

Leseprobe

Walter Doberenz, Thomas Gewinnus

Visual Basic 2015 – Grundlagen, Profiwissen und Rezepte

ISBN (Buch): 978-3-446-44380-8

ISBN (E-Book): 978-3-446-44605-2

Weitere Informationen oder Bestellungen unter

<http://www.hanser-fachbuch.de/978-3-446-44380-8>

sowie im Buchhandel.

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 VB-Programm	53
1.2.1 Vorbereitungen	53
1.2.2 Programm 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	58
1.3.2 Objekt- und ereignisorientierte Programmierung	58
1.3.3 Windows-Programmierung unter Visual Studio 2015	59
1.4 Die Entwicklungsumgebung Visual Studio 2015	61
1.4.1 Neues Projekt	61
1.4.2 Die wichtigsten Fenster	62
1.5 Microsofts .NET-Technologie	64
1.5.1 Zur Geschichte von .NET	64
1.5.2 .NET-Features und Begriffe	66
1.6 Wichtige Neuigkeiten in Visual Studio 2015	74
1.6.1 Entwicklungsumgebung	74
1.6.2 Neue VB-Sprachfeatures	74
1.6.3 Code-Editor	74
1.6.4 NET Framework 4.6	75

1.7	Praxisbeispiele	75
1.7.1	Windows-Anwendung für Einsteiger	75
1.7.2	Windows-Anwendung für fortgeschrittene Einsteiger	79
2	Einführung in Visual Basic	87
2.1	Grundbegriffe	87
2.1.1	Anweisungen	87
2.1.2	Bezeichner	88
2.1.3	Kommentare	89
2.1.4	Zeilenumbruch	90
2.2	Datentypen, Variablen und Konstanten	92
2.2.1	Fundamentale Typen	92
2.2.2	Werttypen versus Verweistypen	93
2.2.3	Benennung von Variablen	93
2.2.4	Deklaration von Variablen	94
2.2.5	Typinferenz	97
2.2.6	Konstanten deklarieren	97
2.2.7	Gültigkeitsbereiche von Deklarationen	98
2.2.8	Lokale Variablen mit Dim	98
2.2.9	Lokale Variablen mit Static	99
2.2.10	Private globale Variablen	99
2.2.11	Public Variablen	100
2.3	Wichtige Datentypen im Überblick	100
2.3.1	Byte, Short, Integer, Long	100
2.3.2	Single, Double und Decimal	101
2.3.3	Char und String	101
2.3.4	Date	102
2.3.5	Boolean	103
2.3.6	Object	103
2.4	Konvertieren von Datentypen	104
2.4.1	Implizite und explizite Konvertierung	104
2.4.2	Welcher Datentyp passt zu welchem?	105
2.4.3	Konvertierungsfunktionen	106
2.4.4	CType-Funktion	107
2.4.5	Konvertieren von Strings	107
2.4.6	Die Convert-Klasse	109
2.4.7	Die Parse-Methode	109
2.4.8	Boxing und Unboxing	110

2.4.9	TryCast-Operator	111
2.4.10	Nullable Types	111
2.5	Operatoren	112
2.5.1	Arithmetische Operatoren	112
2.5.2	Zuweisungsoperatoren	113
2.5.3	Logische Operatoren	114
2.5.4	Vergleichsoperatoren	115
2.5.5	Rangfolge der Operatoren	115
2.6	Kontrollstrukturen	116
2.6.1	Verzweigungsbefehle	116
2.6.2	Schleifenanweisungen	119
2.7	Benutzerdefinierte Datentypen	120
2.7.1	Enumerationen	120
2.7.2	Strukturen	121
2.8	Nutzerdefinierte Funktionen/Prozeduren	124
2.8.1	Deklaration und Syntax	124
2.8.2	Parameterübergabe allgemein	126
2.8.3	Übergabe mit ByVal und ByRef	127
2.8.4	Optionale Parameter	128
2.8.5	Überladene Funktionen/Prozeduren	128
2.9	Praxisbeispiele	129
2.9.1	Vom PAP zum Konsolen-Programm	129
2.9.2	Vom Konsolen- zum Windows-Programm	131
2.9.3	Schleifenanweisungen kennen lernen	133
2.9.4	Methoden überladen	136
2.9.5	Eine Iterationsschleife verstehen	138
2.9.6	Anwendungen von C# nach Visual Basic portieren	141
3	OOP-Konzepte	149
3.1	Strukturierter versus objektorientierter Entwurf	149
3.1.1	Was bedeutet strukturierte Programmierung?	149
3.1.2	Was heißt objektorientierte Programmierung?	150
3.2	Grundbegriffe der OOP	151
3.2.1	Objekt, Klasse, Instanz	151
3.2.2	Kapselung und Wiederverwendbarkeit	152
3.2.3	Vererbung und Polymorphie	152
3.2.4	Sichtbarkeit von Klassen und ihren Mitgliedern	153
3.2.5	Allgemeiner Aufbau einer Klasse	154

3.3	Ein Objekt erzeugen	155
3.3.1	Referenzieren und Instanzieren	156
3.3.2	Klassische Initialisierung	157
3.3.3	Objekt-Initialisierer	157
3.3.4	Arbeiten mit dem Objekt	157
3.3.5	Zerstören des Objekts	158
3.4	OOP-Einführungsbeispiel	158
3.4.1	Vorbereitungen	158
3.4.2	Klasse definieren	159
3.4.3	Objekt erzeugen und initialisieren	160
3.4.4	Objekt verwenden	160
3.4.5	Unterstützung durch die IntelliSense	160
3.4.6	Objekt testen	161
3.4.7	Warum unsere Klasse noch nicht optimal ist	162
3.5	Eigenschaften	162
3.5.1	Eigenschaften kapseln	162
3.5.2	Eigenschaften mit Zugriffsmethoden kapseln	165
3.5.3	Lese-/Schreibschutz für Eigenschaften	166
3.5.4	Statische Eigenschaften	167
3.5.5	Selbst implementierende Eigenschaften	168
3.6	Methoden	169
3.6.1	Öffentliche und private Methoden	169
3.6.2	Überladene Methoden	170
3.6.3	Statische Methoden	171
3.7	Ereignisse	172
3.7.1	Ereignisse deklarieren	172
3.7.2	Ereignis auslösen	173
3.7.3	Ereignis auswerten	173
3.7.4	Benutzerdefinierte Ereignisse (Custom Events)	175
3.8	Arbeiten mit Konstruktor und Destruktor	178
3.8.1	Der Konstruktor erzeugt das Objekt	178
3.8.2	Bequemer geht's mit einem Objekt-Initialisierer	180
3.8.3	Destruktor und Garbage Collector räumen auf	181
3.8.4	Mit Using den Lebenszyklus des Objekts kapseln	184
3.9	Vererbung und Polymorphie	184
3.9.1	Vererbungsbeziehungen im Klassendiagramm	184
3.9.2	Überschreiben von Methoden (Method-Overriding)	186
3.9.3	Klassen implementieren	186

3.9.4	Objekte implementieren	191
3.9.5	Ausblenden von Mitgliedern durch Vererbung	192
3.9.6	Allgemeine Hinweise und Regeln zur Vererbung	194
3.9.7	Polymorphe Methoden	195
3.10	Besondere Klassen und Features	197
3.10.1	Abstrakte Klassen	197
3.10.2	Abstrakte Methoden	198
3.10.3	Versiegelte Klassen	198
3.10.4	Partielle Klassen	199
3.10.5	Die Basisklasse System.Object	201
3.10.6	Property-Accessors	202
3.10.7	Nullbedingter Operator	202
3.11	Schnittstellen (Interfaces)	203
3.11.1	Definition einer Schnittstelle	203
3.11.2	Implementieren einer Schnittstelle	204
3.11.3	Abfragen, ob eine Schnittstelle vorhanden ist	205
3.11.4	Mehrere Schnittstellen implementieren	205
3.11.5	Schnittstellenprogrammierung ist ein weites Feld	205
3.12	Praxisbeispiele	206
3.12.1	Eigenschaften sinnvoll kapseln	206
3.12.2	Eine statische Klasse anwenden	209
3.12.3	Vom fetten zum dünnen Client	211
3.12.4	Schnittstellenvererbung verstehen	220
3.12.5	Aggregation und Vererbung gegenüberstellen	224
3.12.6	Eine Klasse zur Matrizenrechnung entwickeln	230
3.12.7	Rechner für komplexe Zahlen	236
3.12.8	Formel-Rechner mit dem CodeDOM	244
3.12.9	Einen Funktionsverlauf grafisch darstellen	249
3.12.10	Sortieren mit IComparable/IComparer	253
3.12.11	Objektbäume in generischen Listen abspeichern	258
3.12.12	OOP beim Kartenspiel erlernen	264
4	Arrays, Strings, Funktionen	269
4.1	Datenfelder (Arrays)	269
4.1.1	Ein Array deklarieren	269
4.1.2	Zugriff auf Array-Elemente	270
4.1.3	Oberen Index ermitteln	270
4.1.4	Explizite Arraygrenzen	270

4.1.5	Arrays erzeugen und initialisieren	270
4.1.6	Zugriff mittels Schleife	271
4.1.7	Mehrdimensionale Arrays	272
4.1.8	Dynamische Arrays	273
4.1.9	Zuweisen von Arrays	274
4.1.10	Arrays aus Strukturvariablen	275
4.1.11	Löschen von Arrays	276
4.1.12	Eigenschaften und Methoden von Arrays	276
4.1.13	Übergabe von Arrays	279
4.2	Zeichenkettenverarbeitung	280
4.2.1	Strings zuweisen	280
4.2.2	Eigenschaften und Methoden eines Strings	280
4.2.3	Kopieren eines Strings in ein Char-Array	283
4.2.4	Wichtige (statische) Methoden der String-Klasse	283
4.2.5	Die StringBuilder-Klasse	285
4.3	Reguläre Ausdrücke	288
4.3.1	Wozu braucht man reguläre Ausdrücke?	288
4.3.2	Eine kleine Einführung	289
4.3.3	Wichtige Methoden der Klasse Regex	289
4.3.4	Kompilierte reguläre Ausdrücke	291
4.3.5	RegexOptions-Enumeration	292
4.3.6	Metazeichen (Escape-Zeichen)	293
4.3.7	Zeichenmengen (Character Sets)	294
4.3.8	Quantifizierer	295
4.3.9	Zero-Width Assertions	296
4.3.10	Gruppen	300
4.3.11	Text ersetzen	300
4.3.12	Text splitten	301
4.4	Datums- und Zeitberechnungen	302
4.4.1	Grundlegendes	302
4.4.2	Wichtige Eigenschaften von DateTime-Variablen	303
4.4.3	Wichtige Methoden von DateTime-Variablen	304
4.4.4	Wichtige Mitglieder der DateTime-Struktur	305
4.4.5	Konvertieren von Datumstrings in DateTime-Werte	306
4.4.6	Die TimeSpan-Struktur	306
4.5	Vordefinierten Funktionen	308
4.5.1	Mathematik	308
4.5.2	Datums- und Zeitfunktionen	310

4.6	Zahlen formatieren	312
4.6.1	Die ToString-Methode	313
4.6.2	Die Format-Methode	314
4.6.3	Stringinterpolation	316
4.7	Praxisbeispiele	316
4.7.1	Zeichenketten verarbeiten	316
4.7.2	Zeichenketten mittels StringBuilder addieren	319
4.7.3	Reguläre Ausdrücke testen	323
4.7.4	Fehler bei mathematischen Operationen behandeln	325
4.7.5	Methodenaufrufe mit Array-Parametern	328
4.7.6	String in Array kopieren und umgekehrt	331
4.7.7	Ein Byte-Array in einen String konvertieren	333
4.7.8	Strukturvariablen in Arrays einsetzen	335
5	Weitere Sprachfeatures	339
5.1	Namespaces (Namensräume)	339
5.1.1	Ein kleiner Überblick	339
5.1.2	Die Imports-Anweisung	341
5.1.3	Namespace-Alias	341
5.1.4	Namespaces in Projekteigenschaften	342
5.1.5	Namespace Alias Qualifizierer	343
5.1.6	Eigene Namespaces einrichten	343
5.2	Überladen von Operatoren	344
5.2.1	Syntaxregeln	344
5.2.2	Praktische Anwendung	345
5.2.3	Konvertierungsoperatoren überladen	346
5.3	Auflistungen (Collections)	347
5.3.1	Beziehungen zwischen den Schnittstellen	347
5.3.2	IEnumerable	348
5.3.3	ICollection	349
5.3.4	IList	349
5.3.5	Iteratoren	349
5.3.6	Die ArrayList-Collection	350
5.3.7	Die Hashtable	351
5.4	Generische Datentypen	352
5.4.1	Wie es früher einmal war	352
5.4.2	Typsicherheit durch Generics	354
5.4.3	List-Collection ersetzt ArrayList	355

5.4.4	Über die Vorzüge generischer Collections	356
5.4.5	Typbeschränkungen durch Constraints	357
5.4.6	Collection-Initialisierer	358
5.4.7	Generische Methoden	358
5.5	Delegates	359
5.5.1	Delegates sind Methodenzeiger	359
5.5.2	Delegate-Typ definieren	360
5.5.3	Delegate-Objekt erzeugen	361
5.5.4	Delegates vereinfacht instanziiieren	361
5.5.5	Relaxed Delegates	362
5.5.6	Anonyme Methoden	362
5.5.7	Lambda-Ausdrücke	363
5.5.8	Lambda-Ausdrücke in der Task Parallel Library	364
5.6	Dynamische Programmierung	366
5.6.1	Wozu dynamische Programmierung?	366
5.6.2	Das Prinzip der dynamischen Programmierung	366
5.6.3	Kovarianz und Kontravarianz	370
5.7	Weitere Datentypen	371
5.7.1	BigInteger	371
5.7.2	Complex	373
5.7.3	Tuple(Of T)	374
5.7.4	SortedSet(Of T)	374
5.8	Praxisbeispiele	376
5.8.1	ArrayList versus generische List	376
5.8.2	Delegates und Lambda Expressions	379
5.8.3	Mit dynamischem Objekt eine Datei durchsuchen	382
6	Einführung in LINQ	387
6.1	LINQ-Grundlagen	387
6.1.1	Die LINQ-Architektur	387
6.1.2	LINQ-Implementierungen	388
6.1.3	Anonyme Typen	388
6.1.4	Erweiterungsmethoden	390
6.2	Abfragen mit LINQ to Objects	391
6.2.1	Grundlegendes zur LINQ-Syntax	391
6.2.2	Zwei alternative Schreibweisen von LINQ-Abfragen	392
6.2.3	Übersicht der wichtigsten Abfrage-Operatoren	394

- 6.3 LINQ-Abfragen im Detail 395
 - 6.3.1 Die Projektionsoperatoren Select und SelectMany 396
 - 6.3.2 Der Restriktionsoperator Where 398
 - 6.3.3 Die Sortierungsoperatoren OrderBy und ThenBy 398
 - 6.3.4 Der Gruppierungsoperator GroupBy 400
 - 6.3.5 Verknüpfen mit Join 401
 - 6.3.6 Aggregat-Operatoren 402
 - 6.3.7 Verzögertes Ausführen von LINQ-Abfragen 404
 - 6.3.8 Konvertierungsmethoden 405
 - 6.3.9 Abfragen mit PLINQ 405
- 6.4 Praxisbeispiele 408
 - 6.4.1 Die Syntax von LINQ-Abfragen verstehen 408
 - 6.4.2 Aggregat-Abfragen mit LINQ 411
 - 6.4.3 LINQ im Schnelldurchgang erlernen 413
 - 6.4.4 Strings mit LINQ abfragen und filtern 416
 - 6.4.5 Duplikate aus einer Liste oder einem Array entfernen 417
 - 6.4.6 Arrays mit LINQ initialisieren 420
 - 6.4.7 Arrays per LINQ mit Zufallszahlen füllen 422
 - 6.4.8 Einen String mit Wiederholmuster erzeugen 423
 - 6.4.9 Mit LINQ Zahlen und Strings sortieren 425
 - 6.4.10 Mit LINQ Collections von Objekten sortieren 426
 - 6.4.11 Ergebnisse von LINQ-Abfragen in ein Array kopieren 428

Teil II: Technologien

- 7 Zugriff auf das Dateisystem 431**
 - 7.1 Grundlagen 431
 - 7.1.1 Klassen für Verzeichnis- und Dateioperationen 432
 - 7.1.2 Statische versus Instanzen-Klasse 432
 - 7.2 Übersichten 433
 - 7.2.1 Methoden der Directory-Klasse 433
 - 7.2.2 Methoden eines DirectoryInfo-Objekts 434
 - 7.2.3 Eigenschaften eines DirectoryInfo-Objekts 434
 - 7.2.4 Methoden der File-Klasse 434
 - 7.2.5 Methoden eines FileInfo-Objekts 435
 - 7.2.6 Eigenschaften eines FileInfo-Objekts 436

7.3	Operationen auf Verzeichnisebene	436
7.3.1	Existenz eines Verzeichnisses/einer Datei feststellen	436
7.3.2	Verzeichnisse erzeugen und löschen	437
7.3.3	Verzeichnisse verschieben und umbenennen	438
7.3.4	Aktuelles Verzeichnis bestimmen	438
7.3.5	Unterverzeichnisse ermitteln	438
7.3.6	Alle Laufwerke ermitteln	439
7.3.7	Dateien kopieren und verschieben	440
7.3.8	Dateien umbenennen	441
7.3.9	Dateiattribute feststellen	441
7.3.10	Verzeichnis einer Datei ermitteln	443
7.3.11	Alle im Verzeichnis enthaltene Dateien ermitteln	443
7.3.12	Dateien und Unterverzeichnisse ermitteln	443
7.4	Zugriffsberechtigungen	444
7.4.1	ACL und ACE	444
7.4.2	SetAccessControl-Methode	445
7.4.3	Zugriffsrechte anzeigen	445
7.5	Weitere wichtige Klassen	446
7.5.1	Die Path-Klasse	446
7.5.2	Die Klasse FileSystemWatcher	447
7.6	Datei- und Verzeichnisdialoge	449
7.6.1	OpenFileDialog und SaveFileDialog	449
7.6.2	FolderBrowserDialog	451
7.7	Praxisbeispiele	452
7.7.1	Infos über Verzeichnisse und Dateien gewinnen	452
7.7.2	Die Verzeichnisstruktur in eine TreeView einlesen	455
7.7.3	Mit LINQ und RegEx Verzeichnisbäume durchsuchen	457
8	Dateien lesen und schreiben	463
8.1	Grundprinzip der Datenpersistenz	463
8.1.1	Dateien und Streams	463
8.1.2	Die wichtigsten Klassen	464
8.1.3	Erzeugen eines Streams	465
8.2	Dateiparameter	465
8.2.1	FileAccess	465
8.2.2	FileMode	465
8.2.3	FileShare	466

8.3	Textdateien	466
8.3.1	Eine Textdatei beschreiben bzw. neu anlegen	466
8.3.2	Eine Textdatei lesen	468
8.4	Binärdateien	470
8.4.1	Lese-/Schreibzugriff	470
8.4.2	Die Methoden ReadAllBytes und WriteAllBytes	470
8.4.3	BinaryReader/BinaryWriter erzeugen	471
8.5	Sequenzielle Dateien	471
8.5.1	Lesen und Schreiben von strukturierten Daten	471
8.5.2	Serialisieren von Objekten	472
8.6	Dateien verschlüsseln und komprimieren	473
8.6.1	Das Methodenpärchen Encrypt-/Decrypt	474
8.6.2	Verschlüsseln unter Windows Vista/7/8/10	474
8.6.3	Verschlüsseln mit der CryptoStream-Klasse	475
8.6.4	Dateien komprimieren	476
8.7	Memory Mapped Files	477
8.7.1	Grundprinzip	477
8.7.2	Erzeugen eines MMF	478
8.7.3	Erstellen eines Map View	478
8.8	Praxisbeispiele	479
8.8.1	Auf eine Textdatei zugreifen	479
8.8.2	Einen Objektbaum speichern	483
8.8.3	Ein Memory Mapped File (MMF) verwenden	490
8.8.4	Hex-Dezimal-Bytes-Konverter	492
8.8.5	Eine Datei verschlüsseln	496
8.8.6	Eine Datei komprimieren	499
8.8.7	Echte ZIP-Dateien erstellen	501
8.8.8	PDFs erstellen/exportieren	502
8.8.9	Eine CSV-Datei erstellen	506
8.8.10	Eine CSV-Datei mit LINQ lesen und auswerten	509
8.8.11	Einen korrekten Dateinamen erzeugen	511
9	Asynchrone Programmierung	513
9.1	Übersicht	513
9.1.1	Multitasking versus Multithreading	514
9.1.2	Deadlocks	515
9.1.3	Racing	515

9.2	Programmieren mit Threads	517
9.2.1	Einführungsbeispiel	517
9.2.2	Wichtige Thread-Methoden	518
9.2.3	Wichtige Thread-Eigenschaften	520
9.2.4	Einsatz der ThreadPool-Klasse	521
9.3	Sperrmechanismen	523
9.3.1	Threading ohne SyncLock	523
9.3.2	Threading mit SyncLock	524
9.3.3	Die Monitor-Klasse	527
9.3.4	Mutex	530
9.3.5	Methoden für die parallele Ausführung sperren	531
9.3.6	Semaphore	532
9.4	Interaktion mit der Programmoberfläche	533
9.4.1	Die Werkzeuge	534
9.4.2	Einzelne Steuerelemente mit Invoke aktualisieren	534
9.4.3	Mehrere Steuerelemente aktualisieren	535
9.4.4	Ist ein Invoke-Aufruf nötig?	536
9.4.5	Und was ist mit WPF?	536
9.5	Timer-Threads	538
9.6	Die BackgroundWorker-Komponente	539
9.7	Asynchrone Programmier-Entwurfsmuster	542
9.7.1	Kurzübersicht	542
9.7.2	Polling	543
9.7.3	Callback verwenden	544
9.7.4	Callback mit Parameterübergabe verwenden	545
9.7.5	Callback mit Zugriff auf die Programm-Oberfläche	546
9.8	Asynchroner Aufruf beliebiger Methoden	547
9.8.1	Die Beispielklasse	547
9.8.2	Asynchroner Aufruf ohne Callback	549
9.8.3	Asynchroner Aufruf mit Callback und Anzeigefunktion	549
9.8.4	Aufruf mit Rückgabewerten (per Eigenschaft)	550
9.8.5	Aufruf mit Rückgabewerten (per EndInvoke)	551
9.9	Es geht auch einfacher – Async und Await	552
9.9.1	Der Weg von synchron zu asynchron	552
9.9.2	Achtung: Fehlerquellen!	554
9.9.3	Eigene asynchrone Methoden entwickeln	556

9.10	Praxisbeispiele	558
9.10.1	Spieltrieb & Multithreading erleben	558
9.10.2	Prozess- und Thread-Informationen gewinnen	570
9.10.3	Ein externes Programm starten	575
10	Die Task Parallel Library	579
10.1	Überblick	579
10.1.1	Parallel-Programmierung	579
10.1.2	Möglichkeiten der TPL	582
10.1.3	Der CLR-Threadpool	582
10.2	Parallele Verarbeitung mit Parallel.Invoke	583
10.2.1	Aufrufvarianten	584
10.2.2	Einschränkungen	585
10.3	Verwendung von Parallel.For	585
10.3.1	Abbrechen der Verarbeitung	587
10.3.2	Auswerten des Verarbeitungsstatus	588
10.3.3	Und was ist mit anderen Iterator-Schrittweiten?	588
10.4	Collections mit Parallel.ForEach verarbeiten	589
10.5	Die Task-Klasse	590
10.5.1	Einen Task erzeugen	590
10.5.2	Task starten	591
10.5.3	Datenübergabe an den Task	592
10.5.4	Wie warte ich auf das Taskende?	593
10.5.5	Tasks mit Rückgabewerten	595
10.5.6	Die Verarbeitung abbrechen	598
10.5.7	Fehlerbehandlung	602
10.5.8	Weitere Eigenschaften	602
10.6	Zugriff auf das Userinterface	604
10.6.1	Task-Ende und Zugriff auf die Oberfläche	604
10.6.2	Zugriff auf das UI aus dem Task heraus	605
10.7	Weitere Datenstrukturen im Überblick	607
10.7.1	Threadsichere Collections	607
10.7.2	Primitive für die Threadsynchroisation	608
10.8	Parallel LINQ (PLINQ)	608
10.9	Praxisbeispiel: Spieltrieb – Version 2	609
10.9.1	Aufgabenstellung	609
10.9.2	Global-Klasse	609
10.9.3	Controller	610

10.9.4	LKWs	612
10.9.5	Schiff-Klasse	613
10.9.6	Oberfläche	615
11	Fehlersuche und Behandlung	617
11.1	Der Debugger	617
11.1.1	Allgemeine Beschreibung	617
11.1.2	Die wichtigsten Fenster	618
11.1.3	Debugging-Optionen	621
11.1.4	Praktisches Debugging am Beispiel	623
11.2	Arbeiten mit Debug und Trace	627
11.2.1	Wichtige Methoden von Debug und Trace	627
11.2.2	Besonderheiten der Trace-Klasse	630
11.2.3	TraceListener-Objekte	631
11.3	Caller Information	634
11.3.1	Attribute	634
11.3.2	Anwendung	634
11.4	Fehlerbehandlung	635
11.4.1	Anweisungen zur Fehlerbehandlung	635
11.4.2	Try-Catch	635
11.4.3	Try-Finally	640
11.4.4	Das Standardverhalten bei Ausnahmen festlegen	642
11.4.5	Die Exception-Klasse	643
11.4.6	Fehler/Ausnahmen auslösen	643
11.4.7	Eigene Fehlerklassen	644
11.4.8	Exceptionhandling zur Entwurfszeit	646
11.4.9	Code Contracts	646
12	XML in Theorie und Praxis	649
12.1	XML – etwas Theorie	649
12.1.1	Übersicht	649
12.1.2	Der XML-Grundaufbau	652
12.1.3	Wohlgeformte Dokumente	653
12.1.4	Processing Instructions (PI)	656
12.1.5	Elemente und Attribute	656
12.1.6	Verwendbare Zeichensätze	658

12.2	XSD-Schemas	660
12.2.1	XSD-Schemas und ADO.NET	660
12.2.2	XML-Schemas in Visual Studio analysieren	662
12.2.3	XML-Datei mit XSD-Schema erzeugen	665
12.2.4	XSD-Schema aus einer XML-Datei erzeugen	666
12.3	XML-Integration in Visual Basic	667
12.3.1	XML-Literale	667
12.3.2	Einfaches Navigieren durch späte Bindung	670
12.3.3	Die LINQ to XML-API	672
12.3.4	Neue XML-Dokumente erzeugen	673
12.3.5	Laden und Sichern von XML-Dokumenten	675
12.3.6	Navigieren in XML-Daten	677
12.3.7	Auswählen und Filtern	679
12.3.8	Manipulieren der XML-Daten	679
12.3.9	XML-Dokumente transformieren	681
12.4	Verwendung des DOM unter .NET	684
12.4.1	Übersicht	684
12.4.2	DOM-Integration in Visual Basic	685
12.4.3	Laden von Dokumenten	685
12.4.4	Erzeugen von XML-Dokumenten	686
12.4.5	Auslesen von XML-Dateien	688
12.4.6	Direktzugriff auf einzelne Elemente	689
12.4.7	Einfügen von Informationen	690
12.4.8	Suchen in den Baumzweigen	692
12.5	Weitere Möglichkeiten der XML-Verarbeitung	696
12.5.1	Die relationale Sicht mit XmlDataDocument	696
12.5.2	XML-Daten aus Objektstrukturen erzeugen	699
12.5.3	Schnelles Suchen in XML-Daten mit XPathNavigator	702
12.5.4	Schnelles Auslesen von XML-Daten mit XmlReader	705
12.5.5	Erzeugen von XML-Daten mit XmlWriter	707
12.5.6	XML transformieren mit XSLT	709
12.6	Praxisbeispiele	711
12.6.1	Mit dem DOM in XML-Dokumenten navigieren	711
12.6.2	XML-Daten in eine TreeView einlesen	714
12.6.3	DataSets in XML-Strings konvertieren	718
12.6.4	In Dokumenten mit dem XPathNavigator navigieren	722

13	Einführung in ADO.NET	727
13.1	Eine kleine Übersicht	727
13.1.1	Die ADO.NET-Klassenhierarchie	727
13.1.2	Die Klassen der Datenprovider	728
13.1.3	Das Zusammenspiel der ADO.NET-Klassen	731
13.2	Das Connection-Objekt	732
13.2.1	Allgemeiner Aufbau	732
13.2.2	OleDbConnection	732
13.2.3	Schließen einer Verbindung	734
13.2.4	Eigenschaften des Connection-Objekts	734
13.2.5	Methoden des Connection-Objekts	736
13.2.6	Der SqlConnectionStringBuilder	737
13.3	Das Command-Objekt	738
13.3.1	Erzeugen und Anwenden eines Command-Objekts	738
13.3.2	Erzeugen mittels CreateCommand-Methode	739
13.3.3	Eigenschaften des Command-Objekts	739
13.3.4	Methoden des Command-Objekts	741
13.3.5	Freigabe von Connection- und Command-Objekten	742
13.4	Parameter-Objekte	744
13.4.1	Erzeugen und Anwenden eines Parameter-Objekts	744
13.4.2	Eigenschaften des Parameter-Objekts	744
13.5	Das SqlCommandBuilder-Objekt	745
13.5.1	Erzeugen	745
13.5.2	Anwenden	746
13.6	Das SqlDataReader-Objekt	746
13.6.1	SqlDataReader erzeugen	747
13.6.2	Daten lesen	747
13.6.3	Eigenschaften des SqlDataReader	748
13.6.4	Methoden des SqlDataReader	748
13.7	Das SqlDataAdapter-Objekt	749
13.7.1	SqlDataAdapter erzeugen	749
13.7.2	Command-Eigenschaften	750
13.7.3	Fill-Methode	751
13.7.4	Update-Methode	752
13.8	Praxisbeispiele	753
13.8.1	Wichtige ADO.NET-Objekte im Einsatz	753
13.8.2	Eine Aktionsabfrage ausführen	755
13.8.3	Eine Auswahlabfrage aufrufen	757

13.8.4	Die Datenbank aktualisieren	759
13.8.5	Den ConnectionString speichern	762
14	Das DataSet	765
14.1	Grundlegende Features des DataSets	765
14.1.1	Die Objekthierarchie	766
14.1.2	Die wichtigsten Klassen	766
14.1.3	Erzeugen eines DataSets	767
14.2	Das DataTable-Objekt	769
14.2.1	DataTable erzeugen	769
14.2.2	Spalten hinzufügen	769
14.2.3	Zeilen zur DataTable hinzufügen	770
14.2.4	Auf den Inhalt einer DataTable zugreifen	771
14.3	Die DataView	773
14.3.1	Erzeugen eines DataView	773
14.3.2	Sortieren und Filtern von Datensätzen	773
14.3.3	Suchen von Datensätzen	774
14.4	Typisierte DataSets	774
14.4.1	Ein typisiertes DataSet erzeugen	775
14.4.2	Das Konzept der Datenquellen	776
14.4.3	Typisierte DataSets und TableAdapter	777
14.5	Die Qual der Wahl	778
14.5.1	DataReader – der schnelle Lesezugriff	779
14.5.2	DataSet – die Datenbank im Hauptspeicher	779
14.5.3	Objektrelationales Mapping – die Zukunft?	780
14.6	Praxisbeispiele	781
14.6.1	Im DataView sortieren und filtern	781
14.6.2	Suche nach Datensätzen	783
14.6.3	Ein DataSet in einen XML-String serialisieren	784
14.6.4	Untypisierte in typisierte DataSets konvertieren	789
14.6.5	Eine LINQ to SQL-Abfrage ausführen	794
15	Verteilen von Anwendungen	799
15.1	ClickOnce-Deployment	800
15.1.1	Übersicht/Einschränkungen	800
15.1.2	Die Vorgehensweise	801
15.1.3	Ort der Veröffentlichung	801
15.1.4	Anwendungsdateien	802

15.1.5	Erforderliche Komponenten	802
15.1.6	Aktualisierungen	803
15.1.7	Veröffentlichungsoptionen	804
15.1.8	Veröffentlichen	805
15.1.9	Verzeichnisstruktur	805
15.1.10	Der Webpublishing-Assistent	807
15.1.11	Neue Versionen erstellen	808
15.2	InstallShield	808
15.2.1	Installation	808
15.2.2	Aktivieren	809
15.2.3	Ein neues Setup-Projekt	809
15.2.4	Finaler Test	817
15.3	Hilfdateien programmieren	817
15.3.1	Der HTML Help Workshop	818
15.3.2	Bedienung am Beispiel	819
15.3.3	Hilfdateien in die VB-Anwendung einbinden	821
15.3.4	Eine alternative Hilfe-IDE verwenden	825
16	Weitere Techniken	827
16.1	Zugriff auf die Zwischenablage	827
16.1.1	Das Clipboard-Objekt	827
16.1.2	Zwischenablage-Funktionen für Textboxen	829
16.2	Arbeiten mit der Registry	829
16.2.1	Allgemeines	830
16.2.2	Registry-Unterstützung in .NET	831
16.3	.NET-Reflection	833
16.3.1	Übersicht	833
16.3.2	Assembly laden	833
16.3.3	Mittels GetType und Type Informationen sammeln	834
16.3.4	Dynamisches Laden von Assemblies	836
16.4	Praxisbeispiele	838
16.4.1	Zugriff auf die Registry	838
16.4.2	Dateiverknüpfungen erzeugen	840
16.4.3	Die Zwischenablage überwachen und anzeigen	842
16.4.4	Die WIA-Library kennenlernen	845
16.4.5	Auf eine Webcam zugreifen	857
16.4.6	Auf den Scanner zugreifen	859
16.4.7	OpenOffice.org Writer per OLE steuern	863

16.4.8	Nutzer und Gruppen des Systems ermitteln	871
16.4.9	Testen, ob Nutzer in einer Gruppe enthalten ist	872
16.4.10	Testen, ob der Nutzer ein Administrator ist	874
16.4.11	Die IP-Adressen des Computers bestimmen	875
16.4.12	Die IP-Adresse über den Hostnamen bestimmen	876
16.4.13	Diverse Systeminformationen ermitteln	877
16.4.14	Sound per MCI aufnehmen	886
16.4.15	Mikrofonpegel anzeigen	889
16.4.16	Pegeldiagramm aufzeichnen	891
16.4.17	Sound-und Video-Dateien per MCI abspielen	895
17	Konsolenanwendungen	903
17.1	Grundaufbau/Konzepte	903
17.1.1	Unser Hauptprogramm – Module1.vb	904
17.1.2	Rückgabe eines Fehlerstatus	905
17.1.3	Parameterübergabe	906
17.1.4	Zugriff auf die Umgebungsvariablen	907
17.2	Die Kommandozentrale: System.Console	908
17.2.1	Eigenschaften	908
17.2.2	Methoden/Ereignisse	909
17.2.3	Textausgaben	910
17.2.4	Farbangaben	911
17.2.5	Tastaturabfragen	912
17.2.6	Arbeiten mit Streamdaten	913
17.3	Praxisbeispiel: Farbige Konsolenanwendung	914

