Den Seitenstatus erhalten 962

19.3 App-Darstellung ... 963
19.3.1 Vollbild quer und hochkant ... 963
19.3.2 Was ist mit Andocken und Füllmodus? ... 964
19.3.3 Reagieren auf die Änderung ... 964
19.4 Skalieren von Apps ... 966
19.5 Praxisbeispiele ... 969
19.5.1 Seitennavigation und Parameterübergabe ... 969
19.5.2 Die Fensterkopfzeile anpassen ... 971

20 Die wichtigsten Controls ... 973

20.1 Einfache WinRT-Controls ... 973
20.1.1 TextBlock, RichTextBlock ... 973
20.1.2 Button, HyperlinkButton, RepeatButton ... 976
20.1.3 CheckBox, RadioButton, ToggleButton, ToggleSwitch ... 978
20.1.4 TextBox, PasswordBox, RichEditBox ... 979
20.1.5 Image ... 983
20.1.6 ScrollBar, Slider, ProgressBar, ProgressRing ... 985
20.1.7 Border, Ellipse, Rectangle ... 986
20.2 Layout-Controls ... 987
20.2.1 Canvas ... 987
20.2.2 StackPanel ... 988
20.2.3 ScrollViewer ... 988
20.2.4 Grid ... 989
20.2.5 VariableSizedWrapGrid ... 990
20.2.6 SplitView ... 991
20.2.7 Pivot ... 995
20.2.8 RelativPanel ... 997
20.3 Listendarstellungen ... 999
20.3.1 ComboBox, ListBox ... 999
20.3.2 ListView ... 1003
20.3.3 GridView ... 1004
20.3.4 FlipView ... 1006
20.4 Sonstige Controls ... 1008
20.4.1 CaptureElement ... 1008
20.4.2 MediaElement ... 1010
20.4.3 Frame ... 1011
20.4.4 WebView ... 1011
20.4.5 ToolTip ... 1012

20.4.6 CalendarDatePicker 1014
20.4.7 DatePicker/TimePicker 1015
20.5 Praxisbeispiele 1015
20.5.1 Einen StringFormat-Konverter implementieren 1015
20.5.2 Besonderheiten der TextBox kennen lernen 1017
20.5.3 Daten in der GridView gruppieren 1021
20.5.4 Das SemanticZoom-Control verwenden 1025
20.5.5 Die CollectionViewSource verwenden 1030
20.5.6 Zusammenspiel ListBox/AppBar 1033

21 Apps im Detail 1037

21.1 Ein Windows App-Projekt im Detail 1037
21.1.1 Contracts und Extensions 1038
21.1.2 AssemblyInfo.cs 1038
21.1.3 Verweise 1040
21.1.4 App.xaml und App.xaml.cs 1040
21.1.5 Package.appxmanifest 1041
21.1.6 Application1_TemporaryKey.pfx 1046
21.1.7 MainPage.xaml & MainPage.xaml.cs 1046
21.1.8 Assets/Symbole 1047
21.1.9 Nach dem Kompilieren 1047
21.2 Der Lebenszyklus einer Windows App 1047
21.2.1 Möglichkeiten der Aktivierung von Apps 1049
21.2.2 Der Splash Screen 1051
21.2.3 Suspending 1051
21.2.4 Resuming 1053
21.2.5 Beenden von Apps 1053
21.2.6 Die Ausnahmen von der Regel 1054
21.2.7 Debuggen 1054
21.3 Daten speichern und laden 1058
21.3.1 Grundsätzliche Überlegungen 1058
21.3.2 Worauf und wie kann ich zugreifen? 1058
21.3.3 Das AppData-Verzeichnis 1059
21.3.4 Das Anwendungs-Installationsverzeichnis 1061
21.3.5 Das Downloads-Verzeichnis 1062
21.3.6 Sonstige Verzeichnisse 1063
21.3.7 Anwendungsdaten lokal sichern und laden 1063
21.3.8 Daten in der Cloud ablegen/laden (Roaming) 1065

21.3.9 Aufräumen ... 1067
21.3.10 Sensible Informationen speichern ... 1068
21.4 Praxisbeispiele ... 1069
21.4.1 Die Auto-Play-Funktion unterstützen ... 1069
21.4.2 Einen zusätzlichen Splash Screen einsetzen ... 1073
21.4.3 Eine Dateiverknüpfung erstellen ... 1075

22 App-Techniken ... 1081

22.1 Arbeiten mit Dateien/Verzeichnissen ... 1081
22.1.1 Verzeichnisinformationen auflisten ... 1081
22.1.2 Unterverzeichnisse auflisten ... 1084
22.1.3 Verzeichnisse erstellen/löschen ... 1085
22.1.4 Dateien auflisten ... 1087
22.1.5 Dateien erstellen/schreiben/lesen ... 1089
22.1.6 Dateien kopieren/umbenennen/löschen ... 1093
22.1.7 Verwenden der Dateipicker ... 1094
22.1.8 StorageFile-/StorageFolder-Objekte speichern ... 1099
22.1.9 Verwenden der Most Recently Used-Liste ... 1101
22.2 Datenaustausch zwischen Apps/Programmen ... 1102
22.2.1 Zwischenablage ... 1102
22.2.2 Teilen von Inhalten ... 1109
22.2.3 Eine App als Freigabeziel verwenden ... 1113
22.2.4 Zugriff auf die Kontaktliste ... 1114
22.3 Spezielle Oberflächenelemente ... 1115
22.3.1 MessageDialog ... 1116
22.3.2 ContentDialog ... 1119
22.3.3 Popup-Benachrichtigungen ... 1121
22.3.4 PopUp/Flyouts ... 1129
22.3.5 Das PopupMenu einsetzen ... 1132
22.3.6 Eine AppBar/CommandBar verwenden ... 1133
22.4 Datenbanken und Windows Store Apps ... 1137
22.4.1 Der Retter in der Not: SQLite! ... 1137
22.4.2 Verwendung/Kurzüberblick ... 1138
22.4.3 Installation ... 1139
22.4.4 Wie kommen wir zu einer neuen Datenbank? ... 1140
22.4.5 Wie werden die Daten manipuliert? ... 1144

22.5 Vertrieb der App ... 1146
22.5.1 Verpacken der App ... 1146
22.5.2 App-Installation per Skript ... 1148
22.6 Ein Blick auf die App-Schwachstellen ... 1149
22.6.1 Quellcodes im Installationsverzeichnis ... 1149
22.6.2 Zugriff auf den App-Datenordner ... 1151
22.7 Praxisbeispiele ... 1152
22.7.1 Ein Verzeichnis auf Änderungen überwachen ... 1152
22.7.2 Eine App als Freigabeziel verwenden ... 1154
22.7.3 ToastNotifications einfach erzeugen ... 1159

Anhang

A Glossar ... 1167

B Wichtige Dateiextensions ... 1173

Index ... 1175

Download-Kapitel

LINK: `http://doko-buch.de`

Vorwort zu den Download-Kapiteln 1215

Teil IV: WPF-Anwendungen

23 Einführung in WPF 1219

23.1 Neues aus der Gerüchteküche 1220
23.2 Einführung 1220
23.2.1 Was kann eine WPF-Anwendung? 1220
23.2.2 Die eXtensible Application Markup Language 1222
23.2.3 Verbinden von XAML und C#-Code 1227
23.2.4 Zielplattformen 1232
23.2.5 Applikationstypen 1233
23.2.6 Vor- und Nachteile von WPF-Anwendungen 1234
23.2.7 Weitere Dateien im Überblick 1235
23.3 Alles beginnt mit dem Layout 1237
23.3.1 Allgemeines zum Layout 1237
23.3.2 Positionieren von Steuerelementen 1239
23.3.3 Canvas 1243
23.3.4 StackPanel 1243
23.3.5 DockPanel 1245
23.3.6 WrapPanel 1247
23.3.7 UniformGrid 1247
23.3.8 Grid 1249
23.3.9 ViewBox 1254
23.3.10 TextBlock 1255

23.4 Das WPF-Programm 1258
23.4.1 Die App-Klasse 1258
23.4.2 Das Startobjekt festlegen 1259
23.4.3 Kommandozeilenparameter verarbeiten 1260
23.4.4 Die Anwendung beenden 1261
23.4.5 Auswerten von Anwendungsereignissen 1261
23.5 Die Window-Klasse 1262
23.5.1 Position und Größe festlegen 1262
23.5.2 Rahmen und Beschriftung 1263
23.5.3 Das Fenster-Icon ändern 1263
23.5.4 Anzeige weiterer Fenster 1264
23.5.5 Transparenz 1264
23.5.6 Abstand zum Inhalt festlegen 1265
23.5.7 Fenster ohne Fokus anzeigen 1265
23.5.8 Ereignisfolge bei Fenstern 1266
23.5.9 Ein paar Worte zur Schriftdarstellung 1266
23.5.10 Ein paar Worte zur Darstellung von Controls 1269

24 Übersicht WPF-Controls 1273

24.1 Allgemeingültige Eigenschaften 1273
24.2 Label 1275
24.3 Button, RepeatButton, ToggleButton 1276
24.3.1 Schaltflächen für modale Dialoge 1276
24.3.2 Schaltflächen mit Grafik 1277
24.4 TextBox, PasswordBox 1278
24.4.1 TextBox 1278
24.4.2 PasswordBox 1280
24.5 CheckBox 1281
24.6 RadioButton 1283
24.7 ListBox, ComboBox 1284
24.7.1 ListBox 1284
24.7.2 ComboBox 1287
24.7.3 Den Content formatieren 1289
24.8 Image 1290
24.8.1 Grafik per XAML zuweisen 1290
24.8.2 Grafik zur Laufzeit zuweisen 1291
24.8.3 Bild aus Datei laden 1292
24.8.4 Die Grafikskalierung beeinflussen 1293

24.9 MediaElement ... 1294
24.10 Slider, ScrollBar ... 1296
24.10.1 Slider ... 1296
24.10.2 ScrollBar ... 1297
24.11 ScrollViewer ... 1298
24.12 Menu, ContextMenu ... 1299
24.12.1 Menu ... 1299
24.12.2 Tastenkürzel ... 1300
24.12.3 Grafiken ... 1301
24.12.4 Weitere Möglichkeiten ... 1302
24.12.5 ContextMenu ... 1303
24.13 ToolBar ... 1304
24.14 StatusBar, ProgressBar ... 1307
24.14.1 StatusBar ... 1307
24.14.2 ProgressBar ... 1309
24.15 Border, GroupBox, BulletDecorator ... 1310
24.15.1 Border ... 1310
24.15.2 GroupBox ... 1311
24.15.3 BulletDecorator ... 1312
24.16 RichTextBox ... 1314
24.16.1 Verwendung und Anzeige von vordefiniertem Text ... 1314
24.16.2 Neues Dokument zur Laufzeit erzeugen ... 1316
24.16.3 Sichern von Dokumenten ... 1316
24.16.4 Laden von Dokumenten ... 1318
24.16.5 Texte per Code einfügen/modifizieren ... 1319
24.16.6 Texte formatieren ... 1320
24.16.7 EditingCommands ... 1321
24.16.8 Grafiken/Objekte einfügen ... 1322
24.16.9 Rechtschreibkontrolle ... 1324
24.17 FlowDocumentPageViewer & Co. ... 1324
24.17.1 FlowDocumentPageViewer ... 1324
24.17.2 FlowDocumentReader ... 1325
24.17.3 FlowDocumentScrollViewer ... 1325
24.18 FlowDocument ... 1325
24.18.1 FlowDocument per XAML beschreiben ... 1326
24.18.2 FlowDocument per Code erstellen ... 1328
24.19 DocumentViewer ... 1329
24.20 Expander, TabControl ... 1330

24.20.1 Expander ... 1330
24.20.2 TabControl ... 1332
24.21 Popup ... 1333
24.22 TreeView ... 1335
24.23 ListView ... 1338
24.24 DataGrid ... 1339
24.25 Calendar/DatePicker ... 1340
24.26 InkCanvas ... 1344
24.26.1 Stift-Parameter definieren ... 1344
24.26.2 Die Zeichenmodi ... 1345
24.26.3 Inhalte laden und sichern ... 1346
24.26.4 Konvertieren in eine Bitmap ... 1346
24.26.5 Weitere Eigenschaften ... 1347
24.27 Ellipse, Rectangle, Line und Co. ... 1348
24.27.1 Ellipse ... 1348
24.27.2 Rectangle ... 1348
24.27.3 Line ... 1349
24.28 Browser ... 1349
24.29 Ribbon ... 1351
24.29.1 Allgemeine Grundlagen ... 1352
24.29.2 Download/Installation ... 1353
24.29.3 Erste Schritte ... 1354
24.29.4 Registerkarten und Gruppen ... 1355
24.29.5 Kontextabhängige Registerkarten ... 1356
24.29.6 Einfache Beschriftungen ... 1356
24.29.7 Schaltflächen ... 1357
24.29.8 Auswahllisten ... 1359
24.29.9 Optionsauswahl ... 1361
24.29.10 Texteingaben ... 1362
24.29.11 Screentips ... 1362
24.29.12 Symbolleiste für den Schnellzugriff ... 1363
24.29.13 Das RibbonWindow ... 1364
24.29.14 Menüs ... 1365
24.29.15 Anwendungsmenü ... 1366
24.29.16 Alternativen ... 1369
24.30 Chart ... 1370
24.31 WindowsFormsHost ... 1371

25 Wichtige WPF-Techniken 1375

25.1 Eigenschaften 1375
25.1.1 Abhängige Eigenschaften (Dependency Properties) 1375
25.1.2 Angehängte Eigenschaften (Attached Properties) 1377
25.2 Einsatz von Ressourcen 1377
25.2.1 Was sind eigentlich Ressourcen? 1377
25.2.2 Wo können Ressourcen gespeichert werden? 1377
25.2.3 Wie definiere ich eine Ressource? 1379
25.2.4 Statische und dynamische Ressourcen 1380
25.2.5 Wie werden Ressourcen adressiert? 1381
25.2.6 System-Ressourcen einbinden 1382
25.3 Das WPF-Ereignis-Modell 1382
25.3.1 Einführung 1382
25.3.2 Routed Events 1383
25.3.3 Direkte Events 1385
25.4 Verwendung von Commands 1386
25.4.1 Einführung zu Commands 1386
25.4.2 Verwendung vordefinierter Commands 1386
25.4.3 Das Ziel des Commands 1388
25.4.4 Vordefinierte Commands 1389
25.4.5 Commands an Ereignismethoden binden 1390
25.4.6 Wie kann ich ein Command per Code auslösen? 1391
25.4.7 Command-Ausführung verhindern 1392
25.5 Das WPF-Style-System 1392
25.5.1 Übersicht 1392
25.5.2 Benannte Styles 1393
25.5.3 Typ-Styles 1394
25.5.4 Styles anpassen und vererben 1395
25.6 Verwenden von Triggern 1398
25.6.1 Eigenschaften-Trigger (Property Triggers) 1398
25.6.2 Ereignis-Trigger 1400
25.6.3 Daten-Trigger 1401
25.7 Einsatz von Templates 1402
25.7.1 Neues Template erstellen 1402
25.7.2 Template abrufen und verändern 1406

25.8 Transformationen, Animationen, StoryBoards ... 1409
25.8.1 Transformationen ... 1409
25.8.2 Animationen mit dem StoryBoard realisieren ... 1415
25.9 Praxisbeispiel ... 1419

26 WPF-Datenbindung ... 1423

26.1 Grundprinzip ... 1423
26.1.1 Bindungsarten ... 1424
26.1.2 Wann eigentlich wird die Quelle aktualisiert? ... 1425
26.1.3 Geht es auch etwas langsamer? ... 1426
26.1.4 Bindung zur Laufzeit realisieren ... 1427
26.2 Binden an Objekte ... 1429
26.2.1 Objekte im XAML-Code instanziieren ... 1429
26.2.2 Verwenden der Instanz im C#-Quellcode ... 1431
26.2.3 Anforderungen an die Quell-Klasse ... 1431
26.2.4 Instanziieren von Objekten per C#-Code ... 1433
26.3 Binden von Collections ... 1434
26.3.1 Anforderung an die Collection ... 1434
26.3.2 Einfache Anzeige ... 1435
26.3.3 Navigieren zwischen den Objekten ... 1436
26.3.4 Einfache Anzeige in einer ListBox ... 1438
26.3.5 DataTemplates zur Anzeigeformatierung ... 1439
26.3.6 Mehr zu List- und ComboBox ... 1440
26.3.7 Verwendung der ListView ... 1442
26.4 Noch einmal zurück zu den Details ... 1444
26.4.1 Navigieren in den Daten ... 1444
26.4.2 Sortieren ... 1446
26.4.3 Filtern ... 1446
26.4.4 Live Shaping ... 1447
26.5 Anzeige von Datenbankinhalten ... 1448
26.5.1 Datenmodell per LINQ to SQL-Designer erzeugen ... 1449
26.5.2 Die Programm-Oberfläche ... 1450
26.5.3 Der Zugriff auf die Daten ... 1451
26.6 Drag & Drop-Datenbindung ... 1452
26.6.1 Vorgehensweise ... 1452
26.6.2 Weitere Möglichkeiten ... 1455
26.7 Formatieren von Werten ... 1456
26.7.1 IValueConverter ... 1457

26.7.2 BindingBase.StringFormat-Eigenschaft ... 1459
26.8 Das DataGrid als Universalwerkzeug ... 1461
26.8.1 Grundlagen der Anzeige ... 1461
26.8.2 UI-Virtualisierung ... 1462
26.8.3 Spalten selbst definieren ... 1462
26.8.4 Zusatzinformationen in den Zeilen anzeigen ... 1464
26.8.5 Vom Betrachten zum Editieren ... 1465
26.9 Praxisbeispiele ... 1465
26.9.1 Collections in Hintergrundthreads füllen ... 1465
26.9.2 Drag & Drop-Bindung bei 1:n-Beziehungen ... 1469

27 Druckausgabe mit WPF ... 1475

27.1 Grundlagen ... 1475
27.1.1 XPS-Dokumente ... 1475
27.1.2 System.Printing ... 1476
27.1.3 System.Windows.Xps ... 1477
27.2 Einfache Druckausgaben mit dem PrintDialog ... 1477
27.3 Mehrseitige Druckvorschau-Funktion ... 1480
27.3.1 Fix-Dokumente ... 1480
27.3.2 Flow-Dokumente ... 1486
27.4 Druckerinfos, -auswahl, -konfiguration ... 1489
27.4.1 Die installierten Drucker bestimmen ... 1490
27.4.2 Den Standarddrucker bestimmen ... 1491
27.4.3 Mehr über einzelne Drucker erfahren ... 1491
27.4.4 Spezifische Druckeinstellungen vornehmen ... 1493

Teil V: Windows Forms

28 Windows Forms-Anwendungen ... 1499

28.1 Grundaufbau/Konzepte ... 1499
28.1.1 Das Hauptprogramm – Program.cs ... 1500
28.1.2 Die Oberflächendefinition – Form1.Designer.cs ... 1504
28.1.3 Die Spielwiese des Programmierers – Form1.cs ... 1505
28.1.4 Die Datei AssemblyInfo.cs ... 1506
28.1.5 Resources.resx/Resources.Designer.cs ... 1507
28.1.6 Settings.settings/Settings.Designer.cs ... 1508
28.1.7 Settings.cs ... 1509

28.2 Ein Blick auf die Application-Klasse 1510
28.2.1 Eigenschaften 1510
28.2.2 Methoden 1511
28.2.3 Ereignisse 1513
28.3 Allgemeine Eigenschaften von Komponenten 1513
28.3.1 Font 1514
28.3.2 Handle 1516
28.3.3 Tag 1517
28.3.4 Modifiers 1517
28.4 Allgemeine Ereignisse von Komponenten 1518
28.4.1 Die Eventhandler-Argumente 1518
28.4.2 Sender 1518
28.4.3 Der Parameter e 1520
28.4.4 Mausereignisse 1520
28.4.5 KeyPreview 1522
28.4.6 Weitere Ereignisse 1523
28.4.7 Validätsprüfungen 1523
28.4.8 SendKeys 1524
28.5 Allgemeine Methoden von Komponenten 1525

29 Windows Forms-Formulare 1527

29.1 Übersicht 1527
29.1.1 Wichtige Eigenschaften des Form-Objekts 1528
29.1.2 Wichtige Ereignisse des Form-Objekts 1530
29.1.3 Wichtige Methoden des Form-Objekts 1531
29.2 Praktische Aufgabenstellungen 1532
29.2.1 Fenster anzeigen 1532
29.2.2 Splash Screens beim Anwendungsstart anzeigen 1535
29.2.3 Eine Sicherheitsabfrage vor dem Schließen anzeigen 1537
29.2.4 Ein Formular durchsichtig machen 1538
29.2.5 Die Tabulatorreihenfolge festlegen 1539
29.2.6 Ausrichten von Komponenten im Formular 1539
29.2.7 Spezielle Panels für flexible Layouts 1542
29.2.8 Menüs erzeugen 1543
29.3 MDI-Anwendungen 1547
29.3.1 "Falsche" MDI-Fenster bzw. Verwenden von Parent 1547
29.3.2 Die echten MDI-Fenster 1548
29.3.3 Die Kindfenster 1549

29.3.4 Automatisches Anordnen der Kindfenster 1550
29.3.5 Zugriff auf die geöffneten MDI-Kindfenster 1551
29.3.6 Zugriff auf das aktive MDI-Kindfenster 1552
29.3.7 Mischen von Kindfenstermenü/MDIContainer-Menü 1552
29.4 Praxisbeispiele 1553
29.4.1 Informationsaustausch zwischen Formularen 1553
29.4.2 Ereigniskette beim Laden/Entladen eines Formulars 1560

30 Windows Forms-Komponenten 1567

30.1 Allgemeine Hinweise 1567
30.1.1 Hinzufügen von Komponenten 1567
30.1.2 Komponenten zur Laufzeit per Code erzeugen 1568
30.2 Allgemeine Steuerelemente 1570
30.2.1 Label 1570
30.2.2 LinkLabel 1571
30.2.3 Button 1572
30.2.4 TextBox 1573
30.2.5 MaskedTextBox 1576
30.2.6 CheckBox 1577
30.2.7 RadioButton 1579
30.2.8 ListBox 1579
30.2.9 CheckedListBox 1581
30.2.10 ComboBox 1581
30.2.11 PictureBox 1582
30.2.12 DateTimePicker 1583
30.2.13 MonthCalendar 1583
30.2.14 HScrollBar, VScrollBar 1584
30.2.15 TrackBar 1585
30.2.16 NumericUpDown 1585
30.2.17 DomainUpDown 1586
30.2.18 ProgressBar 1586
30.2.19 RichTextBox 1587
30.2.20 ListView 1588
30.2.21 TreeView 1594
30.2.22 WebBrowser 1599
30.3 Container 1600
30.3.1 FlowLayout/TableLayout/SplitContainer 1600
30.3.2 Panel 1600

30.3.3 GroupBox ... 1601
30.3.4 TabControl ... 1601
30.3.5 ImageList ... 1603
30.4 Menüs & Symbolleisten ... 1604
30.4.1 MenuStrip und ContextMenuStrip ... 1604
30.4.2 ToolStrip ... 1605
30.4.3 StatusStrip ... 1605
30.4.4 ToolStripContainer ... 1605
30.5 Daten ... 1606
30.5.1 DataSet ... 1606
30.5.2 DataGridView/DataGrid ... 1606
30.5.3 BindingNavigator/BindingSource ... 1606
30.5.4 Chart ... 1607
30.6 Komponenten ... 1608
30.6.1 ErrorProvider ... 1608
30.6.2 HelpProvider ... 1608
30.6.3 ToolTip ... 1608
30.6.4 BackgroundWorker ... 1608
30.6.5 Timer ... 1609
30.6.6 SerialPort ... 1609
30.7 Drucken ... 1609
30.7.1 PrintPreviewControl ... 1609
30.7.2 PrintDocument ... 1609
30.8 Dialoge ... 1610
30.8.1 OpenFileDialog/SaveFileDialog/FolderBrowserDialog ... 1610
30.8.2 FontDialog/ColorDialog ... 1610
30.9 WPF-Unterstützung mit dem ElementHost ... 1610
30.10 Praxisbeispiele ... 1611
30.10.1 Mit der CheckBox arbeiten ... 1611
30.10.2 Steuerelemente per Code selbst erzeugen ... 1612
30.10.3 Controls-Auflistung im TreeView anzeigen ... 1615
30.10.4 WPF-Komponenten mit dem ElementHost anzeigen ... 1618

31 Grundlagen Grafikausgabe ... 1623

31.1 Übersicht und erste Schritte ... 1623
31.1.1 GDI+ – Ein erster Einblick für Umsteiger ... 1624
31.1.2 Namespaces für die Grafikausgabe ... 1625

31.2 Darstellen von Grafiken 1627
31.2.1 Die PictureBox-Komponente 1627
31.2.2 Das Image-Objekt 1628
31.2.3 Laden von Grafiken zur Laufzeit 1629
31.2.4 Sichern von Grafiken 1629
31.2.5 Grafikeigenschaften ermitteln 1630
31.2.6 Erzeugen von Vorschaugrafiken (Thumbnails) 1631
31.2.7 Die Methode RotateFlip 1632
31.2.8 Skalieren von Grafiken 1633
31.3 Das .NET-Koordinatensystem 1634
31.3.1 Globale Koordinaten 1635
31.3.2 Seitenkoordinaten (globale Transformation) 1636
31.3.3 Gerätekoordinaten (Seitentransformation) 1638
31.4 Grundlegende Zeichenfunktionen von GDI+ 1639
31.4.1 Das zentrale Graphics-Objekt 1639
31.4.2 Punkte zeichnen/abfragen 1642
31.4.3 Linien 1643
31.4.4 Kantenglättung mit Antialiasing 1644
31.4.5 PolyLine 1645
31.4.6 Rechtecke 1645
31.4.7 Polygone 1647
31.4.8 Splines 1648
31.4.9 Bézierkurven 1649
31.4.10 Kreise und Ellipsen 1650
31.4.11 Tortenstück (Segment) 1650
31.4.12 Bogenstück 1652
31.4.13 Wo sind die Rechtecke mit den runden Ecken? 1653
31.4.14 Textausgabe 1654
31.4.15 Ausgabe von Grafiken 1658
31.5 Unser Werkzeugkasten 1659
31.5.1 Einfache Objekte 1659
31.5.2 Vordefinierte Objekte 1661
31.5.3 Farben/Transparenz 1663
31.5.4 Stifte (Pen) 1664
31.5.5 Pinsel (Brush) 1667
31.5.6 SolidBrush 1668
31.5.7 HatchBrush 1668
31.5.8 TextureBrush 1669

31.5.9 LinearGradientBrush ... 1670
31.5.10 PathGradientBrush ... 1671
31.5.11 Fonts ... 1672
31.5.12 Path-Objekt ... 1673
31.5.13 Clipping/Region ... 1676
31.6 Standarddialoge ... 1680
31.6.1 Schriftauswahl ... 1680
31.6.2 Farbauswahl ... 1681
31.7 Praxisbeispiele ... 1683
31.7.1 Ein Graphics-Objekt erzeugen ... 1683
31.7.2 Zeichenoperationen mit der Maus realisieren ... 1685

32 Druckausgabe ... 1689

32.1 Einstieg und Übersicht ... 1689
32.1.1 Nichts geht über ein Beispiel ... 1689
32.1.2 Programmiermodell ... 1691
32.1.3 Kurzübersicht der Objekte ... 1692
32.2 Auswerten der Druckereinstellungen ... 1692
32.2.1 Die vorhandenen Drucker ... 1692
32.2.2 Der Standarddrucker ... 1693
32.2.3 Verfügbare Papierformate/Seitenabmessungen ... 1694
32.2.4 Der eigentliche Druckbereich ... 1695
32.2.5 Die Seitenausrichtung ermitteln ... 1696
32.2.6 Ermitteln der Farbfähigkeit ... 1696
32.2.7 Die Druckauflösung abfragen ... 1696
32.2.8 Ist beidseitiger Druck möglich? ... 1697
32.2.9 Einen "Informationsgerätekontext" erzeugen ... 1697
32.2.10 Abfragen von Werten während des Drucks ... 1698
32.3 Festlegen von Druckereinstellungen ... 1699
32.3.1 Einen Drucker auswählen ... 1699
32.3.2 Drucken in Millimetern ... 1699
32.3.3 Festlegen der Seitenränder ... 1700
32.3.4 Druckjobname ... 1701
32.3.5 Anzahl der Kopien ... 1702
32.3.6 Beidseitiger Druck ... 1702
32.3.7 Seitenzahlen festlegen ... 1703
32.3.8 Druckqualität verändern ... 1706
32.3.9 Ausgabemöglichkeiten des Chart-Controls nutzen ... 1707

32.4 Die Druckdialoge verwenden ... 1707
32.4.1 PrintDialog ... 1708
32.4.2 PageSetupDialog ... 1709
32.4.3 PrintPreviewDialog ... 1711
32.4.4 Ein eigenes Druckvorschau-Fenster realisieren ... 1712
32.5 Drucken mit OLE-Automation ... 1713
32.5.1 Kurzeinstieg in die OLE-Automation ... 1713
32.5.2 Drucken mit Microsoft Word ... 1715
32.6 Praxisbeispiele ... 1717
32.6.1 Den Drucker umfassend konfigurieren ... 1717
32.6.2 Diagramme mit dem Chart-Control drucken ... 1727
32.6.3 Druckausgabe mit Word ... 1729

33 Windows Forms-Datenbindung ... 1735

33.1 Prinzipielle Möglichkeiten ... 1735
33.2 Manuelle Bindung an einfache Datenfelder ... 1735
33.2.1 BindingSource erzeugen ... 1736
33.2.2 Binding-Objekt ... 1736
33.2.3 DataBindings-Collection ... 1737
33.3 Manuelle Bindung an Listen und Tabellen ... 1737
33.3.1 DataGridView ... 1737
33.3.2 Datenbindung von ComboBox und ListBox ... 1738
33.4 Entwurfszeit-Bindung an typisierte DataSets ... 1738
33.5 Drag & Drop-Datenbindung ... 1740
33.6 Navigations- und Bearbeitungsfunktionen ... 1740
33.6.1 Navigieren zwischen den Datensätzen ... 1740
33.6.2 Hinzufügen und Löschen ... 1740
33.6.3 Aktualisieren und Abbrechen ... 1741
33.6.4 Verwendung des BindingNavigators ... 1741
33.7 Die Anzeigedaten formatieren ... 1742
33.8 Praxisbeispiele ... 1742
33.8.1 Einrichten und Verwenden einer Datenquelle ... 1742
33.8.2 Eine Auswahlabfrage im DataGridView anzeigen ... 1746
33.8.3 Master-Detailbeziehungen im DataGrid anzeigen ... 1749
33.8.4 Datenbindung Chart-Control ... 1750

34 Erweiterte Grafikausgabe 1755
34.1 Transformieren mit der Matrix-Klasse 1755
34.1.1 Übersicht 1755
34.1.2 Translation 1756
34.1.3 Skalierung 1756
34.1.4 Rotation 1757
34.1.5 Scherung 1757
34.1.6 Zuweisen der Matrix 1758
34.2 Low-Level-Grafikmanipulationen 1758
34.2.1 Worauf zeigt Scan0? 1759
34.2.2 Anzahl der Spalten bestimmen 1760
34.2.3 Anzahl der Zeilen bestimmen 1761
34.2.4 Zugriff im Detail (erster Versuch) 1761
34.2.5 Zugriff im Detail (zweiter Versuch) 1763
34.2.6 Invertieren 1765
34.2.7 In Graustufen umwandeln 1766
34.2.8 Heller/Dunkler 1767
34.2.9 Kontrast 1769
34.2.10 Gamma-Wert 1770
34.2.11 Histogramm spreizen 1770
34.2.12 Ein universeller Grafikfilter 1773
34.3 Fortgeschrittene Techniken 1777
34.3.1 Flackerfrei dank Double Buffering 1777
34.3.2 Animationen 1779
34.3.3 Animated GIFs 1782
34.3.4 Auf einzelne GIF-Frames zugreifen 1784
34.3.5 Transparenz realisieren 1786
34.3.6 Eine Grafik maskieren 1787
34.3.7 JPEG-Qualität beim Sichern bestimmen 1789
34.4 Grundlagen der 3D-Vektorgrafik 1790
34.4.1 Datentypen für die Verwaltung 1790
34.4.2 Eine universelle 3D-Grafik-Klasse 1791
34.4.3 Grundlegende Betrachtungen 1792
34.4.4 Translation 1795
34.4.5 Streckung/Skalierung 1796
34.4.6 Rotation 1797
34.4.7 Die eigentlichen Zeichenroutinen 1799

34.5 Und doch wieder GDI-Funktionen ... 1801
34.5.1 Am Anfang war das Handle ... 1801
34.5.2 Gerätekontext (Device Context Types) 1804
34.5.3 Koordinatensysteme und Abbildungsmodi 1806
34.5.4 Zeichenwerkzeuge/Objekte 1810
34.5.5 Bitmaps 1812
34.6 Praxisbeispiele 1816
34.6.1 Die Transformationsmatrix verstehen 1816
34.6.2 Eine 3D-Grafikausgabe in Aktion 1819
34.6.3 Einen Fenster-Screenshot erzeugen 1822

35 Ressourcen/Lokalisierung 1825

35.1 Manifestressourcen 1825
35.1.1 Erstellen von Manifestressourcen 1825
35.1.2 Zugriff auf Manifestressourcen 1827
35.2 Typisierte Ressourcen 1829
35.2.1 Erzeugen von .resources-Dateien 1829
35.2.2 Hinzufügen der .resources-Datei zum Projekt 1829
35.2.3 Zugriff auf die Inhalte von .resources-Dateien 1830
35.2.4 ResourceManager einer .resources-Datei erzeugen 1830
35.2.5 Was sind .resx-Dateien? 1831
35.3 Streng typisierte Ressourcen 1831
35.3.1 Erzeugen streng typisierter Ressourcen 1832
35.3.2 Verwenden streng typisierter Ressourcen 1832
35.3.3 Streng typisierte Ressourcen per Reflection auslesen 1833
35.4 Anwendungen lokalisieren 1835
35.4.1 Localizable und Language 1835
35.4.2 Beispiel "Landesfahnen" 1835

36 Komponentenentwicklung 1841

36.1 Überblick 1841
36.2 Benutzersteuerelement 1842
36.2.1 Entwickeln einer Auswahl-ListBox 1842
36.2.2 Komponente verwenden 1844
36.3 Benutzerdefiniertes Steuerelement 1845
36.3.1 Entwickeln eines BlinkLabels 1845
36.3.2 Verwenden der Komponente 1848
36.4 Komponentenklasse 1848

36.5 Eigenschaften ... 1849
36.5.1 Einfache Eigenschaften ... 1849
36.5.2 Schreib-/Lesezugriff (Get/Set) ... 1849
36.5.3 Nur Lese-Eigenschaft (ReadOnly) ... 1850
36.5.4 Nur-Schreibzugriff (WriteOnly) ... 1851
36.5.5 Hinzufügen von Beschreibungen ... 1851
36.5.6 Ausblenden im Eigenschaftenfenster ... 1851
36.5.7 Einfügen in Kategorien ... 1852
36.5.8 Default-Wert einstellen ... 1852
36.5.9 Standard-Eigenschaft (Indexer) ... 1853
36.5.10 Wertebereichsbeschränkung und Fehlerprüfung ... 1853
36.5.11 Eigenschaften von Aufzählungstypen ... 1855
36.5.12 Standard Objekt-Eigenschaften ... 1856
36.6 Methoden ... 1858
36.6.1 Konstruktor ... 1859
36.6.2 Class-Konstruktor ... 1860
36.6.3 Destruktor ... 1861
36.6.4 Aufruf des Basisklassen-Konstruktors ... 1861
36.6.5 Aufruf von Basisklassen-Methoden ... 1862
36.7 Ereignisse (Events) ... 1862
36.7.1 Ereignis mit Standardargument definieren ... 1862
36.7.2 Ereignis mit eigenen Argumenten ... 1864
36.7.3 Ein Default-Ereignis festlegen ... 1865
36.7.4 Mit Ereignissen auf Windows-Messages reagieren ... 1865
36.8 Weitere Themen ... 1867
36.8.1 Wohin mit der Komponente? ... 1867
36.8.2 Assembly-Informationen festlegen ... 1868
36.8.3 Assemblies signieren ... 1871
36.8.4 Komponenten-Ressourcen einbetten ... 1871
36.8.5 Der Komponente ein Icon zuordnen ... 1872
36.8.6 Den Designmodus erkennen ... 1873
36.9 Praxisbeispiele ... 1877
36.9.1 AnimGif – Anzeige von Animationen ... 1877
36.9.2 Eine FontComboBox entwickeln ... 1880
36.9.3 Das PropertyGrid verwenden ... 1882

Index ... 1885

Vorwort

C# ist eine noch eine relativ junge Sprache, sie bietet die Möglichkeiten und Flexibilität von C++ und erlaubt trotzdem eine schnelle und unkomplizierte Programmierpraxis wie 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- , WinRT- und Silverlight-Anwendungen bis hin zu systemnahen Applikationen.

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

- 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# 2015 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# 2015 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 haben wir in fünf Themenkomplexen gruppiert:

1. Grundlagen der Programmierung mit C#
2. Technologien
3. Windows Store Apps
4. WPF-Anwendungen
5. Windows Forms-Anwendungen

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.

Im gedruckten Teil dieses Buchs finden Sie die ersten drei Themenkomplexe, denn bereits hier sind wir an die Grenze des drucktechnisch Machbaren gestoßen. Die übrigen zwei Themenkomplexe mussten wir als PDF auslagern, welche Sie sich kostenlos aus dem Internet herunterladen können.

Zu den Codebeispielen

Alle Beispieldaten dieses Buchs und die Kapitel des vierten und fünften Teils können Sie sich unter der folgenden Adresse herunterladen:

LINK: `http://www.doko-buch.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 sowie der Microsoft Internet Information Server (ASP.NET) erforderlich.
- Um mit den WinRT-Projekten arbeiten zu können, müssen Sie Visual Studio unter Windows 8 bzw. 10 ausführen und das Windows SDK installiert haben.
- Beachten Sie die zu einigen Beispielen beigefügten *Liesmich.txt*-Dateien, die Sie auf besondere Probleme hinweisen.

Nobody is perfect

Sie werden – trotz der rund 1900 Seiten – in diesem Buch nicht alles finden, was Visual C# 2015 bzw. das .NET Framework 4.6 zu bieten haben. Manches ist sicher in einem anderen Spezialtitel noch besser oder ausführlicher beschrieben. Aber Sie halten mit unserem Buch einen optimalen und 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 über unsere Autoren-Website kontaktieren:

LINK: `http://www.doko-buch.de`

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.

Walter Doberenz und Thomas Gewinnus *Wintersdorf/Frankfurt/O., im August 2015*

Kapitel 3

OOP-Konzepte

C# erlaubt es Ihnen, bereits ohne fundierte OOP-Kenntnisse objektorientiert zu programmieren! Davon haben Sie bereits vor der Lektüre dieses Kapitels, mehr oder weniger unbewusst, Gebrauch gemacht: Sie haben Ereignisbehandlungsroutinen (Eventhandler) geschrieben und den Objekten der visuellen Benutzerschnittstelle (Form, Steuerelemente) Eigenschaften zugewiesen bzw. deren Methoden aufgerufen.

Die Entwicklungsumgebung von Visual Studio erlaubt objektorientiertes Programmieren bereits mit einem Minimum an Vorkenntnissen. Das vorliegende Kapitel will etwas tiefer in die OOP-Problematik eindringen und präsentiert Ihnen neben einigen grundlegenden Ausführungen die für den Einstieg wichtigsten objektspezifischen Features von C# im Überblick.

HINWEIS: In C# haben Sie grundsätzlich die Möglichkeit, zwischen Klassen (sind Verweistypen) und Strukturen (sind Wertetypen) zu wählen. Strukturen (*struct*) wurden bereits im Sprachkapitel (Abschnitt 2.6.2) einführend behandelt, bieten allerdings noch weitaus mehr Möglichkeiten, die fast an die von Klassen heranreichen. Wir aber wollen uns im vorliegenden Kapitel ausschließlich mit Klassen und Objekten, den Grundelementen der OOP, beschäftigen.

3.1 Kleine Einführung in die OOP

In .NET ist alles ein Objekt! Viele Entwickler – insbesondere wenn sie mit "altem" Code zu kämpfen haben – tun sich immer noch ziemlich schwer mit OOP, weil ihnen die Komplexität einer vollständigen Anwendung zu hoch erscheint.

3.1.1 Historische Entwicklung

Im Unterschied zur objektorientierten ist die klassische strukturierte Programmierung ziemlich sprachunabhängig und hatte Zeit genug, um auch in den letzten Winkel der Programmierwelt vorzudringen.

Demgegenüber stand es um die Akzeptanz der objektorientierten Programmierung bis Anbruch des .NET-Zeitalters zu Beginn dieses Jahrtausends noch nicht zum Besten, das aber hat sich seitdem dramatisch geändert.