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 Store 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	968
19.5.1	Seitennavigation und Parameterübergabe	968
19.5.2	Die Fensterkopfzeile anpassen	970
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	984
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	996
20.3	Listendarstellungen	998
20.3.1	ComboBox, ListBox	998
20.3.2	ListView	1002
20.3.3	GridView	1004
20.3.4	FlipView	1006
20.4	Sonstige Controls	1008
20.4.1	CaptureElement	1008
20.4.2	MediaElement	1009
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	1020
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.vb	1038
21.1.3	Verweise	1040
21.1.4	App.xaml und App.xaml.vb	1040
21.1.5	Package.appxmanifest	1041
21.1.6	Application1_TemporaryKey.pfx	1046
21.1.7	MainPage.xaml & MainPage.xaml.vb	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	1052
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?	1059
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	1064
21.3.8	Daten in der Cloud ablegen/laden (Roaming)	1066
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	1086
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	1095
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	1112
22.2.4	Zugriff auf die Kontaktliste	1113
22.3	Spezielle Oberflächenelemente	1115
22.3.1	MessageDialog	1115
22.3.2	ContentDialog	1118
22.3.3	Popup-Benachrichtigungen	1120
22.3.4	PopUp/Flyouts	1127
22.3.5	Das PopupMenu einsetzen	1131
22.3.6	Eine AppBar 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	1137
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	1145
22.5.1	Verpacken der App	1145
22.5.2	App-Installation per Skript	1147

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	1151
22.7.1	Ein Verzeichnis auf Änderungen überwachen	1151
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	Einführung	1220
23.1.1	Was kann eine WPF-Anwendung?	1220
23.1.2	Die eXtensible Application Markup Language	1222
23.1.3	Verbinden von XAML und Basic-Code	1227
23.1.4	Zielpattformen	1232
23.1.5	Applikationstypen	1233
23.1.6	Vorteile und Nachteile von WPF-Anwendungen	1234
23.1.7	Weitere Dateien im Überblick	1235
23.2	Alles beginnt mit dem Layout	1238
23.2.1	Allgemeines zum Layout	1238
23.2.2	Positionieren von Steuerelementen	1240
23.2.3	Canvas	1243
23.2.4	StackPanel	1244
23.2.5	DockPanel	1246
23.2.6	WrapPanel	1248
23.2.7	UniformGrid	1248
23.2.8	Grid	1250
23.2.9	ViewBox	1254
23.2.10	TextBlock	1255
23.3	Das WPF-Programm	1258
23.3.1	Die Application-Klasse	1259
23.3.2	Das Startobjekt festlegen	1259
23.3.3	Kommandozeilenparameter verarbeiten	1261

23.3.4	Die Anwendung beenden	1261
23.3.5	Auswerten von Anwendungsereignissen	1262
23.4	Die Window-Klasse	1263
23.4.1	Position und Größe festlegen	1263
23.4.2	Rahmen und Beschriftung	1263
23.4.3	Das Fenster-Icon ändern	1264
23.4.4	Anzeige weiterer Fenster	1264
23.4.5	Transparenz	1264
23.4.6	Abstand zum Inhalt festlegen	1265
23.4.7	Fenster ohne Fokus anzeigen	1266
23.4.8	Ereignisfolge bei Fenstern	1266
23.4.9	Ein paar Worte zur Schriftdarstellung	1267
23.4.10	Ein paar Worte zur Controldarstellung	1269
23.4.11	Wird mein Fenster komplett mit WPF gerendert?	1271
24	Übersicht WPF-Controls	1273
24.1	Allgemeingültige Eigenschaften	1273
24.2	Label	1275
24.3	Button, RepeatButton, ToggleButton	1275
24.3.1	Schaltflächen für modale Dialoge	1276
24.3.2	Schaltflächen mit Grafik	1277
24.4	TextBox, PasswortBox	1278
24.4.1	TextBox	1278
24.4.2	PasswortBox	1280
24.5	CheckBox	1281
24.6	RadioButton	1282
24.7	ListBox, ComboBox	1284
24.7.1	ListBox	1284
24.7.2	ComboBox	1287
24.7.3	Den Content formatieren	1288
24.8	Image	1290
24.8.1	Grafik per XAML zuweisen	1290
24.8.2	Grafik zur Laufzeit zuweisen	1290
24.8.3	Bild aus Datei laden	1291
24.8.4	Die Grafikskalierung beeinflussen	1292
24.9	MediaElement	1293
24.10	Slider, ScrollBar	1296
24.10.1	Slider	1296

24.10.2	ScrollBar	1297
24.11	ScrollView	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	1322
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	FlowDocumentScrollView	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	1339
24.26	InkCanvas	1343
24.26.1	Stift-Parameter definieren	1344
24.26.2	Die Zeichenmodi	1345
24.26.3	Inhalte laden und sichern	1345
24.26.4	Konvertieren in eine Bitmap	1346
24.26.5	Weitere Eigenschaften	1347
24.27	Ellipse, Rectangle, Line und Co.	1347
24.27.1	Ellipse	1347
24.27.2	Rectangle	1348
24.27.3	Line	1348
24.28	Browser	1349
24.29	Ribbon	1351
24.29.1	Allgemeine Grundlagen	1351
24.29.2	Download/Installation	1353
24.29.3	Erste Schritte	1353
24.29.4	Registerkarten und Gruppen	1354
24.29.5	Kontextabhängige Registerkarten	1355
24.29.6	Einfache Beschriftungen	1356
24.29.7	Schaltflächen	1357
24.29.8	Auswahllisten	1358
24.29.9	Optionsauswahl	1361
24.29.10	Texteingaben	1361
24.29.11	ScreenTips	1362
24.29.12	Symbolleiste für den Schnellzugriff	1363
24.29.13	Das RibbonWindow	1363
24.29.14	Menüs	1364
24.29.15	Anwendungsmenü	1366
24.29.16	Alternativen	1369
24.30	Chart	1369
24.31	WindowsFormsHost	1370

25	Wichtige WPF-Techniken	1373
25.1	Eigenschaften	1373
25.1.1	Abhängige Eigenschaften (Dependency Properties)	1373
25.1.2	Angehängte Eigenschaften (Attached Properties)	1374
25.2	Einsatz von Ressourcen	1375
25.2.1	Was sind eigentlich Ressourcen?	1375
25.2.2	Wo können Ressourcen gespeichert werden?	1375
25.2.3	Wie definiere ich eine Ressource?	1377
25.2.4	Statische und dynamische Ressourcen	1378
25.2.5	Wie werden Ressourcen adressiert?	1379
25.2.6	System-Ressourcen einbinden	1380
25.3	Das WPF-Ereignis-Modell	1380
25.3.1	Einführung	1380
25.3.2	Routed Events	1381
25.3.3	Direkte Events	1383
25.4	Verwendung von Commands	1383
25.4.1	Einführung in Commands	1384
25.4.2	Verwendung vordefinierter Commands	1384
25.4.3	Das Ziel des Commands	1386
25.4.4	Vordefinierte Commands	1387
25.4.5	Commands an Ereignismethoden binden	1387
25.4.6	Wie kann ich ein Command per Code auslösen?	1389
25.4.7	Command-Ausführung verhindern	1389
25.5	Das WPF-Style-System	1389
25.5.1	Übersicht	1389
25.5.2	Benannte Styles	1390
25.5.3	Typ-Styles	1392
25.5.4	Styles anpassen und vererben	1393
25.6	Verwenden von Triggern	1395
25.6.1	Eigenschaften-Trigger (Property triggers)	1395
25.6.2	Ereignis-Trigger	1397
25.6.3	Daten-Trigger	1398
25.7	Einsatz von Templates	1399
25.7.1	Template abrufen und verändern	1403
25.8	Transformationen, Animationen, StoryBoards	1406
25.8.1	Transformationen	1406
25.8.2	Animationen mit dem StoryBoard realisieren	1411
25.9	Praxisbeispiel	1415

26	WPF-Datenbindung	1419
26.1	Grundprinzip	1419
26.1.1	Bindungsarten	1420
26.1.2	Wann wird eigentlich die Quelle aktualisiert?	1421
26.1.3	Geht es auch etwas langsamer?	1422
26.1.4	Bindung zur Laufzeit realisieren	1423
26.2	Binden an Objekte	1425
26.2.1	Objekte im Code instanziiieren	1425
26.2.2	Verwenden der Instanz im VB-Quellcode	1427
26.2.3	Anforderungen an die Quell-Klasse	1427
26.2.4	Instanziiieren von Objekten per VB-Code	1429
26.3	Binden von Collections	1430
26.3.1	Anforderung an die Collection	1430
26.3.2	Einfache Anzeige	1431
26.3.3	Navigation zwischen den Objekten	1432
26.3.4	Einfache Anzeige in einer ListBox	1433
26.3.5	DataTemplates zur Anzeigeformatierung	1435
26.3.6	Mehr zu List- und ComboBox	1436
26.3.7	Verwenden der ListView	1438
26.4	Noch einmal zurück zu den Details	1440
26.4.1	Navigieren in den Daten	1440
26.4.2	Sortieren	1441
26.4.3	Filtern	1442
26.4.4	Live Shaping	1443
26.5	Anzeige von Datenbankinhalten	1444
26.5.1	Datenmodell per LINQ to SQL-Designer erzeugen	1444
26.5.2	Die Programm-Oberfläche	1445
26.5.3	Der Zugriff auf die Daten	1447
26.6	Drag & Drop-Datenbindung	1448
26.6.1	Vorgehensweise	1448
26.6.2	Weitere Möglichkeiten	1451
26.7	Formatieren von Werten	1452
26.7.1	IValueConverter	1453
26.7.2	BindingBase.StringFormat-Eigenschaft	1455
26.8	Das DataGridView als Universalwerkzeug	1456
26.8.1	Grundlagen der Anzeige	1457
26.8.2	Vom Betrachten zum Editieren	1461

26.9	Praxisbeispiele	1461
26.9.1	Collections in Hintergrundthreads füllen	1461
26.9.2	Drag & Drop-Bindung bei 1:n-Beziehungen	1465
27	Druckausgabe mit WPF	1469
27.1	Grundlagen	1469
27.1.1	XPS-Dokumente	1469
27.1.2	System.Printing	1470
27.1.3	System.Windows.Xps	1471
27.2	Einfache Druckausgaben mit dem PrintDialog	1471
27.3	Mehrseitige Druckvorschau-Funktion	1474
27.3.1	Fix-Dokumente	1474
27.3.2	Flow-Dokumente	1480
27.4	Druckerinfos, -auswahl, -konfiguration	1483
27.4.1	Die installierten Drucker bestimmen	1484
27.4.2	Den Standarddrucker bestimmen	1485
27.4.3	Mehr über einzelne Drucker erfahren	1485
27.4.4	Spezifische Druckeinstellungen vornehmen	1487
27.4.5	Direkte Druckausgabe	1489

Teil V: Windows Forms

28	Windows Forms-Anwendungen	1493
28.1	Grundaufbau/Konzepte	1493
28.1.1	Wo ist das Hauptprogramm?	1494
28.1.2	Die Oberflächendefinition – Form1.Designer.vb	1499
28.1.3	Die Spielwiese des Programmierers – Form1.vb	1500
28.1.4	Die Datei AssemblyInfo.vb	1501
28.1.5	Resources.resx/Resources.Designer.vb	1502
28.1.6	Settings.settings/Settings.Designer.vb	1503
28.2	Ein Blick auf die Application-Klasse	1505
28.2.1	Eigenschaften	1505
28.2.2	Methoden	1506
28.2.3	Ereignisse	1507
28.3	Allgemeine Eigenschaften von Komponenten	1508
28.3.1	Font	1509
28.3.2	Handle	1510

28.3.3	Tag	1511
28.3.4	Modifiers	1511
28.4	Allgemeine Ereignisse von Komponenten	1512
28.4.1	Die Eventhandler-Argumente	1512
28.4.2	Sender	1512
28.4.3	Der Parameter e	1514
28.4.4	Mausereignisse	1514
28.4.5	KeyPreview	1516
28.4.6	Weitere Ereignisse	1517
28.4.7	Validierung	1518
28.4.8	SendKeys	1518
28.5	Allgemeine Methoden von Komponenten	1519
29	Windows Forms-Formulare	1521
29.1	Übersicht	1521
29.1.1	Wichtige Eigenschaften des Form-Objekts	1522
29.1.2	Wichtige Ereignisse des Form-Objekts	1524
29.1.3	Wichtige Methoden des Form-Objekts	1525
29.2	Praktische Aufgabenstellungen	1526
29.2.1	Fenster anzeigen	1526
29.2.2	Splash Screens beim Anwendungsstart anzeigen	1529
29.2.3	Eine Sicherheitsabfrage vor dem Schließen anzeigen	1531
29.2.4	Ein Formular durchsichtig machen	1532
29.2.5	Die Tabulatorreihenfolge festlegen	1532
29.2.6	Ausrichten und Platzieren von Komponenten	1533
29.2.7	Spezielle Panels für flexibles Layout	1536
29.2.8	Menüs erzeugen	1537
29.3	MDI-Anwendungen	1541
29.3.1	"Falsche" MDI-Fenster bzw. Verwenden von Parent	1541
29.3.2	Die echten MDI-Fenster	1542
29.3.3	Die Kindfenster	1543
29.3.4	Automatisches Anordnen der Kindfenster	1544
29.3.5	Zugriff auf die geöffneten MDI-Kindfenster	1546
29.3.6	Zugriff auf das aktive MDI-Kindfenster	1546
29.3.7	Mischen von Kindfenstermenü/MDIContainer-Menü	1546
29.4	Praxisbeispiele	1547
29.4.1	Informationsaustausch zwischen Formularen	1547
29.4.2	Ereigniskette beim Laden/Entladen eines Formulars	1555

30	Komponenten-Übersicht	1561
30.1	Allgemeine Hinweise	1561
30.1.1	Hinzufügen von Komponenten	1561
30.1.2	Komponenten zur Laufzeit per Code erzeugen	1562
30.2	Allgemeine Steuerelemente	1564
30.2.1	Label	1564
30.2.2	LinkLabel	1565
30.2.3	Button	1566
30.2.4	TextBox	1567
30.2.5	MaskedTextBox	1570
30.2.6	CheckBox	1571
30.2.7	RadioButton	1573
30.2.8	ListBox	1574
30.2.9	CheckedListBox	1575
30.2.10	ComboBox	1576
30.2.11	PictureBox	1577
30.2.12	DateTimePicker	1577
30.2.13	MonthCalendar	1578
30.2.14	HScrollBar, VScrollBar	1578
30.2.15	TrackBar	1579
30.2.16	NumericUpDown	1580
30.2.17	DomainUpDown	1581
30.2.18	ProgressBar	1581
30.2.19	RichTextBox	1582
30.2.20	ListView	1583
30.2.21	TreeView	1589
30.2.22	WebBrowser	1594
30.3	Container	1595
30.3.1	FlowLayout/TableLayout/SplitContainer	1595
30.3.2	Panel	1595
30.3.3	GroupBox	1596
30.3.4	TabControl	1597
30.3.5	ImageList	1599
30.4	Menüs & Symbolleisten	1600
30.4.1	MenuStrip und ContextMenuStrip	1600
30.4.2	ToolStrip	1600
30.4.3	StatusStrip	1600
30.4.4	ToolStripContainer	1601

30.5	Daten	1601
30.5.1	DataSet	1601
30.5.2	DataGridView/DataGrid	1602
30.5.3	BindingNavigator/BindingSource	1602
30.5.4	Chart	1602
30.6	Komponenten	1603
30.6.1	ErrorProvider	1603
30.6.2	HelpProvider	1604
30.6.3	ToolTip	1604
30.6.4	Timer	1604
30.6.5	BackgroundWorker	1604
30.6.6	SerialPort	1604
30.7	Drucken	1605
30.7.1	PrintPreviewControl	1605
30.7.2	PrintDocument	1605
30.8	Dialoge	1605
30.8.1	OpenFileDialog/SaveFileDialog/FolderBrowserDialog	1605
30.8.2	FontDialog/ColorDialog	1605
30.9	WPF-Unterstützung mit dem ElementHost	1605
30.10	Praxisbeispiele	1606
30.10.1	Mit der CheckBox arbeiten	1606
30.10.2	Steuerelemente per Code selbst erzeugen	1607
30.10.3	Controls-Auflistung im TreeView anzeigen	1610
30.10.4	WPF-Komponenten mit dem ElementHost anzeigen	1613
31	Grundlagen der Grafikausgabe	1617
31.1	Übersicht und erste Schritte	1617
31.1.1	GDI+ – Ein erster Blick für Umsteiger	1618
31.1.2	Namespaces für die Grafikausgabe	1619
31.2	Darstellen von Grafiken	1621
31.2.1	Die PictureBox-Komponente	1621
31.2.2	Das Image-Objekt	1622
31.2.3	Laden von Grafiken zur Laufzeit	1623
31.2.4	Sichern von Grafiken	1623
31.2.5	Grafikeigenschaften ermitteln	1624
31.2.6	Erzeugen von Vorschaugrafiken (Thumbnails)	1625
31.2.7	Die Methode RotateFlip	1626
31.2.8	Skalieren von Grafiken	1627

31.3	Das .NET-Koordinatensystem	1628
31.3.1	Globale Koordinaten	1629
31.3.2	Seitenkoordinaten (globale Transformation)	1630
31.3.3	Gerätekoordinaten (Seitentransformation)	1632
31.4	Grundlegende Zeichenfunktionen von GDI+	1633
31.4.1	Das zentrale Graphics-Objekt	1633
31.4.2	Punkte zeichnen/abfragen	1636
31.4.3	Linien	1637
31.4.4	Kantenglättung mit Antialiasing	1638
31.4.5	PolyLine	1639
31.4.6	Rechtecke	1639
31.4.7	Polygone	1641
31.4.8	Splines	1641
31.4.9	Bézierkurven	1643
31.4.10	Kreise und Ellipsen	1644
31.4.11	Tortestück (Segment)	1644
31.4.12	Bogenstück	1646
31.4.13	Wo sind die Rechtecke mit den "runden Ecken"?	1646
31.4.14	Textausgabe	1648
31.4.15	Ausgabe von Grafiken	1652
31.5	Unser Werkzeugkasten	1653
31.5.1	Einfache Objekte	1653
31.5.2	Vordefinierte Objekte	1654
31.5.3	Farben/Transparenz	1656
31.5.4	Stifte (Pen)	1658
31.5.5	Pinsel (Brush)	1661
31.5.6	SolidBrush	1661
31.5.7	HatchBrush	1661
31.5.8	TextureBrush	1663
31.5.9	LinearGradientBrush	1663
31.5.10	PathGradientBrush	1665
31.5.11	Fonts	1666
31.5.12	Path-Objekt	1667
31.5.13	Clipping/Region	1670
31.6	Standarddialoge	1673
31.6.1	Schriftauswahl	1673
31.6.2	Farbauswahl	1674

31.7	Praxisbeispiele	1676
31.7.1	Ein Graphics-Objekt erzeugen	1676
31.7.2	Zeichenoperationen mit der Maus realisieren	1679
32	Druckausgabe	1683
32.1	Einstieg und Übersicht	1683
32.1.1	Nichts geht über ein Beispiel	1683
32.1.2	Programmiermodell	1685
32.1.3	Kurzübersicht der Objekte	1686
32.2	Auswerten der Druckereinstellungen	1686
32.2.1	Die vorhandenen Drucker	1686
32.2.2	Der Standarddrucker	1687
32.2.3	Verfügbare Papierformate/Seitenabmessungen	1687
32.2.4	Der eigentliche Druckbereich	1689
32.2.5	Die Seitenausrichtung ermitteln	1689
32.2.6	Ermitteln der Farbfähigkeit	1690
32.2.7	Die Druckauflösung abfragen	1690
32.2.8	Ist beidseitiger Druck möglich?	1690
32.2.9	Einen "Informationsgerätekontext" erzeugen	1691
32.2.10	Abfragen von Werten während des Drucks	1692
32.3	Festlegen von Druckereinstellungen	1693
32.3.1	Einen Drucker auswählen	1693
32.3.2	Drucken in Millimetern	1693
32.3.3	Festlegen der Seitenränder	1694
32.3.4	Druckjobname	1695
32.3.5	Die Anzahl der Kopien festlegen	1695
32.3.6	Beidseitiger Druck	1696
32.3.7	Seitenzahlen festlegen	1697
32.3.8	Druckqualität verändern	1700
32.3.9	Ausgabemöglichkeiten des Chart-Controls nutzen	1700
32.4	Die Druckdialoge verwenden	1701
32.4.1	PrintDialog	1701
32.4.2	PageSetupDialog	1702
32.4.3	PrintPreviewDialog	1704
32.4.4	Ein eigenes Druckvorschau-Fenster realisieren	1705
32.5	Drucken mit OLE-Automation	1706
32.5.1	Kurzeinstieg in die OLE-Automation	1706
32.5.2	Drucken mit Microsoft Word	1709

32.6	Praxisbeispiele	1712
32.6.1	Den Drucker umfassend konfigurieren	1712
32.6.2	Diagramme mit dem Chart-Control drucken	1721
32.6.3	Drucken mit Word	1723
33	Windows Forms-Datenbindung	1729
33.1	Prinzipielle Möglichkeiten	1729
33.2	Manuelle Bindung an einfache Datenfelder	1729
33.2.1	BindingSource erzeugen	1730
33.2.2	Binding-Objekt	1730
33.2.3	DataBindings-Collection	1731
33.3	Manuelle Bindung an Listen und Tabellen	1731
33.3.1	DataGridView	1731
33.3.2	Datenbindung von ComboBox und ListBox	1732
33.4	Entwurfszeit-Bindung an typisierte DataSets	1732
33.5	Drag & Drop-Datenbindung	1734
33.6	Navigations- und Bearbeitungsfunktionen	1734
33.6.1	Navigieren zwischen den Datensätzen	1734
33.6.2	Hinzufügen und Löschen	1734
33.6.3	Aktualisieren und Abbrechen	1735
33.6.4	Verwendung des BindingNavigators	1735
33.7	Die Anzeigedaten formatieren	1736
33.8	Praxisbeispiele	1736
33.8.1	Einrichten und Verwenden einer Datenquelle	1736
33.8.2	Eine Auswahlabfrage im DataGridView anzeigen	1740
33.8.3	Master-Detailbeziehungen im DataGrid anzeigen	1743
33.8.4	Datenbindung Chart-Control	1744
34	Erweiterte Grafikausgabe	1749
34.1	Transformieren mit der Matrix-Klasse	1749
34.1.1	Übersicht	1749
34.1.2	Translation	1750
34.1.3	Skalierung	1750
34.1.4	Rotation	1751
34.1.5	Scherung	1751
34.1.6	Zuweisen der Matrix	1752
34.2	Low-Level-Grafikmanipulationen	1752
34.2.1	Worauf zeigt Scan0?	1753

34.2.2	Anzahl der Spalten bestimmen	1754
34.2.3	Anzahl der Zeilen bestimmen	1755
34.2.4	Zugriff im Detail (erster Versuch)	1755
34.2.5	Zugriff im Detail (zweiter Versuch)	1757
34.2.6	Invertieren	1759
34.2.7	In Graustufen umwandeln	1760
34.2.8	Heller/Dunkler	1761
34.2.9	Kontrast	1762
34.2.10	Gamma-Wert	1763
34.2.11	Histogramm spreizen	1764
34.2.12	Ein universeller Grafikfilter	1766
34.3	Fortgeschrittene Techniken	1770
34.3.1	Flackerfrei dank Double Buffering	1770
34.3.2	Animationen	1772
34.3.3	Animated GIFs	1775
34.3.4	Auf einzelne GIF-Frames zugreifen	1778
34.3.5	Transparenz realisieren	1779
34.3.6	Eine Grafik maskieren	1781
34.3.7	JPEG-Qualität beim Sichern bestimmen	1782
34.4	Grundlagen der 3D-Vektorgrafik	1783
34.4.1	Datentypen für die Verwaltung	1784
34.4.2	Eine universelle 3D-Grafik-Klasse	1785
34.4.3	Grundlegende Betrachtungen	1786
34.4.4	Translation	1789
34.4.5	Streckung/Skalierung	1789
34.4.6	Rotation	1790
34.4.7	Die eigentlichen Zeichenroutinen	1792
34.5	Und doch wieder GDI-Funktionen	1795
34.5.1	Am Anfang war das Handle	1795
34.5.2	Gerätekontext (Device Context Types)	1797
34.5.3	Koordinatensysteme und Abbildungsmodi	1799
34.5.4	Zeichenwerkzeuge/Objekte	1804
34.5.5	Bitmaps	1806
34.6	Praxisbeispiele	1810
34.6.1	Die Transformationsmatrix verstehen	1810
34.6.2	Eine 3D-Grafikausgabe in Aktion	1813
34.6.3	Einen Fenster-Screenshot erzeugen	1816

35	Ressourcen/Lokalisierung	1819
35.1	Manifestressourcen	1819
35.1.1	Erstellen von Manifestressourcen	1819
35.1.2	Zugriff auf Manifestressourcen	1821
35.2	Typisierte Ressourcen	1822
35.2.1	Erzeugen von .resources-Dateien	1822
35.2.2	Hinzufügen der .resources-Datei zum Projekt	1823
35.2.3	Zugriff auf die Inhalte von .resources-Dateien	1823
35.2.4	ResourceManager direkt aus der .resources-Datei erzeugen	1824
35.2.5	Was sind .resx-Dateien?	1825
35.3	Streng typisierte Ressourcen	1825
35.3.1	Erzeugen streng typisierter Ressourcen	1825
35.3.2	Verwenden streng typisierter Ressourcen	1826
35.3.3	Streng typisierte Ressourcen per Reflection auslesen	1826
35.4	Anwendungen lokalisieren	1828
35.4.1	Localizable und Language	1829
35.4.2	Beispiel "Landesfahnen"	1829
35.4.3	Einstellen der aktuellen Kultur zur Laufzeit	1832
35.5	Praxisbeispiel	1833
35.5.1	Betrachter für Manifestressourcen	1833
36	Komponentenentwicklung	1837
36.1	Überblick	1837
36.2	Benutzersteuerelement	1838
36.2.1	Entwickeln einer Auswahl-ListBox	1838
36.2.2	Komponente verwenden	1840
36.3	Benutzerdefiniertes Steuerelement	1841
36.3.1	Entwickeln eines BlinkLabels	1841
36.3.2	Verwenden der Komponente	1843
36.4	Komponentenklasse	1843
36.5	Eigenschaften	1844
36.5.1	Einfache Eigenschaften	1844
36.5.2	Schreib-/Lesezugriff (Get/Set)	1845
36.5.3	Nur Lese-Eigenschaft (ReadOnly)	1845
36.5.4	Nur-Schreibzugriff (WriteOnly)	1846
36.5.5	Hinzufügen von Beschreibungen	1846
36.5.6	Ausblenden im Eigenschaftenfenster	1847
36.5.7	Einfügen in Kategorien	1847

36.5.8	Default-Wert einstellen	1848
36.5.9	Standard-Eigenschaft	1849
36.5.10	Wertebereichsbeschränkung und Fehlerprüfung	1849
36.5.11	Eigenschaften von Aufzählungstypen	1851
36.5.12	Standard Objekt-Eigenschaften	1852
36.5.13	Eigene Objekt-Eigenschaften	1853
36.6	Methoden	1855
36.6.1	Konstruktor	1855
36.6.2	Class-Konstruktor	1857
36.6.3	Destruktor	1858
36.6.4	Aufruf des Basisklassen-Konstruktors	1858
36.6.5	Aufruf von Basisklassen-Methoden	1859
36.7	Ereignisse (Events)	1859
36.7.1	Ereignis mit Standardargument definieren	1859
36.7.2	Ereignis mit eigenen Argumenten	1860
36.7.3	Ein Default-Ereignis festlegen	1861
36.7.4	Mit Ereignissen auf Windows-Messages reagieren	1862
36.8	Weitere Themen	1863
36.8.1	Wohin mit der Komponente?	1863
36.8.2	Assembly-Informationen festlegen	1865
36.8.3	Assemblies signieren	1866
36.8.4	Komponenten-Ressourcen einbetten	1867
36.8.5	Der Komponente ein Icon zuordnen	1867
36.8.6	Den Designmodus erkennen	1868
36.8.7	Komponenten lizenzieren	1869
36.9	Praxisbeispiele	1873
36.9.1	AnimGif für die Anzeige von Animationen	1873
36.9.2	Eine FontComboBox entwickeln	1876
36.9.3	Das PropertyGrid verwenden	1878
	Index	1881



Vorwort

Die Zeit, in der Visual Basic-Programmierer meinten, mit ein paar Klicks auf ein Formular und mit wenigen Zeilen Quellcode eine vollständige Applikation erschaffen zu können, ist zumindest seit Anbruch der .NET-Epoche endgültig vorbei. Vorbei ist aber auch die Zeit, in der mancher C-Programmierer mitleidig auf den VB-Kollegen herabblicken konnte. VB ist seit langem zu einem vollwertigen und professionellen Werkzeug zum Programmieren beliebiger Komponenten für das Microsoft .NET Framework geworden, beginnend bei Windows Forms- über WPF-, ASP.NET-Anwendungen bis hin zu systemnahen Applikationen.

Das vorliegende Buch zu Visual Basic 2015 soll ein faires Angebot sowohl für künftige als auch für fortgeschrittene VB-Programmierer sein. Seine Philosophie knüpft an die vielen anderen Titel an, die wir in den vergangenen siebzehn 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 Visual Basic 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.

Das ist auch der Grund, warum das vorliegende Buch keinen ausgesprochenen Lehrbuchcharakter trägt, sondern mehr ein mit sorgfältig gewählten Beispielen durchsetztes Nachschlagewerk der wichtigsten Elemente der .NET-Programmierung unter Visual Basic 2015 ist.

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 Basic 2015 zu liefern oder all die Informatio-

nen 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 Schneise durch den Urwald der .NET-Programmierung mit Visual Basic 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 VB.NET
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 folgender 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.

- Bei der Fehlermeldung "Der Microsoft.Jet.OLEDB.4.0-Provider ist nicht auf dem lokalen Computer registriert." müssen Sie als Zielpattform für das Projekt x86 wählen, da es sich bei OLEDB um einen 32-Bit-Treiber handelt.
- Um mit den WinRT-Projekten arbeiten zu können, müssen Sie Visual Studio 2015 unter Windows 8 bzw. 10 ausführen.
- 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 Basic 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 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

OOP-Konzepte

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.

Visual Basic erlaubt es Ihnen aber, 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 (Event-Handler) 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 Visual Basic im Überblick.

3.1 Strukturierter versus objektorientierter Entwurf

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.

3.1.1 Was bedeutet 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 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.

Dass VB 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 VB 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. anstatt mit Methoden mit Funktionen und Prozeduren und anstatt mit Klassen mit strukturierten Datentypen (*Structure*) gearbeitet. Tatsächlich können Sie aber mit OOP alles machen, was auch die strukturierte Programmierung erlaubt.

Anstatt globale Variablen in einem Modul zu deklarieren, können Sie statische Klasseigenschaften verwenden.

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

3.1.2 Was heißt objektorientierte Programmierung?

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.

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.

Das Konzept der objektorientierten Programmierung (OOP) überwindet den prozeduralen Ansatz der klassischen strukturellen Programmierung zugunsten einer realitätsnahen Modellierung.

3.2 Grundbegriffe der OOP

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

3.2.1 Objekt, Klasse, Instanz

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.

Eine *Klasse*¹ 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: In VB haben Sie grundsätzlich die Möglichkeit, zwischen Klassen und Strukturen zu wählen. Letztere wurden bereits im Sprachkapitel (Abschnitt 2.7.2) 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 beschäftigen.

¹ Oft wird anstatt "Klasse" mit völlig gleichwertiger Bedeutung auch der Begriff "Objektyp" (oder auch "Typ") verwendet.

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: Objekt, Klasse, Instanz

Auf dem Montageband werden zahlreiche Auto-Objekte nach ein und demselben 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.

3.2.2 Kapselung und Wiederverwendbarkeit

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.

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.

3.2.3 Vererbung und Polymorphie

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 (auch oft als Superklasse bezeichnet) 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".

Visual Basic unterstützt das Überschreiben (*Overriding*) von Methoden¹ der Oberklasse mit alternativen Methoden der Unterklasse.

¹ Nicht zu verwechseln mit dem Überladen (Overloading) von Methoden.

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: Vererbung

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.

3.2.4 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
<i>Public</i>	Unbeschränkt. Auch von anderen Assemblierungen aus können Objekte der Klasse erstellt werden.
<i>Friend</i>	Nur innerhalb des aktuellen Projekts. Außerhalb des Projekts ist kein Objekt dieser Klasse erstellbar. Gilt als Standard, falls kein Modifizierer vorangestellt wird.
<i>Private</i>	Nur innerhalb einer anderen Klasse.

Klassenmitglieder

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

Modifizierer	Sichtbarkeit
<i>Public</i>	Unbeschränkt.
<i>Friend</i>	Innerhalb der aktuellen Anwendung wie <i>Public</i> , sonst wie <i>Private</i> .
<i>Protected</i>	Wie <i>Private</i> , Mitglieder dürfen aber auch in abgeleiteten Klassen verwendet werden.
<i>Protected Friend</i>	Innerhalb der aktuellen Anwendung wie <i>Protected</i> , sonst wie <i>Private</i> .
<i>Private</i>	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 3.9 (Vererbung und Polymorphie).

3.2.5 Allgemeiner Aufbau einer Klasse

Bevor der Einsteiger seine erste Klasse schreibt, sollte er sich zunächst im einführenden Sprachkapitel mit den Strukturen (siehe Abschnitt 2.7.2) anfreunden, die in Aufbau und Anwendung starke Ähnlichkeiten zu Klassen aufweisen (der wesentliche Unterschied ist, dass Strukturen Wertetypen, Klassen hingegen Referenztypen sind). Auch im Aufbau von Funktionen bzw. Methoden sollte sich der Lernende auskennen (Abschnitt 2.8).

Im Unterschied zu einer Struktur (Schlüsselwort *Structure*) wird eine Klasse mit dem Schlüsselwort *Class* deklariert. Hier die (stark vereinfachte) Syntax:

```
SYNTAX: Modifizierer Class Bezeichner
           ' ... Felder
           ' ... Konstruktoren
           ' ... Eigenschaften
           ' ... Methoden
           ' ... Ereignisse
           End Class
```

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.

```
VB Public Class CKunde
    Private _anrede As String      ' Feld
    Private _name As String        ' dto.

    Public Sub New(anr As String, nam As String) ' Konstruktor
        _anrede = anr
    End Sub
End Class
```

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

```

    _name = nam
End Sub
Public Property name() As String           ' Eigenschaft
    Get
        Return _name
    End Get
    Set(value As String)
        _name = value
    End Set
End Property

Public Function adresse() As String       ' Methode
    Dim s As String = _anrede & " " & _name
    Return s
End Function
End Class

```

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.

3.3 Ein Objekt erzeugen

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 *Nothing* 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 *Nothing*)
- 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.

3.3.1 Referenzieren und Instanzieren

Es stehen zwei Varianten zur Verfügung.

Variante 1 erfordert zwei Schritte:

SYNTAX: *Modifizierer myObject As Klasse*
myObjekt = New Klasse(Parameter)

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

```
VB Private kunde1 As CKunde           ' Referenzieren
    kunde1 = New CKunde1()           ' Erzeugen
```

In *Variante 2*, der Kurzform, sind beide Schritte in einer Anweisung zusammengefasst, d.h., das Objekt wird zusammen mit seiner Deklaration erzeugt.

SYNTAX: *Modifizierer myObject = New Klasse()*

BEISPIEL 3.5: Das Äquivalent zum Vorgängerbeispiel.

```
VB Private kunde1 As 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 (*Imports*-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 Erzeugen¹.

BEISPIEL 3.6: Eine mögliche Fehlerbehandlung

```
VB Private kunde1 As CKunde
    Try
        kunde1 = New CKunde()
    Catch e As Exception
        MessageBox.Show(ex.Message)
    End Try
```

¹ Aus Platz- und Bequemlichkeitsgründen halten sich auch die Autoren nicht immer an diese Empfehlung.

3.3.2 Klassische Initialisierung

Anstatt die Anfangswerte einzeln zuzuweisen, können Sie diese zusammen mit einem Konstruktor übergeben. Zunächst ein Beispiel ohne eigenen Konstruktor, wobei der parameterlose Standardkonstruktor zum Einsatz kommt.

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

```
VB Dim kunde1 As New CKunde()  
kunde1.anrede = "Frau"  
kunde1.name = "Müller"
```

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

```
VB Dim kunde1 As New CKunde("Frau", "Müller")
```

HINWEIS: Weitere Einzelheiten entnehmen Sie dem Abschnitt 3.8.1.

3.3.3 Objekt-Initialisierer

Man kann ein Objekt auch dann erzeugen und seine Eigenschaften (keine privaten Felder!) initialisieren, wenn es dazu keinen Konstruktor gibt.

BEISPIEL 3.9: Das Vorgängerbeispiel mit Objekt-Initialisierer

```
VB Private kunde1 As New CKunde With {.anrede = "Frau", .name = "Müller"}
```

HINWEIS: Mehr zu Objekt-Initialisierern siehe Abschnitt 3.8.2!

3.3.4 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.

```
VB kunde1.Guthaben = 10  
Label1.Text = kunde1.adresse()
```

3.3.5 Zerstören des Objekts

Wenn Sie das Objekt nicht mehr brauchen, können Sie die Objektvariable auf *Nothing* setzen. Vorher können Sie (müssen aber nicht) die *Dispose*-Methode des Objekts aufrufen, vorausgesetzt, Sie haben Sie auch implementiert.

BEISPIEL 3.11: Der *kunde1* wird entfernt.

```
VB kunde1.Dispose()  
kunde1 = Nothing
```

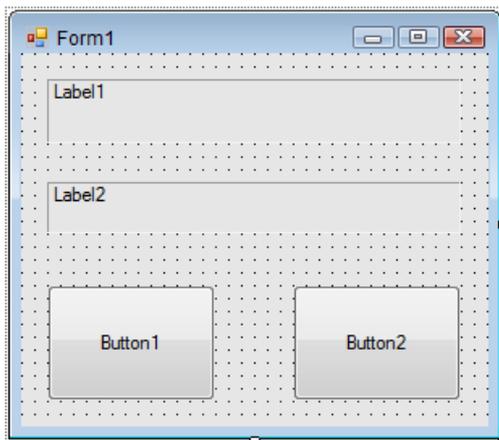
HINWEIS: Das Objekt wird allerdings erst dann zerstört, wenn der Garbage Collector festgestellt hat, dass es nicht länger benötigt wird.

3.4 OOP-Einführungsbeispiel

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

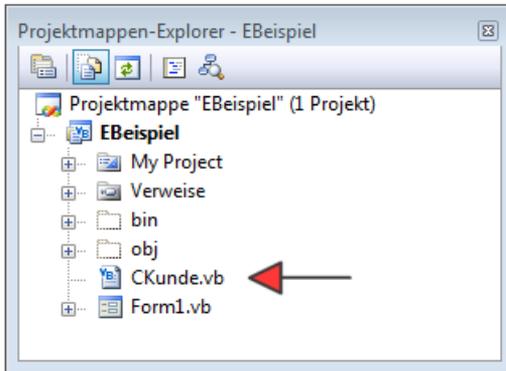
3.4.1 Vorbereitungen

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



- Nachdem Sie den Menüpunkt *Projekt|Klasse hinzufügen...* gewählt haben, geben Sie im Dialogfenster den Namen *CKunde.vb* ein und klicken "Hinzufügen".

Der Projektmappen-Explorer zeigt jetzt die neue Klasse:



HINWEIS: 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.vb*) hinzufügen. Das Verwenden eigener Klassenmodule (idealerweise eins pro Klasse) steigert aber die Übersichtlichkeit des Programmcodes und erleichtert dessen Wiederverwendbarkeit.

3.4.2 Klasse definieren

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

```
Public Class CKunde
    Public Anrede As String          ' einfache Eigenschaften
    Public Name As String            '      dto.
    Public PLZ As Integer            '      dto.
    Public Ort As String             '      dto.
    Public Stammkunde As Boolean     '      dto.
    Public Guthaben As Decimal       '      dto.

    Public Function getAdresse() As String ' erste Methode
        Dim s As String = Anrede & " " & Name & vbCrLf & PLZ.ToString & " " & Ort
        Return s
    End Function

    Public Sub addGuthaben(betrag As Decimal) ' zweite Methode
        If Stammkunde Then Guthaben += betrag
    End Sub
End Class
```

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* (eine Funktion und eine Prozedur). Die Funktion *getAdresse()* liefert als Rückgabewert die komplette Anschrift des Kunden.
- Die Prozedur (*Sub*) *addGuthaben()* hingegen liefert keinen Wert zurück, sie erhöht den Wert des *Guthaben*-Felds bei jedem Aufruf um einen bestimmten Betrag.

3.4.3 Objekt erzeugen und initialisieren

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

Zu Beginn deklarieren Sie eine Objektvariable *kunde1*:

```
Private kunde1 As CKunde          ' Objekt referenzieren
```

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

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    kunde1 = New CKunde()          ' Objekt erzeugen (instanzieren)
    With Kunde1                    ' Objekt initialisieren:
        .Anrede = "Herr"
        .Name = "Müller"
        .PLZ = 12345
        .Ort = "Berlin"
        .Stammkunde = True
    End With
End Sub
```

3.4.4 Objekt verwenden

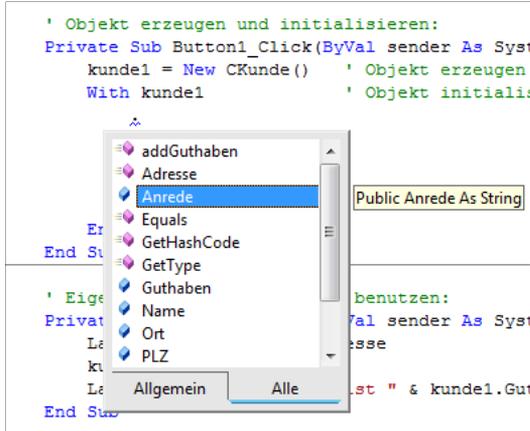
Hinterlegen Sie nun den rechten Button mit der Beschriftung "Eigenschaften und Methoden verwenden" wie folgt:

```
Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
    Label1.Text = Kunde1.getAdresse ' erste Methode aufrufen
    Kunde1.addGuthaben(50D)         ' zweite Methode aufrufen
    Label2.Text = "Guthaben ist " & Kunde1.Guthaben.ToString("C") ' Eigenschaft lesen
End Sub
```

3.4.5 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 doppelt auf den gewünschten Eintrag klicken.

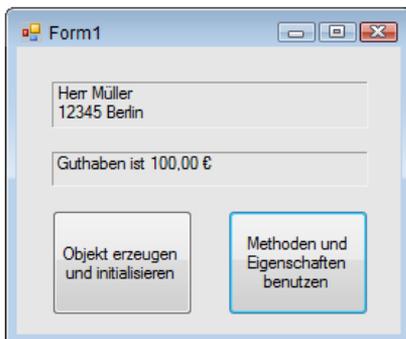


Falls das gewünschte Klassenmitglied nicht erscheint, müssen Sie sofort stutzig werden und es keinesfalls mit dem gewaltsamen Eintippen des Namens versuchen, denn dann gibt es wahrscheinlich einen Fehler beim Kompilieren. Überprüfen Sie stattdessen lieber nochmals die Klassendeklaration, z.B. ob vielleicht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

3.4.6 Objekt testen

Nun ist es endlich so weit, dass Sie Ihr erstes eigenes VB-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.



3.4.7 Warum unsere Klasse noch nicht optimal ist

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 Visual Basic 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.
- Weiterhin war das Initialisieren der Eigenschaften über mehrere Codezeilen ziemlich mühselig (von einem hilfreichen Konstruktor haben wir noch keinerlei Gebrauch gemacht).
- 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, können und wollen wir aber nicht weiter aufschieben und es gleich im folgenden Abschnitt behandeln.

3.5 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.

3.5.1 Eigenschaften 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.

Deklariieren von Eigenschaften

Eigenschaften werden ähnlich wie öffentliche Methoden deklariert. Innerhalb der Deklaration implementieren Sie für den Lesezugriff eine *Get*- und für den Schreibzugriff eine *Set*-Zugriffsmethode. Während die *Get*-Methode ihren Rückgabewert über *Return* liefert, erhält die *Set*-Methode den zu schreibenden Wert über den Parameter *value*.

```
SYNTAX: [Public|Friend|Protected] Property Eigenschaftsname As Type
    Get
        ' hier Lesezugriff (Wert=priv.Felder) implementieren
        Return Wert
    End Get
    Set(value As Type)
        ' hier Schreibzugriff (priv.Felder=value) implementieren
    End Set
End Property
```

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.

```
Public Class CKunde
    Private _anrede As String      ' privates Feld
    Private _name As String       ' dto.
    ...
End Class
```

Der Schreibzugriff auf die Eigenschaft *Anrede* soll so kontrolliert werden, dass nur die Werte "Herr" oder "Frau" zulässig sind. Geben Sie die erste Zeile der *Property*-Deklaration ein, so generiert Visual Studio automatisch den kompletten Rahmencode:

```
Public Property Anrede As String
    Get

    End Get
    Set(value As String)

    End Set
End Property
```

Sie brauchen dann nur noch den Lese- und den Schreibzugriff zu implementieren, sodass die komplette Eigenschaftsdefinition schließlich folgendermaßen aussieht:

```
Public Property Anrede() As String
    Get
        Return _anrede
    End Get
    Set(value As String)
        If (value = "Herr") Or (value = "Frau") Then
            _anrede = value
        Else
        End If
    End Set
End Property
```

```

        MessageBox.Show("Die Anrede '" & value & "' ist nicht zulässig!",
                        "Fehler bei der Eingabe!")
    End If
End Set
End Property

```

Beim Implementieren der Eigenschaft *Name* machen wir es uns etwas einfacher. Hier soll uns die einfache Kapselung genügen (es gibt also keinerlei Zugriffskontrolle):

```

Public Property Name() As String
    Get
        Return _name
    End Get
    Set(value As String)
        _name = value
    End Set
End Property

```

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.

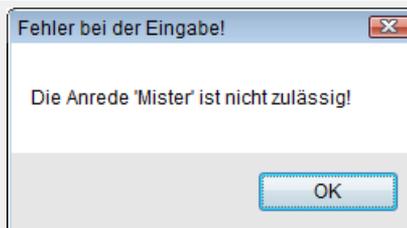
BEISPIEL 3.12: Sie wollen die Anrede "Mister" zuweisen, was zu einem Laufzeitfehler führt.

```

VB Dim kunde1 As New CKunde()
    kunde1.Anrede = "Mister" ' Fehler!

```

Ergebnis



Bemerkung

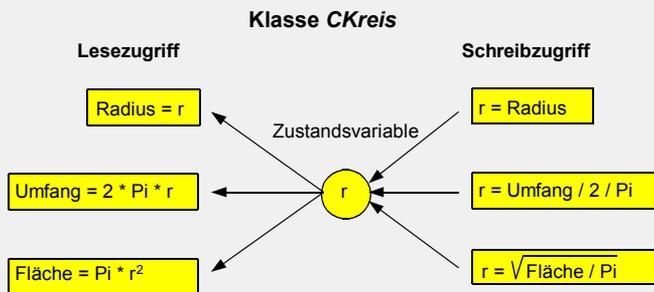
- Beim Schreiben des Quellcodes in der Entwicklungsumgebung Visual Studio merken Sie den "feinen" Unterschied zwischen "einfachen" und "richtigen" Eigenschaften, denn die IntelliSense zeigt dafür unterschiedliche Symbole.
- In unserem Beispiel verhält sich nur die Eigenschaft *Anrede* "intelligent", 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 bei der *Name*-Eigenschaft nur wegen des Lerneffekts betrieben.

3.5.2 Eigenschaften mit Zugriffsmethoden kapseln

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.

BEISPIEL 3.13: Eigenschaften mit Zugriffsmethoden kapseln

VB 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 r As Double ' die einzige Zustandsvariable
```

Die Eigenschaft *Radius*:

```
Public Property Radius() As String
    Get
        Return r.ToString("#,##0.00")
    End Get
    Set(value As String)
        If value <> String.Empty Then
            r = Cdbl(value)
        Else
            r = 0
        End If
    End Set
End Property
```

Die Eigenschaft *Umfang*:

```
Public Property Umfang() As String
    Get
        Return (2 * Math.PI * r).ToString("#,##0.00")
    End Get
    Set(value As String)
```

BEISPIEL 3.13: Eigenschaften mit Zugriffsmethoden kapseln

```

        If value <> String.Empty Then
            r = Cdbl(value) / 2 / Math.PI
        Else
            r = 0
        End If
    End Set
End Property

```

Die Eigenschaft *Fläche*:

```

Public Property Fläche() As String
    Get
        Return (Math.PI * Math.Pow(r, 2)).ToString("#,##0.00")
    End Get
    Set(value As String)
        If value <> String.Empty Then
            r = Math.Sqrt(Cdbl(value) / Math.PI)
        Else
            r = 0
        End If
    End Set
End Property
End Class

```

Das komplette Programm finden Sie unter

► 3.12.1 Eigenschaften sinnvoll kapseln

3.5.3 Lese-/Schreibschutz für Eigenschaften

Es kommt häufig vor, dass bestimmte Eigenschaften nur gelesen oder nur geschrieben werden dürfen (*ReadOnly* bzw. *WriteOnly*). Um diese Art der Zugriffsbeschränkung zu realisieren, ist keinerlei 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 3.14: Lese-/Schreibschutz für Eigenschaften

VB In unserer *CKunde*-Klasse soll das Guthaben für den direkten Schreibzugriff gesperrt werden. Das klingt logisch, da zur Erhöhung des Guthabens bereits die Methode *addGuthaben* existiert.

```

Public Class CKunde
    ...
    Public ReadOnly Property Guthaben() As Decimal
        Get

```

BEISPIEL 3.14: Lese-/Schreibschutz für Eigenschaften

```

        Return _guthaben
    End Get
End Property
...
End Class

```

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

```

        .Ort = "Berlin"
        .Stammkunde = True
        Guthaben = 10
    End
End Sub

```

Die Eigenschaft "Guthaben" ist ReadOnly.

3.5.4 Statische Eigenschaften

Mitunter gibt es 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 Eigenschaften (*Klasseneigenschaften*) werden mit dem Schlüsselwort *Shared* deklariert.

Außer dem *Shared*-Schlüsselwort gibt es beim Deklarieren keine Unterschiede zu den normalen Instanzeigenschaften.

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

BEISPIEL 3.15: Die Klasse *CKunde* soll zusätzlich eine "einfache" Eigenschaft *Rabatt* bekommen, die für jedes Kundenobjekt immer den gleichen Wert hat.

```

VB Public Class CKunde
    ...
    Public Shared Rabatt As Double
    ...
End Class

```

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

BEISPIEL 3.16: Allen Kunden wird ein Rabatt von 15% zugewiesen.

```

VB CKunde.Rabatt = 0.15

```

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

3.5.5 Selbst implementierende Eigenschaften

Bei sehr einfachen Eigenschaften (vergleichbar mit denen, die Sie bislang "unsauber" als *Public*-Felder deklariert haben) können Sie seit VB 2010 so genannte *Auto-implemented Properties* verwenden. Der VB-Compiler generiert im Hintergrund für Sie die entsprechenden *Get*- und *Set*-Zugriffsmethoden und erzeugt außerdem ein privates Feld, um den Wert der Eigenschaft zu speichern.

Wie viel lästige Schreiarbeit Sie sparen können, soll das folgende Beispiel verdeutlichen.

BEISPIEL 3.17: Die folgende Deklaration einer selbst implementierenden Eigenschaft

```
VB Property Ort As String = "München"
    ist äquivalent zu
    Private _Ort As String = "München"           ' backing field
    Property Ort As String
        Get
            Return _Ort
        End Get
        Set(value As String)
            _Ort = value
        End Set
    End Property
```

Wie Sie sehen, kann der Eigenschaft auch ein Standardwert zugewiesen werden. Der Name des automatisch angelegten "backing fields" entspricht dem Namen der Eigenschaft mit vorangestelltem Unterstrich.

HINWEIS: Achten Sie auf Namenskonflikte, die entstehen können, wenn Sie eigene Felder zur Klasse hinzufügen, die auch mit einem Unterstrich () beginnen!

Komplette Eigenschaftsdeklarationen lassen sich in einer einzigen Zeile erledigen, wie es die folgenden Beispiele zeigen.

BEISPIEL 3.18: Einige selbst implementierende Eigenschaften

```
VB Public Property FullName As String
    Public Property FullName As String = "Max Muster"
    Public Property ID As New Guid()
    Public Property ErstesQuartal As New List(Of String) From {"Januar", "Februar", "März"}
```

Seit VB 2015 gibt es auch selbst implementierende ReadOnly-Eigenschaften. Wie bei normalen Auto-Properties können Sie jetzt auch Werte zu ReadOnly-Properties zuweisen und das auch im Konstruktor.

BEISPIEL 3.19: ReadOnly-Eigenschaften

```
VB Class CAutor
    Public ReadOnly Property Tags As New List(Of String)
    Public ReadOnly Property Name As String = ""
    Public ReadOnly Property Datei As String

    Sub New(datei As String)
        Me.Datei = datei
    End Sub
End Class
```

3.6 Methoden

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

3.6.1 Öffentliche und private Methoden

Bereits im Kapitel 2 haben Sie 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 3.20: Die beiden öffentlichen Methoden *getAdresse()* und *addGuthaben()* arbeiten mit sechs privaten Feldern zusammen.

```
VB Public Class CKunde
    Private Variablen:
        Private _anrede As String
        Private _name As String
        Private _plz As Integer
        Private _ort As String
        Private _stammkunde As Boolean
```

BEISPIEL 3.20: Die beiden öffentlichen Methoden *getAdresse()* und *addGuthaben()* arbeiten mit sechs privaten Feldern zusammen.

```
Private _guthaben As Decimal

Öffentliche Methoden:

Public Function getAdresse() As String
    Dim s As String = _anrede & " " & _name & vbCrLf & _plz.ToString & " " & _ort
    Return s
End Function

Public Sub addGuthaben(betrag As Decimal)
    If _stammkunde Then _guthaben += betrag
End Sub

Der Aufruf:

Private kunde1 As New CKunde()
...
Label1.Text = kunde1.getAdresse           ' erste Methode (Funktion) aufrufen
kunde1.addGuthaben(50)                    ' zweite Methode (Prozedur) aufrufen
```

3.6.2 Überladene Methoden

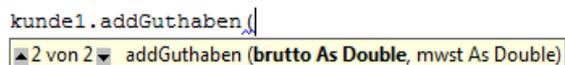
Innerhalb des Klassenkörpers dürfen zwei und mehr gleichnamige Methoden konfliktfrei nebeneinander existieren, wenn sie eine unterschiedliche Signatur (Reihenfolge und Datentyp der Übergabeparameter) besitzen.

BEISPIEL 3.21: Zwei überladene Versionen einer Methode in der Klasse *CKunde*, die erste hat nur den Nettobetrag als Parameter die zweite den Bruttobetrag und die Mehrwertsteuer.

```
VB Public Class CKunde
    ...
    Public Sub addGuthaben(betrag As Decimal)
        If _stammkunde Then _guthaben += betrag
    End Sub

    Public Sub addGuthaben(brutto As Double, mwst As Double)
        If _stammkunde Then _guthaben += CDec(brutto / (1 + mwst))
    End Sub
    ...
End Class
```

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



The screenshot shows a code editor with the text `kunde1.addGuthaben(`. A dropdown menu is open, showing two options: `addGuthaben (betrag As Decimal)` and `addGuthaben (brutto As Double, mwst As Double)`. The second option is highlighted, indicating it is the selected method.

3.6.3 Statische Methoden

Genauso wie *statischen Eigenschaften* können *statische Methoden* (auch als *Klassenmethoden* bezeichnet) ohne Verwendung eines Objekts aufgerufen werden. Statische Methoden werden ebenfalls mit dem *Shared*-Modifizierer gekennzeichnet und 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 instanziierten müssten.

BEISPIEL 3.22: Wir bauen eine Klasse, in der wir wahllos einige von uns häufig benötigte Berechnungsformeln verpacken.

```
VB Public Class MeineFormeln

    Public Shared Function kreisUmfang(radius As Double) As Double
        Return 2 * Math.PI * radius
    End Function

    Public Shared Function kugelVolumen(radius As Double) As Double
        Return 4 / 3 * Math.PI * Math.Pow(radius, 3)
    End Function

    Public Shared Function Netto(brutto As Decimal, mwst As Decimal) As Decimal
        Return brutto / (1 + mwst)
    End Function

    ...

End Class
```

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

HINWEIS: Leider kann bei Klassen die *With*-Anweisung nicht verwendet werden, da diese nur bei Objekten funktioniert.

BEISPIEL 3.23: (Fortsetzung) Die statischen Methoden der Klasse *MeineFormeln* werden in einer Eingabemaske aufgerufen.

```
VB Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    Dim r As Double = Convert.ToDouble(TextBox1.Text) ' Kreisradius konvertieren

    Label1.Text = MeineFormeln.kreisUmfang(r).ToString("0.000")
    Label2.Text = MeineFormeln.kugelVolumen(r).ToString("0.000")
End Sub
```

BEISPIEL 3.23: (Fortsetzung) Die statischen Methoden der Klasse *MeineFormeln* werden in einer Eingabemaske aufgerufen.

```
Dim b As Double = Convert.ToDouble(TextBox2.Text) ' Brutto konvertieren
Label3.Text = MeineFormeln.Netto(b, 0.19).ToString("C")
End Sub
```

Ergebnis

3.7 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.

HINWEIS: Eine ausführliche Einführung in das .NET-Ereignismodell erhalten Sie im Kapitel 26 (Microsoft Event Pattern).

Im Folgenden werden deshalb nur die wichtigsten Grundlagen der Ereignismodellierung erläutert.

3.7.1 Ereignisse deklarieren

Ein Ereignis fügen Sie der Klassendefinition über das *Event*-Schlüsselwort zu.

SYNTAX: `Public Event Ereignisname([Parameterdeklarationen])`

BEISPIEL 3.24: In der Klasse *CKunde* wird ein Ereignis mit dem Namen *GuthabenLeer* deklariert.

```
VB Public Class CKunde
    Private _guthaben As Decimal

    Public Event GuthabenLeer(sender As Object, e As String)

    Public Sub New(betrag As Decimal)      ' Konstruktor
        ...
        _guthaben = betrag
    End Sub
    ...

```

3.7.2 Ereignis auslösen

Ausgelöst wird das Ereignis ebenfalls im Klassenkörper über das Schlüsselwort *RaiseEvent*.

SYNTAX: `RaiseEvent Ereignisname([Parameterwerte])`

BEISPIEL 3.25: (Fortsetzung) Das Ereignis *GuthabenLeer* "feuert" innerhalb der Methode *addGuthaben* dann, wenn das Guthaben den Wert von 10 € unterschreitet.

```
VB Public Sub addGuthaben(betrag As Decimal)
    _guthaben += betrag
    If _guthaben <= 10 Then
        Dim msg As String = "Das Guthaben beträgt nur noch" &
            _guthaben.ToString("C") & "!"
        RaiseEvent GuthabenLeer(Me, msg)
    End If
End Sub
...
End Class

```

Die Ereignisdefinition ist in diesem Beispiel bewusst einfach gehalten, um den Einsteiger nicht zu verschrecken. Normalerweise sollten Ereignisse immer zwei Parameter an die aufrufende Instanz übergeben: eine Referenz auf das Objekt, welches das Ereignis ausgelöst hat, und ein Objekt der *EventArgs*- oder einer davon abgeleiteten Klasse (siehe 20.2.2). Auf Letzteres haben wir der Einfachheit wegen verzichtet und stattdessen einen einfachen Meldungsstring übergeben.

3.7.3 Ereignis auswerten

Um dem Compiler mitzuteilen, dass das Objekt über Ereignisse verfügt, müssen Sie bei der Referenzierung der Objektvariablen vor dem Objektbezeichner das Schlüsselwort *WithEvents* einfügen.

SYNTAX: `Private|Public WithEvents Objektname As Klassenname`

BEISPIEL 3.26: (Fortsetzung) Wir verwenden die im Vorgängerbeispiel definierte Klasse *CKunde* in einem Formular *Form1* mit einem *Button*
VB Public Class Form1

Der Einfachheit wegen erledigen wir die Referenzierung und die Instanziierung der Objektvariablen *kunde1* in einem Schritt und weisen dabei dem Kunden ein Anfangsguthaben von 100 Euro zu.

```
Private WithEvents kunde1 As New CKunde(100)
```

Der Eventhandler für das Ereignis *GuthabenLeer* des Kunden:

```
Private Sub kunde1_GuthabenLeer(sender As Object, e As String) Handles kunde1.GuthabenLeer
    Label1.Text = e.ToString
End Sub
```

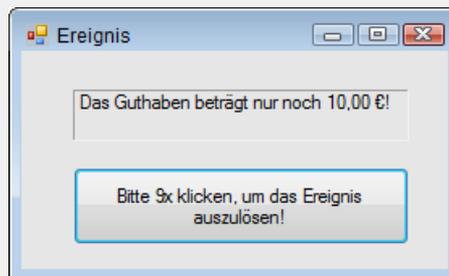
```
End Class
```

Bei jedem Klick auf den *Button* wird das Guthaben des Kunden um 10 € verringert:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    Kunde1.addGuthaben(-10)
End Sub
```

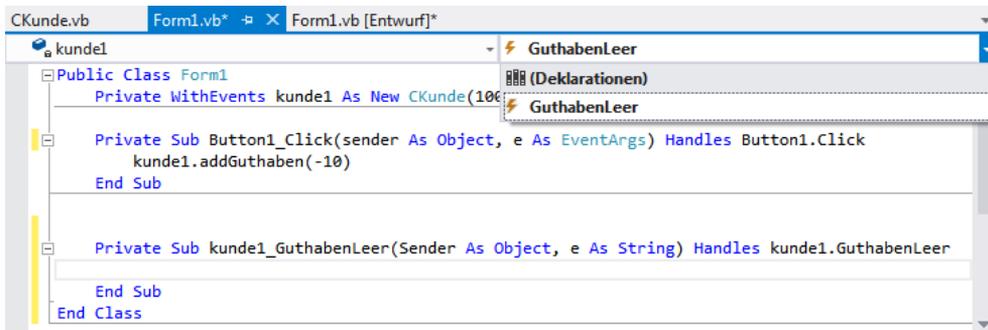
Ergebnis

Nachdem Sie den *Button* neunmal geklickt haben, feuert das Ereignis und im *Label* erscheint eine Meldung:



Visual Studio unterstützt Sie beim Erstellen des Rahmencodes der Event-Handler für selbst definierte Ereignisse natürlich genauso, wie Sie das z.B. für das *Click*-Ereignis eines *Buttons* zur Genüge gewöhnt sind.

Nach dem Deklarieren der Objektvariablen wird in der linken oberen Klappbox das Objekt *kunde1* selektiert und in der rechten das Ereignis *GuthabenLeer*. Der Rahmencode des Eventhandlers wird automatisch generiert:



3.7.4 Benutzerdefinierte Ereignisse (Custom Events)

Meistens deklarieren Sie ein Ereignis mittels *Event*-Schlüsselwort unter Angabe eines Ereignisdelegaten, die Codegenerierung für die Ereignisverwaltung erfolgt automatisch.

BEISPIEL 3.27: Standardmäßige Ereignisdefinition in zwei Schritten

```

VB Public Delegate Sub NumberChangedHandler(i As Integer)
    Public Event NumberChanged As NumberChangedHandler

```

In der Regel ist diese Art der Deklaration auch völlig ausreichend, aber in einigen Fällen möchten Sie die Ereignisverwaltung doch lieber selbst in die Hand nehmen.

Benutzerdefinierte Ereignis-Accessoren (Custom Events) erlauben dem Programmierer die genaue Definition der Vorgänge, die beim Hinzufügen bzw. Entfernen eines Eventhandlers und beim Auslösen eines Events ablaufen sollen (siehe dazu auch die Ausführungen zum Microsoft Event Pattern im Kapitel 26).

Deklaration

Durch Deklarieren eines Custom Events wird dem Compiler mitgeteilt, dass er die Codegenerierung für die Ereignisverwaltung außer Kraft setzen soll, der Programmierer muss sich also nun selbst darum kümmern wie die Ereignishandler an-/abgemeldet und aufgerufen werden.

Ein Custom Event wird durch das der *Event*-Deklaration vorangestellte *Custom*-Schlüsselwort unter Angabe eines Ereignisdelegaten definiert.

BEISPIEL 3.28: Deklaration eines benutzerdefinierten Ereignisses *NumberChanged*

```

VB Public Delegate Sub NumberChangedHandler(i As Integer)
    Public Custom Event NumberChanged As NumberChangedHandler

```

Wenn Sie nach einer solchen Ereignisdeklaration die *Enter*-Taste drücken, erstellt Visual Studio automatisch den Rahmencode für die Ereignis-Acessoren (auf ähnliche Weise wie für Properties):

```
Public Custom Event NumberChanged As NumberChangedHandler
    AddHandler(value As NumberChangedHandler)

    End AddHandler

    RemoveHandler(value As NumberChangedHandler)

    End RemoveHandler

    RaiseEvent(i As Integer)

    End RaiseEvent
End Event
```

Wie Sie sehen, besteht die Deklaration eines Custom Events aus drei Sektionen, die unter folgenden Bedingungen abgearbeitet werden:

- *AddHandler*, wenn ein Ereignishandler entweder mittels *Handles*-Schlüsselwort oder *AddHandler*-Methode hinzugefügt wird.
- *RemoveHandler*, wenn ein Handler mittels *RemoveHandler*-Methode wieder entfernt wird.
- *RaiseEvent*, wenn ein Ereignis ausgelöst wird.

Sie müssen sich jetzt also selbst um das Hinzufügen und Entfernen der Handler zu bzw. aus ihrem Container (üblicherweise eine Collection), sowie um die Benachrichtigung der angeschlossenen Handler beim Auslösen des Ereignisses kümmern. Obwohl das mehr Codezeilen erfordert als eine standardmäßige Implementierung, haben Sie eine größere Flexibilität beim Programmieren.

Anwendung

Da das alles ziemlich verwirrend klingen mag, soll ein Beispiel Licht in die Dunkelheit bringen.

BEISPIEL 3.29: Benutzerdefinierte Ereignisse

VB Eine Klasse *CCounter* erzeugt im Sekundentakt eine Zahl von 1 bis 10 und löst dabei das benutzerdefinierte Ereignis *NumberChanged* aus. Als Container für die angemeldeten Eventhandler dient eine (generische) *List*:

```
Public Class CCounter

    Public Delegate Sub NumberChangedHandler(i As Integer)
    Private handlers As New List(Of NumberChangedHandler)

    Public Custom Event NumberChanged As NumberChangedHandler
        AddHandler(value As NumberChangedHandler)
            If handlers.Count <= 3 Then handlers.Add(value)
        End AddHandler
```

BEISPIEL 3.29: Benutzerdefinierte Ereignisse

```

    RemoveHandler(value As NumberChangedHandler)
        handlers.Remove(value)
    End RemoveHandler

    RaiseEvent(i As Integer)
        If i > 30 Then
            For Each handler As NumberChangedHandler In handlers
                handler.Invoke(i)
            Next
        End If
    End RaiseEvent
End Event

```

Die Methode, in welcher das Ereignis ausgelöst wird:

```

Public Sub DoCount()
    For i As Integer = 1 To 10
        System.Threading.Thread.Sleep(1000)
        RaiseEvent NumberChanged(i * 10)
    Next
End Sub
End Class

```

Ein Eventhandler zeigt die erzeugten Zahlen in einer *ListBox* an:

```

Private Sub c_NumberChanged(i As Integer)
    ListBox1.Items.Add(i.ToString)
End Sub

```

Schließlich der Test der Klasse *CCounter*:

```

Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click

```

Instanziierung:

```

Dim c As New CCounter

```

Eventhandler anmelden:

```

AddHandler c.NumberChanged, AddressOf c_NumberChanged

```

Methode aufrufen, die das Ereignis auslöst:

```

c.DoCount()

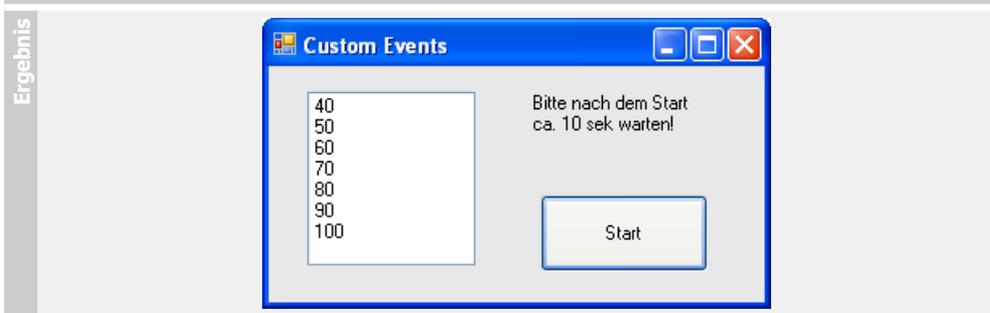
```

Eventhandler abmelden:

```

RemoveHandler c.NumberChanged, AddressOf c_NumberChanged
End Sub

```

BEISPIEL 3.29: Benutzerdefinierte Ereignisse

Wie Sie an diesem Beispiel sehen, ermöglichen Ihnen benutzerdefinierte Ereignisse eine freizügige Programmierung der Verwaltung Ihrer Ereignishandler. Im obigen Code haben wir als Container für die Ereignisdelegaten eine typisierte *List* gewählt, wir hätten aber auch z.B. eine (untypisierte) *ArrayList* nehmen können. In die *AddHandler*-Sektion wurde von uns eine Bedingung eingebaut, die die Anzahl der angeschlossenen Eventhandler auf 3 limitiert. Die *RaiseEvent*-Sektion bewirkt, dass die angeschlossenen Handler nur dann benachrichtigt werden, wenn die übergebene Zahl größer als 30 ist.

Ein nicht zu unterschätzender Vorteil von Custom Events ist auch das Vorhandensein eines zentralen Containers (im Beispiel die *handlers*-Liste), welcher alle angemeldeten Eventhandler innerhalb der Eventdeklaration kapselt.