Strukturierte Programmierung

Gern bezeichnet man die strukturierte Programmierung auch als Vorläufer der objektorientierten Programmierung, obwohl dieser Vergleich hinkt. Richtig ist, dass sowohl strukturierte als auch objektorientierte Programmierung fundamentale Denkmuster[1] sind, die gleichberechtigt nebeneinander existieren.

Die Grundkonzepte der strukturierten Programmierung wurden beginnend mit dem Ende der Sechzigerjahre entwickelt und lassen sich mit folgenden Stichwörtern charakterisieren: hierarchische Programmorganisation, logische Programmeinheiten, zentrale Programmsteuerung, beschränkte Datenverfügbarkeit.

Ziel der strukturierten Programmierung ist es, Algorithmen so darzustellen, dass ihr Ablauf einfach zu erfassen und zu verändern ist.

Gegenstand der strukturierten Programmierung ist also die bestmögliche Anordnung von Code, um dessen Transparenz, Testbarkeit und Wiederverwendbarkeit zu maximieren[2].

Dass C# eine konsequent objektorientierte Sprache ist, bedeutet noch lange nicht, dass man damit nicht auch strukturiert programmieren könnte, im Gegenteil. Im Kapitel 2, wo sich alles um die grundlegenden sprachlichen Elemente von C# dreht, haben wir uns fast ausschließlich auf dem Boden der traditionellen strukturierten Programmierung bewegt und versucht, die OOP noch weitestgehend auszuklammern.

So haben wir es größtenteils ignoriert, dass selbst die einfachen Datentypen Objekte sind, und haben z.B. statt mit Methoden mit Funktionen und Prozeduren und statt mit Klassen mit strukturierten Datentypen (*struct*) gearbeitet. Tatsächlich können Sie aber mit OOP alles machen, was auch die strukturierte Programmierung erlaubt. Statt beispielsweise globale Variablen in einem Modul zu deklarieren, können Sie statische Klasseneigenschaften verwenden.

Um fit für die aktuellen Herausforderungen zu sein, sollten Sie deshalb – wo immer es vertretbar ist – nach objektorientierten Lösungen streben.

Objektorientiertes Programmieren

Die objektorientierte Programmierung entfaltete auf breiter Basis erst seit Ende der 80er-Jahre mit dem Beginn des Windows-Zeitalters ihre Wirkung. Sehr bekannte Vertreter objektorientierter Sprachen sind C++, Java, Smalltalk und Borland Delphi – aber auch das alte Visual Basic war bereits in vielen wesentlichen Zügen objektorientiert aufgebaut[3].

---

[1] Im Fachjargon heißt das "Paradigma".

[2] Bahnbrechendes auf diesem Gebiet leistete Prof. Niklaus Wirth mit seinen Sprachen Pascal und Modula.

[3] Mit VB.NET wurde auch Visual Basic endlich vollwertiges Mitglied der OOP-Welt.

Objektorientierte Programmierung ist ein Denkmuster, bei dem Programme als Menge von über Nachrichten kooperierenden Objekten organisiert werden und jedes Objekt Instanz einer Klasse ist.

Im Unterschied zur strukturierten Programmierung bedeutet "objektorientiert" also, dass Daten und Algorithmen nicht mehr nebeneinander existieren, sondern in Objekten zusammengefasst sind.

Während Module in der strukturierten Programmierung zwar auch Daten und Code zusammenfassen, stellen Klassen jetzt Vorlagen dar, von denen immer neue Kopien (Instanzen) angefertigt werden können. Diese Instanzen, d.h. die Objekte, kapseln den Zugriff auf die enthaltenen Daten hinter Schnittstellen (Interfaces).

Der große Vorteil der OOP ist ihre Ähnlichkeit mit den menschlichen Denkstrukturen. Dadurch wird vor allem dem Einsteiger, der bisher über keine bzw. wenig Programmiererfahrung verfügt, das Verständnis der OOP erleichtert.

HINWEIS: Die OOP verlangt eine Anpassung des Software-Entwicklungsprozesses und der eingesetzten Methoden an den Denkstil des Programmierers – nicht umgekehrt!

Die OOP ist eine der wenigen Fälle, in denen der Einsteiger gegenüber dem Profi zumindest einen kleinen Vorteil besitzt: Er ist noch nicht in der Denkweise klassischer Programmiersprachen gefangen, die dazu erziehen, in Abläufen zu denken, bei denen die in der realen Welt zu beobachtenden Abläufe Schritt für Schritt in Algorithmen umgesetzt werden, etwa um betriebliche Prozesse per Programm zu automatisieren.

Die OOP entspricht hingegen der üblichen menschlichen Denkweise, indem sie z.B. reale Objekte aus der abzubildenden Umwelt identifiziert und in ihrer Art beschreibt.

3.1.2 Grundbegriffe der OOP

Bereits im Kapitel 1hatten Sie gesehen, dass Objekte durch Eigenschaften, Methoden und Ereignisse beschrieben werden. Auf diese und andere Weise überwindet das Konzept der objektorientierten Programmierung (OOP) den prozeduralen Ansatz der klassischen strukturellen Programmierung zugunsten einer realitätsnahen Modellierung.

Bevor wir uns den Details zuwenden, sollen die wichtigsten Begriffe der objektorientierten Programmierung (OOP) zunächst allgemein, d.h. ohne Bezug auf eine konkrete Programmiersprache, erörtert werden.

Objekt

Der Programmierer versteht unter einem *Objekt* die Zusammenfassung (Kapselung) von Daten und zugehörigen Funktionalitäten. Ein solches Softwareobjekt wird auch oft benutzt, um Dinge des täglichen Lebens für Zwecke der Datenverarbeitung abzubilden. Aber das ist nur ein Aspekt, denn Objekte sind ganz allgemein Dinge, die Sie in Ihrem Code beschreiben wollen, es sind Gruppen von Eigenschaften, Methoden und Ereignissen, die logisch zusammengehören. Als Programmierer arbeiten Sie mit einem Objekt, indem Sie dessen Eigenschaften und Methoden manipulieren und auf seine Ereignisse reagieren.

Klasse

Eine *Klasse* ist nicht mehr und nicht weniger als ein "Bauplan", auf dessen Grundlage die entsprechenden Objekte zur Programmlaufzeit erzeugt werden. Gewissermaßen als Vorlage (Prägestempel) für das Objekt legt die Klasse fest, wie das Objekt auszusehen hat und wie es sich verhalten soll. Es handelt sich bei einer Klasse also um eine reine Softwarekonstruktion, die Eigenschaften, Methoden und Ereignisse eines Objekts definiert, ohne das Objekt zu erzeugen.

HINWEIS: Oft wird anstatt des Begriffs "Klasse" mit völlig gleichwertiger Bedeutung auch "Objekttyp" verwendet.

Instanz

Man erhält erst dann ein konkretes Objekt, wenn man eine *Instanz* einer Klasse bildet. Es lassen sich viele Objekte mit einer einzigen Klassendefinition erzeugen.

BEISPIEL 3.1: Instanz

Auf dem Montageband werden zahlreiche Auto-Objekte nach ein und denselben Konstruktionsvorschriften für die Klasse *Auto* gebaut. Diesen Vorgang könnte man auch als Bildung von Instanzen der Klasse *Auto* bezeichnen. Während die Klasse lediglich die Eigenschaft *Farbe* definiert, wird der konkrete Wert (rot, blau, grün ...) erst beim Erzeugen des Objekts (der Instanz) zugewiesen.

Kapselung

Klassen realisieren das Prinzip der *Kapselung* von Objekten, das es ermöglicht, die Implementierung der Klasse (der Code im Inneren) von deren Schnittstelle bzw. Interface (die öffentlichen Eigenschaften, Methoden und Ereignisse) sauber zu trennen. Durch das Verbergen der inneren Struktur werden die internen Daten und einige verborgene Methoden geschützt, sind also von außen nicht zugänglich. Die Manipulation des Objekts kann lediglich über streng definierte, über die Schnittstelle zur Verfügung gestellte öffentliche Methoden erfolgen.

Wiederverwendbarkeit

Klassen ermöglichen die *Wiederverwendbarkeit* von Code. Nachdem eine Klasse geschrieben wurde, können Sie diese an verschiedenen Stellen innerhalb einer Applikation verwenden. Klassen reduzieren somit den redundanten Code einer Anwendung, sie erleichtern außerdem die Wartung des Codes.

Vererbung

Echte Vererbung (*Implementierungsvererbung*) ermöglicht es Klassen zu definieren, die von anderen Klassen abgeleitet werden, wobei nicht nur die Schnittstelle, sondern auch der dahinter liegende Code (die Implementierung) vom Nachkommen übernommen wird.

Da es nun möglich ist, die Implementierung einer Klasse für weitere Klassen als Grundlage zu verwenden, kann man Unterklassen bilden, die alle Eigenschaften und Methoden ihrer Oberklasse erben. Diese Unterklassen können zu den geerbten Eigenschaften neue hinzufügen oder Eigenschaften der Oberklasse verstecken, indem sie diese überschreiben.

Wird von einer solchen Unterklasse ein Objekt erzeugt (also eine Instanz der Unterklasse gebildet), dann dient für dieses Objekt sowohl die Ober- als auch die Unterklasse als "Bauplan".

C# unterstützt das Überschreiben (*Overriding*) von Methoden[1] der Oberklasse mit alternativen Methoden der Unterklasse (siehe Abschnitt 3.6.2).

Polymorphie

OOP macht es möglich, ein und dieselbe Methode für ganz verschiedene Objekte zu verwenden, man nennt dies dann *Polymorphie* (Vielgestaltigkeit). Jedes dieser Objekte kann die Ausführung unterschiedlich realisieren. Für das aufrufende Objekt bleibt der Vorgang trotzdem derselbe.

BEISPIEL 3.2: Polymorphie

Die Methode *Beschleunigen* ist in einer *Fahrzeug*-Klasse definiert, welche an die Unterklassen *Auto* und *Fahrrad* vererbt. Es ist klar, dass diese Methoden in beiden Unterklassen überschrieben, d.h. völlig unterschiedlich implementiert werden müssen.

Als Polymorphie, die aufs Engste mit der Vererbung verknüpft ist, kann man also die Fähigkeit von Unterklassen bezeichnen, Eigenschaften und Methoden mit dem gleichen Namen, aber mit unterschiedlichen Implementierungen aufzurufen (siehe Abschnitt 3.6.6).

3.1.3 Sichtbarkeit von Klassen und ihren Mitgliedern

Um die Klasse bzw. ihre Mitglieder (Member, Elemente) gezielt zu verbergen oder offen zu legen, sollten Sie von den Zugriffsmodifizierern Gebrauch machen, die den Gültigkeitsbereich (bzw. die *Sichtbarkeit*) einschränken.

Klassen

Die folgende Tabelle zeigt die möglichen Einschränkungen bei der Sichtbarkeit von Klassen:

Modifizierer	Sichtbarkeit
public	Unbeschränkt. Auch von anderen Assemblierungen aus können Objekte der Klasse erstellt werden.
internal	Nur innerhalb des aktuellen Projekts. Außerhalb des Projekts ist kein Objekt dieser Klasse erstellbar. Gilt als Standard, falls kein Modifizierer vorangestellt wird.
private	Nur innerhalb einer anderen Klasse.

[1] Nicht zu verwechseln mit dem Überladen (Overloading) von Methoden (siehe 3.3.2).

Klassenmitglieder

Die folgende Tabelle zeigt die Zugriffsmöglichkeiten auf die Klassenmitglieder (Member).

Modifizierer	Sichtbarkeit
public	Unbeschränkt.
protected	Innerhalb der Klasse und der daraus abgeleiteten Klassen.
internal	Innerhalb des aktuellen Projekts.
internal protected	Innerhalb des aktuellen Projekts oder der abgeleiteten Klassen.
private	Nur innerhalb der Klasse.

Die Schlüsselwörter *private* und *public* definieren immer die beiden Extreme des Zugriffs. Betrachtet man die jeweiligen Klassen als allein stehend, so reichen diese beiden Zugriffsarten völlig aus. Mit solchen, quasi isolierten, Klassen lassen sich allerdings keine komplexeren Probleme lösen.

Um einzelne Klassen miteinander zu verbinden, verwenden Sie den mächtigen Mechanismus der Vererbung. In diesem Zusammenhang gewinnt die *protected*-Deklaration wie folgt an Bedeutung:

- Da eine abgeleitete Klasse auf die *protected*-Member zugreifen kann, sind diese Member für die abgeleitete Klasse quasi *public*.
- Ist eine Klasse nicht von einer anderen abgeleitet, kann sie nicht auf deren *protected*-Member zugreifen, da diese dann quasi *private* sind.

Mehr zu diesem Thema finden Sie im Abschnitt 3.6 (Vererbung).

3.1.4 Allgemeiner Aufbau einer Klasse

Bevor der Einsteiger seine erste Klasse schreibt, sollte er sich zunächst im einführenden Sprachkapitel 2 mit den Strukturen (*struct*, siehe Abschnitt 2.6.2) anfreunden, die in Aufbau und Anwendung starke Ähnlichkeiten zu Klassen aufweisen[1]. Auch im Aufbau von Methoden sollte er sich auskennen (Abschnitt 2.7).

Im Unterschied zu einer Struktur (Schlüsselwort *struct*) wird eine Klasse mit dem Schlüsselwort *class* deklariert. Die (stark vereinfachte) Syntax:

SYNTAX:
```
Modifizierer class Bezeichner
{
  // ... Felder
  // ... Konstruktoren
  // ... Eigenschaften
  // ... Methoden
  // ... Ereignisse
}
```

[1] Der wesentliche Unterschied ist der, dass Strukturen Wertetypen, Klassen hingegen Referenztypen sind.

Zur Bedeutung der (Zugriffs-)*Modifizierer* wird auf obige Tabellen verwiesen.

Im Klassenkörper haben es wir es mit "Klassenmitgliedern" (Member) wie Feldern, Konstruktoren, Eigenschaften, Methoden und Ereignissen zu tun, auf die wir noch detailliert zu sprechen kommen werden.

Die Definition der Klassenmitglieder bezeichnet man auch als *Implementation* der Klasse.

BEISPIEL 3.3: Eine einfache Klasse *CKunde* wird deklariert und implementiert.

C#

```
public class CKunde
{
   private string _anrede;         //     Feld
   private string _name;           //     dto.

  public CKunde(string anr, string nam)       // Konstruktor
  {
    _anrede = anr;
    _name = nam;
  }

  public string name            // Eigenschaft
    {
        get {return(_name); }
        set {_name = value; }
    }

    public string adresse()             // Methode
    {
        string s = _anrede + " " + _name;
        return(s);
    }
  }
```

Unsere Klasse verfügt damit über zwei Felder, eine Eigenschaft und eine Methode. Da die beiden Felder mit dem *private*-Modifizierer deklariert wurden, sind sie von außen nicht sichtbar.

3.1.5 Das Erzeugen eines Objekts

Existiert eine Klasse, so steht dem Erzeugen von Objektvariablen nichts mehr im Weg. Eine Objektvariable ist ein Verweistyp, sie enthält also nicht das Objekt selbst, sondern stellt lediglich einen Zeiger (Adresse) auf den Speicherbereich des Objekts bereit. Es können sich also durchaus mehrere Objektvariablen auf ein und dasselbe Objekt beziehen. Wenn eine Objektvariable den Wert *null* enthält, bedeutet das, dass sie momentan "ins Leere" zeigt, also kein Objekt referenziert.

Unter der Voraussetzung, dass eine gültige Klasse existiert, verläuft der Lebenszyklus eines Objekts in Ihrem Programm in folgenden Etappen:

- Referenzierung (eine Objektvariable wird deklariert, sie verweist momentan noch auf *null*)
- Instanziierung (die Objektvariable zeigt jetzt auf einen konkreten Speicherplatzbereich)
- Initialisierung (die Datenfelder der Objektvariablen werden mit Anfangswerten gefüllt)
- Arbeiten mit dem Objekt (es wird auf Eigenschaften und Methoden des Objekts zugegriffen, Ereignisse werden ausgelöst)
- Zerstören des Objekts (das Objekt wird dereferenziert, der belegte Speicherplatz wird wieder freigegeben)

Werfen wir nun einen genaueren Blick auf die einzelnen Etappen.

Referenzieren und Instanziieren

Es stehen zwei Varianten zur Verfügung.

In *Variante 1* wird pro Schritt eine Anweisung verwendet:

SYNTAX:
```
Modifizierer Klasse Object;
Object = new Klasse(Parameter);
```

BEISPIEL 3.4: Ein Objekt *kunde1* wird referenziert und erzeugt.

```
private CKunde kunde1;                    // Referenzieren
kunde1 = new CKunde1();                   // Erzeugen
```

In *Variante2*, der Kurzform, sind beide Schritte in einer Anweisung zusammengefasst, d.h., das Objekt wird zusammen mit seiner Deklaration erzeugt.

SYNTAX:
```
Klasse Object = new Klasse();
```

BEISPIEL 3.5: Das Äquivalent zum Vorgängerbeispiel.

```
private CKunde kunde1 = new CKunde();
```

Dem Klassenbezeichner (*Klasse*) müsste genauer genommen noch der Name der Klassenbibliothek (bzw. Name des Projekts) vorangestellt werden, doch dies wird unter Visual Studio nicht erforderlich sein, da der entsprechende Namensraum (*Namespace*) bereits automatisch eingebunden wurde (*using*-Anweisung).

Obwohl die Kurzform sehr eindrucksvoll ist, können Sie hier keine Fehlerbehandlung (*try...catch*-Block) durchführen. Diese Einschränkung macht diese Art von Deklaration weniger nützlich.

Empfehlenswert ist also fast immer das getrennte Deklarieren und Erzeugen[1].

[1] Aus Platzgründen halten sich die Autoren leider nicht immer an diese Empfehlung.

BEISPIEL 3.6: Eine mögliche Fehlerbehandlung

C#
```
private CKunde kunde1;
try
{
    kunde1 = new CKunde();
}
catch(Exception ex)
{
 MessageBox.Show(ex.Message);
}
```

Klassische Initialisierung

Statt die Anfangswerte einzeln zuzuweisen, können Sie diese zusammen mit einem Konstruktor übergeben.

BEISPIEL 3.7: Das Objekt *kunde1* wird erzeugt (Standardkonstruktor), zwei Eigenschaften werden einzeln zugewiesen.

C#
```
CKunde kunde1 = new CKunde();
kunde1.anrede = "Frau"; kunde1.name = "Müller";
```

BEISPIEL 3.8: Das Objekt *kunde1* wird erzeugt und mit einem Konstruktor initialisiert.

C#
```
CKunde kunde1 = new CKunde("Frau", "Müller");
```

Weitere Einzelheiten entnehmen Sie dem Abschnitt 3.5.1.

Objekt-Initialisierer

Ab C# 3.0 wurden – vor allem in Hinblick auf die in der LINQ-Technologie erforderlichen anonymen Typen (siehe Abschnitt 2.2) – so genannte *Objektinitialisierer* eingeführt. Damit können nun öffentliche Eigenschaften und Felder von Objekten ohne das explizite Vorhandensein des jeweiligen Konstruktors in beliebiger Reihenfolge initialisiert werden. Das Initialisieren geschieht über geschweifte Klammern, in denen die einzelnen Felder/Eigenschaften des Objekts mit Werten belegt werden.

BEISPIEL 3.9: Gegeben ist eine Klasse *CPerson*:

C#
```
public class CPerson
{
      public string Name;
      public string Strasse;
      public int PLZ;
      public string Ort;
}
```

BEISPIEL 3.9: Gegeben ist eine Klasse *CPerson*:

Das Erzeugen und Initialisieren einer Instanz von *CPerson* bedarf keines Konstruktors:

```
CPerson person1 = new CPerson { Name = "Müller", Strasse = "Am Waldesrand 7", PLZ = 12345,
                                Ort = "Musterhausen" };
```

Arbeiten mit dem Objekt

Wie Sie bereits wissen, erfolgt der Zugriff auf Eigenschaften und Methoden eines Objekts, indem der Name des Objekts mit einem Punkt (.) vom Namen der Eigenschaft/Methode getrennt wird.

SYNTAX: *Objekt.Eigenschaft|Methode()*

BEISPIEL 3.10: Die Eigenschaft *guthaben* des Objekts *kunde1* wird zugewiesen und die Methode *adresse* aufgerufen

C#

```
kunde1.guthaben = 10;
label1.Text = kunde1.adresse();
```

Zerstören des Objekts

Wenn Sie das Objekt nicht mehr brauchen, können Sie die Objektvariable auf *null* setzen.

BEISPIEL 3.11: Der *kunde1* wird in die ewigen Jagdgründe befördert

C#

```
kunde1 = null;
```

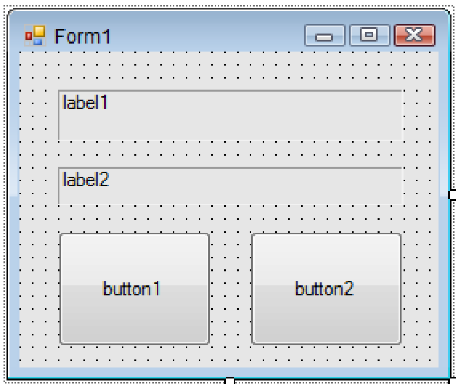
Das Objekt wird allerdings erst dann zerstört, wenn der Garbage Collector festgestellt hat, dass es nicht länger benötigt wird (siehe Abschnitt 3.5.2).

3.1.6 Einführungsbeispiel

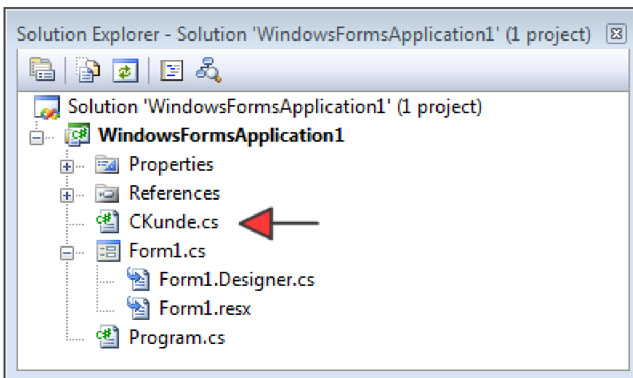
Raus aus dem muffigen Hörsaal, lassen Sie uns endlich einmal selbst eine einfache Klasse erstellen und beschnuppern!

Vorbereitungen

- Öffnen Sie ein neues Projekt (z.B. mit dem Namen "Kunden") als Windows Forms-Anwendung.
- Auf das Startformular (*Form1*) platzieren Sie zwei *Label*s und zwei *Button*s.



- Nachdem Sie den Menüpunkt *Projekt|Klasse hinzufügen...* gewählt haben, geben Sie im Dialogfenster den Namen *CKunde.cs* ein und klicken "Hinzufügen". Der Projektmappen-Explorer zeigt jetzt die neue Klasse:



Sie müssen eine Klasse nicht unbedingt in einem eigenen Klassenmodul definieren, Sie könnten die Klasse z.B. auch zum bereits vorhandenen Code des Formulars (*Form1.cs*) hinzufügen. Das Verwenden eigener Klassenmodule (idealerweise eins pro Klasse) steigert aber die Übersichtlichkeit des Programmcodes und erleichtert dessen Wiederverwendbarkeit.

Klasse definieren

Im Code-Fenster *CKunde.cs* ist bereits der Rahmencode für unsere Klasse vorbereitet:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace WindowsFormsApplication1
{
    class CKunde
    {
    }
}
```

Tragen Sie dann in den Klassenkörper die Implementierung der Klasse ein, sodass der komplette Code der Klasse schließlich folgendermaßen ausschaut:

```
public class CKunde
{
    private const char LF = (char) 10;  //  private Konstante (Zeilenumbruch)

    public string anrede;          //    öffentliches Feld
    public string name;            //      dto.
    public int plz;                //      dto.
    public string ort;             //      dto.
    public bool stammkunde;        //      dto.
    public decimal guthaben;       //      dto.

    public string adresse()                  //    öffentliche Methode
    {
      string s = anrede + " " + name + LF + plz.ToString() + " " + ort;
      return(s);
    }

    public void addGuthaben(decimal betrag)   // öffentliche Methode
    {
        if (stammkunde) guthaben += betrag;
    }
}
```

Bemerkungen

- Die Klasse verfügt über sechs "einfache" Eigenschaften, und zwar sind das alle als *public* deklarierten Variablen, die man auch als "öffentliche Felder" bezeichnet. Die Betonung liegt hier auf "einfach", da wir später noch lernen werden, wie man "richtige" Eigenschaften programmiert.
- Weiterhin verfügt die Klasse über zwei *Methoden*. Die *string*-Methode *adresse()* liefert einen Rückgabewert, nämlich die komplette Anschrift.
- Die *void*-Methode *addGuthaben* hingegen liefert keinen Wert zurück, sie erhöht den Wert des *guthaben*-Felds bei jedem Aufruf um 50 €.
- Die private Konstante *LF* wird von der Methode *adresse()* für das Einfügen des Zeilenumbruchs benötigt.

Objekt erzeugen und initialisieren

Wechseln Sie nun in das Code-Fenster von *Form1*.

Auf Klassenebene deklarieren Sie eine Objektvariable *kunde1*:

```
private CKunde kunde1;              // Objekt referenzieren
```

Dem linken Button geben Sie die Beschriftung "Objekt erzeugen und initialisieren" und belegen das *Click*-Ereignis wie folgt:

```
private void button1_Click(object sender, EventArgs e)
{
    kunde1 = new CKunde();          // Objekt erzeugen

    // Objektfelder initialisieren:
    kunde1.anrede = "Herr";
    kunde1.name = "Müller";
    kunde1.plz = 12345;
    kunde1.ort = "Berlin";
    kunde1.stammkunde = true;
}
```

Objekt verwenden

Hinterlegen Sie nun den rechten Button mit der Beschriftung "Methoden und Eigenschaften verwenden" wie folgt:

```
private void button2_Click(object sender, System.EventArgs e)
{
    label1.Text = kunde1.adresse();  // erste Methode aufrufen
    kunde1.addGuthaben(50M);         // zweite Methode aufrufen
    label2.Text = "Guthaben ist " + kunde1.guthaben.ToString("C");     // Eigenschaft lesen
}
```

Unterstützung durch die IntelliSense

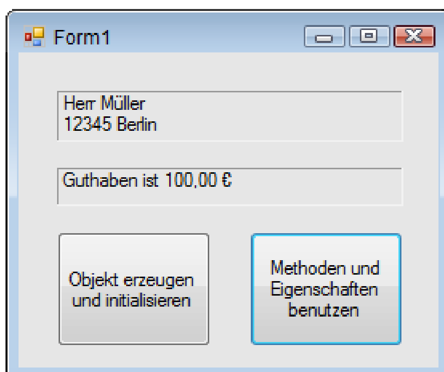
Sie haben beim Eintippen des Quelltextes (insbesondere im Code-Fenster von *Form1*) bereits gemerkt, dass Sie durch die IntelliSense von Visual Studio eifrigst unterstützt werden.

Die IntelliSense weist Sie z.B. auf die verfügbaren Klassenmitglieder (Eigenschaften und Methoden) hin und ergänzt den Quellcode automatisch, wenn Sie auf den gewünschten Eintrag doppelklicken.

Falls das gewünschte Klassenmitglied nicht erscheint, müssen Sie sofort stutzig werden und es keinesfalls mit dem gewaltsamen Eintippen des Namens versuchen, denn dann gibt es wahrscheinlich einen Fehler beim Kompilieren. Überprüfen Sie stattdessen lieber nochmals die Klassendeklaration, z.B. ob vielleicht nicht doch der *public*-Modifizierer vergessen wurde.

Objekt testen

Nun ist es endlich so weit, dass Sie Ihr erstes eigenes C#-Objekt vom Stapel lassen können. Unmittelbar nach Programmstart betätigen Sie den linken Button und danach den rechten. Durch mehrmaliges Klicken auf den zweiten Button wird sich das Guthaben des Kunden Müller in 50-€-Schritten erhöhen.



Falls Sie zu voreilig gewesen sind und unmittelbar nach Programmstart den zweiten statt den ersten Button gedrückt haben, stürzt Ihnen das Programm mit der Laufzeit-Fehlermeldung "Der Objektverweis wurde nicht auf eine Objektinstanz festgelegt." ab.

Bemerkungen

Unsere Klasse funktioniert nach außen hin zwar ohne erkennbare Mängel, ist hinsichtlich ihrer inneren Konstruktion aber keinesfalls als optimal zu bezeichnen. Wir haben deshalb keinerlei Grund, uns zufrieden zurückzulehnen, denn das uns unter C# zur Verfügung stehende OOP-Instrumentarium wurde von uns bei weitem noch nicht ausgeschöpft.

- Beispielsweise haben wir nur "einfache" Eigenschaften, nämlich *public*-Felder verwendet, was eigentlich eine schwere Sünde in den Augen der OOP-Puristen ist (siehe Abschnitt 3.2.1).
- Weiterhin war das Initialisieren der Eigenschaften über mehrere Codezeilen ziemlich mühselig (von einem hilfreichen Konstruktor haben wir noch keinerlei Gebrauch gemacht, siehe Abschnitt 3.5.1).
- Außerdem wird eine Klasse erst dann so richtig effektiv, wenn wir davon nicht nur eine, sondern mehrere Instanzen (sprich Objekte) ableiten. Diese wiederum kann man ziemlich elegant in so genannten Auflistungen (Collections) verwalten (siehe Kapitel 5).

Doch zur Beseitigung dieser und anderer Unzulänglichkeiten kommen wir erst später. Ein weiteres Problem, was uns unter den Nägeln brennt, duldet keinen weiteren Aufschub und wir wollen es deshalb gleich im folgenden Abschnitt behandeln.

3.2 Eigenschaften

Eigenschaften bestimmen die statischen Attribute eines Objekts, sie leiten sich von dessen *Zustand* ab, wie er in den Zustandsvariablen (Objektfeldern) gespeichert ist. Im Unterschied zu den Methoden, die von allen Instanzen der Klasse gemeinsam genutzt werden, sind die den Eigenschaften zugewiesenen Werte für alle Objekte einer Klasse meist unterschiedlich.

3.2.1 Eigenschaften mit Zugriffsmethoden kapseln

Von den im Objekt enthaltenen Feldern sind die *public*-Felder als "einfache" Eigenschaften zu betrachten.

In unserem Beispiel hatten wir für die Klasse *CKunde* solche "einfachen" Eigenschaften als *public*-Variable deklariert. Das allerdings ist nicht die "feine Art" der objektorientierten Programmierung, denn das Veröffentlichen von Feldern widerspricht dem hochgelobten Prinzip der Kapselung und erlaubt keinerlei Zugriffskontrolle wie z.B. Wertebereichsüberprüfung oder die Vergabe von Lese- und Schreibrechten.

Idealerweise sind deshalb in einem Objekt nur private Felder enthalten, und der Zugriff auf diese wird durch Accessoren (Zugriffsmethoden) gesteuert.

In diesem Sinn ist eine *Eigenschaft* gewissermaßen ein Mittelding zwischen Feld und Methode. Sie verwenden die Eigenschaft wie ein öffentliches Feld. Vom Compiler aber wird der Feldzugriff in den Aufruf von Accessoren – das sind spezielle Zugriffsmethoden auf private Felder – übersetzt. Doch schauen wir uns das Ganze lieber in der Praxis an.

Deklarieren von Eigenschaften

Eigenschaften werden ähnlich wie öffentliche Methoden deklariert. Innerhalb der Deklaration implementieren Sie für den Lesezugriff eine *set*- und für den Schreibzugriff eine g*et*-Zugriffsmethode. Während die *get*-Methode ihren Rückgabewert über *return* liefert, erhält die *set*-Methode den zu schreibenden Wert über *value*.

SYNTAX:

```
Modifizierer Datentyp Eigenschaftsname
{
    get
    {
        // hier Lesezugriff auf priv. Felder implementieren
        return(privatesFeld);
    }
    set
    {
        // hier Schreibzugriff auf priv. Felder implementieren
        privatesFeld = value;
    }
}
```

Wir wollen nun unser Beispiel mit "echten" Eigenschaften ausstatten. Dazu werden zunächst die public-Felder in *private* verwandelt und durch Voranstellen von "_" umbenannt, um Namenskonflikte mit den gleichnamigen Eigenschafts-Deklarationen zu vermeiden.

Der Schreibzugriff auf die Eigenschaft *anrede* wird so kontrolliert, dass nur die Werte "Herr" oder "Frau" zulässig sind.

```
public class CKunde
{
    private string _anrede;          //    privates Feld
    private string _name;            //      dto.
    ...

    public string anrede
    {
        get {return(_anrede); }
        set
        {
            if (value == "Herr" || value == "Frau") _anrede = value;
            else MessageBox.Show("Die Anrede '" + value + "' ist nicht zulässig!");
        }
    }

    public string name
    {
        get {return(_name); }
        set {_name = value; }
    }
```

```
    ...
  }
}
```

Zugriff

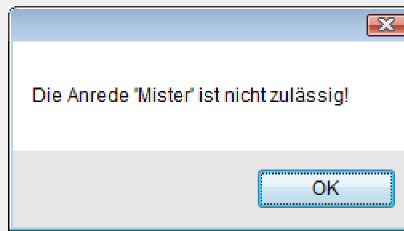
Wenn Sie ein Objekt verwenden, merken Sie auf Anhieb natürlich nicht, ob es noch über "einfache" oder schon über "richtige" Eigenschaften verfügt, es sei denn, die in die *get*- bzw. *set*-Methoden eingebauten Zugriffsbeschränkungen werden verletzt und Sie erhalten entsprechende Fehlermeldungen.

BEISPIEL 3.12: Sie wollen die Anrede "Mister" zuweisen, was zu einem Laufzeitfehler führt.

C#

```
kunde1 = new CKunde();
kunde1.anrede = "Mister";      // Fehler!
```

Ergebnis



Bemerkung

- Beim Schreiben des Quellcodes in der Entwicklungsumgebung Visual Studio merken Sie den "feinen" Unterschied zwischen "einfachen" und "richtigen" Eigenschaften, denn die IntelliSense zeigt dafür unterschiedliche Symbole[1].
- In unserem Beispiel verhält sich nur die Eigenschaft *anrede* "intelligent", d.h., sie unterliegt einer Zugriffskontrolle. Bei den übrigen Eigenschaften erfolgt lediglich eine 1:1-Zuordnung zu den privaten Feldern. Hier sollte man nicht "päpstlicher als der Papst" sein und es bei den ursprünglichen *public*-Feldern belassen. Wir aber haben diesen (eigentlich sinnlosen) Aufwand nur wegen des Lerneffekts betrieben.

3.2.2 Berechnete Eigenschaften

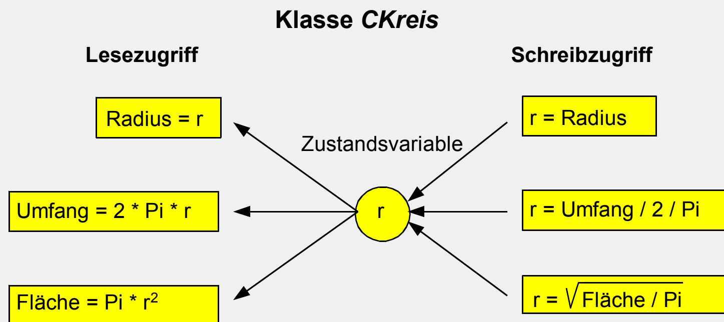
Mit Zugriffsmethoden lässt sich weit mehr anstellen, als nur den Zugriff auf private Felder der Klasse zu kontrollieren. So können z.B. innerhalb der Methode komplexe Berechnungen mit den Feldern (die man auch *Zustandsvariablen* nennt) und den übergebenen Parametern ausgeführt werden.

[1] Probieren Sie das bitte selbst aus!

BEISPIEL 3.13: Berechnete Eigenschaften

C#

Eine Klasse *CKreis* hat die Eigenschaften *radius*, *umfang* und *fläche*. In der einzigen Zustandsvariablen *r* braucht aber nur der Radius abgespeichert zu werden, da sich die übrigen Eigenschaften aus *r* berechnen lassen (*get* = Lesezugriff) bzw. umgekehrt (*set* = Schreibzugriff).




```
public class CKreis
{
    private double r;                    // das einzige Feld (Zustandsvariable)
```

Die Eigenschaft *radius*:

```
    public string radius
    {
       get {return (r.ToString("#,#0.00")); }
       set
       {
           if (value != "")  r = Convert.ToDouble(value);
           else r = 0;
       }
    }
```

Die Eigenschaft *umfang*:

```
    public string umfang
    {
        get {return (2 * Math.PI * r).ToString("#,#0.00"); }
        set
        {
            if (value != "")  r = Convert.ToDouble(value) / 2 / Math.PI;
            else r = 0;
        }
    }
```

Die Eigenschaft *fläche*:

```
    public string fläche
    {
```

BEISPIEL 3.13: Berechnete Eigenschaften

```
        get {return (Math.PI * Math.Pow(r, 2)).ToString("#,#0.00");}
        set
        {
            if (value != "") r = Math.Sqrt(Convert.ToDouble(value) / Math.PI);
            else r = 0;
        }
    }
}
```

Das komplette Programm finden Sie im Praxisbeispiel

► 3.9.1 Eigenschaften sinnvoll kapseln

3.2.3 Lese-/Schreibschutz

Es kommt häufig vor, dass bestimmte Felder bzw. Eigenschaften nur gelesen oder nur geschrieben werden dürfen.

Für Felder kann man einen Schreibschutz einfach durch Voranstellen des *readonly*-Modifizierers realisieren.

BEISPIEL 3.14: Ein schreibgeschütztes öffentliches Feld wird deklariert und initialisiert.

C#

```
public readonly double mwst = 0.19;
```

HINWEIS: Außer beim Deklarieren kann man ein ReadOnly-Feld auch in einem Konstruktor initialisieren! Genau dadurch unterscheidet sich das *readonly*- vom *const*-Schlüsselwort, denn *readonly*-Felder können – abhängig vom verwendeten Konstruktor – über unterschiedliche Werte verfügen.

Verwendet man statt öffentlicher Felder "richtige" Eigenschaften, so ist für eine Zugriffsbeschränkung keinerlei zusätzlicher Aufwand erforderlich – im Gegenteil:

HINWEIS: Um eine Eigenschaft allein für den Lese- bzw. Schreibzugriff zu deklarieren, lässt man einfach die *get*- bzw. die *set*-Zugriffsmethode weg.

BEISPIEL 3.15: In der *CKunde*-Klasse soll das Guthaben für den Schreibzugriff gesperrt werden

C#

Das klingt sicher logisch, da zur Erhöhung des Guthabens bereits die Methode *addGuthaben* existiert.

```
public decimal guthaben
{
    get {return(_guthaben); }
}
```

BEISPIEL 3.15: In der *CKunde*-Klasse soll das Guthaben für den Schreibzugriff gesperrt werden

Ergebnis

In der Entwicklungsumgebung von Visual Studio wird nun der Versuch abgewiesen, dieser Eigenschaft einen Wert zuzuweisen:

```
kunde1.stammkunde = true;
kunde1.guthaben = 10M;
    Einer Eigenschaft oder einem Indexer 'Kunden.CKunde.guthaben' kann nicht zugewiesen werden -- sie sind schreibgeschützt
```

3.2.4 Property-Accessoren

Es möglich, den Zugriff auf *get*- oder *set*- Accessoren von Eigenschaften zu beschränken. Meist ist dies nur für den *set*-Accessor sinnvoll, während der *get*-Accessor in der Regel öffentlich bleibt.

BEISPIEL 3.16: Property-Accessoren

C#

Eine Eigenschaft mit *get*- und *set*-Accessoren. Der *get*-Accessor besitzt die gleiche Sichtbarkeit wie die *KontoNummer*-Eigenschaft, während der *set*-Accessor nur einen *protected*-Zugriff erlaubt.

```
public string KontoNummer
{
    get
    {
        return _knr;
    }
    protected set
    {
        _knr = value;
    }
}
```

3.2.5 Statische Felder/Eigenschaften

Mitunter gibt es Felder bzw. Eigenschaften, deren Werte für alle aus der Klasse instanziierten Objekte identisch sind und die deshalb nur einmal in der Klasse gespeichert zu werden brauchen.

HINWEIS: Statische Felder/Eigenschaften (*Klasseneigenschaften*) werden mit dem Schlüsselwort *static* deklariert.

Statische Eigenschaften bzw. öffentliche Felder können benutzt werden, ohne dass dazu eine Objektvariable deklariert und ein Objekt instanziiert werden müsste! Es genügt das Voranstellen des Klassenbezeichners.

BEISPIEL 3.17: Die Klasse *CKunde* soll zusätzlich ein öffentliches Feld (bzw. eine "einfache" Eigenschaft) *rabatt* bekommen, die für jedes Kundenobjekt immer den gleichen Wert hat.

C#
```
public class CKunde
{
    ...
    public static double rabatt;
    ...
}
```

Der Zugriff ist sofort über den Klassenbezeichner möglich, ohne dass dazu eine Objektvariable erzeugt werden müsste.

BEISPIEL 3.18: Allen Kunden wird ein Rabatt von 15% zugewiesen.

C#
```
CKunde.rabatt = 0.15;
```

Vielen Umsteigern, die aus der strukturierten Programmierung kommen, bereitet es Schwierigkeiten, auf ihre geliebten globalen Variablen zu verzichten, mit denen sie bequem Werte zwischen verschiedenen Programmmodulen austauschen konnten. Genau hier bieten sich statische Eigenschaften bzw. öffentliche statische Felder an, die z.B. in einer extra für derlei Zwecke angelegten Klasse *CAllerlei* abgelegt werden könnten.

BEISPIEL 3.19: Die Klassen *Form1* und *Form2* greifen für allgemeine Berechnungen auf eine statische Eigenschaft *MWSt* der Klasse *CAllerlei* zu.

C#
```
public class CAllerlei
{
    private static double _mwst;

    public static double MWSt
    {
        get { return (_mwst);}
        set { _mwst = value; }
    }
    ...
```

Hier könnten z.B. auch nichtstatische Klassenmitglieder eingefügt werden, die natürlich auch auf das statische Feld *_mwst* zugreifen dürfen.