Die doppelte Verwendung von *RaiseEvent* (innerhalb der *DoCount*-Methode und innerhalb der *Custom Event*-Ereignisdefinition) kann möglicherweise Verwirrung stiften. Deshalb sei hier nochmals auf den grundlegenden Unterschied hingewiesen: *RaiseEvent* in der *DoCount*-Methode löst das *NumberChanged*-Ereignis aus, die *RaiseEvent*-Sektion hingegen spezifiziert den beim Auslösen des Ereignisses auszuführenden "organisatorischen" Code, d.h., alle im Container (*List*) enthaltenen Eventhandler (Delegates) werden durchlaufen und aufgerufen.

3.8 Arbeiten mit Konstruktor und Destruktor

Eine "richtige" objektorientierte Sprache, zu der Visual Basic ja mittlerweile auch gehört, realisiert das Erzeugen und Entfernen von Objekten mit Hilfe von Konstruktoren und Destruktoren.

HINWEIS: Wenn Sie in einigen bisherigen Beispielen in den von Ihnen selbst entwickelten Klassen keinen eigenen Konstruktor definiert hatten, so wurde automatisch der von *System.Object* geerbte parameterlose *New*-Standardkonstruktor verwendet.

3.8.1 Der Konstruktor erzeugt das Objekt

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

HINWEIS: Der Konstruktor ist immer eine *Sub* mit dem Namen *New()*.

SYNTAX: Public Sub **New**([Parameterliste])
 ' Parameter den Feldern der Klasse zuweisen
 ' Initialisierungscode ausführen
 End Sub

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

HINWEIS: 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, d.h., der von *System.Object* geerbte parameterlose Konstruktor steht nicht mehr zur Verfügung!

Deklaration

Einen Konstruktor fügen Sie dem Klassenkörper ähnlich wie eine *Set*-Eigenschaftsprozedur mit dem Namen *New* hinzu. Als Parameter übergeben Sie die Werte für die Felder, die Sie initialisieren möchten.

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

BEISPIEL 3.30: Unserer Klasse *CKunde* werden zwei überladene Konstruktoren hinzugefügt.

VB Public Class CKunde

Die (natürlich privaten) Zustandsvariablen:

```
Private _anrede As String
Private _name As String
Private _plz As Integer
Private _ort As String
Private _stammkunde As Boolean
Private _guthaben As Decimal
```

Der erste Konstruktor initialisiert nur zwei private Variablen:

```
Public Sub New(Anrede As String, Nachname As String)
  _anrede = Anrede
  _name = Nachname
End Sub
```

Der zweite Konstruktor initialisiert alle Variablen der Klasse. Um den Code etwas zu kürzen, wird für die ersten beiden Variablen der erste Konstruktor bemüht:

```
Public Sub New(Anrede As String, Nachname As String, PLZ As Integer,
              Ort As String, Stammkunde As Boolean, Guthaben As Decimal)
  Me.New(Anrede, Nachname)
```

BEISPIEL 3.30: Unserer Klasse *CKunde* werden zwei überladene Konstruktoren hinzugefügt.

```

        _plz = PLZ
        _ort = Ort
        _stammkunde = Stammkunde
        _guthaben = Guthaben
    End Sub
End Class

```

Aufruf

Da wir der Klasse *CKunde* zwei Konstruktoren hinzugefügt haben, ist die bisher gewohnte parameterlose Instanziierung von Objekten mit dem *New()*-Standardkonstruktor nicht mehr möglich (es sei denn, Sie fügen selbst eine weitere Überladung hinzu, die keine Parameter entgegen nimmt)!

BEISPIEL 3.31: (Fortsetzung) Zwei Objekte der oben deklarierten Klasse *CKunde* werden erzeugt und mit Anfangswerten initialisiert. Für jedes Objekt wird ein anderer überladener Konstruktor verwendet.

```

VB Try
    Dim kunde1 As New CKunde("Herr", "Müller")
    Dim kunde2 As New CKunde("Frau", "Hummel", 12345, "Berlin", True, 100)
    ' Dim kunde3 As New CKunde() ' geht nicht mehr!!!
    MessageBox.Show("Objekte erfolgreich erzeugt!")
Catch ex As Exception
    MessageBox.Show("Fehler beim Erzeugen des Objekts!")
End Try

```

Sie sehen, dass das Initialisieren der Objekte viel einfacher geworden ist. Anstatt umständlich eine Eigenschaft nach der anderen zuzuweisen, geht das jetzt mit einer einzigen Anweisung.

3.8.2 Bequemer geht's mit einem Objekt-Initialisierer

Vor allem in Hinblick auf die in der neuen LINQ-Technologie erforderlichen anonymen Typen (siehe Kapitel 6) wurden so genannte Objekt-Initialisierer eingeführt. Damit können nun öffentliche Eigenschaften und Felder von Objekten ohne das explizite Vorhandensein des jeweiligen Konstruktors in beliebiger Reihenfolge initialisiert werden.

Der Objekt-Initialisierer erwartet das Schlüsselwort *With* unmittelbar nach dem Erzeugen des Objekts. Anschließend folgt die in geschweiften Klammern eingeschlossene Liste der zu initialisierenden Mitglieder.

HINWEIS: Einem Objektinitialisierer können nur Eigenschaften oder öffentliche Felder übergeben werden, das Initialisieren privater Felder, wie im obigen Konstruktor-Beispiel, ist nicht möglich!

BEISPIEL 3.32: Erzeugen einer Instanz der Klasse *CPerson*

```
VB Public Class CPerson
    Public Name As String
    Public Strasse As String
    Public PLZ As Integer
    Public Ort As String
End Class
```

Der Objektinitialisierer:

```
Dim person1 As New CPerson With {.Name = "Müller", .Strasse = "Am Waldesrand 7",
    .PLZ = 12345, .Ort = "Musterhausen"}
```

BEISPIEL 3.33: Verschachtelte Objektinitialisierung beim Erzeugen einer Instanz der Klasse *Rectangle*

```
VB Dim rect As New Rectangle With {.Location = New Point With {.X = 3, .Y = 7},
    .Size = New Size With {.Width = 19, .Height = 34} }
```

BEISPIEL 3.34: Initialisieren einer Collection aus Objekten

```
VB Dim personen() = {New CPerson With {.Name = "Müller", .Strasse = "Am Wald 7",
    .PLZ = 12345, .Ort = "Musterhausen"},
    New CPerson With {.Name = "Meier", .Strasse = "Hauptstr. 2",
    .PLZ = 2344, .Ort = "Walldorf"},
    New CPerson With {.Name = "Schulz", .Strasse = "Wiesenweg 5",
    .PLZ = 32111, .Ort = "Biesdorf"}}
```

3.8.3 Destruktor und Garbage Collector räumen auf

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.

Der *Public*-Zugriffsmodifizierer entfällt, da Sie selbst den Destruktor nicht aufrufen dürfen, auch Parameter dürfen nicht übergeben werden.

SYNTAX: Protected Overrides Sub **Finalize()**
 ' hier Code für Aufräumarbeiten implementieren
 End Sub

BEISPIEL 3.35: Destruktor und Garbage Collector

VB 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 Shared anzahl As Integer = 0
```

Konstruktor:

```
Public Sub New()
    anzahl += 1
End Sub
```

Quasi-Destruktor:

```
Protected Overrides Sub Finalize()
    anzahl -= 1
    MyBase.Finalize()      ' Aufruf der Basisklassenmethode
End Sub
```

End Class

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

Zum Code der Klasse *Form1* fügen Sie hinzu:

```
Private kunde1 As CKunde          ' Objekt referenzieren
```

Objekt erzeugen:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    kunde1 = New CKunde
End Sub
```

Objekt entfernen:

```
Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
    kunde1 = Nothing              ' Objekt dereferenzieren
End Sub
```

Anzeige der im Speicher befindlichen Instanzen im Sekundentakt:

```
Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
    Label1.Text = CKunde.anzahl.ToString
End Sub
```

BEISPIEL 3.35: Destruktor und Garbage Collector

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 ziemlich lange 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 aufruft¹.

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!

Da wegen der Unberechenbarkeit der Objektvernichtung der Umgang mit der *Finalize*-Methode ziemlich problematisch ist, sollten Sie für das definierte Freigeben von Objekten besser eine separate Methode verwenden (siehe *Dispose*-Methode).

3.8.4 Mit Using den Lebenszyklus des Objekts kapseln

Auch mit dem Schlüsselwort *Using* kann man selbst für das sichere Erzeugen und Vernichten von Objekten sorgen. Voraussetzung dafür ist, dass das Objekt die *IDisposable*-Schnittstelle implemen-

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

tiert. Hinter den Kulissen wird ein *Try-Finally*-Block um das entsprechende Objekt generiert und beim Beenden für das Objekt *Dispose()* aufgerufen.

BEISPIEL 3.36: Sicheres Erzeugen und Freigeben von ADO.NET-Objekten (siehe 23.3.5)

```
VB ...
Using conn As New SqlConnection(connString)
    Using cmd As New SqlCommand(cmdString, conn)
        conn.Open()
        cmd.ExecuteNonQuery()
    End Using
End Using
...
```

Ein weiteres Beispiel für *Using* finden Sie im Praxisbeispiel

► 8.8.3 Ein Memory Mapped File verwenden

3.9 Vererbung und Polymorphie

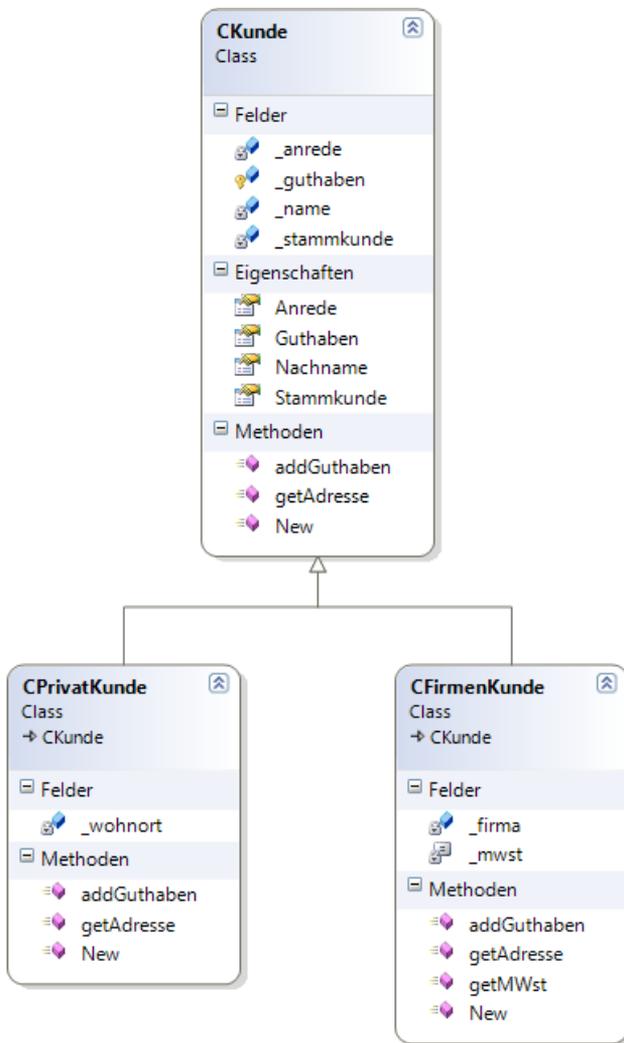
Ein zentrales OOP-Thema ist die *Vererbung*, die es ermöglicht, Klassen zu definieren, die von anderen Klassen abhängen. Eng mit der Vererbung verknüpft ist die *Polymorphie* (Vielfältigkeit). Man versteht darunter die Fähigkeit von Subklassen, die Methoden der Basisklasse mit unterschiedlichen Implementierungen zu verwenden. Visual Basic 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!

3.9.1 Vererbungsbeziehungen im Klassendiagramm

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



Das obige, 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 *New*-Methoden sind nichts weiter als die Konstruktoren der entsprechenden Klassen.

3.9.2 Überschreiben von Methoden (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* tauchen allerdings nochmals in den beiden Subklassen auf, warum? In unserem Beispiel handelt es sich um so genannte *überschriebene Methoden*, d.h., Adresse und Guthaben sollen für Privatkunden auf andere Weise als für Firmenkunden ermittelt werden. Genauereres dazu erfahren Sie im nächsten Abschnitt.

HINWEIS: Verwechseln Sie das *Überschreiben* von Methoden nicht mit dem in 2.8.5 beschriebenen *Überladen* von Methoden. Beides hat nichts, aber auch gar nichts, miteinander zu tun!

3.9.3 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 *Overridable*-Methodendeklarationen¹.

```
Public Class CKunde
```

Die privaten Felder:

```
Private _anrede As String
Private _name As String
Private _stammkunde As Boolean
```

Auf das folgende Feld soll auch eine Subklasse zugreifen können:

```
Protected _guthaben As Decimal
```

Der eigene Konstruktor:

```
Public Sub New(Anrede As String, Nachname As String)
    _anrede = Anrede
    _name = Nachname
End Sub
```

Die Eigenschaften:

```
Public WriteOnly Property Anrede()
    Set(value)
```

¹ Eigentlich hätten wir die Klasse auch noch als *MustInherit* deklarieren müssen (siehe dazu 3.6.6).

```
        _anrede = value
    End Set
End Property

Public WriteOnly Property Nachname()
    Set(value)
        _name = value
    End Set
End Property

Public Property Stammkunde() As Boolean
    Get
        Return _stammkunde
    End Get
    Set(value As Boolean)
        _stammkunde = value
    End Set
End Property

Public ReadOnly Property Guthaben() As Decimal
    Get
        Return _guthaben
    End Get
End Property
```

Die folgenden beiden Methoden sollen von den Subklassen überschrieben werden können:

```
Public Overridable Function getAdresse() As String
    Return _anrede & " " & _name
End Function

Public Overridable Sub addGuthaben(betrag As Decimal)
    If _stammkunde Then _guthaben += betrag
End Sub
```

```
End Class
```

Subklassen

Die erste Methode in der Subklasse ist in der Regel der Konstruktor. Dieser Konstruktor benutzt das *MyBase*-Schlüsselwort, um den Konstruktor der Basisklasse aufzurufen. Falls aber die Basisklasse über keinen eigenen Konstruktor verfügt, wird der Standardkonstruktor automatisch aufgerufen, wenn ein Objekt aus einer Subklasse erzeugt wird.

Das *Overrides*-Schlüsselwort der beiden Funktionen bedeutet, dass hier die in der Basisklasse definierten Funktionen überschrieben werden. Das erlaubt der Subklasse, eine eigene Implementierung der Funktionen zu realisieren.

HINWEIS: Wenn Sie das *Overrides*-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.

Der Code für die Subklasse *CPrivatKunde*:

```
Public Class CPrivatKunde
    Inherits CKunde
    Private _wohnort As String
```

Der Konstruktor ist notwendig, weil auch die Basisklasse einen eigenen Konstruktor verwendet:

```
Public Sub New(Anr As String, Name As String, Ort As String)
    MyBase.New(Anr, Name) ' Aufruf des Konstruktors der Basisklasse
    Me._wohnort = Ort     ' klassenspezifische Ergänzung
End Sub
```

Die Methoden werden überschrieben:

```
Public Overrides Function getAdresse() As String
    Return MyBase.Adresse & vbCrLf & _wohnort
End Function
```

```
Public Overrides Sub addGuthaben(geld As Decimal)
```

5% des Rechnungsbetrags werden als Guthaben angerechnet (Direktzugriff auf die *Protected*-Variable der Basisklasse *CKunde*):

```
    _guthaben += 0.05 * geld
End Sub
End Class
```

Der Code für die Subklasse *CFirmenKunde* unterscheidet sich in einigen Details:

```
Public Class CFirmenKunde
    Inherits CKunde
    Private _firma As String
    Private Const _mwst As Double = 0.19
```

Konstruktor (notwendig, weil Basisklasse eigenen Konstruktor verwendet):

```
Public Sub New(Anr As String, Name As String, Frm As String)
    MyBase.New(Anr, Name) ' Aufruf des ererbten Konstruktors
    Me._firma = Frm
End Sub
```

Überschreiben der überschreibbaren Methoden:

```
Public Overrides Function getAdresse() As String
    Return MyBase.Adresse & vbCrLf & _firma
End Function
```

```
Public Overrides Sub addGuthaben(brutto As Decimal)
    Dim netto As Decimal = brutto / CDec((1 + _mwst)) ' Netto berechnen
    MyBase.addGuthaben(netto * 0.01D) ' 1% als Guthaben angerechnet
End Sub
```

Eine normale Methode:

```
Public Function getMWst() As Double
    Return _mwst
End Function
```

End Class

Subklasse CPrivatKunde

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

Der Code für die Subklasse *CPrivatKunde*:

```
Public Class CPrivatKunde
    Inherits CKunde ' erbt von der Basisklasse CKunde!
    Private _wohnort As String
```

Ein eigener Konstruktor ist notwendig, weil auch die Basisklasse einen eigenen Konstruktor verwendet:

```
Public Sub New(Anr As String, Name As String, Ort As String)
    MyBase.New(Anr, Name) ' Aufruf des Konstruktors der Basisklasse
    Me._wohnort = Ort ' klassenspezifische Ergänzung
End Sub
```

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

```
Public Overrides Function getAdresse() As String
    Return MyBase.getAdresse & vbCrLf & _wohnort
End Function
```

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

```
Public Overrides Sub addGuthaben(geld As Decimal)
```

Hier erfolgt ein Direktzugriff auf die *Protected*-Variable *_guthaben* der Basisklasse *CKunde*:

```
    _guthaben += 0.05 * geld
End Sub
```

End Class

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
    Inherits CKunde

    Private _firma As String
    Private Const _mwst As Double = 0.19
```

Auch hier ist ein Konstruktor notwendig (weil Basisklasse eigenen Konstruktor verwendet):

```
Public Sub New(Anr As String, Name As String, Frm As String)
    MyBase.New(Anr, Name) ' Aufruf des ererbten Konstruktors
    Me._firma = Frm
End Sub
```

Überschreiben der überschreibbaren Methoden:

```
Public Overrides Function getAdresse() As String
    Return MyBase.getAdresse & vbCrLf & _firma
End Function

Public Overrides Sub addGuthaben(brutto As Decimal)
    Dim netto As Decimal = brutto / CDec((1 + _mwst)) ' Netto berechnen
    MyBase.addGuthaben(netto * 0.01) ' 1% als Guthaben angerechnet
End Sub
```

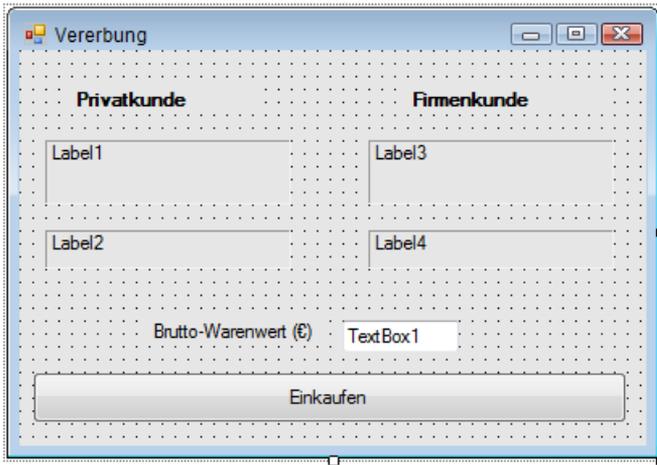
Und zum Schluss noch eine ganz normale Methode:

```
Public Function getMWst() As Double
    Return _mwst
End Function
End Class
```

Die Implementierung unserer drei Klassen ist geschafft!

Testoberfläche

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



3.9.4 Objekte implementieren

Es genügt, wenn wir mit nur zwei Objekten (ein Privat- und ein Firmenkunde) arbeiten:

```
Public Class Form1
    Private kunde1 As CPrivatkunde
    Private kunde2 As CFirmenkunde
```

Im Konstruktor des Formulars werden die beiden Objekte erzeugt. Fügen Sie zunächst den Rahmencode des Konstruktors hinzu, indem Sie in den beiden Comboboxen am oberen Rand des Codefenster Folgendes einstellen: *Klassenname = Form1, Methodennamen = New*.

Die Ja-/Nein-Eigenschaft *StammKunde* muss allerdings extra zugewiesen werden, da es dazu keinen passenden Konstruktor gibt.

```
Public Sub New()
    ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.
    InitializeComponent()
    ' Fügen Sie Initialisierungen nach dem InitializeComponent()-Aufruf hinzu.

    kunde1 = New CPrivatkunde("Herr", "Krause", "Leipzig")
    kunde1.Stammkunde = False

    kunde2 = New CFirmenkunde("Frau", "Müller", "Master Soft GmbH")
    kunde2.Stammkunde = True
    TextBox1.Text = "100"
End Sub
```

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

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    Dim brutto As Decimal = Convert.ToDecimal(TextBox1.Text)
```

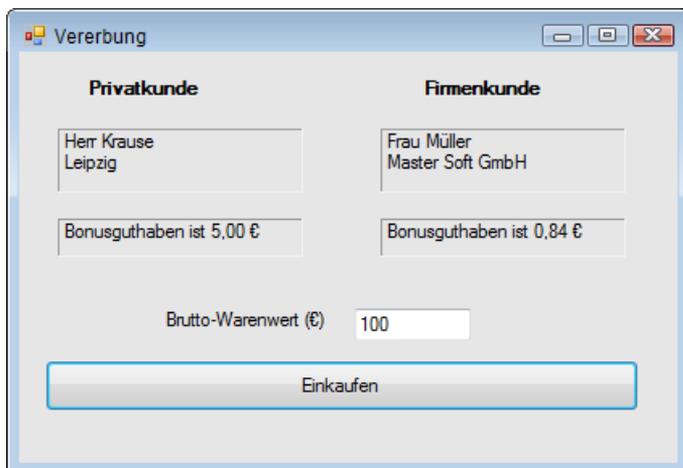
```

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")
End Sub
End Class

```

Praxistest

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



Die Werte in der 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.

3.9.5 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 3.37: Zur Klasse *CKunde* fügen wir eine Methode *test* hinzu

```
VB Public Class CKunde                                ' Basisklasse
    ...
    Public Sub test()
        MessageBox.Show("Hallo Kunde!")
    End Sub
End Class
```

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

```
Public Class CPrivatkunde                          ' abgeleitete Klasse
    Inherits CKunde

    Public Sub test()
        MessageBox.Show("Hallo Privatkunde!")
    End Sub

End Class
```

Im Quellcode-Editor erscheint der Name *test()* grün unterschlängelt. Der entsprechende Warnhinweis lautet: *Sub "test" führt Shadowing für einen überladbaren Member durch, der in der Basisklasse CKunde deklariert ist. Wenn Sie die Basismethode überladen möchten, muss die Methode als "Overloads" deklariert werden.*

Sie lassen diese Warnung unbeachtet. Der Test erfolgt in *Form1*:

```
Public Class Form1
    Private kunde1 As New CPrivatkunde()

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        kunde1.test()
    End Sub
End Class
```

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 Schlüsselwort *Overloads* markieren.

BEISPIEL 3.38: Die folgende Änderung macht das Vorgängerbeispiel transparenter (Ergebnis bleibt dasselbe)

```

VB Public Class CPrivatkunde      ' abgeleitete Klasse
    Inherits CKunde
    ...
    Public Overloads Sub test()
        MessageBox.Show("Hallo Privatkunde!")
    End Sub
End Class

```

3.9.6 Allgemeine Hinweise und Regeln zur Vererbung

Nachdem wir nun am praktischen Beispiel die Programmierung von Vererbungsbeziehungen kennen gelernt 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 *Overridable* deklariert werden.
- Fehlt das Schlüsselwort *Overridable* 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 *Overrides* deklariert werden.
- Wenn Sie das *Overrides*-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 das der Klassendeklaration nachgestellte Schlüsselwort *Inherits* bekannt gemacht:

SYNTAX: Public Class *SubKlasse*
 Inherits *Basisklasse*
 ' ... Implementierungscode
 End Class

- Eine Subklasse kann immer nur von einer einzigen Basisklasse abgeleitet werden (keine multiple Vererbung möglich).
- Mit dem *MyBase*-Objekt kann von den Subklassen auf die Basisklasse zugegriffen werden, mit dem *Me*-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 (*MyBase*-Schlüsselwort).
- Falls aber die Basisklasse über keinen eigenen Konstruktor verfügt, wird der Standardkonstruktor automatisch aufgerufen, wenn ein Objekt aus einer Subklasse erzeugt wird.

3.9.7 Polymorphe Methoden

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 Kompilierens 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 3.39: Polymorphe Methoden

VB 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, die diesmal in einem Array mit drei Feldern vom Typ der Basisklasse *CKunde* abgelegt werden sollen:

```
Public Class Form1
    Private Kunden(2) As CKunde           ' Referenz auf Basisklasse CKunde!
```

Im Konstruktorcode des Formulars erzeugen wir einen Privat- und zwei Firmenkunden:

```
Public Sub New()
    InitializeComponent()
    Dim kunde1 As New CPrivatKunde("Herr", "Krause", "Leipzig")
    kunde1.Stammkunde = False
    Dim kunde2 As New CFirmenKunde("Frau", "Müller", "Master Soft GmbH")
    kunde2.Stammkunde = True
    Dim Kunde3 As New CFirmenKunde("Herr", "Maus", "Manfreds Internet AG")
    Kunde3.Stammkunde = False
```

Unser eingangs deklariertes Array nimmt nun Privat- und Firmenkunden in wahlloser Reihenfolge auf:

BEISPIEL 3.39: Polymorphe Methoden

```

    Kunden(0) = kunde1
    Kunden(1) = kunde2
    Kunden(2) = Kunde3
    TextBox1.Text = "100"
End Sub

```

Das Array wird zunächst in einer *For-Next*-Schleife durchlaufen. Dabei werden die polymorphen Methoden (das sind die mit *Overridable* bzw. *Overrides* deklarierten) für alle Objekte aufgerufen. Es werden also sowohl die Adressen ausgegeben als auch für jeden Kunden das Guthaben berechnet, nachdem er für 100 € Waren gekauft hat.

```

Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    Dim brutto As Decimal = Convert.ToDecimal(TextBox1.Text)
    Label1.Text = String.Empty
    For i As Integer = 0 To Kunden.Length - 1
        Kunden(i).addGuthaben(brutto)
        Label1.Text = Label1.Text & vbCrLf & Kunden(i).getAdresse & vbCrLf &
            Kunden(i).Guthaben.ToString("C") & vbCrLf
    Next
End Sub

```

Obwohl im Array die Objekte bunt durcheinander gewürfelt sind, "weiß" das Programm zur Laufzeit genau, welche Implementierung der Methoden *getAdresse* und *addGuthaben* für den Privat- und für den Firmenkunden die richtige ist: Genau darin liegt der springende Punkt zum Verständnis der Polymorphie!

BEISPIEL 3.40: Die alternative Implementierung obigen Codes mittels *For Each*-Schleife bringt das Problem der Polymorphie noch deutlicher auf den Punkt

```

VB Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
    Dim brutto As Decimal = Convert.ToDecimal(TextBox1.Text)
    Label1.Text = String.Empty

```

Die Schleifenvariable *ku* ist eine Referenz auf die Basisklasse *CKunde*:

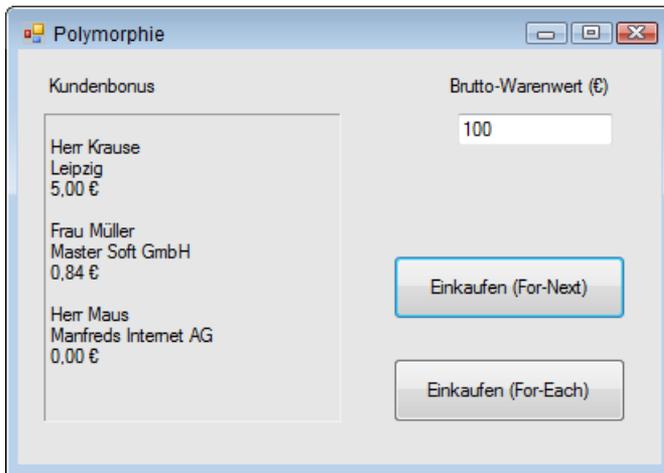
```

    For Each ku As CKunde In Kunden
        ku.addGuthaben(brutto)
        Label1.Text = Label1.Text & vbCrLf & ku.getAdresse & vbCrLf &
            ku.Guthaben.ToString("C") & vbCrLf
    Next
End Sub
End Class

```

Praxistest

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



HINWEIS: 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.

3.10 Besondere Klassen und Features

Zum Schluss wollen wir noch auf einige wichtige Klassen und Features eingehen, die in der OOP eine besondere Rolle spielen.

3.10.1 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*¹. Um zu verhindern, dass von abstrakten Klassen Instanzen gebildet werden, können diese mit dem Schlüsselwort *MustInherit* gekennzeichnet werden.

BEISPIEL 3.41: In unserem Vorgängerbeispiel werden von der Klasse *CKunde* keine Instanzen gebildet, sie kann deshalb als abstrakt deklariert werden.

```
VB Public MustInherit Class CKunde
    ...
End Class
```

Während die Referenzierung nach wie vor möglich ist

```
Dim Kunde As CKunde
```

¹ Können Sie sich vielleicht vorstellen, wie eine Instanz der Klasse *Fahrzeug* konkret aussehen soll?

BEISPIEL 3.41: In unserem Vorgängerbeispiel werden von der Klasse *CKunde* keine Instanzen gebildet, sie kann deshalb als abstrakt deklariert werden.

schlägt der Versuch einer Instanziierung fehl:

```
Kunde = New CKunde("Herr", "Krause")           ' Fehler
```

HINWEIS: Abstrakte Klassen ähneln einem weiteren wichtigen Softwarekonstrukt der OOP, der Schnittstelle (siehe Abschnitt 3.10.1).

3.10.2 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 *Overrides* überschrieben werden. Zur Kennzeichnung abstrakter Methoden verwenden Sie das Schlüsselwort *MustOverride*.

HINWEIS: Die Deklaration einer abstrakten Methode erfolgt in einer Zeile, also ohne Rumpf.

BEISPIEL 3.42: Die Funktion *getAdresse* einer abstrakten *CKunde*-Klasse wird in den Subklassen komplett überschrieben und kann deshalb anstatt mit *Overridable* mit *MustOverride* deklariert werden.

```
VB Public MustInherit Class CKunde
    ...
    Public MustOverride Function getAdresse() As String ' abstrakte virtuelle Methode
    ...
End Class

Public Class CPrivatKunde
    Inherits CKunde
    ...
    Public Overrides Function getAdresse() As String ' überschreibt Methode der Basisklasse
        Return _Anrede & " " & _Name & " " & _Wohnort
    End Function
    ...
End Class
```

3.10.3 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 *NotInheritable* schützen.

BEISPIEL 3.43: Die Klasse *CPrivatKunde* wird versiegelt und darf deshalb keine Nachkommen haben.

```
VB Public NotInheritable Class CPrivatKunde
    Inherits CKunde
    ...
End Class
```

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

```
Public Class CStudent          ' Fehler!!!
    Inherits CPrivatKunde
    ...
End Class
```

HINWEIS: Eine versiegelte Klasse kann niemals eine Basisklasse sein. Vererbungsmodifikatoren wie *MustInherit* und *Overridable* 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 vorschreibt.

3.10.4 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 (z.B. *Form1.Designer.vb*) vom Code des Entwicklers (*Form1.vb*) getrennt, was zu einer gesteigerten Übersichtlichkeit beiträgt, wovon man sich nach Öffnen eines neuen Windows Forms-Projekts selbst überzeugen kann).

Die Programmierung ist denkbar einfach, denn alle Teile der Klasse sind lediglich mit dem Modifizierer *Partial* zu kennzeichnen.

BEISPIEL 3.44: Eine einfache Klasse *CKunde*

```
VB Public Class CKunde
    Private _name As String
    Protected _guthaben As Decimal = 0

    Public Property NachName() As String
        Get
            Return _name
        End Get
        Set(value As String)
            _name = value
        End Set
    End Property
```

BEISPIEL 3.44: Eine einfache Klasse *CKunde*

```

Public Property Guthaben() As Decimal
    Get
        Return _guthaben
    End Get
    Set(value As Decimal)
        _guthaben = value
    End Set
End Property
Public Sub addGuthaben(betrag As Decimal)
    _guthaben += betrag
End Sub
End Class

```

Obige Klasse könnte (als eine von mehreren Möglichkeiten) wie folgt in drei partielle Klassen aufgesplittet werden:

```

Partial Public Class CKunde
    Private _name As String
    Protected _guthaben As Decimal = 0
End Class

Public Class CKunde
    Public Property NachName() As String
    Get
        Return _name
    End Get
    Set(value As String)
        _name = value
    End Set
End Property

    Public Property Guthaben() As Decimal
    Get
        Return _guthaben
    End Get
    Set(value As Decimal)
        _guthaben = value
    End Set
End Property
End Class

Partial Public Class CKunde
    Public Sub addGuthaben(betrag As Decimal)
        _guthaben += betrag
    End Sub
End Class

```

Wie Sie sehen, kann (muss aber nicht) bei einer der partiellen Definitionen auf *Partial* verzichtet werden, diese Klasse ist dann gewissermaßen die Ausgangsklasse, die durch den Code der anderen partiellen Klassen erweitert wird.

HINWEIS: Auch *Structure*- oder *Interface*-Definitionen können mittels *Partial*-Modifizierer gesplittet werden!

3.10.5 Die Basisklasse *System.Object*

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

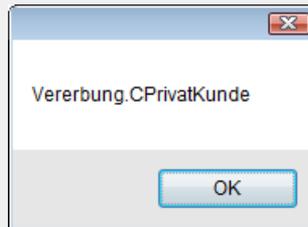
Alle öffentlichen Eigenschaften und Methoden von *System.Object* stehen automatisch auch in jedem Objekt zur Verfügung, welches Sie erzeugt haben. Beispielsweise ist in *System.Object* bereits ein Standardkonstruktor enthalten. Wenn Sie in Ihrem Objekt keinen eigenen Konstruktor definiert haben, wird 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 3.45: Die *ToString*-Methode liefert den Namen der Anwendungskomponente (die Windows-Anwendung heißt hier Vererbung) und die Klassenzugehörigkeit von *kunde1*.

VB `MessageBox.Show(kunde1.ToString)`

Ergebnis



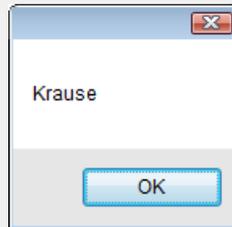
Sie können das standardmäßige Verhalten von *ToString* mittels *Overrides*-Schlüsselwort verändern. Dies erlaubt Ihnen eine individuelle Implementierung einiger Eigenschaften bzw. Methoden von *System.Object*.

BEISPIEL 3.46: Die gleiche *ToString*-Methode des Vorgängerbeispiels liefert nun den Namen des Kunden, wenn Sie die folgende Methode zum Klassenkörper von *CKunde* hinzufügen.

VB `Public Overrides Function ToString() As String
Return _name
End Function`

BEISPIEL 3.46: Die gleiche *ToString*-Methode des Vorgängerbeispiels liefert nun den Namen des Kunden, wenn Sie die folgende Methode zum Klassenkörper von *CKunde* hinzufügen.

Ergebnis



3.10.6 Property-Accessors

Die *Get*- und *Set*-Accessoren von Eigenschaften hatten in den ersten .NET-Versionen von Visual Basic die gleiche Sichtbarkeit wie die Eigenschaft zu der sie gehören. Seit .NET 2.0 ist es möglich, den Zugriff auf einen dieser Accessoren zu beschränken. Meist ist dies für den *Set*-Accessor sinnvoll, während der *Get*-Accessor in der Regel öffentlich bleiben soll.

BEISPIEL 3.47: Eine Eigenschaft mit *Get*- und *Set*-Accessoren. Der *Get*-Accessor besitzt die gleiche Sichtbarkeit wie die *KontoNummer*-Eigenschaft, während der *Set*-Accessor nur einen *Friend*-Zugriff erlaubt.

```
VB Private _knr As String
    Public Property KontoNummer() As String
        Get
            Return _knr
        End Get
        Friend Set(value As String)
            _knr = value
        End Set
    End Property
```

3.10.7 Nullbedingter Operator

Dieser mit VB 2015 eingeführte Operator ist eine bequeme Abkürzung für all jene Fälle, in denen auf *Nothing* zu prüfen ist:

BEISPIEL 3.48: Nullbedingter Operator

```
VB Dim frm = firma.Adresse?.Land
ist eine Abkürzung für
Dim _tmp = firma.Adresse
Dim frm = If(_tmp IsNot Nothing, _tmp.Adresse.Land, Nothing)
```

BEISPIEL 3.49: Es wird nur dann abgezweigt, wenn der Bewerber zugewiesen und älter als 60 Jahre ist

```
VB ...  
If bewerber?.Age > 60 Then ...  
...
```

BEISPIEL 3.50: Ein Standardnamen wird zugeteilt wenn der Kunde nicht zugewiesen ist

```
VB ...  
Dim name = If(kunde?.Name, "Müller")  
...
```

BEISPIEL 3.51: Nur wenn ein *kunde*-Objekt vorhanden ist wird *ToString()* ausgeführt

```
VB ...  
Dim s = kunde?.ToString()  
...
```

3.11 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.

Einen Ausweg bietet die Verwendung von Schnittstellen, diese bieten fast alle Möglichkeiten der Mehrfachvererbung, vermeiden aber deren Nachteile.

HINWEIS: Schnittstellen dienen dazu, um gemeinsame Merkmale ansonsten unabhängiger Klassen beschreiben zu können.

Eine Schnittstelle können Sie sich zunächst wie eine abstrakte Klasse (siehe 3.10.1) vorstellen, in welcher nur abstrakte Methoden definiert werden¹.

3.11.1 Definition einer Schnittstelle

Eine Schnittstelle können Sie zu Ihrem Projekt genauso hinzufügen wie eine neue Klasse. Anstatt des Schlüsselworts *Class* verwenden Sie aber *Interface*.

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

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

BEISPIEL 3.52: Eine Schnittstelle *IPerson*, die zwei Eigenschaften und eine Methode definiert²

```

VB Public Interface IPerson
    Property Nachname As String
    Property Vorname As String
    Function getName() As String
End Interface

```

Vielleicht vermissen Sie im obigen Beispiel die Zugriffsmodifizierer (**Public Property ...**), diese aber haben 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!

3.11.2 Implementieren einer Schnittstelle

Die implementierende Klasse benutzt anstatt des Schlüsselworts *Inherits* das Schlüsselwort *Implements*.

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

BEISPIEL 3.53: Die Klasse *CKunde* implementiert die Schnittstelle *IPerson*

```

VB Class CKunde
    Implements IPerson
    ...

    Die von IPerson geerbten abstrakten Klassenmitglieder müssen implementiert werden:

    Public Property Nachname As String Implements IPerson.Nachname
    Public Property Vorname As String Implements IPerson.Vorname

    Public Function getName() As String Implements IPerson.getName
        Return _Vorname & " " & _Nachname
    End Function

    Es folgen die normalen Klassenmitglieder:

    ...
End Class

```

Den kompletten Code finden Sie im Praxisbeispiel

► 3.12.4 Schnittstellenvererbung verstehen

² Das Interface verwendet selbstimplementierende Eigenschaften.

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

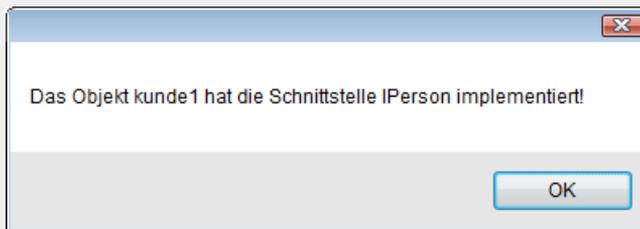
3.11.3 Abfragen, ob eine Schnittstelle vorhanden ist

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 *TypeOf*-Operator.

BEISPIEL 3.54: Wir ergänzen die Bedienoberfläche des Vorgängerbeispiels um eine zweite Schaltfläche.

```
VB Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
    If TypeOf kunde1 Is IPerson Then
        MessageBox.Show("Das Objekt kunde1 hat die Schnittstelle IPerson implementiert!")
    End If
End Sub
```

Ergebnis



3.11.4 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 3.55: Eine Klasse implementiert zwei Schnittstellen

```
VB Public Class CPrivatkunde
    Implements IPerson, IKunde
    ...
End Class
```

3.11.5 Schnittstellenprogrammierung ist ein weites Feld

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

- Anstatt 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).

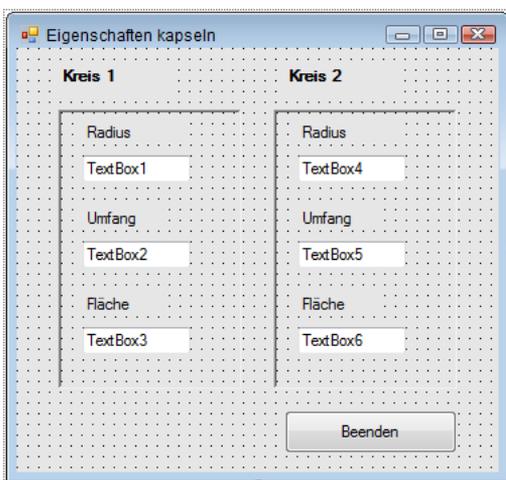
3.12 Praxisbeispiele

3.12.1 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 Beispiel handelt es sich um eine Klasse *CKreis* mit den Eigenschaften *Radius*, *Umfang* und *Fläche*. Diese Klasse speichert intern eine einzige Zustandsvariable *radZ*, aus welcher direkt beim Zugriff alle Eigenschaften berechnet werden.

Oberfläche

Um einen weiteren Vorteil der OOP zu demonstrieren, d.h. ohne viel Mehraufwand beliebig viele Instanzen aus einer Klasse bilden, 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.vb*.

```
Public Class CKreis
```

Die private Zustandsvariable speichert den Wert des Radius:

```
Private radZ As Double
```

Die Eigenschaft *Radius* (Rahmencode wird automatisch erzeugt!):

```
Public Property Radius() As String
    Get
        Return radZ.ToString("#,##0.00")
    End Get
    Set(value As String)
        If value <> String.Empty Then
            radZ = Cdbl(value)
        Else
            radZ = 0
        End If
    End Set
End Property
```

Die Eigenschaft *Umfang*:

```
Public Property Umfang() As String
    Get
        Return (2 * Math.PI * radZ).ToString("#,##0.00")
    End Get
    Set(value As String)
        If value <> String.Empty Then
            radZ = Cdbl(value) / 2 / Math.PI
        Else
            radZ = 0
        End If
    End Set
End Property
```

Die Eigenschaft *Fläche*:

```
Public Property Fläche() As String
    Get
        Return (Math.PI * Math.Pow(radZ, 2)).ToString("#,##0.00")
    End Get
    Set(value As String)
        If value <> String.Empty Then
            radZ = Math.Sqrt(Cdbl(value) / Math.PI)
        Else
            radZ = 0
        End If
    End Set
End Property
```

```

        End If
    End Set
End Property
End Class

```

Wechseln Sie nun in den Klassencode von *Form1*.

Quellcode Form1

```
Public Class Form1
```

Ein Objekt wird erzeugt:

```
    Private Kreis1 As New CKreis()
```

Die folgenden Event-Handler sind einfach und übersichtlich, da die Objekte die inneren Funktionalitäten wegekapseln. Den Radius ändern:

```

    Private Sub TextBox1_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox1.KeyUp
        With Kreis1
            .Radius = TextBox1.Text
            TextBox2.Text = .Umfang
            TextBox3.Text = .Fläche
        End With
    End Sub

```

Den Umfang ändern:

```

    Private Sub TextBox2_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox2.KeyUp
        With Kreis1
            .Umfang = TextBox2.Text
            TextBox1.Text = .Radius
            TextBox3.Text = .Fläche
        End With
    End Sub

```

Die Fläche ändern:

```

    Private Sub TextBox3_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox3.KeyUp
        With Kreis1
            .Fläche = TextBox3.Text
            TextBox1.Text = .Radius
            TextBox2.Text = .Umfang
        End With
    End Sub

```

Der Code für *Kreis2* ist analog aufgebaut und braucht deshalb hier nicht wiederholt zu werden (siehe Beispieldaten).

```
End Class
```

Test

Sobald Sie eine beliebige Eigenschaft ändern, werden die anderen zwei sofort aktualisiert! Wegen der in der Klasse eingebauten Eingabepfung verursacht ein leerer Eingabewert keinen Fehler. Aus Gründen der Übersichtlichkeit wurde aber auf das Abfangen weiterer Eingaben, die sich nicht in einen numerischen Wert konvertieren lassen, verzichtet.

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

Objektinitialisierer

Man kann ein Objekt auch dann erzeugen und initialisieren, wenn es dazu keinen Konstruktor gibt. Sie könnten also im Code von *Form1* die Instanziierung der Klasse durch direktes Zuweisen ihrer Eigenschaften wie folgt vornehmen:

```
Private Kreis1 As New CKreis With {.Radius = "1.0"}
```

HINWEIS: Mehr zu Objektinitialisierern erfahren Sie im Abschnitt 3.8.2!

3.12.2 Eine statische Klasse anwenden

Als "statisch" wollen wir hier solche Klassen bezeichnen, die lediglich statische (*Shared*) Mitglieder haben. Solche Klassen eignen sich beispielsweise ideal für Formelsammlungen (siehe *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

Lediglich ein *Formular* mit zwei *TextBoxen* zur Eingabe von Kugeldurchmesser und Kugelvolumen ist erforderlich (siehe Laufzeitanzeige).

Quellcode CKugel

Statische Funktionen werden mit dem Schlüsselwort *Shared* gekennzeichnet.

```
Public Class CKugel
    Public Shared Function Durchmesser_Volumen(durchmesser As String) As Double
        Dim dur As Double = System.Double.Parse(durchmesser)
        Dim vol As Double = Math.Pow(dur, 3) * Math.PI / 6.0
    End Function
End Class
```

```

    Return vol
End Function

Public Shared Function Volumen_Durchmesser(volumen As String) As Double
    Dim vol As Double = System.Double.Parse(volumen)
    Dim dur As Double = Math.Pow(6 / Math.PI * vol, 1 / 3.0)
    Return dur
End Function
End Class

```

Quellcode Form1

Die Verwendung der Klasse im Formularcode:

```
Public Class Form1
```

Die Berechnung startet nach Betätigen der *Enter*-Taste:

```

Private Sub TextBox1_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox1.KeyUp
    If e.KeyCode = Keys.Enter And TextBox1.Text <> String.Empty Then
        TextBox2.Text = CKugel.Durchmesser_Volumen(TextBox1.Text).ToString("#,##0.000")
    End If
End Sub

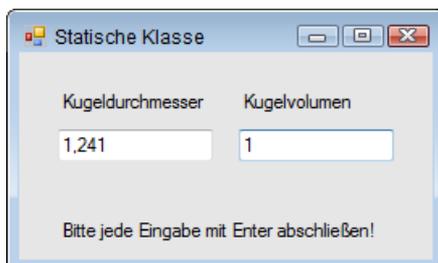
Private Sub TextBox2_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox2.KeyUp
    If e.KeyCode = Keys.Enter And TextBox2.Text <> String.Empty Then
        TextBox1.Text = CKugel.Volumen_Durchmesser(TextBox2.Text).ToString("#,##0.000")
    End If
End Sub
End Class

```

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:



3.12.3 Vom fetten zum dünnen 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 Programmieretechniken 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 damit die gesamte 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

Siehe Laufzeitansicht.

Quellcode (Fat Client)

```
Public Class Form1
```

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

```
Private op As Char           ' aktueller Operator (+, -, *, /)
Private reg1 As String = Nothing ' erstes Register (Operand)
Private reg2 As String = Nothing ' zweites Register (Operand)
```

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

```
Private state As Byte = 1           ' aktuelles Register (1 oder 2)
```

Typisch für die nun folgenden Ereignisbehandlungen ist, dass die durchgeführten Aktionen vom Wert der Zustandsvariablen *state* abhängig sind.

Zur Eingabe einer Ziffer (0..9) benutzt der gesamte Ziffernblock eine gemeinsame Ereignisbehandlung:

```
Private Sub ButtonZ_Click(sender As Object, e As EventArgs) Handles Button1.Click,
    Button2.Click, Button3.Click, Button4.Click, Button5.Click, Button6.Click,
    Button7.Click, Button8.Click, Button9.Click, Button10.Click, Button11.Click
    Dim cmd As Button = CType(sender, Button)
    Select Case state
        Case 1           ' zum ersten Operanden hinzufügen:
            reg1 &= cmd.Text.Chars(0)
            Label1.Text = reg1
        Case 2           ' zum zweiten Operanden hinzufügen:
            reg2 &= cmd.Text.Chars(0)
            Label1.Text = reg1 & " " & op & " " & reg2
    End Select
End Sub
```

Für die Eingabe der Operation (+, -, *, /) wird ähnlich verfahren:

```
Private Sub ButtonOp_Click(sender As Object, e As EventArgs) _
    Handles ButtonAdd.Click, ButtonSub.Click, ButtonMult.Click, ButtonDiv.Click
    Dim cmd As Button = CType(sender, Button)
    Select Case state
        Case 1
            op = cmd.Text.Chars(0)           ' neuer Operand ...
            state = 2                         ' ... und Zustandswechsel
        Case 2
            ergebnis()                     ' erst Zwischenergebnis mit altem Operand ermitteln ...
            op = cmd.Text.Chars(0)         ' ... dann neuer Operand
    End Select
    Label1.Text = reg1.ToString & " " & op
    reg2 = Nothing                          ' Reg2 löschen
End Sub
```

Die folgende Hilfsprozedur führt die Rechenoperation aus und speichert deren Ergebnis in *reg1*:

```
Private Sub ergebnis()
    Dim r1 As Double = Convert.ToDouble(reg1)
    Dim r2 As Double = Convert.ToDouble(reg2)
    Select Case op
        Case "+"c
            reg1 = (r1 + r2).ToString
        Case "-"c
            reg1 = (r1 - r2).ToString
```

```

        Case "*"c
            reg1 = (r1 * r2).ToString
        Case "/"c
            reg1 = (r1 / r2).ToString
    End Select
    reg2 = Nothing ' löscht zweites Register
End Sub

```

Die Ergebnistaste (=):

```

Private Sub ButtonResult_Click(sender As Object, e As EventArgs) Handles ButtonResult.Click
    If state = 2 Then
        ergebnis()
        Label1.Text &= " = " & reg1
        state = 1
    Else
        Label1.Text = reg1
        reg2 = Nothing ' löscht zweites Register
    End If
End Sub

```

Letztes eingegebenes Zeichen löschen (CE):

```

Private Sub ButtonCE_Click(sender As Object, e As EventArgs) Handles ButtonCE.Click
    Select Case state
        Case 1
            If Not (reg1 = Nothing) Then
                reg1 = reg1.Remove(reg1.Length - 1, 1)
                Label1.Text = reg1
            End If
        Case 2
            If Not (reg2 = Nothing) Then
                reg2 = reg2.Remove(reg2.Length - 1, 1)
                Label1.Text = reg2
            End If
    End Select
End Sub

```

Alle Register sowie Anzeige löschen und Anfangszustand herstellen:

```

Private Sub ButtonCLR_Click(sender As Object, e As EventArgs) Handles ButtonCLR.Click
    reg1 = Nothing : reg2 = Nothing : Label1.Text = String.Empty
    state = 1
End Sub

```

Schließlich noch der Vorzeichenwechsel (+/-):

```

Private Sub ButtonVZ_Click(sender As Object, e As EventArgs) Handles ButtonVZ.Click
    Dim r As Double
    Select Case state
        Case 1
            r = -Convert.ToDouble(reg1)

```

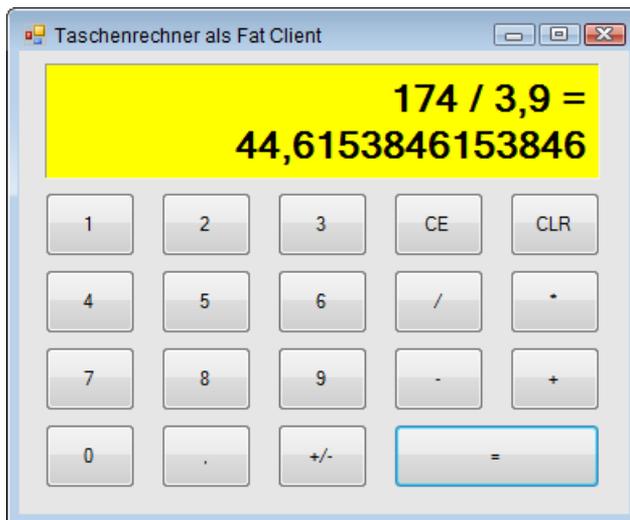
```

    reg1 = r.ToString
    Label1.Text = reg1
  Case 2
    r = -Convert.ToDouble(reg2)
    reg2 = r.ToString
    Label1.Text = reg1 & " " & op & " " & reg2
  End Select
End Sub
End Class

```

Test

Der Vorzug gegenüber üblichen Rechnern (oder auch dem im Windows-Zubehör) sticht sofort ins Auge: Man kann den Rechenvorgang mitverfolgen, weil der komplette Ausdruck angezeigt wird.



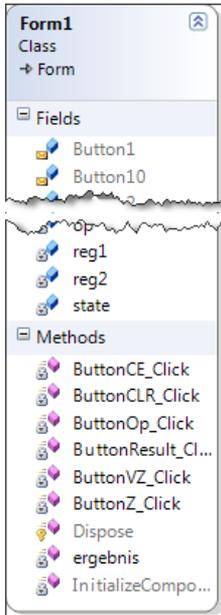
HINWEIS: Wie bei jedem anderen einfachen Taschenrechner auch, bleibt hier die Rangfolge der Operatoren (Punktrechnung geht vor Strichrechnung) unberücksichtigt. Bei der Eingabe von mehreren Operationen hintereinander, z.B. $3 + 4 * 12$, ist deshalb zu beachten, dass erst die höherwertige Operation auszuführen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

Bemerkungen zum fetten Client

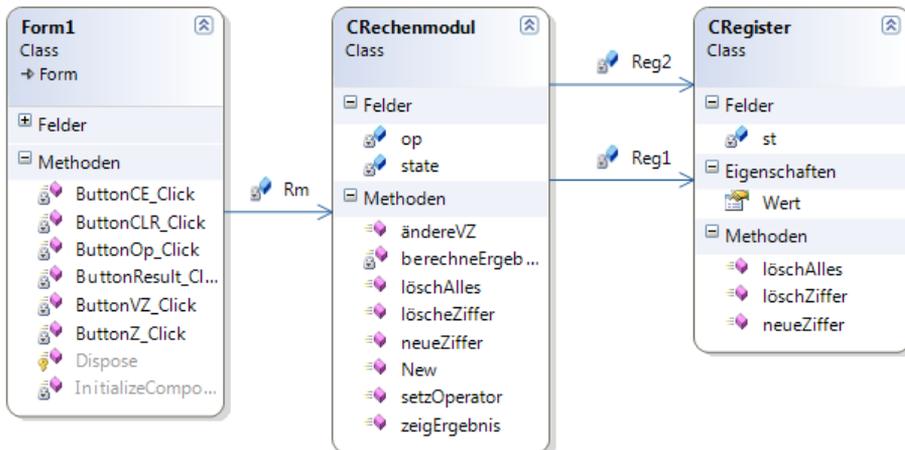
- Den Programmablauf könnte man in Gestalt eines Zustandsüberförungsdiagramms (*State Chart*) noch anschaulicher darstellen (siehe Kapitel 25).
- Leider ist die gesamte Intelligenz der Anwendung in der Benutzerschnittstelle *Form1* enthalten, also ein typischer "fetter" Client. Transparenz, Wiederverwendbarkeit und Wartbarkeit des Codes sind demzufolge katastrophal! Wie man das Programm auf ein höheres objektorientiertes Niveau heben kann, soll die folgende alternative Realisierung unseres Taschenrechners zeigen.

Abmagerungskur für den fetten Client

Dass es sich bei unserem alten Programm tatsächlich um einen Fat Client handelt, zeigt das zugehörige Klassendiagramm. Klicken Sie dazu im Projektmappen-Explorer mit der rechten Maustaste auf *Form1.vb* 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:



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

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 st As String = Nothing
```

Zugriff auf den numerischen Wert von *st*:

```
    Public Property Wert() As Double
        Get
            Try
                Wert = Convert.ToDouble(st)
            Catch
                Wert = 0
            End Try
        End Get
        Set(value As Double)
            st = value.ToString
        End Set
    End Property
```

Hinzufügen einer einzelnen Ziffer und Rückgabe des Anzeigestrings:

```
    Public Function neueZiffer(z As Char) As String
        If (Char.IsDigit(z) Or z = ",") Then
            st &= z
```

```

    Return (st)
Else
    Return (String.Empty)
End If
End Function

```

Letzte Ziffer löschen und Anzeigestring zurückgeben:

```

Public Function löschtZiffer() As String
    If st.Length > 0 Then
        st = st.Remove(st.Length - 1, 1)
    End If
    Return (st)
End Function

```

Gesamtes Register löschen:

```

Public Sub löschtAlles()
    st = String.Empty
End Sub

```

```
End Class
```

Quellcode für CRechenmodul

```

Public Class CRechenmodul

    Private state As Byte = 1          ' Startmodus (Zustandsvariable)
    Private op As Char                 ' aktueller Operator
    Private Reg1, Reg2 As CRegister    ' zwei Rechenregister

```

Im Konstruktor werden zwei Register-Objekte erzeugt:

```

Public Sub New()
    Reg1 = New CRegister()
    Reg2 = New CRegister()
End Sub

```

Zifferneingabe in aktuelles Register:

```

Public Function neueZiffer(z As Char) As String
    If state = 1 Then                ' zum ersten Register hinzufügen:
        Return (Reg1.neueZiffer(z))
    Else                             ' zum zweiten Register hinzufügen:
        Return (Reg1.Wert.ToString & " " & op & " " & Reg2.neueZiffer(z))
    End If
End Function

```

Letzte Ziffer des aktuellen Registers löschen und resultierenden Registerinhalt zurückgeben:

```

Public Function löscheZiffer() As String
    If state = 1 Then
        Return (Reg1.löschtZiffer())
    End If
End Function

```

```

Else
    Return (Reg2.löschZiffer())
End If
End Function

```

Vorzeichen des aktuellen Registers umkehren:

```

Public Function ändereVZ() As String
    If state = 1 Then
        Reg1.Wert = -Reg1.Wert
        Return (Reg1.Wert.ToString)
    Else
        Reg2.Wert = -Reg2.Wert
        Return (Reg1.Wert.ToString & " " & op & " " & Reg2.Wert.ToString)
    End If
End Function

```

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

```

Public Function setzOperator(o As Char) As String
    If state = 1 Then
        state = 2
    Else
        berechneErgebnis()          ' Zwischenergebnis (mit altem Operator) ermitteln
    End If
    op = o                          ' neuen Operator übernehmen
    Reg2.löschAlles()              ' zweites Register löschen
    Return (Reg1.Wert.ToString & " " & op)
End Function

```

Die abschließende Rechenoperation ausführen und das Ergebnis liefern:

```

Public Function zeigErgebnis() As String
    Dim s As String = ""
    If state = 1 Then                ' im Startmodus wird noch nichts berechnet, ...
        Reg2.löschAlles()           ' ... lediglich zweites Register gelöscht
    Else
        s = " = " & berechneErgebnis()
        state = 1
    End If
    Return (s)
End Function

```

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

```

Private Function berechneErgebnis() As String
    Select Case op
        Case "+"c
            Reg1.Wert = Reg1.Wert + Reg2.Wert
        Case "-"c

```

```

        Reg1.Wert = Reg1.Wert - Reg2.Wert
    Case "*"c
        Reg1.Wert = Reg1.Wert * Reg2.Wert
    Case "/"c
        Reg1.Wert = Reg1.Wert / Reg2.Wert
End Select
Reg2.löschAlles()      ' zweites Register löschen
Return Reg1.Wert.ToString
End Function

```

Alle Register löschen und Startzustand wieder herstellen:

```

Public Sub löschAlles()
    Reg1.löschAlles()
    Reg2.löschAlles()
    state = 1
End Sub
End Class

```

Quellcode für Form1

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

```
Public Class Form1
```

Einzig globale Variable ist eine Instanz der Klasse *CRechenmodul*:

```
Private Rm As New CRechenmodul()
```

Eine Ziffer eingeben (0..9):

```

Private Sub ButtonZ_Click(sender As Object, e As EventArgs) Handles Button1.Click,
    Button9.Click, Button8.Click, Button7.Click, Button6.Click, Button5.Click,
    Button4.Click, Button3.Click, Button2.Click, ButtonKomma.Click, Button0.Click
    Dim cmd As Button = CType(sender, Button)
    Label1.Text = Rm.neueZiffer(cmd.Text.Chars(0))
End Sub

```

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

```

Private Sub ButtonOp_Click(sender As Object, e As EventArgs) _
    Handles ButtonAdd.Click, ButtonSub.Click, ButtonMult.Click, ButtonDiv.Click
    Dim cmd As Button = CType(sender, Button)
    Label1.Text = Rm.setzOperator(cmd.Text.Chars(0))
End Sub

```

Ergebnis anzeigen (=):

```

Private Sub ButtonResult_Click(sender As Object, e As EventArgs) Handles ButtonResult.Click
    Label1.Text &= Rm.zeigErgebnis
End Sub

```

Letztes eingegebenes Zeichen löschen (CE):

```
Private Sub ButtonCE_Click(sender As Object, e As EventArgs) Handles ButtonCE.Click
    Label1.Text = Rm.löscheZiffer()
End Sub
```

Alle Register sowie Anzeige löschen und Anfangszustand wieder herstellen:

```
Private Sub ButtonCLR_Click(sender As Object, e As EventArgs) Handles ButtonCLR.Click
    Rm.löschAlles()
    Label1.Text = String.Empty
End Sub
```

Vorzeichenwechsel (+/-):

```
Private Sub ButtonVZ_Click(sender As Object, e As EventArgs) Handles ButtonVZ.Click
    Label1.Text = Rm.ändereVZ()
End Sub
```

```
End Class
```

Test

Sie werden keinerlei Unterschied in Aussehen und Funktion 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 3.12.8.

3.12.4 Schnittstellenvererbung verstehen

Ein mächtiges Feature der OOP ist das Konzept der Schnittstellenvererbung (siehe Abschnitt 3.11). Die Schnittstellenvererbung erschließt sich dem Einsteiger am leichtesten, wenn er sich vorher das Konzept abstrakter Klassen und Methoden (siehe Abschnitt 3.10.1) verinnerlicht hat.

In unserem Demobeispiel wollen wir Geldbeträge auf das Konto eines Kunden einzahlen bzw. von dort abheben. Die erste Lösung soll mittels einer abstrakten Klasse erfolgen¹. Die zweite Lösung benutzt eine Schnittstelle (Interface).

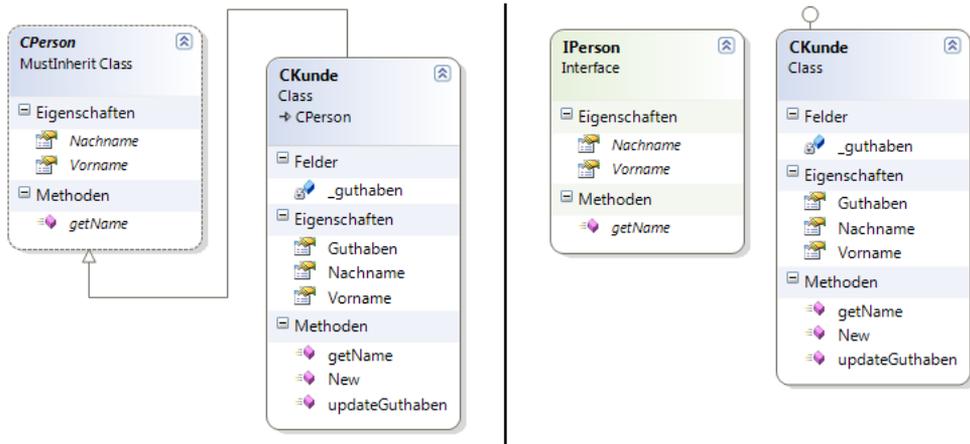
Klassendiagramme

Die erste Variante zeigt das links abgebildete Klassendiagramm. Hier erbt die Klasse *CKunde* von der abstrakten Klasse *CPerson*. Letztere verfügt ausschließlich über abstrakte Klassenmitglieder.

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

Diese enthalten nur die Eigenschafts- bzw. Methodendeklaration, also keinerlei Code. Die Implementierung muss komplett in der erbbenden 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.



Oberfläche Form1

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

Lassen Sie uns mit der ersten Variante beginnen!

Quellcode CPerson

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

```
Public MustInherit Class CPerson
    Public MustOverride Property Nachname As String
    Public MustOverride Property Vorname As String
    Public MustOverride Function getName() As String
End Class
```

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:

```
Public Class CKunde
    Inherits CPerson

    Private _guthaben As Decimal
```

Die drei von *CPerson* geerbten abstrakten Klassenmitglieder müssen überschrieben werden (wir verwenden selbst implementierende Eigenschaften, die die entsprechenden Zustandsvariablen *_Vorname* und *_Nachname* automatisch im Hintergrund anlegen):

```
Public Overrides Property Vorname As String
Public Overrides Property Nachname As String

Public Overrides Function getName() As String
    Return _Vorname & " " & _Nachname
End Function
```

Es folgen die normalen Klassenmitglieder:

```
Public Sub New(vor As String, nach As String) ' ein Konstruktor
    _Vorname = vor
    _Nachname = nach
End Sub

Public ReadOnly Property Guthaben As Decimal
    Get
        Return _guthaben
    End Get
End Property

Public Sub updateGuthaben(betrag As Decimal)
    _guthaben += betrag
End Sub
End Class
```

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

Quellcode Form1

```
Public Class Form1
```

Zu Beginn wird ein Kunde erzeugt, initialisiert und angezeigt:

```
Private kunde1 As CKunde
Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
    kunde1 = New CKunde("Max", "Müller")
    TextBox1.Text = kunde1.Vorname
    TextBox2.Text = kunde1.Nachname
    TextBox3.Text = "10,50"
End Sub
```

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 Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    kunde1.Vorname = TextBox1.Text
```

```
kunde1.Nachname = TextBox2.Text
Dim betrag As Decimal = 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") & " !"
```

```
End Sub
End Class
```

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 und vereinfachen Sie den Code wie folgt:

```
Public Interface IPerson
    Property Nachname As String
    Property Vorname As String
    Function getName() As String
End Interface
```

Quellcode CKunde

Auch hier sind nur minimale Änderungen erforderlich: An die Stelle von *Inherits* tritt *Implements*. Bei den von *IPerson* geerbten Schnittstellenmitgliedern fehlen die *Overrides*-Modifizierer. Stattdessen werden durch abschließende Kennzeichnung der Methodendeklarationen mit *Implements* und Benennung der Schnittstellenmitglieder Mehrdeutigkeiten vermieden.

```
Public Class CKunde
    Implements IPerson

    Private _guthaben As Decimal
    Public Property Vorname As String Implements IPerson.Vorname
    Public Property Nachname As String Implements IPerson.Nachname

    Public Function getName() As String Implements IPerson.getName
        Return _Vorname & " " & _Nachname
    End Function

    ...

```

Der restliche Code bleibt unverändert.

Das war es auch schon, denn der Quellcode von *Form1*, in welchem die Klasse *CKunde* instanziiert und verwendet wird, gleicht bis ins letzte Detail dem seines Vorgängers.

Auch beim Testen des Codes werden Sie keinerlei Veränderungen zum Vorgängerprojekt feststellen.

Vergleichen Sie beide Varianten, so stellen Sie fest, dass die Realisierung mittels Schnittstelle die Transparenz und Übersichtlichkeit des Codes deutlich steigert.

3.12.5 Aggregation und Vererbung gegenüberstellen

Jeder Programmierer hat den Ehrgeiz, mit möglichst wenig Schreiarbeit auszukommen und möglichst viel von seinem Code wieder verwenden zu können. Voraussetzung dafür sind optimale Klassendiagramme, für die es unter dem Aspekt der Wiederverwendbarkeit von Code zwei wesentliche Beziehungen gibt:

- Aggregation/Komposition
- Vererbung

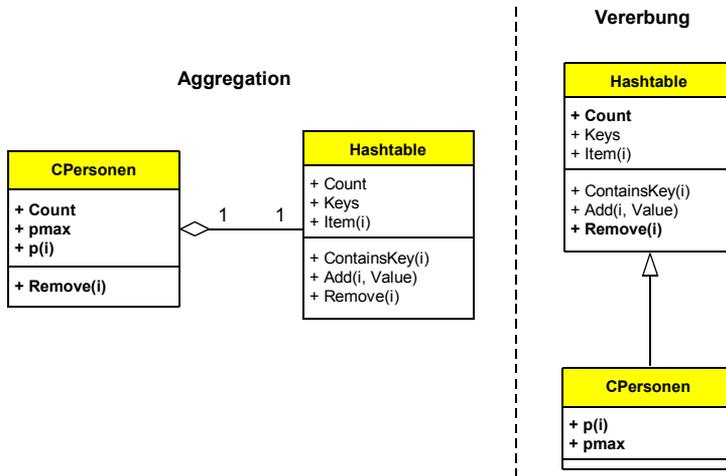
Damit sind wir bereits bei zwei fundamentalen Begriffen der OOP angelangt. Ob wir von *Aggregation* oder *Komposition* sprechen ist in diesem Zusammenhang unerheblich, denn die *Komposition* stellt lediglich die stärkere Form der *Aggregation* dar.

Unter Vererbung ist hier genau genommen die *Implementierungsvererbung* gemeint, denn die unter .NET ebenfalls mögliche *Interfacevererbung* erspart keinerlei Schreiarbeit.

HINWEIS: Sowohl Aggregation/Komposition als auch Vererbung verlangen eine spezifische Herangehensweise bei der Implementierung, die clientseitige Nutzung der Klasse ist aber identisch.

In unserem Beispiel soll dies an Hand einer kleinen Personalverwaltung demonstriert werden, wobei alle Personen in einer Auflistung gespeichert sind, welche die Funktionalität der recht leistungsfähigen *Hashtable*-Klasse nutzt.