```
}
```

Sowohl von *Form1* als auch von *Form2* aus kann direkt auf die statische Eigenschaft *MWst* zugegriffen werden, eine Instanz von *CAllerlei* braucht dazu nicht erzeugt zu werden:

Zuweisen der Mehrwertsteuer in *Form1*:

```
public partial class Form1 : Form
{
    ...
```

BEISPIEL 3.19: Die Klassen *Form1* und *Form2* greifen für allgemeine Berechnungen auf eine statische Eigenschaft *MWSt* der Klasse *CAllerlei* zu.

```
        CAllerlei.MWSt = 0.19;
    ...
}
```

Anzeige der Mehrwertsteuer in *Form2*:

```
public partial class Form2 : Form
{
    ...
    private void button1_Click(object sender, EventArgs e)
    {
        textBox1.Text = CAllerlei.MWSt.ToString();
    }
}
```

Es versteht sich, dass innerhalb der *get*- bzw. *set*-Accessoren statischer Eigenschaften nur auf statische Klassenmitglieder (Felder) zugegriffen werden kann. Gleiches gilt übrigens auch für statische Methoden (siehe 3.3.3).

Konstante Felder

Obwohl ein Feld nicht als *static const* deklariert werden kann, verhält sich ein *const*-Feld im Wesentlichen statisch. Deshalb kann auf *const*-Felder mit der gleichen Notation wie bei *static*-Feldern zugegriffen werden (eine mit *new* erzeugte Objektinstanz ist nicht erforderlich).

BEISPIEL 3.20: Deklaration einer öffentlichen Konstanten und Zugriff

C#

```
public class CAllerlei
{
    public const int anzahl = 50;
    ...
}
```

Der Zugriff ist direkt möglich:

```
public partial class Form1 : Form
{
    ...
    n = CAllerlei.anzahl;
    ...
}
```

Worin unterscheiden sich denn dann statische und konstante Felder? Der wesentliche Unterschied ist der, dass man den Wert statischer Felder beim Erzeugen eines Objekts (durch Aufruf unterschiedlicher Konstruktoren) oder zur Laufzeit (durch Methodenaufrufe) ändern kann, bei konstanten Feldern geht das natürlich nicht.

3.2.6 Einfache Eigenschaften automatisch implementieren

Wie bereits im Abschnitt 3.2.1 erwähnt, gehört es zum schlechten Programmierstil, wenn *public*-Variablen quasi die Rolle von einfachen Eigenschaften übernehmen. Getreu der Devise "Hauptsache es funktioniert" ist es aber für den schreibfaulen Programmierer oft der bequemere Weg, erspart er sich damit doch viele Zeilen stupiden Codes. Automatisch implementierte Eigenschaften befreien den Programmierer aus diesem Zwiespalt, sie benötigen in der Regel auch nur eine einzige Codezeile, trotzdem erfolgt im Hintergrund eine exakte Implementierung mit *get*- und *set*-Zugriffsmethoden.

BEISPIEL 3.21: Drei Varianten für eine einfache Eigenschaft *Nachname.*

C#

1. Der kurze, aber schlechte Programmierstil:

```
public class CKunde
{
    public string Nachname;
    ...
}
```

2. Die exakte, aber umständliche Schreibweise:

```
public class CKunde
{
    private string _nachname;
    public string Nachname
    {
        get {return (_nachname);}
        set {_nachname = value;}
    }
    ...
}
```

3. Die Eigenschaft wird automatisch und sauber implementiert:

```
public class CKunde
{
    public string Nachname { get; set; }
    ...
}
```

Wie Sie sehen, ist nur eine einzige Codezeile erforderlich: Der Compiler erstellt hier ein privates, anonymes dahinter liegendes Feld (analog zu *_nachname* bei der ersten Variante), das nur durch die *get*- und *set*-Accessoren aufgerufen werden kann.

HINWEIS: Soll die Eigenschaft schreibgeschützt sein, so legen Sie einfach einen privaten *set*-Accessor fest.

BEISPIEL 3.22: Die automatisch implementierte Eigenschaft *Nachname* ist schreibgeschützt.

```
C#  public string Nachname { get; private set; }
```

HINWEIS: Es sei hier nochmals betont, dass sich nur einfache Eigenschaften automatisch implementieren lassen, keine berechneten Eigenschaften!

Verbesserungen (C# 6.0)

Seit C# 6.0 ist auch die automatische Initialisierung von Auto-Properties möglich.

BEISPIEL 3.23: Die Auto-Eigenschaft *Nachname* wird initialisiert.

```
C#  public string Nachname { get; set; } = "Doberenz";
```

Eine neue Funktionalität (Getter-only Auto-Properties) ermöglicht jetzt auch die Definition schreibgeschützter Eigenschaften ohne Deklaration eines Setters:

BEISPIEL 3.24: Die Auto-Eigenschaft *Nachname* ist schreibgeschützt und wird initialisiert.

```
C#  public string Nachname {get;} = "Doberenz";
```

3.3 Methoden

Methoden bestimmen die dynamischen Attribute eines Objekts, also sein Verhalten. Eine Methode ist eine Funktion, die im Körper der Klasse implementiert ist.

3.3.1 Öffentliche und private Methoden

Bereits im Kapitel 2 haben wir gelernt, wie man Methoden programmiert. Jetzt wollen wir noch etwas nachhaken und den Fokus auf die Methoden richten, die in unseren selbst programmierten Klassen zum Einsatz kommen.

Genau wie das bei "richtigen" Eigenschaften der Fall ist, arbeiten in einer sauber programmierten Klasse alle Methoden ausschließlich mit privaten Feldern (Zustandsvariablen) zusammen.

HINWEIS: Wenn Sie eine Methode als *private* deklarieren, ist sie nur innerhalb der Klasse sichtbar, und es handelt sich um keine Methode im eigentlichen Sinn der OOP, sondern eher um eine Funktion/Prozedur im herkömmlichen Sinn.

BEISPIEL 3.25: Die Methoden *adresse* und *addGuthaben* arbeiten mit sechs privaten Feldern zusammen

```
C#  public class CKunde
    {
```

BEISPIEL 3.25: Die Methoden *adresse* und *addGuthaben* arbeiten mit sechs privaten Feldern zusammen

```
    private string _anrede;         //    privates Feld
    private string _name;           //      dto.
    private int _plz;               //      dto.
    private string _ort;            //      dto.
    private bool _stammkunde;       //      dto.
    private decimal _guthaben;      //      dto.

    public string adresse()         //    öffentliche Methode
    {
        string s = _anrede + " " + _name  + _plz.ToString() + " " + _ort;
        return(s);
    }

    public void addGuthaben(decimal betrag)   // öffentliche Methode
    {
        if (stammkunde) _guthaben += betrag;
    }
}
```

Deklaration und Aufruf:

```
CKunde kunde1 = new CKunde();
...
label1.Text = kunde1.adresse();   // erste Methode aufrufen
kunde1.addGuthaben(50M);          // zweite Methode aufrufen
```

3.3.2 Überladene Methoden

Obwohl wir bereits in Abschnitt 2.7.5 ganz allgemein auf dieses Thema eingegangen sind, soll es hier nochmals im OOP-Kontext diskutiert werden.

Innerhalb des Klassenkörpers dürfen zwei und mehr gleichnamige Methoden konfliktfrei nebeneinander existieren, wenn sie eine unterschiedliche Signatur (Reihenfolge und Datentyp der Übergabeparameter) besitzen.

BEISPIEL 3.26: Überladene Methoden

C#

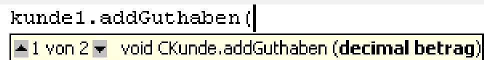
Zwei überladene Versionen einer Methode in der Klasse *CKunde*, die erste hat nur den Nettobetrag als Parameter die zweite den Bruttobetrag und die Mehrwertsteuer.

```
public void addGuthaben(decimal betrag)              // erste Überladung
{
  if (stammkunde) _guthaben += betrag;
}
```

BEISPIEL 3.26: Überladene Methoden

```
public void addGuthaben(decimal brutto, decimal mwst)   // zweite Überladung
{
  _guthaben += Convert.ToDecimal(brutto/(1 + mwst));
}
```

Wenn Sie diese Methoden verwenden wollen, so fällt die Auswahl im Code-Fenster leicht:



3.3.3 Statische Methoden

Genauso wie die unter 3.2.5 erläuterten *statischen Eigenschaften* können *statische Methoden* (auch als *Klassenmethoden* bezeichnet) ohne Verwendung eines Objekts aufgerufen werden. Statische Methoden eignen sich z.B. gut für diverse Formelsammlungen (ähnlich *Math*-Klassenbibliothek). Auch können Sie damit auf private statische Klassenmitglieder zugreifen.

HINWEIS: Der Einsatz statischer Methoden für relativ einfache Aufgaben ist bequemer und ressourcenschonender als das Arbeiten mit Objekten, die Sie jedes Mal extra instanziieren müssten.

BEISPIEL 3.27: Wir bauen eine Klasse, in der wir wahllos einige von uns häufig benötigte Berechnungsformeln verpacken.

C#

```
public class CMeineFormeln
{
    public static double kreisUmfang(double radius)
    {
       return (2 * Math.PI * radius);
    }

    public static double kugelVolumen(double radius)
    {
       return ( 4 / 3.0 * Math.PI * Math.Pow(radius, 3));
    }

    public static decimal netto(decimal brutto, double mwst)
    {
        return(brutto/ Convert.ToDecimal(1 + mwst));
    }

   // .... weitere Methoden
}
```

Der Zugriff von außerhalb ist absolut problemlos, weil man sich nicht mehr um das lästige Instanziieren einer Objektvariablen kümmern muss.

BEISPIEL 3.28: Die statischen Methoden der Klasse *meineFormeln* werden in einer Eingabemaske aufgerufen.

C#

```
private void button1_Click(object sender, EventArgs e)
{
     double r = Convert.ToDouble(textBox1.Text);              // Kreisradius konvertieren

     label1.Text = CMeineFormeln.kreisUmfang(r).ToString("0.000");
     label2.Text = CMeineFormeln.kugelVolumen(r).ToString("0.000");

     decimal b = Convert.ToDecimal(textBox2.Text);            // Brutto konvertieren
     label3.Text = CMeineFormeln.netto(b, 0.19).ToString("C");
}
```

Ergebnis

Bemerkungen

- Sie können mit *static* auch ein Feld deklarieren, das von allen Instanzen der Klasse gemeinsam genutzt werden kann und nicht für jedes Objekt extra zugewiesen werden muss. Zum Initialisieren kann ein so genannter *statischer Konstruktor* Verwendung finden (siehe 3.5.1).
- Eine Klasse mit ausschließlich statischen Mitgliedern kann mit dem Schlüsselwort *static* deklariert werden, siehe 3.7.4 oder Praxisbeispiel

  ▶ 3.9.2 Eine statische Klasse anwenden.

3.4 Ereignisse

Nachdem wir uns den Eigenschaften und Methoden von Objekten ausführlich gewidmet haben, wollen wir die Dritten im Bunde, die Ereignisse, nicht vergessen. Wie Sie bereits wissen, werden Ereignisse unter bestimmten Bedingungen vom Objekt ausgelöst und können dann in einer Ereignisbehandlungsroutine abgefangen und ausgewertet werden.

Allerdings bieten bei weitem nicht alle Klassen Ereignisse an, denn diese werden nur benötigt, wenn auf bestimmte Änderungen eines Objekts reagiert werden soll.

Nachdem wir mit dem Deklarieren von Eigenschaften und Methoden überhaupt keine Probleme hatten, hört aber bei Ereignissen der Spaß auf. Im Folgenden werden deshalb nur die wichtigsten Grundlagen der Ereignismodellierung erläutert.

3.4.1 Ereignis hinzufügen

Um einer Klasse ein Ereignis hinzuzufügen, sind drei Schritte erforderlich:

1. Die Deklaration des Ereignistyps (*delegate*-Schlüsselwort)
2. Die Instanziierung des Ereignisses (*event*-Schlüsselwort)
3. Das Auslösen des Ereignisses (innerhalb einer Methode oder Eigenschaft)

Um einer heillosen Verwirrung vorzubeugen, machen wir es diesmal umgekehrt und beginnen gleich mit einem Beispiel, ehe wir später die Syntax und weitere Einzelheiten erklären.

BEISPIEL 3.29: Ereignis-Delegate hinzufügen

C#

In unserer *CKunde*-Klasse wird ein Ereignis-Delegate mit dem Namen *GuthabenLeer* deklariert, davon wird ein Ereignis mit dem Namen *guthabenLeer1* instanziiert. Dieses Ereignis "feuert" innerhalb der Methode *addGuthaben* genau dann, wenn das Guthaben den Wert von 10 € unterschreitet.

```
public class CKunde
{
```

Um den Code für den Benutzer der Klasse etwas zu vereinfachen, werden den privaten Feldern Standardwerte zugewiesen[1]:

```
    private bool _stammkunde = true;        //      initialisiertes Feld
    private decimal _guthaben = 100M;       //      dto.
```

1. Schritt: den Ereignistyp definieren:

```
    public delegate void GuthabenLeer(object sender, string e);
```

2. Schritt: eine Ereignisinstanz *guthabenLeer1* deklarieren:

```
    public event GuthabenLeer guthabenLeer1;
```

[1] Später werden wir diese Aufgabe dem Konstruktor übertragen.

BEISPIEL 3.29: Ereignis-Delegate hinzufügen

Die Methode, in welcher das Ereignis ausgelöst wird:

```
    public void addGuthaben(decimal betrag)
    {
        if (stammkunde) _guthaben += betrag;
```

3. Schritt: Ereignis auslösen:

```
        if (_guthaben <= 10)
        {
            // das Ereignis feuert nur, wenn ...
            if (guthabenLeer1 != null)  // ... mindestens ein Eventhandler angemeldet ist
            {
                string msg = "Das Guthaben beträgt nur noch " +
                             _guthaben.ToString("C") + "!";
                guthabenLeer1(this, msg);
            }
        }
    }
    // ... weitere Implementierungen
}
```

Nun kommen wir zu den sicherlich dringend notwendigen Erklärungen:

Ereignis deklarieren

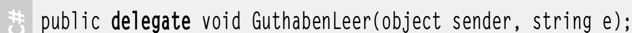
Wie bereits kurz erwähnt, werden Ereignisse von so genannten Delegaten abgeleitet. Ein Delegate ist ein Ereignistyp, er sieht – bis auf das *delegate*-Schlüsselwort – wie eine Methode aus und verhält sich auch ähnlich.

HINWEIS: Delegaten ermöglichen es, ein Framework von Rückruf- und Benachrichtigungsmethoden für miteinander kooperierende Klassen zu implementieren.

SYNTAX: *Modifizierer* **delegate** *Datentyp delegateName* (*Datentyp Parameter*);

Falls der Delegate keinen Rückgabewert liefert, wird dieser – wie bei einer Methode – als *void* angegeben.

BEISPIEL 3.30: Die Deklaration des Delegaten aus dem Vorgängerbeispiel



```
public delegate void GuthabenLeer(object sender, string e);
```

Ereignis instanziieren

Nachdem Sie mittels Delegaten den Ereignistyp deklariert haben, steht dem Erzeugen einer Delegatinstanz, also eines spezifischen Ereignisses, nichts mehr im Wege. Sie verwenden dazu das *event*-Schlüsselwort.

SYNTAX: `Modifizierer event delegateName ereignisName;`

Ähnlich wie bei Objekten (diese sind bekanntlich Instanzen einer Klasse) handelt es sich bei einem Ereignis um eine Instanz des Delegaten. Da der Aufbau bereits feststeht, genügen die Angabe des Namens der Ereignisdeklaration (*delegateName*) und der spezielle Name des Ereignisses (*ereignisName*).

BEISPIEL 3.31: Von im Vorgängerbeispiel deklarierten Delegaten wird ein Ereignis mit dem Namen *guthabenLeer1* instanziiert.

C#

```
public event GuthabenLeer guthabenLeer1;
```

Es ist durchaus möglich und üblich, auch mehrere Ereignisse vom gleichen Delegaten abzuleiten.

Ereignis auslösen

Ein Ereignis wird immer innerhalb der Klasse ausgelöst, in der es deklariert und erzeugt wurde. Das kann an verschiedenen Stellen innerhalb von Methoden oder Eigenschaften geschehen.

Das Auslösen erfolgt wie ein normaler Methodenaufruf:

SYNTAX: `ereignisName(Parameter);`

Die Signatur der Parameter (Reihenfolge und Datentyp) muss der im entsprechenden Delegate festgelegten Parameterliste entsprechen.

HINWEIS: Ereignisse sind Verweistypen und können demzufolge auch auf *null* abgefragt werden.

BEISPIEL 3.32: Ereignis auslösen

C#

Das im Vorgängerbeispiel deklarierte Ereignis wird innerhalb der Methode *addGuthaben* ausgelöst[1]. Vor dem Aufruf wird getestet, ob zumindest ein Eventhandler für dieses Ereignis angemeldet ist (was genau unter "Anmelden" zu verstehen ist, erfahren Sie im nächsten Abschnitt).

```
public void addGuthaben(decimal betrag)
{
    if (stammkunde) _guthaben += betrag;
    if (_guthaben <= 10)
    {
        if (guthabenLeer1 != null)    // mindestens ein Eventhandler angemeldet?
        {
            string msg = "Das Guthaben beträgt nur noch " +
                                _guthaben.ToString("C") + "!";
```

[1] Im Programmiererjargon sagt man auch "Das Ereignis feuert"!

BEISPIEL 3.32: Ereignis auslösen

```
                guthabenLeer1(this, msg);
            }
        }
    }
```

3.4.2 Ereignis verwenden

In der Klasse, in welcher wir mit dem Ereignis arbeiten wollen, sind – zusätzlich zur Erzeugung der Objektvariablen – zwei Schritte durchzuführen:

1. Ereignisbehandlung (Eventhandler) schreiben
2. Eventhandler anmelden

Lassen Sie uns auch hier mit einem Beispiel beginnen.

BEISPIEL 3.33: Ereignisse verwenden

C#

Wir nutzen die im Vorgängerabschnitt definierte Klasse *CKunde*, welche von uns gerade mit dem Ereignis *guthabenLeer1* nachgerüstet wurde.

Auf Klassenebene referenzieren wir zunächst die übliche Objektvariable:

```
private CKunde kunde1;           // Objekt referenzieren
```

1. Schritt: Wir schreiben nun eine Ereignisbehandlung (Eventhandler) für das Ereignis:

```
private void guthabenKontrolle(object o, string s)
{
    CKunde k = (CKunde)o;
    label2.Text = k.nachName + ": " + s;       // Ausgabe einer Warnung
}
```

2. Schritt: Um das Ereignis mit dem Eventhandler zu verbinden, ist eine Anmeldung erforderlich, die wir in den Konstruktorcode von *Form1* einfügen können:

```
public Form1()
{
        InitializeComponent();
        kunde1 = new CKunde();     // Objekt instanziieren

        // Eventhandler anmelden:
        kunde1.guthabenLeer1 += new CKunde.GuthabenLeer(guthabenKontrolle);
}
```

Das Ereignis ist jetzt eingebunden, und einem Funktionstest steht nichts mehr im Weg. Dazu rufen wir wiederholt die Methode *addGuthaben* auf, die das Guthaben jedes Mal um 10 € verringert:

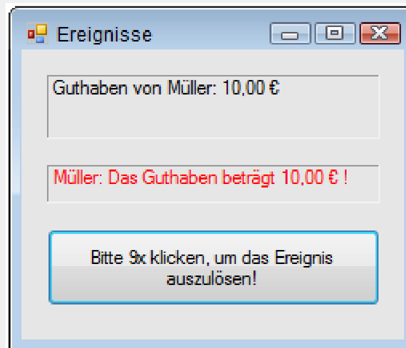
```
private void button1_Click(object sender, EventArgs e)
{
```

BEISPIEL 3.33: Ereignisse verwenden

```
        label1.Text = kunde1.adresse();
        kunde1.addGuthaben(-10M);        // Guthaben verringern
        label2.Text = "Guthaben ist " + kunde1.guthaben.ToString("C");
    }
```

Ergebnis

Nachdem Sie den *Button* neunmal geklickt haben, wird das Ereignis ausgelöst:



Den kompletten Quellcode entnehmen Sie bitte den Buchbeispielen.

Nun zu den Details.

Ereignisbehandlung schreiben

Die Frage "Was soll passieren, wenn das Ereignis ausgelöst wurde?" wird in einer Ereignisbehandlungsmethode (Eventhandler) beantwortet.

SYNTAX: `Modifizierer Datentyp methodenName (Datentyp Parameter)`

Den Namen der Methode können Sie frei wählen. Den konkreten Namen des Ereignisses finden Sie hier nicht, d.h., eine Ereignisbehandlung lässt sich auch von mehreren Ereignissen gemeinsam verwenden. Lediglich die Methodensignatur (*Datentyp Parameter*) muss der des Delegates entsprechen, nach dessen Muster das Ereignis erzeugt wurde.

BEISPIEL 3.34: Ereignisbehandlung schreiben

C#

Beim Auftreten des Ereignisses wird nicht nur der Parameter *s* angezeigt, in der Titelleiste erscheint zusätzlich noch der Name des Kunden, den wir durch explizite Typkonvertierung aus dem ebenfalls übergebenen *object*-Parameter "herausziehen".

```
private void guthabenKontrolle(object o, string s)
{
    CKunde k = (CKunde) o;                  // Typecasting
    label2.Text = k.nachName + ": " + s;    // Warnung ausgeben
}
```

Ereignisbehandlung anmelden

Um dem Compiler mitzuteilen, welcher Eventhandler bei Auftreten des Ereignisses denn nun aufzurufen ist, müssen Sie die gewünschte Ereignisbehandlung beim Objekt (der Klasseninstanz) anmelden.

SYNTAX: `Objekt.ereignisName += new delegateName(eventHandlerName);`

BEISPIEL 3.35: Dem Objekt *kunde1* wird mitgeteilt, dass bei Auftreten des Ereignisses *guthabenLeer1* der Eventhandler *guthabenKontrolle* aufzurufen ist

C#
```
kunde1.guthabenLeer1 += new CKunde.GuthabenLeer(guthabenKontrolle);
```

Wichtig ist dabei die Verwendung des Operators += (siehe Sprachkapitel, Abschnitt 2.4), denn pro Ereignis sind durchaus mehrere Eventhandler möglich. In diesem Fall erfolgt deren Abarbeitung in der Reihenfolge der Anmeldung.

BEISPIEL 3.36: Zum Ereignis *guthabenLeer1* wird ein zweiter Eventhandler hinzugefügt

C#

Beim Eintreten des Ereignisses erscheint zunächst das vom ersten Eventhandler produzierte Meldungsfenster (siehe oben) und anschließend der Name des Kunden in einem Label.

```
// zweiten Eventhandler implementieren
private void kundenAnschrift(object o, string s)
{
    CKunde k = (CKunde) o;
    label3.Text = k.name;
}
 ....
// zweiten Eventhandler hinzufügen:
kunde1.guthabenLeer1 += new CKunde.GuthabenLeer(kundenAnschrift);
```

Falls ein Eventhandler nicht mehr benötigt wird, sollten Sie ihn wieder abmelden.

BEISPIEL 3.37: Abmelden eines Eventhandlers *kundenAnschrift*

C#
```
kunde1.guthabenLeer1 -= new CKunde.GuthabenLeer(kundenAnschrift);
```

Bemerkungen

Wenn Sie im Eigenschaften-Fenster der Visual Studio-Entwicklungsumgebung auf bekannte Weise Eventhandler für die Objekte der Bedienoberfläche erzeugen, so hat die IDE nicht nur den Rahmencode des Eventhandlers für Sie generiert, sondern – quasi im Verborgenen – auch die benutzten Ereignisse angemeldet. Üblicherweise übergeben diese Ereignisse zwei Parameter an die aufrufende Instanz: eine Referenz auf das Objekt, welches das Ereignis ausgelöst hat, und ein Objekt der *EvenArgs*- oder einer davon abgeleiteten Klasse.

BEISPIEL 3.38: Rahmencode des automatisch generierten Eventhandlers für das *Click*-Ereignis eines *Button*

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

    }
```

Klappen Sie die Region *Vom Windows Form-Designer generierter Code* auf, so finden Sie die entsprechende Befehlszeile für die Anmeldung des Eventhandlers:

```
    this.button1.Click += new EventHandler(this.button1_Click);
```

Die folgende Abbildung veranschaulicht nochmals die Syntax dieser Zeile.

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

Objekt | Ereignis | Delegat | Event-Handler

Und zum Schluss noch ein Hinweis auf einen ziemlich häufigen Unterlassungsfehler:

HINWEIS: Wenn Sie die Codezeilen eines Eventhandlers komplett per Hand löschen (man sollte das eigentlich nicht tun), so müssen Sie auch die entsprechende Anmeldungszeile (siehe oben) löschen, ansonsten gibt es einen Compilerfehler!

3.5 Arbeiten mit Konstruktor und Destruktor

Eine "richtige" objektorientierte Sprache wie C# realisiert das Erzeugen und Entfernen von Objekten mit Hilfe von Konstruktoren und Destruktoren.

"Bis jetzt sind wir doch glänzend ohne Konstruktor ausgekommen!", werden Sie jetzt vielleicht einwenden. Ganz stimmt das nicht, denn wenn Sie sich um keinen eigenen Konstruktor kümmern, wird der von *System.Object* geerbte parameterlose *new*-Standardkonstruktor verwendet.

3.5.1 Konstruktor und Objektinitialisierer

Der Konstruktor[1] ist gewissermaßen die Standardmethode der Klasse und kann in mehreren Überladungen vorhanden sein.

HINWEIS: Der Name des Konstruktors ist immer identisch mit dem Namen der Klasse.

Der Konstruktor wird automatisch bei der Instanziierung eines Objekts (*new*) aufgerufen und dient vor allem dazu, den Feldern des neu erzeugten Objekts Anfangswerte zuzuweisen.

[1] Seit C# 6.0 verfügen auch Strukturen (*struct*) über einen (parameterlosen) Konstruktor.

Deklaration

Einen Konstruktor fügen Sie dem Klassenkörper ähnlich wie eine *public void*-Methode hinzu, nur dass Sie der Methode den Namen der Klasse geben und auf das *void*-Schlüsselwort verzichten. Als Parameter übergeben Sie die Werte für die Felder, die initialisiert werden sollen.

SYNTAX:

```
public KlassenName(Datentyp Parameter)
{
   // Initialisierung der Klasse
}
```

Wie bei jeder anderen Methode können Sie auch hier mehrere überladene Konstruktoren implementieren.

BEISPIEL 3.39: Unserer Klasse *CKunde* werden zwei überladene Konstruktoren hinzugefügt

C#

```
public class CKunde
{
```

Die Felder:

```
    private string _anrede;
    private string _name;
    private int _plz;
    private string _ort;
    private bool _stammKunde;
    private decimal _guthaben;
```

Der erste Konstruktor initialisiert nur zwei Felder:

```
   public CKunde(string anr, string nam)
   {
       _anrede = anr; _name = nam;
   }
```

Der zweite Konstruktor initialisiert **alle** Felder der Klasse:

```
   public CKunde(string a, string n, int p, string o, bool s, decimal g)
   {
       _anrede = a;   _name = n;     _plz = p;
       _ort = o;  _stammKunde = s;   _guthaben = g;
   }
   ...
}
```

Aufruf

Nachdem Sie einer Klasse einen oder mehrere Konstruktoren hinzugefügt haben, sind Sie auch zur Verwendung von mindestens einem davon verpflichtet. Die bisher gewohnte einfache Instanziierung von Objekten ist nicht mehr möglich!

BEISPIEL 3.40: Zwei Objekte der Klasse *CKunde* werden erzeugt und mit Anfangswerten initialisiert.

C#

Objekte referenzieren:

```
CKunde kunde1, kunde2, kunde3;
```

Sicherheitshalber bauen wir das Erzeugen der Objekte in einen Exception-Handler ein:

```
try
{
```

Für jedes Objekt wird ein anderer überladener Konstruktor verwendet:

```
    kunde1 = new CKunde("Herr", "Müller");
    kunde2 = new CKunde("Frau", "Hummel", 12345, "Berlin", true, 100);
    //  kunde3 = new CKunde();                          // erzeugt Compilerfehler!!!
    MessageBox.Show("Objekte erfolgreich erzeugt!");
}
catch(Exception ex)
{
    MessageBox.Show(ex.Message + " Sch... Konstruktor!");
}
```

Wenn Sie den Code mit dem Anfangsbeispiel im Abschnitt 3.1.6 vergleichen, so sehen Sie, dass das Initialisieren der Objekte viel übersichtlicher geworden ist. Statt umständlich eine Eigenschaft nach der anderen zuzuweisen, geht das jetzt in einer einzigen Befehlszeile.

Objektinitialisierer

Objektinitialisierer erlauben das Erzeugen und Initialisieren von Instanzen auf Basis von (öffentlichen) Objekteigenschaften auch ohne das explizite Vorhandensein eines Konstruktors[1].

BEISPIEL 3.41: Erzeugen eines Objekts mittels Objektinitialisierer (das ist kein Konstruktoraufruf!)

C#

```
kunde1 = new CKunde {Anrede = "Herr", Name = "Müller"};
```

Statischer Konstruktor

Neben einem oder mehreren "normalen" Konstruktoren kann eine Klasse auch über einen *statischen* Konstruktor verfügen. Ein solcher Konstruktor wird verwendet, um *static*-Felder (siehe 3.2.5) zu initialisieren oder um einmaligen Initialisierungscode auszuführen. Der Aufruf erfolgt automatisch, bevor die erste Instanz erstellt oder auf statische Klassenmitglieder verwiesen wird.

[1] Notwendig wurde diese Spracherweiterung vor allem wegen der LINQ-Technologie, deren anonyme Typen beispielsweise nach einem solchen Feature verlangen (siehe Kapitel 6).

BEISPIEL 3.42: Eine Klasse besitzt die statische Eigenschaft *MWSt*, welche zu Beginn mit dem Wert 0,19 initialisiert werden soll.

C#

```
public class CAllerlei
{
    private static double _mwst;

    public static double MWSt
    {
        get { return (_mwst); }
        set { _mwst = value; }
    }

    static CAllerlei()                              // statischer Konstruktor
    {
        _mwst = 0.19;
    }
    ...
}
```

Der Zugriff von einem Formular *Form1* aus:

```
public partial class Form1 : Form
{
    ...
    textBox1.Text = CAllerlei.MWSt.ToString();      // zeigt 0,19
    ...
}
```

HINWEIS: Ein statischer Konstruktor akzeptiert weder Zugriffsmodifizierer, noch besitzt er Parameter, er kann auch nicht direkt aufgerufen werden.

3.5.2 Destruktor und Garbage Collector

Das Pendant zum Konstruktor ist aus objektorientierter Sicht der Destruktor. Da der Lebenszyklus eines Objektes bekanntlich mit dessen Zerstörung und der Freigabe der belegten Speicherplatzressourcen endet, ist der Destruktor für das Erledigen von "Aufräumarbeiten" zuständig, kurz bevor das Objekt sein Leben aushaucht.

In .NET haben wir allerdings keine echten Destruktoren, da hier die endgültige Zerstörung eines Objekts nicht per Code, sondern automatisch vom Garbage Collector vorgenommen wird. Dieser durchstöbert willkürlich und in unregelmäßigen Zeitabständen den Heap nach Objekten, um diejenigen zu suchen, die nicht mehr referenziert werden.

An die Stelle eines echten Destruktors tritt ein Quasi-Destruktor. Das ist eine Finalisierungsmethode, die zu einem unbestimmbaren Zeitpunkt vom Garbage Collector aufgerufen wird, kurz bevor dieser das Objekt vernichtet.

Ähnlich wie beim Konstruktor wird auch hier der Name der Klasse als Methodenbezeichner verwendet, allerdings mit einer Tilde (~) als Präfix. Der *public*-Zugriffsmodifizierer entfällt, da Sie selbst den Destruktor nicht aufrufen dürfen, auch Parameter dürfen nicht übergeben werden.

SYNTAX:

```
~KlassenName()
{
    // hier Code für Aufräumarbeiten implementieren
}
```

BEISPIEL 3.43: Destruktor und Garbage Collector

C#

Unsere Klasse *CKunde* erhält ein öffentliches statisches Feld, welches durch den Konstruktor inkrementiert und durch den Quasi-Destruktor dekrementiert werden soll. Wir beabsichtigen damit, die Anzahl der momentan instanziierten Klassen (sprich Anzahl der Kunden) abzufragen.

Der auf das Wesentliche reduzierte Code von *CKunde*:

```
public class CKunde
{
     public static int anzahl = 0;

     // Konstruktor:
     public CKunde()
     {
         anzahl++;
     }

     // Destruktor:
      ~CKunde()
     {
          anzahl--;
     }
}
```

Wir verwenden zum Testen der Klasse ein Windows-Formular mit zwei *Button*s, einer *Timer*-Komponente (*Interval* = 1000, *Enabled* = *True*) und einem *Label*.

Zum Code der Klasse *Form1* fügen Sie hinzu:

```
CKunde kunde1;  // Objekt referenzieren

// Objekt hinzufügen:
private void button1_Click(object sender, EventArgs e)
{
    kunde1 = new CKunde();
}

// Objekt entfernen:
private void button2_Click(object sender, EventArgs e)
{
```

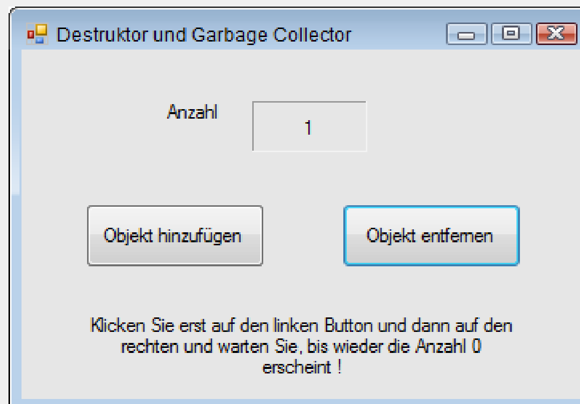
BEISPIEL 3.43: Destruktor und Garbage Collector

```
        kunde1 = null;     // dereferenzieren
    }

    // Anzeige der im Speicher befindlichen Instanzen im Sekundentakt:
    private void timer1_Tick(object sender, EventArgs e)
    {
        label1.Text = CKunde.anzahl.ToString();
    }
```

Ergebnis

Beim Programmtest müssen Sie etwas Geduld aufbringen.

Nach dem Programmstart fügen Sie durch Klicken auf den linken Button ein Objekt *kunde1* hinzu, wonach sich die Anzeige von 0 auf 1 ändert. Anschließend klicken Sie auf den rechten Button, um das Objekt wieder zu entfernen.



Es kann einige Zeit dauern, bis die Anzeige wieder auf 0 zurück geht, nämlich dann, wenn dem Garbage Collector gerade einmal wieder die Lust zum Aufräumen überkommt und er den Quasi-Destruktor aufruft[1].

Übrigens können Sie auch den linken Button mehrmals hintereinander klicken. Die Anzeige zählt zwar hoch, das aber täuscht, denn es bleibt bei nur einer Objektvariablen (*kunde1*). Allerdings wird Ressourcenverschwendung betrieben, denn dem Objekt wird immer wieder ein neuer Speicherbereich zugewiesen. Der vorher belegte Speicher liegt brach und wartet auf die Freigabe durch den Garbage Collector.

HINWEIS: Obiges Beispiel sollten Sie aufgrund seiner Unberechenbarkeit keinesfalls als Vorbild für ähnliche Zählaufgaben verwenden!

[1] Der Garbage Collector läuft in einem eigenen Thread, er wird nur dann aufgerufen, wenn sich die anderen Threads in einem sicheren Zustand befinden.

Da wegen der Unberechenbarkeit der Objektvernichtung der Umgang mit dem Quasi-Destruktor ziemlich problematisch ist, sollten Sie für das definierte Freigeben von Objekten besser eine separate Methode (*Close*- bzw. *Dispose*) oder *using* verwenden (siehe folgender Abschnitt).

3.5.3 Mit using den Lebenszyklus des Objekts kapseln

Mit dem Schlüsselwort *using* kann man nicht nur Namespaces einbinden (siehe Kapitel 5), sondern in völlig anderer Bedeutung auch für das sichere Erzeugen und Vernichten von Objekten sorgen. Hinter den Kulissen wird ein *try-finally*-Block um das entsprechende Objekt generiert und beim Beendigen für das Objekt *Dispose()* aufgerufen. Das folgende (eigentlich lächerliche) Beispiel soll lediglich das Prinzip verdeutlichen.

BEISPIEL 3.44: Zum Prinzip von *using*

C#

```
using (CKunde kunde1 = new CKunde("Herr", "Müller"))      // Erzeugen des Objekts
{
    kunde1.Wohnort = "Berlin";                            // Arbeiten mit dem Objekt
    ...
}                                                         // Freigabe des Objekts
```

Weitaus sinnvollere Beispiele für *using* finden Sie im Praxisbeispiel

▶ 8.8.3 Ein Memory Mapped File (MMF) verwenden

und im Abschnitt 23.3.5 des ADO.NET-Kapitels (Datenbankzugriff).

3.5.4 Verzögerte Initialisierung

Erzeugen Sie wie bisher ein Objekt mit *new*, so wird der Speicher gleich bei der Initialisierung belegt. Die verzögerte Initialisierung (*Lazy Initialization*) von Objekten hat hingegen den Vorteil, dass die Objekte erst dann Speicherplatz belegen, wenn sie tatsächlich verwendet werden. Das lohnt sich besonders für Anwendungen mit sehr vielen oder sehr umfangreichen Klasseninstanzen.

Realisiert wird die verzögerte Initialisierung mit der ab .NET 4.0 eingeführten generischen Klasse *System.Lazy*<>, welche die tatsächliche Klasse kapselt.

Um das Prinzip zu verdeutlichen, gehen wir von einer allgemeinen Klasse aus:

```
public class Klasse1
{
    public Klasse1()
    {
        // Konstruktor
    }

    public string Eigenschaft1
    { get; set; }
```

```
    public void Methode1()
    {
        // ...
    }
}
```

Die verzögerte Initialisierung wird vorbereitet:

```
Lazy<Klasse1> objInit;
objInit = new Lazy<Klasse1>();
```

Bis jetzt wurde das Objekt noch nicht erstellt (der Wert der *objInit.IsValueCreated*-Eigenschaft ist *false*).

Erst beim Zugriff auf eines seiner Mitglieder wird das Objekt erzeugt:

```
objInit.Value.Eigenschaft1 = "Das ist der erste Wert!";
```

Damit ist das Objekt initialisiert (die *objInit.IsValueCreated*-Eigenschaft ist *true*).

HINWEIS: Standardwerte von Feldern werden erst bei der erstmaligen Verwendung zugewiesen, dasselbe gilt für die Ausführung eines evtl. vorhandenen Konstruktors.

3.6 Vererbung und Polymorphie

Ein zentrales OOP-Thema ist die *Vererbung*, die es ermöglicht, Klassen zu definieren, die von anderen Klassen abhängen. Eng mit der Vererbung verknüpft ist die *Polymorphie* (Vielgestaltigkeit). Man versteht darunter die Fähigkeit von Subklassen, die Methoden der Basisklasse mit unterschiedlichen Implementierungen zu verwenden. C# unterstützt sowohl Vererbung als auch polymorphes Verhalten, da das Überschreiben (*Overriding*) der Basisklassenmethoden mit alternativen Implementierungen erlaubt ist.

Durch Vererbung können Sie sich die Programmierarbeit wesentlich erleichtern, indem Sie spezialisierte Subklassen verwenden, die den Code zum großen Teil von einer allgemeinen Basisklasse erben. Die Subklassen heißen auch *abgeleitete Klassen, Kind*- oder *Unterklassen*, die Basisklasse wird auch als *Super*- oder *Elternklasse* bezeichnet. In den Subklassen können Sie bestimmte Funktionalitäten überschreiben, um spezielle Prozesse auszuführen.

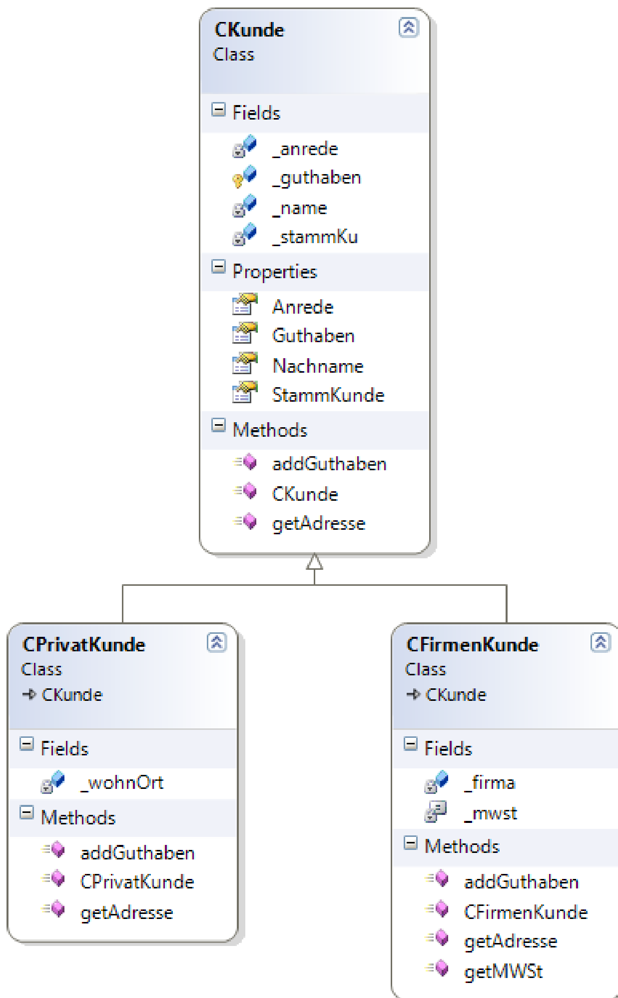
Lassen Sie uns anhand eines kurzen und dennoch ausführlichen Beispiels die wichtigsten Vererbungstechniken demonstrieren! Wir beginnen mit dem Klassendiagramm.

3.6.1 Klassendiagramm

Mittels *Unified Modeling Language* (UML) lassen sich Vererbungsbeziehungen zwischen verschiedenen Klassen grafisch darstellen.

Das folgende, mit Visual Studio erzeugte, Klassendiagramm zeigt eine Basisklasse *CKunde*, von der die Klassen *CPrivatKunde* und *CFirmenKunde* "erben". Die Basisklasse hat die Eigenschaften *Anrede*, *Nachname*, *StammKunde* (ja/nein) und *Guthaben* und die Methoden *getAdresse()* und

addGuthaben() (das Guthaben ist hier als Bonus zu verstehen, der den Kunden in prozentualer Abhängigkeit von den getätigten Einkäufen gewährt wird). Die Methode *CKunde* ist nichts weiter als der Konstruktor.



3.6.2 Method-Overriding

Die Subklassen *CPrivatKunde* und *CFirmenKunde* können auf sämtliche Eigenschaften und Methoden der Basisklasse zugreifen und fügen selbst eigene Methoden (auch Eigenschaften wären natürlich möglich) hinzu.

Die "geerbten" Methoden *getAdresse* und *addGuthaben* sind in unserem Beispiel allerdings so genannte *überschriebene Methoden* (*Method-Overriding*), d.h., Adresse und Guthaben sollen für Privatkunden auf andere Weise als für Firmenkunden ermittelt werden. Genaueres dazu erfahren Sie im nächsten Abschnitt.

3.6.3 Klassen implementieren

Vorbild für die drei zu implementierenden Klassen ist obiges Klassendiagramm.

Basisklasse CKunde

Die Deklaration entspricht (fast) der einer normalen Klasse. Dass es sich um eine Basisklasse handelt, erkennt man in unserem konkreten Fall eigentlich nur an dem *protected*-Feld und an den *virtual*-Methodendeklarationen[1].

```
public class CKunde                // Basisklasse
{
```

Die privaten Felder:

```
    private string _anrede;
    private string _name;
    private bool _stammKu;
```

Durch den *protected*-Modifizierer für das *guthaben*-Feld wird es möglich, dass auch die beiden Subklassen auf dieses Feld zugreifen können:

```
    protected decimal _guthaben = 0;      // Feld ist in Subklassen sichtbar!
```

Ein eigener Konstruktor ersetzt den Standardkonstruktor:

```
    public CKunde(string anr, string nName)     // Konstruktor
    {
        _anrede = anrede; _name = nName;
    }
```

Die Eigenschaften:

```
    public string Anrede
    {
         get { return (_anrede); }
         set { _anrede = value; }
    }

    public string Nachname
    {
         get { return (_name); }
         set { _name = value; }
    }
    public bool StammKunde
    {
       get {return(_stammKu); }
       set {_stammKu = value; }
    }
```

[1] Eigentlich hätten wir unsere Klasse auch noch als *abstract* deklarieren müssen (siehe dazu Abschnitt 3.7.1).

```
     public decimal Guthaben        // ReadOnly
     {
         get {return(_guthaben); }
     }
```

Nun zu den beiden Methoden, die durch die Subklassen überschrieben werden können.

Der Rückgabewert der virtuellen Methode *getAdresse()* setzt sich aus Anrede und Namen des Kunden zusammen:

```
    public virtual string getAdresse()                    // virtuelle Methode
    {
        string s = _anrede + " " + _name;
        return(s);
    }
```

Die virtuelle Methode *addGuthaben()* erhöht das Guthaben des Kunden um den im Argument übergebenen Bonusbetrag. Allerdings kommen nur Stammkunden in diesen Genuss:

```
    public virtual void addGuthaben(decimal betrag)      // virtuelle Methode
    {
        if (_stammKu) _guthaben += betrag;
    }
}
```

Subklasse CPrivatKunde

Diese Klasse erbt alle Eigenschaften und Methoden der Basisklasse, wird also sozusagen um deren Code "erweitert". Das *override*-Schlüsselwort der beiden Methoden bedeutet, dass hier die in der Basisklasse als *virtual* definierten Funktionen überschrieben werden. Das erlaubt der Subklasse, eine eigene Implementierung der Funktionen zu realisieren.

```
public class CPrivatKunde : CKunde             // erbt von der Basisklasse CKunde!
{
   private string _wohnOrt;
```

Der Konstruktor ist unbedingt notwendig, weil auch die Basisklasse einen eigenen Konstruktor verwendet. Es wird das *base*-Schlüsselwort benutzt, um den Konstruktor der Basisklasse aufzurufen.

```
    public CPrivatKunde(string anrede, string name, string ort): base(anrede, name)
    {
        _wohnOrt = ort;              // klassenspezifische Ergänzung
    }
```

Die Methode *getAdresse()* wird so überschrieben, dass zusätzlich zu Anrede und Name (von der Basisklasse geerbt) noch der Wohnort des Privatkunden angezeigt wird.

```
    public override string getAdresse()
    {
        const char LF = (char) 10;       // Zeilenvorschub
        return(base.adresse() + LF + _wohnOrt);
    }
```

Die Methode *addGuthaben()* wird komplett neu überschrieben. Ohne Rücksicht auf die Zugehörigkeit zur Stammkundschaft werden jedem Privatkunden 5% vom Rechnungsbetrag als Bonusguthaben angerechnet:

```
    public override void addGuthaben(decimal geld)
    {
         // Zugriff auf protected-Variable in Basisklasse:
         _guthaben += 0.05M * geld;
    }
}
```

Subklasse CFirmenKunde

Der Code für die Subklasse *CFirmenKunde* unterscheidet sich in folgenden Details von der Klasse *CPrivatKunde*:

- Die Methode *getAdresse()* liefert statt des Wohnorts den Namen der Firma des Kunden.
- Die *addGuthaben()*-Methode berechnet zunächst den Nettobetrag und addiert davon 1% zum Bonusguthaben. Damit nur Stammkunden in den Genuss dieser Vergünstigung kommen, wird dazu die gleichnamige Methode der Basisklasse aufgerufen.
- Die neu hinzugekommene "stinknormale" Methode *getMWSt()* erlaubt einen Lesezugriff auf die Mehrwertsteuer-Konstante.

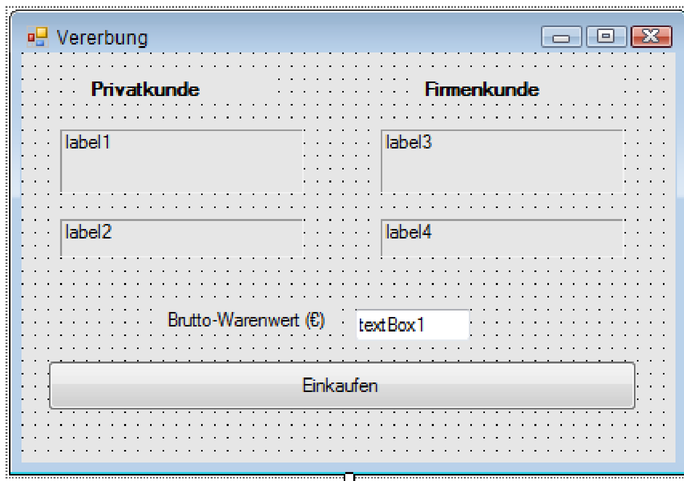
```
public class CFirmenKunde : CKunde
{
    private string _firma;
    private  const float _mwst = 0.19F;     // Mehrwertsteuer

    public CFirmenKunde(string anrede, string name, string frm): base(anrede, name)
    {
        _firma = frm;
    }

    public override string adresse()
    {
        const char LF = (char) 10;      // Zeilenvorschub
        return(base.adresse() + LF + _firma);
    }

    public override void addGuthaben(decimal brutto)
    {
        decimal netto = brutto / Convert.ToDecimal(1 + _mwst);
        base.addGuthaben(netto * 0.01M);   // Aufruf der Methode der Basisklasse
    }

    public double getMWSt()    // eine ganz normale Methode
    {
```

```
        return(_mwst);
    }
}
```

Die Implementierung unserer drei Klassen ist geschafft!

Testoberfläche

Um die Funktionsfähigkeit der drei Klassen zu testen, gestalten Sie die folgende Benutzerschnittstelle:



3.6.4 Implementieren der Objekte

Für einen kleinen Test genügt es, wenn wir mit nur zwei Objekten (ein Privat- und ein Firmenkunde) arbeiten.

```
CPrivatKunde kunde1;
CFirmenKunde kunde2;
```

Im Konstruktor des Formulars werden die beiden Objekte erzeugt. Die Ja-/Nein-Eigenschaft *StammKunde* muss allerdings extra zugewiesen werden, da es dazu keinen passenden Konstruktor gibt.

```
public partial class Form1 : Form
{ ...

    public Form1()
    {
        InitializeComponent();
        kunde1 = new CPrivatKunde("Herr", "Krause", "Leipzig");
        kunde1.StammKunde = false;
        kunde2 = new CFirmenKunde("Frau", "Müller", "Master Soft GmbH");
        kunde2.StammKunde = true;
```

```
        textBox1.Text = "100";
    }
```

Bei Klick auf den "Einkaufen"-Button werden für jedes Objekt diverse Eigenschaften abgefragt und Methoden aufgerufen:

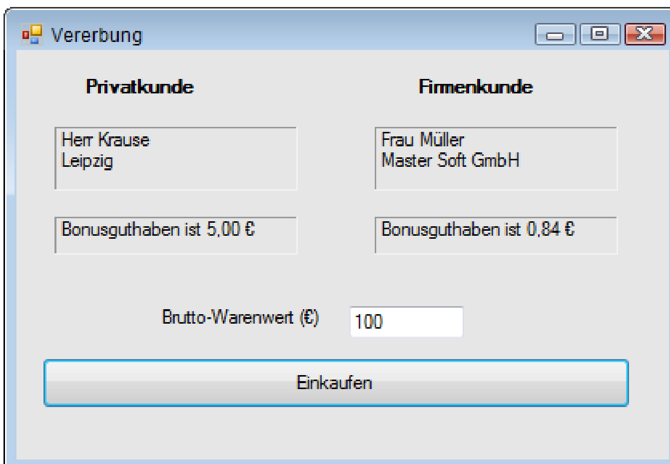
```
    private void button1_Click(object sender, EventArgs e)
    {
        decimal brutto = Convert.ToDecimal(textBox1.Text);
        label1.Text = kunde1.getAdresse();
        kunde1.addGuthaben(brutto);
        label2.Text = "Bonusguthaben ist " + kunde1.Guthaben.ToString("C");

        label3.Text = kunde2.getAdresse();
        kunde2.addGuthaben(brutto);
        label4.Text = "Bonusguthaben ist " + kunde2.guthaben.ToString("C");
    }
}
```

Praxistest

Überzeugen Sie sich nun davon, dass die drei Klassen wie gewünscht zusammenarbeiten und dass Vererbung tatsächlich funktioniert.

Die Werte in der folgenden Laufzeitabbildung sind wie folgt zu interpretieren:

- Dem Privatkunden Krause wurde ein Guthaben von 5 € (5% aus 100 €) zugebilligt (Stammkundschaft spielt bei Privatkunden keine Rolle, da die Methode *addGuthaben()* komplett überschrieben ist).
- Frau Müller ist eine Firmenkundin und erhält – nur weil sie Stammkundin ist – ein mickriges Guthaben von 0,84 € (1% auf den Nettowert).
- Durch wiederholtes Klicken auf "Einkaufen" kumulieren die Bonusguthaben.



3.6.5 Ausblenden von Mitgliedern durch Vererbung

Durch in eine abgeleitete Klasse oder Struktur eingeführte Mitglieder (Konstanten, Felder, Eigenschaften, Methoden, Ereignisse oder Typen) werden alle gleichnamigen Basisklassenelemente verdeckt bzw. ausgeblendet.

BEISPIEL 3.45: Zur Klasse *CKunde* fügen wir eine Methode *test* hinzu

C#

```
class CKunde                       // Basisklasse
{
   ...
   public void test()
   {
        MessageBox.Show("Hallo Kunde!");
   }
}
```

Eine Methode gleichen Namens fügen wir auch zur Klasse *CPrivatKunde* hinzu:

```
class CPrivatKunde : CKunde     // abgeleitete Klasse
{
   ...
   public void test()
   {
     MessageBox.Show("Hallo PrivatKunde!");
   }
}
```

Im Quellcode-Editor erscheint der Name *test()* grün unterschlängelt. Der entsprechende Warnhinweis lautet: *"WindowsFormsApplication1.CPrivatKunde.test()" blendet den versteckten Member "WindowsFormsApplication1.CKunde.test()" aus. Verwenden Sie das new-Schlüsselwort, wenn das Ausblenden vorgesehen war.*

Der Test erfolgt in *Form1*:

```
public partial class Form1 : Form
{
    private CPrivatKunde kunde1 = new CPrivatKunde();
    ...
    private void button1_Click(object sender, EventArgs e)
    {
       ...
       kunde1.test();
    }
 }
```

BEISPIEL 3.45: Zur Klasse *CKunde* fügen wir eine Methode *test* hinzu

Ergebnis

Wie Sie sehen, wurde die Methode *test()* der Klasse *CKunde* durch die Methode *test()* der Klasse *CPrivatKunde* ausgeblendet:

Um Missverständnissen vorzubeugen (und um obigen Warnhinweis im Quellcode zu vermeiden), sollte man die gleichnamige Methode in der abgeleiteten Klasse mit dem *new*-Schlüsselwort markieren.

BEISPIEL 3.46: Die folgende Änderung macht das Vorgängerbeispiel transparenter (Ergebnis bleibt dasselbe)

C#

```
...
class CPrivatKunde : CKunde     // abgeleitete Klasse
{
   ...
   new public void test()
   {
     MessageBox.Show("Hallo PrivatKunde!");
   }
}
```

HINWEIS: Die Deklaration eines neuen Mitglieds verdeckt ein geerbtes Mitglied nur innerhalb des Gültigkeitsbereichs des neuen Mitglieds!

3.6.6 Allgemeine Hinweise und Regeln zur Vererbung

Nachdem wir nun am praktischen Beispiel die Programmierung von Vererbungsbeziehungen kennengelernt haben, werden wir auch die folgenden Regeln und Hinweise verstehen:

- Alle öffentlichen Eigenschaften und Methoden der Basisklasse sind auch über die abgeleiteten Subklassen verfügbar.
- Methoden der Basisklasse, die von den abgeleiteten Subklassen überschrieben werden dürfen (so genannte *virtuelle Methoden*), müssen mit dem Schlüsselwort *virtual* deklariert werden.
- Fehlt das Schlüsselwort *virtual* bei der Methodendeklaration, so bedeutet das, dass dies die einzige Implementierung der Methode ist.
- Methoden der Subklassen, welche die gleichnamige Methode der Basisklasse überschreiben, müssen mit dem Schlüsselwort *override* deklariert werden.

- Wenn Sie das *override*-Schlüsselwort in der Subklasse vergessen, wird angenommen, dass es sich um eine "Schattenfunktion" der originalen Funktion handelt. Eine solche Funktion hat denselben Namen wie das Original, überschreibt dieses aber nicht.
- Private Felder der Basisklasse, auf die die Subklassen zugreifen dürfen, müssen mit *protected* deklariert werden.
- Die Basisklasse wird der Subklasse durch einen der Klassendeklaration nachgestellten Doppelpunkt bekannt gemacht:

  SYNTAX:
  ```
  class SubKlasse : BasisKlasse
  {
      // ... Implementierungscode
  }
  ```

- Eine Subklasse kann immer nur von einer einzigen Basisklasse abgeleitet werden (keine multiple Vererbung möglich).
- Mit dem *base*-Objekt kann von den Subklassen auf die Basisklasse zugegriffen werden, mit dem *this*-Objekt auf die eigene Klasse.
- Wenn die Basisklasse einen eigenen Konstruktor verwendet, so müssen in den Subklassen ebenfalls eigene Konstruktoren definiert werden (Konstruktoren können nicht vererbt werden!).
- Der Konstruktor einer Subklasse muss den Konstruktor seiner Basisklasse aufrufen (*base*-Schlüsselwort).
- Falls aber die Basisklasse über keinen eigenen Konstruktor verfügt, wird der Standardkonstruktor automatisch aufgerufen, wenn ein Objekt aus einer Subklasse erzeugt wird.

Wenn Sie mit Vererbung arbeiten, sollten Sie Folgendes beachten:

- Es gibt keinerlei Beschränkung bezüglich der Stufenanzahl der Vererbungshierarchie. Sie können die Hierarchie so tief wie nötig staffeln, die Eigenschaften/Methoden werden trotzdem durch alle Vererbungsstufen hindurchgereicht. Allgemein gilt, je weiter unten sich eine Klasse in der Hierarchie befindet, umso spezialisierter ist ihr Verhalten. Zum Beispiel eine *CHochschulKunden*-Klasse, die von einer *CSchulKunden* erbt und diese wiederum von der *CKunden*-Klasse.
- Um die Komplexität zu minimieren und die Wartbarkeit des Codes zu vereinfachen, sollten Sie die Vererbungshierarchie nicht tiefer als ca. vier Stufen staffeln.
- Jede Subklasse kann nur von einer Basisklasse erben! So kann z.B. eine *CHochSchulKunden*-Klasse nicht sowohl von der *CKunden*-Klasse und einer *CSchulKunden*-Klasse erben. Das ist in Ordnung so, denn eine solche multiple Vererbung könnte sehr schnell zu einem komplexen, unübersichtlichen und nicht mehr beherrschbaren Ungetüm entarten.

Es gibt zwei primäre Anwendungsfälle für Vererbung in Anwendungen:

- Sie verwenden Objekte unterschiedlichen Typs mit ähnlicher Funktionalität. So erben z.B. *CSchulKunden*-Klasse und *CStaatsKunden*-Klasse von der *CKunden*-Klasse.

- Sie haben gleiche Prozesse mit einer Menge von Objekten auszuführen. So erbt z.B. jeder Typ eines Geschäftsobjekts von einer Business Object(BO)-Klasse.

Sie sollten in folgenden Fällen auf Vererbung verzichten:

- Sie brauchen nur eine einzige Funktion von der Basisklasse. In diesem Fall sollten Sie die Funktion in die eigene Klasse delegieren, statt von einer anderen zu erben.
- Sie möchten alle Funktionen überschreiben. In einem solchen Fall sollten Sie eine Schnittstelle (*Interface*, siehe Abschnitt) statt Vererbung verwenden.

3.6.7 Polymorphes Verhalten

Untrennbar mit der Vererbung verbunden ist die so genannte Polymorphie (Vielgestaltigkeit). Polymorphes Verhalten bedeutet, dass erst zur Laufzeit einer Anwendung entschieden wird, welche der möglichen Methodenimplementierungen aufgerufen wird, da dies zum Zeitpunkt des Compilierens noch unbekannt ist.

Im obigen Beispiel hatten wir von den Vorzügen der Polymorphie allerdings noch keinen Gebrauch gemacht, denn Privat- und Firmenkunde wurden in einzelnen Objektvariablen gespeichert und bereits per Programmcode fest mit ihren Methoden *getAdresse()* und *addGuthaben()* verbunden.

Um Polymorphie sichtbar zu machen, müssen wir das bei der Implementierung der Objekte zielgerichtet ausnutzen. Wie wir gleich sehen werden, treten die Vorzüge von Polymorphie besonders augenscheinlich zutage, wenn Objekte unterschiedlicher Klassenzugehörigkeit nacheinander in Arrays oder Auflistungen abgespeichert werden.

BEISPIEL 3.47: Polymorphes Verhalten

C#

Wir nehmen die drei Klassen des Vorgängerbeispiels (*CKunde*, *CPrivatKunde*, *CFirmenKunde*) als Grundlage. An deren Implementierungen brauchen wir keinerlei Veränderungen vorzunehmen, denn polymorphes Verhalten ergibt sich als logische Konsequenz aus der Vererbung von Klassen. Änderungen müssen wir lediglich beim Abspeichern der Objektvariablen vornehmen.

Aus den Subklassen *CPrivatKunde* und *CFirmenKunde* wollen wir insgesamt drei Objekte (*kunde1*, *kunde2*, *kunde3*) instanziieren (ein Privatkunde, zwei Firmenkunden).

Innerhalb des Klassencodes von *Form1* deklarieren Sie:

```
private CPrivatKunde kunde1;
private CFirmenKunde kunde2, kunde3;
private CKunde[] kunden = new CKunde[3];      // Array für 3 Objekte!

private const char LF = (char) 10;            // für Zeilenvorschub
```

Im Konstruktor von *Form1* werden die notwendigen Initialisierungen vorgenommen:

```
    public Form1()
    {
        kunde1 = new CPrivatKunde("Herr", "Krause", "Leipzig");
```

BEISPIEL 3.47: Polymorphes Verhalten

```
        kunde1.StammKunde = false;
        kunde2 = new CFirmenKunde("Frau", "Müller", "Master Soft GmbH");
        kunde2.StammKunde = true;
        kunde3 = new CFirmenKunde("Herr", "Maus", "Manfreds Internet AG");
        kunde3.StammKunde = false;
```

Da das Array vom Typ der Basisklasse ist, kann es auch Objekte der Subklassen (Privat- und Firmenkunden) in wahlloser Reihenfolge aufnehmen:

```
        kunden[0] = kunde1;
        kunden[1] = kunde2;
        kunden[2] = kunde3;
        textBox1.Text = "100";
}
```

Das Array wird in einer *for*-Schleife durchlaufen und ausgelesen. Dabei werden die polymorphen Methoden (das sind die mit *virtual* bzw. *override* deklarierten) für alle Objekte aufgerufen:

```
    private void button1_Click(object sender, EventArgs e)
    {
        decimal brutto = Convert.ToDecimal(textBox1.Text);
        label1.Text = String.Empty;
        for (int i = 0; i < kunden.Length; i++)
        {
            kunden[i].addGuthaben(brutto);
            label1.Text = label1.Text + LF + kunden[i].getAdresse() + LF
                          + kunden[i].Guthaben.ToString("C");
        }
    }
```

Obwohl im Array die Objekte bunt durcheinander gewürfelt sein können, "weiß" das Programm zur Laufzeit genau, welche Implementierung der beiden polymorphen Methoden *getAdresse()* und *addGuthaben()* jeweils für Privat- und für Firmenkunden die richtige ist.

HINWEIS: Genau hier liegt der springende Punkt zum Verständnis der Polymorphie!

BEISPIEL 3.48: Polymorphes Verhalten Variante 2

C#

Eine alternative Implementierung mittels *foreach*-Schleife bringt die Polymorphie noch deutlicher ans Tageslicht, da die Methodenaufrufe nicht mit Objekten der Subklassen *CPrivatKunde*/*CFirmenkunde*, sondern mit Objekten der Basisklasse *CKunde* verknüpft sind:

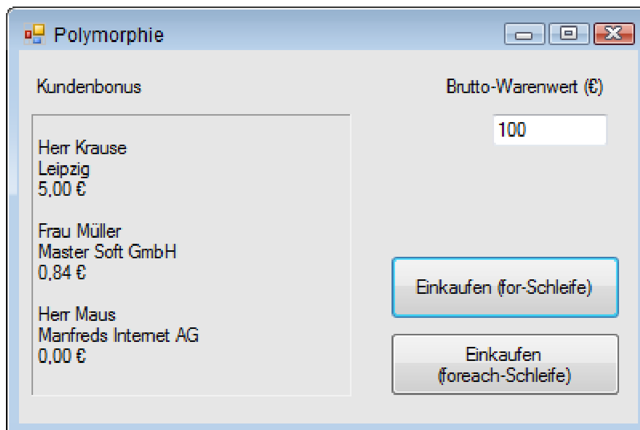
```
    private void button2_Click(object sender, EventArgs e)
    {
        decimal brutto = Convert.ToDecimal(textBox1.Text);
```

BEISPIEL 3.48: Polymorphes Verhalten Variante 2

```
        label1.Text = String.Empty;
        foreach (CKunde ku in kunden)
        {
            ku.addGuthaben(brutto);
            label1.Text = label1.Text + LF + ku.getAdresse() + LF +
                                                ku.Guthaben.ToString("C");
        }
    }
```

Praxistest

Das Ergebnis anhand der abgebildeten Testoberfläche beweist, dass Vererbung und Polymorphie tatsächlich untrennbar miteinander verbunden sind. Egal ob Privat- oder Firmenkunde – es werden immer die jeweils passenden Methodenimplementierungen aufgerufen[1].



Das tiefere Verständnis der Polymorphie ist mit Sicherheit der schwierigste Part der OOP, deshalb wurde unser Beispiel bewusst einfach gehalten, damit Sie zunächst zu einem Grundverständnis gelangen, welches Sie später weiter ausbauen können.

3.6.8 Die Rolle von System.Object

Jedes Objekt in .NET erbt von der Basisklasse *System.Object*. Diese Klasse ist Teil des Microsoft .NET Frameworks und beinhaltet die Basiseigenschaften und -methoden, wie sie für ein .NET-Objekt erforderlich sind.

Alle öffentlichen Eigenschaften und Methoden von *System.Object* stehen automatisch auch in jedem Objekt zur Verfügung, welches Sie erzeugt haben. Beispielsweise ist in *System.Object*

[1] Tja, der arme Herr Maus. Weil er kein Stammkunde ist, bekommt er auch kein Bonusguthaben.

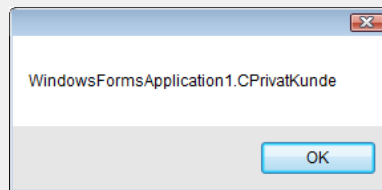
bereits ein Standardkonstruktor enthalten. Wenn Sie in Ihrem Objekt keinen eigenen Konstruktor definiert haben, wird es mit diesem Konstruktor erzeugt.

Viele der öffentlichen Eigenschaften und Methoden von *System.Object* haben eine Standardimplementation. Das heißt, Sie brauchen selbst keinerlei Code zu schreiben, um sie zu verwenden.

BEISPIEL 3.49: Die *ToString*-Methode liefert den Namen der Anwendungskomponente (die Windows-Anwendung heißt hier *Vererbung1*) und die Klassenzugehörigkeit von *kunde1*.

C#
```
MessageBox.Show(kunde1.ToString());
```

Ergebnis

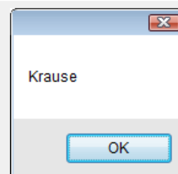


Sie können das standardmäßige Verhalten von *ToString()* mittels *override*-Schlüsselwort verändern. Dies erlaubt Ihnen eine individuelle Implementierung einiger Eigenschaften bzw. Methoden von *System.Object*.

BEISPIEL 3.50: Die gleiche *ToString()*-Methode des Vorgängerbeispiels liefert nun den Namen des Kunden, wenn Sie die folgende Methode zum Klassenkörper von *CKunde* hinzufügen.

C#
```
public override string ToString()
{
   return (_name);
}
```

Ergebnis



3.7 Spezielle Klassen

Es gibt einige Klassen, die spezielle Features aufweisen, bzw. nur für einen beschränkten Einsatzbereich infrage kommen. Gekennzeichnet werden diese Klassen in der Regel durch einen vorangestellten Modifizierer (*abstract*, *sealed*, *partial*, *static*).

3.7.1 Abstrakte Klassen

Klassen, die lediglich ihr "Erbmaterial" an andere Klassen weitergeben und von denen selbst keine Instanzen gebildet werden, bezeichnet man als *abstrakt*. Typische Beispiele für abstrakte Klassen wären *Fahrzeug*, *Tier* oder *Nahrung*.

Um zu verhindern, dass von abstrakten Klassen Instanzen gebildet werden, müssen diese mit dem Modifikator *abstract* gekennzeichnet werden.

BEISPIEL 3.51: In unserem Vorgängerbeispiel werden von der Klasse *CKunde* keine Instanzen gebildet, sie kann deshalb als abstrakt deklariert werden.

C#

```
public abstract class CKunde
{ ... }
```

Während die Referenzierung nach wie vor möglich ist

```
CKunde kunde;
```

schlägt der Versuch einer Instanziierung fehl:

```
kunde = new CKunde("Herr", "Krause");              // Fehler
```

Abstrakte Klassen ähneln einem weiteren wichtigen Softwarekonstrukt der OOP, den Schnittstellen (Interfaces). Eine Schnittstelle müssen Sie sich wie eine abstrakte Klasse vorstellen, in welcher nur öffentliche Methoden definiert, aber nicht implementiert werden (die Methodenrümpfe bleiben leer). Die Deklaration einer Schnittstelle ähnelt der einer Klasse, nur dass das Schlüsselwort *class* gegen das Schlüsselwort *interface* ausgetauscht wird (siehe Abschnitt 3.8).

Abstrakte Methoden

In Verbindung mit polymorphem Verhalten finden sich innerhalb abstrakter Klassen oft auch *abstrakte Methoden*, diese enthalten grundsätzlich keinen Code, da sie in den abgeleiteten Klassen komplett mit *override* überschrieben werden. Zur Kennzeichnung abstrakter Methoden verwenden Sie das Schlüsselwort *abstract.* Die Deklaration erfolgt in einer Zeile, also ohne Rumpf.

BEISPIEL 3.52: Die Funktion *adresse()* der abstrakten *CKunde*-Klasse wird in der Subklasse *CPrivatKunde* komplett implementiert.

C#

```
public abstract class CKunde
{   ...
    public abstract string adresse();             // abstrakte Methode
    ...
}
```

Da eine abstrakte Methode implizit eine virtuelle Methode darstellt, muss sie in der Subklasse mit *override* deklariert werden.

```
public class CPrivatKunde : CKunde;
{   ...
    public override string adresse()           // überschreibt abstrakte Methode
    {
        return(_anrede + "  " + _name + "  " & _wohnort)
    }
    ...
}
```

3.7.2 Versiegelte Klassen

Wenn Sie unbedingt verhindern möchten, dass andere Programmierer von einer von Ihnen entwickelten Komponente weitere Subklassen ableiten, so müssen Sie Ihre Klasse mit Hilfe des Modifikators *sealed* schützen.

BEISPIEL 3.53: Die Klasse *CPrivatKunde* wird versiegelt und darf deshalb keine Nachkommen haben.

C#

```
public sealed class CPrivatKunde : CKunde
{    ...
}
```

Beim Versuch, davon eine Subklasse abzuleiten, schlägt Ihnen der Compiler erbarmungslos auf die Pfoten:

```
public class CStudent : CPrivatKunde         // Fehler!!!
{
...
}
```

HINWEIS: Vererbungsmodifikatoren wie *abstract* und *virtual* führen in einer versiegelten Klasse zum Compilerfehler, da sie keinen Sinn ergeben!

Übrigens: Ein bekanntes Beispiel für eine versiegelte Klasse ist der *string*-Datentyp, was jedweden Begehrlichkeiten einen Riegel vorschiebt.

3.7.3 Partielle Klassen

Das Konzept partieller Klassen ermöglicht es, den Quellcode einer Klasse auf mehrere einzelne Dateien aufzusplitten. In Visual Studio wird zum Beispiel auf diese Weise der vom Designer automatisch erzeugte Layout-Code vom Code des Entwicklers getrennt, was zu einer gesteigerten Übersichtlichkeit beiträgt, wovon man sich nach Öffnen eines neuen Windows Forms Projekts selbst überzeugen kann (siehe auch Code-Beside-Modell von ASP.NET).

Die Programmierung ist denkbar einfach, denn alle Teile der Klasse sind lediglich mit dem Modifizierer *partial* zu kennzeichnen, dieser muss hinter dem Sichtbarkeitsmodifizierer (*private, public, protected*, ...) stehen.

BEISPIEL 3.54: Eine einfache Klasse *CKunde*

C#

```
public class CKunde
{
        private string _name;
        protected decimal _guthaben = 0;

        public CKunde(string nachName)
        { _name = nachName; }
```

BEISPIEL 3.54: Eine einfache Klasse *CKunde*

```
        public string Name
        { get { return (_name); } }

        public decimal Guthaben
        { get { return (_guthaben); } }

        public void addGuthaben(decimal betrag)
        { _guthaben += betrag; }
}
...
```

Die Klasse *CKunde* könnte (als eine von vielen Möglichkeiten) wie folgt in drei partielle Klassen aufgesplittet werden:

```
public partial class CKunde
{
        private string _name;
        protected decimal _guthaben = 0;

        public CKunde(string nachName)
        { _name = nachName; }
}

public partial class CKunde
{
        public string Name
        { get { return (_name); } }
        public decimal Guthaben
        { get { return (_guthaben); } }
}

public partial class CKunde
{
        public void addGuthaben(decimal betrag)
        { _guthaben += betrag; }
}
```

3.7.4 Statische Klassen

Mit dem *static*-Modifizierer kann man nicht nur statische Klassenmitglieder (Eigenschaften, Methoden, siehe 3.2.5 und 3.3.3), sondern auch komplette *statische* Klassen deklarieren. Eine solche Klasse kann nicht instanziiert werden und hat nur statische Mitglieder. Praktische Anwendungen ergeben sich z.B. für Formelsammlungen. Auch in der .NET-Klassenbibliothek finden sich zahlreiche vordefinierte statische Klassen (z.B. *File*, *Directory*, *Debug*, ...).

BEISPIEL 3.55: Eine statische Klasse zur Berechnung des Kugelvolumens bei gegebenem Durchmesser (und umgekehrt)

C#

```
public static class CKugel
{
    private static double Pi = Math.PI;

    public static double Durchmesser_Volumen(double durchmesser)
    {
        double vol = Math.Pow(durchmesser, 3) * Pi/6.0;
        return vol;
    }

    public static double Volumen_Durchmesser(double volumen)
    {
        double dur = Math.Pow(6/Pi * volumen, 1/3.0);
        return dur;
    }
}
```

Beispielhafte Anwendung der statischen Klasse *CKugel* zur Berechnung des Durchmessers einer Kugel mit 1 Kubikmeter Rauminhalt:

```
double v = 1.0d;
double d = CKugel.Volumen_Durchmesser(v);          // liefert 1,24...
```

Die Verwendung einer ähnlichen Klasse wird im folgenden Praxisbeispiel demonstriert:

► 3.9.2 Eine statische Klasse anwenden

3.8 Schnittstellen (Interfaces)

Das .NET-Framework (die CLR) unterstützt keine Mehrfachvererbung, d.h., eine Unterklasse kann immer nur von einer einzigen Oberklasse erben. Dies ist wohl mehr ein Segen als ein Fluch, denn allzu leicht würde sonst der Programmierer im Gestrüpp mehrfacher Vererbungsbeziehungen über mehrere Hierarchie-Ebenen hinweg die Übersicht verlieren, instabiler Code und Chaos wären die Folge.

Ein Ausweg ist die Verwendung von Schnittstellen, diese bieten fast alle Möglichkeiten der Mehrfachvererbung, vermeiden aber deren Nachteile. Schnittstellen dienen dazu, um gemeinsame Merkmale ansonsten unabhängiger Klassen beschreiben zu können.

Eine Schnittstelle können Sie sich zunächst wie eine abstrakte Klasse (siehe 3.7.1) vorstellen, in welcher nur abstrakte Methoden definiert werden[1].

[1] Dieser Vergleich hinkt natürlich wegen der auch bei abstrakten Klassen nicht möglichen Mehrfachvererbung.

3.8.1 Definition einer Schnittstelle

Eine Schnittstelle können Sie zu Ihrem Projekt genauso hinzufügen wie eine neue Klasse. Statt des Schlüsselworts *class* verwenden Sie aber *interface*.

HINWEIS: Laut Konvention sollte der Namen einer Schnittstelle mit "I" beginnen.

BEISPIEL 3.56: Eine Schnittstelle *IPerson*, die zwei Eigenschaften und eine Methode definiert

C#
```
public interface IPerson
{
    string Nachname
    { get; set; }

    string Vorname
    { get; set; }

    string getName();
}
```

Vielleicht vermissen Sie im obigen Beispiel die Zugriffsmodifizierer (*public string Nachname ...*), diese haben aber in einer Schnittstellendefinition generell nichts zu suchen.

HINWEIS: Die Festlegung der Zugriffsmodifizierer für die Mitglieder der Schnittstelle ist allein Angelegenheit der Klasse, die die Schnittstelle implementiert!

3.8.2 Implementieren einer Schnittstelle

Die Syntax entspricht der bei der normalen Implementierurngsvererbung[1], d.h., an die Deklaration der erbenden Klasse wird ein Doppelpunkt angefügt, dem der Namen der Schnittstelle folgt.

HINWEIS: Die implementierende Klasse geht die Verpflichtung ein, ausnahmslos **alle** Mitglieder der Schnittstelle zu implementieren!

BEISPIEL 3.57: Die Klasse *CKunde* implementiert die Schnittstelle *IPerson*

C#
```
class CKunde : IPerson
{
       private string _nachname;
       private string _vorname;
       ...
```

[1] Der Vergleich trifft zumindest dann zu, wenn nur eine einzige Schnittstelle implementiert wird.

BEISPIEL 3.57: Die Klasse *CKunde* implementiert die Schnittstelle *IPerson*

Alle von *IPerson* geerbten abstrakten Klassenmitglieder müssen implementiert werden:

```
        public string Nachname
        {
            get { return _nachname; }
            set { _nachname = value; }
        }

        public string Vorname
        {
            get { return _vorname; }
            set { _vorname = value; }
        }

        public override string  getName()
        {
           return _vorname + " " +_nachname ;
        }
        ...
}
```

Den kompletten Code finden Sie im Praxisbeispiel

▶ 3.9.4 Schnittstellenvererbung verstehen

Dort wird auch gezeigt, wie man eine mit abstrakten Methoden ausgestattete abstrakte Klasse ganz leicht in eine Schnittstelle überführen kann.

3.8.3 Abfragen, ob Schnittstelle vorhanden ist

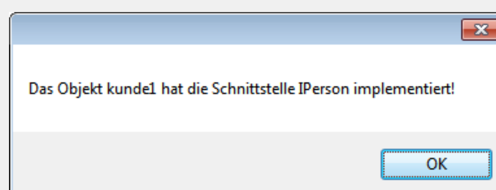
Manchmal möchte man vor der eigentlichen Arbeit mit einem Objekt wissen, ob dieses eine bestimmte Schnittstelle implementiert hat. Eine Lösung bietet eine Abfrage mit dem *is*-Operator.

BEISPIEL 3.58: Abfragen, ob das Objekt *kunde1* die Schnittstelle *IPerson* implementiert hat

C#

```
    private void button2_Click(object sender, EventArgs e)
    {
      if (kunde1 is IPerson)
          MessageBox.Show("Das Objekt kunde1 hat die Schnittstelle IPerson implementiert!");
    }
```

Ergebnis



3.8.4 Mehrere Schnittstellen implementieren

Eine Klasse kann nicht nur eine, sondern auch mehrere Schnittstellen gleichzeitig implementieren, was quasi Mehrfachvererbung bedeutet, wie sie mit der klassischen Implementierungsvererbung unmöglich ist.

BEISPIEL 3.59: Eine Klasse implementiert zwei Schnittstellen

C#
```
public class CPrivatkunde : IPerson, IKunde
{
    ...
}
```

3.8.5 Schnittstellenprogrammierung ist ein weites Feld

... und bis jetzt haben wir nur an der Oberfläche gekratzt. Wichtige Prinzipien hier nochmals in Kürze:

- Statt von einer abstrakten Klasse zu erben, werden die abstrakten Methoden über eine Schnittstelle veröffentlicht. Damit erlangt man gewissermaßen die Funktionalität der Mehrfachvererbung und umgeht deren Nachteile.
- Eine Schnittstelle ist wie ein Vertrag: Sobald eine Klasse eine Schnittstelle implementiert, muss sie auch ausnahmslos alle (!) Mitglieder der Schnittstelle implementieren und veröffentlichen.
- Der Name der implementierten Methode sowie deren Signatur muss mit deren Definition in der Schnittstelle exakt übereinstimmen.
- Mehrere Schnittstellen können zu einer neuen Schnittstelle zusammengefasst werden und selbst wieder Schnittstellen implementieren.