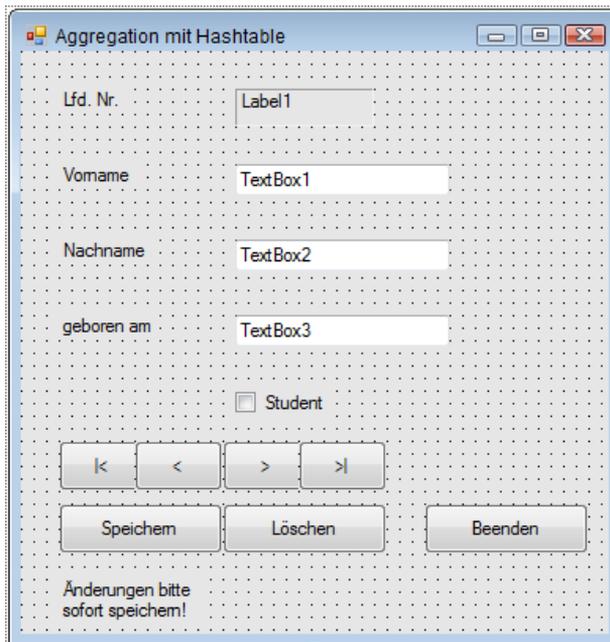
In der folgenden Abbildung sind die Klassendiagramme beider Varianten gegenübergestellt.



Die Klasse *CPersonen* soll über die Standardeigenschaft $p(i)$ einen indizierten Lese- und Schreibzugriff auf die Elemente der Auflistung bereitstellen, um damit die *Item*- und die *Add*-Methode von *Hashtable* zu kapseln. Weitere Eigenschaften sind *Count* (Gesamtzahl der abgespeicherten Personen) sowie *pmax* (höchster Index bzw. Schlüsselwert innerhalb der Auflistung). Die von *Hashtable* direkt geerbte *Remove*-Methode ermöglicht das Löschen einer bestimmten Person.

Benutzeroberfläche

Das abgebildete Hauptformular bedarf wohl keiner besonderen Erläuterung:



Variante 1: Klassen CPerson und CPersonen mit Aggregation

```
Public Class Form1
```

Der Übersichtlichkeit halber haben wir hier den Code beider Klassen zum *Form1*-Klassenmodul hinzugefügt, Sie könnten ihn aber auch ohne weiteres in ein oder zwei separate Klassenmodule auslagern.

```
Public Class CPerson
    Public Vorname, Nachname As String
    Public Geburt As Date
    Public student As Boolean
End Class

Public Class CPersonen
```

Innerhalb der Klasse *CPersonen* wird die Klasse *Hashtable* instanziiert – das ist Aggregation pur!

```
Private ht As New Hashtable()
```

Mit einem kleinen Trick, wir definieren eine Eigenschaft *p* als Standardeigenschaft, implementieren wir einen *Quasi-Indexer*¹ für den indizierten Zugriff auf die Personenliste.

HINWEIS: Mittels dieser Standardeigenschaft kann eine *CPersonen*-Collection auf die gleiche Weise wie ein Array indiziert werden!

```
Default Public Property p(i As Integer) As CPerson
    Get
        If ht.ContainsKey(i) Then
            Return CType(ht.Item(i), CPerson)
        Else
            Return Nothing
        End If
    End Get
    Set(ByVal value As CPerson)
        If ht.ContainsKey(i) Then
            ht.Item(i) = value ' überschreiben, falls Schlüssel vorhanden
        Else
            ht.Add(i, value) ' anhängen, falls Schlüssel noch nicht besetzt
        End If
    End Set
End Property
```

Das Löschen eines Elements der Auflistung:

```
Public Sub Remove(i As Integer)
    ht.Remove(i)
End Sub
End Class
```

¹ Im Unterschied zu C# erlaubt Visual Basic nicht die direkte Implementierung eines Indexers.

Nachdem die Klassen *CPerson* und *CPersonen* implementiert sind, geht es mit dem eigentlichen Code von *Form1* weiter.

Objekte instanziiieren:

```
Private pListe As New CPersonen()    ' die Personenliste
Private person As CPerson           ' die aktuelle Person
```

Zustandsvariablen zum Steuern der Anzeige:

```
Private pos As Integer = 1          ' die aktuelle Position
Private pmax As Integer             ' die max. Anzahl von Personen
```

Die Startaktivitäten:

```
Protected Overrides Sub OnLoad(e As EventArgs)
    Label1.Text = pos.ToString()
    anzeigeLöschen()
    MyBase.OnLoad(e)
End Sub
```

Die Anzeige der aktuellen Person:

```
Private Sub anzeigen()
    Label1.Text = pos.ToString()
    person = pListe(pos)           ' Zugriff wie über Indexer!
    Try
        With person
            TextBox1.Text = .Vorname
            TextBox2.Text = .Nachname
            TextBox3.Text = .Geburt.ToString("dd.MM.yyyy")
            CheckBox1.Checked = .student
        End With
    Catch
        anzeigeLöschen()
    End Try
End Sub
```

Die Hilfsroutine zum Löschen der Anzeige:

```
Private Sub anzeigeLöschen()
    Label1.Text = String.Empty
    TextBox1.Text = String.Empty : TextBox2.Text = String.Empty
    TextBox3.Text = "00:00:00"
    CheckBox1.Checked = False
End Sub
```

Vorwärts blättern:

```
Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
    pos += 1
    anzeigen()
End Sub
```

Rückwärts blättern:

```
Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
    If pos > 1 Then
        pos -= 1
        anzeigen()
    End If
End Sub
```

Zum Anfang:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    pos = 1
    anzeigen()
End Sub
```

Zum Ende:

```
Private Sub Button4_Click(sender As Object, e As EventArgs) Handles Button4.Click
    pos = pmax
    anzeigen()
End Sub
```

Speichern der aktuellen Person:

```
Private Sub Button5_Click(sender As Object, e As EventArgs) Handles Button5.Click
    person = New CPerson() ' nur Instanzen können hinzugefügt werden!!!
    With person
        .Vorname = TextBox1.Text
        .Nachname = TextBox2.Text
        .Geburt = Convert.ToDateTime(TextBox3.Text)
        .student = CheckBox1.Checked
    End With
    pListe(pos) = person ' Zugriff wie über Indexer
    If pos > pmax Then pmax = pos ' max. Schlüsselwert
    anzeigen()
End Sub
```

Löschen der aktuellen Person:

```
Private Sub Button6_Click(sender As Object, e As EventArgs) Handles Button6.Click
    pListe.Remove(pos)
    anzeigeLöschen()
End Sub
...
End Class
```

Test

Es können beliebig viele Personen eingegeben werden. Der zugeteilte Schlüsselwert entspricht der "Lfd.Nr"-Anzeige. Wenn unter dem Schlüssel bereits eine Person existiert, wird diese überschrieben, anderenfalls neu angelegt.

HINWEIS: Das Abspeichern passiert nicht automatisch beim Weiterblättern, sondern Sie müssen **vor** dem Weiterblättern die *Speichern*-Schaltfläche klicken (ansonsten sind die Änderungen futsch)!

Variante 2: Klasse CPersonen mit Vererbung

Alternativ zur Aggregation können wir die Klasse *CPersonen* auch so implementieren, dass die Eigenschaften/Methoden direkt von der Klasse *Hashtable* "geerbt" werden (siehe Abbildung zu Beginn).

```
Public Class CPersonen
    Inherits Hashtable          ' Vererbung

    Default Public Property p(i As Integer) As CPerson
    Get
        If ContainsKey(i) Then
            Return CType(Item(i), CPerson)
        Else
            Return Nothing
        End If
    End Get
    Set(ByVal value As CPerson)
        If ContainsKey(i) Then
            Item(i) = value      ' überschreiben, falls Schlüssel vorhanden
        Else
            Add(i, value)        ' anhängen, falls Schlüssel noch nicht besetzt
        End If
    End Set
End Class
```

```
End Set
End Property
End Class
```

Vergleichen Sie diesen Code mit der ersten Variante, so stellen Sie fest, dass eine Instanziierung von *Hashtable* nicht mehr erforderlich ist. Stattdessen können die benötigten Eigenschaften und Methoden der Basisklasse direkt aufgerufen werden.

Da dank Implementierungsvererbung alle öffentlichen Eigenschaften/Methoden von *Hashtable* jetzt auch in der Schnittstelle von *CPersonen* verfügbar sind, entfällt auch die Implementierungen der *Remove*-Methode.

Bis auf die vereinfachte Klasse *CPersonen* sind, gegenüber der ersten Variante (Aggregation) keine weiteren Unterschiede festzustellen, der Client "sieht" die gleiche Schnittstelle.

Test

Das Ergebnis ist erwartungsgemäß identisch mit Variante 1.

HINWEIS: Damit der mühsam eingegebene Personalbestand das Ausschalten des Rechners überlebt, ist das Abspeichern in eine Datei erforderlich.

3.12.6 Eine Klasse zur Matrizenrechnung entwickeln

Eine Matrix ist nichts weiter wie der mathematische Begriff für ein Array. In diesem Beispiel soll am Beispiel einer Klasse *CMatrix* die grundlegende Vorgehensweise bei der Entwicklung einer Klasse erläutert werden, die schon etwas anspruchsvoller ist als z.B. eine triviale *CPerson*-Klasse.

Die Schwerpunktthemen sind:

- überladener Konstruktor
- überladene Methoden
- Eigenschaftsmethoden
- Standardeigenschaft als 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 Programmier Techniken im Zusammenhang mit dem Array-Zugriff 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 _rows, _cols As Integer
    Private _array(,) As Double
```

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.

```
Sub New()
    MyBase.New()
    _rows = 1
    _cols = 1
    ReDim _array(_rows, _cols)
End Sub

Sub New(R As Integer, C As Integer) ' überladener Konstruktor
    MyBase.New()
    _rows = R
    _cols = C
    ReDim _array(_rows, _cols)
End Sub
```

Der Zugriff auf die (privaten) Zustandsvariablen *_rows* und *_cols* wird über die Eigenschaften *Rows* und *Cols* gekapselt.

```
Public Property Rows() As Integer ' Eigenschaft zum Zugriff auf Zeilenanzahl
    Get
        Return _rows
    End Get
    Set(Value As Integer)
        _rows = Value
    End Set
End Property

Public Property Cols() As Integer ' Eigenschaft zum Zugriff auf Spaltenanzahl
    Get
        Return _cols
    End Get
```

```

    Set(Value As Integer)
        _cols = Value
    End Set
End Property

```

Der Zugriff auf ein bestimmtes Matrix-Element wird elegant über die Standardeigenschaft realisiert, die hier quasi wie ein Indexer funktioniert:

```

Default Public Property Cell(row As Integer, col As Integer) As Double
    Get
        Return _array(row, col)
    End Get
    Set(Value As Double)
        _array(row, col) = Value
    End Set
End Property

```

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 Overloads Shared Function Add(A As CMatrix, B As CMatrix) As CMatrix
    If Not (A.Rows = B.Rows And A.Cols = B.Cols) Then
        Add = New CMatrix()
        Exit Function
    End If
    Dim newMatrix As New CMatrix(A.Rows, A.Cols)
    For row As Integer = 0 To A.Rows - 1
        For col As Integer = 0 To A.Cols - 1
            newMatrix(row, col) = A(row, col) + B(row, col)
        Next
    Next
    Return newMatrix
End Function

```

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 Overloads Function Add(A As CMatrix) As CMatrix
    If Not (A.Rows = MyClass.Rows And A.Cols = MyClass.Cols) Then
        Return New CMatrix()
        Exit Function
    End If
    Dim newMatrix As New CMatrix(MyClass.Rows, MyClass.Cols)

```

```

    For row As Integer = 0 To MyClass.Rows - 1
        For col As Integer = 0 To MyClass.Cols - 1
            newMatrix(row, col) = A(row, col) + MyClass.Cell(row, col)
        Next
    Next
    Return newMatrix
End Function

```

End Class

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*:

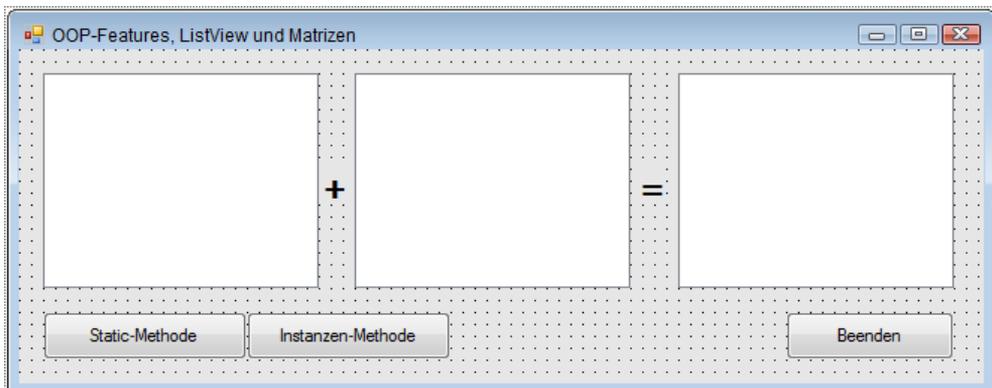
```

Dim A, B, C As CMatrix
...
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 Schaltflächen. Setzen Sie folgende zwei Eigenschaften für jede *ListView*: *View = Details* und *GridLines = True*.



Quellcode von Form1

```
Public Class Form1
```

Wir verwenden für beide Schaltflächen einen gemeinsamen Eventhandler:

```

Private Sub Button_Click(sender As Object, e As EventArgs) _
    Handles Button1.Click, Button2.Click

```

Unser Beispiel benutzt Matrizen mit 9 Zeilen und 6 Spalten:

```

Const rows As Integer = 9 ' Anzahl Zeilen
Const cols As Integer = 6 ' Anzahl Spalten

```

Zufallszahlengenerator instanziiieren:

```
Dim rnd As New System.Random()
```

Die Matrix *A* instanziiieren, mit Zufallszahlen füllen und anzeigen (man beachte den bequemen Zugriff über den Indexer!):

```
Dim A As New CMatrix(rows - 1, cols - 1)
For i As Integer = 0 To A.Rows - 1
    For j As Integer = 0 To A.Cols - 1
        A(i, j) = rnd.Next(100)      ' Zugriff auf Matricelement über Indexer!
    Next
Next
showListView(A, ListView1)        ' Anzeige in linker ListView
```

Gleiches geschieht mit Matrix *B*:

```
Dim B As New CMatrix(rows - 1, cols - 1)
For i As Integer = 0 To B.Rows - 1
    For j As Integer = 0 To B.Cols - 1
        B(i, j) = rnd.Next(100)
    Next
Next
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!

```
Dim C As CMatrix
If CType(sender, Button) Is Button1 Then
    C = CMatrix.Add(A, B)          ' Aufruf Shared-Methode
Else
    C = A.Add(B)                  ' Aufruf Instanzen-Methode
End If
showListView(C, ListView3)       ' Anzeige in rechter ListView
End Sub
```

Der Anzeigeroutine *showListView* werden ein *CMatrix*-Objekt und eine *ListView*-Komponente übergeben:

```
Private Sub showListView(M As CMatrix, lv As ListView)
    With lv
        .Clear()
```

Alle Spalten erzeugen und beschriften:

```
.Columns.Add("", 20, HorizontalAlignment.Right)    ' linke (leere) Randspalte
For j As Integer = 0 To M.Cols - 1
```

Spaltennummerierung und Formatierung in Kopfzeile:

```
.Columns.Add(j.ToString, 30, HorizontalAlignment.Right)
Next
```

Alle Zeilen erzeugen, beschriften und Zellen füllen:

```
For i As Integer = 0 To M.Rows - 1
```

Pro Zeile ein *ListViewItem*, Zeilennummerierung in linke Randspalte eintragen:

```
Dim item As New ListViewItem(i.ToString)
For j As Integer = 0 To M.Cols - 1
```

Alle Zellen füllen (pro Zelle ein *SubItem*):

```
item.SubItems.Add(M(i, j).ToString)
Next
```

Zeile zur *ListView* hinzufügen:

```
.Items.Add(item)
Next
End With
End Sub
End Class
```

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 Summenmatrix mit dem richtigen Ergebnis gefüllt:

The screenshot shows a Windows application window with the title "OOP-Features, ListView und Matrizen". It displays three 5x5 matrices. The first two matrices are separated by a plus sign (+), and the third matrix is separated by an equals sign (=). Below the matrices are three buttons: "Static-Methode", "Instanzen-Methode", and "Beenden".

	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

Bemerkung

Das Resultat einer Matrix-Operation ist immer eine neue Matrix, wenn allerdings beide Matrizen inkompatibel sind, wird eine leere Matrix zurückgegeben. Alternativ könnte in einem solchen Fall auch eine Ausnahme innerhalb der Methoden erzeugt werden (*Throw New System.ArgumentException()*).

3.12.7 Rechner für komplexe Zahlen

Auch mit dieser Anwendung 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 Visual Basic,
- 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.

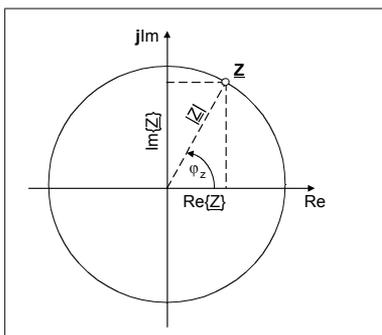
Zur Darstellung einer komplexen Zahl \underline{Z} bieten sich zwei Möglichkeiten an:

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

Die folgende Tabelle zeigt eine Zusammenstellung der Umrechnungsformeln:

Kartesische Koordinaten	Polarkoordinaten
$\underline{Z} = \text{Re}\{\underline{Z}\} + j\text{Im}\{\underline{Z}\}$	$\underline{Z} = \underline{Z} e^{j\varphi_z}$
Realteil: $\text{Re}\{\underline{Z}\} = \underline{Z} \cos \varphi_z$	Betrag: $ \underline{Z} = \sqrt{(\text{Re}\{\underline{Z}\})^2 + (\text{Im}\{\underline{Z}\})^2}$
Imaginärteil: $\text{Im}\{\underline{Z}\} = \underline{Z} \sin \varphi_z$	Phasenwinkel: $\varphi_z = \arctan \frac{\text{Im}\{\underline{Z}\}}{\text{Re}\{\underline{Z}\}}$

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

$$\mathbf{Z}_1 = \mathbf{a}_1 + \mathbf{j}\mathbf{b}_1 \quad \text{und} \quad \mathbf{Z}_2 = \mathbf{a}_2 + \mathbf{j}\mathbf{b}_2$$

ergibt sich

$$\mathbf{Z}_1 + \mathbf{Z}_2 = \mathbf{a}_1 + \mathbf{a}_2 + \mathbf{j}(\mathbf{b}_1 + \mathbf{b}_2) \quad \text{bzw.} \quad \mathbf{Z}_1 - \mathbf{Z}_2 = \mathbf{a}_1 - \mathbf{a}_2 + \mathbf{j}(\mathbf{b}_1 - \mathbf{b}_2)$$

Andererseits bevorzugt man für Multiplikation und Division die Zeigerform:

Mit

$$\mathbf{Z}_1 = \mathbf{c}_1 \cdot \mathbf{e}^{\mathbf{j}\varphi_1} \quad \text{und} \quad \mathbf{Z}_2 = \mathbf{c}_2 \cdot \mathbf{e}^{\mathbf{j}\varphi_2}$$

erhalten wir

$$\mathbf{Z}_1 \cdot \mathbf{Z}_2 = \mathbf{c}_1 \cdot \mathbf{c}_2 \cdot \mathbf{e}^{\mathbf{j}(\varphi_1 + \varphi_2)} \quad \text{bzw.} \quad \mathbf{Z}_1 / \mathbf{Z}_2 = \mathbf{c}_1 / \mathbf{c}_2 \cdot \mathbf{e}^{\mathbf{j}(\varphi_1 - \varphi_2)}$$

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

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

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 wird¹.

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.

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

```
Public Class CComplexN
```

Die beiden öffentlichen Zustandsvariablen *Re* und *Im* bilden das "Gedächtnis" der Klasse und können quasi wie Eigenschaften benutzt werden¹:

```
Public Re, Im As Double          ' Real- und Imaginärteil
```

Ein Konstruktor (den Rahmencode können Sie sich von der IDE erzeugen lassen) setzt die Zustandsvariablen auf ihre Anfangswerte:

```
Public Sub New(r As Double, i As Double)
    Re = r : Im = i
End Sub
```

Die "intelligente" Eigenschaftsmethode *Ang* berechnet den Phasenwinkel aus den Zustandsvariablen *Re* und *Im*:

```
Public Property Ang() As Double
    Get
        Dim g As Double = 0
        If Re <> 0 Then
            g = 180 / Math.PI * Math.Atan(Im / Re)
        If Re < 0 Then g += 180
        Else
            If Im <> 0 Then
                If Im > 0 Then g = 90
            Else
                g = -90
            End If
        End If
        Return g
    End Get
    Set(value As Double)
        Dim b, l As Double
        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
    End Set
End Property
```

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

```
Public Property Len() As Double
    Get
        Return Math.Sqrt(Re * Re + Im * Im)
    End Get
    Set(value As Double)
        Dim b As Double = Math.Atan(Im / Re)
```

¹ Die Verwendung öffentlicher Zustandsvariablen als Eigenschaften ist zwar nicht der sauberste, in unserem Fall aber der effektivste Weg.

```

        Re = value * Math.Cos(b)
        Im = value * Math.Sin(b)
    End Set
End Property

```

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 Shared Operator +(a As CComplexN, b As CComplexN) As CComplexN
    Dim z As New CComplexN(0, 0)
    z.Re = a.Re + b.Re
    z.Im = a.Im + b.Im
    Return z
End Operator

```

Der "*" -Operator multipliziert zwei komplexe Zahlen:

```

Public Shared Operator *(a As CComplexN, b As CComplexN) As CComplexN
    Dim z As 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
End Operator

```

Der "/"-Operator dividiert zwei komplexe Zahlen:

```

Public Shared Operator /(a As CComplexN, b As CComplexN) As CComplexN
    Dim z As 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
End Operator
End Class

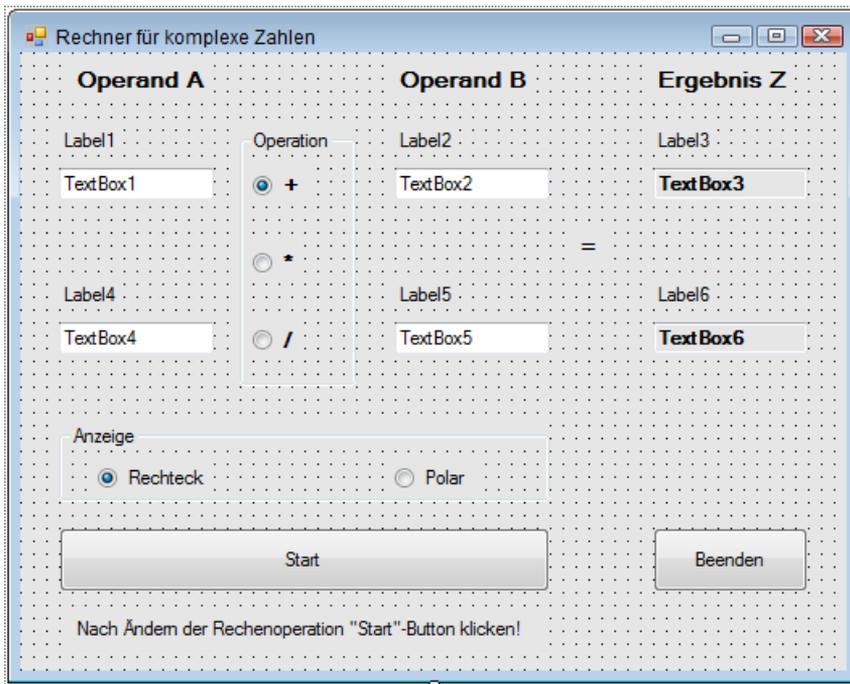
```

HINWEIS: Vielleicht sticht Ihnen bereits jetzt ein gravierender Unterschied zur klassischen "Geradeausprogrammierung" ins Auge: Spezielle Methoden zur Umrechnung zwischen Kartesischen- und Polarkoordinaten sind fehlangezeigt, 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 Geradeausprogrammierung, 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 Class Form1
```

Die benötigten Objektvariablen:

```
Private A As New CComplexN(1, 1) ' Operand A
Private B As New CComplexN(1, 1) ' Operand B
Private Z As New CComplexN(0, 0) ' Ergebnis Z
```

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

```

Private Sub Eingabe()
    If RadioButton4.Checked Then ' 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)
    End If
End Sub

```

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 Sub Verarbeitung()
    If RadioButton1.Checked Then Z = A + B ' Addition
    If RadioButton2.Checked Then Z = A * B ' Multiplikation
    If RadioButton3.Checked Then Z = A / B ' Division
End Sub

```

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 Sub Anzeige()
    If RadioButton4.Checked Then ' 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"
    End If
End Sub

```

```

        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
    End If
End Sub

```

Die *Start*-Schaltfläche:

```

Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    Eingabe()
    Verarbeitung()
    Anzeige()
End Sub

```

Nach Umschaltung zwischen Rechteck- und Polarkoordinaten muss die Anzeige aktualisiert werden:

```

Private Sub RadioButton4_CheckedChanged(sender As Object,
    e As EventArgs) Handles RadioButton4.CheckedChanged
    Anzeige()
End Sub
...

```

Programmtest

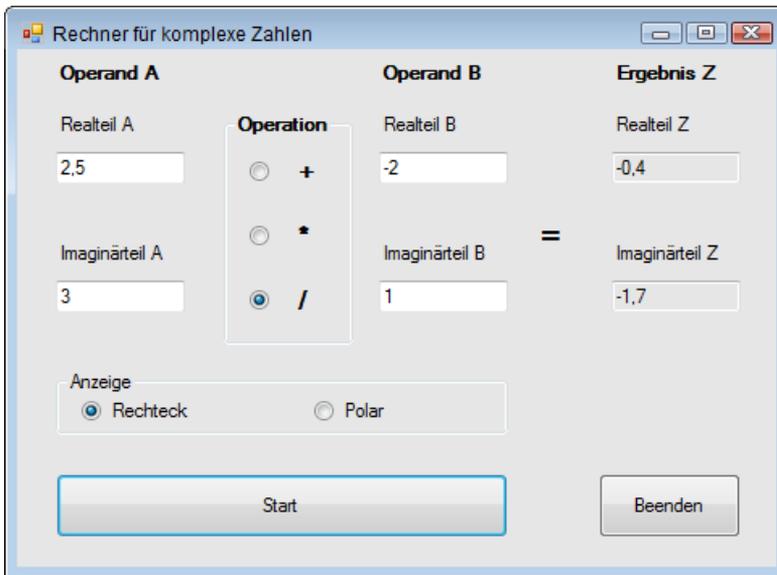
Wenn zum Beispiel die Aufgabe

$$(2.5 + 3j) / (-2 + j)$$

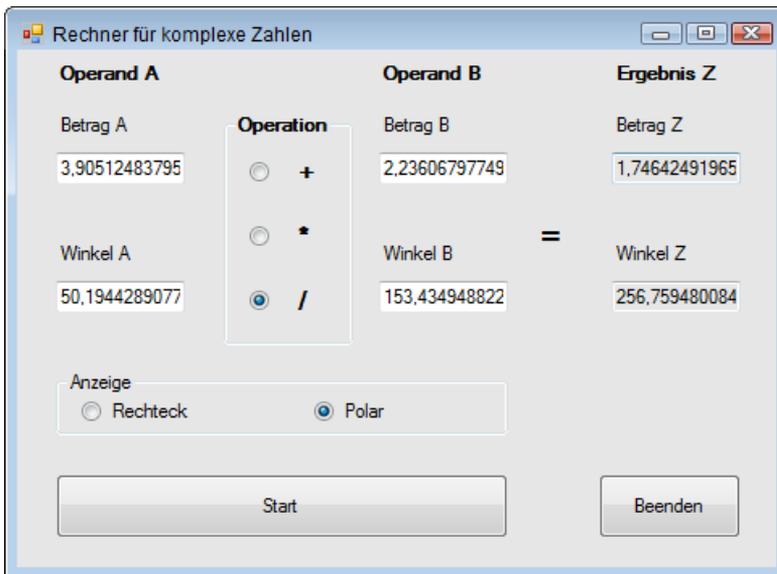
gelöst werden soll, so stellen Sie zunächst die Anzeige auf "Rechteck" ein. Geben Sie dann links oben den Realteil (2,5) und darunter den Imaginärteil (3) des ersten Operanden ein. Analog dazu geben Sie rechts oben den Realteil (-2) und darunter den Imaginärteil (1) des zweiten Operanden ein. Abschließend klicken Sie auf die gewünschte Operation (/).

Nach Betätigen der *Start*-Taste erscheint als Ergebnis die komplexe Zahl $-0.4 - 1.7j$ (siehe folgende Abbildung).

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.



HINWEIS: Wenn Sie die Anzeige zwischen Rechteck- und Polarkoordinaten umgeschaltet haben, müssen Sie anschließend die *Start*-Schaltfläche klicken!



Bemerkungen

- Die Vorteile eines gut strukturierten, objektorientierten Programms liegen bekanntermaßen in der leichteren Lesbarkeit des Quellcodes ("sprechender" Code) und in der besseren Wartbarkeit und Erweiterungsfähigkeit.
- Beim Arbeiten mit Visual Studio informiert Sie die Intellisense stets aktuell über die vorhandenen Klassenmitglieder und deren Signaturen.
- Bereits mit .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.

3.12.8 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 Formel-ausdrücken durchzuführen, sei es um eine Werteliste zu erstellen oder um eine Diagramm auszu-drucken.

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!

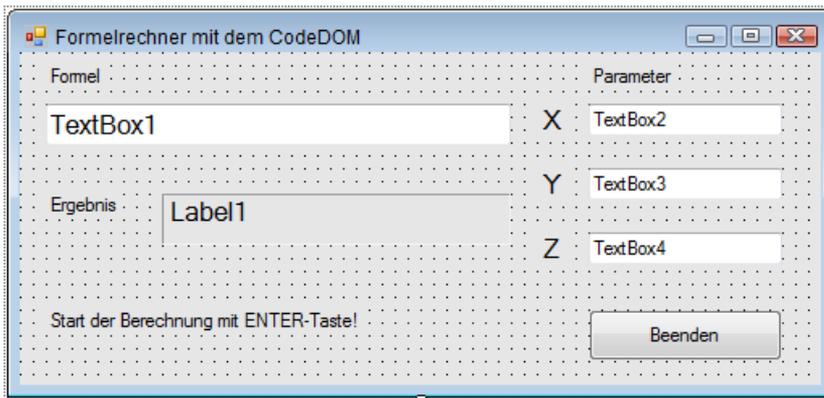
In welcher Sprache Sie die zu berechnende Formel zusammenbauen ist egal, Voraussetzung ist lediglich das Vorhandensein eines zum Compiler passenden *CodeDomProviders*. Im vorliegenden Fall haben wir uns, wen wundert's, für VB entschieden. Wir hätten aber auch den C#-Provider nehmen können, für den Endanwender ist aber die VB-Syntax einfacher zu verstehen, scheitern doch viele bereits an der peniblen 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 mit folgendem Aussehen:



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, wie sie bei "normalen" Taschenrechnern üblich ist. Stattdessen werden wir die Berechnung durch einfaches Betätigen der Enter-Taste starten.

Außerdem gönnen wir uns noch drei weitere *TextBoxen*, um auch Parameter in die Formel einbauen zu können (es empfiehlt sich, dazu die *KeyPreview*-Eigenschaft des Formulars auf *True* zu setzen).

Die Klasse CCalculator

Fügen Sie zum Projekt eine neue Klasse mit dem Namen *C 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.

```
Imports System.CodeDom.Compiler
Imports System.Reflection
```

```
Public Class CCalculator
```

Zwischenspeichern der Assembly und der Verweise:

```
Private Shared ass As Assembly
Private Shared aClass As Type
Private Shared aMethode As MethodInfo
Private Shared obj As Object
```

Der Berechnungsfunktion wird der Formelausdruck als String übergeben:

```
Public Shared Function Calc(expr As String) As Double
    If expr.Length = 0 Then Return 0.0
```

Im Formelausdruck werden die Dezimalkommas durch Dezimalpunkte ersetzt:

```
    expr = expr.Replace(",", ".")
```

Compilerparameter definieren:

```
    Dim opt As New CompilerParameters(Nothing, 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 Namespace schreiben:

```
    Dim src As String = "Imports System.Math" & vbCrLf &
        "Public Class Calculate" & vbCrLf &
        "    Public Function Calc() As Double" & vbCrLf &
        "        Return " & expr & vbCrLf &
        "    End Function" & vbCrLf &
        "End Class" & vbCrLf
```

Nun kann unser VB-Quellcode kompiliert werden:

```
    Dim res As CompilerResults = New VBCodeProvider().CompileAssemblyFromSource(opt, src)
```

Auf eine Fehlerauswertung sollte nicht verzichtet werden:

```
    If res.Errors.Count > 0 Then
        Dim errors As String = String.Empty
        For Each cerr As CompilerError In res.Errors
            errors = errors & cerr.ToString() & vbCrLf
        Next
        ass = Nothing
        expr = String.Empty
        Throw New ApplicationException(errors)
    End If
```

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")
```

Jetzt kommen wir auch an die interne *Calc*-Methode heran:

```
aMethod = aClass.GetMethod("Calc")
```

Eine Instanz der internen Klasse erzeugen, die interne *Calc*-Methode aufrufen und das Ergebnis zurück liefern:

```
obj = Activator.CreateInstance(aClass)
Return Convert.ToDouble(aMethod.Invoke(obj, Nothing))
End Function
```

```
End Class
```

Quellcode Form1

```
Public Class Form1
```

Die zentrale Anlaufstelle nach Änderung der Eingabewerte ist der Aufruf dieser Methode:

```
Private Sub Berechnung()
```

Den Formelausdruck zuweisen:

```
Dim str As String = 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 *C 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
    Dim res As Double = CCalculator.Calc(str)
    str = res.ToString()
```

Um das Dezimaltrennzeichen einheitlich als Punkt darzustellen, wandeln wir im Ergebnisstring das Komma einfach in einen Punkt um:

```
Label1.Text = str.Replace(",", ".")
Catch ex As Exception
    Label1.Text = String.Empty
    MessageBox.Show(ex.Message)
End Try
End Sub
```

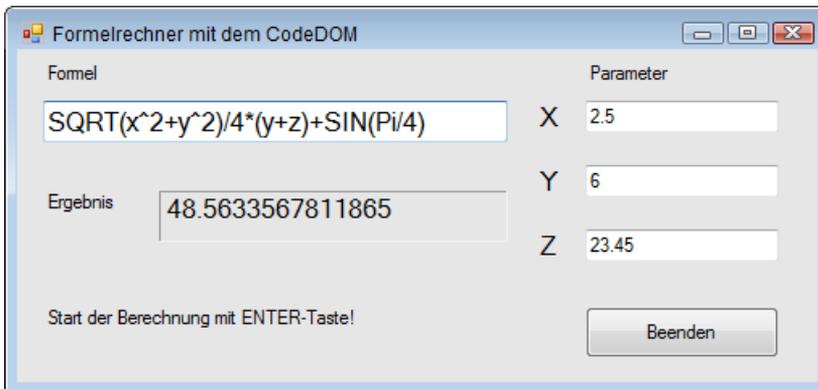
Alle vier *TextBox*en verwenden den folgenden gemeinsamen *KeyPress*-Eventhandler, er sorgt dafür, dass die Berechnung mittels Enter-Taste gestartet wird:

```
Private Sub TextBox_KeyPress(sender As Object, e As KeyPressEventArgs) _
    Handles TextBox1.KeyPress, TextBox4.KeyPress, TextBox3.KeyPress, TextBox2.KeyPress
    If e.KeyChar = ChrW(Keys.Enter) Then
        Berechnung()
        e.Handled = True
    End If
End Sub
...
End Class
```

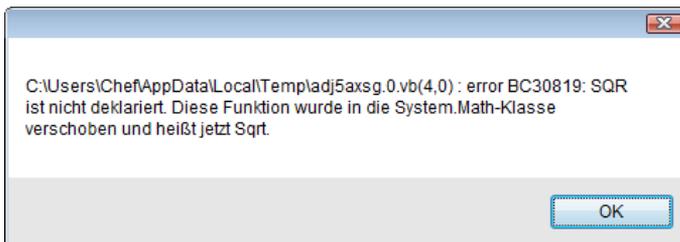
Test

Geben Sie einen beliebig komplizierten bzw. verschachtelten arithmetischen Ausdruck ein (mit oder ohne Parameter x, y, z). Grundlage ist die VB-Syntax, d.h. auch, die Groß-/Kleinschreibung ist ohne Bedeutung.