HINWEIS: Mehr zur Schnittstellenprogrammierung finden Sie beispielsweise im Kapitel 5 (*IEnumerable*-Interface).

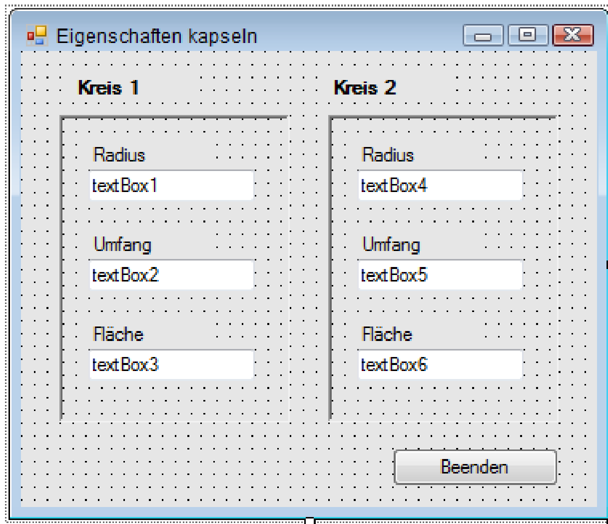
3.9 Praxisbeispiele

3.9.1 Eigenschaften sinnvoll kapseln

Das Deklarieren von Eigenschaften als öffentliche Variablen der Klasse heißt immer, das Brett an der dünnsten Stelle zu bohren. Der fortgeschrittene Programmierer verwendet stattdessen sogenannte Property-Methoden, die einen kontrollierten Zugriff erlauben. Außerdem ermöglichen die Property-Methoden auch die Implementierung von *berechneten Eigenschaften*, die aus den (privaten) Zustandsvariablen ermittelt werden. Im vorliegenden Praxisbeispiel handelt es sich um eine Klasse *CKreis* mit den Eigenschaften *radius*, *umfang* und *fläche*. Diese Klasse speichert intern eine einzige Zustandsvariable *r*, aus welcher direkt beim Zugriff alle Eigenschaften berechnet werden.

Oberfläche

Um einen großen Vorteil der OOP zu demonstrieren (ohne viel Mehraufwand lassen sich beliebig viele Instanzen einer Klasse erzeugen), wollen wir mit zwei Objekten (*kreis1* und *kreis2*) arbeiten.



Quellcode (CKreis)

Unsere Klasse wird außerhalb von *Form1* definiert, wir werden sie sogar in ein separates Klassenmodul auslagern. Wählen Sie das Menü *Projekt|Klasse hinzufügen...* und geben Sie dem Klassenmodul den Namen *CKreis.cs*.

Deklarieren und implementieren Sie nun die Klasse wie folgt:

```
public class CKreis
{
  private double r;                    // das einzige Feld (Zustandsvariable)
```

Die Rückgabewerte der Eigenschaften sind hier vom *string*-Datentyp, um die Bedienoberfläche von den lästigen Konvertierungen *string* in *double* und umgekehrt zu entlasten.

Die Eigenschaft *radius*:

```
  public string radius
  {
     get {return (r.ToString("#,#0.00")); }
     set
     {
       if (value != "")  r = Convert.ToDouble(value);
       else r = 0;
     }
  }
```

Die Eigenschaft *umfang*:

```
public string umfang
{
   get {return (2 * Math.PI * r).ToString("#,#0.00"); }
   set
   { if (value != "")  r = Convert.ToDouble(value) / 2 / Math.PI;
     else r = 0;
   }
}
```

Die Eigenschaft *fläche*:

```
public string fläche
{
   get {return (Math.PI * Math.Pow(r, 2)).ToString("#,#0.00");}
   set
   { if (value != "") r = Math.Sqrt(Convert.ToDouble(value) / Math.PI);
     else r = 0;
   }
}
```

Der eigene Konstruktor:

```
  public CKreis(double rad)
   { r = rad;  }
}
```

Quellcode (Form1)

Wechseln Sie in den Klassencode von *Form1*.

```
public partial class Form1 : Form
{  ...
```

Ein Objekt *kreis1* wird erzeugt und kann (ein Dankeschön an den Konstruktor) gleich mit dem Radius 1.0 (Maßeinheit soll hier keine Rolle spielen) initialisiert werden:

```
CKreis kreis1 = new CKreis(1.0);
```

Die folgenden Eventhandler sind einfach und übersichtlich, da die Objekte ihre inneren Funktionalitäten kapseln:

```
    private void textBox1_KeyUp(object sender, KeyEventArgs e)
    {
        kreis1.radius = textBox1.Text;
        textBox2.Text = kreis1.umfang;
        textBox3.Text = kreis1.fläche;
    }
    private void textBox2_KeyUp(object sender, KeyEventArgs e)
    {
        kreis1.umfang = textBox2.Text;
        textBox1.Text = kreis1.radius;
```

```
        textBox3.Text = kreis1.fläche;
    }

    private void textBox3_KeyUp(object sender, KeyEventArgs e)
    {
        kreis1.fläche = textBox3.Text;
        textBox1.Text = kreis1.radius;
        textBox2.Text = kreis1.umfang;
    }
    ...
}
```

Der Code für das zweite Objekt (*kreis2*) ist völlig analog zu *kreis1*, sodass hier auf die Wiedergabe des Quellcodes verzichtet werden kann (siehe Beispieldaten zum Buch).

Test

Sobald Sie eine beliebige Eigenschaft ändern, werden die anderen zwei sofort aktualisiert! Wegen der in der Klasse eingebauten Eingabeprüfung verursacht ein leerer Eingabewert keinen Fehler.

HINWEIS: Geben Sie als Dezimaltrennzeichen immer das Komma (,) ein, als Tausender-Separator dürfen Sie den Punkt (.) verwenden.

Bemerkungen

- Aus Gründen der Übersichtlichkeit wurde aber auf das Abfangen weiterer Eingaben, die sich nicht in einen numerischen Wert konvertieren lassen, verzichtet.
- Man ein Objekt auch dann erzeugen und initialisieren, wenn es dazu keinen passenden Konstruktor gibt. Sie könnten also auf den Konstruktor im Code von *CKreis* verzichten und stattdessen im Code von *Form1* die Instanziierung der Klasse durch direktes Zuweisen ihrer Eigenschaften (nicht private Zustandsvariablen!) mittels eines Objektinitialisierers wie folgt vornehmen:

```
CKreis kreis1 = new CKreis { radius = "1.0" };
```

3.9.2 Eine statische Klasse anwenden

Statische Klassen eignen sich ideal für Formelsammlungen (siehe z.B. *Math*-Klasse), da keine Objekte erzeugt werden müssen, denn es kann gleich "losgerechnet" werden. Das vorliegende Beispiel demonstriert dies anhand einer statischen Klasse *CKugel* zur Berechnung des Kugelvolumens bei gegebenem Durchmesser (und umgekehrt).

$V = 4/3 * Pi * r^3$

Nimmt man anstatt des Radius den Durchmesser *d* der Kugel, so ergibt sich daraus nach einigen Umstellungen die folgende Berechnungsformel für das Volumen *V*:

$V = d^3 * Pi/6$

Oberfläche

Ein Formular mit zwei Textfeldern zur Eingabe von Kugeldurchmesser und Kugelvolumen.

Quellcode

```
using static Math1;

public static class CKugel
{
        public static double Durchmesser_Volumen(string durchmesser)
        {
            double dur = System.Double.Parse(durchmesser);
            double vol = Math.Pow(dur, 3) * Math.PI/6.0;
            return vol;
        }

        public static double Volumen_Durchmesser(string volumen)
        {
            double vol = System.Double.Parse(volumen);
            double dur = Math.Pow(6/Math.PI * vol, 1/3.0);
            return dur;
        }
}
```

Die Verwendung der Klasse im Formularcode:

```
using static CKugel;

public partial class Form1 : Form
{  ...
```

Die Berechnung startet nach Betätigen der Enter-Taste:

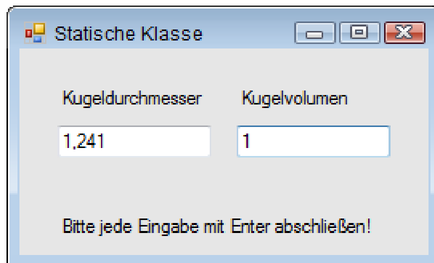
```
         private void textBox1_KeyUp(object sender, KeyEventArgs e)
         {
            if ((e.KeyCode == Keys.Enter) && (textBox1.Text != ""))
             textBox2.Text = Durchmesser_Volumen(textBox1.Text).ToString("#,##0.000");
        }

        private void textBox2_KeyUp(object sender, KeyEventArgs e)
        {
            if ((e.KeyCode == Keys.Enter) && (textBox2.Text != ""))
                textBox1.Text = Volumen_Durchmesser(textBox2.Text).ToString("#,##0.000");
        }
}
```

---

[1] Die *using static*-Anweisung gehört zu den Neuerungen von C# 6.0, sie lohnt sich allerdings erst dann, wenn wiederholt auf verschiedene Member einer statischen Klasse zugegriffen werden muss.

Test

Es ist egal, ob Sie den Radius oder das Volumen eingeben. Nach Betätigen der *Enter*-Taste wird der Inhalt des jeweils anderen Textfelds sofort aktualisiert.



Die Maßeinheit spielt bei der Programmierung keine Rolle, da sie für beide Eingabefelder identisch ist. Um beispielsweise einen Wasserbehälter mit 1 Kubikzentimeter Inhalt zu realisieren, ist eine Kugel mit dem Durchmesser von 1,241 Zentimetern erforderlich, für 1 Kubikmeter (1000 Liter) wären es 1,241 Meter.

3.9.3 Vom fetten zum schlanken Client

Lassen Sie sich durch den martialischen Titel nicht irritieren, wir wollen damit lediglich Ihr Interesse für eine Methodik wecken, die Ihnen hilft, den Horizont herkömmlicher Programmiertechniken zu überschreiten und damit einen leichteren Zugang zur OOP ermöglicht. Dabei gehen wir von folgender Erfahrung aus:

Typisch für den OOP-Ignoranten ist, dass er getreu der Devise "Hauptsache es funktioniert" mit Ausdauer und Beharrlichkeit immer wieder so genannte "fette" Clients (Fat Clients) programmiert. In einem solchen *Fat Client* ist in der Regel die gesamte Intelligenz (Geschäfts- bzw. Serverlogik) der Anwendung konzentriert, d.h., eine Aufteilung in Klassen bzw. Schichten hat nie stattgefunden.

Ein qualifizierter objektorientierter Entwurf zeichnet sich aber dadurch aus, dass der Client möglichst "dumm" bzw. "dünn" ist. Ein *Thin Client* verwaltet ausschließlich das User-Interface, die Aufgaben beschränken sich auf die Entgegennahme der Benutzereingaben und deren Weiterleitung an die Geschäftslogik bzw. umgekehrt auf die Ausgabe und Anzeige der von der Geschäftslogik ermittelten Ergebnisse. Der Server hingegen umfasst die Geschäftslogik und kapselt die Intelligenz der Anwendung.

Die Vorteile einer solchen mehrschichtigen "Thin Client"-Strategie sind:

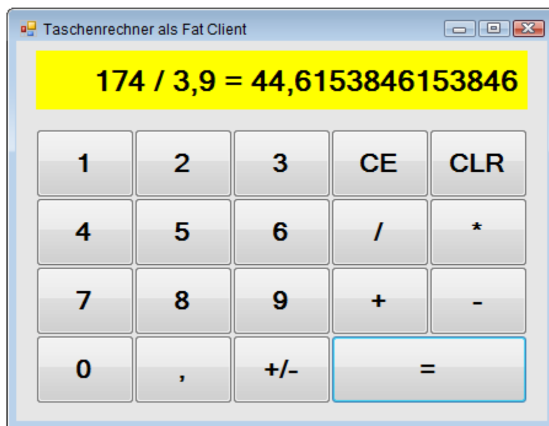
- gesteigerte Übersichtlichkeit und leichte Wiederverwendbarkeit der Software,
- Realisierung als verteilte Anwendung im Netzwerk ist möglich,
- Wartbarkeit und Erweiterbarkeit der Geschäftslogik sind möglich, ohne dass die Clients geändert werden müssten.

In unserem zweiteiligen Beispiel geht es um einen einfachen "Taschenrechner", den wir in zwei Versionen realisieren wollen.

In unserer ersten Windows Forms-Anwendung haben wir es mit einem Musterbeispiel für einen "fetten" Client zu tun. Im zweiten Teil verwandeln wir das Programm in eine mehrschichtige Anwendung mit einem "dünnen" Client. Neugierig geworden?

Oberfläche

So oder ähnlich sollte unser "Rechenkünstler" (egal ob dick oder dünn) zur Laufzeit aussehen:



Quellcode (Fat Client)

```
public partial class Form1 : Form
{   ...
```

Über die Bedeutung der folgenden drei globalen Zustandsvariablen brauchen wir wohl keine weiteren Worte zu verlieren:

```
    private string op;                      // aktueller Operator (+, - , *, /)
    private string reg1 = String.Empty;     // erstes Register   (Operand)
    private string reg2 = String.Empty;     // zweites Register  (Operand)
```

Wir wollen zur Steuerung des Programmablaufs eine spezielle Variable *state* verwenden, die den aktuellen Zustand speichert:

```
    private byte state = 1;           // aktuelles Register (1 oder 2)
```

Die nun folgenden Ereignisbehandlungen für die Schaltflächen des Ziffernblocks und die vier Rechenoperationen lassen wir diesmal nicht, wie ansonsten üblich, automatisch von der Entwicklungsumgebung generieren, sondern programmieren sie "per Hand". Grund dafür ist der geringere Schreibaufwand, da gemeinsame Eventhandler benutzt werden. Wir ergänzen also den Konstruktorcode mit dem Anmelden der entsprechenden Eventhandler (würden Sie das Anmelden wie üblich der IDE überlassen, so fänden Sie den automatisch generierten Code in der partiellen Klasse *Form1.Designer.cs*).

```
public Form1()
{
    InitializeComponent();
```

Die erste Gruppe bezieht sich auf den Ziffernblock und verweist auf die gemeinsam genutzte Methode *buttonZ_Click*:

```
    this.button1.Click += new EventHandler(this.buttonZ_Click);        // 1
    this.button2.Click += new EventHandler(this.buttonZ_Click);        // 2
    this.button3.Click += new EventHandler(this.buttonZ_Click);        // 3
    this.button4.Click += new EventHandler(this.buttonZ_Click);        // 4
    this.button5.Click += new EventHandler(this.buttonZ_Click);        // 5
    this.button6.Click += new EventHandler(this.buttonZ_Click);        // 6
    this.button7.Click += new EventHandler(this.buttonZ_Click);        // 7
    this.button8.Click += new EventHandler(this.buttonZ_Click);        // 8
    this.button9.Click += new EventHandler(this.buttonZ_Click);        // 9
    this.button10.Click += new EventHandler(this.buttonZ_Click);       // 0
    this.button11.Click += new EventHandler(this.buttonZ_Click);       // ,
```

Die zweite Gruppe bezieht sich auf die vier Operationstasten, welche auf die gemeinsame Methode *buttonOp_Click* verweisen:

```
    this.button12.Click += new EventHandler(this.buttonOp_Click);  // +
    this.button13.Click += new EventHandler(this.buttonOp_Click);  // -
    this.button14.Click += new EventHandler(this.buttonOp_Click);  // *
    this.button15.Click += new EventHandler(this.buttonOp_Click);  // :
}
```

Typisch für die nun folgenden Implementierungen der Ereignisbehandlungen *buttonZ_Click* und *buttonOp_Click* ist, dass die durchgeführten Aktionen vom Wert der Zustandsvariablen *state* abhängig sind, die somit die Rolle einer "Programmweiche" übernimmt.

Zunächst die Zifferneingabe:

```
private void buttonZ_Click(object sender, EventArgs e)
{
    Button cmd = (Button)sender;
```

In Abhängigkeit von *state* wird die Eingabe zum ersten oder zum zweiten Register hinzugefügt. Der Wert der einzugebenden Ziffer wird dabei der *Text*-Eigenschaft des Buttons entnommen:

```
    switch (state)
    {
        case 1:                // zum ersten Operanden hinzufügen
            reg1 += cmd.Text[0];
            label1.Text = reg1; break;
        case 2:                // zum zweiten Operanden hinzufügen
            reg2 += cmd.Text[0];
            label1.Text = reg1 + " " + op + " " + reg2; break;
    }
}
```

Bei der Eingabe des Operators (+, -, *, /) wird ähnlich verfahren:

```
private void buttonOp_Click(object sender, EventArgs e)
{
    Button cmd = (Button) sender;
    switch (state)
    {
        case 1:                        // neuer Operand
            op = cmd.Text;
            state = 2; break;
        case 2:
            ergebnis();                // Zwischenergebnis mit altem Operand ermitteln
            op = cmd.Text; break;      // ... dann neuer Operand
    }
    label1.Text = reg1.ToString() + " " + op;
    reg2 = String.Empty;              // Register2 löschen
}
```

Die folgende Hilfsprozedur führt die Rechenoperation aus und speichert deren Ergebnis im Register1:

```
private void ergebnis()
{
    double r1 = Convert.ToDouble(reg1);
    double r2 = Convert.ToDouble(reg2);
    switch (op)
    {
        case "+":
            reg1 = (r1 + r2).ToString(); break;
        case "-":
            reg1 = (r1 - r2).ToString(); break;
        case "*":
            reg1 = (r1 * r2).ToString(); break;
        case "/":
            reg1 = (r1 / r2).ToString(); break;
    }
    reg2 = String.Empty;              // löscht Register2
}
```

Die Ergebnistaste (=):

```
private void buttonResult_Click(object sender, EventArgs e)
{
    if (state == 2)
    {
        ergebnis();
        label1.Text += " = " + reg1;
        state = 1;
    }
    else
    {
```

```
            label1.Text = reg1;
            reg2 = "";            // löscht Register2
        }
    }
```

Letztes eingegebenes Zeichen löschen (CE):

```
    private void button17_Click(object sender, EventArgs e)
    {
        switch (state)
        {
            case 1:
                if (!(reg1 == ""))
                {
                    reg1 = reg1.Remove(reg1.Length - 1, 1);
                    label1.Text = reg1;
                };
                break;
            case 2:
                if (!(reg2 == ""))
                {
                    reg2 = reg2.Remove(reg2.Length - 1, 1);
                    label1.Text = reg2;
                };
                break;
        }
    }
```

Alle Register sowie Anzeige löschen und Anfangszustand herstellen (CLR):

```
    private void buttonCLR_Click(object sender, EventArgs e)
    {
        reg1 = String.Empty;
        reg2 = String.Empty;
        label1.Text = String.Empty;
        state = 1;
    }
```

Schließlich noch der Vorzeichenwechsel (+/-):

```
    private void buttonVZ_Click(object sender, EventArgs e)
    {
        double r;
        switch (state)
        {
            case 1:
                r = -Convert.ToDouble(reg1);
                reg1 = r.ToString();
                label1.Text = reg1;
                break;
```

```
            case 2:
                r = -Convert.ToDouble(reg2);
                reg2 = r.ToString();
                label1.Text = reg1 + " " + op + " " + reg2;
                break;
        }
    }
...
}
```

Test

Der Vorzug gegenüber üblichen Rechnern sticht sofort ins Auge: Man kann den Rechenvorgang mitverfolgen, weil der komplette Ausdruck angezeigt wird (siehe obige Abbildung).

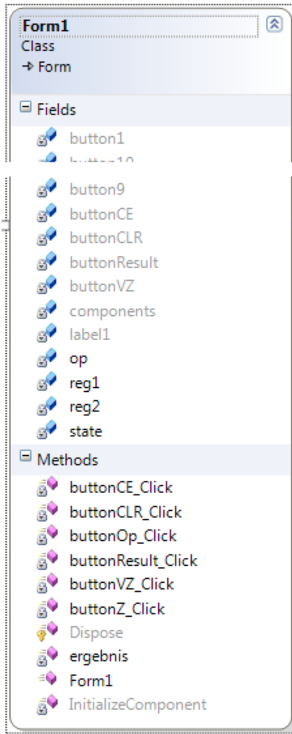
HINWEIS: Wie bei jedem anderen einfachen Taschenrechner auch, bleibt hier leider die Rangfolge der Operatoren (Punktrechnung geht vor Strichrechnung) unberücksichtigt. Bei der Eingabe von mehreren Operationen hintereinander, z.B. 3 + 4 * 12, ist deshalb zu beachten, dass erst die höherwertige Operation auszuführen ist (4 * 12).

Bemerkungen

- Den Programmablauf könnte man in Gestalt eines Zustandsüberführungsdiagramms (*State Chart*) noch anschaulicher darstellen.
- Leider ist die gesamte Intelligenz der Anwendung in der Benutzerschnittstelle *Form1* enthalten, also ein typischer "fetter" Client. Transparenz, Wiederverwendbarkeit und Wartbarkeit des Codes sind demzufolge katastrophal! Wie man das Programm auf ein höheres objektorientiertes Niveau heben kann, soll das folgende Beispiel zeigen.

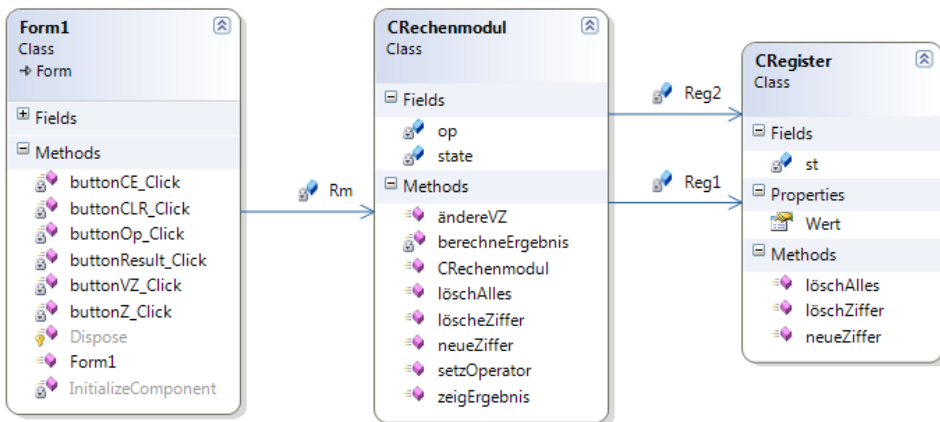
Abmagerungskur für den fetten Client

Dass es sich bei unserem Programm tatsächlich um einen Fat Client handelt, zeigt das zugehörige Klassendiagramm. Klicken Sie dazu im Projektmappen-Explorer mit der rechten Maustaste auf *Form1.cs* und wählen Sie im Kontextmenü *Klassendiagramm anzeigen*. Es vergeht eine kleine Weile und dann bietet sich Ihnen der in der Abbildung nur auszugsweise gezeigt Anblick mit einer schier endlosen Auflistung der in der Klasse implementierten Felder und Methoden.



Schluss mit diesem Chaos! Durch Auslagern der Intelligenz in die Klassen *CRechenmodul* und *CRegister* erhalten wir nach einigem Hin und Her als Ergebnis der "Abmagerungskur" schließlich das abgebildete neue Klassendiagramm (siehe folgende Abbildung).

Die Klasse *Form1* (Thin Client) ist gegenüber dem Vorgänger deutlich abgemagert und beschränkt sich nur noch auf ihre eigentliche Aufgabe, die Verwaltung der Benutzerschnittstelle. Die Klassen *CRechenmodul* und *CRegister* stellen hingegen die zweischichtige Geschäftslogik (Server) der Anwendung dar, kapseln also die Intelligenz.



HINWEIS: Damit Sie im Klassendiagramm die durch Pfeilverbindungen gekennzeichneten Zuordnungen (Assoziationen) zwischen den Klassen sehen, müssen Sie mit der rechten Maustaste auf das Feld *Rm* in der Klasse *Form1* klicken und im Kontextmenü den Eintrag *Als Zuordnung anzeigen* wählen. Analog verfahren Sie mit den Feldern *Reg1* und *Reg2* in *CRechenmodul*.

Eine Erklärung des Klassendiagramms mit anderen Worten: Unser Thin Client benutzt eine Instanz der Klasse *CRechenmodul*. In dieser wiederum sind zwei Instanzen der Klasse *CRegister* enthalten. Hier geht es also noch nicht um Vererbung, Polymorphie etc, sondern nur um das zweckmäßige "Wegkapseln" von Funktionalität, wie das in unserem Fall etwa auch der physikalischen Realität entspricht, denn auch ein "richtiger" Taschenrechner enthält ein Rechenmodul, in diesem wiederum sind ein oder mehrere Register enthalten.

Allerdings stellt diese Thin Client-Lösung nur eine von mehreren Möglichkeiten dar und ist das Ergebnis einer Analyse des Ausgangscodes nach den Kriterien der Wiederverwendbarkeit ("Code Reuse").

Quellcode für CRegister

```
public class CRegister
{
```

Die globale Variable *st* speichert den Registerinhalt als Zeichenkette:

```
    private string st = String.Empty;
```

Zugriff auf den numerischen Wert von *st:*

```
    public double Wert
    {
       get
       {
          try
             {return Convert.ToDouble(st);}
          catch
          {return 0;}
       }
       set
       {
          st = value.ToString();
       }
    }
```

Hinzufügen einer einzelnen Ziffer und Rückgabe des Anzeigestrings:

```
    public string neueZiffer(char z)
    {
        if ((Char.IsDigit(z)) || (z.ToString() == ","))
        {
            st += z;
```

```
            return (st);
        }
        else
            return "";
    }
```

Letzte Ziffer löschen und Anzeigestring zurückgeben:

```
    public string löschZiffer()
    {
        if (st.Length > 0) st = st.Remove(st.Length - 1, 1);
        return (st);
    }
```

Gesamtes Register löschen:

```
    public void löschAlles()
    {
        st = "";
    }
}
```

Quellcode für CRechenmodul

```
public class CRechenmodul
{
    private byte state = 1;                 // Startmodus (Zustandsvariable)
    private char op;                        // aktueller Operator
    private CRegister reg1, reg2;           // zwei Rechenregister
```

Im Konstruktor werden zwei Register-Objekte erzeugt:

```
    public CRechenmodul()
    {
        Reg1 = new CRegister();
        Reg2 = new CRegister();
    }
```

Zifferneingabe in aktuelles Register:

```
    public string neueZiffer(string ziff)
    {
        char z = ziff[0];
        if (state == 1)             // zum ersten Register hinzufügen
            return (Reg1.neueZiffer(z));
        else                        // zum zweiten Register hinzufügen
            return (Reg1.Wert.ToString() + " " + op + " " + Reg2.neueZiffer(z));
    }
```

Letzte Ziffer des aktuellen Registers löschen und resultierenden Registerinhalt zurückgeben:

```
    public string löscheZiffer()
    {
```

```
        if (state == 1)
        return (Reg1.löschZiffer());
        else
            return (Reg2.löschZiffer());
    }
```

Vorzeichen des aktuellen Registers umkehren:

```
    public string ändereVZ()
    {
        if (state == 1)
        {
            Reg1.Wert = -Reg1.Wert;
            return (Reg1.Wert.ToString());
        }
        else
        {
            Reg2.Wert = -Reg2.Wert;
            return (Reg1.Wert.ToString() + " " + op + " " + Reg2.Wert.ToString());
        }
    }
```

Der Operator wird übernommen. Rückgabewert ist der String des ersten Operanden mit abschließendem Operatorenzeichen:

```
    public string setzOperator(string o)
    {
        if (state == 1)
            state = 2;
        else
            berechneErgebnis();          // Zwischenergebnis (mit altem Operator) ermitteln
        op = o[0];                       // neuen Operator übernehmen
        Reg2.löschAlles();               // zweites Register löschen
        return (Reg1.Wert.ToString() + " " + op);
    }
```

Die abschließende Rechenoperation ausführen und das Ergebnis liefern:

```
    public string zeigErgebnis()
    {
        string s = "";
        if (state == 1)                  // im Startmodus wird noch nichts berechnet, ...
            Reg2.löschAlles();           // ... lediglich zweites Register wird gelöscht
        else
        {
            s = " = " + berechneErgebnis();
            state = 1;
        }
        return (s);
    }
```

Eine Hilfsmethode zum Ausführen der Rechenoperation nebst Abspeichern des Ergebnisses im ersten Register (überschreibt erstes Register mit Ergebnis der Operation):

```
    private string berechneErgebnis()
    {
        switch (op.ToString())
        {
            case "+":
                Reg1.Wert = Reg1.Wert + Reg2.Wert; break;
            case "-":
                Reg1.Wert = Reg1.Wert - Reg2.Wert; break;
            case "*":
                Reg1.Wert = Reg1.Wert * Reg2.Wert; break;
            case "/":
                Reg1.Wert = Reg1.Wert / Reg2.Wert; break;
        }
        Reg2.löschAlles();           // zweites Register löschen
        return (Reg1.Wert.ToString());
    }
```

Alle Register löschen und Startzustand wiederherstellen:

```
    public void löschAlles()
    {
        Reg1.löschAlles();
        Reg2.löschAlles();
        state = 1;
    }
}
```

Quellcode für Form1

Die Oberfläche unseres Thin Client entspricht 100%ig dem "fetten" Vorgänger. Die Programmierung ist allerdings – dank des *CRechenmodul*-Objekts – deutlich einfacher und transparenter geworden:

```
public partial class Form1 : Form
{
```

Einzige globale Variable ist der Verweis auf die Klasse *CRechenmodul*:

```
    private CRechenmodul Rm;
```

Im Konstruktorcode müssen, wie im Vorgängerbeispiel, zunächst die Eventhandler für Ziffernblock und Operationstasten angemeldet werden:

```
    public Form1()
    {
        InitializeComponent();
        // Ziffernblock:
        this.button1.Click += new EventHandler(this.buttonZ_Click);  // 1
        ...
```

Neu ist das Instanziieren der Geschäftslogik (bzw. des Servers):

```
        Rm = new CRechenmodul();
    }
```

Eine Ziffer eingeben (0..9,):

```
    private void buttonZ_Click(object sender, EventArgs e)
    {
        Button cmd = (Button) sender;
        label1.Text = Rm.neueZiffer(cmd.Text);
```

Die Operation eingeben (+, -, *, /):

```
    private void buttonOp_Click(object sender, EventArgs e)
    {
        Button cmd = (Button) sender;
        label1.Text = Rm.setzOperator(cmd.Text);
    }
```

Ergebnis anzeigen (=):

```
    private void buttonResult_Click(object sender, EventArgs e)
    {
          label1.Text += Rm.zeigErgebnis();
    }
```

Letztes eingegebenes Zeichen löschen (CE):

```
    private void buttonCE_Click(object sender, EventArgs e)
    {
        label1.Text = Rm.löscheZiffer();
    }
```

Alle Register sowie Anzeige löschen und Anfangszustand wieder herstellen:

```
    private void buttonCLR_Click(object sender, EventArgs e)
    {
        Rm.löschAlles(); label1.Text = "";
    }
```

Vorzeichenwechsel (+/-):

```
    private void buttonVZ_Click(object sender, EventArgs e)
    {
        label1.Text = Rm.ändereVZ();
    }
}
```

Test

Sie werden keinerlei Unterschied in Aussehen und Verhalten unseres "dünnen"- Taschenrechners zu seinem "fetten" Vorgänger feststellen, was uns in der Auffassung bestätigt, dass den Hauptnutzen aus der OOP nicht der Anwender, sondern der Programmierer hat!

Bemerkungen

- Unter Einsatz einer Formelparser-Klasse wären auch Klammerrechnungen möglich, das dürfte weniger aufwändig sein als das Hinzufügen weiterer Register.
- Einen wesentlich leistungsfähigeren wissenschaftlichen Taschenrechner, der allerdings nach einem völlig anderen Prinzip funktioniert (Code DOM), finden Sie im Praxisbeispiel 3.9.6 "Formel-Rechner mit dem CodeDOM".

3.9.4 Schnittstellenvererbung verstehen

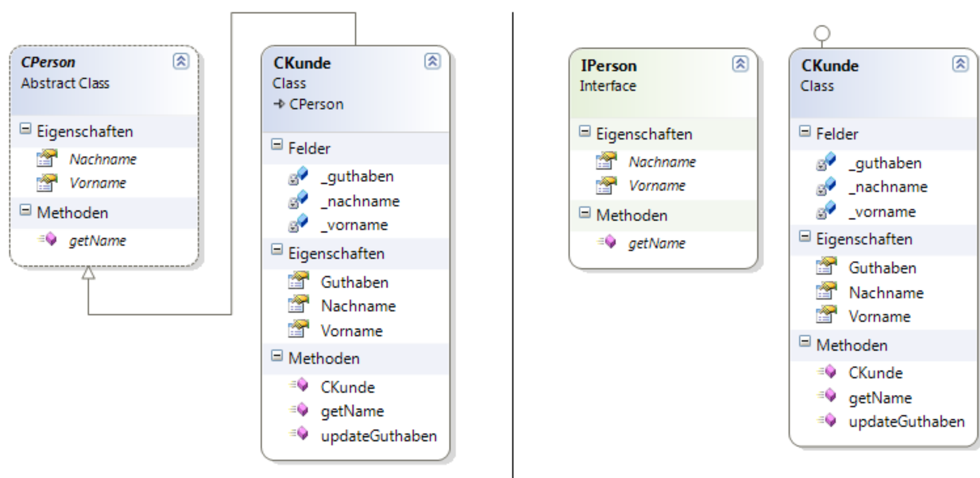
Ein mächtiges Feature der OOP ist das Konzept der Schnittstellenvererbung (siehe Abschnitt 3.8). Die Schnittstellenvererbung erschließt sich dem Einsteiger am leichtesten, wenn er sich vorher das Konzept abstrakter Klassen und Methoden (siehe Abschnitt 3.7.1) verinnerlicht hat.

In unserem kleinen Demobeispiel wollen wir Geldbeträge auf das Konto eines Kunden einzahlen bzw. von dort abheben. Die erste Lösung soll mittels einer abstrakten Klasse, die nur über abstrakte Methoden verfügt, erfolgen[1]. Die zweite Lösung benutzt eine Schnittstelle (Interface).

Klassendiagramme

Die erste Variante zeigt das links abgebildete Klassendiagramm. Hier erbt die Klasse *CKunde* von der abstrakten Klasse *CPerson*. Letztere verfügt ausschließlich über abstrakte Klassenmitglieder. Diese enthalten nur die Eigenschafts- bzw. Methodendeklaration, also keinerlei Code. Die Implementierung muss komplett in der erbenden Klasse *CKunde* erfolgen.

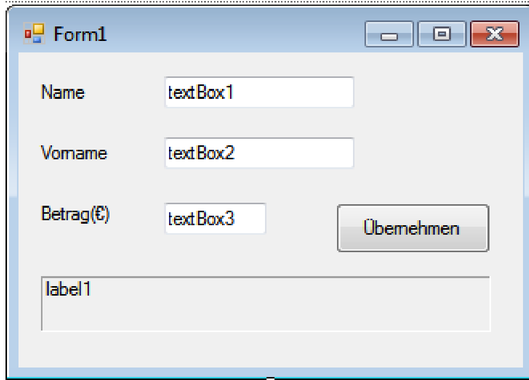
Das rechte Klassendiagramm zeigt die zweite Lösung, bei welcher die abstrakte Klasse *CPerson* von der Schnittstelle *IPerson* ersetzt wird. Weitere Unterschiede sind auf den ersten Blick nicht zu erkennen, dazu müssen wir uns den Quellcode näher anschauen.



[1] Dies ist allerdings die extremste Form der Implementierungsvererbung, denn es wird de facto keinerlei Code vererbt.

Oberfläche Form1

Öffnen Sie eine neue Windows Forms-Anwendung und gestalten Sie das Startformular wie gezeigt.



Beginnen wir mit der ersten Variante!

Quellcode CPerson

Fügen Sie dem Projekt eine neue Klasse *CPerson* zu:

```
public abstract class CPerson
{
    public abstract string Nachname
    { get; set; }

    public abstract string Vorname
    { get; set; }

    public abstract string getName();
}
```

Wie Sie sehen, ist die Klasse abstrakt und enthält die abstrakten Eigenschaften *Nachname* und *Vorname*) sowie die abstrakte Methode *getName*.

Quellcode CKunde

Fügen Sie dem Projekt eine Klasse *CKunde* hinzu:

```
class CKunde : CPerson
{
        private string _nachname;
        private string _vorname;
        private decimal _guthaben;
```

Die von *CPerson* geerbten abstrakten Klassenmitglieder müssen überschrieben werden:

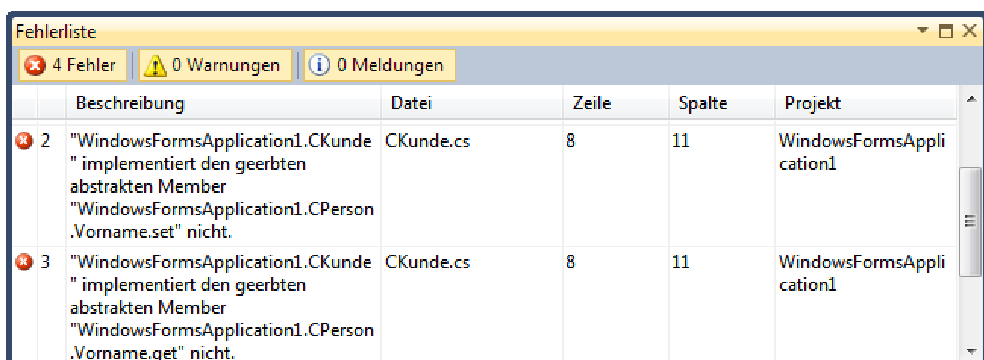
```
        public override string Nachname
```

```
        {
            get { return _nachname; }
            set {_nachname = value; }
        }

        public override string Vorname
        {
            get { return _vorname; }
            set { _vorname = value; }
        }

        public override string  getName()
        {
         return _vorname + " " +_nachname ;
        }
```

Es folgen die normalen Klassenmitglieder:

```
        public CKunde(string vor, string nach)       // ein Konstruktor
        {
            _vorname = vor; _nachname = nach;
        }

        public decimal Guthaben
        {
            get { return _guthaben; }
        }

        public decimal updateGuthaben(decimal betrag)
        {
            return _guthaben += betrag;
        }
    }
```

HINWEIS: Es müssen **alle** geerbten abstrakten Klassenmitglieder überschrieben werden, ansonsten erfolgt eine Fehlermeldung des Compilers (siehe Abbildung).



Quellcode Form1

```
...
public partial class Form1 : Form
{
```

Zu Beginn wird ein Kunde erzeugt, initialisiert und angezeigt:

```
    CKunde kunde1 = new CKunde("Max", "Müller");

    private void Form1_Load(object sender, EventArgs e)
    {
        textBox1.Text = kunde1.Vorname;
        textBox2.Text = kunde1.Nachname;
        textBox3.Text = "0";
    }
```

Bei jedem Klick auf die Schaltfläche werden Vor- und Nachname des Kunden neu zugewiesen. Der eingegebene Betrag wird dem Guthaben hinzugefügt bzw. (bei negativem VZ) abgezogen:

```
    private void button1_Click(object sender, EventArgs e)
    {
        kunde1.Vorname = textBox1.Text;
        kunde1.Nachname = textBox2.Text;
        decimal betrag = Convert.ToDecimal(textBox3.Text);
        kunde1.updateGuthaben(betrag);
```

Die abgeschlossene Buchung wird mit einem Meldungstext quittiert:

```
        label1.Text = kunde1.getName() + " hat ein Guthaben von " +
                      kunde1.Guthaben.ToString("C") + " !";
    }
}
```

Test

Nehmen Sie einige Ein- oder Auszahlungen vor.

Nach erfolgreichem Test dieser ersten Variante wollen wir die zweite Variante in Angriff nehmen und die Klasse *CPerson* durch ein Interface *IPerson* ersetzen.

Quellcode IPerson

Benennen Sie einfach im Projektmappenexplorer die Klasse *CPerson* in *IPerson* um (die Abfrage, ob auch alle Verweise geändert werden sollen, beantworten Sie positiv):

```
public interface IPerson
{
    string Nachname
    { get; set; }

    string Vorname
    { get; set; }

    string getName();
}
```

Quellcode CKunde

Auch hier sind nur minimale Änderungen erforderlich, denn bei den von *IPerson* geerbten Schnittstellenmitgliedern fehlen lediglich die *override*-Modifizierer.

```
class CKunde : IPerson
{
    ...
     public string Nachname                  // ohne override !
     {
          get { return _nachname; }
          set {_nachname = value; }
     }
     usw. ...
}
```

Das war es auch schon, denn der Quellcode von *Form1* kann unverändert bleiben!

Beim Testen des Codes werden Sie keinerlei Veränderungen zum Vorgängerprojekt feststellen, allerdings dürfte der gesamte Code etwas übersichtlicher geworden sein.

Bemerkungen

- Ebenso wie von abstrakten Klassen können auch von Schnittstellen keine Instanzen erzeugt werden.
- Es müssen ausnahmslos **alle** Schnittstellenmitglieder implementiert werden (eine Schnittstelle ist wie ein Vertrag, der bedingungslos einzuhalten ist!).

- Zugriffsmodifizierer (*public*) für die Mitglieder einer Schnittstelle fehlen komplett, denn diese haben in einer Schnittstellendefinition nichts zu suchen.
- Eine Klasse kann von mehreren Schnittstellen erben, jedoch nur von einer einzigen Klasse. Das ist der Hauptunterschied (oder auch Hauptvorteil!) der Schnittstellenvererbung gegenüber der Implementierungsvererbung.

3.9.5 Rechner für komplexe Zahlen

Auch mit dieser Beispiel wollen wir nicht nur die Lösung eines mathematischen Problems zeigen, sondern (was viel wichtiger ist) grundlegendes Handwerkszeug des .NET-Programmierers demonstrieren:

- Sinnvolle Auslagerung von Quellcode in Klassen, um das Verständnis der OOP zu vertiefen,
- Prinzip der Operatorenüberladung in C#,
- Strukturierung des Codes der Benutzerschnittstelle nach dem EVA-Prinzip (Eingabe-Verarbeitung-Ausgabe).

Doch ehe wir mit der Praxis beginnen, scheint ein kurzer Abstieg in die Untiefen der Mathematik unumgänglich.

Was sind komplexe Zahlen?

Eine besondere Bedeutung haben komplexe Zahlen beispielsweise in der Schwingungslehre und in der Wechselstromtechnik, einem bedeutenden Teilgebiet der Elektrotechnik.

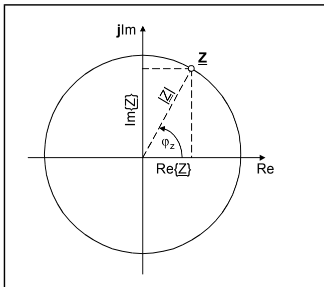
Zur Darstellung einer komplexen Zahl $\underline{Z}$ bieten sich zwei Möglichkeiten an:

- Kartesische Koordinaten (Real-/Imaginärteil)
- Polarkoordinaten (Betrags-/Winkeldarstellung)

Die folgende Tabelle zeigt eine Zusammenstellung der Umrechnungsformeln:

Kartesische Koordinaten	Polarkoordinaten
$\underline{Z} = Re\{\underline{Z}\} + jIm\{\underline{Z}\}$	$\underline{Z} = \lvert\underline{Z}\rvert\, e^{j\varphi z}$
Realteil: $Re\{\underline{Z}\} = \lvert\underline{Z}\rvert \cos \varphi_z$	Betrag: $\lvert\underline{Z}\rvert = \sqrt{(Re\{Z\})^2 + (Im\{Z\})^2}$
Imaginärteil: $Im\{\underline{Z}\} = \lvert\underline{Z}\rvert \sin \varphi_z$	Phasenwinkel: $\varphi_Z = \arctan \frac{Im\{Z\}}{Re\{Z\}}$

Am besten lassen sich diese Zusammenhänge am Einheitskreis erläutern, wobei $\underline{Z}$ als Punkt in der komplexen Ebene erscheint:



Die kartesische Form eignet sich besonders gut für die Ausführung von Addition und Subtraktion:

Mit

$\mathbf{Z_1 = a_1 + jb_1}$ und $\mathbf{Z_2 = a_2 + jb_2}$

ergibt sich

$\mathbf{Z_1 + Z_2 = a_1 + a_2 + j(b_1 + b_2)}$ bzw. $\mathbf{Z_1 - Z_2 = a_1 - a_2 + j(b_1 - b_2)}$

Andererseits bevorzugt man für Multiplikation und Division die Zeigerform:

Mit

$\mathbf{Z_1 = c_1 \cdot e^{jj\,1}}$ und $\mathbf{Z_2 = c_2 \cdot e^{jj\,2}}$

erhalten wir

$\mathbf{Z_1 \cdot Z_2 = c_1 \cdot c_2 \cdot e^{j(j\,1 + j\,2)}}$ bzw. $\mathbf{Z_1/Z_2 = c_1/c_2 \cdot e^{j(j\,1 - j\,2)}}$

Für die Angabe des Phasenwinkels hat man die Wahl zwischen Radiant (Bogenmaß) und Grad. Für die gegenseitige Umrechnung gilt die Beziehung

$$\varphi(Rad) = \frac{\pi}{180}\varphi(Grad)$$

HINWEIS: Die Maßeinheit "Grad" wird aufgrund ihrer Anschaulichkeit vom Praktiker für die Ein- und Ausgabe bevorzugt, während "Radiant" für interne Berechnungen günstiger ist.

Programmierung der Klasse CComplexN

Öffnen Sie ein neues Projekt vom Typ Windows Forms-Anwendung. Das Startformular *Form1* lassen Sie zunächst unbeachtet liegen, denn der routinierte .NET-Programmierer kapselt seinen Code in Klassen anstatt ihn einfach zum Formularcode hinzuzufügen.

Die zweckmäßige Aufteilung einer praktischen Problemstellung in verschiedene Klassen und die Definition der Abhängigkeiten ist sicherlich der schwierigste Part der OOP und erfordert einige

Übung und Routine, bis das dazu erforderliche abstrakte Denken schließlich zur Selbstverständlichkeit wird[1].

Die hier vorgeschlagene Lösung benutzt die Klasse *CComplexN*, welche eine komplexe Zahl repräsentiert. Diese Klasse speichert in den Zustandsvariablen *Re* und *Im* (die in unserem Fall gleichzeitig Eigenschaften sind) den Wert der komplexen Zahl in Kartesischen Koordinaten. Die beiden anderen Eigenschaften (*Len* und *Ang*) repräsentieren dieselbe Zahl in Polar-Koordinaten. Allerdings werden *Len* und *Ang* nicht direkt in den Objekten gespeichert, sondern in so genannten *Eigenschaftenmethoden* (*property procedures*) jeweils aus *Re* und *Im* berechnet.

Über das Menü *Projekt|Klasse hinzufügen...* erstellen Sie den Rahmencode der Klasse.

```
public class CComplexN
{
```

Die beiden öffentlichen Zustandsvariablen bilden das "Gedächtnis" der Klasse und können quasi wie Eigenschaften benutzt werden (nicht der sauberste, aber der effektivste Weg):

```
    public double Re, I;          // Real- und Imaginärteil
```

Der Konstruktor setzt die Zustandsvariablen auf ihre Anfangswerte:

```
    public CComplexN(double r, double i)
    {
        Re = r;
        Im = i;
    }
```

Die Eigenschaftenmethode *Ang* berechnet den Phasenwinkel aus den Zustandsvariablen *Re* und *Im*:

```
    public double Ang
    {
        get
        {
            double g = 0;
            if (Re != 0)
            {
               g = 180 / Math.PI * Math.Atan(Im / Re);
               if (Re < 0)   g += 180;
            }
            else
            {
                if (Im != 0)
                {
                    if (Im > 0) g = 90;
                    else g = -90;
                }
            }
            return g;
```

[1] Die UML (Unified Modelling Language) stellt dazu spezielle Werkzeuge bereit.

```
        }
        set
        {
            double b, l;
            b = value * Math.PI / 180;
            l = Math.Sqrt(Re * Re + Im * Im);
            Re = l * Math.Cos(b);        // neuer Realteil
            Im = l * Math.Sin(b);        // neuer Imaginärteil
        }
    }
```

Die Eigenschaft *Len* ermittelt den Betrag (die Länge des Zeigers) aus *Re* und *Im:*

```
    public double Len
    {
        get
        { return Math.Sqrt(Re * Re + Im * Im);  }
        set
        {
            double b;
            b = Math.Atan(Im / Re);
            Re = value * Math.Cos(b);
            Im = value * Math.Sin(b);
        }
    }
```

Besonders interessant sind die folgenden drei (statischen) Methoden, welche die Operatorenüberladungen für Addition, Multiplikation und Division neu definieren.

Der "+"-Operator erhält eine neue Bedeutung, er addiert jetzt zwei komplexe Zahlen:

```
    public static CComplexN operator +(CComplexN a, CComplexN b)
    {
        CComplexN z = new CComplexN(0, 0);
        z.Re = a.Re + b.Re;
        z.Im = a.Im + b.Im;
        return z;
    }
```

Der "*"-Operator multipliziert zwei komplexe Zahlen:

```
    public static CComplexN operator *(CComplexN a, CComplexN b)
    {
        CComplexN z = new CComplexN(0, 0);
        z.Re = a.Re * b.Re - a.Im * b.Im;
        z.Im = a.Re * b.Im + a.Im * b.Re;
        return z;
    }
```

Der "/"-Operator dividiert zwei komplexe Zahlen:

```
    public static CComplexN operator /(CComplexN a, CComplexN b)
    {
```

```
        CComplexN z = new CComplexN(0, 0);
        z.Re = (a.Re * b.Re + a.Im * b.Im) / (b.Re * b.Re + b.Im * b.Im);
        z.Im = (a.Im * b.Re - a.Re * b.Im) / (b.Re * b.Re + b.Im * b.Im);
        return z;
    }
}
```

HINWEIS: Vielleicht sticht Ihnen bereits jetzt ein gravierender Unterschied zur klassischen "Geradeausprogrammierung" ins Auge: Spezielle Methoden zur Umrechnung zwischen Kartesischen- und Polarkoordinaten sind nicht mehr erforderlich, da ein Objekt vom Typ *CComplexN* beide Darstellungen bereits als Eigenschaften kapselt!

Bedienoberfläche für Testprogramm

Um uns von der Funktionsfähigkeit der entwickelten Klassen zu überzeugen, brauchen wir ein kleines Testprogramm, das die Ein- und Ausgabe von komplexen Zahlen und die Auswahl der Rechenoperation sowie der Koordinatendarstellung ermöglicht.

Wir verwenden dazu das bereits vorhandene Startformular *Form1*, das wir entsprechend der folgenden Abbildung gestalten.

HINWEIS: Es kann nicht schaden, wenn Sie *ReadOnly* für *textBox3* und *textBox6* auf *True* und *TabStop* auf *False* setzen, da Sie diese beiden rechten Felder nur zur Ergebnisanzeige brauchen.

Quellcode für Testprogramm

Das an legendäre DOS-Zeiten erinnernde EVA-Prinzip (Eingabe, Verarbeitung, Anzeige) hat auch unter .NET nichts von seiner grundlegenden Bedeutung eingebüßt.

Der clientseitige Quellcode entspricht vom prinzipiellen Ablauf her der klassischen Geradeaus-programmierung, ist allerdings deutlich übersichtlicher und problemnäher, denn wir arbeiten mit drei Objektvariablen, die bereits komplexe Zahlen sind und nicht mit einer Vielzahl skalarer Variablen!

```
public partial class Form1 : Form
{
    ...
```

Die benötigten Objektvariablen:

```
    private CKomplexN A = new CKomplexN(1,1);            // Operand A
    private CKomplexN B = new CKomplexN(1,1);            // Operand B
    private CKomplexN Z = new CKomplexN(1,1);            // Ergebnis
```

Unter Berücksichtigung der eingestellten Anzeigeart (Rechteck- oder Polarkoordinaten) liest die folgende Methode die Werte der Eingabemaske in die Objekte:

```
    private void Eingabe()
    {
        if (radioButton4.Checked)   // Rechteck-Koordinaten
        {
            A.Re = Convert.ToDouble(textBox1.Text);
            B.Re = Convert.ToDouble(textBox2.Text);
            A.Im = Convert.ToDouble(textBox4.Text);
            B.Im = Convert.ToDouble(textBox5.Text);
        }
        else                        // Polar-Koordinaten
        {
            A.Len = Convert.ToDouble(textBox1.Text);
            B.Len = Convert.ToDouble(textBox2.Text);
            A.Ang = Convert.ToDouble(textBox4.Text);
            B.Ang = Convert.ToDouble(textBox5.Text);
        }
    }
```

Die Verarbeitungsroutine führt die gewünschte Rechenoperation mit den bekannten Symbolen für Addition, Multiplikation und Division aus. Dazu werden die in der Klasse *CcomplexN* definierten Operatorenüberladungen benutzt:

```
    private void Verarbeitung()
    {
        if (radioButton1.Checked) Z = A + B;                 // Addition
        if (radioButton2.Checked) Z = A * B;                 // Multiplikation
        if (radioButton3.Checked) Z = A / B;                 // Division
    }
```

Als Pendant zur *Eingabe*-Methode sorgt die Methode *Ausgabe* für die Anzeige von *A*, *B* und *Z*, wozu auch die Anpassung der Beschriftung der Eingabefelder gehört:

```
private void Anzeige()
{
    if (radioButton4.Checked)          // Anzeige in Rechteck-Koordinaten
    {
        label1.Text = "Realteil A";
        label2.Text = "Realteil B";
        label3.Text = "Realteil Z";
        label4.Text = "Imaginärteil A";
        label5.Text = "Imaginärteil B";
        label6.Text = "Imaginärteil Z";
        textBox1.Text = A.Re.ToString();          // Anzeige Realteil A
        textBox4.Text = A.Im.ToString();          // Anzeige Imaginärteil A
        textBox2.Text = B.Re.ToString();          // Anzeige Realteil B
        textBox5.Text = B.Im.ToString();          // Anzeige Imaginärteil B
        textBox3.Text = Z.Re.ToString();          // Anzeige Realteil Z
        textBox6.Text = Z.Im.ToString();          // Anzeige Imaginärteil Z
    }
    else                               // Anzeige in Polarkoordinaten
    {
        label1.Text = "Betrag A";
        label2.Text = "Betrag B";
        label3.Text = "Betrag Z";
        label4.Text = "Winkel A";
        label5.Text = "Winkel B";
        label6.Text = "Winkel Z";
        textBox1.Text = A.Len.ToString();         // Anzeige Betrag A
        textBox4.Text = A.Ang.ToString();         // Anzeige Winkel A
        textBox2.Text = B.Len.ToString();         // Anzeige Betrag B
        textBox5.Text = B.Ang.ToString();         // Anzeige Winkel B
        textBox3.Text = Z.Len.ToString();         // Anzeige Betrag Z
        textBox6.Text = Z.Ang.ToString();         // Anzeige Winkel Z
    }
}
```

Initialisierungen bei Programmstart lassen sich am saubersten durch Überschreiben der *OnLoad*-Methode der Basisklasse (*Form*) realisieren:

```
protected override void OnLoad(System.EventArgs e)
{
    Z = A + B;
    Anzeige();
    base.OnLoad(e);                  // Aufruf der Basisklassenmethode
}
```

Die "Start"-Schaltfläche:

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

```
        Eingabe();
        Verarbeitung();
        Anzeige();
    }
```

Die Anzeige wurde zwischen Rechteck- und Polarkoordinaten umgeschaltet:

```
    private void radioButton4_CheckedChanged(object sender, EventArgs e)
    {
        Anzeige();
    }
    ...
}
```

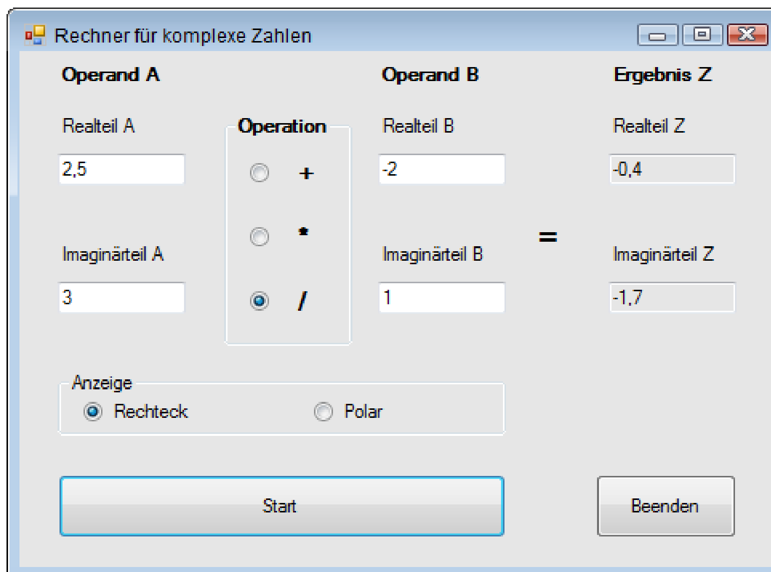
Programmtest

Wenn zum Beispiel die Aufgabe

(2.5 + 3j) / (-2 + j)

gelöst werden soll, so stellen Sie zunächst die Anzeige auf "Rechteck" ein. Geben Sie dann links oben den Realteil (2,5) und darunter den Imaginärteil (3) des ersten Operanden ein. Analog dazu geben Sie rechts oben den Realteil (-2) und darunter den Imaginärteil (1) des zweiten Operanden ein. Abschließend klicken Sie auf die gewünschte Operation (/).

Nach Betätigen der *Start*-Taste erscheint als Ergebnis die komplexe Zahl -0.4 -1.7j.



Die äquivalenten Polarkoordinaten liefern für das gleiche Beispiel einen Zeiger mit der Länge von ca. 1.746 und einem Winkel von ca. 256.76 Grad:

Rechner für komplexe Zahlen

Operand A | Operation | Operand B | Ergebnis Z
Betrag A: 3,90512483795 | Betrag B: 2,23606797749 | Betrag Z: 1,74642491965
Winkel A: 50,1944289077 | Winkel B: 153,434948822 | Winkel Z: 256,759480084
Operation: + * /
Anzeige: Rechteck Polar
Start
Beenden

HINWEIS: Wenn Sie die Anzeige zwischen Rechteck- und Polarkoordinaten umgeschaltet haben, müssen Sie anschließend die *Start*-Schaltfläche klicken!

Bemerkungen

- Man sieht einem objektorientierten Programm seine Herkunft nicht an. Die Vorteile liegen aber bekanntermaßen in der leichteren Lesbarkeit des Quellcodes ("sprechender" Code) und in der Wartbarkeit und Erweiterungsfähigkeit.
- Nicht der Endanwender, sondern der Entwickler zieht den größten Vorteil aus der OOP, da seine Arbeit übersichtlicher und transparenter wird!
- Beim Arbeiten mit Visual Studio informiert Sie die IntelliSense stets aktuell über die vorhandenen Klassenmitglieder und deren Signaturen.
- Ab .NET 4.0 wurde die Klasse *System.Numerics.Complex* eingeführt. Alle Versuche der Autoren, diese Klasse als Ersatz für *CComplexN* zu verwenden und den "Rechner für komplexe Zahlen" damit zu realisieren, scheiterten am unbefriedigenden und praxisfremden Programmiermodell der *Complex*-Klasse. So sind die Eigenschaften *Real* und *Imaginary* schreibgeschützt und nur über den Konstruktor zuweisbar, die Polarkoordinaten hingegen können nur über eine statische Methode gesetzt werden. Der Code wird dadurch unnötig aufgebläht und verliert an Transparenz. Wir haben deshalb auf die Anwendung dieser Klasse verzichtet und bevorzugen weiterhin unsere "Eigenproduktion" *CComplexN*.

HINWEIS: Wer mit der systemeigenen Klasse *Complex* dennoch experimentieren möchte, muss in der Regel vorher einen Verweis auf die *System.Numerics.dll* hinzufügen.

3.9.6 Formel-Rechner mit dem CodeDOM

Jeder, der in einer Mathematik-Ausbildung steht oder im Bereich wissenschaftlich-technischer Anwendungen arbeitet, hat sicher schon vor der Aufgabe gestanden, Berechnungen von Formelausdrücken durchzuführen, sei es um eine Werteliste zu erstellen oder um eine Diagramm auszudrucken.

Wer jetzt befürchtet, dafür erst einen aufwändigen Formelparser entwickeln zu müssen, den können wir beruhigen, denn unter .NET erlaubt das *Code Document Object Model* aus dem Namensraum *System.CodeDOM* eine verblüffend einfache Realisierungsmöglichkeit: Sie können den Quellcode einer .NET-Programmiersprache zur Laufzeit "zusammenbasteln", kompilieren und ausführen! Aus der so erzeugten Assembly kann mittels Reflexion die "zusammengebastelte" Funktion aufgerufen und das Ergebnis ausgewertet werden!

HINWEIS: Ein Rechner nach diesem Prinzip stellt bezüglich seiner Leistungsfähigkeit die bekannten Windows-Taschenrechner weit in den Schatten!

In welcher Sprache Sie die zu berechnende Formel zusammenbauen ist egal, Voraussetzung ist lediglich das Vorhandensein eines zum Compiler passenden *CodeDomProvider*s.

Im vorliegenden Fall haben wir uns nicht für C#, sondern für VB entschieden, schließlich wollen Sie als C#-Programmierer auch mal einen Blick über den Tellerrand werfen. Viel wichtiger aber ist, dass für den Endnutzer die VB-Syntax einfacher zu verstehen ist, scheitern doch viele bereits an der penibeln Groß-/Kleinschreibung von C#.

HINWEIS: Ganz abgesehen von seinem Nutzen als universeller Formelrechner bietet dieses Beispiel eine eindrucksvolle Demonstration des Prinzips und der Leistungsfähigkeit des Reflection-Mechanismus von .NET.

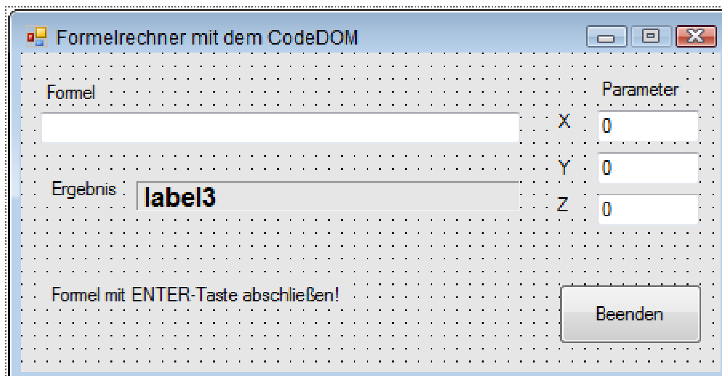
Das Grundprinzip des Formelrechners soll zunächst an einer auf das Wesentliche beschränkten Variante demonstriert werden.

Entwurf Bedienoberfläche

Öffnen Sie eine neue Windows Forms-Anwendung und erstellen Sie ein Formular entsprechend folgender Abbildung.

Bei der Gestaltung der Benutzerschnittstelle (*Form1*) haben Sie viel Spielraum, sodass obige Abbildung lediglich als Vorschlag zu verstehen ist. Wir können sogar auf eine Ergebnis-Schaltfläche verzichten, da wir die Berechnung durch einfaches Betätigen der *Enter*-Taste starten wollen.

Außerdem gönnen wir uns noch drei weitere *TextBox*en, um auch Parameter in die Formel einbauen zu können (es empfiehlt sich, dazu die *KeyPreview*-Eigenschaft des Formulars auf *True* zu setzen).



Die Klasse Calculator

Fügen Sie zum Projekt eine neue Klasse mit dem Namen *Calculator* hinzu. Die Klasse stellt einzig und allein die statische Methode *Calc()* bereit, welcher der zu berechnende Ausdruck als String zu übergeben ist. Der Rückgabewert (*double*) entspricht dem Ergebnis der Berechnung.

Die *Calc()*-Methode erzeugt den Quellcode für ein gültiges VB-Modul mit einer Klasse, die eine ganz einfache Funktion (ebenfalls mit dem Namen *Calc*) zur Berechnung dieses Ausdrucks kapselt. Der Code wird kompiliert und ausgeführt. Um den Code dem VB-Compiler zu übergeben, kommt das CodeDOM (*Code Document Object Model*) zum Einsatz, mit dem sich aus einer Anwendung heraus Programmcode erzeugen lässt. Nach dem Kompilieren wird mittels Reflection auf die erzeugte Assembly zugegriffen und der Ausdruck berechnet.

```
...
using System.CodeDom.Compiler;
using Microsoft.VisualBasic;
using System.Reflection;

namespace CodeDOM
{
    static class Calculator
    {
```

Zwischenspeichern der Assembly und der Verweise:

```
        private static Assembly ass = null;
        private static Type aClass = null;
        private static MethodInfo aMethod = null;
        private static Object obj;
```

Der Berechnungsfunktion wird der Formelausdruck als String übergeben:

```
        static public Double Calc(String expr)
        {
            if (expr.Length == 0) return 0.0;
```

Im Formelausdruck werden die Dezimalkommas durch Dezimalpunkte ersetzt:

```
expr = expr.Replace(',', '.');
```

Compilerparameter definieren:

```
CompilerParameters opt = new CompilerParameters(null, String.Empty, false);
opt.GenerateExecutable = false;
opt.GenerateInMemory = true;
```

Den zu kompilierenden VB-Quellcode müssen wir natürlich noch zeilenweise zusammenbauen, mittendrin findet sich unser zu berechnender Ausdruck. Durch die Anweisung *Imports System.-Math* können wir mathematische Funktionen wie *Sin* ... auch ohne vorangestellten Namensraum schreiben. Das Voranstellen von *Return* kann in den Code eingebaut werden ("\n" bewirkt eine neue Zeile, also eine neue VB-Anweisung):

```
 String src = "Imports System.Math\n" +
"Public Class Calculate\n" +
"Public Function Calc() As Double\n" +
"Return " + expr + "\n" +                // der zu berechnende Ausdruck!
"End Function\n" +
"End Class\n";
```

Nun kann der VB-Quellcode kompiliert werden:

```
CompilerResults res = new VBCodeProvider().CompileAssemblyFromSource(opt, src);
```

Auf die Fehlerauswertung sollte nicht verzichtet werden:

```
if (res.Errors.Count > 0)
{
    String errors = String.Empty;
    foreach (CompilerError cerr in res.Errors)
            errors = errors + cerr.ToString() + "/n";
    ass = null;
    expr = String.Empty;
    throw new ApplicationException(errors);
}
```

Die vom Compiler erzeugte Assembly kann nun ermittelt und mit dem *Reflection*-Mechanismus ausgewertet werden:

```
ass = res.CompiledAssembly;
```

Die interne Klasse aus der Assembly herausziehen:

```
aClass = ass.GetType("Calculate");
```

Die interne Methode:

```
aMethod = aClass.GetMethod("Calc");
```

Eine Instanz der internen Klasse erzeugen:

```
obj = Activator.CreateInstance(aClass);
```

Die interne Methode aufrufen und das Ergebnis zurück liefern:

```
            return Convert.ToDouble(aMethod.Invoke(obj, null));
        }
    }
}
```

Quellcode Form1

```
...
namespace CodeDOM
{
    public partial class Form1 : Form
    {
```

Die zentrale Anlaufstelle bei Änderung der Eingabewerte:

```
        private void Berechnung()
        {
```

Den Formelausdruck zuweisen:

```
            String str = textBox1.Text.ToUpper();
```

Die Parameter X, Y, Z direkt in den Formelausdruck einbauen:

```
            str = str.Replace("X", textBox2.Text).Replace("Y", textBox3.Text).
                                                  Replace("Z", textBox4.Text);
```

Start der Berechnung (eine Instanziierung der Klasse *Calculator* kann entfallen, da lediglich ein statischer Methodenaufruf erfolgt). Aufgrund der vielen möglichen Compilerfehler bei Syntaxverstößen wird der entscheidende Methodenaufruf in einer Fehlerbehandlung gekapselt:

```
            try
            {
                double res = Calculator.Calc(str);
                str = res.ToString();
```

Um das Dezimaltrennzeichen einheitlich als Punkt darzustellen, wandeln wir im Ergebnisstring das Komma einfach in einen Punkt um:

```
                label3.Text = str.Replace(",", ".");
            }
            catch (Exception ex)
            {
                label3.Text = String.Empty;
                MessageBox.Show(ex.Message);
            }
        }
```

Alle vier *TextBox*en verwenden den folgenden gemeinsamen *KeyPress*-Eventhandler, er sorgt dafür, dass die Berechnung mittels Enter-Taste gestartet wird:

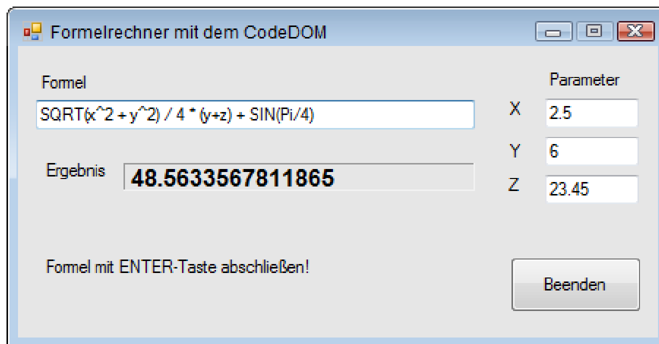
```
        private void textBox_KeyPress(object sender, KeyPressEventArgs e)
        {
```

```
            if (e.KeyChar == (Char)Keys.Enter)
            {
                Berechnung();
                e.Handled = true;
            }
        }
        ...
    }
}
```

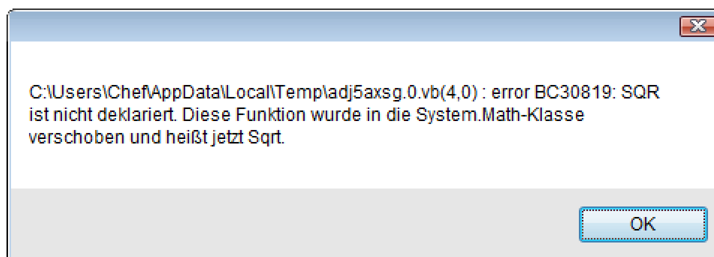
Test

Geben Sie einen beliebig komplizierten bzw. verschachtelten arithmetischen Ausdruck ein (mit oder ohne Parameter *x*, *y*, *z*). Grundlage ist zwar die VB-Syntax, aber die Unterschiede zu C# sind minimal. Die Groß-/Kleinschreibung ist ohne Bedeutung, was hier ein großer Vorteil ist.

Starten Sie die Berechnung mit der *Enter*-Taste!



Bei syntaktischen Verstößen erfolgen in der Regel recht ausführliche Fehlermeldungen. Das Beispiel in der folgenden Abbildung zeigt die Meldung, wenn versehentlich der VB-Quadratwurzel-Operator *SQRT* mit *SQR* verwechselt wurde:



Bemerkungen

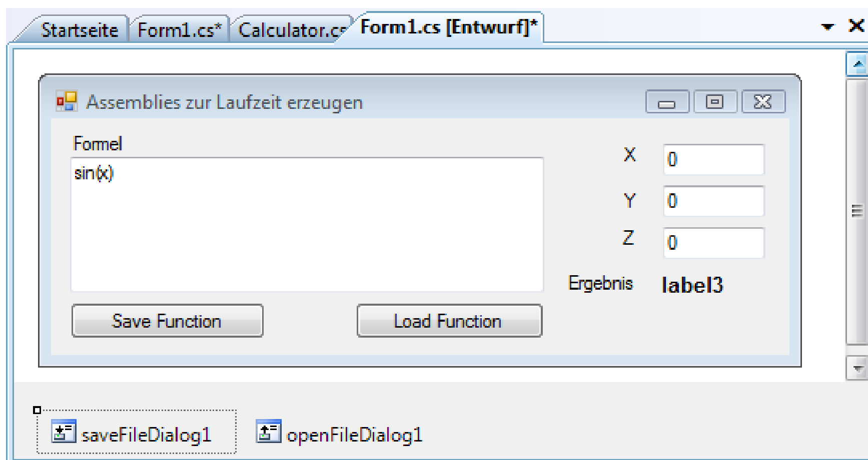
- Die Klasse *Calculator* ist ausbaufähig, denn sie kann nicht nur einen einzigen Ausdruck, sondern auch einen kompletten Algorithmus berechnen, in welchem weitere Funktionen aufgerufen werden können. In diesem Fall empfiehlt sich ein mehrzeiliges Textfeld, in das der VB-Code einzugeben ist. Vorher ist auf das Vorhandensein der *Return*-Anweisung zu prüfen, sodass diese nur im Bedarfsfall (wie in unserem Beispiel) per Programmcode hinzugefügt werden muss.
- Wollen Sie für den zu berechnenden Ausdruck nicht die VB-, sondern die C#-Syntax verwenden, muss natürlich ein anderer Quellcode "zusammengebaut" werden und Sie müssen den entsprechende C#-Codeprovider instanziieren. Bei der Eingabe der Berechnungsformel wäre dann penibel auf die Groß-/Kleinschreibung zu achten.

Assembly auf Festplatte speichern

In Erweiterung der Grundversion des Formelrechners möchten wir das Programm so modifizieren, dass eine Assembly nicht nur im Arbeitsspeicher dynamisch erstellt wird, sondern in persistenter Form auf der Festplatte abgelegt und zu einem späteren Zeitpunkt erneut geladen und genutzt werden kann.

Oberfläche

Erweitern Sie das Formular um zwei zusätzliche Schaltflächen, sowie um einen *OpenFileDialog* und einen *SaveFileDialog*:



Quelltext (Form1)

HINWEIS: Wir beschränken uns auf die Änderungen im Programm.

```
    public partial class Form1 : Form
    {
```

Die Assembly als Datei erstellen:

```
        private void button1_Click_1(object sender, EventArgs e)
        {
            if (saveFileDialog1.ShowDialog() == DialogResult.OK)
            {
```

Gleicher Ablauf wie bei der Klasse *Calculator*:

```
                string expr = textBox1.Text.Trim().ToLower();
                if (expr.Length == 0) return;
                CompilerParameters opt = new CompilerParameters(null, "", false);
                opt.GenerateExecutable = false;
```

Doch hier kommt der Unterschied:

```
                opt.GenerateInMemory = false;
                opt.OutputAssembly = saveFileDialog1.FileName;
                String src = "Imports System.Math\n" +
                             "Public Class Calculate\n" +
                             "Public Function Calc(X As Double, Y As Double, " +
                             "Z As Double) As Double\n" +
                             "Return " + expr + "\n" +        // der zu berechnende Ausdruck!
                             "End Function\n" +
```

Zusätzlich fügen wir eine zweite Funktion in die Assembly ein, über die wir das Funktionsergebnis als String zurückgeliefert bekommen:

```
                             "Public Function CalcExpression() As String\n" +
                             "Return \"" + expr + "\"\n" +
                             "End Function\n" +
                             "End Class\n";
```

Etwas Fehlerbehandlung:

```
                CompilerResults res = new VBCodeProvider().CompileAssemblyFromSource(opt, src);
                if (res.Errors.Count > 0)
                {
                       String errors = "";
                       foreach (CompilerError cerr in res.Errors)
                           errors = errors + cerr.ToString() + "\n";
                       MessageBox.Show(errors);
                }
        }
```

Ist eine Assembly erstellt, können wir diese zu jedem späteren Zeitpunkt wieder laden:

```
        private void button2_Click(object sender, EventArgs e)
        {
            if (openFileDialog1.ShowDialog() == DialogResult.OK)
            {
```

```
            double x, y, z;
            x = Convert.ToDouble(textBox2.Text);
            y = Convert.ToDouble(textBox3.Text);
            z = Convert.ToDouble(textBox4.Text);
```

Typ abrufen und Instanz erstellen:

```
            Type typ = Assembly.LoadFrom(openFileDialog1.FileName).GetType("Calculate");
            object obj = Activator.CreateInstance(typ);
```

Die beiden Methoden parametrieren und aufrufen:

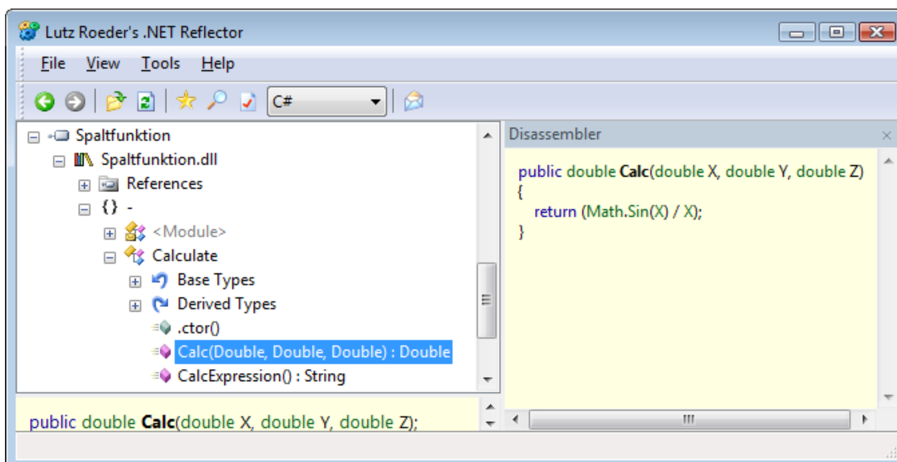
```
            MethodInfo meth1 = typ.GetMethod("Calc");
            double ret1 = (double)meth1.Invoke(obj, new object[] { x, y, z });
            MethodInfo meth2 = typ.GetMethod("CalcExpression");
            string ret2 = (string)meth2.Invoke(obj, new object[] {});
```

Anzeige der Funktionsergebnisse:

```
            MessageBox.Show("Funktion = " + ret2.ToString());
            MessageBox.Show("Ergebnis = " + ret1.ToString());
        }
    }
}
```

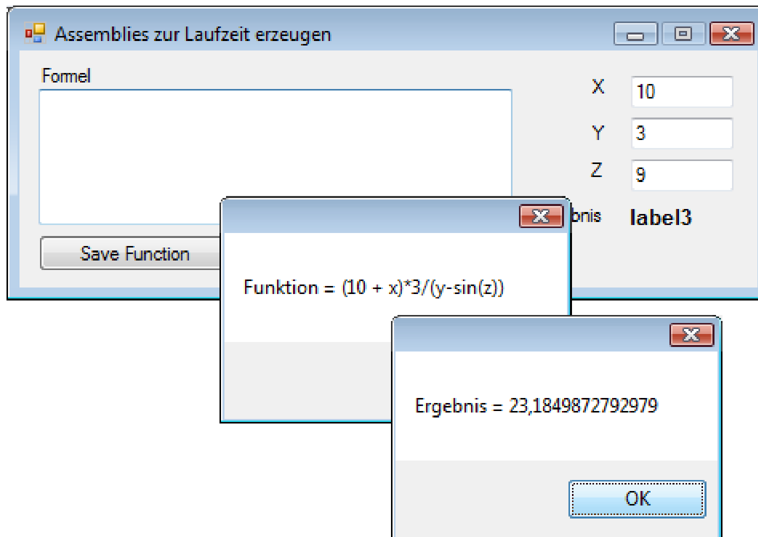
Test

Geben Sie einen beliebig komplizierten bzw. verschachtelten arithmetischen Ausdruck ein (mit oder ohne Parameter *x*, *y*, *z*) und speichern Sie nach erfolgreichem Test die Assembly ab. Mit dem .NET-Reflector (oder auch ILSpy) können Sie jetzt einen Blick in die neu erstellte Assembly werfen:



Wie nicht anders zu erwarten, finden sich hier beide Methoden, auch der zugehörige Quellcode dürfte Ihnen bekannt vorkommen.

Nach dem Laden der Assembly werden die Werte der drei Textboxen für die Berechnung herangezogen:



3.9.7 Berechnungsergebnisse als Diagramm darstellen

In der Basisversion des Formelrechners wurde gezeigt, wie man mittels CodeDOM beliebig komplizierte mathematische Formeln auswerten kann. Will man allerdings eine grafische Auswertung (Funktionsdiagramm) vornehmen, so sind in der Regel einige hundert aufeinanderfolgende Aufrufe der Formel bei schrittweise sich ändernden Parametern erforderlich. Damit stößt das bereits gezeigte Verfahren an seine Grenzen, denn das wiederholte Kompilieren und Auswerten würde auch bei einem schnellen Rechner unzumutbar viel Zeit verbrauchen.

HINWEIS: Da es nicht sinnvoll ist, bei gleichem Formelausdruck und geänderten Variablen die Assembly immer wieder erneut zu kompilieren, speichern wir die entsprechenden Verweise auf die einmal erzeugte Assembly intern (im Arbeitsspeicher) ab und greifen bei Bedarf darauf zu.

Wir wollen die Vorgehensweise anhand einer bescheidenen Grafik für die bekannte Spaltfunktion *sin(x)/x* erläutern.

Oberfläche

Das nackte Startformular (*Form1*), ausgestattet mit einer einzigen *TextBox*, genügt! Weisen Sie der *TextBox* die *Text*-Eigenschaft "*sin(x)/x*" zu (Leerzeichen und Groß-/Kleinschreibung spielen keine Rolle).