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 *C 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 instanziiieren. Bei der Eingabe der Berechnungsformel wäre dann penibel auf die Groß-/Kleinschreibung zu achten.

3.12.9 Einen Funktionsverlauf grafisch 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 Verfahren an seine Grenzen, denn das wiederholte Kompilieren und Auswerten würde auch bei einem schnellen Rechner unzumutbar viel Zeit verbrauchen.

Da es nicht sinnvoll ist, bei geänderten Parametern aber gleichem Formel Ausdruck 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 dabei keine Rolle).

Die Klasse C CalculatorX

Die Änderungen gegenüber der Klasse *C Calculator* sind durch Fettdruck hervorgehoben:

```
Imports System.CodeDom.Compiler
Imports System.Reflection

Public Class C CalculatorX
    Private Shared ass As Assembly
    Private Shared aClass As Type
    Private Shared aMethod As MethodInfo
    Private Shared obj As Object
```

Private Shared expression As String

```
Public Shared Function Calc(expr As String, x As Double) As Double
    If expr.Length = 0 Then Return 0.0
    expr = expr.Replace(",", ".")
```

Ein erneutes Kompilieren ist nur dann erforderlich, wenn sich der Formelausdruck *expr* geändert hat:

```
    If expr <> expression Then
        expression = expr
        Dim opt As New CompilerParameters(Nothing, String.Empty, False)
        opt.GenerateExecutable = False
        opt.GenerateInMemory = True

        Dim src As String = "Imports System.Math" & vbCrLf &
            "Public Class Calculate" & vbCrLf &
            "    Public Function Calc(X As Double) As Double" & vbCrLf &
            "        Return " & expr & vbCrLf &
            "    End Function" & vbCrLf &
            "End Class" & vbCrLf

        Dim res As CompilerResults = New VBCodeProvider().CompileAssemblyFromSource(
            opt, src)

        If res.Errors.Count > 0 Then
            Dim errors As String = String.Empty
            For Each cerr As CompilerError In res.Errors
                errors = errors & cerr.ToString() & vbCrLf
            Next
            ass = Nothing
            expr = String.Empty
            Throw New ApplicationException(errors)
        End If
        ass = res.CompiledAssembly
        aClass = ass.GetType("Calculate")
        aMethod = aClass.GetMethod("Calc")
        obj = Activator.CreateInstance(aClass)
    End If
```

Falls sich *expr* nicht geändert hat, genügt die Übergabe des neuen Wertes für den Parameter *x*:

```
        Return Convert.ToDouble(aMethod.Invoke(obj, New Object() {x}))
    End Function
```

```
End Class
```

Quellcode Form1

```
Public Class Form1
```

Wir überschreiben die *OnPaint*-Methode des Formulars, dadurch wird auch nach vorübergehendem Verdecken des Fensters das Diagramm automatisch neu erstellt:

```
Protected Overrides Sub OnPaint(e As System.Windows.Forms.PaintEventArgs)
```

Um den Quellcode überschaubar zu halten, haben wir die Einstellungen des Koordinatensystems für das Diagramm der Funktion $y = \sin(x)/x$ optimiert:

```
Dim x As Single = -40
Dim y As Single = 0
Dim xold As Single = x
Dim yold As Single = y
Dim scalex As Single = Me.Width / (2 * x)
Dim scaley As Single = -(Me.Height - 20) / 2
Dim incr As Single = 80.0F / Me.Width
Dim p As New Pen(Color.Black, 2)
```

Graphics-Objekts, auf welches das Diagramm gezeichnet werden kann:

```
Dim g As Graphics = e.Graphics
```

Koordinatensystem verschieben:

```
g.TranslateTransform(Me.Width / 2, Me.Height / 2)
```

Koordinatenachsen zeichnen:

```
g.DrawLine(Pens.Red, -Me.Width, 0, Me.Width, 0)
g.DrawLine(Pens.Red, 0, -Me.Height, 0, Me.Height)
```

Anzeige optimieren:

```
g.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.AntiAlias
```

Die einzelnen Funktionswerte berechnen und zeichnen:

```
For i As Integer = 0 To Me.Width - 1
    If x = 0 Then 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
Next
```

Die folgende Anweisung nicht vergessen, sonst wird *Paint*-Event nicht ausgelöst:

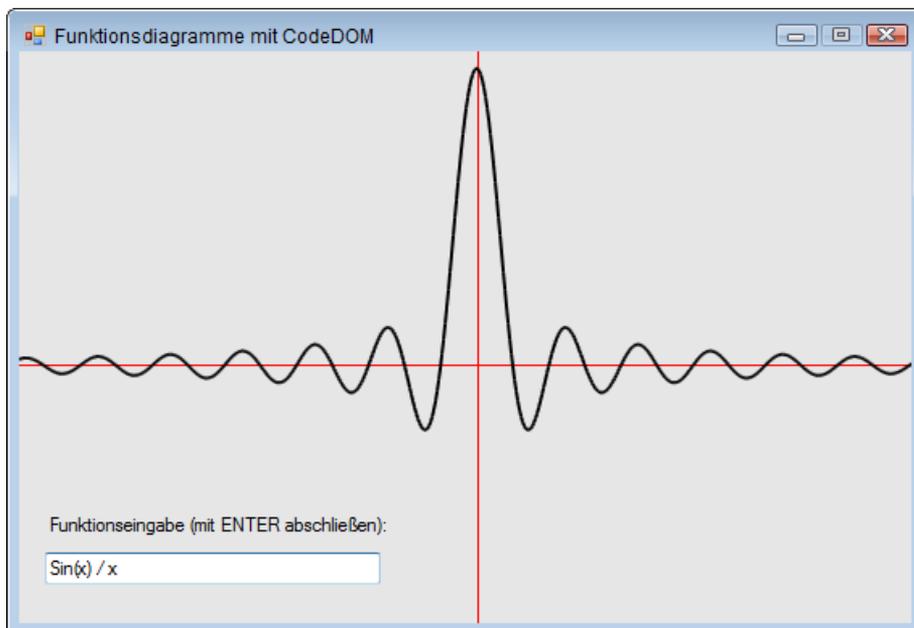
```
MyBase.OnPaint(e)
End Sub
```

Der folgende Eventhandler sorgt für die Umwandlung eines Dezimalkommas in einen Dezimalpunkt und für das Neuzeichnen des Formulars nach Betätigen der Enter-Taste:

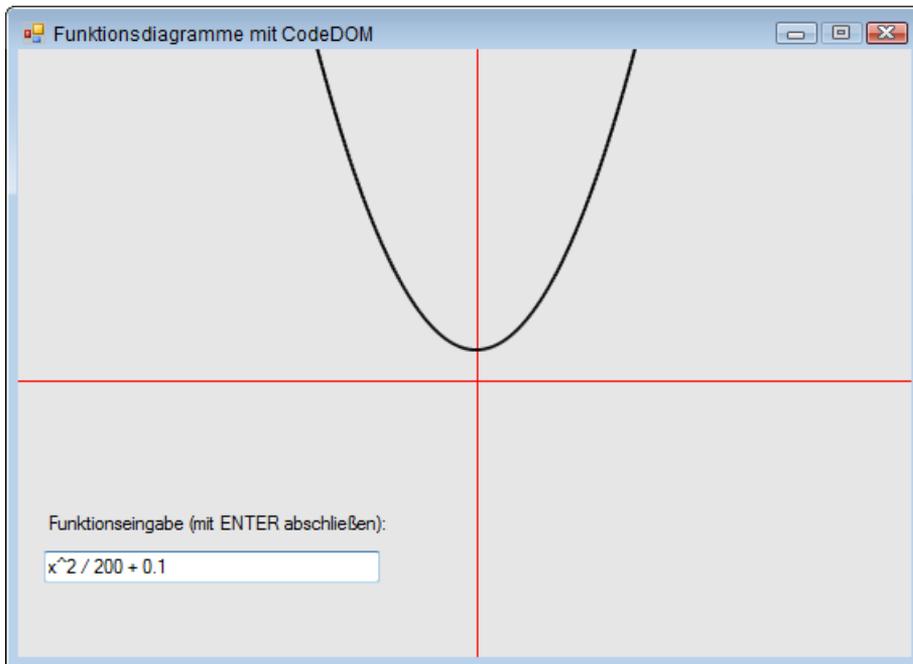
```
Private Sub TextBox1_KeyPress(sender As Object, e As KeyPressEventArgs) _
    Handles TextBox1.KeyPress
    If e.KeyChar = "," Then
        e.KeyChar = "."
    Else
        If e.KeyChar = ChrW(Keys.Enter) Then
            Me.Refresh()
            e.Handled = True
        End If
    End If
End Sub
End Class
```

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 Formel Ausdruck 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 *C CalculatorX* hat gegenüber der Klasse *C 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 *C 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.

3.12.10 Sortieren mit *IComparable/IComparer*

Haben Sie eine *ArrayList* oder eine generische *List(Of T)* 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
  Implements IComparable(Of CStudent)

  Public Property Nummer As Integer
  Public Property Vorname As String
  Public Property Nachname As String

  Public Sub New(numm As Integer, vor As String, nach As String)
    Nummer = numm
    Vorname = vor
    Nachname = nach
  End Sub
```

Die alphabetische Sortierung nach dem Nachnamen wird als Standardvergleich festgelegt:

```
Public Function CompareTo(stud As CStudent) As Integer _
  Implements IComparable(Of CStudent).CompareTo
  Return Me.Nachname.CompareTo(stud.Nachname)
End Function
```

Um die Datenbindung zu vereinfachen überschreiben wir die *ToString*-Methode:

```
Public Overrides Function ToString() As String
  Return Nummer.ToString() & " , " & Nachname & " , " & Vorname
End Function
End Class
```

MainWindow.xaml.vb

Im Hauptfenster implementieren wir zunächst eine Methode, die eine Studentenliste erzeugt und initialisiert:

```
Class MainWindow

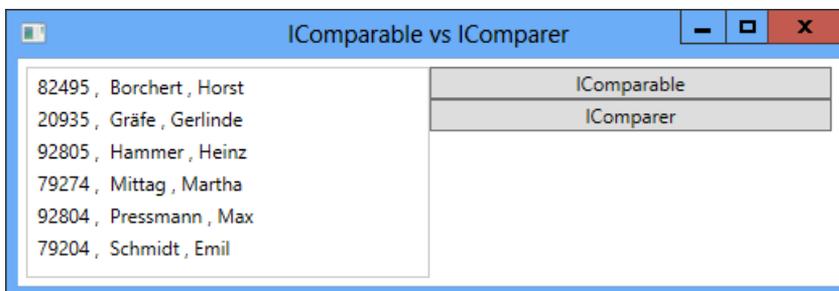
    Private Function getStudenten() As List(Of CStudent)
        Dim studenten As New List(Of CStudent)()
        studenten.Add(New CStudent(82495, "Horst", "Borchert"))
        studenten.Add(New CStudent(20935, "Gerlinde", "Gräfe"))
        studenten.Add(New CStudent(79274, "Martha", "Mittag"))
        studenten.Add(New CStudent(79204, "Emil", "Schmidt"))
        studenten.Add(New CStudent(92804, "Max", "Pressmann"))
        studenten.Add(New CStudent(92805, "Heinz", "Hammer"))
        Return studenten
    End Function
```

Im *Click-Event* des ersten *Buttons* wird die Liste (entsprechend der Implementierung von *CompareTo* in *CStudent*) sortiert und angezeigt:

```
Private Sub Button_Click_1(sender As Object, e As RoutedEventArgs)
    Dim studenten = getStudenten()
    studenten.Sort()
    lb.DataContext = studenten
End Sub
...
```

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(Of 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 Studentenummer 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(Of T)* bereit, welches über eine Methode *Compare()* verfügt, die zwei Objektparameter X, Y entgegennimmt und ein Integer zurückgibt.

Klasse CSortNummer

Fügen Sie zum Projekt eine Klasse *CSortNummer* hinzu, welche das *IComparer*-Interface implementiert:

```
Public Class CSortNummer
    Implements IComparer(Of CStudent)
```

Eine Zustandsvariable bestimmt die Sortierung in aufsteigender (*True*) bzw. absteigender (*False*) Folge:

```
Private _auf As Boolean
```

Wir übergeben im Konstruktor einen Parameter, der die gewünschte Sortierfolge spezifiziert.

```
Public Sub New(auf As Boolean)
    _auf = auf
End Sub
```

Die Implementierung des Sortierkriteriums erfolgt in der *Compare*-Methode:

```
Public Function Compare(a As CStudent, b As CStudent) As Integer _
    Implements IComparer(Of CStudent).Compare

    If _auf Then ' aufsteigend
        If (a.Nummer > b.Nummer) Then
            Return 1
        ElseIf a.Nummer < b.Nummer Then
            Return -1
        Else
            Return 0
        End If
    Else ' absteigend
        If a.Nummer < b.Nummer Then
            Return 1
        ElseIf a.Nummer > b.Nummer Then
            Return -1
        Else
            Return 0
        End If
    End If
```

```

End If
End Function
End Class

```

Ergänzung MainWindow.xaml.vb

Wir ergänzen den Codebehind, um die zweite Sortiervariante zu testen:

```

...
Private Sub Button_Click_2(sender As Object, e As RoutedEventArgs)
    Dim studenten = getStudenten()

```

Wir wollen die Sortierfolge umkehren:

```

    Dim son As New CSortNumer(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 *CSortNumer* definierte zusätzliche Sortierung, die wir hier ausgewählt haben.

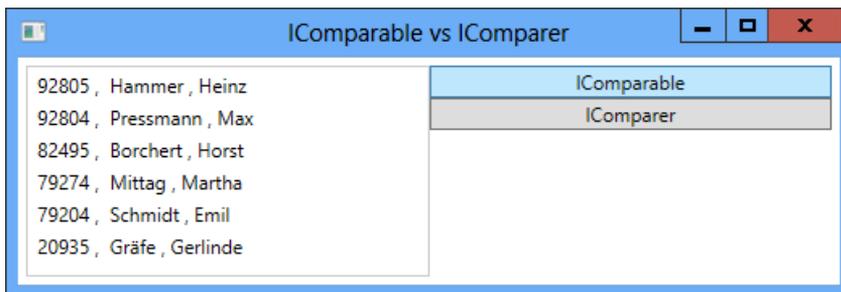
```

    Tb.DataContext = studenten
End Sub
...

```

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 zusätzliches Sortierkriterium ist eine Klasse erforderlich, die das *IComparer*-Interface (*Compare*-Methode) implementiert.
- *IComparer* funktioniert auch dann, wenn keine Standardsortierung existiert, also wenn *IComparable* nicht implementiert ist.

3.12.11 Objektbäume 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.

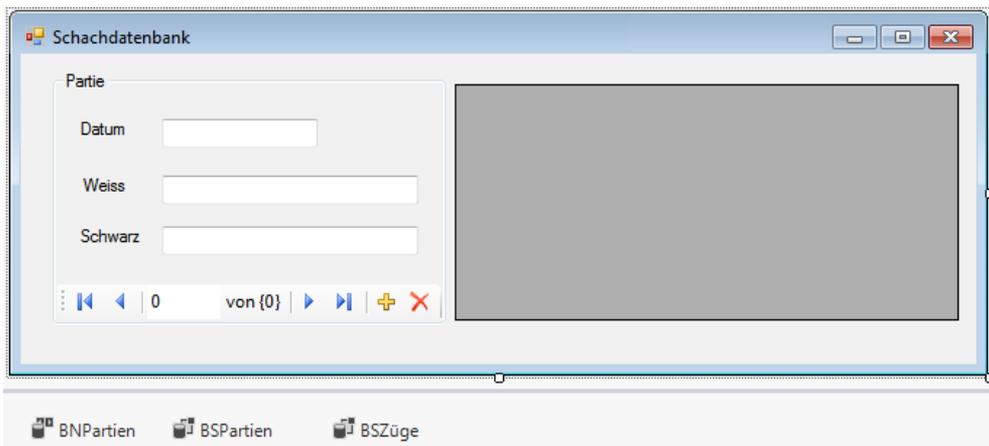
Wir demonstrieren dies am Beispiel einer ausbaufähigen Schachdatenbank, in der Sie beliebig viele Schachpartien abspeichern können¹.

HINWEIS: Das vorliegende Beispiel verwendet 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 Beispiels zu verstehen, muss man kein Schachspieler sein.

Die Klasse CZug

Diese Klasse stellt die Blattebene unseres Objektbaums dar und repräsentiert einen Doppelzug nebst Kommentar (einfachheitshalber sind alle Eigenschaften selbst implementierend):

```
<Serializable> Public Class CZug
    Public Property Weiss As String      ' z.B. "e4" oder Le7, Se6, 0-0, Lb4+, c:d5, S:a4, ...
    Public Property Schwarz As String
    Public Property Kommentar As String ' z.B. "schwerer Fehler!"
End Class
```

Die Klasse CPartie

```
<Serializable> Public Class CPartie
```

Eine generische Liste speichert alle Züge einer Partie:

```
    Private _züge As List(Of CZug)
```

Im Konstruktor wird die (zunächst leere) Zugliste erstellt:

```
    Sub New()
        _züge = New List(Of CZug)
    End Sub
```

Der Zugriff auf die Zugliste:

```
    Public Property Züge As IList(Of CZug)
        Get
            Return _züge
        End Get
        Set(value As IList(Of CZug))
            _züge = value
        End Set
    End Property
```

Einige selbst implementierende Eigenschaften:

```
    Public Property Datum As DateTime      ' Datum der Partie
    Public Property Weiss As String        ' Name des Weiss-Spielers
    Public Property Schwarz As String      ' Name des Schwarz-Spielers
End Class
```

Die Klasse CSchachDB

Dies ist die Wurzelklasse unseres Objektbaums.

```
<Serializable> Public Class CSchachDB
```

Die folgende (generische) Liste speichert alle Partien der Datenbank:

```
    Private _partien As IList(Of CPartie)
```

Im Konstruktor wird die (zunächst leere) Partienliste erzeugt:

```
Sub New()
    _partien = New List(Of CPartie)
End Sub
```

Der Zugriff auf die Liste der Partien:

```
Public Property Partien() As IList(Of CPartie) ' Liste der Partien
    Get
        Return _partien
    End Get
    Set(value As IList(Of CPartie))
        _partien = value
    End Set
End Property
```

```
End Class
```

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.

```
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Binary
```

```
Public Class CPersistenz
```

Aufgabe dieser Methode ist es, den kompletten Objektbaum zu serialisieren und als Datei abzuspeichern:

```
Public Shared Sub saveObject(o As Object, pfad As String)
    Dim fs As New FileStream(pfad, FileMode.Create, FileAccess.Write, FileShare.None)
    Dim bf As New BinaryFormatter()
    bf.Serialize(fs, o)
    fs.Close()
End Sub
```

Die folgende Methode lädt die Datei zurück in den Arbeitsspeicher und rekonstruiert den Objektbaum:

```
Public Shared Function loadObject(pfad As String) As Object
    Dim fs As New FileStream(pfad, FileMode.Open, FileAccess.Read, FileShare.Read)
    Dim bf As New BinaryFormatter()
    Dim o As Object = bf.Deserialize(fs)
    fs.Close()
    Return o
End Function
End Class
```

Die Klasse Form1

```
Public Class Form1
```

Der komplette Objektbaum:

```
Private _schachDB As New CSchachDB()
```

Der Standort der Datenbankdatei:

```
Private Const PFAD As String = "SchachDB.dat"
```

Der Programmstart:

```
Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
```

Zunächst wird versucht, die Datenbankdatei zu laden, im Fehlerfall entsteht eine neue leere Datei:

```
Try
    _schachDB = CType(CPersistenz.loadObject(PFAD), CSchachDB)
Catch ex As Exception
    MessageBox.Show(ex.Message)
End Try
```

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:

```
DataGridView1.DataSource = BSZüge
End Sub
```

Erzeugen Sie einen Eventhandler für das *CurrentChanged*-Ereignis der *BindingSource* der Partienliste:

```
Private Sub BSPartien_CurrentChanged(sender As Object, e As EventArgs) _
    Handles BSPartien.CurrentChanged
```

Beim Navigieren zu einer anderen Partie wird die *BindingSource* der Zügeliste umgeklemt:

```
Dim partie As CPartie = CType(BSPartien.Current, CPartie)
BSZüge.DataSource = partie.Züge
End Sub
```

Beim Schließen des Formulars versuchen wir, den Objektbaum in der Datei abzuspeichern:

```
Private Sub Form1_FormClosing(sender As Object, e As FormClosingEventArgs) _
    Handles Me.FormClosing

Try
    CPersistenz.saveObject(_schachDB, PFAD)
Catch ex As Exception
```

```

        MessageBox.Show(ex.Message)
    End Try
End Sub

```

Damit Sie nicht leichtfertig eine komplette Schachpartie mitsamt allen mühsam eingegebenen Zügen löschen, soll nach Klick auf die *Delete*-Schaltfläche des *BindingNavigators* zunächst eine Sicherheitswarnung erscheinen:

```

Private Sub BindingNavigatorDeleteItem2_Click(sender As Object, e As EventArgs) _
    Handles BindingNavigatorDeleteItem2.Click
    If MsgBox("Soll die Partie wirklich gelöscht werden?", MsgBoxStyle.YesNo,
        "Sicherheitsabfrage") = MsgBoxResult.Yes Then
        BSPartien.RemoveCurrent()
    End If
End Sub

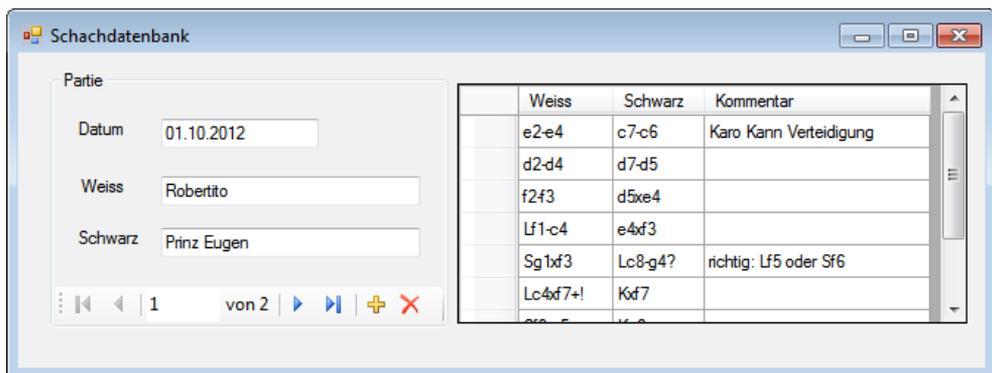
End Class

```

HINWEIS: Obige Methode funktioniert nur dann, wenn Sie die *DeleteItem*-Eigenschaft des *BindingNavigators* 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.



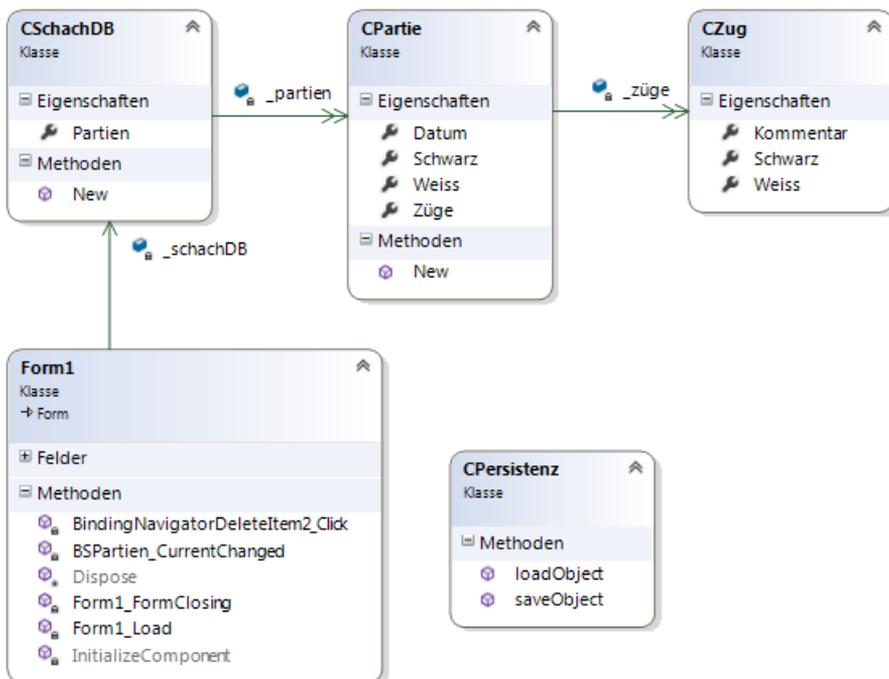
Klicken Sie zunächst auf die "+"-Schaltfläche im *BindingNavigator*, um eine neue Schachpartie mit dem Datum¹ und den Namen der Spieler anzulegen. Danach können Sie auf bekannte Weise im Datengitter nacheinander die einzelnen Züge eintippen.

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 typischer!).

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.



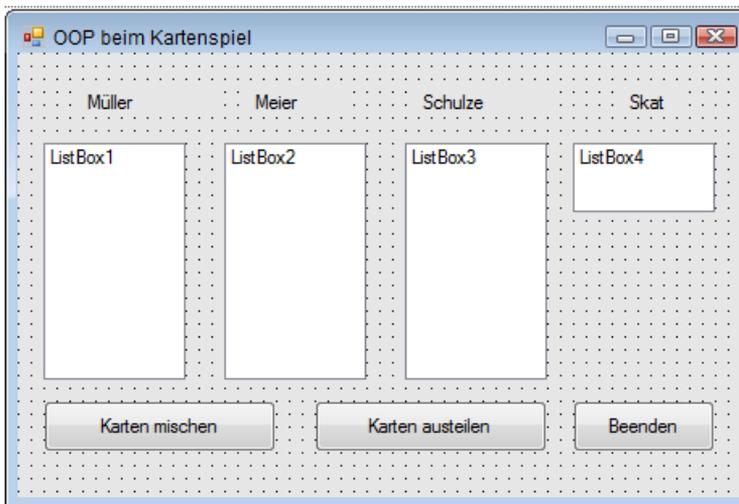
3.12.12 OOP beim Kartenspiel erlernen

Das vorliegende Beispiel soll Ihnen helfen, Ihre OOP-Grundlagen "spielend" 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 dieses Praxisbeispiels 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 Klassen ineinander verschachteln kann. In diesem Sinne implementieren wir innerhalb des Klassencodes von *Form1* die Klassen *CKarte* und *CSpiel*:

```
Public Class Form1
    Public Class CKarte
        Public Farbe As String
        Public Wert As String
```

Der Konstruktor erzeugt ein bestimmtes Kartenobjekt, welches durch Farbe und Wert charakterisiert ist:

```
Public Sub New(f As String, w As String)
```

```

        Farbe = f
        Wert = w
    End Sub
End Class ' von CKarte

```

Die Klasse *CSpiel* kapselt 32 Instanzen der Klasse *CKarte*:

```
Public Class CSpiel
```

Die Eigenschaft *Karten* (das ist ein Array mit 32 Karten!):

```
    Public Karten(32) As CKarte
```

Eine Hilfsmethode soll das Generieren der Karten vereinfachen:

```

    Private Sub createKarten(farbe As String, a As Integer)
        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")
    End Sub

```

Der Konstruktor erzeugt und füllt das Spiel der Reihe nach mit allen 32 Karten:

```

    Public Sub New()
        Call createKarten("Eichel", 1)
        Call createKarten("Grün", 9)
        Call createKarten("Rot", 17)
        Call createKarten("Schell", 25)
    End Sub

```

Die Methode zum Mischen der Karten:

```

    Public Sub mischen()
        Dim z, i As Integer

```

Der Zwischenspeicher für den Kartentausch:

```

        Dim tmp As CKarte
        Dim rnd As New Random()

```

Alle Karten nacheinander durchlaufen:

```
        For i = 1 To Karten.Length - 1
```

Den Index einer zufälligen anderen Karte bestimmen:

```
            z = rnd.Next(1, Karten.Length)
```

Die zufällige Karte mit der aktueller Karte vertauschen:

```

            tmp = Karten(z)
            Karten(z) = Karten(i)

```

```

        Karten(i) = tmp
    Next
End Sub
End Class
' von CSpiel

```

HINWEIS: Aus Gründen der Einfachheit bleibt der Index 0 des *Karten*-Arrays ungenutzt.

Nun kommen wir zum eigentlichen Klassencode von *Form1*:

Ein neues Kartenspiel wird erzeugt:

```
Dim spiel As New CSpiel()
```

Eine Hilfsroutine zum Löschen der Anzeige:

```
Private Sub loeschen()
    ListBox1.Items.Clear(): ListBox2.Items.Clear()
    ListBox3.Items.Clear(): ListBox4.Items.Clear()
End Sub

```

Alle Karten mischen:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    loeschen()
    spiel.mischen()
End Sub

```

Alle Karten austeilen:

```
Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
    loeschen()

```

Die Karten für Müller:

```
For i As Integer = 1 To 10
    ListBox1.Items.Add(spiel.Karten(i).Farbe & " " & spiel.Karten(i).Wert)
Next

```

Die Karten für Meier:

```
For i As Integer = 11 To 20
    ListBox2.Items.Add(spiel.Karten(i).Farbe & " " & spiel.Karten(i).Wert)
Next

```

Die Karten für Schulze:

```
For i As Integer = 21 To 30
    ListBox3.Items.Add(spiel.Karten(i).Farbe & " " & spiel.Karten(i).Wert)
Next

```

Die restlichen zwei Karten wandern in den Skat:

```
For i As Integer = 31 To 32
    ListBox4.Items.Add(spiel.Karten(i).Farbe & " " & spiel.Karten(i).Wert)
Next

```