Die Klasse CalculatorX

Die Änderungen gegenüber der Klasse *Calculator* sind durch Fettdruck hervorgehoben:

```
...
using System.CodeDom.Compiler;
using Microsoft.VisualBasic;
using System.Reflection;
...
    static class CalculatorX
    {
        private static Assembly ass = null;
        private static Type aClass = null;
        private static MethodInfo aMethod = null;
        private static String expression = String.Empty;
        private static Object obj = null;

        static public Double Calc(String expr, double x)
        {
            if (expr.Length == 0) return 0.0;

            if (expr != expression)             // expr hat sich geändert
            {
                expression = expr;
                CompilerParameters opt = new CompilerParameters(null, String.Empty, false);
                opt.GenerateExecutable = false;
                opt.GenerateInMemory = true;

                String src = "Imports System.Math\n" +
                             "Public Class Calculate\n" +
                             "Public Function Calc(X As Double) As Double\n" +
                             "Return " + expr + "\n" +
                             "End Function\n" +
                             "End Class\n";

                CompilerResults res = new VBCodeProvider().CompileAssemblyFromSource(opt, src);
                if (res.Errors.Count > 0)
                {
                    String errors = String.Empty;
                    foreach (CompilerError cerr in res.Errors)
                        errors = errors + cerr.ToString() + "\n";
                    ass = null;
                    expression = String.Empty;
                    throw new ApplicationException(errors);
                }
                ass = res.CompiledAssembly;
                aClass = ass.GetType("Calculate");
                aMethod = aClass.GetMethod("Calc");
                obj = Activator.CreateInstance(aClass);
```

```
            }
            return Convert.ToDouble(aMethod.Invoke(obj, new Object[] {x}));
        }
    }
}
```

Quellcode Form1

```
public partial class Form1 : Form
{    ....
```

Wir überschreiben die *OnPaint*-Methode des Formulars, dadurch wird auch nach vorübergehendem Verdecken des Fensters das Diagramm immer wieder neu erstellt:

```
    protected override void OnPaint(PaintEventArgs e)
    {
```

Um den Quellcode überschaubar zu halten und nicht vom Wesentlichen abzulenken, haben wir hier die Anfangseinstellungen für das Diagramm der Funktion *y = sin(x)/x* optimiert:

```
        Single x = -40;
        Single y = 0;
        Single xold = x;
        Single yold = y;
        Single scalex = this.Width/(2*x);
        Single scaley = -(this.Height-20) / 2;
        Single incr = 80f / this.Width;
        Pen p = new Pen(Color.Black, 2);
```

Nicht zu vergessen ist das Zuweisen eines *Graphics*-Objekts, auf welches das Diagramm gezeichnet werden kann:

```
        Graphics g = e.Graphics;
```

Koordinatensystem verschieben:

```
        g.TranslateTransform(this.Width / 2, this.Height / 2);
```

Koordinatenachsen zeichnen:

```
        g.DrawLine(Pens.Red,-this.Width, 0, this.Width,0);
        g.DrawLine(Pens.Red,0, -this.Height, 0, this.Height);
```

Anzeige optimieren:

```
        g.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.AntiAlias;
```

Funktion berechnen und ausgeben:

```
        for (int i = 0; i < this.Width; i++)
        {
            if (x == 0) x += 0.000001f;
            y = Convert.ToSingle(CalculatorX.Calc(textBox1.Text, x));
            x += incr;
            g.DrawLine(p, x * scalex, y * scaley, xold * scalex, yold * scaley);
```

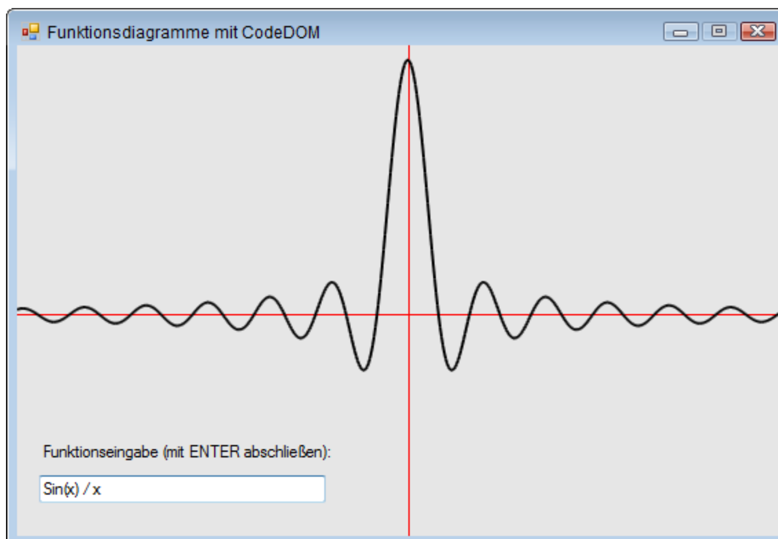
```
            xold = x;
            yold = y;
        }
        base.OnPaint(e);
    }
```

Der folgende Eventhandler sorgt für die Umwandlung eines Dezimalkommas in einen Dezimalpunkt und für das Neuzeichnen des Formulars nach Betätigen der Enter-Taste:

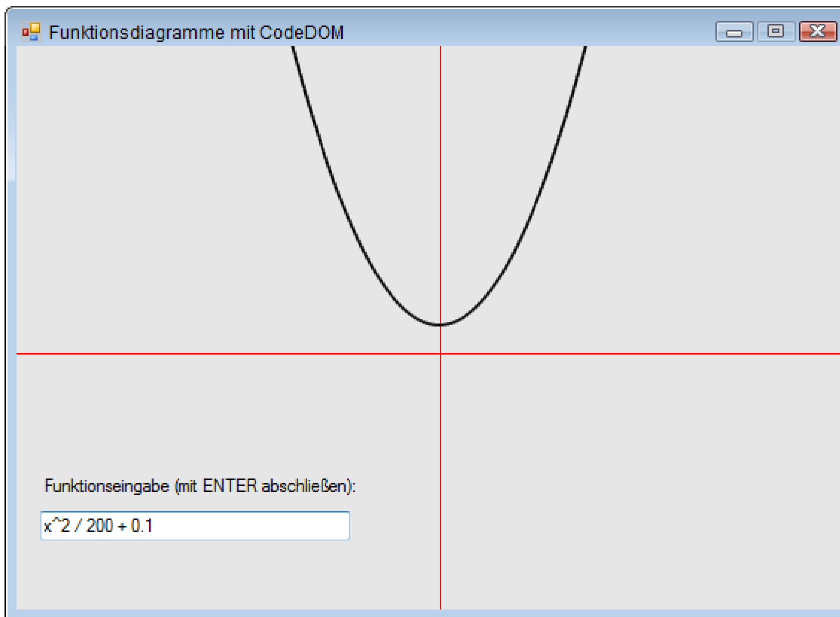
```
    private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
    {
        if (e.KeyChar == ',') e.KeyChar = '.';
        else if (e.KeyChar == (Char)Keys.Enter)
        {
            this.Refresh();
            e.Handled = true;
        }
     }
}
```

Test

Gleich nach Programmstart erscheint das Diagramm der Funktion *sin(x)/x*. Die Grafik wird auch nach vorübergehendem Verdecken des Fensters in Windeseile wieder aufgebaut, da nur einmal kompiliert werden muss.



Ändern Sie den Formelausdruck und schließen Sie mit der Enter-Taste ab. Unter der Voraussetzung, dass das starre Koordinatensystem eine geeignete Darstellung zulässt, lassen sich interessante Experimente durchführen:



Bemerkungen

- Das Programm lässt sich durch Hinzufügen weiterer Bedienelemente für Inkremente, Maßstabsfaktoren, Skalenteilung etc. so verfeinern, dass eine optimale Darstellung nahezu beliebiger Funktionen ermöglicht wird.
- Die spezialisierte Klasse *CalculatorX* hat gegenüber der Klasse *Calculator* den Vorteil, dass sie wiederholte Berechnungen mit einem geänderten Parameter *x* gestattet, ohne dass dazu die Assembly erneut erzeugt werden müsste. Das bedeutet einen erheblichen Performancegewinn.
- Der Klasse *Calculator* bleibt hingegen der Vorteil der universellen Verwendbarkeit für beliebige Ausdrücke mit beliebig vielen Parametern, sie eignet sich deshalb besonders für einmalig auszuführende komplizierte Berechnungen.

3.9.8 Sortieren mit IComparable/IComparer

Haben Sie eine *ArrayList* oder eine generische *List*<> von Typen, wie Strings oder Integers, die bereits *IComparer* unterstützen, so können Sie dieses Array oder die Liste ohne irgendeine explizite Referenz auf *IComparer* sortieren. In diesem Fall werden die Elemente des Arrays automatisch in die standardmäßige Implementierung von *IComparer* gecastet, eine *Sort*-Methode ist also bereits "eingebaut".

Haben Sie aber nutzerdefinierte Objekte, so müssen Sie selbst entweder eines oder beide der Interfaces *IComparable* oder *IComparer* implementieren.

Oberfläche

Öffnen Sie eine neue WPF-Anwendung und platzieren Sie eine *ListBox* (*Name=lb*) und zwei Schaltflächen im Formular:

```
<Grid Margin="5">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <ListBox Name="lb"  Grid.Column="0" ItemsSource="{Binding}"/>
    <StackPanel Grid.Column="1">
        <Button Content="IComparable" Click="Button_Click_1" />
        <Button Content="IComparer"  Click="Button_Click_2" />
    </StackPanel>
</Grid>
```

Klasse CStudent

Fügen Sie zum Projekt eine neue Klasse *CStudent* hinzu, die das *IComparable*-Interface implementiert:

```
public class CStudent : IComparable<CStudent>
{
    public int Nummer{get; set;}
    public string Vorname { get; set; }
    public string Nachname { get; set; }

    public CStudent(int numm, string vor, string nach)
    {
        Nummer = numm;
        Vorname = vor;
        Nachname = nach;
    }
```

Eine sinnvolle *ToString*-Methode für die Datenbindung:

```
    public override string ToString()
    {
        return Nummer.ToString() + " ,  " + Nachname + " , " + Vorname;
    }
```

Die alphabetische Sortierung nach dem Nachnamen wird als Standardvergleich festgelegt:

```
    public int CompareTo(CStudent stud)
    {
        return this.Nachname.CompareTo(stud.Nachname);
    }
}
```

MainWindow.xaml.cs

Im Hauptfenster implementieren wir zunächst eine Methode, die eine Studentenliste erzeugt und initialisiert:

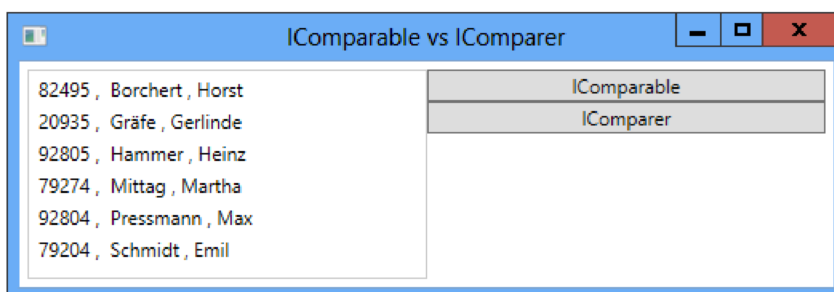
```
        private List<CStudent> getStudenten()
        {
            List<CStudent> studenten = new List<CStudent>();

            studenten.Add(new CStudent(82495, "Horst", "Borchert"));
            studenten.Add(new CStudent(20935, "Gerlinde", "Gräfe"));
            ...
            return studenten;
        }
```

Im *Click*-Event des ersten *Button*s wird die Liste (entsprechend der Implementierung von *CompareTo* in *CStudent*) sortiert und angezeigt:

```
  private void Button_Click_1(object sender, RoutedEventArgs e)
        {
            List<CStudent> studenten = getStudenten();
            studenten.Sort();
            lb.DataContext = studenten;
        }
...
```

Test von IComparable

Bei Klick auf die Schaltfläche *IComparable* erscheint die Liste alphabetisch nach den Nachnamen sortiert:



Bemerkungen zu IComparable

- Wenn eine Klasse das *IComparable* Interface implementiert, müssen wir auch die Methode *CompareTo(T)* implementieren. In dieser Methode können wir unseren Sortieralgorithmus definieren (Standardvergleich). In unserem Beispiel haben wir die Liste in alphabetischer Reihenfolge sortiert.

- Wir verwenden *IComparable<T>*, wenn die Klasse über einen Standardvergleich verfügen soll. Das Sortierkriterium muss also bereits bekannt sein, bevor wir mit der Implementierung der Klasse beginnen. In unserem Beispiel mussten wir deshalb vorher entscheiden, dass wir nach dem Nachnamen und nicht nach der Studentennummer sortieren wollen. Es gibt aber Situationen, wo wir nicht nur den Standardvergleich, sondern mehrere Sortierkriterien benötigen.
- Um letztgenanntes Problem zu lösen, stellt .NET ein spezielles Interface *IComparer*<> bereit, welches über eine Methode *Compare()* verfügt, die zwei Objektparameter X, Y entgegennimmt und ein Integer zurückgibt.

Klasse CSortNummer

Wir wollen nach der Nummer des Studenten wahlweise in auf- oder absteigender Folge sortieren lassen. Um dafür nicht zwei Klassen schreiben zu müssen, übergeben wir im Konstruktor einen Parameter, der die gewünschte Sortierfolge spezifiziert.

```
public class CSortNummer : IComparer<CStudent>
{
```

Sortierung in aufsteigender (*true*) bzw. absteigender (*false*) Ordnung:

```
    private bool _auf;
```

Festlegung der Sortierfolge im Konstruktor:

```
    public CSortNummer(bool auf)
    {
        _auf = auf;
    }
```

Die Implementierung des Sortierkriteriums muss in der *Compare*-Methode erfolgen:

```
    public int Compare(CStudent a, CStudent b)
    {
        if (_auf)     // aufsteigend
        {
            if (a.Nummer > b.Nummer) return 1;
            else if (a.Nummer < b.Nummer) return -1;
            else return 0;
        }
        else          // absteigend
        {
            if (a.Nummer < b.Nummer) return 1;
            else if (a.Nummer > b.Nummer) return -1;
            else return 0;
        }
    }
}
```

Ergänzung MainWindow.xaml.cs

Wir ergänzen den Code, um die zweite Sortiervariante zu testen:

```
        private void Button_Click_2(object sender, RoutedEventArgs e)
        {
            List<CStudent> studenten = getStudenten();
```

Wir wollen die Sortierfolge umkehren:

```
            CSortNummer son = new CSortNummer(false);
```

Die Liste wird nun nach fallenden Nummern sortiert:

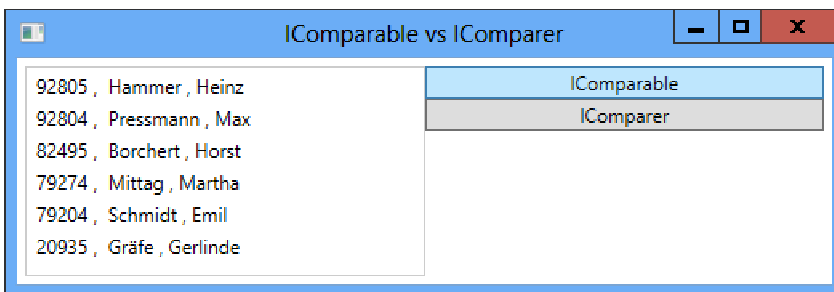
```
            studenten.Sort(son);
```

Noch eine Bemerkung zu obiger Zeile: Die IntelliSense wird Ihnen zwei Überladungen der *Sort*-Methode anbieten, eine davon ist die bereits implementierte Standardsortierung (*IComparable*), die andere die in *CSortNummer* definierte zusätzliche Sortierung, die wir hier ausgewählt haben.

```
            lb.DataContext = studenten;
        }
```

Test von IComparer

Die Liste ist jetzt nach fallenden Nummern sortiert:



Bemerkungen zu IComparer

- Wir verwenden *IComparer* dann, wenn wir ein anderes Sortierkriterium als den von der Klasse bereitgestellten Standardvergleich benötigen (oder mehrere Vergleichskriterien).
- Pro zusätzlichem Sortierkriterium ist eine extra Klasse erforderlich, die das *Icomparer*-Interface (*Compare*-Methode) implementiert.
- *IComparer* funktioniert auch dann, wenn keine Standardsortierung existiert, also wenn *IComparable* nicht implementiert ist.

3.9.9 Einen Objektbaum in generischen Listen abspeichern

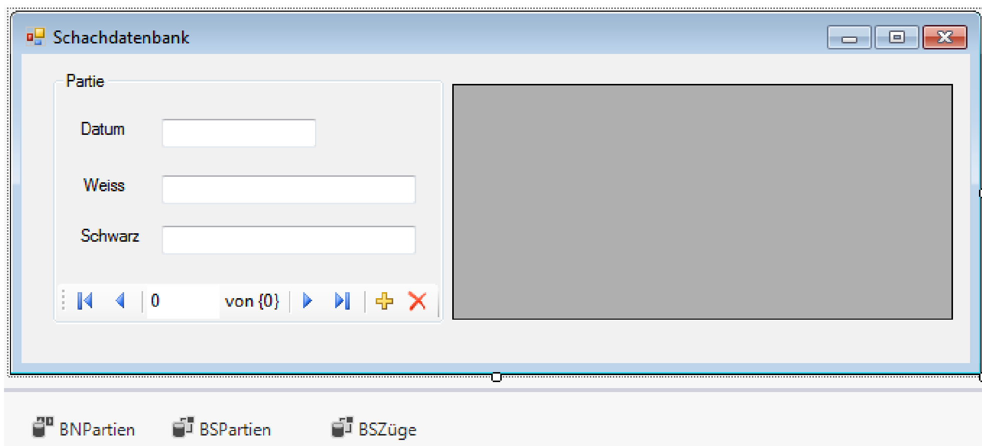
Die generische *List*-Klasse hat gegenüber der altbackenen *ArrayList* gravierende Vorteile, so entfallen vor allem die umständlichen Typkonvertierungen. Nicht nur einfache Objekte, sondern auch komplette Objektbäume können mittels *BinaryFormatter* bequem serialisiert bzw. deserialisiert werden, um die Daten dauerhaft zu sichern.

Wir demonstrieren dies am Beispiel einer ausbaufähigen Schachdatenbank, in der Sie beliebig viele Schachpartien abspeichern können[1].

HINWEIS: Das vorliegende Beispiel benutzt keine Datenbank, sondern eine einfache Binärdatei als Speichermedium!

Oberfläche

Starten Sie Visual Studio und öffnen Sie eine neue Windows Forms-Anwendung. Auf dem Startformular *Form1* findet links eine *GroupBox* ihren Platz. In diese setzen Sie drei *TextBox*-Steuerelemente und einen *BindingNavigator*, der am unteren Rand angedockt wird (*Dock* = *Bottom*) und den Sie in *BNPartien* umbenennen.



Rechts platzieren Sie eine *DataGridView*-Komponente. Fügen Sie noch zwei *BindingSource*-Komponenten hinzu, die Sie in *BSPartien* und *BSZüge* umbenennen. Verbinden Sie nun die *BindingSource*-Eigenschaft von *BNPartien* mit *BSPartien*.

Über das Menü *Projekt/Klasse hinzufügen...* erweitern Sie das Projekt um die Klassen *CZug*, *CPartie* und *CSchachDB*, die beide am Objektbaum beteiligt sind. Alle werden mit dem *Serializable*- Attribut ausgestattet, da wir den Objektbaum nicht nur im Arbeitsspeicher ablegen, sondern auch auf der Festplatte sichern wollen.

[1] Um den Sinn dieses Praxisbeispiels zu verstehen, muss man kein Schachspieler sein.

Die Klasse CZug

Diese Klasse repräsentiert einen Doppelzug nebst Kommentar (einfachheitshalber sind alle Eigenschaften selbst implementierend):

```
[Serializable] public class CZug
{
     public string Weiss {get; set;}         // Zug des Weiss-Spielers, z.B. "d2-d4"
     public string Schwarz { get; set;}      // Antwortzug des Schwarz-Spielers, z.B. "d7-d5"
     public string Kommentar { get; set;}    // z.B. "geschlossene Eröffnung"
}
```

Die Klasse CPartie

```
[Serializable] public class CPartie
{
```

Eine generische Liste speichert alle Züge einer Partie:

```
    private IList<CZug> _züge;
```

Im Konstruktor wird die (zunächst leere) Zugliste erstellt:

```
    public CPartie()
    {
        _züge = new List<CZug>();
    }
```

Der Zugriff auf die Zugliste:

```
   public IList<CZug> Züge
   {
        get { return _züge; }
        set { _züge = value; }
   }
```

Einige selbst implementierende Eigenschaften:

```
   public DateTime Datum { get; set; }      // Datum der Partie
   public string Weiss {get; set;}          // Name des Weiss-Spielers
   public string Schwarz {get; set;}        // Name des Schwarz-Spielers
}
```

Die Klasse CSchachDB

Dies ist die Wurzelklasse unseres Objektbaums.

```
[Serializable] public class CSchachDB
{
```

Die folgende (generische) Liste speichert alle Partien der Datenbank:

```
    private IList<CPartie> _partien;
```

Im Konstruktor wird die (zunächst leere) Partienliste erzeugt:

```
    public CSchachDB()
    {
        _partien = new List<CPartie>();
    }
```

Der Zugriff auf die Liste der Partien:

```
    public IList<CPartie> Partien
    {
        get { return _partien; }
        set { _partien = value; }
    }
}
```

Die Klasse CPersistenz

Damit nach dem Ausschalten des Rechners nicht alle Daten auf Nimmerwiedersehen verschwunden sind, stellt diese Klasse die Methoden *saveObject* und *loadObject* bereit, die sich um die Datenpersistenz kümmern.

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
...
 public class CPersistenz
 {
```

Aufgabe dieser Methode ist es, den kompletten Objektbaum zu serialisieren und als Datei abzuspeichern:

```
    public static void saveObject(object o, string pfad)
    {
       FileStream fs = new FileStream(pfad, FileMode.Create, FileAccess.Write, FileShare.None);
       BinaryFormatter bf = new BinaryFormatter();
       bf.Serialize(fs, o);
       fs.Close();
    }
```

Die folgende Methode lädt die Datei zurück in den Arbeitsspeicher und rekonstruiert den Objektbaum:

```
    public static object loadObject(string pfad)
    {
       FileStream fs = new FileStream(pfad, FileMode.Open, FileAccess.Read, FileShare.Read);
       BinaryFormatter bf = new BinaryFormatter();
       object o = bf.Deserialize(fs);
       fs.Close();
       return o;
    }
 }
```

Die Klasse Form1

```
public partial class Form1 : Form
{
```

Der komplette Objektbaum:

```
    private CSchachDB _schachDB = new CSchachDB();
```

Der Standort der Datenbankdatei:

```
    private const string PFAD = "SchachDB.dat";
```

Der Programmstart:

```
    private void Form1_Load(object sender, EventArgs e)
    {
```

Zunächst wird versucht, die Datenbankdatei zu laden, im Fehlerfall entsteht eine neue leere Datei:

```
        try
        {
            _schachDB = (CSchachDB) CPersistenz.loadObject(PFAD);
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
```

Verbinden der *Partien*-Liste mit der entsprechenden *BindingSource*-Komponente:

```
        BSPartien.DataSource = _schachDB.Partien;
```

Anbinden der Steuerelemente:

```
        textBox1.DataBindings.Add("Text", BSPartien, "Datum", true);
        textBox2.DataBindings.Add("Text", BSPartien, "Weiss", true);
        textBox3.DataBindings.Add("Text", BSPartien, "Schwarz", true);
```

Anbinden des Datengitters an die Liste der Züge:

```
       dDataGridView1.DataSource = BSZüge;
    }
```

Erzeugen Sie einen Eventhandler für das *CurrentChanged*-Ereignis der *BindingSource* der Partienliste:

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

Beim Navigieren zu einer anderen Partie wird die *BindingSource* der Zügeliste umgeklemmt:

```
        CPartie partie  = (CPartie) BSPartien.Current;
        BSZüge.DataSource = partie.Züge;
    }
```

Beim Schließen des Formulars versuchen wir, den Objektbaum in der Datei abzuspeichern:

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
        try
        {
           CPersistenz.saveObject(_schachDB, PFAD);
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
}
```

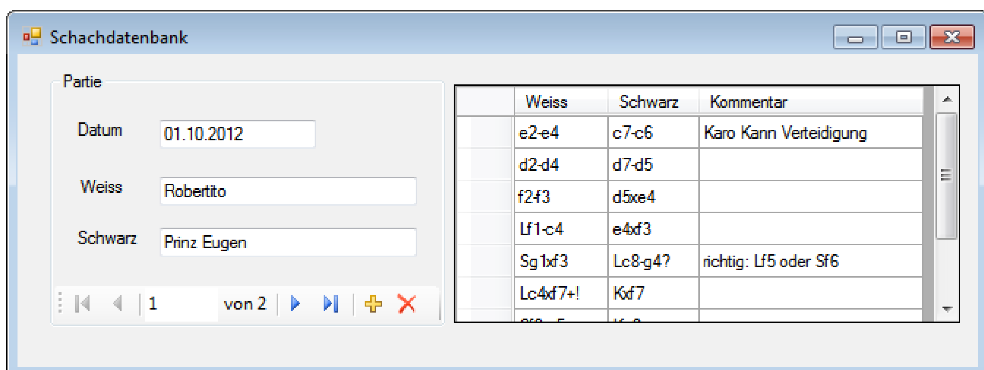
Damit Sie nicht leichtfertig eine komplette Schachpartie mitsamt allen mühsam eingegebenen Zügen löschen, soll nach Klick auf die *Delete*-Schaltfläche des *BindingNavigator*s zunächst eine Sicherheitswarnung erscheinen:

```
private void BindingNavigatorDeleteItem2_Click(object sender, EventArgs e)
{
        if (MessageBox.Show("Soll die Partie wirklich gelöscht werden?",
                     "Sicherheitsabfrage", MessageBoxButtons.YesNo) == DialogResult.Yes)
        BSPartien.RemoveCurrent();
}
}
```

HINWEIS: Obige Methode funktioniert nur dann, wenn Sie die *DeleteItem*-Eigenschaft des *BindingNavigator*s auf *None* (keine) gesetzt haben.

Test

Lassen Sie sich nicht davon irritieren, dass nach Programmstart zunächst ein Meldungsfenster erscheint, welches Sie auf die noch nicht vorhandene Datei hinweist.



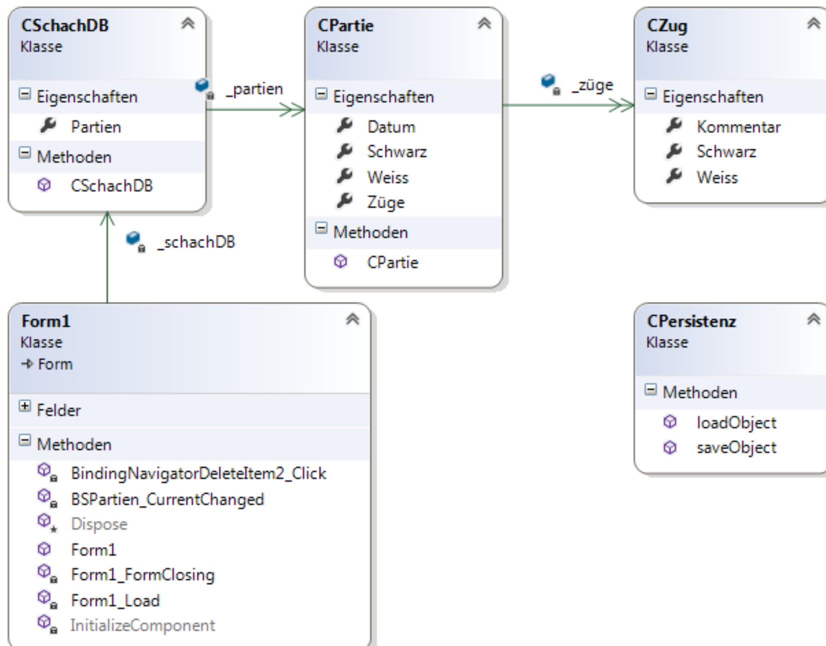
Klicken Sie zunächst auf die "+"-Schaltfläche im *BindingNavigator*, um eine neue Schachpartie mit dem Datum[1] und den Namen der Spieler anzulegen. Danach können Sie auf bekannte Weise im Datengitter nacheinander die einzelnen Züge eintippen.

Nach Schließen des Formulars werden alle Partien automatisch in der im Anwendungsverzeichnis abgelegten Datei *Schachpartien.dat* gespeichert und stehen nach erneutem Programmstart wieder zur Verfügung.

Bemerkungen

- Für den Schachinteressenten ergeben sich mannigfaltige Erweiterungsmöglichkeiten, wie zum Beispiel die Suche nach Partien mit einem bestimmten Gegner oder nach gleichen Eröffnungszügen. Ziemlich aufwändig, aber auch sehr praktisch ist eine grafische Brettdarstellung, mittels welcher man die Züge per Drag&Drop eingeben kann.

- Einen guten Überblick über die Programmstruktur und die Beziehungen zwischen den Klassen liefert das **Klassendiagramm**, welches man sich im Projektmappen-Explorer generieren lassen kann (Kontextmenü *Klassendiagramm anzeigen*). Die Auflistungszuordnungen (Assoziationen) der Felder *_partien* und *_züge* sind an den doppelten Pfeilspitzen erkennbar. Man erhält sie, wenn man zunächst auf das entsprechende Feld in der Klasse klickt und dann im Kontextmenü *Als Auflistungszuordnung anzeigen* wählt. Die folgende Abbildung zeigt nur eine von vielen möglichen Versionen des Klassendiagramms:



[1] Haben Sie ein ungültiges Datum eingegeben, so lässt sich das Formular erst dann wieder schließen, wenn Sie diesen Fehler korrigiert haben (generische Listen sind typsicher!).

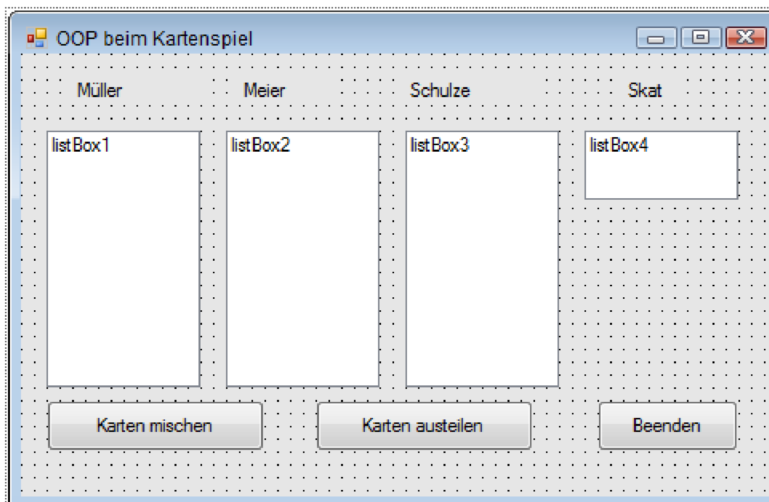
3.9.10 OOP beim Kartenspiel erlernen

Am Beispiel eines Skatspiels soll das vorliegende Praxisbeispiel Ihnen helfen, Ihre OOP-Grundlagen weiter auszubauen. Es wird gezeigt, wie die in einem Array abgespeicherten *Karten*-Objekte durch einen Konstruktor erzeugt werden und selbst wiederum nach außen als Eigenschaften einer Klasse *Spiel* in Erscheinung treten.

HINWEIS: Zum Verständnis ist die Kenntnis der Skatregeln keinesfalls Voraussetzung. Es reicht aus zu wissen, dass das Spiel aus 32 Karten besteht und jeder der drei Spieler zu Beginn 10 Karten erhält und die restlichen zwei davon im so genannten "Skat" verbleiben.

Oberfläche

Die folgende Abbildung bedarf wohl keines weiteren Kommentars:



Quellcode

Es dient der Übersicht, wenn man für jede Klasse ein eigenes Klassenmodul verwendet. Diesmal aber wollen wir zeigen, dass man auch mehrere Klassen gemeinsam in einem Modul implementieren kann und erweitern dazu das bereits vorhandene *Form1*-Modul. Unterhalb des Klassencodes von *Form1* deklarieren wir die Klassen *CKarte* und *CSpiel*:

```
public class CKarte
{
    public string Farbe;
    public string Wert;
```

Der Konstruktor:

```
    public CKarte(string f, string w)
    {
        Farbe = f;
        Wert = w;
    }
}
```

Die Klasse *CSpiel* kapselt 32 Instanzen der Klasse *CKarte*:

```
public class CSpiel
{
```

Die Eigenschaft *Karten* (das ist ein Array mit 32 Karten!):

```
    public CKarte[] Karten = new CKarte[32];
```

Eine Hilfsmethode soll das Generieren der Karten vereinfachen:

```
    private void createKarten(string farbe, int a)
    {
        Karten[a] = new CKarte(farbe, "Sieben");
        Karten[a + 1] = new CKarte(farbe, "Acht");
        Karten[a + 2] = new CKarte(farbe, "Neun");
        Karten[a + 3] = new CKarte(farbe, "Zehn");
        Karten[a + 4] = new CKarte(farbe, "Bube");
        Karten[a + 5] = new CKarte(farbe, "Dame");
        Karten[a + 6] = new CKarte(farbe, "König");
        Karten[a + 7] = new CKarte(farbe, "As");
    }
```

Der Konstruktor erzeugt und füllt das Spiel der Reihe nach mit allen 32 Karten:

```
    public CSpiel()
    {
         createKarten("Eichel", 0);
         createKarten("Grün", 8);
         createKarten("Rot", 16);
         createKarten("Schell", 24);
    }
```

Die Methode zum Mischen der Karten:

```
    public void mischen()
    {
```

Der Zwischenspeicher für den Kartentausch:

```
        CKarte tmp;
```

Ein Zufallszahlengenerator:

```
        Random rnd = new Random();
```

Alle Karten nacheinander durchlaufen:

```
        for (int i = 0; i < Karten.Length; i++)
        {
```

Den Zufallsindex einer anderen Karte bestimmen:

```
            int z = rnd.Next(0, Karten.Length);
```

Die zufällige Karte mit der aktuellen Karte vertauschen:

```
            tmp = Karten[z];
            Karten[z] = Karten[i];
            Karten[i] = tmp;
        }
    }
}
```

HINWEIS: Aus Gründen der Einfachheit bleibt der Index 0 des *Karten*-Arrays ungenutzt.

Nun zum Klassencode von *Form1*:

```
public partial class Form1 : Form
{
    ...
```

Ein neues Kartenspiel wird erzeugt:

```
    CSpiel spiel = new CSpiel();
```

Eine Hilfsroutine zum Löschen der Anzeige:

```
    private void loeschen()
    {
        listBox1.Items.Clear();
        listBox2.Items.Clear();
        listBox3.Items.Clear();
        listBox4.Items.Clear();
    }
```

Alle Karten mischen:

```
    private void button1_Click(object sender, System.EventArgs e)
    {
        loeschen();
        spiel.mischen();
    }
```

Alle Karten austeilen:

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

Die Karten für Müller:

```
        for (int i = 0; i < 10; i++)
            listBox1.Items.Add(spiel.Karten[i].Farbe + "  " + spiel.Karten[i].Wert);
```

Die Karten für Meier:

```
        for (int i = 10; i < 20; i++)
            listBox2.Items.Add(spiel.Karten[i].Farbe + "  " + spiel.Karten[i].Wert);
```

Die Karten für Schulze:

```
        for (int i = 20; i < 30; i++)
            listBox3.Items.Add(spiel.Karten[i].Farbe + "  " + spiel.Karten[i].Wert);
```

Die restlichen zwei Karten wandern in den Skat:

```
        for (int i = 30; i < 32; i++)
            listBox4.Items.Add(spiel.Karten[i].Farbe + "  " + spiel.Karten[i].Wert);
    }
    ...
}
```

Test

Wenn Sie, vor lauter Ungeduld, unmittelbar nach Programmstart auf die Schaltfläche "Karten austeilen" klicken, werden Sie von allen drei Spielern laute Protestrufe ernten, da die Karten offensichtlich nicht gemischt wurden:

Erst nach ein- oder mehrmaligem Klick auf die Schaltfläche "Karten mischen" hat die Gerechtigkeit ihren Einzug gehalten und der Zufall bestimmt, welche Karten die Spieler Müller, Meier und Schulze erhalten:

3.9.11 Eine Klasse zur Matrizenrechnung entwickeln

Am Beispiel einer Klasse *CMatrix* soll die grundlegende Vorgehensweise bei der Entwicklung einer Klasse erläutert werden, die schon etwas anspruchsvoller ist als z.B. eine triviale *CPerson*-Klasse.

Schwerpunktthemen sind:

- überladener Konstruktor
- überladene Methoden
- Eigenschaftsmethoden
- Indexer
- Unterschied zwischen statischen (Shared-) Methoden und Instanzen-Methoden

Die Klasse *CMatrix* soll Funktionalität zur Verfügung stellen, die Sie zur Ausführung von Matrixoperationen benötigen (Addition, Multiplikation ...).

Obwohl wir hier nur die Addition implementieren werden, kann die Klasse von Ihnen nach dem gezeigten Muster selbständig um weitere Matrizenoperationen erweitert werden, wie z.B. Multiplikation oder Inversion.

HINWEIS: Wer sich nicht für Mathematik interessiert, kann das Beispiel trotzdem sehr gut verwenden, da der Schwerpunkt auf den verwendeten Programmiertechniken liegt!

Quellcode der Klasse CMatrix

Wir beginnen diesmal nicht mit dem Startformular (*Form1*), sondern erweitern zunächst über den Menüpunkt *Projekt|Klasse hinzufügen...* unser Projekt um eine neue Klasse mit dem Namen *CMatrix*.

Die Klasse *CMatrix* verwaltet ein zweidimensionales Array aus *double*-Zahlen. Die Zustandsvariablen *_rows* und *_cols* speichern die Anzahl der Zeilen und Spalten.

```
public class CMatrix
{
    private int _rows, _cols;
    private double[,] _array;
```

Ein neues Array wird über den Konstruktor instanziiert, der in zwei Versionen vorliegt. Falls Sie später *new()* ohne Argument aufrufen, wird eine Matrix mit einem einzigen Element generiert, ansonsten mit den gewünschten Dimensionen.

```
    public CMatrix()                    // Standardkonstruktor
    {
        _rows = 1;  _cols = 1;
        _array = new double[_rows, _cols];
    }

    public CMatrix(int r, int c)        // überladener Konstruktor
    {
        _rows = r; _cols = c;
        _array = new double[_rows, _cols];
    }
```

Der Zugriff auf die (privaten) Zustandsvariablen *_rows* und *_cols* wird über die Eigenschaften *Rows* und *Cols* gekapselt.

```
    public int Rows                    // Eigenschaft zur Ermittlung der Zeilenanzahl
    {
        get {return _rows;}
        set {_rows = value;}
    }

    public int Cols                    // Eigenschaft zur Ermittlung der Spaltenanzahl
    {
        get {return _cols;}
        set {_cols = value;}
    }
```

Der Zugriff auf ein bestimmtes Matrix-Element wird natürlich elegant über einen Indexer realisiert:

```
    public double this [int row, int col]
    {
        get {return _array[row, col];}
        set {_array[row, col] = value;}
    }
```

Die *Add*-Methode akzeptiert entweder ein oder zwei *CMatrix*-Objekte als Parameter, falls Sie nur ein *CMatrix*-Objekt übergeben, wird die aktuelle Instanz der Matrix als zweiter Operand verwendet.

Die erste Überladung der *Add*-Methode ist statisch, sie wird also nicht über einem *CMatrix*-Objekt, sondern direkt über der *CMatrix*-Klasse ausgeführt! Die Methode nimmt beide Operanden (*CMatrix*-Objekte) entgegen und liefert ein *CMatrix*-Objekt zurück.

```
public static CMatrix Add(CMatrix A, CMatrix B)
{
    if (!((A.Rows == B.Rows) && (A.Cols == B.Cols)))
        return new CMatrix();
    else
    {
        CMatrix newMatrix = new CMatrix(A.Rows, A.Cols);
        for (int row = 0; row < A.Rows; row++)
        {
            for (int col = 0; col < A.Cols; col++)
                newMatrix[row, col] = A[row, col] + B[row, col];
        }
        return newMatrix;
    }
}
```

Obige Methode wird mit einer leeren "Verlegenheitsmatrix" verlassen, wenn beide Operanden nicht die gleichen Dimensionen aufweisen sollten.

Bei der zweiten Überladung handelt es sich um eine normale Instanzen-Methode, sie nimmt als Parameter nur ein einziges *CMatrix*-Objekt entgegen. Der zweite Operand ist naturgemäß die aktuelle *CMatrix*-Instanz, die diese Methode aufruft.

```
public CMatrix Add(CMatrix A)
{
    if (!((A.Rows == this.Rows) && (A.Cols == this.Cols)))
         return new CMatrix();
    else
    {
        CMatrix newMatrix = new CMatrix(this.Rows, this.Cols);
        for (int row = 0; row < this.Rows; row++)
            for (int col = 0; col < this.Cols; col++)
                 newMatrix[row, col] = A[row, col] + this[row, col];
            return newMatrix;
        }
    }
}
```

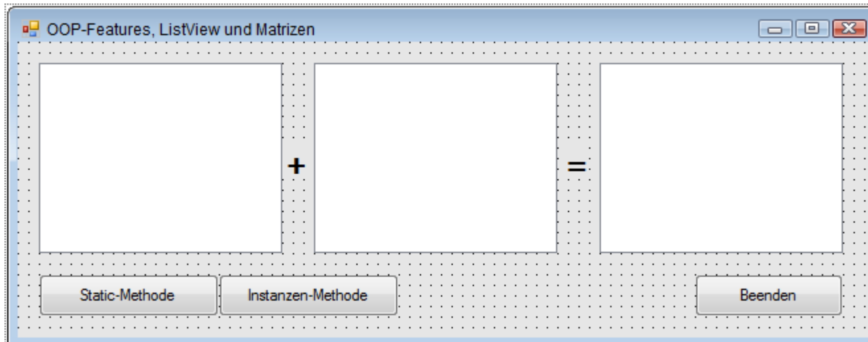
Der Unterschied zwischen statischen- und Instanzen-Methode dürfte Ihnen so richtig erst beim Sichten des Codes von *Form1* klar werden, wo beide Überladungen aufgerufen werden.