```

End Sub
...
End Class ' Form1

```

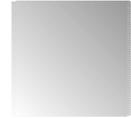
Test

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 noch 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:





Index

.NET WinRT-Profil 924
.NET-Framework 66

A

Abbruchbedingung 624
Abort 519
Abs 308
Abstrakte Klassen 197
Abstrakte Methoden 198
Access Control Entries 444
Accessor 202
ACE 444
ACL 444
Activator 836
Add 680
AddAccessRule 444
AddAfterSelf 680
AddBeforeSelf 680
AddDays 304, 311
AddExtension 481
AddFirst 680
AddHandler 147
AddHours 304, 311
AddMinutes 304, 311
AddMonths 304, 311
AddRange 351
AddressList 876
AddressOf 147
AddressWidth 883
AddToSchedule 1164
AddYears 304, 311
Administrator 874
ADO.NET-Klassen 731
ADO.NET-Objektmodell 728
Aggregation 224
Aktionsabfrage 755
Aktivierung 1049
Alias 343
Alternator 291
Ancestors 677
And 116
Anfangswerte 95
Anonyme Methoden 362
Anonyme Typen 388
Anwendungsdienste 945
Anwendungseinstellungen 762
App.config 632, 762
App.xaml 1040
AppBar 1033, 1133
AppData 1059
Append 321, 465
AppendChild 686, 691, 692
AppendLinesAsync 1091

AppendText 467
AppendTextAsync 1091
Application 877
ApplicationDataContainer 1064
Arithmetische Operatoren 112
Array 145, 269, 331, 420, 425
ArrayList 350, 356
As 357
Assemblierung 57, 70
Assemblies 883
Assembly 70, 246
 dynamisch laden 836
 GetExecutingAssembly 833
 Laden 833
 LoadFrom 833
AssemblyInfo.vb 903, 1038
Assemblyinformation 1039
Assets 1047
Async 552
Asynchrone Programmierentwurfsmuster 542
Atn 308
Attribute 72, 653, 656, 684, 1162
Attributes 441, 1082
Auflistung 347
Ausgabefenster 620
Ausnahmen 642
Ausschneiden 829
Aussteuerungsbalken 889
Auswahlabfrage 757
Auto-Play 1069
Auto-Property 169
AutoPlay 1010
AutoResetEvent 611, 613
AvailablePhysicalMemory 883
AvailableVirtualMemory 883
Await 552

B

BackgroundWorker 539
Barrier 608
BatteryChargeStatus 882
BatteryFullLifetime 882
BatteryLifePercent 882
BatteryLifeRemaining 882
Beep 909
Befehlsdesign 957
Befehlsfenster 618
Befehlsleisten 957
BeginInvoke 535, 548
BeginUpdate 456
Begrüßungsbildschirm 1043
BEISPIEL 12.4: Prolog 652
BEISPIEL 12.5: DTD 652
Benannte Parameter 126
Benutzerschnittstelle 944
Bezeichner 88
BigInteger 371
Bild einlesen 851
Bilder drucken 853
Bildschirme 879
Binärdatei 470
BinaryFormatter 258, 464, 473
BinaryReader 464, 470, 471
BinarySearch 278
BinaryWriter 464, 470, 471
BindingNavigator 483
BindingSource 258, 483
BlockingCollection 608
Boolean 92, 103
BootMode 878
Border 986
BottomAppBar 1133
Boxing 110
Breakpoints 622
Button 976

ByRef 127
Byte 92, 100
Byte-Array 333, 718
ByVal 127

C

C# 141
Callback 544
Callback-Timer 559, 563
Caller Information 634
CallerFilePath 634
CallerLineNumber 634
CallerMemberName 634
CameraDeviceType 855
CancellationPending 540
CancellationToken 599, 610
CancellationTokenSource 600
Canvas 987
CaptureElement 1008
Case 117
Catch 144, 327
ChangeClipboardChain 843
ChangeDatabase 737
Char 92, 101
Char-Array 283
CharacterFormat.BackgroundColor 981
CharacterFormat.Size 981
CharmBar, jetzt per Win+H erreichbar) 1109
Chars 281
CheckAccess 536
CheckBox 978
CheckFileExists 450, 480
CheckPathExists 450
ChildNodes 689
Class 146, 154, 159
ClassesRoot 831
ClassLoader 69
Clear 278, 349
ClickOnce-Deployment 800
Clipboard 827
Clone 277, 287, 768
Close 736
ClosedByUser 1050
Cloud 1066
CLR 67, 69
CLR-Threadpool 582
CLR-Version 885
CLS 67
Code Contracts 646
Code Document Object Model 244
Code Manager 69
CodeDOM 244
CodeDomProviders 244
Codefenster 64
Collection 347, 355, 426
Collection-Initialisierer 271
CollectionBase 699
CollectionViewSource 1022, 1030
ColumnName 771
Columns 769, 771
ColumnSpan 990
COM-Komponenten 71
COM-Marshaller 70
ComboBox 998
Command 738, 750
CommandBuilder 745
CommandLine 884
CommandText 739
CommandTimeout 739
CommandType 740
Common Language Runtime 64, 69
Common Language Specification 67
Common Type System 67, 68
CompanyName 884
CompareDocumentOrder 677
CompiledAssembly 247
Complex 373

- CompositeTransform 983
 - ComputerInfo 878
 - ComputerName 881
 - ConcurrentBag 608
 - ConcurrentDictionary 608
 - ConcurrentQueue 608
 - ConcurrentStack 608
 - Connection 732, 739
 - ConnectionString 734, 762
 - ConnectionStringBuilder 737
 - ConnectionTimeout 736
 - Console
 - BackgroundColor 915
 - Clear 916
 - Title 916
 - Console.In 913
 - Console.Out 913
 - ConsoleKeyInfo 912
 - Const 97
 - Constraint 357
 - ContactPicker 1114
 - Contains 349
 - Contract 922, 1038
 - Convert 108, 137
 - ConvertStringToByteArray 496
 - ConvertXMLToDataSet 719
 - Copy 440, 767
 - CopyAsync 1094
 - Copyright 884
 - CopyTo 277, 287, 349, 441
 - Cos 308
 - Count 403
 - CountdownEvent 608
 - Create 466
 - CreateCommand 737, 739
 - CreateDirectory 437
 - CreateElement 687
 - CreateEncryptor 475
 - CreateFileAsync 1089
 - CreateFolderQueryWithOptions 1084
 - CreateFromFile 478
 - CreateGraphics 892
 - CreateInstance 247
 - CreateNavigator 702, 723
 - CreateNew 478
 - CreateSubdirectory 437
 - CreateSubKey 832
 - CreateText 466
 - CreateToastNotifier 1161
 - CreateViewAccessor 479
 - CreationCollisionOption 1061, 1086
 - CreationTime 441
 - Credentials 1068
 - CRUD 780
 - CryptographicException 497
 - CryptoStream 475
 - CryptoStreams 496
 - CSV-Datei 506, 509
 - CTS 67
 - CType 107
 - CultureInvariant 292
 - Currency 92
 - CurrentClockSpeed 883
 - CurrentConfig 831
 - CurrentCulture 878
 - CurrentDirectory 884
 - CurrentStateChanged 1010
 - CurrentUser 831
 - CursorLeft 909
 - CursorTop 909
 - CursorVisible 909
 - Custom Events 175
- D**
- Data Encryption Standard 496
 - DataAdapter 749
 - Database 735

- DataChanged 1067
- DataColumn 769
- DataContext 794
- DataGrid 718
- DataGridView 258
- DataPackage 1103, 1109
- DataPackageView 1103
- DataReader 746
- DataRequested 1109
- DataSet 718
- DataSource 735
- DataTable 769
- DataTemplate 1006
- DateTime 303
- DataTransfer 1103
- DataTransferManager 1109, 1110
- DataGridView 773, 781
- Date 92, 102
- DateCreated 1082
- Datei komprimieren 499
- Datei verschlüsseln 496
- Dateiattribute 441
- Dateien umbenennen 441
- Dateiname 511
- Dateiparameter 465
- Dateipicker 1095
- Dateitypzuordnung 1075
- Dateiverknüpfung 1075
- Dateiverknüpfungen 840
- Datenkonsument 727
- Datenprovider 727, 728
- Datenquelle 776, 795
- Datentypen 92, 100, 143
- Datentypzuordnung 1070
- Datenzugriff 122
- DateTime 92, 303
- Datumsformatierung 313
- Day 303, 310
- DayOfWeek 303, 310
- DayOfYear 303, 310
- DaysInMonth 305, 312
- DbProviderFactories 730
- Deadlocks 515
- Debug 627
 - Write 628
 - WriteIf 628
 - WriteLineIf 628
- Debuggen 1054
- Debugger 617
- DebugView 1056
- Decimal 101
- Declare 146
- Decrypt 474
- DefaultExt 480
- DefaultFileExtension 1097
- Deklarationen 1045
- Delegate 359, 379
- Delegate instanziiieren 361
- Delete 437, 772
- DeleteAsync 1086, 1094
- DeleteCommand 750
- DeleteContainer 1068
- DeleteSubKey 832
- DeleteSubKeyTree 832
- DeleteValue 832
- Delta 984
- Deployment 801
- DereferenceLinks 450
- DES 496
- Descendants 677
- Description 884
- DESCryptoServiceProvider 475, 496
- Designer 63
- DesktopDirectory 451
- Destruktor 178, 181
- DeviceManager 848
- DialogResult 481
- Dictionary 356

- DictionaryEntry 881
- Dim 94
- Direction 745
- Directory 432, 436, 437
- DirectoryInfo 432, 437
- DirectoryName 436
- DirectorySecurity 444
- DisplayName 1082
- DisplayType 1082
- Dispose 183, 743
- Distinct 418
- Do 119
- Do Until...Loop 134
- Do While 119, 144
- Do While...Loop 134
- Do...Loop 119
- Do...Loop Until 134
- Do...Loop While 134
- DOCTYPE 656
- Document Object Model 684
- Document Type 684
- Document Type Definition 652
- DocumentsLibrary 1063
- DOM 684
- Double 92, 101
- DownloadsFolder 1062
- DragItemsStarting 1003
- DriveInfo 432
- DynamicObject 367
- dynamische Eigenschaft 382
- dynamische Programmierung 366
- dynamisches Objekt 382
- DynData 831
- E**
- Eigenschaften 163
- Eigenschaften-Fenster 63
- Eigenschaftsmethoden 230
- Einfügen 829
- Einzelschritt-Modus 625
- Element 653, 656, 684
- Elements 677
- Else 117
- ElseIf 116, 117, 144
- EnableRaisingEvents 447
- Encoding 333
- Encrypt 474
- EndInvoke 535, 548
- EndsWith 281
- Energiestatus 882
- Enter 527
- Entwicklungsumgebung 61
- Enum 145
- Enumerable 420
- Enumerationen 145
- Environment 878, 885, 907
- Environment Variablen 881
- Equals 419
- Erase 276
- Ereignis 147, 172
- Ereignisse 59
- Erweiterungsmethoden 390, 393, 408
- Escape-Zeichen 293
- Event 147, 172
- EventLog 633
- EventLogTraceListener 633
- Events 59
- Exception 325, 643
- ExceptionHandler 69
- ExecutablePath 884
- ExecuteNonQuery 738, 741
- ExecuteReader 738, 741, 747
- ExecuteScalar 738, 742
- Exists 441
- Exit 527
- Exit Do 134
- Exp 308

ExpandoObject 368
Exponentialfunktion 309
ExtClock 883
Extension 436, 1038
Extension Method-Syntax 392, 408

F

FailIfExists 1062
Families 879
Fast and Fluid 921
Fehler 325
Fehlerbehandlung 635
Fehlerklassen 636, 644
Fehlerstatus 905
Fehlersuche 85
FieldCount 748
File 432, 464
File Type Association 1075
FileAccess 465
FileAttribute 442
FileDropList 827
FileExtension 852
FileInfo 432, 464, 467
FileIO 1090, 1092
FileMode 465
FileName 450, 481
FileOpenPicker 1095
FileSavePicker 1095, 1097
FileSecurity 444
FileShare 466
FileStream 464
FileSystemAccessRule 444
FileSystemWatcher 432, 447
FileTypeChoices 1097
Fill 751
Filter 447, 480, 481
FilterIndex 450
Filtern 773
Filters 450
Finalize 182
Find 772
Fingereingabe 960
FirstChild 688, 711, 722
FlipView 1006
Flyout 1127
FolderBrowserDialog 451
FolderPicker 1095, 1098
FolderRelativeId 1082
FontFamily 879
Fonts 451
FontSmoothingContrast 879
FontSmoothingType 879
For 119, 144
For Each 145, 271, 356
for-each 710
For...Next 119, 133
Form1.vb 63
Format 314
Formelparser 331
Formular 58
Frame 955, 1011
Freigabeziel 1112, 1154
Friend 153
From 396
FromCurrentSynchronizationContext 604, 615
FullName 436
Function 124, 145
Funktionen 124, 145, 1044
Funktionsdiagramm 249
FutureAccessList 1099

G

Garbage Collector 181
GenerateUniqueName 1061, 1086
Generics 352, 354
generische Schnittstelle 419

Geräteeigenschaften 850
Gestensteuerung 983
Get 163, 202
GetBitmapAsync 1103
GetChanges 768
GetCommandLineArgs 906
GetCreationTime 441
GetCurrent 875
GetCurrentDirectory 438
GetDataAsync 1103
GetDataObject 827
GetDataPresent 828
GetDefaultView 1030
GetDirectories 438
GetDrives 439
GetDynamicMemberNames 367
GetElementsByTagName 693
GetEnumerator 349
GetEnvironmentVariables 881, 907
GetExecutingAssembly 884, 1040
GetFactoryClasses 730
GetFields 834
GetFiles 443
GetFoldersAsync 1084
GetHostEntry 876
GetHostName 875
GetLength 277, 287
GetMembers 834
GetMethods 835
GetProcessById 572
GetProperties 835
GetStorageItemsAsync 1103, 1157
GetSubKeyNames 832
GetTextAsync 1157
GetValue 748
GetValueNames 832
GetValues 748
Global 343
Glühbirne 85

GPS 944
Grafikbearbeitung 856
Grid 989
GridView 1004
GridView gruppieren 1020
GroupBy 400
Gruppen 871
GZipStream 499

H

Haltepunkte 623
Hardware-Informationen 882
Hashtable 224, 351
Hauptprogramm 904
HeaderTemplate 1022
Help 822
HelpMaker 825
HelpNamespace 824
HelpProvider 824
Hilfe-IDE 825
Hilfedatei 817, 821
Hilfemenü 822
HomeGroup 1063
Hour 303, 310
Hover 976
HTML 649
HTML 5 954
HTML Help Workshop 818
HtmlFormatHelper 1104
HyperlinkButton 976

I

IAsyncResult 544
ICollection 349
IComparable 253
IComparer 253
ICryptoTransform 497

- IDataObjekt 828
- IDisposable 184, 743
- Idle-Prozesse 571
- IEnumerable 348, 354, 420
- IEqualityComparer 419
- If 116, 144
- IgnorePatternWhitespace 292
- IgnoreWhitespace 701
- IInspectable 946
- IList 349
- ILSpy 925
- Image 983
- ImageFile 852, 856
- ImageProcess 856
- Implizite Zeilenfortsetzung 90
- ImportRow 770
- Imports 156, 341
- Indent 708
- IndentChars 708
- IndentLevel 629
- IndentSize 629
- Index 269
- Indexer 227, 230, 345
- IndexOf 278, 281, 317, 349
- Inherits 188
- InitialDirectory 481
- Initialisierer 420
- Initialisierung 157
- Initialize 277, 287
- InnerText 694
- InputScopeName 1018
- Insert 281, 349
- InsertCommand 750
- Installationsverzeichnis 1061
- InstallShield 808
- Instanzieren 156
- Int16 92
- Int32 92
- Int64 92
- Integer 92, 100
- Intellisense 160
- Interface 203
- InteropServices 895
- Interrupt 519
- InvalidOperationException 326
- Invoke 534, 546, 548, 836
- InvokeRequired 536
- IP-Adresse 875
- IsAbstract 834
- IsActive 984
- IsAdmin 874
- IsAfter 678
- IsAlive 520
- IsBackGround 520
- IsBefore 678
- IsClass 834
- IsClosed 748
- IsCOMObject 834
- IsCompleted 588
- IsEnum 834
- IsFixedSize 349
- IsFontSmoothingEnabled 879
- IsInterface 834
- IsLeapYear 305, 312
- IsLooping 1010
- IsMatch 289
- IsMuted 1010
- IsPublic 834
- IsReadOnly 349
- IsSealed 834
- IsSourceGrouped 1021
- IsSynchronized 349
- Item 349, 748
- ItemsControl 998
- ItemsPanelTemplate 1022
- Iterationsschleife 138
- Iterator 588
- iTextSharp 503

IUnknown 946

J

JavaScript 954

JIT-Compiler 65

Join 284, 331, 402, 519

K

Kapselung 152

Kartenspiel 264

Kartesische Koordinaten 236

Klasse 151

Klassendefinition 146

KnownFolders 1063

Kommandozeilenparameter 906

Kommentare 89, 142, 653

komplexe Zahlen 236

Komponenten 59

Komposition 224

Komprimieren 476

Konfigurationsdatei 762

Konsolenanwendung 53, 903

Konstanten 92, 97

Konstruktor 178, 571

Konstruktor überladen 230

Kontaktliste 1113

Kontextmenü 840

Kontravarianz 370

Konvertieren 106

Konvertierungsoperatoren 346

Kopieren 829

Kovarianz 370

kritische Abschnitte 559

L

Lambda Expression 379

Lambda-Ausdrücke 363, 393, 408

Language Projection 924, 946

LastAccessTime 441

LastWriteTime 441

Late Binding 370

Laufwerke 439

Launch-Contract 1049

Layout-Control 987

Lebenszyklus 1047

Length 277, 281, 286, 287

LINQ 387, 425, 426, 428, 509

Abfrage-Operatoren 394

Aggregat-Operatoren 402

AsEnumerable 405

Count 402

GroupBy 400

Gruppierungsoperator 400

Join 401

Konvertierungsmethoden 405

OrderBy 398

OrderByDescending 398

Projektionsoperatoren 396

Restriktionsoperator 398

Reverse 400

Select 396

SelectMany 396

Sortierungsoperatoren 398

Sum 403

ThenBy 398

ToArray 405

ToDictionary 405

ToList 405

ToLookup 405

Verzögerte Ausführung 404

Where 398

LINQ to Entities 780

LINQ to SQL 780

LINQ to XML-API 672

LINQ-Abfrageoperatoren 391

- LINQ-Provider 388
 - LINQ-Syntax 391
 - List 355
 - ListBox 332, 998
 - ListView 1002
 - Load 675
 - LoadCompleted 1012
 - LoadedAssemblies 884
 - LoadXml 685
 - LocalApplicationData 451
 - LocalFolder 1059
 - LocalMachine 831
 - LocalState 1059
 - Log 308
 - Log10 308
 - Logarithmus 309
 - LogicalDpi 967
 - Logische Operatoren 114
 - Lokal-Fenster 619
 - Long 92, 100
 - LongRunning 603
 - Lookahead 298
 - Lookbehind 299
 - Loop 144
 - Loop While 144
 - LowestBreakIteration 588
- M**
- MachineName 881
 - ManagementObjectSearcher 871
 - ManipulationDelta 983
 - ManipulationMode 983
 - ManualResetEvent 613
 - ManualResetEventSlim 608
 - Manufacturer 883
 - Map View 478
 - Mapperklassen 794
 - Match 289, 290
 - Matches 289, 291
 - Matrix 230
 - Matrizen 235
 - Max 308
 - MaximumRowsOrColumns 990
 - MCI 886, 895
 - MediaCapture 1008
 - MediaElement 1009
 - MediaEnded 1010
 - MediaOpened 1010
 - Mehrfachdeklaration 94
 - Memory Mapped File 477, 490
 - MemoryMappedFile 478
 - MemoryStream 718, 789
 - MenuStrip 877
 - MessageBox 129
 - MessageDialog 1035, 1115
 - Messwertliste 411
 - Metadaten 71, 925
 - Metasprache 649
 - Metazeichen 293
 - Method-Overriding 186
 - Methoden 59, 124, 145, 169
 - generische 358
 - überladen 136, 230
 - Methodenzeiger 359
 - MethodImpl 531
 - MethodInfo 246
 - Methods 59
 - Microsoft Intermediate Language Code 65
 - Mikrofon 888
 - Mikrofonpegel 889
 - Min 308, 490
 - Minute 303, 310
 - MMF 478
 - Mod 115
 - Module 57
 - Monitor 527
 - MonitorCount 880

MonitorsSameDisplayFormat 880
Month 303, 310
MostRecentlyUsedList 1101
Move 438, 440
MoveAndReplaceAsync 1094
MoveAsync 1094
MoveBufferArea 909
MoveTo 441
MoveToNext 702, 722
MoveToPrevious 702, 722
MoveToRoot 722
MRU-Liste 1102
MSIL-Code 65
Multiselect 450
Multitasking 514
Multithreading 73, 514, 558
MusicLibrary 1063
MustInherit 197
MustOverride 198
Mutex 530
My Project 63
MyBase 187
MyComputer 451
MyDocuments 451
MySettings 764

N

Namespace 70, 156, 343, 834
Narrowing 346
Navigate 961
NavigateToString 1011
Navigation 961
NavigationCacheMode 962
Navigationsdesign 959
NET-Reflection 833
Network 881
Netzwerk 881
New 146, 178

NewRow 770
Next 422
NextMatch 290
NextSibling 689, 711
Nodes 678
NodeType 706
Not 116
Nothing 158
Notification 944
NotifyFilter 447
NotInheritable 198
NotRunning 1050
Now 305, 312
NTFS 475
Nullable Type 111
Nullbedingter Operator 202
Nutzer 871

O

Object 92, 103
Objekt 151
Objekt-Initialisierer 157, 180
Objektbaum 483
Objektbäume 258
Objekte 147
Objektinitialisierer 209
Objektrelationales Mapping 780
OleDbConnection 732
OnActivated 1050
OnCachedFileUpdaterActivated 1050
OnFileActivated 1050, 1070, 1071
OnFileOpenPickerActivated 1051
OnFileSavePickerActivated 1051
OnLaunched 961, 1040, 1049
OnNavigatedTo 962, 1157
OnSeachActivated 1051
OnShareTargetActivated 1051, 1113, 1156
OnSuspending 1040

- OOOP 73, 149, 264
- OOWLib 864
- Open 466, 736
- OpenFileDialog 449, 480
- OpenOffice.org 863
- OpenOrCreate 466
- OpenReadAsync 1158
- OpenText 468
- Operatoren 89, 112, 143
- Operatorenüberladung 236, 344
- Option Explicit 95
- Option Strict 96
- Optional 128
- Optionale Parameter 128
- OrderBy 398
- ORM 780
- OSFullName 878
- OSVersion 878
- Overrides 187, 189
- OverwritePrompt 481

- P**

- Package.appxmanifest 1041
- PadLeft 281
- PadRight 281
- Page 955
- Paragraph 976
- Parallel LINQ 364, 608
- Parallel-Programmierung 579
- Parallel.For 585
- Parallel.ForEach 589
- Parallel.Invoke 583
- ParallelLoopResult 588
- Parameter 744
- ParameterName 744
- Parameterübergabe 126, 906, 961, 968
- ParentNode 711
- Parse 108, 305, 312, 676
- Parser 656
- Partielle Klassen 199
- PasswordBox 979
- PasswordVault 1068
- Path 432, 446
- Pattern 294
- Pattern Matching 288
- Pause 1010
- PC-Name 881
- PDF 502
- PDFsharp 504
- PeekChar 470
- Pegeldiagramm 891
- PerformanceData 832
- PI 308, 656
- PickMultipleContactsAsync 1114
- PickSaveFileAsync 1097
- PicturesLibrary 1063
- PlacementTarget 1012
- Platform 878
- Play 1010
- PLINQ 405, 608
- PointToScreen 823
- Polarkoordinaten 236
- Polling 543
- Polymorphes Verhalten 195
- Polymorphie 152, 184, 197
- PopUp 1127
- Popup-Benachrichtigungen 1120
- PopUp-Hilfe 823
- PopupMenu 1131
- portieren 141
- Position 1010
- Potenz 309
- Pow 308
- PowerLineStatus 882
- PowerStatus 882
- PreferFairness 603
- Preserve 273

Press 976
PreviousExecutionState 1050
PreviousSibling 711
PrimaryMonitorMaximizedWindowSize 880
PrimaryMonitorSize 880
Priority 520
Private 99, 153
Procedure-Step 622
Process 570, 575
Process Sandboxing 922
Process.Start 576
Processing Instruction 652, 656, 684
ProcessorCount 883
ProcessThread 570
ProductName 884
ProductVersion 884
Program.vb 903
Programm starten 575
Programmablaufplan 139
ProgressBar 889, 984
ProgressChanged 540
ProgressRing 984
Projektmappen-Explorer 62
Projekttyp 62
Properties 59
Property 146, 163, 206
Protected 153
Protected Friend 153
Protokolldatei 1054
Provider 732, 735
Prozedur 125
Prozeduren 145
Prozedurschritt 626
Prozesse 570
Public 100, 153
Public Property 168
Pulse 527, 528
PulseAll 527, 528

Q

Quantifizierer 289, 295
Query Expression-Syntax 392, 408
QueryOptions 1084
Queue 354, 356
QueueUserWorkItem 522

R

Racing 515
RadioButton 978
RaiseEvent 147, 173
Random 422
RandomAccessStreamReference 1105
Rank 277, 286, 287
Read 465
ReadAllBytes 470
ReadAllLines 468
ReadAllText 468
ReadBufferAsync 1091
ReadContentAsFloat 707
ReadKey 909, 912
ReadLine 57
ReadLines 468
ReadLinesAsync 1091
ReadOnly 166
ReadOnly-Eigenschaft 169
ReadTextAsync 1091
ReadToEnd 468
ReadWrite 465
ReadXml 696, 784, 789
Recordvariable 123
ReDim 273
Referenzieren 156
Referenztypen 280
Reflection-Mechanismus 245
Reflexion 71
Regedit.exe 830

Regex 289
RegexOptions 292
Registrierungsdatenbank 830
Registrierungseeditor 830
Registry 829, 831, 838, 878
RegistryKey 829, 831
Reguläre Ausdrücke 288
RegularExpressions 288
Relationen 718
Relaxed Delegates 362
Release 976
ReleaseMutex 530
Remote-Debugging 931
RemovableDevices 1063
Remove 281, 349, 681, 772
RemoveAccessRule 444
RemoveAll 681
RemoveAnnotations 681
RemoveAt 349
RemoveAttributes 681
RemoveContent 681
RenderTransform 983
Repeat 422
RepeatButton 976
Replace 281, 289
ReportStarted 1157
ResolutionScale 967
Resume 519
Resuming 1052
RetrievePropertiesAsync 1083
RichEditBox 979
RichTextBlock 973
RightToLeft 292
Roaming 1066
Roamingdaten 1068
RoamingFolder 1060
RoamingStorageQuota 1060
Rotation 984
Round 308

RowFilter 774
Rows 771
RowSpan 990
Rückrufmethode 543
Rücksprung 626
Run 976
Running 1047

S

Sandboxing 920
SaveFileDialog 449, 480
Scanner 859
Scanner-Assistent 854
ScannerDeviceType 855
Schaltjahr 305
ScheduledToastNotification 1163
Schleifen 144
Schleifenabbruch 587
Schleifenanweisungen 119, 133
Schlüsselwörter 88
Schnittstelle 203
Schriftarten 879, 960
ScreenOrientation 880
ScrollBar 984
ScrollView 988
Second 303, 310
Security Engine 69
Seitennavigation 968
Seitenstatus 962
SELECT 396, 702, 757
Select Case 117, 144
SelectCommand 750
SelectMany 397
SelectNodes 693
SelectSingleNode 689, 692, 693, 722
SemanticZoom 1025
Semaphore 532
SemaphoreSlim 608

- SendMessage 843
- Sequenzielle Datei 471
- Serialisieren 472
- Serialisierung 72
- Serializable 258, 472
- ServerVersion 735
- ServicePack 878
- Set 163, 202
- SetAccessControl 445
- SetAttributeValue 680
- SetBitmap 1103, 1105
- SetBufferSize 909
- SetClipboardViewer 843
- SetCurrentDirectory 438
- SetCursorPosition 909
- SetData 1103
- SetDataObject 827
- SetElementValue 680
- SetSource 1010
- SetStorageItems 1103, 1106
- SetText 981
- Settingsbereich 959
- Setup-Projekt 808
- SetWindowPosition 909
- SetWindowSize 909
- Shared 146, 167, 171
- Shared-Methoden 230
- ShareOperation 1156
- Short 100
- ShowAcquireImage 863
- ShowAcquisitionWizard 855
- ShowDialog 481
- ShowForSelectionAsync 1131
- ShowHelp 822
- ShowPopup 823
- ShowSelectDevice 860
- ShowShareUI 1111
- Sichtbarkeit 153
- Sign 308
- Sin 308
- Single 92, 101
- Single-Step 622
- SizeChanged 964
- Skip 679
- SkipWhile 679
- Sleep 519
- Slider 984
- SnapsTo 985
- SocketDesignation 883
- Sort 253, 278, 773
- SortedList 356
- SortedSet 374
- Sortieren 425, 773
- Sound 886, 895
- Soundkarte 884
- SpecialFolder 451
- Sperrmechanismen 523
- SpinLock 608
- SpinWait 608
- Splash Screen 1043, 1051, 1073
- Split 281, 284, 289, 331
- SQLite 1137
- Sqr 308
- Stack 354
- StackPanel 988
- Stammelement 654
- StandardDataFormats 1157
- Standardeigenschaft 225
- StartPreviewAsync 1009
- Startseite 961
- StartsWidth 281
- State 736
- statische Klasse 209
- Statische Methoden 171
- Statusmitteilung 1122
- StepValue 985
- Stop 1010
- Stopped 1047

- Storage-Interface 944
 - StorageApplicationPermissions 1099
 - StorageDeleteOption 1087
 - StorageFile 1089, 1093
 - StorageFileQueryResult 1152
 - StorageFolder 1081
 - Store 926
 - StoredProcedure 740
 - Streamdaten 913
 - StreamReader 464, 468, 1092
 - StreamWriter 464, 467, 1091
 - String 92, 101, 331, 333
 - Stringaddition 319
 - StringBuilder 285, 319
 - StringFormatConverter 1015
 - Stringinterpolation 316
 - StringReader 464
 - Stringvergleich 284
 - StringWriter 464
 - Structure 121, 145, 336
 - Strukturen 145
 - Strukturvariablen 335
 - Sub 125, 146
 - Sub Main 904
 - SubKeyCount 832
 - Subklassen 187
 - SubmitChanges 797
 - Substring 281, 317
 - Success 290
 - Suchen 774
 - Suchfunktionen 331
 - SuggestedStartLocation 1099
 - Sum 403
 - Suspend 518
 - Suspended 1050
 - Suspending 1047, 1051
 - SuspendingDeferral 1052
 - SVG 983
 - SyncLock 524
 - SyncRoot 349
 - System 92, 451
 - System.Console 908
 - System.Dynamic 367
 - System.Environment 877
 - System.IO.Compression 476
 - System.IO.FileStream 463
 - System.IO.Stream 463
 - System.Management 871
 - System.Net 875
 - System.Object 201
 - System.Reflection 833
 - System.Security.AccessControl 444
 - System.Security.Cryptography 475
 - System.Security.Principal 874
 - System.Text 334
 - System.Threading 517, 582
 - System.Threading.Tasks 582
 - System.XML 685
 - System.Xml.Linq 672
 - System.Xml.Serialization 701
 - SystemInformation 877
- T**
- TableAdapter 777
 - TableDirect 740
 - TableName 771
 - Tablet-Simulator 984
 - Take 679
 - TakeWhile 679
 - Tan 308
 - Task
 - Canceled 603
 - ContinueWith 596, 604
 - Created 603
 - Datenübergabe 592
 - Faulted 603
 - Fehlerbehandlung 602

- IsCanceled 603
- IsCompleted 603
- IsFaulted 603
- RanToCompletion 603
- Result 596
- Return 598
- Rückgabewerte 595
- Running 603
- Status 603
- Task-Ende 604
- Task-Id 602
- TaskCreationOptions 603
- Taskende 593
- Userinterface 604
- Verarbeitung abbrechen 598
- Wait 594
- WaitAll 595
- WaitingForActivation 603
- WaitingForChildrenToComplete 603
- WaitingToRun 603
- Weitere Eigenschaften 602
- Task Parallel Library 364
- Task starten 591
- Task.Factory.StartNew 590
- TaskCreationOptions 610
- TaskScheduler 604, 610
- Tastaturabfrage 912
- Tasteneingaben 1019
- TemporaryFolder 1061
- TemporaryKey.pfx 1046
- TerminalServerSession 881
- Terminated 1050
- TextBlock 973
- TextBox 979
- Textdatei 382, 466, 479
- TextWriterTraceListener 631
- Then 116, 144
- ThenBy 399
- ThenByDescending 399
- Thin Client 211
- Thread 517, 518, 558
 - initialisieren 558
 - synchronisieren 559
- Thread Service 70
- Thread(Of T) 556
- ThreadInterruptedException 519
- ThreadPool 521
- Threads 570
- threadsicher 533
- ThreadState 520
- ThreadWaitReason 573
- Throw 325, 326, 638, 643
- ThrowIfCancellationRequested 600
- Thumbnail-Ansicht 1096
- TickValues 985
- Tiles 920
- Timer 889
- Timer-Threads 538
- TimeSpan 319
- Title 450, 481, 884
- ToArray 417
- ToastNotificationFactory 1121
- ToastNotificationManager 1161
- ToastNotifications 1159
- ToCharArray 281
- Today 305, 312
- ToDouble 137
- ToggleButton 978
- ToggleSwitch 978
- ToLongDateString 304, 311
- ToLongTimeString 304, 311
- ToLower 281
- ToolTip 1012
- ToolTipPlacement 1012
- ToolTipService 1012
- TopAppBar 1133
- ToShortDateString 304, 311
- ToShortTimeString 304, 311

ToString 107, 313, 321
TotalPhysicalMemory 883
TotalVirtualMemory 883
Touchscreen 939
ToUpper 281
TPL 582
TPL Datenstrukturen 607
Trace 627, 630
TraceListener 631
Trademark 884
Transform 709
Transformationsdatei 709
TranslateX 984
TranslateY 984
TreeView 455, 714
Trefferanzahl 625
Trennzeichen 332
Trim 281
Truncate 466
Try 144
Try-Catch 325, 635
Try-Finally 640
TryBinaryOperation 367
TryCast 111
TryConvert 367
TryCreateInstance 367
TryEnter 527, 530
TryGetIndex 367
TryGetMember 367
TryInvokeMember 367
TrySetIndex 367
TrySetMember 367
Tuple 374
Type 834
Typecasting 379
TypeOf 205
Typinferenz 408
Typisierte DataSets 774
Typsicherheit 353

U

Überladen 128, 344
Überladene Methoden 170
Überwachungsfenster 620
Uhr anzeigen 305
UICommand 1131
Umgebungsvariablen 54, 907
UML 184
Unboxing 110
Unicode 101
UnicodeEncoding 496
Unified Modeling Language 184
UnIndent 629
UnspecifiedDeviceType 855
Unterverzeichnis 438
Update 746, 752
UpdateCommand 750
Updates 803
User-Name 881
UserDomainName 881
UserInteractive 881, 884
UserName 881
UserProfile 451
Users 832
Using 184, 469, 490, 491, 743
UTF-16 656
UTF-8 656

V

ValidateNames 481
Value 744
ValueCount 832
Variablen 92
Variablentypen 92
VariableSizedWrapGrid 990, 1023
VB-Compiler 53
VB-Source-Datei 55

- vlc.exe 53
- Verarbeitungsstatus 588
- Vererbung 152, 224, 229
- Vergleichsoperatoren 115
- Veröffentlichen 804, 805
- Verpacken 1046, 1145
- Verschlüsseln 473
- Versiegelte Klassen 198
- VersionString 878
- Vertrieb 1145
- Verweistypen 93
- Verzweigungen 144
- Video 895
- VideoDeviceType 855
- VideosLibrary 1063
- ViewManagement 964
- ViewMode 1096
- VirtualScreen 880
- Virtuelle Tastatur 1017
- Visual Studio 51
- Visual Studio Enterprise 52
- Visual Studio Professional 52
- VisualStateManager 965
- Vollbildmodus 926
- Volume 1010

- W**

- W3C 684
- Wait 527, 528
- WaitOne 530, 532, 612
- WAV 886
- Webcam 857, 944
- Webpublishing-Assistent 801, 807
- WebView 1011
- Wend 119
- Wertetypen 93
- Where 398, 679
- While 119
- WIA 845
- wiaaut.dll 846
- Widening 346
- Wiederholmuster 423
- Wiederverwendbarkeit 152
- Wildcards 294
- Win32_SoundDevice 884
- Window 955
- Windows 10 928
- Windows App 1037
- Windows Management Instrumentations 877
- Windows Runtime 919
- Windows Store 926
- Windows Store App 920
- Windows-Philosophie 57
- Windows-Simulator 928
- WindowsIdentity 872, 875
- Winkel 309
- WinMD 948
- WinMD-Dateien 924
- winmm.dll 896
- WinRT 919, 941
- WinRT-API 943
- WinRT-COM 924
- WinRT-Namespaces 945
- With 123
- WithEvents 147, 173
- WMI 877
- work stealing 583
- WorkerReportsProgress 540
- WorkingArea 880
- WorkingSet 884
- WPF 536
- Write 465
- WriteAllBytes 470
- WriteAttributeString 707
- WriteBufferAsync 1091
- WriteEndDocument 708
- WriteEndElement 707

WriteIf 627
WriteLine 57, 627
WriteOnly 166
Writer 863
WriteStartDocument 707
WriteTextAsync 1091
WriteXml 784, 789
WriteXmlSchema 696
Wurzel 309

X

XAML 954
XAttribute 673, 1163
XComment 673
XDeclaration 673
XDocument 672, 675
XDocumentType 673
XElement 673, 1163
XLINQ 667
XML 649
XML transformieren 681
XML-Schema 662
XML-Strings 718
XmlAttribute 685, 701
XmlCDATASection 685
XmlCharacterData 685
XmlComment 685
XmlDataDocument 696
XmlDocument 696, 702, 711, 723, 1163
XmlDocumentType 685
XmlElement 685, 701
XmlEntity 685
XmlAttribute 701
XmlAttribute 701
XmlImplementation 685
XmlAttributeNamedNodeMap 684
XmlAttributeNode 684, 688
XmlAttributeNodeList 684

XmlAttributeError 685
XmlAttributeProcessingInstruction 685
XmlAttributeReader 705, 717
XmlAttributeRoot 701
XmlAttributeSerializer 699
XmlAttributeText 685
XmlAttributeWriter 707
XmlAttributeWriterSettings 708
XmlAttributeNode 673
XmlAttribute 116
XmlAttributeDocument 702
XmlAttributeNavigator 702, 722, 723
XmlAttributeNodeIterator 702
XmlAttributeProcessingInstruction 673
XmlAttributeSchema 660
xsd.exe 666
XmlAttributeCompiledTransform 709
XmlAttributeSLT 709

Y

Year 303, 310
Yield 349

Z

Zahlenformatierung 313
Zeichenmengen 294
Zeilenumbruch 90
Zeitmessung 319
Zielplattformen 927
ZoomedInView 1026
ZoomedOutView 1026
Zufallszahlen 422
Zugriffsberechtigung 444
Zugriffsmodifizierer 153
Zuweisungsoperatoren 113
Zwischenablage 827, 842, 1102