Hier ein Vorgriff auf den Code von *Form1*:

```
CMatrix A, B, C;
C = CMatrix.Add(A, B);                // Aufruf der  statischen Methode
C = A.Add(B);                         // Aufruf der Instanzen-Methode
```

Oberfläche

Wir benötigen drei *ListView*-Komponenten und drei *Button*s. Setzen Sie folgende zwei Eigenschaften für jede *ListView*: *View* = *Details* und *GridLines* = *True*.



Quellcode von Form1

```
public partial class Form1 : Form
{
    ...
```

Wir verwenden für beide *Button*s einen gemeinsamen Eventhandler:

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

Unser Beispiel benutzt Matrizen mit 9 Zeilen und 6 Spalten:

```
        const int rows = 9;              // Anzahl Zeilen
        const int cols = 6;              // Anzahl Spalten
```

Zufallszahlengenerator instanziieren:

```
        System.Random rnd = new System.Random();
```

Die Matrix *A* instanziieren, mit Zufallszahlen füllen und anzeigen (man beachte den bequemen Zugriff über den Indexer!):

```
        CMatrix A = new CMatrix(rows - 1, cols - 1);
        for (int i = 0; i < A.Rows; i++)
            for (int j = 0; j < A.Cols; j++)
                A[i, j] = rnd.Next(100);    // Zugriff auf Matrixelement über Indexer!
        showListView(A, listView1);        // Anzeige in linker ListView
```

Gleiches geschieht mit Matrix *B*:

```
        CMatrix B = new CMatrix(rows - 1, cols - 1);
        for (int i = 0; i < B.Rows; i++)
            for (int j = 0; j < B.Cols; j++)
                B[i, j] = rnd.Next(100);
        showListView(B, listView2);       // Anzeige in mittlerer ListView
```

Die resultierende Matrix *C* berechnen wir – in Abhängigkeit vom geklickten *Button* – mit der ersten oder mit der zweiten Überladung der *Add*-Methode.

HINWEIS: Beide Überladungen der *Add*-Methode leisten absolut das Gleiche, nur die Aufruf-Syntax ist unterschiedlich!

```
        CMatrix C;
        if ((Button) sender == button1)
            C = CMatrix.Add(A, B);           // Aufruf über statische Methode
        else
            C = A.Add(B);                    // Aufruf über Instanzen-Methode
        showListView(C, listView3);     // Resultatanzeige in rechter ListView
    }
```

Der Anzeigeroutine *showListView* werden ein *CMatrix*-Objekt und eine *ListView*-Komponente übergeben:

```
    private void showListView(CMatrix M, ListView lv)
    {
        lv.Clear();
```

Alle Spalten erzeugen und beschriften:

```
        lv.Columns.Add("", 20, HorizontalAlignment.Right);     // linke (leere) Randspalte
        for (int j = 0; j < M.Cols; j++)                       // alle Spalten erzeugen
```

Spaltennummerierung und Formatierung in Kopfzeile:

```
            lv.Columns.Add(j.ToString(), 30, HorizontalAlignment.Right);
```

Alle Zeilen erzeugen, beschriften und Zellen füllen:

```
            for (int i = 0; i < M.Rows; i++)
            {
```

Pro Zeile ein *ListViewItem*, Zeilennummerierung in linke Randspalte eintragen:

```
                ListViewItem item = new ListViewItem(i.ToString());
                for (int j = 0; j < M.Cols; j++)
```

Alle Zellen füllen (pro Zelle ein *SubItem*):

```
                    item.SubItems.Add(M[i, j].ToString());
```

Zeile zur *ListView* hinzufügen:

```
                lv.Items.Add(item);
            }
        }
    ...
```

Test

Nach Programmstart werden die beiden ersten Matrizen mit Zufallszahlen zwischen 0 und 100 gefüllt. Ob Sie dann *Button1* oder *Button2* klicken ist völlig egal, in beiden Fällen wird die resultierende Summen-Matrix mit dem richtigen Ergebnis gefüllt:

OOP-Features, ListView und Matrizen

	0	1	2	3	4
0	33	68	99	31	47
1	15	56	79	77	33
2	20	55	67	54	21
3	74	94	28	97	4
4	19	24	71	14	96
5	78	96	1	46	65

+

	0	1	2	3	4
0	97	62	36	6	32
1	39	39	70	82	58
2	42	58	95	86	34
3	9	97	85	35	69
4	18	54	33	12	41
5	97	66	26	7	32

=

	0	1	2	3	4
0	130	130	135	37	79
1	54	95	149	159	91
2	62	113	162	140	55
3	83	191	113	132	73
4	37	78	104	26	137
5	175	162	27	53	97

Static-Methode Instanzen-Methode Beenden

Index

??-Operator 101
.NET WinRT-Profil 924
.NET-Framework 67

A

Abbruchbedingung 620
Abort 511
Abs 312
abstract 202, 203
Abstraktion 65
Access Control Entries 437
Accessor 168
ACE 437
ACL 437
Activator 828, 830
Add 683
AddAccessRule 437
AddAfterSelf 683
AddBeforeSelf 683
AddDays 309
AddExtension 472
AddFirst 683
AddHours 309
AddMinutes 309
AddMonths 309
AddRange 340
AddressList 873
AddressWidth 879
AddYears 309
Administrator 871
ADO.NET-Klassen 725
ADO.NET-Objektmodell 722
Aktionsabfrage 748
Aktivierung 1049
Alias 334
Ancestors 680
Anfangswerte 95
Anonyme Methoden 353, 374
Anonyme Typen 382
Anwendungsdienste 945
App.config 628
App.xaml 1040
AppBar 1033, 1133
AppData 1059
Append 457
AppendChild 666, 670, 671
AppendLine 499
AppendLinesAsync 1091
AppendText 459
AppendTextAsync 1091
Application 874
ApplicationDataCompositeValue 1064
ApplicationDataContainer 1064

ApplicationServices 880
args 904
Arithmetische Operatoren 110
Array 145, 273, 284, 413, 418
ArrayList 339, 344, 348
as-Operator 104
Assemblierung 57, 71, 80
Assembly 241
  dynamisch laden 828
  GetExecutingAssembly 826
  Laden 825
  LoadFrom 825
AssemblyInfo 880
AssemblyInfo.cs 901, 1038
Assemblyinformation 1039
Assets 1047
async 545
Asynchrone Programmierentwurfsmuster 534
Atn 312
Attribute 72, 649, 652, 663
Attributes 434, 1082
Auflistung 337
Ausgabefenster 616
Ausnahmen 639
Ausschneiden 821
Aussteuerungsbalken 887
Auswahlabfrage 751
Auto-Play 1069
Auto-Properties 172
AutoPlay 1010
AutoResetEvent 606
AvailablePhysicalMemory 879
AvailableVirtualMemory 879
await 545

B

BackgroundColor 907
BackgroundWorker 531
Barrier 603
base 192
Basisklassen 191
BatteryChargeStatus 880
Beep 510, 908
Befehlsdesign 957
Befehlsfenster 614
Befehlsleisten 957
BeginInvoke 527, 541
Begrüßungsbildschirm 1043
Benutzerschnittstelle 944
BigInteger 362
Bild einlesen 847
Bilder drucken 849
Bildschirm 877, 882
Binärdatei 461
BinaryFormatter 257, 259, 456, 464
BinaryReader 456, 461, 462
BinarySearch 283
BinaryWriter 456, 461, 462
BindingNavigator 474
BindingSource 257, 474
BitsPerPixel 883
BlockingCollection 603
bool 93
Boolesche Operatoren 113
BootMode 875
Border 986
BottomAppBar 1133
Bounds 883
Boxing 108
break 121, 145, 461
Breakpoints 618
Button 976
Byte 93, 484
Byte-Array 709, 777

C

C#-Compiler 53
C#-Source-Datei 55
Callback 537
CallbackTimer 552, 556
Caller Information 630
CallerFilePath 630
CallerLineNumber 630
CallerMemberName 630
CameraDeviceType 851
CancellationPending 532
CancellationToken 594, 605
CancellationTokenSource 594
Canvas 987
CaptureElement 1008
case 116, 145
ChangeClipboardChain 839
ChangeDatabase 731
char 93
CharmBar, jetzt per Win+H erreichbar) 1109
CheckAccess 529
CheckBox 978
CheckFileExists 443, 472
CheckPathExits 443
ChildNodes 668
class 56, 147, 155
ClassesRoot 824
ClassLoader 70
Clear 281, 283
ClickOnce-Deployment 792
Clipboard 819
  ContainsText 820
  GetImage 820
  SetText 820
Clone 282, 291, 292, 760
Close 188, 730
ClosedByUser 1050
Cloud 1065
CLR 68, 70
CLR-Threadpool 576
CLS 67
Code Contracts 644
Code Document Object Model 240
Code Manager 70
CodeDOM 240
CodeDomProviders 240
Codefenster 64
Collection 337, 419
CollectionBase 690
CollectionViewSource 1022, 1030
ColumnName 763
Columns 761, 763
ColumnSpan 990
COM-Komponenten 71
COM-Marshaller 70
ComboBox 999
Command 731, 744
CommandBar 1133
CommandBuilder 739
CommandLine 880
Commands 1035
CommandText 733
CommandTimeout 733
CommandType 734
Common Language Runtime 65, 70
Common Language Specification 67, 68
Common Type System 67, 69
CompanyName 880
CompareDocumentOrder 680
CompiledAssembly 242
Compiler 241
CompilerParameters 242
Complex 365
ComputerName 877
ConcurrentBag 603
ConcurrentDictionary 603
ConcurrentQueue 603

ConcurrentStack 603
Connection 726, 733
ConnectionString 728
ConnectionStringBuilder 731
ConnectionTimeout 730
ConsoleKeyInfo 910
ConsoleModifiers 911
const 99
Constraint 349
ContentsChanged 1153
continue 119
Contract 922, 1038
Convert 106
ConvertDataSetToXML 709
ConvertStringToByteArray 488
ConvertXMLToDataSet 710
Copy 434, 759
CopyAsync 1093
Copyright 880
CopyTo 283, 291, 292, 434
Cos 312
CountdownEvent 603
Create 458
CreateCommand 731, 732
CreateDirectory 431
CreateElement 666
CreateEncryptor 467
CreateFileAsync 1089
CreateFileQueryWithOptions 1087
CreateFolderAsync 1086
CreateFolderQueryWithOptions 1084
CreateFromFile 469
CreateGraphics 889
CreateInstance 242, 830
CreateNavigator 693, 717
CreateNew 470
CreateSubdirectory 431
CreateSubKey 824, 832
CreateText 458
CreateViewAccessor 470
CreationCollisionOption 1061, 1086
CreationTime 434
Credentials 1068
CRUD 772
CryptoStream 466, 467
CryptoStreams 487
CSV-Datei 497, 500
CTS 67
Current 338
CurrentClockSpeed 879
CurrentConfig 824
CurrentCulture 875
CurrentDirectory 880
CurrentStateChanged 1010
CurrentUser 824
CursorLeft 907
CursorTop 907
CursorVisible 907

D

Data Encryption Standard 487
DataAdapter 743
Database 729
DataBindings 260
DataChanged 1066
DataColumn 761
DataContext 786
DataGrid 709
DataGridView 257
DataPackage 1103, 1109
DataPackageView 1102
DataReader 740
DataRequested 1110
DataSet 709
DataSource 729
DataTable 760
DataTemplate 1007

DataTime 308
DataTransfer 1102
DataTransferManager 1109, 1110
DataView 765, 773
DateCreated 1082
Datei komprimieren 490
Datei verschlüsseln 487
Dateiattribute 434
Dateien kopieren und verschieben 433
Dateien umbenennen 434
Dateiname 503
Dateiparameter 457
Dateipicker 1094
Dateitypzuordnung 1075
Dateiverknüpfung 1075
Dateiverknüpfungen 833
Datenkonsument 721
Datenprovider 721, 722
Datenquelle 768, 787
Datenstrukturen 602
Datentypen 92, 144
Datentypzuordnung 1070
Datenzugriff 124
DateTime 308
Datumsformatierung 316
Datumsfunktionen 307
Day 308
DayOfWeek 308
DayOfYear 308
DaysInMonth 309
DbProviderFactories 724
Deadlocks 507
Debug 623
    Write 624
    WriteIf 624
    WriteLineIf 624
Debuggen 1054
Debugger 613
DebugView 1056
decimal 93
Decrypt 465
default 116, 119
DefaultExt 472
DefaultFileExtension 1097
Deklarationen 1045
Dekrement 110
Delegate 148, 176, 349, 374
Delegate instanziieren 352
Delete 431, 764
DeleteAsync 1086, 1093
DeleteCommand 744
DeleteContainer 1067
DeleteSubKey 824
DeleteSubKeyTree 824, 833
DeleteValue 824
Deployment 793
Depth 696
DereferenceLinks 443
DES 487
Descendants 681
Description 880
DESCryptoServiceProvider 467, 488
Deserialize 259
Designer 63
DesktopDirectory 444
Destruktor 182, 185
DeviceInformation 950, 1008
DeviceInformationCollection 1008
DeviceManager 844
DeviceName 883
Dezimalzahlen 483
Diagnostics 509
DialogResult 472
Dictionary 348, 905
Dimensionsgrenzen 278
Direction 739
Directory 426, 430
DirectoryInfo 426

DirectoryName 430
DirectorySecurity 437
DisplayType 1082
Dispose 736
Distinct 410
do 119, 120
DOCTYPE 652
Document Object Model 663
Document Type 663
Document Type Definition 648
DocumentsLibrary 1063
DOM 663
DotNetZip Library 492
double 93
DownloadsFolder 1062
DragItemsStarting 1003
DriveInfo 426
DTD 648
Duplikate 410
Dynamische Programmierung 358
DynData 824

E

Eigenschaften 59, 164
Eigenschaften-Fenster 64
Eigenschaftsmethoden 267
Einfügen 821
Einzelschritt-Modus 621
Element 649, 652, 663, 708, 1162
Elements 681
else 144
else if 116
EnableRaisingEvents 440
Encrypt 465
EndElement 708
EndInvoke 527, 541
EndsWith 286
Enter 519
Entwicklungsumgebung 60
enum 122, 146
Enumerable 413, 418
Enumerationen 146
Environment 904
Environment Variablen 878
Ereignis 148, 176
Ereignis auslösen 178
Ereignisse 59, 176
Erweiterungsmethoden 384, 401
event 148, 176
EventLog 629
EventLogTraceListener 629
Events 59
Exception 640
ExceptionManager 70
ExecutablePath 880
ExecuteNonQuery 732, 735
ExecuteReader 732, 735, 740
ExecuteScalar 732, 735
Exists 435
Exit 519
Exp 312
ExpandAll 706
Exponentialfunktion 314
ExtClock 879
Extension 430, 1038
Extension Method-Syntax 386, 401

F

FailIfExists 1062
Fast and Fluid 921
Fehlerbehandlung 631
Fehlerklassen 633, 641
FieldCount 742
File 426, 456
File Type Association 1075
FileAccess 457

FileDropList 819
FileExtension 848
FileInfo 426, 456, 459
FileIO 1090, 1091
FileMode 457
FileName 442, 472
FileOpenPicker 1094
FileSavePicker 1094, 1097
FileSecurity 437
FileShare 458
FileStream 259, 456
FileSystemAccessRule 437
FileSystemWatcher 426, 440
FileTypeChoices 1097
Fill 745
Filter 440, 472
FilterIndex 442
Filtern 765
Filters 442
Find 764
FindAllAsync 950
Fingereingabe 960
FirstChild 668, 702, 716
FlipView 1006
float 93
Flyout 1129
FolderBrowserDialog 443
FolderDepth 1087, 1088
FolderPicker 1094, 1098
FolderRelativeId 1082
FontFamily 876
Fonts 444
FontSmoothingContrast 876
FontSmoothingType 876
for 119, 120, 145
for-each 701
foreach 145, 200, 276, 348
ForegroundColor 907
Format 317
Formatters 259
Formulare 58
Frame 955, 1011
Freigabeziel 1113, 1154
FromCurrentSynchronizationContext 599, 611
FullName 430
Funktionen 146, 1044
FutureAccessList 1099

G

Garbage Collector 185
GenerateExecutable 242
GenerateInMemory 242
GenerateUniqueName 1061, 1086
Generics 343, 345
generische Schnittstelle 412
Geräteeigenschaften 846
Gestensteuerung 983
get 147, 164, 168
GetBasicPropertiesAsync 1082
GetBitmapAsync 1103
GetChanges 760
GetCommandLineArgs 904
GetCreationTime 435
GetCurrent 872
GetCurrentDirectory 432, 884
GetDataAsync 1103
GetDataObject 819
GetDataPresent 820
GetDefaultView 1030
GetDirectories 432
GetElementsByTagName 672
GetEnumerator 338, 347
GetEnvironmentVariables 878, 905
GetExecutingAssembly 880, 1040
GetFactoryClasses 724
GetFields 827
GetFiles 436

GetFilesAsync 1087
GetFoldersAsync 1084
GetHostEntry 873, 874
GetHostName 873
GetLength 283, 292
GetManifestResourceNames 836
GetManifestResourceStream 837
GetMembers 827
GetMethod 830
GetMethods 827
GetProcessById 567
GetProcesses 566
GetProperties 827
GetShortPathName 884
GetStorageItemsAsync 1103
GetString 710
GetSubKeyNames 824
GetTypeInfo 1039
GetTypes 826
GetValue 742, 832
GetValueNames 824
GetValues 742
Gigabyte 484
goto 119
GPS 944
Grafikbearbeitung 852
Grid 989
GridView 1004
GridView gruppieren 1021
Gruppen 868
GZipStream 490

H

Haltepunkte 620
Hashtable 341
HasValue 101
Hauptprogramm 902
HeaderTemplate 1022
Help 814
HelpMaker 817
HelpNameSpace 817
HelpProvider 816
Hexadezimal 483
Hilfe-IDE 817
Hilfedatei 809, 813
Hilfemenü 814
HomeGroup 1063
Hour 308
Hover 976
HTML 645
HTML 5 954
HTML Help Workshop 810
HtmlFormatHelper 1104
HyperlinkButton 976

I

IAsyncResult 536, 537, 538, 543
ICollection 338
IComparable 252
IComparer 252
ICryptoTransform 489
IDataObjekt 820
IDisposable 736
Idle-Prozesse 566
IEnumerable 337, 345, 413, 421
IEnumerator 338
IEqualityComparer 412
if 116, 144
IgnoreWhitespace 692, 708
IInspectable 946
ILSpy 925
Image 983
ImageFile 848, 852
ImageProcess 852
ImportRow 762
Indent 699

IndentChars 699
IndentLevel 625
IndentSize 625
Index 273
Indexer 267, 336, 343, 372
IndexerOption 1088
IndexOf 283, 286
Indexprüfung 275
InitialDirectory 472
Initialisierer 413
Initialisierung 157
Initialize 283, 292
Inkrement 110
InnerText 673
InputScopeName 1018
Insert 286
InsertCommand 744
Installationsverzeichnis 1061
InstalledLocation 1061
InstallShield 800
Instanz 152
Instanziieren 156
int 93
Int16 93
Int32 93
Int64 93
Intellisense 161
internal 153
internal protected 154
InteropServices 892
Interrupt 511
Invoke 526, 539, 541, 828, 830
InvokeRequired 528
Ionic.Zip.dll 492
IP-Adresse 872
IPAddress 874
IsAbstract 827
IsActive 985
IsAdmin 871
IsAfter 681
IsAlive 512
IsBackGround 513
IsBefore 681
IsClass 826
IsClosed 742
IsCOMObject 827
IsCompleted 536, 582
IsEnum 827
IsFontSmoothingEnabled 876
IsInterface 827
IsLeapYear 309
IsLooping 1010
IsMuted 1010
IsPublic 827
IsSealed 827
IsSourceGrouped 1021
Item 742
ItemsControl 999
ItemsPanelTemplate 1022
Iterator 583
Iteratoren 347
iTextSharp 495
IUnknown 946

J

JavaScript 954
JIT-Compiler 66
Join 499, 511

K

Kapselung 65, 152
Kartenspiel 263
Kartesische Koordinaten 231
Kilobyte 484
Klasse 152
Klassendefinition 147

KnownFolders 1063
Kommentare 91, 649
komplexe Zahlen 231
Komprimieren 467
Konsolenanwendung 133, 901
Konstante Felder 170
Konstanten 92, 99
Konstruktor 182, 565
Konstruktor überladen 267
Kontaktliste 1114
Kontextmenü 833
Kontravarianz 362
Konverter 483
Kopieren 821
Kovarianz 362
kritische Abschnitte 552
Kurz-Operatoren 111
Kurzschlussauswertung 114

L

Lambda Expression 374
Lambda-Ausdruck 407
Lambda-Ausdrücke 354, 401
Language Projection 924, 946
LastAccessTime 434
LastWriteTime 435
Laufwerke 432
Launch-Contract 1049
Layout-Control 987
Lebenszyklus 1047
Length 282, 286, 291, 292
LINQ 407, 409, 418, 419, 422, 501
  Abfrage-Operatoren 387
  Aggregat-Operatoren 395
  AsEnumerable 398
  Count 395
  GroupBy 392
  Grundlagen 381
  Gruppierungsoperator 392
  Join 395
  Konvertierungsmethoden 398
  OrderBy 391
  OrderByDescending 391
  Projektionsoperatoren 389
  Restriktionsoperator 390
  Reverse 392
  Select 389
  SelectMany 389
  Sortierungsoperatoren 391
  Sum 396
  ThenBy 391
  ToArray 398
  ToDictionary 398
  ToList 398
  ToLookup 398
  Where 390
LINQ to XML-API 675
LINQ-Abfrageoperatoren 385
LINQ-Architektur 381
LINQ-Provider 382
LINQ-Syntax 385
List 343, 348
List Klasse 348
ListBox 999
ListView 1003
Load 679
LoadCompleted 1012
LoadedAssemblies 881
LoadXml 664
LocalApplicationData 444
LocalFolder 1059
LocalMachine 824
LocalState 1059
lock 517
Log 312
Log10 312
Logarithmus 314

LogicalDpi 967
Logische Operatoren 112
Lokal-Fenster 615
Lokale Variablen 102
long 93
LongRunning 598
LowestBreakIteration 582

M

MachineName 877
Main 903
MainModule 566
ManagementObject 868
ManagementObjectSearcher 868
Manifestressourcen
  Betrachter 835
ManipulationDelta 983
ManipulationMode 983
ManualResetEvent 609
ManualResetEventSlim 603
Manufacturer 879
Map View 470
Mapperklassen 787
Matrix 267
Matrizen 272
Max 313
MaximumRowsOrColumns 990
MCI 883, 892
mciGetErrorString 884, 893
mciSendString 884, 893
Media 509
MediaCapture 1008
MediaElement 1010
MediaEnded 1010
MediaOpened 1010
Megabyte 484
Memory Mapped File 468, 481
MemoryMappedFile 469
MemoryStream 709, 777, 782
MenuStrip 874
MessageDialog 1035, 1116
Messwertliste 404
Metadaten 71, 925
Metasprache 645
Methoden 59, 125, 146, 172
  generische 346
  überladen 139
Methoden überladen 267
Methodenzeiger 350
MethodImpl 524
MethodInfo 241
Methods 59
Microsoft Intermediate Language Code 66
Microsoft.VisualBasic 874
Microsoft.VisualBasic.Devices 875
Mikrofon 886
Mikrofonpegel 887
Min 313, 482
Minute 308
MMF 468
Modifiers 911
Monitor 519
MonitorCount 877
MonitorsSameDisplayFormat 877
Month 308
MostRecentlyUsedList 1101
Move 431, 434
MoveAndReplaceAsync 1093
MoveAsync 1093
MoveBufferArea 907
MoveNext 338
MoveTo 434
MoveToNext 694, 716
MoveToPrevious 694, 716
MoveToRoot 716
MRU-Liste 1101
MSIL-Code 66, 80

Multiselect 443
Multitasking 506
Multithreading 73, 506, 552
MusicLibrary 1063
Mutex 522
MyComputer 444
MyDocuments 444

N

nameof 479
Namespace 70, 156, 826
Navigate 961
NavigateToString 1011
Navigation 961
NavigationCacheMode 963
Navigationsdesign 959
NET-Reflection 825
Network 877
Netzwerk 877
new 148, 182, 274
NewRow 762
Next 415
NextSibling 668, 702
Nodes 681
NodeType 697
Notification 944
NotifyFilter 440
NotRunning 1050
Now 309
NTFS 466
null 100, 158
Nullable Type 100
Nutzer 868

O

object 93, 99
Objekt 151
Objektbaum 257, 474
Objekte 148
Objektinitialisierer 184, 212, 383
ODER 114
OleDbConnection 726
OnActivated 1051
OnCachedFileUpdaterActivated 1051
OnFileActivated 1051, 1069, 1071
OnFileOpenPickerActivated 1051
OnFileSavePickerActivated 1051
OnLaunched 961, 1040, 1049
OnNavigatedTo 962
OnSeachActivated 1051
OnShareTargetActivated 1051, 1113
OnSuspending 1040
OOP 149, 263
Open 458, 730
OpenFileDialog 441, 471
OpenOffice.org 860
OpenOrCreate 458
OpenSubKey 833, 876
OpenText 460
Operatoren 109, 144
Operatorenüberladung 231
Optionale Parameter 361
orderby 419
OSFullName 875
OSVersion 875
out 129
override 192
OverwritePrompt 472

P

Package.appxmanifest 1041
PadLeft 286
PadRight 286
Page 955
PAP 133

Paragraph 976
Parallel LINQ 603
Parallel-Programmierung 573
Parallel.For 579
Parallel.ForEach 583
Parallel.Invoke 577
ParallelLoopResult 582
Parameter 737
ParameterName 738
Parameterübergabe 128, 129, 904, 961, 969
ParentNode 702
Parse 106, 309
Parser 652
PasswordBox 979
PasswordVault 1068
Path 426, 439
Pause 1010
PC-Name 877
PDF 494
PDFsharp 496
PeekChar 462
Pegeldiagramm 888
PerformanceData 824
PI 313, 652
PickSaveFileAsync 1097
PicturesLibrary 1063
PlacementTarget 1012
Platform 875
PlatformID.Win32Windows 876
Play 1010
PLINQ 398, 603
PointToScreen 816
Polarkoordinaten 231
Polling 535
Polymorphes Verhalten 199
Polymorphie 65, 153, 189, 201
PopUp 1129
Popup-Benachrichtigungen 1121
PopUp-Hilfe 815
PopupMenu 1132
portieren 142
Position 1011
Potenz 313
Pow 313
PowerLineStatus 880
PowerStatus 880
PreferFairness 598
Press 976
PreviousExecutionState 1050
PreviousSibling 702
Primary 883
PrimaryMonitorMaximizedWindowSize 877
PrimaryMonitorSize 877
Priority 512
private 153
Procedure-Step 618
Process 565, 570
Process Sandboxing 922
Process.Start 571
Processing Instructions 648, 652, 663
ProcessName 566
ProcessorCount 879
ProcessThread 565
ProductName 880
ProductVersion 880
Program 902
Program.cs 901
Programm starten 570
ProgressBar 887, 985
ProgressChanged 533
ProgressRing 985
ProhibitDtd 708
Projektmappen-Explorer 62
Projekttyp 62
Prolog 648
Properties 59
Property 209
Property-Accessoren 168

PropertyChangedEventHandler 934
protected 154
Protokolldatei 1055
Provider 726, 729
Prozeduren 146
Prozedurschritt 622
Prozesse 565
public 153
Pulse 519, 520
PulseAll 519, 520

Q

Query Expression-Syntax 386, 401
QueryOptions 1084, 1087
Queue 345, 348
QueueUserWorkItem 514

R

Racing 507
RadioButton 978
Random 264, 314, 415
RandomAccessStreamReference 1105
Range 414
Rank 282, 291, 292
Read 457
ReadAllBytes 462
ReadAllLines 460
ReadAllText 460
ReadBufferAsync 1091
ReadContentAsFloat 698
ReadKey 907, 910
ReadLine 56
ReadLines 460
ReadLinesAsync 1091
ReadTextAsync 1091
ReadToEnd 460
ReadWrite 457
ReadXml 687, 714, 777, 782
ref 128, 129
Referenzieren 156
Referenztyp 99
Referenztypen 285
Reflection 241, 1039
Reflection-Mechanismus 240
Reflexion 71
Regedit.exe 822
Regex 1020
Registrierungsdatenbank 822
Registrierungseditor 822
Registry 821, 824, 831, 876
RegistryKey 822, 824, 831
Relationen 709
Release 976
ReleaseMutex 523
Remote-Debugging 931

RemovableDevices 1063
Remove 286, 684, 764
RemoveAccessRule 437
RemoveAll 684
RemoveAnnotations 684
RemoveAttributes 684
RemoveContent 684
Repeat 415
RepeatButton 976
Replace 286
Reset 338
ResolutionScale 967
Resume 511
Resuming 1053
RetrievePropertiesAsync 1083
return 119, 147, 347
RichEditBox 979
RichTextBlock 973
Roaming 1065
Roamingdaten 1067
RoamingFolder 1060
RoamingStorageQuota 1060
Round 313
RowFilter 765
Rows 763
RowSpan 990
Rückrufmethode 535
Rücksprung 622
Run 976
Runden 313
Running 1047

S

Sandboxing 920
SaveFileDialog 441, 471
Scanner 855
Scanner-Assistent 850
ScannerDeviceType 851
Schaltjahr 309
ScheduledToastNotification 1163
Schleifen 145
Schleifenabbruch 581
Schleifenanweisungen 119
Schlüsselwörter 91
Schriftarten 876, 960
Screen 882
ScreenOrientation 877
ScrollBar 985
ScrollViewer 988
sealed 204
Second 308
Security Engine 70
Seitennavigation 969
Seitenstatus 962
SELECT 419, 694, 751
SelectCommand 744
SelectNodes 672
SelectSingleNode 669, 670, 671, 672, 716
SemanticZoom 1025
Semaphore 524
SemaphoreSlim 603
SendMessage 839
Sequenzielle Datei 463
Serialisieren 464
Serialisierung 73
Serializable 257, 464
Serializable-Attribut 475
Serialization 259
Serialize 259
ServerVersion 729
ServicePack 875
set 147, 164, 168
SetAccessControl 438
SetAttributeValue 683
SetBitmap 1103, 1105
SetBufferSize 907
SetClipboardViewer 839

SetCurrentDirectory 432
SetCursorPosition 907
SetData 1103
SetDataObject 819
SetElementValue 683
SetSource 1010
SetStorageItems 1103, 1107
Settingsbereich 959
Setup-Projekt 800
SetValue 832
SetWindowPosition 907
SetWindowSize 907
Shared Methoden 267
ShareOperation 1157
short 93
ShowAcquireImage 859
ShowAcquisitionWizard 851
ShowAsync 1116
ShowDialog 472
ShowForSelectionAsync 1132
ShowHelp 814
ShowPopup 815
ShowShareUI 1112
Sign 313
Sin 313
Single-Step 618
SizeChanged 964
Skip 682
SkipWhile 682
Sleep 511
Slider 985
SnapsTo 985
SocketDesignation 879
Sort 252, 283, 765
SortedList 348
SortedSet 366
Sortieren 418, 765
Sound 883, 892
SpecialFolder 443
Sperrmechanismen 515
SpinLock 603
SpinWait 603
Splash Screen 1043, 1051, 1073
Split 286
SQLite 1138
SQLiteConnection 1141
Sqr 313
Stack 345
StackPanel 988
Stammelement 650
StandardDataFormats 1157
StartInfo 570
StartPreviewAsync 1009
Startseite 961
StartsWidth 286
State 730
static 147, 205
Statische Klassen 205
Statische Methoden 174
Statischer Konstruktor 184
Statusmitteilung 1123
StepValue 985
Steuerelemente 58
Stop 1010
Stopped 1047
Storage-Interface 944
StorageApplicationPermissions 1099
StorageDeleteOption 1086
StorageFile 1089, 1093
StorageFileQueryResult 1152
StorageFolder 1081
Store 926
StoredProcedure 734
StreamReader 456, 460, 1092
StreamWriter 456, 459, 498, 1091
String 93, 285
Stringaddition 322
StringBuilder 499, 884

StringFormatConverter 1015
StringReader 456, 777
StringWriter 456
struct 123, 146, 463
Strukturen 146
Strukturvariable 124
SubKeyCount 824
Subklassen 192, 193
SubmitChanges 789
SubString 286
Suchen 766
SuggestedStartLocation 1098
Suspend 511
Suspended 1050
Suspending 1047, 1051
SuspendingDeferral 1052
SVG 983
switch 116, 145
System 93, 444
System.Collections.Concurrent 603
System.Console 906
System.Diagnostics 565
System.Environment 874
System.IO.Compression 467
System.IO.FileStream 455
System.IO.Stream 455
System.Management 868
System.Net 872
System.Nullable 100
System.Object 201
System.Reflection 825
System.Security.AccessControl 438
System.Security.Cryptography 466
System.Security.Principal 871
System.Threading 509, 576
System.Threading.Tasks 576
System.TimeSpan 566
System.Xml 664, 703
System.Xml.Linq 675
System.Xml.Serialization 692
System.Xml.XPath 696
System.Xml.Xsl 700
SystemInformation 874

T

TableAdapter 769
TableDirect 734
TableName 763
Tablet-Simulator 984
Take 682
TakeWhile 682
Tan 313
Task
  Canceled 598
  ContinueWith 590, 599
  Created 598
  Datenübergabe 587
  Faulted 598
  Fehlerbehandlung 596
  IsCanceled 598
  IsCompleted 598
  IsFaulted 598
  Klasse 584
  RanToCompletion 598
  Result 590
  return 593
  Rückgabewerte 590
  Running 598
  Start 586
  Starten 585
  Status 598
  Task-Ende 599
  Task-Id 597
  TaskCreationOptions 598
  User-Interface 598
  Verarbeitung abbrechen 593
  Wait 588

WaitAll 589
WaitingForActivation 598
WaitingForChildrenToComplete 598
WaitingToRun 598
Weitere Eigenschaften 597
Task.Factory.StartNew 584, 606
TaskCreationOptions 605
TaskScheduler 599, 605
Tastaturabfrage 910
Tasteneingaben 1019
TemporaryFolder 1060
TemporaryKey.pfx 1046
TerminalServerSession 878
Terminated 1050
TextBlock 973
TextBox 979
Textdatei 458, 471
TextWriterTraceListener 627
Thin Client 214
Thread 509, 510, 552
initialisieren 552
synchronisieren 552
ThreadState 567
Thread Service 70
Thread<> 549
ThreadInteruptedException 511
ThreadPool 513
Threads 565
threadsicher 526
Threadsichere Collections 602
ThreadState 512
ThreadWaitReason 568
Throw 634, 641
ThrowIfCancellationRequested 594
Thumbnail-Ansicht 1095
TickValues 985
Tiles 920
Timer 887
Timer-Threads 530
TimeSpan 566
TimeSpan-Klasse 322
Title 442, 472, 881
ToArray 410, 419, 421, 709
ToastNotificationFactory 1122
ToastNotificationManager 1161, 1164
ToastNotifications 1159
ToCharArray 286
Today 309
ToggleButton 978
ToggleSwitch 978
ToLongDateString 309
ToLongTimeString 309
ToLower 286
Toolbox 63
ToolTip 1012
ToolTipPlacement 1012
ToolTipService 1012
TopAppBar 1133
ToShortDateString 309
ToShortTimeString 309
ToString 105, 315
TotalPhysicalMemory 879
TotalProcessorTime 566
TotalVirtualMemory 879
Touchscreen 939
ToUpper 286
TPL 576
Trace 623, 627
TraceListener 627
TrackBar 322
Trademark 881
Transform 700
Transformationsdatei 700
TreeView 448, 705
Trefferanzahl 621
Trim 286
Truncate 458
try 145

Try-Catch 632
Try-Finally 637
TryEnter 519, 522
Tuple 365
Type 241, 826
Typecasting 344, 371
Typinferenz 101, 383, 401
Typisierte DataSets 766
Typsicherheit 344
Typsuffixe 96

U

Überladene Methoden 173
Überwachungsfenster 616
Uhr anzeigen 310
UICommand 1035, 1132
Umgebungsvariablen 905
UML 189
Unboxing 108
UND 114
Unicode 97
UnicodeEncoding 488, 710, 777
Unified Modeling Language 189
UnIndent 625
UnspecifiedDeviceType 851
Unterverzeichnis 432
Update 739, 746
UpdateCommand 744
Updates 795
UseIndexerWhenAvailable 1088
User-Name 877
UserDomainName 878
UserInteractive 878, 880
UserName 878
UserProfile 444
Users 824
using 56, 156, 333, 481
using static 213
UTF-16 652
UTF-8 652

V

ValidateNames 472
Value 738
ValueCount 824
var 95, 101
Variablen 92
Variablentypen 92
VariableSizedWrapGrid 990, 1023
VB 142
Verarbeitungsstatus 582
Vererbung 152
Vergleichsoperatoren 112
Veröffentlichen 797
Verpacken 1046, 1146
Verschlüsseln 465
Version 882
VersionString 875
Vertrieb 1146
Verweistypen 94
Verzögerte Initialisierung 188
Verzweigungen 144
Video 892
VideoCapture 1008
VideoDeviceType 851
VideosLibrary 1063
ViewManagement 964
ViewMode 1095
VirtualScreen 877
Virtuelle Tastatur 1017
Visual Studio 51
Visual Studio Enterprise 52
Visual Studio Professional 52
VisualBasic 241
VisualStateManager 965
void 127

Vollbildmodus 926
Volume 1010

W

W3C 663
Wait 519, 520
WaitForExit 570
WaitOne 523, 524, 607, 611
WAV 883
Webcam 853, 944
Webpublishing-Assistent 793, 799
WebView 1011
WebViewBrush 1012
Wechselspeichergeräte 1069
Werkzeugkasten 63
Wertetypen 94
Where 349, 682
while 119, 120, 145
WIA 841
wiaaut.dll 842
Wiederholmuster 417
Wiederverwendbarkeit 152
Window 955
Windows 10 928
Windows App 920, 1037
Windows Management Instrumentations 874
Windows Runtime 919
Windows Store 926
Windows-Simulator 928
WindowsIdentity 869, 872
Winkel 313
WinMD 948
WinMD-Dateien 924
winmm.dll 893
WinRT 919, 941
WinRT-API 943
WinRT-COM 924
WinRT-Namespaces 945
WMI 874
work stealing 577
WorkerReportsProgress 533
WorkingArea 877, 883
WorkingSet 880
WPF 529
Write 457, 908
WriteAllBytes 462
WriteAllText 499
WriteAttributeString 698
WriteBufferAsync 1091
WriteEndDocument 699
WriteEndElement 698
WriteIf 623
WriteLine 56, 623, 908
Writer 860
WriteStartDocument 698
WriteStartElement 698
WriteTextAsync 1091
WriteXml 714, 777, 782
WriteXmlSchema 687
Wurzel 313

X

XAML 954
XAttribute 677, 1162
XComment 677
XDeclaration 677
XDocument 676, 678
XDocumentType 677
XElement 676, 1162
XElement.Load 679
XElement.Parse 679
XML 645
XML transformieren 684
XML-Datei 713
XML-Schema 658
XML-String 709

XmlAttribute 663, 692
XMLCDATASection 664
XMLCharacterData 664
XMLComment 664
XmlDataDocument 687
XmlDocument 687, 693, 702, 717, 1163
XmlDocumentType 664
XmlElement 664, 692
XmlEntity 664
XmlEnum 692
XmlIgnore 692
XmlImplementation 664
XMLNamedNodeMap 663
XmlNode 663, 667
XMLNodeList 663
XmlNodeType 708
XMLParseError 663
XmlProcessingInstructions 664
XmlReader 696, 707
XmlReaderSettings 696, 708
XmlRoot 692
XmlSerializer 690
XmlText 664
XmlTextWriter 777
XmlWriter 698
XmlWriterSettings 699
XNode 677
XOR 114
XPathDocument 693
XPathNavigator 693, 716, 717
XPathNodeIterator 694
XProcessingInstruction 677
XSD-Schema 656
xsd.exe 662
XslCompiledTransform 700
XSLT 700

Y

Year 308
yield 347, 374

Z

Zahlenformatierung 316
Zeitfunktionen 307
Zeitmessung 323
Zielplattformen 927
ZIP 492
ZipFile 493
ZoomedInView 1026
ZoomedOutView 1026
Zufallszahlen 314, 415
Zugriffsberechtigung 437
Zuweisungsoperatoren 111
Zwischenablage 838, 1102